



Spring Boot

Olivier Capuozzo, Frédéric Varni

Version 1.2, 2021-01-12

Table des matières

Validation	1
Présentation	1
Annotations de validation	1
Validation des données de requêtes	2
Validation de niveau attributs de bean	3
Validation de niveau bean	4
Montrer les erreurs (logique de présentation)	9

Validation

Présentation

Le contrôle de validité des **données d'entrée** est une tâche incontournable en développement web. On ne saurait être trop prudent car on ne peut supposer les intentions à l'origine d'une requête HTTP.

Les données d'entrée concernées ici sont celles :

- dans l'url (exemple `product/23` ou `product?id=23`)
- dans le corps de la requête HTTP (comme c'est le cas avec `post`)

La validité de ces données peut être vérifiée à grand coup de `if..else` dans le corps des méthodes contrôleur. Cette façon risque fort d'introduire de la redondance de vérification et alourdira assurément le code.

Comme toujours, la réponse consiste à déléguer le travail de validation à des services dédiés, en particulier via l'API Java Bean Validation.

Annotations de validation

Bean Validation définit un métamodèle et une API pour la validation de JavaBean. Les métadonnées sont représentées par des **annotations** ([JSR-303 est la référence initiale](#))

Bean Validation 2.0 est définie par JSR 380 (juillet 2017) et prend en compte les nouvelles caractéristiques de Java 8 (version minimum requise de Java)

Hibernate propose une implémente de référence (hibernate.org/validator).

L'usage des annotations permet facilement de lier, à un même élément, une à plusieurs contraintes, appelées aussi règles de validation (*validation rules*).

Table 1. Table Annotations

Contrainte	Description
@Null @NotNull	L'élément annoté doit être null ou différent de null
@AssertTrue @AssertFalse	L'élément annoté doit être true ou false
@Min(value)	L'élément annoté doit être un nombre dont la valeur est supérieure ou égale au minimum spécifié.
@Max(value)	L'élément annoté doit être un nombre dont la valeur est inférieure ou égale au maximum spécifié. Voir aussi @DecimalMin , @DecimalMax

@Size(min=, max=)	L'élément annoté doit être un nombre dont la valeur est dans les bornes spécifiées (inclusives)
@Negative	L'élément annoté doit être un nombre strictement négatif (zéro est considéré comme une valeur invalide). Voir aussi @NegativeOrZero , @Positive , @PositiveOrZero , @Digits
@Future	L'élément annoté doit être un instant, date ou time, dans le futur. Voir aussi @Past , @PastOrPresent , @FutureOrPresent
@Pattern(regexp =)	L'élément annoté doit être validé par l'expression régulière
@NotEmpty	L'élément annoté doit être non vide. Type supporté : CharSequence , Collection , Map , array
@NotBlank	L'élément annoté doit être non null et doit contenir au moins un caractère 'non-blanc'. Type supporté : CharSequence
@Email	L'élément annoté (String) doit être une adresse email bien-formée.
@CreditCardNumber	(hibernate) L'élément annoté représente un numéro de carte de credit. Vérifie selon l'implémentation de l'algorithme Luhn (qui traite de la syntaxe, non de la validité !)

Voir une liste ici : [table-builtin-constraints](#)

Validation des données de requêtes

Les données sont celles passées dans l'url, comme par exemple la valeur **23** dans : *product?id=23* ou *product/23*.

```

@Validated
@Controller
@RequestMapping(value = "/person")
public class PersonController {

    @GetMapping("/validateMinNameParam")
    public String validateMinNameParam(
        @RequestParam("name") @Size(min=2) String name) {
        // ...
    }

    @GetMapping("/validateMinName/{name}")
    public String validateMinNamePathVariable(
        @PathVariable("name") @Size(min=2) String name) {
        // ...
    }

}

```

On prendra soin de déclarer l'annotation `@Validated` au niveau de la classe contrôleur.

Par défaut, une exception `ConstraintViolationException` sera déclenchée en cas de non respect de la contrainte sur le paramètre ou variable `name`.

