# Validation avec Bean Validation (Validator)

## 1. Introduction



**Bean Validation** (JSR 380) permet de valider les données avec des annotations.

**Avantages :**

- Déclaratif (annotations sur les champs)
- Réutilisable (validation côté service, controller, persistence)
- Messages d'erreur personnalisables
- Validation groupée et conditionnelle

## 2. Configuration

### 2.1 Dépendance Maven

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```



Spring Boot inclut automatiquement **Hibernate Validator** (implémentation de référence).

### 2.2 Configuration des messages (optionnel)

Fichier `src/main/resources/ValidationMessages.properties` :

```
# Messages personnalisés
jakarta.validation.constraints.NotNull.message=Le champ {field} est obligatoire
jakarta.validation.constraints.Email.message=L'email {validatedValue} n'est pas
valide
jakarta.validation.constraints.Size.message=La taille doit être entre {min} et
{max}

# Messages custom
product.name.invalid=Le nom du produit doit contenir entre 3 et 100 caractères
user.password.weak=Le mot de passe doit contenir au moins 8 caractères
```

## 3. Annotations de validation courantes

### 3.1 Contraintes de base

```
import jakarta.validation.constraints.*;

public class User {
    @NotNull(message = "L'ID ne peut pas être null")
    private UUID id;
    @NotBlank(message = "Le nom d'utilisateur est obligatoire")
    @Size(min = 3, max = 50)
    private String username;
    @Email(message = "Email invalide")
    @NotBlank
    private String email;
    @Min(18)
    @Max(120)
    private Integer age;
    @Pattern(regexp = "^(?=.*[A-Z])(?=.*\\d).{8,}$",
             message = "Mot de passe trop faible")
    private String password;
    @DecimalMin(value = "0.0", inclusive = false)
    @DecimalMax("999999.99")
    private BigDecimal salary;
    @Past(message = "La date de naissance doit être dans le passé")
    private LocalDate birthDate;
    @Future
    private LocalDateTime appointmentDate;
    @AssertTrue(message = "Vous devez accepter les CGU")
    private Boolean termsAccepted;
}
```



**Différence importante :**

- `@NotNull` : Interdit `null` (accepte chaîne vide)
- `@NotEmpty` : Interdit `null` et collection/chaîne vide
- `@NotBlank` : Interdit `null`, vide et espaces uniquement (String uniquement)

### 3.2 Annotations avancées

```
public class Product {
    @NotNull
    @Valid  // ← Valide en cascade l'objet imbriqué
    private Category category;
    @Size(min = 1, max = 10)
    @Valid  // ← Valide chaque élément de la liste
    private List<@NotNull ProductImage> images;
    @URL(protocol = "https")
    private String officialWebsite;
    @CreditCardNumber
```

```
    private String cardNumber;
    @Positive
    private Integer stock;
    @PositiveOrZero
    private BigDecimal discount;
}
```

## 4. Validation dans les différentes couches

### 4.1 Controller (REST API)

```
@RestController
@RequestMapping("/api/users")
public class UserController {
    @PostMapping
    public ResponseEntity<User> createUser(
            @Valid @RequestBody User user,  // ← Validation automatique
            BindingResult result) {  // ← Contient les erreurs
        if (result.hasErrors()) {
            // Gestion manuelle des erreurs
            Map<String, String> errors = new HashMap<>();
            result.getFieldErrors().forEach(error ->
                errors.put(error.getField(), error.getDefaultMessage())
            );
            return ResponseEntity.badRequest().body(errors);
        }
        return ResponseEntity.ok(userService.save(user));
    }
    // Version avec gestion automatique des erreurs
    @PostMapping("/auto")
    public ResponseEntity<User> createUserAuto(
            @Valid @RequestBody User user) {  // ← Lève
MethodArgumentNotValidException si erreur
        return ResponseEntity.ok(userService.save(user));
    }
}
```

