Java Bean Validation

Builds on Spring MVC's "Validation & Exception Handling" lesson -- sign constraints (@Positive/@PositiveOrZero/@Negative/@NegativeOrZero), date/time constraints (@Past /@Future/@PastOrPresent/@FutureOrPresent), precise decimal bounds (@DecimalMin /@DecimalMax/@Digits), customizing validation messages, custom constraint annotations with ConstraintValidator for cross-field validation, and validation groups. The 1st lesson in the Advanced Spring series.

Advanced 35 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/java-bean-validation

---

"Validation & Exception Handling," in Spring MVC, already covered the everyday core of Bean Validation: `@NotNull` / `@NotEmpty` / `@NotBlank`, `@Size` / `@Min` / `@Max`, `@Email` / `@Pattern`, `@Valid`, the `Validator` / `ConstraintViolation` machinery, and cascading validation on nested objects. This lesson doesn't repeat any of that. It picks up exactly where that lesson left off — the built-in constraints it didn't cover, customizing the messages a violation produces, and writing your own validation rules when the built-in ones simply can't express what you need.

Beyond the Basics: What This Lesson Builds On

Everything here assumes you're already comfortable with `@Valid` triggering validation on a `@RequestBody`, and with reading a `ConstraintViolation`. What's new: constraints for signed numbers and dates, constraints for exact decimal precision, message customization, and — the biggest jump — building your own constraint annotation for rules the built-in ones can't express, including rules that span more than one field at once.

Sign Constraints: @Positive, @PositiveOrZero, @Negative, @NegativeOrZero

Four constraints check a numeric value's sign, each either strict or inclusive of zero.

```
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.Negative;
import jakarta.validation.constraints.NegativeOrZero;
import jakarta.validation.constraints.Positive;
import jakarta.validation.constraints.PositiveOrZero;

// Four sign-based constraints, each strict or inclusive of zero:
// @Positive        -- must be > 0 (zero itself fails)
// @PositiveOrZero   -- must be >= 0
// @Negative        -- must be < 0 (zero itself fails)
// @NegativeOrZero   -- must be <= 0
// Exactly like @Min/@Max, "Positive"/"Negative" alone are STRICT, and
// "OrZero" is what makes zero acceptable -- there's no separate
// "at-least-zero" annotation because @PositiveOrZero already is that.
class SignConstraintsExample {

    record StockAdjustment(
            @Positive int quantityAdded,       // restocking: must add at least 1
            @PositiveOrZero int reservedUnits,  // reserving zero units is a valid no-op
            @Negative int quantityRemoved,      // removals are recorded as negative deltas
            @NegativeOrZero int discountCents) { // a discount of exactly 0 cents is allowed
    }

    public static void main(String[] args) {
        Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

        System.out.println(validator.validate(new StockAdjustment(0, 5, -3, -100)).size());
        // 1 -- quantityAdded is 0, but @Positive requires strictly greater than 0

        System.out.println(validator.validate(new StockAdjustment(10, 0, -3, 0)).size());
        // 0 -- reservedUnits and discountCents are exactly 0, and both allow zero
    }
}
```

`@Positive` and `@Negative` are STRICT — zero itself fails both. `@PositiveOrZero` and `@NegativeOrZero` allow zero. Recall that `@Min` / `@Max` are inclusive bounds — `@Min(16)` means "at least 16," not "greater than 16." `@Positive` and `@Negative` work differently: they exclude zero by definition, which is exactly why the "OrZero" variants exist as a separate, explicit choice rather than something you'd express with `@Min(0)`.

Date and Time Constraints: @Past, @Future, @PastOrPresent, @FutureOrPresent

Four more constraints check a date or date-time value against the clock at the moment validation runs, not against any fixed date.

```
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.Future;
import jakarta.validation.constraints.FutureOrPresent;
import jakarta.validation.constraints.Past;
import jakarta.validation.constraints.PastOrPresent;

import java.time.LocalDate;

// Four date/time constraints, compared to "now" at the moment of validation:
// @Past             -- must be strictly before now
// @PastOrPresent     -- must be now or before
// @Future           -- must be strictly after now
// @FutureOrPresent   -- must be now or after
// These work on any date/time type (LocalDate, LocalDateTime, Date, ...) --
// the comparison is always made against the clock at validation time, not
// against some fixed date.
class DateTimeConstraintsExample {

    record ReservationRequest(
            @Past LocalDate dateOfBirth,             // a birth date must already have
happened
            @FutureOrPresent LocalDate checkInDate,    // check-in today or later is fine
            @Future LocalDate checkOutDate) {          // check-out must be strictly in the
future
    }

    public static void main(String[] args) {
        Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

        LocalDate today = LocalDate.now();

        System.out.println(validator.validate(
                new ReservationRequest(today.plusDays(1), today, today.plusDays(3))).
size());
        // 1 -- dateOfBirth is tomorrow, but @Past requires a date before today

        System.out.println(validator.validate(
                new ReservationRequest(today.minusYears(30), today, today.plusDays(3))).
size());
        // 0 -- a birth date 30 years ago, check-in today, check-out in the future
    }
}
```

`@Past` and `@Future` are strict — "now" itself fails both. `@PastOrPresent` and `@FutureOrPresent` allow the current moment. These work on any date/time type Bean Validation recognizes — `LocalDate`, `LocalDateTime`, `java.util.Date`, and others — the comparison logic is identical regardless of which one you use.

