Java Bean Validation

Spring MVC'nin "Validation & Exception Handling" dersinin üzerine inşa edilir -- işaret kısıtları (@Positive/@PositiveOrZero/@Negative/@NegativeOrZero), tarih/saat kısıtları (@Past/@Future/@PastOrPresent/@FutureOrPresent), tam ondalık sınırlar (@DecimalMin /@DecimalMax/@Digits), validation mesajlarını özelleştirmek, custom constraint annotation + ConstraintValidator ile çapraz alan validasyonu, ve validation grupları. Advanced Spring serisinin 1.'si.

İleri 35 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/java-bean-validation

---

Spring MVC'deki "Validation & Exception Handling", Bean Validation'ın günlük çekirdeğini zaten işledi: `@NotNull` / `@NotEmpty` / `@NotBlank`, `@Size` / `@Min` / `@Max`, `@Email` / `@Pattern`, `@Valid`, `Validator` / `ConstraintViolation` mekanizması, ve iç içe nesnelerde cascading validation. Bu ders bunların hiçbirini tekrarlamıyor. Tam olarak o dersin bıraktığı yerden devam ediyor — orada işlenmeyen hazır kısıtlar, bir violation'ın ürettiği mesajları özelleştirmek, ve hazır kısıtların gerçekten ifade edemediği kurallar için kendi validation kuralını yazmak.

Temellerin Ötesi: Bu Ders Neyin Üzerine İnşa Ediyor

Buradaki her şey, `@Valid`'in bir `@RequestBody` üzerinde validasyonu nasıl tetiklediğini ve bir `ConstraintViolation`'ı nasıl okuyacağını zaten bildiğini varsayıyor. Yeni olan: işaretli sayılar ve tarihler için kısıtlar, tam ondalık hassasiyet için kısıtlar, mesaj özelleştirmesi, ve — en büyük sıçrama — hazır kısıtların ifade edemediği kurallar için, birden fazla alanı aynı anda kapsayan kurallar dahil, kendi kısıt annotation'ını yazmak.

İşaret Kısıtları: @Positive, @PositiveOrZero, @Negative, @NegativeOrZero

Dört kısıt, sayısal bir değerin işaretini kontrol eder, her biri ya sıkı ya da sıfırı içeren.

```
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` ve `@Negative` SIKI'dir — sıfırın kendisi ikisini de başarısız kılar. `@PositiveOrZero` ve `@NegativeOrZero` sıfıra izin verir. `@Min` / `@Max` 'in kapsayıcı sınırlar olduğunu hatırla — `@Min(16)`, "en az 16" demektir, "16'dan büyük" değil. `@Positive` ve `@Negative` farklı çalışır: tanımı gereği sıfırı dışlarlar, ve "OrZero" varyantlarının `@Min(0)` ile ifade edebileceğin bir şey yerine ayrı, açık bir seçim olarak var olmasının nedeni tam olarak budur.

Tarih ve Saat Kısıtları: @Past, @Future, @PastOrPresent, @FutureOrPresent

Dört kısıt daha, bir tarih ya da tarih-saat değerini, sabit bir tarihe değil, validasyonun çalıştığı andaki saate karşı kontrol eder.

```
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` ve `@Future` sıkıdır — "şimdi"nin kendisi ikisini de başarısız kılar. `@PastOrPresent` ve `@FutureOrPresent`, şu anki ana izin verir. Bunlar Bean Validation'ın tanıdığı herhangi bir tarih /saat türünde çalışır — `LocalDate`, `LocalDateTime`, `java.util.Date` ve diğerleri — hangisini kullandığından bağımsız olarak karşılaştırma mantığı aynıdır.

Tam Ondalık Sınırlar: @DecimalMin, @DecimalMax ve @Digits

`@Min` / `@Max`, yalnızca tam sayı sınırlarını kabul eder ve yalnızca tam sayı türlerine uygulanır. `@DecimalMin` / `@DecimalMax`, tam olarak bunların ele alamadığı şey için var: bir `BigDecimal` (ya da `double` / `float`) alanında kesirli sınırlar.

```
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
    }
}
```

Sınır bir `String` olarak yazılır — `@DecimalMin(0.01)` değil, `@DecimalMin("0.01")` — böylece herhangi bir floating-point yuvarlama sızıntısı olmadan tam bir ondalık değeri ifade edebilir. `@Digits(integer = 4, fraction = 2)`, ilişkili ama farklı bir kontroldür: bir aralık yerine, ondalık noktadan önce ve sonra kaç basamağa izin verildiğini sınırlar, ki bu tam olarak para tutarlarının genelde nasıl kısıtlandığıdır.

