Java Bean validation

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

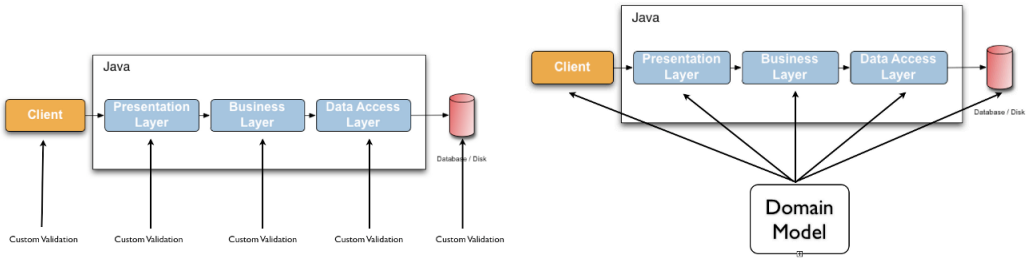
2026-11-01

Table des matières

1 **Qu'est-ce que Bean Validation ?** 1
2 **Pourquoi utiliser Bean Validation ?** 2
3 **Cas d'utilisation courants:** 2
4 Fonctionnement général de Bean Validation 3
5 Field or Property level constraints 3
6 Method or constructor level constraints 4
7 Validation 4
8 Contraintes pré définies 1/2 5
9 Contraintes Pré-définies (2/2) 5
10 Container element constraints 6
11 Validation de Graphes d'Objets 6
12 La Classe Validator 8
13 Messages d'erreur personnalisés 8
14 Contraintes Personnalisées : Adaptez la validation à vos besoins 9
15 Contrainte de classe 11
16 Contraintes groupées 14
17 Pour aller plus loin 16

1 Qu'est-ce que Bean Validation ?

- **Bean validation** est un framework qui permet de valider les données au sein des objets Java (JavaBeans).
- Il vous permet de définir des contraintes sur les propriétés de vos objets et s'assure que les données respectent ces contraintes avant d'être utilisées.
- **Bean Validation** est une API Java standard définie par le JSR 380.
- Il existe plusieurs implémentations de Bean Validation, dont **Hibernate Validator** est l'une des plus populaires.



2 Pourquoi utiliser Bean Validation ?

- **Améliorer la qualité des données:** En s'assurant que les données sont valides avant d'être utilisées, vous réduisez les erreurs et les exceptions.
- **Simplifier le code:** Les annotations de validation rendent le code plus concis et plus lisible.
- **Améliorer la sécurité:** En validant les entrées utilisateur, vous réduisez les risques d'injections de code et d'autres attaques.

3 Cas d'utilisation courants:

- **Validation des formulaires:** S'assurer que les données saisies par l'utilisateur sont conformes aux attentes.
- **Validation des données d'entrée d'API:** Vérifier la validité des données reçues par une API REST.
- **Validation des entités JPA:** S'assurer que les entités persistées respectent les contraintes de la base de données.

```
%%loadFromPOM
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>8.0.1.Final</version>
</dependency>
<dependency>
    <groupId>org.glassfish.expressly</groupId>
    <artifactId>expressly</artifactId>
    <version>5.0.0</version>
</dependency>
```

```
SLF4J(I): Connected with provider of type
[ch.qos.logback.classic.spi.LogbackServiceProvider]
```

4 Fonctionnement général de Bean Validation

Où sont définies les contraintes ?

- **Directement sur les beans:**
  - **Annotations:** Les contraintes sont définies sous forme d'annotations au niveau des attributs, méthodes ou classes.
  - **Exemple:** `@NotNull`, `@Size`, `@Email`
- **Dans des fichiers XML:**
  - Pour des configurations plus complexes ou une séparation des contraintes.

Le rôle du validateur

- **Vérification des contraintes:** Le validateur parcourt les annotations et vérifie si les valeurs des propriétés respectent les contraintes.
- **Génération des violations:** En cas de non-respect d'une contrainte, le validateur génère une violation.