Precise Decimal Bounds: @DecimalMin, @DecimalMax, and @Digits

`@Min` / `@Max` only accept whole-number bounds and only apply to integer types. `@DecimalMin` / `@DecimalMax` exist for exactly what they can't handle: fractional bounds on a `BigDecimal` (or `double` / `float`) field.

```
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.DecimalMax;
import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Digits;

import java.math.BigDecimal;

// @Min/@Max only accept whole-number bounds and only apply to integer types
// (int, long, ...). @DecimalMin/@DecimalMax exist for exactly the case
// @Min/@Max can't handle: fractional bounds on a BigDecimal (or double/
// float) field -- the bound is written as a String so it can express an
// exact decimal value like "0.01" without floating-point rounding.
class DecimalBoundsAndDigitsExample {

    record PriceUpdate(
            @DecimalMin("0.01") @DecimalMax("9999.99") BigDecimal price,
            // @Digits caps how many digits are allowed before and after the
            // decimal point -- here, at most 4 whole-number digits and
            // exactly 2 fractional digits, matching how currency is stored.
            @Digits(integer = 4, fraction = 2) BigDecimal displayPrice) {
    }

    public static void main(String[] args) {
        Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

        System.out.println(validator.validate(
                new PriceUpdate(new BigDecimal("0.00"), new BigDecimal("19.99"))).size());
        // 1 -- 0.00 is below the 0.01 minimum

        System.out.println(validator.validate(
                new PriceUpdate(new BigDecimal("19.99"), new BigDecimal("19.999"))).size());
        // 1 -- displayPrice has 3 fractional digits, but @Digits allows only 2

        System.out.println(validator.validate(
                new PriceUpdate(new BigDecimal("19.99"), new BigDecimal("19.99"))).size());
        // 0 -- both fields satisfy their bounds
    }
}
```

The bound is written as a `String` — `@DecimalMin("0.01")`, not `@DecimalMin(0.01)` — so it can express an exact decimal value without any floating-point rounding creeping in. `@Digits(integer = 4, fraction = 2)` is a related but different check: instead of a range, it caps how many digits are allowed before and after the decimal point, which is exactly how currency amounts are usually constrained.

Combining Constraints on a Realistic DTO

None of these constraints exist in isolation — a real request DTO combines several at once, exactly as you'd expect from "Validation & Exception Handling"'s `@Valid` coverage.

```
import jakarta.validation.Valid;
import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Digits;
import jakarta.validation.constraints.Future;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Positive;
import jakarta.validation.constraints.Size;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;

import java.math.BigDecimal;
import java.time.LocalDate;

// A single, realistic DTO combining constraints from Spring MVC's
// "Validation & Exception Handling" (@NotBlank, @Size) with the ones
// covered so far in this lesson (@Positive, @DecimalMin, @Digits,
// @Future) -- exactly how a real product-creation endpoint would look.
// @Valid on the controller parameter works the same way covered there:
// every constraint below runs before create(...)'s body executes.
@RestController
class ProductController {

    record CreateProductRequest(
            @NotBlank @Size(max = 100) String name,
            @Positive int stockQuantity,
            @DecimalMin("0.01") @Digits(integer = 6, fraction = 2) BigDecimal price,
            @Future LocalDate availableFrom) {
    }

    @PostMapping("/products")
    @ResponseBody
    public String create(@Valid @RequestBody CreateProductRequest request) {
        return "Created: " + request.name();
    }
}
```

`CreateProductRequest` mixes constraints already familiar from Spring MVC's lesson (`@NotBlank` , `@Size`) with the ones covered so far here (`@Positive` , `@DecimalMin` , `@Digits` , `@Future`) — and `@Valid` on the controller parameter runs every single one of them before `create(...)` 's body ever executes, exactly the mechanism already covered there.

