Validation & Exception Handling

Bean Validation (@NotBlank, @Size, @Email, @Pattern) ile @Valid, iç içe nesnelerde cascading, @ExceptionHandler, @RestControllerAdvice ve RFC 7807 ProblemDetail ile standart hata yönetimi.

Orta 45 dk

Son güncelleme: 11 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/validation-exception-handling

---

Validation & Exception Handling

Request ve Response Handling dersinde Jackson'ın `@RequestBody` gövdesini yalnızca **biçim** olarak doğruladığını görmüştük -- JSON geçerli mi, tipler uyuşuyor mu. İş kurallarını (boş olmayan bir isim, pozitif bir miktar, geçerli bir e-posta) doğrulamak sana kalıyordu. Bu ders, o boşluğu iki mekanizmayla kapatıyor: **Bean Validation** (`@Valid` ve arkadaşları), bir isteği controller'a ulaşmadan **önce** reddetmenin standart yolu; **exception handling** (`@ExceptionHandler` , `@RestControllerAdvice` , `ProblemDetail`), bir hata oluştuğunda istemciye tutarlı, standart bir yanıt döndürmenin yolu.

Validation & Exception Handling Nedir?

Bean Validation, bir Java nesnesinin alanlarına **annotation ile kural** yazma ve bu kuralları tek bir çağrıyla kontrol etme standardıdır (JSR-380, `jakarta.validation` paketi). Exception handling ise, bir controller metodunda (ya da validation'da) oluşan bir hatayı, dağınık `try` / `catch` bloklarına gerek kalmadan, merkezi ve tutarlı bir HTTP yanıtına çevirme mekanizmasıdır:

```
record CreateUserRequest(@NotBlank String name, @Email String email) { }

@PostMapping("/users")
public String create(@Valid @RequestBody CreateUserRequest request) {
    // buraya yalnızca name boş değilse ve email geçerliyse ulaşılır
    return "Created: " + request.name();
}
```

Neden Var?

Validation kuralını her controller metodunun başına elle yazmak (`if (name == null || name.isBlank()) throw ...`) hem tekrarlıdır hem unutulmaya açıktır -- bir alan eklenir, kontrolü eklemeyi unutursun. Bean Validation, kuralı **veri tipinin kendisine** taşır: `CreateUserRequest` nerede kullanılırsa kullanılsın, `@NotBlank` kuralı onunla birlikte gelir. Benzer şekilde, her `catch` bloğunda elle bir hata gövdesi inşa etmek tutarsız sonuçlar üretir (bir yerde düz metin, başka bir yerde JSON, bir başkasında hiçbir şey); `@ExceptionHandler` / `@RestControllerAdvice`, bu dönüşümü tek bir yerde toplar.

Tarihçe

Bean Validation, Java EE 6 ile 2009'da JSR-303 olarak standartlaştırıldı; `jakarta.validation` adına geçişi (Java EE'nin Jakarta EE'ye taşınmasıyla) izleyen JSR-380 (Bean Validation 2.0), `@NotEmpty` / `@NotBlank` gibi artık tanıdık annotation'ları ekledi. Hibernate Validator, bu standardın referans implementasyonudur -- `spring-boot-starter-validation` bağımlılığı, projeye tam da bunu (ve Spring'in `@Valid` entegrasyonunu) kazandırır. `@ExceptionHandler` Spring 3.0'da geldi; `@ControllerAdvice` (global karşılığı) Spring 3.2'de, `ProblemDetail` (RFC 7807 desteği) ise Spring 6 / Spring Boot 3'te eklendi.

@NotNull, @NotEmpty, @NotBlank: Boşluk Farkları

Üç annotation da "değer eksik olmasın" der, ama her biri farklı bir eşiği kontrol eder:

```
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotEmpty;
import jakarta.validation.constraints.NotNull;

// Three annotations that sound similar but check different things:
//   @NotNull  -- the value must not be null (an empty string still passes)
//   @NotEmpty -- must not be null AND not empty (a whitespace-only string still passes)
//   @NotBlank -- must not be null, not empty, AND not just whitespace
class NotNullBlankEmptyExample {

    record NotNullField(@NotNull String value) {
    }

    record NotEmptyField(@NotEmpty String value) {
    }

    record NotBlankField(@NotBlank String value) {
    }

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

        System.out.println(validator.validate(new NotNullField(null)).size());
        // 1
        System.out.println(validator.validate(new NotNullField("")).size());
        // 0 -- @NotNull allows an empty string

        System.out.println(validator.validate(new NotEmptyField("")).size());
        // 1
        System.out.println(validator.validate(new NotEmptyField("   ")).size());
        // 0 -- @NotEmpty allows a whitespace-only string

        System.out.println(validator.validate(new NotBlankField("   ")).size());
        // 1 -- @NotBlank rejects whitespace-only
        System.out.println(validator.validate(new NotBlankField("x")).size());
        // 0
    }
}
```