Contraintes personnalisées

- **Création d'annotations personnalisées:** Pour définir des règles de validation spécifiques.
- **Implémentation d'un validateur associé:** Pour vérifier la contrainte personnalisée.

```
%maven org.projectlombok:lombok:1.18.34
```

5 Field or Property level constraints

```
%%compile fr/univtln/bruno/Person.java
package fr.univtln.bruno;

import lombok.AllArgsConstructor;
import jakarta.validation.constraints.*;

@AllArgsConstructor(staticName = "of")
public class Person {
    @NotNull(message = "Name cannot be null")
    private String name;

    @AssertTrue
    private boolean active;

    @Size(min = 18, max = 100,
          message = "Description must be between {min} and {max} characters")
```

```
    private String description;

    @Min(value = 18, message = "Age should not be less than {value}")
    @Max(value = 150, message = "Age should not be greater than {value}")
    private int age;

    @Email(message = "Email '${validatedValue}' is not valid")
    private String email;
}
```

6 Method or constructor level constraints

```
%%compile fr/univtln/bruno/X.java
package fr.univtln.bruno;
import jakarta.validation.constraints.*;
import jakarta.validation.executable.ValidateOnExecution;

@ValidateOnExecution(type=jakarta.validation.executable.ExecutableType.ALL)
public class X {
    private String value;
    public X(@NotNull @Size(min=6) String name) {this.value = value;}

    @Size(min=3)
    public String update(@NotNull @Size(min=1) String a,
                         @NotNull @Size(min=1) String b) { return value=a+b;}

}
```

7 Validation

```
//| echo: false
import java.util.Set;
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.ValidatorFactory;
```

```
ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
```

```
16:15:58.483 [IJava-executor-0] INFO   o.h.validator.internal.util.Version --
HV000001: Hibernate Validator 8.0.1.Final
```

```
//| output-location: slide
import fr.univtln.bruno.Person;
Person p = Person.of("Pierre",true,"Sympa",25,"p@");

Set<ConstraintViolation<Person>> constraintViolations = validator.validate(p);

constraintViolations.stream().forEach(System.out::println);
```

```
ConstraintViolationImpl{interpolatedMessage='Email 'p@' is not valid', propertyPath=email, rootBeanClass=class fr.univtln.bruno.Person, messageTemplate='Email '${validatedValue}' is not valid'}
ConstraintViolationImpl{interpolatedMessage='Description must be between 18 and 100 characters', propertyPath=description, rootBeanClass=class fr.univtln.bruno.Person, messageTemplate='Description must be between {min} and {max} characters'}
```

8 Contraintes pré définies 1/2

- `@NotNull` value is not null.
- `@AssertTrue` value is true.
- `@Size` size between the attributes min and max (String, Collection, Map, and Array).
- `@Min` validates no smaller than the value attribute.
- `@Max` validates no larger than the value attribute.
- `@Email` validates valid email address.
- `@NotEmpty` not null nor empty (String, Collection, Map or Array).
- `@NotBlank` ot null or whitespace (Text only).
- `@Positive` and `@PositiveOrZero` trictly positive, or positive including 0.
- `@Negative` and `@NegativeOrZero` apply to numeric values and validate that they are strictly negative, or negative including 0 (numeric only).
- `@Past` and `@PastOrPresent` date value is in the past or the past including the present.
- `@Future` and `@FutureOrPresent` date value is in the future, or in the future including the present.

9 Contraintes Pré-définies (2/2)

- **Paramètres spécifiques:** De nombreuses contraintes acceptent des paramètres pour affiner leur comportement.
  - Par exemple, `@Size(min=2, max=10)` pour spécifier une taille minimale et maximale.
- **Message personnalisé:** L'attribut `message` permet de définir un message d'erreur personnalisé en cas de violation.
  - **Interpolation:** On peut utiliser des expressions entre accolades `{}` pour insérer des valeurs dynamiques dans le message.

- Exemple : `@Size(min=2, message="{javax.validation.constraints.Size.message}")`
- **Incohérence entre contraintes:** Bean Validation ne vérifie pas l'incohérence logique entre différentes contraintes.
- **Types incompatibles:** Si une contrainte est appliquée à un type de donnée incompatible, une exception est levée.

