Validation & Exception Handling

Bean Validation (@NotBlank, @Size, @Email, @Pattern) with @Valid, cascading for nested objects, @ExceptionHandler, @RestControllerAdvice, and standardized error handling with RFC 7807 ProblemDetail.

Intermediate 45 min

Last updated: August 11, 2026
Read online: https://www.learnforgex.com/en/topics/validation-exception-handling

---

Validation & Exception Handling

In Request and Response Handling we saw that Jackson only validates `@RequestBody` at the **format** level -- is the JSON well-formed, do the types line up. Checking business rules (a non-blank name, a positive quantity, a valid email) was left up to you. This lesson closes that gap with two mechanisms: **Bean Validation** (`@Valid` and friends), the standard way to reject a request **before** it ever reaches the controller; and **exception handling** (`@ExceptionHandler` , `@RestControllerAdvice` , `ProblemDetail`), the way to turn an error into a consistent, standard response for the client.

What Are Validation & Exception Handling?

Bean Validation is the standard for writing rules as annotations on a Java object's fields and checking all of them with a single call (JSR-380, the `jakarta.validation` package). Exception handling is the mechanism for turning an error raised in a controller method (or during validation) into a centralized, consistent HTTP response, without scattered `try` / `catch` blocks:

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

@PostMapping("/users")
public String create(@Valid @RequestBody CreateUserRequest request) {
    // reached only if name is non-blank and email is valid
    return "Created: " + request.name();
}
```

Why Does It Exist?

Writing a validation rule by hand at the top of every controller method (`if (name == null || name.isBlank()) throw ...`) is both repetitive and easy to forget -- a field gets added, and the check gets forgotten. Bean Validation moves the rule onto the **data type itself**: wherever `CreateUserRequest` is used, the `@NotBlank` rule comes along with it. Likewise, building an error body by hand in every `catch` block produces inconsistent results (plain text in one place, JSON in another, nothing at all somewhere else); `@ExceptionHandler` / `@RestControllerAdvice` gathers that conversion into a single place.

History

Bean Validation was standardized as JSR-303 in 2009 with Java EE 6; JSR-380 (Bean Validation 2.0), which followed the move to the `jakarta.validation` package name (as Java EE became Jakarta EE), added annotations that are now familiar, like `@NotEmpty` / `@NotBlank`. Hibernate Validator is the reference implementation of this standard -- the `spring-boot-starter-validation` dependency is exactly what brings it (and Spring's `@Valid` integration) into the project. `@ExceptionHandler` arrived in Spring 3.0; `@ControllerAdvice` (its global counterpart) arrived in Spring 3.2, and `ProblemDetail` (RFC 7807 support) was added in Spring 6 / Spring Boot 3.

@NotNull, @NotEmpty, @NotBlank: The Blank-Value Differences

All three annotations say "don't leave this value missing," but each checks a different threshold:

```
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` only requires the value not to be `null` -- an empty string (`""`) passes. `@NotEmpty` rejects both `null` and an empty string -- but a whitespace-only string (`" "`) still passes. `@NotBlank` rejects all three -- in practice, this is what's usually wanted for a text field coming from a user.

@Size, @Min, @Max: Numeric and Length Bounds

`@Size` checks a String/Collection/array's **length**; `@Min` / `@Max` check a numeric value's **range** -- both bounds are 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)` accepts exactly 3 or exactly 50 characters and rejects 2 or 51; `@Min(1) @Max(1000)` likewise accepts 1 and 1000.

@Email and @Pattern: Format Validation

`@Email` checks for a syntactically valid email format; `@Pattern` checks against any regular expression you provide -- making it the most flexible annotation, since you write the rule yourself:

```
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
    }
}
```

Notice the `message` attribute given to `@Pattern`: most annotations' default message ("must match ...") means nothing to a user; for a free-form rule like `@Pattern`, writing a readable `message` is nearly mandatory.

Validating a Request Body with @Valid

Unlike `@RequestParam` / `@PathVariable`, Bean Validation rules don't run on their own -- putting `@Valid` in front of a parameter tells Spring "validate this object before the controller method runs":

```
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();
    }
}
```

If even one of the `@NotBlank` / `@Email` constraints fails, `create(...)`'s body **never runs at all** -- Spring rejects the request before the method is called. We'll see how this validation actually works in "@Valid's Machinery: Validator and ConstraintViolation".