`@NotNull`, yalnızca `null` olmamasını ister -- boş bir string (`""`) geçer. `@NotEmpty`, `null` da boş string de reddeder -- ama yalnızca boşluklardan oluşan bir string (`" "`) geçer. `@NotBlank`, üçünü de reddeder -- pratikte kullanıcıdan gelen metin alanları için en sık istenen budur.

@Size, @Min, @Max: Sayısal ve Uzunluk Sınırları

`@Size`, bir string/koleksiyon/dizinin **uzunluğunu**; `@Min` / `@Max`, sayısal bir değerin **aralığını** kontrol eder -- her iki sınır da dahildir (inclusive):

```
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.Size;

// @Size checks a String/Collection/array's length; @Min/@Max check a numeric value's
// bounds -- both boundaries are inclusive.
class SizeMinMaxExample {

    record CreateProductRequest(
            @Size(min = 3, max = 50) String name,
            @Min(1) @Max(1000) int quantity) {
    }

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

        System.out.println(validator.validate(new CreateProductRequest("ab", 5)).size());
        // 1 -- "ab" is shorter than the 3-character minimum

        System.out.println(validator.validate(new CreateProductRequest("Keyboard", 0)).
size());
        // 1 -- 0 is below the minimum of 1

        System.out.println(validator.validate(new CreateProductRequest("Keyboard", 2000)).
size());
        // 1 -- 2000 is above the maximum of 1000

        System.out.println(validator.validate(new CreateProductRequest("Keyboard", 5)).
size());
        // 0 -- within every bound
    }
}
```

`@Size(min = 3, max = 50)`, tam 3 ya da tam 50 karakteri kabul eder, 2 ya da 51'i reddeder; `@Min(1) @Max(1000)` de aynı şekilde 1 ve 1000'i kabul eder.

@Email ve @Pattern: Biçim Doğrulama

`@Email`, sözdizimsel olarak geçerli bir e-posta biçimini kontrol eder; `@Pattern`, verdiğin herhangi bir düzenli ifadeye (regex) karşı kontrol eder -- kendi kuralını yazabildiğin için en esnek annotation'dır:

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.Pattern;

import java.util.Set;

// @Email checks for a syntactically valid email address; @Pattern checks against any
// regular expression you provide -- and, unlike most constraints, is commonly given a
// custom `message` because "must match ^[a-z0-9_]{3,16}$" means nothing to a user.
class EmailPatternExample {

    record CreateUserRequest(
            @Email String email,
            @Pattern(
                    regexp = "^[a-z0-9_]{3,16}$",
                    message = "username must be 3-16 lowercase letters, digits, or
underscores"
            ) String username) {
    }

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

        System.out.println(validator.validate(new CreateUserRequest("not-an-email",
"ayse_92")).size());
        // 1

        Set<ConstraintViolation<CreateUserRequest>> violations =
                validator.validate(new CreateUserRequest("ayse@example.com", "AY"));
        System.out.println(violations.size());
        // 1
        violations.forEach(v -> System.out.println(v.getMessage()));
        // username must be 3-16 lowercase letters, digits, or underscores
    }
}
```

`@Pattern`'e verilen `message` attribute'una dikkat et: çoğu annotation'ın varsayılan mesajı ("must match ...") kullanıcıya bir şey ifade etmez; `@Pattern` gibi serbest biçimli kurallarda okunur bir `message` yazmak neredeyse zorunludur.

@Valid ile İstek Gövdesini Doğrulamak

`@RequestParam` / `@PathVariable`'ın aksine, Bean Validation kuralları kendiliğinden çalışmaz -- bir parametrenin önüne `@Valid` koymak, Spring'e "bu nesneyi controller metodu çalışmadan önce doğrula" der:

```
import jakarta.validation.Valid;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;

// @Valid on a @RequestBody parameter tells Spring to run the same kind of Validator
// used directly in ManualValidatorExample BEFORE this method's body ever executes --
// if any constraint fails, create(...) is never called at all.
@Controller
class UserController {

    record CreateUserRequest(@NotBlank String name, @Email String email) {
    }

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

`@NotBlank` / `@Email` kısıtlarından biri bile başarısız olursa, `create(...)` metodunun gövdesi **hiç çalışmaz** -- Spring, metodu çağırmadan önce isteği reddeder. Bu doğrulamanın gerçekte nasıl çalıştığını "@Valid'in Perde Arkası: Validator ve ConstraintViolation" bölümünde göreceğiz.