Remarque : Par la suite `@Validated` devra être retiré pour supprimer l'effet `ConstraintViolationException`.

Validation de niveau attributs de bean

Lorsque les données sont transmises par `post`, ces données peuvent être affectées aux différents attributs d'un **bean** : la logique d'affectation est basée sur correspondance entre le nom de l'attribut (d'un setter) et la clé (la propriété *name*) de la valeur d'entrée.

Pour réaliser cette tâche, Spring utilise les techniques de *data binding* (JavaBeans Specifications) et des convertisseurs.

Dans cet exemple, les données d'entrée (`post`) sont mappées dans un objet `FormPerson` pour être ensuite passé en argument du contrôleur (méthode).

```

1 [...]
2
3 @PostMapping("/add")
4 public String submit(@ModelAttribute FormPerson person){
5     // ...
6     return "redirect:/person/list";
7 }
8
9 [...]

```

Nous pouvons alors demander la validation du bean en question. Par prudence nous utilisons ici un *Form Bean* (l'approche par des beans métier est contestée [1: [C'est peut être un peu tard... Certains considèrent cette pratique comme un défaut de conception, à moins de passer par des DTO...](#)]). Pour ce faire, nous pouvons associer, au niveau d'un attribut (*field*), des contraintes déclaratives ([JSR-380](#) d'août 2017). Exemple :

```
1 public class FormPerson {
2
3     @NotBlank
4     @Size(min=2, max=30)
5     private String name;
6
7     @NotNull
8     @Min(value=18, message="{person.age.minor}") ①
9     private Integer age;
10
11     @Size(min=1, max=3)
12     private List<@NotBlank(message="{person.address.notblank}") String> addresses; ②
```

① La valeur du message peut être une clé i18n (comme ici) ou une constante littérale

② Contrainte sur les éléments de la collection *addresses*

La validation des contraintes doit être demandée. Pour ce faire, deux déclarations sont attendues :

1. placement de l'annotation **@Valid** sur le bean injecté en argument de la méthode contrôleur
2. déclaration d'un paramètre (injecté) de type **BindingResult** qui portera les informations résultantes de la validation.

```
1 [...]
2
3 @PostMapping("/add")
4 public String submit(@Valid @ModelAttribute Person person, BindingResult result){
5     if (result.hasErrors()) {
6         return ... ;
7     }
8     // ...
9     return "redirect:/person/list";
10 }
11 [...]
```

À noter que l'action de validation peut être réalisée manuellement dans le corps de la méthode contrôleur (il faut dans ce cas ne pas utiliser `@Valid` ni `BindingResult`). Un exemple ici : <https://www.baeldung.com/javax-validation>

Validation de niveau bean

Les contraintes présentées jusqu'ici agissent au niveau de l'attribut. Pour des contraintes plus

globales (cas de contraintes interdépendantes sur des attributs) on définira une **contrainte métier** au niveau de l'objet (de sa classe).

Par leur nature très spécifique, ces contraintes doivent être programmées, leur classe doivent implémenter l'interface `org.springframework.validation.Validator`:

```
1 public class FormPersonValidator implements Validator {
2
3     /**
4      * This Validator validates just Person instances
5      */
6     public boolean supports(Class clazz) {
7         return FormPerson.class.equals(clazz); ①
8     }
9
10    public void validate(Object obj, Errors e) { ②
11        ValidationUtils.rejectIfEmpty(e, "name", "name.empty"); ③
12
13        FormPerson p = (FormPerson) obj;
14        // senior must have 2 adresses or more.
15        if (p.getAge() > 80 && p.getAddresses().size() < 2) {
16            e.rejectValue("addresses", "more.one.addresse.senior.citizen", "constraint
senior addresses"); ④
17        }
18    }
19 }
```

- ① Détermine la portée du validateur
- ② Le premier argument est l'objet à valider, le deuxième représente le **BindingResult** (implémente `Errors`) qui sera passé au contrôleur
- ③ Une valeur 'empty' dans ce contexte signifie soit null soit chaîne vide "". L'objet disposant de cet attribut ('name') n'a pas besoin d'être passé à cette méthode car l'objet `Errors` détient déjà une référence interne à l'objet cible.
- ④ La méthode `rejectValue` nous permet de déclarer une erreur liée à un attribut ([rejectValue API](#))

rejectValue

```
void rejectValue(@Nullable
                 java.lang.String field,
                 java.lang.String errorCode,
                 java.lang.String defaultMessage)
```

Register a field error for the specified field of the current object (respecting the current nested path, if any), using the given error description.