10 Container element constraints

```
%%compile fr/univtln/bruno/Message.java
package fr.univtln.bruno;
import jakarta.validation.constraints.*;
import lombok.*;
import java.util.List;
import java.util.Optional;
import java.time.LocalDate;
@Data(staticConstructor="of")
public class Message {
  private final List<@NotBlank String> lines;
  private final Optional<@Past LocalDate> date;
}
```

```
//| echo: false
import fr.univtln.bruno.Message;
import java.time.LocalDate;
import java.util.Optional;
```

```
Message message = Message.of(List.of("Ok","","Again"),
Optional.of(LocalDate.now().plusDays(1)));

validator.validate(message).stream()
    .map(v->"%s %s".formatted(v.getPropertyPath(), v.getMessage()))
    .forEach(System.out::println);
```

```
lines[1].<list element> must not be blank
date must be a past date
```

11 Validation de Graphes d'Objets

- L'annotation `@Valid`
  - **Validation en cascade:** L'annotation `@Valid` permet de déclencher la validation de propriétés qui sont elles-mêmes des objets.

- **Graphe d'objets:** On peut ainsi valider des structures complexes d'objets imbriqués.

```
%%compile fr/univtln/bruno/Maitre.java
package fr.univtln.bruno;
import lombok.Data;
import jakarta.validation.constraints.*;

@Data(staticConstructor="of")
public class Maitre {
    @NotEmpty
    private final String name;
}
```

```
%%compile fr/univtln/bruno/Chien.java
package fr.univtln.bruno;
import lombok.Data;
import jakarta.validation.constraints.*;
import jakarta.validation.Valid;

@Data(staticConstructor="of")
public class Chien {

    @NotNull
    private final String name;

    @Valid
    private final Maitre maitre;
}
```

```
//| echo: false
import fr.univtln.bruno.*;
```

```
factory = Validation.buildDefaultValidatorFactory();
validator = factory.getValidator();
validator.validate(Chien.of("Rex",Maitre.of("")));
```

```
[ConstraintViolationImpl{interpolatedMessage='must not be empty',
propertyPath=maitre.name, rootBeanClass=class fr.univtln.bruno.Chien,
messageTemplate='{jakarta.validation.constraints.NotEmpty.message}'}]
```

12 La Classe Validator

- Le rôle du Validator
  - **Moteur de validation:** La classe `Validator` est responsable de l'exécution des règles de validation définies sur un objet.
  - **Portée de la validation:** La validation peut porter sur l'ensemble d'un objet ou sur une propriété spécifique.
- Invocation du Validator
- **Invocation manuelle:** Vous pouvez instancier un `Validator` et l'utiliser pour valider un objet à tout moment.
- **Invocation automatique:** De nombreux frameworks (JPA, JAX-RS, etc.) intègrent Bean Validation et appellent automatiquement le `Validator` dans des contextes spécifiques (par exemple, avant de persister une entité JPA).
- Exploration des contraintes
  - **Récupération des violations:** Le `Validator` retourne un ensemble de violations en cas d'échec de la validation.
  - **Analyse des erreurs:** Ces violations contiennent des informations détaillées sur les contraintes violées et les propriétés concernées.

```
validator.validateValue(Maitre.class, "name", "Pierre");
```

```
[]
```

```
validator.getConstraintsForClass(Chien.class);
```

```
BeanDescriptorImpl{class='Chien'}
```

13 Messages d'erreur personnalisés

- Messages par défaut et personnalisation
  - **Messages par défaut:** Chaque contrainte possède un message d'erreur par défaut.
  - **Personnalisation:** Vous pouvez redéfinir ce message en utilisant l'attribut `message`.
  - **Interpolation:** Les messages peuvent contenir des expressions interpolées (e.g., `{value}`) pour afficher la valeur de la propriété en erreur.

- Internationalisation
  - **Fichiers de propriétés:** Les messages d'erreur peuvent être définis dans des fichiers de propriétés (souvent dans le répertoire `resources`).
  - **Clés:** Les messages sont associés à des clés uniques pour faciliter la gestion et l'internationalisation.
  - **Interpolation:** Les expressions interpolées peuvent également être utilisées dans les messages définis dans les fichiers de propriétés.