@Valid'in Perde Arkası: Validator ve ConstraintViolation

`@Valid`, kendi doğrulama motorunu icat etmez -- `jakarta.validation.Validator`'ı (container'dan tamamen bağımsız, doğrudan da kullanılabilen bir arayüz) çağırır ve sonucu senin için yorumlar:

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;

import java.util.Set;

// @Valid doesn't invent a new validation engine -- it triggers exactly this: a
// jakarta.validation.Validator (the same interface regardless of framework) checks
// every constraint annotation on the object and returns a set of violations. When
// @Valid fails on a @RequestBody, Spring wraps this same result in a
// MethodArgumentNotValidException (carrying a BindingResult) instead of returning it
// to you directly -- the underlying check is identical to what's shown here.
class ManualValidatorExample {

    record CreateUserRequest(@NotBlank String name, @Email String email) {
    }

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

        CreateUserRequest invalid = new CreateUserRequest("", "not-an-email");
        Set<ConstraintViolation<CreateUserRequest>> violations = validator.validate
(invalid);
        System.out.println("Violation count: " + violations.size());
        // Violation count: 2

        CreateUserRequest valid = new CreateUserRequest("Ayse", "ayse@example.com");
        System.out.println("Valid request violations: " + validator.validate(valid).size());
        // Valid request violations: 0
    }
}
```

`validator.validate(nesne)`, ihlal edilen her kural için bir `ConstraintViolation` içeren bir `Set` döndürür; küme boşsa nesne geçerlidir. `@Valid @RequestBody` başarısız olduğunda, Spring bu aynı sonucu doğrudan sana vermez -- `MethodArgumentNotValidException` içine sarıp (bir `BindingResult` taşıyarak) fırlatır; bu, bir sonraki iki bölümde göreceğimiz `@ExceptionHandler` ile yakalanabilir.

İç İçe Nesnelerde Doğrulama: Cascading ile @Valid

Bean Validation, iç içe bir nesnenin alanlarını **varsayılan olarak** kontrol etmez -- iç nesnenin de doğrulanmasını istiyorsan, o alanın önüne de ayrıca `@Valid` koymak gerekir (buna cascading, "basamaklama" denir):

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

// Bean Validation does NOT automatically validate nested objects -- without @Valid on
// the nested field, its own constraints are silently skipped. Adding @Valid makes the
// validator recurse (cascade) into it too.
class NestedValidationExample {

    record Address(@NotBlank String city) {
    }

    record ShippingRequestWithoutCascade(@NotBlank String customerName, Address address) {
    }

    record ShippingRequestWithCascade(@NotBlank String customerName, @Valid Address
address) {
    }

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

        var withoutCascade = new ShippingRequestWithoutCascade("Ayse", new Address(""));
        System.out.println("Without @Valid on the nested field: " + validator.validate
(withoutCascade).size());
        // Without @Valid on the nested field: 0

        var withCascade = new ShippingRequestWithCascade("Ayse", new Address(""));
        System.out.println("With @Valid on the nested field: " + validator.validate
(withCascade).size());
        // With @Valid on the nested field: 1
    }
}
```

`ShippingRequestWithoutCascade`'in `address` alanının önünde `@Valid` yoktur -- `Address`'in kendi `@NotBlank` kuralı hiç çalıştırılmaz, sonuç her zaman 0 ihlaldir. `ShippingRequestWithCascade`'de `@Valid Address address` ile bu basamaklama açılır ve iç nesnenin ihlalleri de kümeye eklenir.