The field name may be null or empty String to indicate the current object itself rather than a field of it. This may result in a corresponding field error within the nested object graph or a global error if the current object is the top object.

Parameters:

field - the field name (may be null or empty String)

errorCode - error code, interpretable as a message key

defaultMessage - fallback default message

Activation dans un contrôleur

Il existe plusieurs façons de faire, mais toutes sont dans un esprit de délégation.

- par injection du validateur
- via une annotation

Par injection explicite

On déclare le validateur en tant que composant via l'annotation `@Component`

```
import org.springframework.validation.Validator;

@Component
public class FormPersonValidator implements Validator { ... }
```

Puis un injecteur (sur l'attribut) dans la classe contrôleur :

```
1 @Controller
2 @RequestMapping(value = "/person")
3 public class PersonController {
4
5     @Autowired
6     FormPersonValidator personValidator;
7     ...
}
```

Que l'on utilise dans une méthode de ce contrôleur :


```

1 @PostMapping("/add")
2 public String submit(@ModelAttribute FormPerson personDto, BindingResult result){
①
3
4     personValidator.validate(personDto, result); ②
5
6     if (result.hasErrors()) { ③
7         return "person/form";
8     }
9
10    Person person = converToEntity(personDto); ④
11
12    personneDao.save(person);
13    return "redirect:/person/list";
14 }

```

- ① Le paramètre `BindingResult` sert à stocker les différentes erreurs détectées par les validateurs.
- ② Lance la logique de validation.
- ③ En façon simple de détecter la présence d'erreurs. Dans ce cas, on retourne à la vue du formulaire qui se chargera de présenter les erreurs à l'utilisateur (chapitre suivant)
- ④ Conversion d'un DTO vers un objet métier. à ce sujet voir [entity-to-and-from-dto-for-a-java-spring-application](#)

Par annotation

Cette autre façon, assez élégante, consiste à définir une **annotation personnalisée** et s'appuie non plus sur l'API de validation de spring mais sur l'API `javax.validation`, de concert avec la logique de l'annotation `@Valid`.

Pour commencer on définira 2 composants :

- Le validateur (`PersonConstraintValidator`)
- L'annotation associée (`PersonValidated`)

Le validateur :

```

1 import javax.validation.ConstraintValidator;
2 import javax.validation.ConstraintValidatorContext;
3
4
5 public class PersonConstraintValidator
6     implements ConstraintValidator<PersonValidated, FormPerson> { ①
7
8     public void initialize(PersonValidated constraint) { } ②
9
10    public boolean isValid(FormPerson p, ConstraintValidatorContext context) { ③
11
12        // senior must have 2 adresses or more.
13        if (p.getAge() > 80 && p.getAddresses().size() < 2) {
14            return false;
15        }
16
17        return true;
18    }
19 }

```

- ① L'interface est paramétrée par l'interface décrivant l'annotation personnalisée (que nous définissons juste après) et par le type de l'objet à valider.
- ② Pour recevoir les valeurs éventuelles d'attributs associés à l'annotation (aucun ici)
- ③ Cette méthode est le centre d'appel des règles métier.

Nous allons maintenant créer l'annotation que nous appellerons `PersonValidated` - (voir [Annotation API](#)):

```

1 package fr.laerce.cinema.annotation;
2
3 import fr.laerce.cinema.validator.PersonValidator;
4
5 import javax.validation.Constraint;
6 import javax.validation.Payload;
7 import java.lang.annotation.*;
8
9 @Target({ElementType.TYPE}) ①
10 @Retention(RetentionPolicy.RUNTIME) ②
11 @Constraint(validatedBy = PersonConstraintValidator.class) ③
12 @Documented ④
13 public @interface PersonValidated { ⑤
14     String message() default "{person.senioraddresses.invalid}"; ⑥
15     Class<?>[] groups() default {};
16     Class<? extends Payload>[] payload() default {};
17 }

```

- ① La cible de la validation, ici c'est la classe qui est ciblée, sinon ce peut être un attribut (**FIELD**), un constructeur (**CONSTRUCTOR**) ... (classes, méthodes, attributs, paramètres, variables locales et

packages : `ElementType`).