14 Contraintes Personnalisées : Adaptez la validation à vos besoins

- **Flexibilité maximale:**
  - Validez des règles métier spécifiques.
  - Allez au-delà des contraintes standards.
- **Deux niveaux d'application:**
  - **Champ:** Pour une validation précise.
  - **Classe:** Pour une validation globale.
- **Création en 3 étapes simples:**
  1. **Annotation:** Décore l'élément à valider.
  2. **Validateur:** Implémente la logique de validation.
  3. **Message d'erreur:** Personnalisez les messages.

```
%%compile fr/univtln/bruno/CaseMode.java
package fr.univtln.bruno;
public enum CaseMode { UPPER, LOWER }
```

```
%%compile fr/univtln/bruno/CheckCase.java
package fr.univtln.bruno;
import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE, TYPE_USE })
@Retention(RUNTIME)
@Constraint(validatedBy = CheckCaseValidator.class)
@Documented
@Repeatable(CheckCase.List.class)
public @interface CheckCase {
                                    String         message()         default
```

```
"{fr.univtln.bruno.beanvalidation.CheckCase.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};

    CaseMode value();

    @Target({ FIELD, METHOD, PARAMETER, ANNOTATION_TYPE })
    @Retention(RUNTIME)
    @Documented
    @interface List {
        CheckCase[] value();
    }
}
```

```
/tmp/jupyterJava/fr/univtln/bruno/CheckCase.java:10: error: cannot find symbol
@Constraint(validatedBy = CheckCaseValidator.class)
                          ^
  symbol: class CheckCaseValidator
1 error
```

```
%%compile fr/univtln/bruno/CheckCaseValidator.java
package fr.univtln.bruno;
import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class CheckCaseValidator implements ConstraintValidator<CheckCase,
String> {

    private CaseMode caseMode;

    @Override
    public void initialize(CheckCase constraintAnnotation) {
        this.caseMode = constraintAnnotation.value();
    }

    @Override
         public boolean isValid(String object, ConstraintValidatorContext
constraintContext) {
        if ( object == null ) {
            return true;
        }
```

```
        if ( caseMode == CaseMode.UPPER ) {
            return object.equals( object.toUpperCase() );
        }
        else {
            return object.equals( object.toLowerCase() );
        }
    }
}
```

```
%%compile fr/univtln/bruno/TestCheckCase.java
package fr.univtln.bruno;
import lombok.*;
@Data(staticConstructor="of")
public class TestCheckCase {
    @CheckCase(CaseMode.LOWER)
    private final String x;

    @CheckCase(CaseMode.UPPER)
    private final String y;
}
```

```
validator.validate(TestCheckCase.of("A","b")).stream()
    .map(v→"%s %s"
         .formatted(v.getPropertyPath(), v.getMessage()))
    .forEach(System.out::println);
```

```
x {fr.univtln.bruno.beanvalidation.CheckCase.message}
y {fr.univtln.bruno.beanvalidation.CheckCase.message}
```

15 Contrainte de classe

Une contrainte de classe s'applique à **l'ensemble d'un objet** et vérifie qu'il respecte certaines règles globales.

- **Définition:** Une règle qui doit être vérifiée sur tous les attributs ou relations d'une classe.
- **Exemples:**
  - **Cohérence des données:** Vérifier que la date de naissance est antérieure à la date d'embauche.
  - **Règles métier:** S'assurer qu'un client a au moins une commande associée.

- **Intégrité référentielle:** Vérifier que les clés étrangères pointent vers des enregistrements existants.

```
%%compile fr/univtln/bruno/PersonDTO.java
package fr.univtln.bruno;
//Les personnes dont l'ID est entre 1 et 100 doivent avoir
//au moins un name ou un nickname
@PersonValidation
public record PersonDTO(int id, String name, String nickname) {};
```

```
/tmp/jupyterJava/fr/univtln/bruno/PersonDTO.java:4: error: cannot find symbol
@PersonValidation
 ^
  symbol: class PersonValidation
1 error
```

```
%%compile fr/univtln/bruno/PersonValidation.java
package fr.univtln.bruno;
import jakarta.validation.Constraint;
import jakarta.validation.Payload;
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target({ TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = PersonValidator.class)
@Documented
public @interface PersonValidation {
                                        String         message()         default
"{fr.univtln.bruno.beanvalidation.PersonDTO.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
    //no value
}
```

```
/tmp/jupyterJava/fr/univtln/bruno/PersonValidation.java:10: error: cannot find
symbol
@Constraint(validatedBy = PersonValidator.class)
                          ^
```

```
  symbol: class PersonValidator
1 error
```

```
%%compile fr/univtln/bruno/PersonValidator.java
package fr.univtln.bruno;
import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;

public class PersonValidator implements ConstraintValidator<PersonValidation,
PersonDTO> {

    @Override
    public void initialize(PersonValidation constraintAnnotation) {
    }

    @Override
        public boolean isValid(PersonDTO person, ConstraintValidatorContext
constraintContext) {
        if (person==null) return false;
        if ((person.id()>=1) || (person.id()<=100))
            if (person.name().isBlank() && person.nickname().isBlank()) return
false;
        return true;
    }
}
```

```
//| echo: false
import fr.univtln.bruno.*;
factory = Validation.buildDefaultValidatorFactory();
validator = factory.getValidator();
```

```
validator.validate(new PersonDTO(10,"", ""));
```

```
[ConstraintViolationImpl{interpolatedMessage='{fr.univtln.bruno.beanvalidation.PersonDTO.message}
propertyPath=,         rootBeanClass=class         fr.univtln.bruno.PersonDTO,
messageTemplate='{fr.univtln.bruno.beanvalidation.PersonDTO.message}'}]
```