Controller-Seviyesinde Hata Yakalama: @ExceptionHandler

`@ExceptionHandler`, bir controller'ın içindeki bir metoda konduğunda, **aynı** controller'daki herhangi bir handler metodun fırlattığı belirtilen türden bir exception'ı yakalar:

```
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.ResponseStatus;

import java.util.Map;

// @ExceptionHandler, on a method INSIDE a controller, catches exceptions thrown by
// any handler method in that SAME controller -- no manual try/catch needed in every
// method.
@Controller
class ProductController {
    private final Map<Long, String> products = Map.of(1L, "Keyboard");

    static class ProductNotFoundException extends RuntimeException {
        ProductNotFoundException(Long id) {
            super("Product not found: " + id);
        }
    }

    @GetMapping("/products/{id}")
    @ResponseBody
    public String getProduct(@PathVariable Long id) {
        String product = products.get(id);
        if (product == null) {
            throw new ProductNotFoundException(id);
        }
        return product;
    }

    @ExceptionHandler(ProductNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ResponseBody
    public String handleNotFound(ProductNotFoundException e) {
        return e.getMessage();
    }
}
```

`getProduct(...)` bir `ProductNotFoundException` fırlattığında, çağıran kodun gördüğü bir exception değil -- Spring, aynı controller içindeki eşleşen `@ExceptionHandler`'ı bulup çalıştırır ve onun dönüş değerini yanıt olarak gönderir; `@ResponseStatus(HttpStatus.NOT_FOUND)` de yanıtın durum kodunu belirler.

Global Hata Yönetimi: @RestControllerAdvice

`@ExceptionHandler`'ın controller-seviyesinde kalması bir sorun yaratır: aynı tür hatayı (örn. "kaynak bulunamadı") her controller'da ayrı ayrı ele almak gerekir. `@RestControllerAdvice` (`@ControllerAdvice` + `@ResponseBody`), bunu **tüm** controller'lar için tek bir yerde toplar:

```
import org.springframework.http.HttpStatus;
```

```
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestControllerAdvice;

// @RestControllerAdvice = @ControllerAdvice + @ResponseBody, applied GLOBALLY --
// unlike ExceptionHandlerBasicExample's handler (scoped to one controller), every
// controller in the application is covered by this single class.
@RestControllerAdvice
class GlobalExceptionHandler {

    static class ResourceNotFoundException extends RuntimeException {
        ResourceNotFoundException(String message) {
            super(message);
        }
    }

    static class InvalidRequestException extends RuntimeException {
        InvalidRequestException(String message) {
            super(message);
        }
    }

    @ExceptionHandler(ResourceNotFoundException.class)
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public String handleNotFound(ResourceNotFoundException e) {
        return e.getMessage();
    }

    @ExceptionHandler(InvalidRequestException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public String handleInvalidRequest(InvalidRequestException e) {
        return e.getMessage();
    }

    // A catch-all, LAST-resort handler -- Spring always picks the MOST SPECIFIC
    // matching @ExceptionHandler for a given exception, so this only fires when
    // nothing more specific matches.
    @ExceptionHandler(Exception.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public String handleGeneric(Exception e) {
        return "An unexpected error occurred";
    }
}

class RestControllerAdviceExample {
    public static void main(String[] args) {
        GlobalExceptionHandler advice = new GlobalExceptionHandler();

        System.out.println(advice.handleNotFound(new GlobalExceptionHandler.
ResourceNotFoundException("user 5 not found")));
        // user 5 not found
        System.out.println(advice.handleInvalidRequest(new GlobalExceptionHandler.
InvalidRequestException("email is required")));
        // email is required
        System.out.println(advice.handleGeneric(new RuntimeException("disk full")));
        // An unexpected error occurred
    }
}
```

Bu sınıftaki üç `@ExceptionHandler`, uygulamadaki **her** controller'ı kapsar -- "Controller-Seviyesinde Hata Yakalama: @ExceptionHandler" bölümündeki gibi tek bir controller'a özel değildir. Son handler (`Exception.class`), hiçbir spesifik handler eşleşmediğinde devreye giren bir son çare (catch-all); Spring her zaman en **spesifik** eşleşen handler'ı seçer, bu yüzden `Exception.class` yalnızca gerçekten beklenmeyen durumlarda çalışır.