@Valid's Machinery: Validator and ConstraintViolation

`@Valid` doesn't invent its own validation engine -- it calls a `jakarta.validation.Validator` (an interface that's entirely independent of any container, and usable directly on its own) and interprets the result for you:

```
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(object)` returns a `Set` containing one `ConstraintViolation` for each violated rule; an empty set means the object is valid. When `@Valid @RequestBody` fails, Spring doesn't hand you this same result directly -- it wraps it in a `MethodArgumentNotValidException` (carrying a `BindingResult`), which can be caught with the `@ExceptionHandler` we'll see in the next two sections.

Validating Nested Objects: Cascading with @Valid

Bean Validation does **not** check a nested object's fields by default -- if you want the nested object validated too, you need to put `@Valid` on that field as well (this is called cascading):

```
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`'s `address` field has no `@Valid` in front of it -- `Address`'s own `@NotBlank` rule never runs, and the result is always 0 violations. `ShippingRequestWithCascade` turns on that cascading with `@Valid Address address`, and the nested object's violations get added to the set as well.

Catching Errors at the Controller Level: @ExceptionHandler

`@ExceptionHandler`, placed on a method inside a controller, catches an exception of the given type thrown by **any** handler method in that **same** controller:

```
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();
    }
}
```

When `getProduct(...)` throws a `ProductNotFoundException`, it isn't seen by the calling code -- Spring finds and runs the matching `@ExceptionHandler` in the same controller and sends its return value as the response; `@ResponseStatus(HttpStatus.NOT_FOUND)` also sets the response's status code.

Global Error Handling: @RestControllerAdvice

Keeping `@ExceptionHandler` at the controller level creates a problem: the same kind of error (e.g. "resource not found") would need to be handled separately in every controller. `@RestControllerAdvice` (`@ControllerAdvice` + `@ResponseBody`) gathers this into a single place for **every** controller:

```
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
```

```
    }
}
```

The three `@ExceptionHandler`s in this class cover **every** controller in the application -- unlike "Catching Errors at the Controller Level: @ExceptionHandler", they aren't scoped to a single controller. The last handler (`Exception.class`) is a last-resort catch-all that only runs when nothing more specific matches; Spring always picks the **most specific** matching handler, so `Exception.class` only fires for genuinely unexpected cases.

ProblemDetail: A Standard Error Body with RFC 7807

An `@ExceptionHandler`'s return value could just be a plain `String`, but in a real API, every team inventing its own error JSON creates inconsistency. `ProblemDetail` is Spring's built-in implementation of RFC 7807 -- a standardized, self-describing error format:

```
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, detail)` produces an object carrying the status code, a standard `title` (derived automatically from the status code), and the `detail` you provide -- in a real Spring application this gets serialized with `Content-Type: application/problem+json`.

Turning Validation Errors into a ProblemDetail

Beyond its fixed fields (`status`, `detail`, `title`), `ProblemDetail` can also carry **custom** fields via `setProperty(...)` -- exactly what we need to turn the `ConstraintViolation` set

from "@Valid's Machinery: Validator and ConstraintViolation" into a readable list for the client:

```
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(...)` turns each `ConstraintViolation` into a `"field: message"` string and adds them as a custom `errors` property -- the client sees not just "400 Bad Request," but **which** fields were invalid and why, in a single response.

Best Practices

- **Write the validation rule on the request object (the record) itself, not inside the controller** -- as we saw in "Validating a Request Body with @Valid", as long as the rule stays as an annotation on the type, it stays valid wherever that type is used; a hand-written `if` block inside a controller only ever applies to that one method.
- **Don't forget `@Valid` on nested objects** -- as we saw in "Validating Nested Objects: Cascading with @Valid", this is an easy mistake to miss: the outer object looks validated, but the inner object's rules silently never run.
- **Gather error handling into a single `@RestControllerAdvice`, don't write a separate `@ExceptionHandler` per controller** -- as we saw in "Global Error Handling: @RestControllerAdvice", this avoids repetition and guarantees a consistent error format across the whole API.
- **Don't invent your own error JSON -- use `ProblemDetail`** -- as we saw in "ProblemDetail: A Standard Error Body with RFC 7807", it's both standardized and lets you attach the custom fields you need (like validation errors) via `setProperty(...)`.

Common Mistakes

1. Assuming `@NotNull` also rejects an empty string. As we saw in "@NotNull, @NotEmpty, @NotBlank: The Blank-Value Differences", `@NotNull` only rejects `null` -- for a text field coming from a user, `@NotBlank` is almost always what's actually wanted.