Gerçekçi Bir DTO'da Kısıtları Birleştirmek

Bu kısıtların hiçbiri izole var olmaz — gerçek bir request DTO'su, "Validation & Exception Handling"in `@Valid` kapsamından beklediğin gibi, birkaçını aynı anda birleştirir.

```
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`, Spring MVC'nin dersinden zaten tanıdık kısıtları (`@NotBlank`, `@Size`) burada şimdiye kadar işlenenlerle (`@Positive`, `@DecimalMin`, `@Digits`, `@Future`) karıştırıyor — ve controller parametresindeki `@Valid`, `create(...)`'in gövdesi hiç çalışmadan önce hepsini çalıştırır, tam olarak orada zaten işlenen mekanizma.

Validation Mesajlarını Özelleştirmek

Her kısıt bir `message` özniteliğini kabul eder. Ayarlanmazsa, kütüphanenin varsayılan İngilizce ifadesini alırsın; açıkça ayarlarsan, bir violation'ın tam olarak neyi raporladığını kontrol edersin.

```
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.
    }
}
```

Literal bir string (`message = "Display name is required"`), en basit geçersiz kılmadır. Bir `{...}` yer tutucusu (`message = "{user.age.tooYoung}"`), bunun yerine classpath'teki bir `messages.properties` dosyasından çözülür — sıradan UI metni için kullanılan aynı Spring i18n mekanizması — ve bu, aynı kısıtın tek bir sabit kodlanmış string yerine aktif locale'de bir mesaj üretmesini sağlayan şeydir.

Custom, Çapraz Alan Kısıtı Oluşturmak

Bazı kurallar, tek bir alana bağlı hazır bir kısıtla basitçe ifade edilemez — "checkout, check-in'den sonra olmalı", aynı anda İKİ alana bağlıdır, ve ne alandaki tek bir annotation bu ilişkiyi kontrol edemez. Custom bir kısıtın iki parçası vardır: annotation'ın kendisi (hangi `ConstraintValidator`'ın onu uyguladığını bildirir) ve validator sınıfı (gerçek kontrolü içerir).

```
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`, tek bir alanda değil, tam olarak kuralının `checkInDate` ve `checkOutDate`'i aynı anda görmesi gerektiği için SINIF seviyesine konur. `DateRangeValidator implements ConstraintValidator<ValidDateRange, BookingRequest>`, tek bir alanın değerini değil, bütün `BookingRequest`'i alır, ve `isValid(...)`'ten `true`/`false` döndürür — zaten kullandığın her hazır kısıtın arkasındaki AYNI interface, yalnızca elle uygulanmış.

> Custom bir `ConstraintValidator`'ın `null` bir değer için `true` döndürmesi (burada `DateRangeValidator`'ın yaptığı gibi) standart kuraldır — bu, eksik bir değeri raporlama sorumluluğunu ayrı bir `@NotNull` kısıtına bırakır, her kısıtı tam olarak tek bir konuya odaklı tutar.

Validation Grupları

Validation grupları, AYNI DTO'nun hangi işlemin çalıştığına bağlı olarak farklı kısıtları uygulamasına izin verir — bunlar olmadan, "`id`, create'te olmamalı" ve "`id`, update'te zorunlu" ikisi de aynı sınıfa aynı anda uygulanamazdı.

```
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
    }
}
```

Her kısıt ait olduğu grup(lar)la etiketlenir (`@NotNull(groups = OnUpdate.class)`), ve `validator.validate(request, OnCreate.class)` , YALNIZCA o grup için etiketlenmiş kısıtları çalıştırır. Bu, belirli bir sorun şekli için dar bir araçtır — aynı request türünün işlem başına gerçekten farklı kurallar uygulaması gerektiğinde başvur, genel amaçlı bir validation mekanizması olarak değil.