ProblemDetail: RFC 7807 ile Standart Hata Gövdesi

`@ExceptionHandler`'ın dönüş değeri düz bir `String` de olabilir, ama gerçek bir API'de her takımın kendi hata JSON'ını icat etmesi tutarsızlık yaratır. `ProblemDetail`, Spring'in RFC 7807'yi (standart, kendini açıklayan bir hata biçimi) uygulayan yerleşik sınıfıdır:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

// ProblemDetail is Spring's built-in implementation of RFC 7807 -- a standardized,
// self-describing JSON error shape (Content-Type: application/problem+json), instead
// of every team inventing its own ad hoc error object.
@Controller
class ProductLookupController {

    static class ProductNotFoundException extends RuntimeException {
        ProductNotFoundException(Long id) {
            super("Product not found: " + id);
        }
    }

    @GetMapping("/products/{id}")
    public String getProduct(@PathVariable Long id) {
        throw new ProductNotFoundException(id);
    }

    @ExceptionHandler(ProductNotFoundException.class)
    public ProblemDetail handleNotFound(ProductNotFoundException e) {
        return ProblemDetail.forStatusAndDetail(HttpStatus.NOT_FOUND, e.getMessage());
    }
}

class ProblemDetailBasicExample {
    public static void main(String[] args) {
        ProductLookupController controller = new ProductLookupController();

        try {
            controller.getProduct(7L);
        } catch (ProductLookupController.ProductNotFoundException e) {
            ProblemDetail problem = controller.handleNotFound(e);
            System.out.println(problem.getStatus() + " " + problem.getTitle() + ": " +
problem.getDetail());
            // 404 Not Found: Product not found: 7
        }
    }
}
```

`ProblemDetail.forStatusAndDetail(status, detay)`, durum kodunu, standart bir `title`'ı (durum kodundan otomatik türetilir) ve senin verdiğin `detail`'i taşıyan bir nesne üretir -- gerçek bir Spring uygulamasında bu, `Content-Type: application/problem+json` ile serileştirilir.

Doğrulama Hatalarını ProblemDetail'e Dönüştürmek

`ProblemDetail`, sabit alanların (`status`, `detail`, `title`) ötesinde `setProperty(...)` ile **özel** alanlar da taşıyabilir -- bu, "@Valid'in Perde Arkası: Validator ve ConstraintViolation" bölümündeki `ConstraintViolation` kümesini istemciye okunur bir liste olarak döndürmek için tam ihtiyacımız olan şey:

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;

import java.util.Set;
import java.util.stream.Collectors;

// When real Spring MVC's @Valid fails on a @RequestBody, it throws
// MethodArgumentNotValidException; a @RestControllerAdvice typically catches that
// (reading its BindingResult) instead of a raw ConstraintViolation set like this one
// -- but the conversion logic is identical either way: turn each violation into a
// readable message and attach them to the ProblemDetail as a custom property.
class ProblemDetailValidationExample {

    record CreateProductRequest(@NotBlank String name, @Min(1) int quantity) {
    }

    static <T> ProblemDetail toProblemDetail(Set<ConstraintViolation<T>> violations) {
        ProblemDetail problem = ProblemDetail.forStatusAndDetail(HttpStatus.BAD_REQUEST,
"Validation failed");
        problem.setProperty("errors", violations.stream()
                .map(v -> v.getPropertyPath() + ": " + v.getMessage())
                .collect(Collectors.toList()));
        return problem;
    }

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

        // Only "name" is invalid, so there's exactly one, predictable violation.
        CreateProductRequest invalid = new CreateProductRequest("", 5);
        Set<ConstraintViolation<CreateProductRequest>> violations = validator.validate
(invalid);

        ProblemDetail problem = toProblemDetail(violations);
        System.out.println(problem.getStatus() + " " + problem.getDetail());
        // 400 Validation failed
        System.out.println(problem.getProperties());
        // {errors=[name: must not be blank]}
    }
}
```

`toProblemDetail(...)`, her `ConstraintViolation`'ı `"alan: mesaj"` biçiminde bir metne çevirip `errors` adlı özel bir property olarak ekliyor -- istemci, yalnızca "400 Bad Request" değil, **hangi alanların** neden geçersiz olduğunu da tek bir yanıtta görüyor.