2. Assuming a nested object's field gets validated automatically. As we saw in "Validating Nested Objects: Cascading with @Valid", cascading has to be requested explicitly with `@Valid` -- otherwise the nested object's rules are silently skipped, with no error raised.

3. Forgetting `@Valid` and writing only `@RequestBody`. Bean Validation annotations sitting on a type are never **triggered** without `@Valid` -- as we saw in "Validating a Request Body with @Valid", a rule being written doesn't mean it's being checked.

4. Putting `@ExceptionHandler(Exception.class)` first and assuming the other handlers never run. Order doesn't matter -- as we saw in "Global Error Handling: @RestControllerAdvice", Spring always picks the **most specific** handler matching the thrown exception, not the order it appears in the file.

5. Returning only a status code in the error response and never telling the client which field was invalid and why. As we saw in "Turning Validation Errors into a ProblemDetail", `ProblemDetail`'s `setProperty(...)` exists exactly to carry that information -- getting a `400` isn't enough for the client to actually fix the problem.

Summary, Cheat Sheet, and Glossary

Bean Validation is the standard for writing rules as annotations on an object's fields and triggering them automatically with `@Valid`; exception handling is the mechanism, via `@ExceptionHandler` / `@RestControllerAdvice`, for turning an error into a consistent HTTP response (ideally a `ProblemDetail`). Key points:

- `@NotNull` / `@NotEmpty` / `@NotBlank`: three increasingly strict "don't leave this missing" rules
- `@Size` / `@Min` / `@Max`: length and numeric range bounds (both bounds inclusive)
- `@Email` / `@Pattern`: format validation, with `@Pattern` for a free-form regex
- `@Valid`: the trigger for validating a parameter/field; must be written at each level for nested objects (cascading)
- `Validator` / `ConstraintViolation`: the actual mechanism behind `@Valid`, usable directly without a container
- `@ExceptionHandler`: controller-level error catching; `@RestControllerAdvice` makes it global, and the most specific handler wins
- `ProblemDetail`: the RFC 7807 standard error body, able to carry custom fields via `setProperty(...)`

Quick reference:

```
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());
    }
}
```

Glossary

Bean Validation — The standard for writing rules as annotations on a Java object's fields and checking all of them with a single `Validator` call (JSR-380).

`@Valid` — The annotation that triggers Bean Validation of a parameter/field before the call is made.

`ConstraintViolation` — The object carrying the fact that a Bean Validation rule was violated, along with which field and what message.

Cascading — The requirement of putting `@Valid` on a nested object's own field in order for its own constraints to also be checked.

`@ExceptionHandler` — The annotation that lets a method catch an exception of the given type and turn it into an HTTP response.

`@RestControllerAdvice` — The annotation that makes `@ExceptionHandler` methods apply application-wide (rather than to a single controller), also including `@ResponseBody`.

`ProblemDetail` — Spring's built-in standard error body class implementing RFC 7807.

Appendix: Mini Project — User Registration Form

We bring this lesson's validation annotations together in a realistic registration endpoint, and manually simulate `@Valid`'s normally **invisible** inner workings:

```
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(...)` makes what Spring actually does automatically visible: it validates the request **before** it ever reaches `register(...)`, and never calls the method at all if there's any violation. In the invalid request, only the `email` field is broken, so the result is a deterministic single violation.

Appendix: Mini Project — Product Catalog API

The final mini project brings both halves of this lesson (validation and error handling) together in a single API slice -- validation on the way in, `@RestControllerAdvice` kicking in when the controller itself throws an exception:

```
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` and `ProductCatalogExceptionHandler` are **separate** classes -- a real example of the separation we emphasized in "Global Error Handling: @RestControllerAdvice": validation stays tied to the controller's own method (via `@Valid`), while error conversion lives entirely in a separate, shared advice class.
`ProductCatalogApiDemo` exercises all three paths: a valid create+get, an invalid create (printing only the violation count), and a not-found get (calling the advice's `ProblemDetail` manually).

If `ProductCatalogController`'s `create(...)` method were running in a real Spring environment, a failed `@Valid` would throw a `MethodArgumentNotValidException`, which would be caught by a separate `@ExceptionHandler(MethodArgumentNotValidException.class)` added to a `@RestControllerAdvice` -- in this mini project we call the validation by hand inside `ProductCatalogApiDemo`, because without a real `DispatcherServlet` that automatic triggering never happens.