- ② La durée de vie (ici la plus longue)
- ③ Renseigne la classe d'implémentation de la validation
- ④ L'annotation fera partie de la documentation Javadoc
- ⑤ Le nom de l'interface, qui sera le **nom de l'annotation**
- ⑥ Le message (ou la clé du message) utilisé en cas d'invalidation

Nous pouvons maintenant utiliser l'annotation sur la classe métier :

```
1 @PersonValidated ①
2 public class FormPerson {
3     ...
4 }
```

- ① Placement de notre annotation `PersonValidated` (de portée globale à la classe `FormPerson`), qui s'ajoute aux autres.

L'activation des opérations de validation se déclare, là où on en a besoin, par la présence de l'annotation `@Valid`, par exemple ici dans une méthode contrôleur.

```
1 @PostMapping("/add")
2 public String submit(@Valid @ModelAttribute FormPerson personDto, BindingResult
result){ ① ②
3
4     if (result.hasErrors()) { ③
5         return "person/form";
6     }
7
8     Person person = convertToEntity(personDto);
9
10    personneDao.save(person);
11    return "redirect:/person/list";
12 }
```

- ① On remarquera l'annotation `@Valid` placée avant et sur le paramètre `FormPerson`
- ② Suivi de `BindingResult` qui détiendra le résultat des opérations de validations (l'absence de ce paramètre entraînera une erreur)
- ③ Même logique que précédemment

Montrer les erreurs (logique de présentation)

Nous nous plaçons dans le cas d'une logique de présentation déportée à *thymeleaf*.

Logiques de présentation des erreurs

On se place dans le cas où un utilisateur soumet un formulaire. Côté serveur, la validation peut révéler des erreurs d'entrée, suite par exemple à un non respect des consignes communiquées à l'utilisateur.

Le scénario classique consiste alors à retourner le formulaire à l'utilisateur en lui soulignant les points posant problème.

- Agir sur la classe CSS

L'idée consiste, en cas d'erreurs, à ajouter une classe CSS aux classes de l'élément :

```
<input type="text" th:field="*{age}" class="small" th:errorclass="fieldError" />
```

Si l'age (attribut de l'objet liée au formulaire) n'est pas validé (par exemple on attend un age ≥ 18 alors que l'utilisateur a rentré 12), alors la sortie sera de cette forme :

```
<input type="text" id="age" name="age" value="12" class="small fieldError" />
```

- Agir l'ajout d'un message d'erreur

L'idée consiste à ajouter un élément HTML près de celui qui pose problème. Dans cet exemple on ajoute, sous condition d'erreurs, un élément paragraphe `p`.

```
<input type="text" th:field="*{age}" />
<p th:if="${#fields.hasErrors('age')}"
    class="fielderr" th:errors="*{age}">Incorrect date</p>
```

On remarquera l'usage de :

- `th:if` qui conditionne l'insertion du paragraphe
- `th:errors` (au pluriel) afin d'obtenir **tous** les messages d'erreurs liés à l'attribut `age`.