Best Practices

- **Doğrulama kuralını controller'ın içine değil, request nesnesinin (record'un) üzerine yaz** -- "@Valid ile İstek Gövdesini Doğrulamak" bölümünde gördüğümüz gibi, kural annotation olarak tipe bağlı kaldığı sürece o tip nerede kullanılırsa kullanılsın geçerli kalır; controller içindeki elle yazılmış bir `if` bloğu yalnızca o metotta çalışır.
- **İç içe nesnelerde `@Valid`'i unutma** -- "İç İçe Nesnelerde Doğrulama: Cascading ile @Valid" bölümünde gördüğümüz gibi, bu kolayca gözden kaçan bir hatadır: dış nesne doğrulanıyor görünür, ama iç nesnenin kuralları sessizce hiç çalışmaz.
- **Hata yönetimini tek bir `@RestControllerAdvice`'ta topla, her controller'a ayrı `@ExceptionHandler` yazma** -- "Global Hata Yönetimi: @RestControllerAdvice" bölümünde gördüğümüz gibi, bu hem tekrarı önler hem tüm API'de tutarlı bir hata biçimi garanti eder.
- **Kendi hata JSON'unu icat etme, `ProblemDetail` kullan** -- "ProblemDetail: RFC 7807 ile Standart Hata Gövdesi" bölümünde gördüğümüz gibi, bu hem standarttır hem de `setProperty(...)` ile ihtiyacın olan özel alanları (doğrulama hataları gibi) eklemene izin verir.

Yaygın Hatalar

1. `@NotNull`'ın boş string'i de reddettiğini sanmak. "@NotNull, @NotEmpty, @NotBlank: Boşluk Farkları" bölümünde gördüğümüz gibi, `@NotNull` yalnızca `null`'ı reddeder -- kullanıcıdan gelen bir metin alanı için neredeyse her zaman istenen `@NotBlank`'tir.

2. İç içe bir nesnenin alanının otomatik doğrulanacağını varsaymak. "İç İçe Nesnelerde Doğrulama: Cascading ile @Valid" bölümünde gördüğümüz gibi, `@Valid` cascading'i açıkça istemek gerekir -- aksi halde iç nesnenin kuralları sessizce atlanır, hiçbir hata da vermez.

3. `@Valid`'i unutup yalnızca `@RequestBody` yazmak. Bean Validation annotation'ları tipte dursa bile, `@Valid` olmadan hiçbir zaman **tetiklenmezler** -- "@Valid ile İstek Gövdesini Doğrulamak" bölümünde gördüğümüz gibi, kuralın yazılmış olması onun kontrol edildiği anlamına gelmez.

4. `@ExceptionHandler(Exception.class)`'ı en üste yazıp diğer handler'ların hiç çalışmadığını düşünmek. Sıralama önemli değildir -- "Global Hata Yönetimi: @RestControllerAdvice" bölümünde gördüğümüz gibi, Spring her zaman fırlatılan exception'a **en spesifik** eşleşen handler'ı seçer, dosyadaki yazım sırası değil.

5. Hata yanıtında yalnızca durum kodunu dönüp, hangi alanın neden geçersiz olduğunu istemciye hiç söylememek. "Doğrulama Hatalarını ProblemDetail'e Dönüştürmek" bölümünde gördüğümüz gibi, `ProblemDetail`'in `setProperty(...)`'i tam olarak bu bilgiyi taşımak için var -- bir `400` almak, istemcinin sorunu düzeltebilmesi için yeterli değildir.

Özet, Cheat Sheet ve Terimler Sözlüğü

Bean Validation, bir nesnenin alanlarına annotation ile kural yazıp bu kuralları `@Valid` ile otomatik tetikleme standardıdır; exception handling, `@ExceptionHandler` / `@RestControllerAdvice` ile bir hatayı tutarlı bir HTTP yanıtına (idealde bir `ProblemDetail`) çevirme mekanizmasıdır. Öne çıkan noktalar:

- `@NotNull` / `@NotEmpty` / `@NotBlank`: giderek daha sıkı üç "eksik olmasın" kuralı
- `@Size` / `@Min` / `@Max`: uzunluk ve sayısal aralık sınırları (her iki sınır dahil)
- `@Email` / `@Pattern`: biçim doğrulama, `@Pattern` serbest regex ile
- `@Valid`: bir parametreyi/alanı doğrulama tetikleyicisi; iç içe nesnelerde her seviyede ayrıca yazılmalı (cascading)
- `Validator` / `ConstraintViolation`: `@Valid`'in arkasındaki gerçek mekanizma, container olmadan da doğrudan kullanılabilir
- `@ExceptionHandler`: controller-seviyesinde hata yakalama; `@RestControllerAdvice` ile global hale gelir, en spesifik handler kazanır
- `ProblemDetail`: RFC 7807 standart hata gövdesi, `setProperty(...)` ile özel alanlar taşıyabilir

Hızlı referans:

```
record CreateUserRequest(
        @NotBlank @Size(min = 2, max = 50) String name,
        @Email String email) { }

@PostMapping("/users")
public String create(@Valid @RequestBody CreateUserRequest request) {
    return "Created: " + request.name();
}

@RestControllerAdvice
class GlobalExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ProblemDetail handleNotFound(ResourceNotFoundException e) {
        return ProblemDetail.forStatusAndDetail(HttpStatus.NOT_FOUND, e.getMessage());
    }
}
```

Terimler Sözlüğü

Bean Validation — Bir Java nesnesinin alanlarına annotation ile kural yazma ve bu kuralları tek bir `Validator` çağrısıyla kontrol etme standardı (JSR-380).