### 4.2 Service Layer

```
@Service
@Validated  // ← Active la validation sur les méthodes
public class ProductService {
    private final Validator validator;  // Injection du validateur
    public ProductService(Validator validator) {
        this.validator = validator;
    }
    // Validation automatique des paramètres
    public Product createProduct(@Valid Product product) {
        // Spring valide automatiquement avec @Validated sur la classe
```

```
        return productRepository.save(product);
    }
    // Validation manuelle
    public void validateProduct(Product product) {
        Set<ConstraintViolation<Product>> violations = validator.validate(product);
        if (!violations.isEmpty()) {
            String errors = violations.stream()
                .map(v -> v.getPropertyPath() + ": " + v.getMessage())
                .collect(Collectors.joining(", "));
            throw new ValidationException("Erreurs de validation: " + errors);
        }
    }
    // Validation de méthode
    public Product findByName(@NotBlank @Size(min = 3) String name) {
        return productRepository.findByName(name)
            .orElseThrow(() -> new NotFoundException("Product not found"));
    }
}
```

#### 4.3 Entity (JPA)

```
@Entity
public class Order {
    @Id
    @GeneratedValue
    private UUID id;
    @NotNull
    @ManyToOne(fetch = FetchType.LAZY)
    private User user;
    @NotEmpty(message = "Une commande doit contenir au moins un article")
    @Valid  // Valide chaque OrderItem
    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<OrderItem> items = new ArrayList<>();
    @DecimalMin("0.01")
    @Column(nullable = false)
    private BigDecimal totalAmount;
    // Validation appelée avant persist/update
    @PrePersist
    @PreUpdate
    private void validate() {
        calculateTotal();
        if (totalAmount.compareTo(BigDecimal.ZERO) <= 0) {
            throw new ValidationException("Le montant total doit être positif");
        }
    }
    private void calculateTotal() {
        totalAmount = items.stream()
            .map(OrderItem::getSubtotal)
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}
```

## 5. Validation personnalisée

### 5.1 Créer une annotation custom

```
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = StrongPasswordValidator.class)
@Documented
public @interface StrongPassword {
    String message() default "Le mot de passe doit contenir au moins 8 caractères,
" +
                             "une majuscule, un chiffre et un caractère spécial";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

### 5.2 Implémenter le validateur

```
public class StrongPasswordValidator implements ConstraintValidator<StrongPassword,
String> {
    private static final String PASSWORD_PATTERN =
        "^(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[@$!%*?&])[A-Za-z\\d@$!%*?&]{8,}$";
    @Override
    public boolean isValid(String password, ConstraintValidatorContext context) {
        if (password == null) {
            return false;
        }
        return password.matches(PASSWORD_PATTERN);
    }
}
```

### 5.3 Utilisation

```
public class UserRegistrationRequest {
    @NotBlank
    @Email
    private String email;
    @StrongPassword  // ← Notre annotation custom
    private String password;
}
```

## 6. Validation conditionnelle avec Groups

```
public interface OnCreate {}
public interface OnUpdate {}

public class Product {
    @Null(groups = OnCreate.class)  // Null lors de la création
    @NotNull(groups = OnUpdate.class)  // Obligatoire lors de la MAJ
    private UUID id;
    @NotBlank(groups = {OnCreate.class, OnUpdate.class})
    private String name;
    @NotNull(groups = OnCreate.class)
    @DecimalMin(value = "0.01", groups = {OnCreate.class, OnUpdate.class})
    private BigDecimal price;
}
```

Utilisation dans le controller :

```
@PostMapping
public Product create(@Validated(OnCreate.class) @RequestBody Product product) {
    return productService.save(product);
}

@PutMapping("/{id}")
public Product update(
        @PathVariable UUID id,
        @Validated(OnUpdate.class) @RequestBody Product product) {
    product.setId(id);
    return productService.update(product);
}
```

## 7. Gestion globale des erreurs

```
@RestControllerAdvice
public class ValidationExceptionHandler {
    @ExceptionHandler(MethodArgumentNotValidException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public Map<String, Object>
handleValidationErrors(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult().getFieldErrors().forEach(error ->
            errors.put(error.getField(), error.getDefaultMessage())
        );
        return Map.of(
            "timestamp", LocalDateTime.now(),
            "status", 400,
            "errors", errors
        );
    }
    @ExceptionHandler(ConstraintViolationException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public Map<String, Object>
handleConstraintViolation(ConstraintViolationException ex) {
```