16 Contraintes groupées

Des contraintes peuvent être regroupées pour former des **ensembles de règles** à appliquer simultanément.

- **Définition:** Un groupe de contraintes qui doivent toutes être satisfaites pour que la validation soit considérée comme réussie.
- **Exemples:**
  - **Validation de formulaire:** Un formulaire de contact peut avoir un groupe de contraintes pour vérifier la validité de l'adresse email, du nom et du message.
  - **Règles métier:** Une commande peut avoir un groupe de contraintes pour vérifier la disponibilité des produits, le montant minimum de la commande et la validité de l'adresse de livraison.

```
%%compile fr/univtln/bruno/SimpleUser.java
package fr.univtln.bruno;
import lombok.*;
import jakarta.validation.constraints.*;
import jakarta.validation.constraints.Pattern;
@Data(staticConstructor="of")
public class SimpleUser {
    @NotNull
    @Pattern(regexp = "[a-zA-Z0-9]+", message = "only digits and letters")
     @Size(min = 6, max = 10, message = "must have between {min} and {max}
characters")
    private final String login;

    @NotNull
    @Pattern(regexp = "[a-zA-Z0-9]+", message = "only digits and letters")
     @Size(min = 6, max = 10, message = "must have between {min} and {max}
characters")
    private final String nickname;
}
```

```
//| echo: false
import fr.univtln.bruno.SimpleUser;
```

```
SimpleUser user = SimpleUser.of("Dave","rick");

validator.validate(user).stream()
    .map(v→"%s %s"
```

```
            .formatted(v.getPropertyPath(), v.getMessage()))
        .forEach(System.out::println);
```

```
login must have between 6 and 10 characters
nickname must have between 6 and 10 characters
```

```
%%compile fr/univtln/bruno/ValidIdentifier.java
package fr.univtln.bruno;
import jakarta.validation.Constraint;
import jakarta.validation.ReportAsSingleViolation;
import jakarta.validation.constraints.*;
import java.lang.annotation.*;
import jakarta.validation.Payload;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
@NotNull
@Pattern(regexp = ".*\\d.*", message = "must contain at least one numeric
character")
@Size(min = 6, max = 32, message = "must have between 6 and 32 characters")
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = {})
@ReportAsSingleViolation
public @interface ValidIdentifier {
     String message() default "field should have a valid length and contain
numeric character(s).";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

```
%%compile fr/univtln/bruno/SimpleUserBis.java
package fr.univtln.bruno;
import lombok.*;
import jakarta.validation.constraints.*;
import jakarta.validation.constraints.Pattern;
@Data(staticConstructor="of")
public class SimpleUserBis {
    @ValidIdentifier
    private final String login;
```

```
    @ValidIdentifier
    private final String nickname;
}
```

```
SimpleUserBis userBis = SimpleUserBis.of("Dave","rick");

validator.validate(userBis).stream()
    .map(v->"%s %s"
         .formatted(v.getPropertyPath(), v.getMessage()))
    .forEach(System.out::println);
```

```
nickname field should have a valid length and contain numeric character(s).
login field should have a valid length and contain numeric character(s).
```

17 Pour aller plus loin

Lire le guide de référence.