Best Practices

- Varsayılan olarak sıkı işaret/tarih kısıtlarını (`@Positive` , `@Future` , ...) kullan, ve sınır değeri gerçekten geçerli bir durum olduğunda "OrZero"/"OrPresent" varyantına başvur.
- Bir `BigDecimal` 'i uygulama kodunda elle karşılaştırmak yerine `@DecimalMin` / `@DecimalMax` string sınırlarını tercih et — kısıt, kuralı doğrudan alanda belgeler.

- Uygulaman birden fazla locale desteklemesi gerektiği anda, her kullanıcıya yönelik validation mesajını sabit kodlamak yerine bir `messages.properties` yer tutucusuna taşı.
- Custom bir `ConstraintValidator`'a yalnızca bir kural gerçekten hazır kısıtların birleşimiyle ifade edilemediğinde başvur — ve her custom kısıtı tek bir kurala odaklı tut.

Yaygın Hatalar

- `@DecimalMin("0.01")` yerine `@DecimalMin(0.01)` yazmak — sınır bir sayısal literal değil, bir `String` olmalıdır.
- `@Positive`'in, `@Min(0)`'ın yapacağı gibi sıfırı kabul ettiğini varsaymak — etmez; bu durum için kısıt `@PositiveOrZero`'dur.
- Çapraz alan bir kuralı, gerçekte her iki alanı aynı anda görebilen tek bir sınıf-seviyesi custom kısıt yerine iki bağımsız tek-alan kısıtıyla doğrulamaya çalışmak.
- Bir validation grubunun yalnızca kendisi için açıkça etiketlenmiş kısıtları çalıştırdığını unutmak — etiketlenmemiş bir kısıt (hiç `groups` özniteliği olmayan) belirli bir gruba karşı doğrulama yapılırken sessizce atlanır.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- `@Positive` / `@Negative` sıfır konusunda sıkıdır; `@PositiveOrZero` / `@NegativeOrZero` sıfıra izin verir.
- `@Past` / `@Future` şu anki an konusunda sıkıdır; `@PastOrPresent` / `@FutureOrPresent` ona izin verir.
- `@DecimalMin` / `@DecimalMax`, tam ondalık hassasiyet için `String` sınırları alır; `@Digits`, ondalık noktadan önce ve sonraki basamak sayısını sınırlar.
- Bir `message` özniteliği (literal bir string ya da `messages.properties`'ten çözülen bir `{...}` yer tutucusu), bir violation'ın neyi raporladığını özelleştirir.
- Custom bir kısıt, bir annotation'ı bir `ConstraintValidator`'la eşleştirir; onu sınıf seviyesine koymak, bir kuralın aynı anda birden fazla alanı kapsamasına izin verir.
- Validation grupları, tek bir DTO'nun hangi gruba karşı doğrulandığına bağlı olarak farklı kısıtları uygulamasına izin verir.

Cheat Sheet

```
// İşaret ve tarih kısıtları
record Adjustment(@Positive int added, @PositiveOrZero int reserved) {}
record Booking(@FutureOrPresent LocalDate checkIn, @Future LocalDate checkOut) {}

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

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

// Custom, çapraz alan kısıtı
@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 grupları
validator.validate(request, OnCreate.class);
```

Terimler Sözlüğü

- **İşaret kısıtı**: sayısal bir değerin işaretini kontrol eden, sıkı (`@Positive` / `@Negative`) ya da sıfırı içeren (`@PositiveOrZero` / `@NegativeOrZero`) bir kısıt.
- **Mesaj yer tutucusu**: bir kısıtın `message` özniteliğindeki, bir `messages.properties` paketinden çözülen `{...}` ile sarmalanmış bir anahtar.
- **Custom kısıt**: hazır kısıtların ifade edemediği kurallar için, elle yazılmış bir `ConstraintValidator`'la eşleştirilmiş bir kısıt annotation'ı.
- **Çapraz alan validasyonu (cross-field validation)**: aynı anda birden fazla alana bağlı, genelde sınıf-seviyesi bir custom kısıt olarak uygulanan bir validation kuralı.
- **Validation grubu**: belirli bir validation çağrısı için hangi kısıtların çalışacağını etiketlemek amacıyla kullanılan, bir türün farklı bağlamlarda farklı kurallar uygulamasına izin veren bir marker interface.