Customizing Validation Messages

Every constraint accepts a `message` attribute. Left unset, you get the library's default English wording; set explicitly, you control exactly what a violation reports.

```
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;

// Every constraint accepts a "message" attribute -- without it, you get
// the library's default English wording. Two ways to override it:
// (1) a literal string, written directly on the annotation;
// (2) a "{...}" placeholder, resolved from a messages.properties file on
//     the classpath (the standard Spring i18n mechanism, reused here) --
//     this is what lets the SAME constraint produce a translated message
//     depending on the active locale, instead of a hardcoded string.
class CustomValidationMessageExample {

    record RegisterUserRequest(
            @NotBlank(message = "Display name is required") String displayName,
            @Min(value = 16, message = "{user.age.tooYoung}") int age) {
        // messages.properties: user.age.tooYoung=You must be at least 16 years old
        // messages_tr.properties: user.age.tooYoung=En az 16 yaşında olmalısınız
    }

    public static void main(String[] args) {
        Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

        validator.validate(new RegisterUserRequest("", 12)).forEach(violation ->
                System.out.println(violation.getPropertyPath() + ": " + violation.
getMessage()));
        // displayName: Display name is required
        // age: {user.age.tooYoung} -- a plain Validator instance (used
        //     directly here, outside Spring) does not resolve message
        //     bundles; inside a Spring MVC controller, this placeholder is
        //     automatically resolved through the same MessageSource
        //     mechanism used for regular UI text.
    }
}
```

A literal string (`message = "Display name is required"`) is the simplest override. A `{...}` placeholder (`message = "{user.age.tooYoung}"`) is resolved from a `messages.properties` file on the classpath instead — the same Spring i18n mechanism used for ordinary UI text — which is what lets the exact same constraint produce a message in the active locale rather than a single hardcoded string.

Building a Custom, Cross-Field Constraint

Some rules simply can't be expressed by any single-field, built-in constraint — "checkout must be after check-in" depends on TWO fields at once, and no annotation on either field

alone could check that relationship. A custom constraint has two parts: the annotation itself (declaring which `ConstraintValidator` implements it) and the validator class (containing the actual check).

```
import jakarta.validation.Constraint;
import jakarta.validation.ConstraintValidator;
import jakarta.validation.ConstraintValidatorContext;
import jakarta.validation.Payload;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.time.LocalDate;

// A custom constraint has two parts: the annotation itself (declaring
// which ConstraintValidator implements it), and the validator class
// (containing the actual check). Placed at CLASS level (not on a single
// field) because the rule -- "checkOut must be after checkIn" -- depends
// on TWO fields at once; no single-field annotation like @Future could
// express this on its own.
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = DateRangeValidator.class)
@interface ValidDateRange {
    String message() default "checkOutDate must be after checkInDate";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}

class DateRangeValidator implements ConstraintValidator<ValidDateRange,
CrossFieldCustomConstraintExample.BookingRequest> {

    @Override
    public boolean isValid(CrossFieldCustomConstraintExample.BookingRequest booking,
ConstraintValidatorContext context) {
        if (booking.checkInDate() == null || booking.checkOutDate() == null) {
            return true; // let @NotNull (if present) report missing values -- this
            //               validator only checks the RELATIONSHIP between the two
        }
        return booking.checkOutDate().isAfter(booking.checkInDate());
    }
}

public class CrossFieldCustomConstraintExample {

    @ValidDateRange
    record BookingRequest(LocalDate checkInDate, LocalDate checkOutDate) {
    }

    public static void main(String[] args) {
        var validator = jakarta.validation.Validation.buildDefaultValidatorFactory().
getValidator();

        LocalDate today = LocalDate.now();

        System.out.println(validator.validate(new BookingRequest(today, today.minusDays
```

```
(1))).size());
        // 1 -- checkOutDate is before checkInDate

        System.out.println(validator.validate(new BookingRequest(today, today.plusDays(3))).
size());
        // 0 -- a valid range
    }
}
```

`@ValidDateRange` is placed at CLASS level, not on a single field, precisely because its rule needs to see both `checkInDate` and `checkOutDate` at once. `DateRangeValidator implements ConstraintValidator<ValidDateRange, BookingRequest>` receives the whole `BookingRequest`, not one field's value, and returns `true` / `false` from `isValid(...)` — the exact same interface backing every built-in constraint you've already used, just implemented by hand.

> A custom `ConstraintValidator` returning `true` for a `null` value (as `DateRangeValidator` does here) is the standard convention — it lets a separate `@NotNull` constraint be the one responsible for reporting a missing value, keeping each constraint focused on exactly one concern.