`@Valid` — Bir parametreyi/alanı, çağrı gerçekleşmeden önce Bean Validation kurallarına göre doğrulamayı tetikleyen annotation.

`ConstraintViolation` — Bir Bean Validation kuralının ihlal edildiğini, hangi alanda ve hangi mesajla ihlal edildiğini taşıyan nesne.

Cascading — İç içe bir nesnenin kendi kısıtlarının da kontrol edilmesi için, o alanın önüne ayrıca `@Valid` yazma gerekliliği.

`@ExceptionHandler` — Bir metodun, belirtilen türden bir exception'ı yakalayıp bir HTTP yanıtına çevirmesini sağlayan annotation.

`@RestControllerAdvice` — `@ExceptionHandler` metotlarını tüm uygulama genelinde (tek bir controller'a değil) geçerli kılan, `@ResponseBody`'yi de içeren annotation.

`ProblemDetail` — RFC 7807'yi uygulayan, Spring'in yerleşik standart hata gövdesi sınıfı.

Ek: Mini Proje — Kullanıcı Kayıt Formu

Bu dersteki doğrulama annotation'larını gerçekçi bir kayıt endpoint'inde bir araya getiriyoruz, ve `@Valid`'in normalde **görünmeyen** iç işleyişini elle simüle ediyoruz:

```
import jakarta.validation.Valid;
import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;

// A realistic registration endpoint, guarded by every annotation family from this
// lesson. See UserRegistrationDemo for how Spring's validation gate actually runs
// before this method is ever called.
@Controller
class UserRegistrationController {

    record RegisterRequest(
            @NotBlank @Size(min = 2, max = 50) String name,
            @Email String email,
            @NotBlank @Size(min = 8) String password) {
    }

    @PostMapping("/register")
    @ResponseBody
    public String register(@Valid @RequestBody RegisterRequest request) {
        return "Registered: " + request.name();
    }
}
```

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.Validator;

import java.util.Set;

// This is what Spring's validation gate does internally, made visible: validate the
// request BEFORE the controller method body ever runs, and never call the method at
// all if any constraint fails.
class UserRegistrationDemo {

    static String dispatch(UserRegistrationController controller, Validator validator,
            UserRegistrationController.RegisterRequest request) {
        Set<ConstraintViolation<UserRegistrationController.RegisterRequest>> violations =
                validator.validate(request);

        if (!violations.isEmpty()) {
            return "400 Bad Request (" + violations.size() + " violation(s))";
        }
        return "200 OK -> " + controller.register(request);
    }

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

        var valid = new UserRegistrationController.RegisterRequest("Ada Lovelace",
"ada@example.com", "s3cretpw!");
        System.out.println(dispatch(controller, validator, valid));
        // 200 OK -> Registered: Ada Lovelace

        // Only the email is malformed, so this triggers exactly one violation.
        var invalidEmail = new UserRegistrationController.RegisterRequest("Ada Lovelace",
"not-an-email", "s3cretpw!");
        System.out.println(dispatch(controller, validator, invalidEmail));
        // 400 Bad Request (1 violation(s))
    }
}
```

`dispatch(...)`, Spring'in gerçekte otomatik yaptığını görünür kılıyor: isteği `register(...)` metoduna ulaşmadan **önce** doğruluyor, herhangi bir ihlal varsa metodu hiç çağırmıyor. Geçersiz istekte yalnızca `email` alanı bozuk olduğu için sonuç deterministik bir tek ihlal.

Ek: Mini Proje — Ürün Kataloğu API'si

Son mini proje, bu dersin iki yarısını (doğrulama ve hata yönetimi) tek bir API diliminde birleştiriyor -- doğrulama girişte, `@RestControllerAdvice` ise controller'ın kendi fırlattığı bir exception'da devreye giriyor:

```
import jakarta.validation.Valid;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import org.springframework.http.HttpStatus;
```

```
import org.springframework.http.ProblemDetail;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
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.RestControllerAdvice;

import java.util.LinkedHashMap;
import java.util.Map;

// A small, complete slice of a real API: Bean Validation guards the input, a
// controller-level exception signals a missing resource, and a SEPARATE
// @RestControllerAdvice turns that exception into a standard ProblemDetail --
// exactly the two mechanisms from this lesson, working together.
@Controller
class ProductCatalogController {

    record CreateProductRequest(@NotBlank String name, @Min(1) int quantity) {
    }

    static class ProductNotFoundException extends RuntimeException {
        ProductNotFoundException(Long id) {
            super("Product not found: " + id);
        }
    }

    private final Map<Long, String> products = new LinkedHashMap<>();
    private long nextId = 1;

    @PostMapping("/products")
    @ResponseBody
    public Long create(@Valid @RequestBody CreateProductRequest request) {
        long id = nextId++;
        products.put(id, request.name());
        return id;
    }

    @GetMapping("/products/{id}")
    @ResponseBody
    public String getOne(@PathVariable Long id) {
        String product = products.get(id);
        if (product == null) {
            throw new ProductNotFoundException(id);
        }
        return product;
    }
}

@RestControllerAdvice
class ProductCatalogExceptionHandler {

    @ExceptionHandler(ProductCatalogController.ProductNotFoundException.class)
    public ProblemDetail handleNotFound(ProductCatalogController.ProductNotFoundException
e) {
        return ProblemDetail.forStatusAndDetail(HttpStatus.NOT_FOUND, e.getMessage());
```

```
    }
}
```

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Validation;
import jakarta.validation.Validator;
import org.springframework.http.ProblemDetail;

import java.util.Set;

// Exercises ProductCatalogApi end to end: a valid create+read, an invalid create
// (rejected before it would ever reach the controller), and a lookup that fails
// inside the controller and is handled by the separate advice class.
class ProductCatalogApiDemo {
    public static void main(String[] args) {
        ProductCatalogController controller = new ProductCatalogController();
        ProductCatalogExceptionHandler advice = new ProductCatalogExceptionHandler();
        Validator validator = Validation.buildDefaultValidatorFactory().getValidator();

        var validRequest = new ProductCatalogController.CreateProductRequest("Keyboard",
10);
        Long id = controller.create(validRequest);
        System.out.println("Created id: " + id);
        // Created id: 1
        System.out.println(controller.getOne(id));
        // Keyboard

        var invalidRequest = new ProductCatalogController.CreateProductRequest("", 10);
        Set<ConstraintViolation<ProductCatalogController.CreateProductRequest>> violations =
                validator.validate(invalidRequest);
        System.out.println("Violations: " + violations.size());
        // Violations: 1

        try {
            controller.getOne(99L);
        } catch (ProductCatalogController.ProductNotFoundException e) {
            ProblemDetail problem = advice.handleNotFound(e);
            System.out.println(problem.getStatus() + " " + problem.getDetail());
            // 404 Product not found: 99
        }
    }
}
```

`ProductCatalogController` ve `ProductCatalogExceptionHandler` **ayrı** sınıflar -- "Global Hata Yönetimi: @RestControllerAdvice" bölümünde vurguladığımız ayrımın gerçek bir örneği: doğrulama, controller'ın kendi metoduna (`@Valid` ile) bağlı kalırken, hata dönüşümü tamamen ayrı, paylaşılan bir advice sınıfında yaşıyor. `ProductCatalogApiDemo`, geçerli bir create+get, geçersiz bir create (yalnızca ihlal sayısını yazdırarak) ve bulunamayan bir get (advice'ın `ProblemDetail`'i elle çağırarak) olmak üzere üç yolu da çalıştırıyor.

> `ProductCatalogController`'ın `create(...)` metodu gerçek bir Spring ortamında olsaydı, `@Valid` başarısız olduğunda `MethodArgumentNotValidException` fırlar ve bu istisna da bir `@RestControllerAdvice`'a eklenecek ayrı bir `@ExceptionHandler(MethodArgumentNotValidException.class)` ile yakalanırdı -- bu mini projede doğrulamayı `ProductCatalogApiDemo` içinde elle çağırıyoruz, çünkü bir gerçek `DispatcherServlet` olmadan bu otomatik tetiklenme gerçekleşmez.