On se fera à [th : validation-and-error-messages](#)

à titre d'exemple :

```

1 package fr.laerce.cinema.annotation;
2
3 import fr.laerce.cinema.validator.PersonValidator;
4
5 import javax.validation.Constraint;
6 import javax.validation.Payload;
7 import java.lang.annotation.*;
8
9 @Target({ ElementType.TYPE})
10 @Retention(RetentionPolicy.RUNTIME)
11 @Constraint(validatedBy = PersonConstraintValidator.class)
12 @Documented
13 public @interface PersonValidated {
14     String message() default "{person.bean.invalid}";
15     Class<?>[] groups() default {};
16     Class<? extends Payload>[] payload() default {};
17 }
18
19 ...
20
21 public class PersonConstraintValidator
22     implements ConstraintValidator<PersonValidated, FormPerson> {
23
24     public void initialize(PersonValidated constraint) {
25
26     }
27
28     public boolean isValid(FormPerson p, ConstraintValidatorContext context) {
29
30         if (p.getSurname() == null || p.getSurname().isEmpty())
31             return false;
32
33         if (p.getBirthYear() == null || p.getBirthYear() < 0) {
34             return false;
35         }
36
37         return true;
38     }
39 }

```

```

<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <th:block th:replace="fragments/header :: header-css"/>
</head>
<body>
<div th:replace="fragments/header :: menu(activeTab='actors')"/>

<main role="main">

```

```

<div class="jumbotron">
  <div class="container">
    <h5 class="display-5"
      th:text="{person.id} ?
      #{form.person.update} :
      #{form.person.add}">Person</h5>

    <div th:if="{#fields.hasErrors('{person.*'})}">
      <ul>
        <li th:each="e : #{#fields.detailedErrors('{person.*'})}"
          th:class="{e.global}? globalerr : fielderr">
          <span th:text="{e.global}? '*' : {e.fieldName}"> * or field</span> |
          <span th:text="{e.message}">The error message</span>
        </li>
      </ul>
    </div>

    <form action="#" th:action="@{/person/add}"
      th:object="{person}" enctype="multipart/form-data"
      method="post">

      <input type="hidden" th:field="*{id}"/>
      <input type="hidden" th:field="*{imagePath}"/>

      <div class="form-group">
        <label th:text="{form.givenname}">First name</label>
        <input type="text" class="form-control"
          aria-describedby="givennameHelp"
          th:attr="placeholder={form.givenname}"
          th:field="*{givenname}"/>
        <p th:if="{#fields.hasErrors('givenname')}"
          th:errors="*{givenname}" class="fielderr">Incorrect
          givenname</p>
      </div>

      <div class="form-group">

        <input type="text" class="form-control"
          aria-describedby="surnameHelp"
          th:attr="placeholder={form.surname}"
          th:field="*{surname}"/>
        <p th:if="{#fields.hasErrors('surname')}"
          th:errors="*{surname}" class="fielderr">Incorrect
          surname</p>
      </div>

      <div class="form-group">
        <label th:text="{form.birthyear}">Annee de naissance</label>
        <input type="text" class="form-control"
          aria-describedby="birthYearHelp"
          th:attr="placeholder={form.birthyear}"
          th:field="*{birthYear}"/>

```

```

        <p th:if="${#fields.hasErrors('birthYear')}}"
            th:errors="*{birthYear}" class="fielderr">Incorrect date</p>
    </div>

    <div class="form-group">
        <label th:text="#{form.person.picture}">file input</label>
        <p><input type="file" class="form-control-file" name="photo"/>
            <span th:text="*{imagePath}">Fichier photo</span>
        </p>
    </div>

    <div th:if="${person.id}" class="card"
        style="width: 15rem; margin-top: 2px; margin-bottom: 2px;">
        
        <div class="card-body">
            <p class="card-text"
                th:text="${person.givenname + ' ' + person.surname}"></p>
        </div>
    </div>
    <button type="submit" th:text="#{form.submit}"
        class="btn btn-primary">Submit</button>
</form>
</div>
</div>
</main>
<div th:replace="fragments/footer.html :: footer"/>
</body>
</html>

```

Et une interprétation du code résultant par le navigateur :

Ajouter Personne

- * | Données invalides
- surname | Nombre de caractères insuffisant

Prénom

aa

Nom

bb

Nombre de caractères insuffisant

Année de naissance

-2

Photo de la personne

Parcourir...

Aucun fichier sélectionné.

Valider

Remarque : le validateur est utilisé pour déterminer la validité de l'année de naissance à titre d'exemple (pourrait être délégué à une annotation dans le cas d'une logique simple de contrôle)