```
        Map<String, String> errors = new HashMap<>();
        ex.getConstraintViolations().forEach(violation -> {
            String propertyPath = violation.getPropertyPath().toString();
            String message = violation.getMessage();
            errors.put(propertyPath, message);
        });
        return Map.of(
            "timestamp", LocalDateTime.now(),
            "status", 400,
            "errors", errors
        );
    }
}
```

## 8. Tests de validation

### 8.1 Test unitaire

```
@SpringBootTest
class UserValidationTest {
    @Autowired
    private Validator validator;
    @Test
    void shouldFailWhenEmailInvalid() {
        // Given
        User user = new User();
        user.setUsername("john");
        user.setEmail("invalid-email");
        // When
        Set<ConstraintViolation<User>> violations = validator.validate(user);
        // Then
        assertThat(violations).hasSize(1);
        assertThat(violations)
            .extracting(v -> v.getPropertyPath().toString())
            .containsExactly("email");
    }
    @Test
    void shouldValidateSuccessfully() {
        // Given
        User user = new User();
        user.setUsername("john");
        user.setEmail("john@example.com");
        user.setAge(25);
        // When
        Set<ConstraintViolation<User>> violations = validator.validate(user);
        // Then
        assertThat(violations).isEmpty();
    }
}
```

### 8.2 Test d'intégration Controller

```
@SpringBootTest
@AutoConfigureMockMvc
class UserControllerValidationTest {
    @Autowired
    private MockMvc mockMvc;
    @Autowired
    private ObjectMapper objectMapper;
    @Test
    void shouldReturn400WhenUserInvalid() throws Exception {
        // Given
        User invalidUser = new User();
        invalidUser.setUsername("ab");  // Trop court
        invalidUser.setEmail("invalid");
        // When/Then
        mockMvc.perform(post("/api/users")
                .contentType(MediaType.APPLICATION_JSON)
                .content(objectMapper.writeValueAsString(invalidUser)))
            .andExpect(status().isBadRequest())
            .andExpect(jsonPath("$.errors.username").exists())
            .andExpect(jsonPath("$.errors.email").exists());
    }
}
```

## 9. Bonnes pratiques



**DO**

- Valider au plus tôt (couche controller/API)
- Utiliser `@Valid` sur les objets imbriqués
- Créer des annotations custom pour logique métier complexe
- Utiliser les groups pour contextes différents (create/update)
- Centraliser la gestion d'erreurs avec `@RestControllerAdvice`

**DON'T**

- Ne pas dupliquer la validation dans plusieurs couches
- Éviter la validation dans les getters/setters
- Ne pas ignorer les `ConstraintViolation` retournées
- Ne pas mélanger validation technique et règles métier complexes

## 10. Aide-mémoire

| Annotation | Usage | Exemple |
|---|---|---|
| `@NotNull` | Interdit null | `@NotNull UUID id` |
| `@NotBlank` | String non null/vide/espaces | `@NotBlank String name` |
| `@Email` | Format email valide | `@Email String email` |
| `@Size` | Taille min/max | `@Size(min=3, max=50)` |

| Annotation | Usage | Exemple |
|---|---|---|
| `@Min` / `@Max` | Valeur numérique min/max | `@Min(0) Integer stock` |
| `@Pattern` | Regex | `@Pattern(regexp="[A-Z]{2}")` |
| `@Past` / `@Future` | Date passée/future | `@Past LocalDate birth` |
| `@Valid` | Validation en cascade | `@Valid Address address` |
| `@Validated` | Active validation méthodes | `@Validated` sur classe |

## Ressources

- Spring Validation Documentation
- Bean Validation Specification
- Baeldung - Java Bean Validation

From:
http://slamwiki2.kobject.net/ - **SlamWiki 2.1**

Permanent link:
**http://slamwiki2.kobject.net/framework-web/spring/validation**

Last update: **2025/10/08 00:45**