Validation Groups

Validation groups let the SAME DTO enforce different constraints depending on which operation is running — without them, "`id` must be absent on create" and "`id` is required on update" couldn't both apply to one class at once.

```
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.Null;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;

// Validation groups let the SAME DTO enforce different constraints
// depending on which operation is running -- without groups, "id must be
// null on create" and "id must be present on update" couldn't both apply
// to one class. Each constraint is tagged with which group(s) it belongs
// to; validate(object, SomeGroup.class) then runs ONLY the constraints
// tagged for that group.
class ValidationGroupsExample {

    interface OnCreate {}
    interface OnUpdate {}

    record UserRequest(
            @Null(groups = OnCreate.class) @NotNull(groups = OnUpdate.class) Long id,
            @NotBlank(groups = {OnCreate.class, OnUpdate.class}) String name) {
    }

    public static void main(String[] args) {
        Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

        System.out.println(validator.validate(new UserRequest(1L, "Alice"), OnCreate.class).
size());
        // 1 -- @Null(groups = OnCreate.class) fails: id must be absent when creating

        System.out.println(validator.validate(new UserRequest(null, "Alice"), OnUpdate.
class).size());
        // 1 -- @NotNull(groups = OnUpdate.class) fails: id is required when updating

        System.out.println(validator.validate(new UserRequest(null, "Alice"), OnCreate.
class).size());
        // 0 -- no id and a name: exactly what OnCreate requires
    }
}
```

Each constraint is tagged with the group(s) it belongs to (`@NotNull(groups = OnUpdate.class)`), and `validator.validate(request, OnCreate.class)` runs ONLY the constraints tagged for that group. This is a narrow tool for a specific shape of problem — reach for it when the same request type genuinely needs to enforce different rules per operation, not as a general-purpose validation mechanism.

Best Practices

- Use the strict sign/date constraints (`@Positive` , `@Future` , ...) by default, and only reach for the "OrZero"/"OrPresent" variant when the boundary value is genuinely a valid case.
- Prefer `@DecimalMin` / `@DecimalMax` string bounds over comparing a `BigDecimal` manually in application code — the constraint documents the rule directly on the field.

- Move every user-facing validation message into a `messages.properties` placeholder rather than hardcoding it, the moment your application needs to support more than one locale.
- Reach for a custom `ConstraintValidator` only when a rule genuinely can't be expressed with a combination of built-in constraints — and keep each custom constraint focused on one rule.

Common Mistakes

- Writing `@DecimalMin(0.01)` instead of `@DecimalMin("0.01")` — the bound must be a `String`, not a numeric literal.
- Assuming `@Positive` accepts zero, the way `@Min(0)` would — it doesn't; `@PositiveOrZero` is the constraint for that case.
- Trying to validate a cross-field rule with two independent single-field constraints instead of one class-level custom constraint that can actually see both fields at once.
- Forgetting that a validation group only runs the constraints explicitly tagged for it — an untagged constraint (no `groups` attribute at all) is silently skipped when validating against a specific group.

Summary, Cheat Sheet, and Glossary

Summary

- `@Positive` / `@Negative` are strict about zero; `@PositiveOrZero` / `@NegativeOrZero` allow it.
- `@Past` / `@Future` are strict about the current moment; `@PastOrPresent` / `@FutureOrPresent` allow it.
- `@DecimalMin` / `@DecimalMax` take `String` bounds for exact decimal precision; `@Digits` caps digit counts before and after the decimal point.
- A `message` attribute (a literal string or a `{...}` placeholder resolved from `messages.properties`) customizes what a violation reports.
- A custom constraint pairs an annotation with a `ConstraintValidator`; placing it at class level lets a rule span multiple fields at once.
- Validation groups let one DTO enforce different constraints depending on which group it's validated against.

Cheat Sheet

```
// Sign and date constraints
record Adjustment(@Positive int added, @PositiveOrZero int reserved) {}
record Booking(@FutureOrPresent LocalDate checkIn, @Future LocalDate checkOut) {}

// Decimal precision
record Price(@DecimalMin("0.01") @Digits(integer = 6, fraction = 2) BigDecimal amount) {}

// Custom message
@NotBlank(message = "Display name is required")
@Min(value = 16, message = "{user.age.tooYoung}")

// Custom, cross-field constraint
@Target(ElementType.TYPE)
@Constraint(validatedBy = DateRangeValidator.class)
@interface ValidDateRange { ... }

class DateRangeValidator implements ConstraintValidator<ValidDateRange, Booking> {
    public boolean isValid(Booking b, ConstraintValidatorContext ctx) {
        return b.checkOut().isAfter(b.checkIn());
    }
}

// Validation groups
validator.validate(request, OnCreate.class);
```

Glossary

- **Sign constraint**: a constraint checking a numeric value's sign, strict (`@Positive` / `@Negative`) or inclusive of zero (`@PositiveOrZero` / `@NegativeOrZero`).
- **Message placeholder**: a `{...}`-wrapped key in a constraint's `message` attribute, resolved from a `messages.properties` bundle.
- **Custom constraint**: a constraint annotation paired with a hand-written `ConstraintValidator`, for rules the built-in constraints can't express.
- **Cross-field validation**: a validation rule that depends on more than one field at once, typically implemented as a class-level custom constraint.
- **Validation group**: a marker interface used to tag which constraints should run for a particular validation call, letting one type enforce different rules in different contexts.