Exception Handling

Builds on Spring MVC's "Validation & Exception Handling" and this category's "Java Bean Validation" -- what MethodArgumentNotValidException actually carries, turning validation failures into a ProblemDetail (including cross-field failures), custom ProblemDetail properties, mapping domain exceptions to the right HTTP status code (400/404/409/422/500), centralizing with ResponseEntityExceptionHandler, and safe error responses that don't leak internal details. The 2nd and final lesson in the Advanced Spring series.

Advanced 35 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/exception-handling

---

"Validation & Exception Handling," in Spring MVC, already covered `@ExceptionHandler`, `@RestControllerAdvice`, and a first look at RFC 7807 `ProblemDetail`. "Java Bean Validation," earlier in this category, added a much richer set of constraints — including a custom, cross-field one — that can now fail in more varied ways. This lesson connects the two: what actually happens when validation fails, how to design status codes and response bodies for a real REST API's failures in general, and how to centralize all of it without leaking anything a client shouldn't see.

Why Not Every Error Is a Generic 500

Returning `500 Internal Server Error` for every failure is easy to write and almost useless to a client — it can't tell "you sent bad data" apart from "our database is down" apart from "this resource doesn't exist," three situations that call for three completely different client responses. A well-designed API distinguishes CLIENT errors (bad input, a business rule violation, a missing resource) — which the client can potentially fix and retry — from genuine SERVER errors, which it can't. Everything in this lesson is about making that distinction concrete.

MethodArgumentNotValidException: What @Valid Actually Throws

"Validation & Exception Handling" showed `@Valid` triggering validation, without naming what actually happens when it fails on a `@RequestBody`: Spring throws a `MethodArgumentNotValidException`, before your controller method's body ever runs.

```
import org.springframework.core.MethodParameter;
import org.springframework.validation.BeanPropertyBindingResult;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;

// This is exactly what a failed @Valid produces for real, behind the
// scenes, when a @RequestBody's constraints don't pass: Spring throws a
// MethodArgumentNotValidException BEFORE the controller method ever runs,
// carrying a BindingResult with one FieldError per failed constraint.
class MethodArgumentNotValidExceptionExample {

    record CreateProductRequest(String name, int quantity) {
    }

    // A stand-in for the real controller method, used only so a real
    // MethodParameter can be constructed below -- in an actual failing
    // request, Spring builds this exception itself; it's built by hand
    // here just to show precisely what it contains.
    void create(CreateProductRequest request) {
    }

    public static void main(String[] args) throws NoSuchMethodException {
        var target = new CreateProductRequest("", -1);
        var bindingResult = new BeanPropertyBindingResult(target, "createProductRequest");
        bindingResult.addError(new FieldError("createProductRequest", "name", "must not be
blank"));
        bindingResult.addError(new FieldError("createProductRequest", "quantity", "must be
positive"));

        var method = MethodArgumentNotValidExceptionExample.class
                .getDeclaredMethod("create", CreateProductRequest.class);
        var parameter = new MethodParameter(method, 0);

        var exception = new MethodArgumentNotValidException(parameter, bindingResult);

        // getBindingResult().getFieldErrors() is how a handler reads each
        // individual failure -- exactly what a @RestControllerAdvice
        // method receiving this exception type would call.
        for (FieldError error : exception.getBindingResult().getFieldErrors()) {
            System.out.println(error.getField() + ": " + error.getDefaultMessage());
        }
        // name: must not be blank
        // quantity: must be positive
    }
}
```

The exception carries a `BindingResult` — the exact same type Spring MVC uses for traditional form binding — with one `FieldError` per failed constraint, each naming its field and the constraint's message. `exception.getBindingResult().getFieldErrors()` is how a handler reads every individual failure at once, rather than getting only the first one.

Turning Validation Failures Into a ProblemDetail

A `@RestControllerAdvice` method that catches `MethodArgumentNotValidException` converts that `BindingResult` into a response — and needs to read more than just per-field errors.

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.validation.ObjectError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.List;
import java.util.stream.Collectors;

// Where MethodArgumentNotValidExceptionExample showed what the exception
// CONTAINS, this shows the handler that actually turns it into a response
// -- reading BOTH kinds of errors it can carry: per-field errors (from a
// failed @NotBlank, @Positive, ...) and object-level errors (from a
// class-level custom constraint like "Java Bean Validation"'s
// @ValidDateRange, which isn't about any single field).
@RestControllerAdvice
class ValidationExceptionAdvice {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ProblemDetail handleValidation(MethodArgumentNotValidException ex) {
        ProblemDetail problem = ProblemDetail.forStatusAndDetail(
                HttpStatus.BAD_REQUEST, "One or more fields failed validation");

        List<String> fieldErrors = ex.getBindingResult().getFieldErrors().stream()
                .map(e -> e.getField() + ": " + e.getDefaultMessage())
                .collect(Collectors.toList());

        List<String> objectErrors = ex.getBindingResult().getGlobalErrors().stream()
                .map(ObjectError::getDefaultMessage) // class-level errors have no single
field
                .collect(Collectors.toList());

        problem.setProperty("fieldErrors", fieldErrors);
        if (!objectErrors.isEmpty()) {
            problem.setProperty("objectErrors", objectErrors); // e.g. a failed
@ValidDateRange
        }
        return problem;
    }
}
```

`getFieldErrors()` covers ordinary single-field failures (`@NotBlank` , `@Positive` , and the rest). `getGlobalErrors()` covers something "Validation & Exception Handling" never needed: a class-level custom constraint like "Java Bean Validation"'s `@ValidDateRange` , which isn't

attached to any single field, so it surfaces as an `ObjectError` instead of a `FieldError`. A handler that only reads `getFieldErrors()` would silently drop a failed cross-field rule out of its response entirely.

Custom ProblemDetail Properties

Spring MVC's lesson attached a single `"errors"` property to a `ProblemDetail`. In practice, a real API's error body usually needs more than that.

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

import java.time.Instant;

// Spring MVC's own lesson used setProperty(...) once, for a single
// "errors" list. ProblemDetail accepts as many custom properties as an
// API needs -- here, a machine-readable error code, a timestamp, and a
// trace id a client can quote back when asking for support, all attached
// to the SAME RFC 7807 body alongside its standard fields.
class CustomProblemDetailPropertiesExample {

    static ProblemDetail insufficientStock(String productId, int requested, int available) {
        ProblemDetail problem = ProblemDetail.forStatusAndDetail(
                HttpStatus.UNPROCESSABLE_ENTITY,
                "Cannot fulfill the requested quantity");

        problem.setType(java.net.URI.create("https://api.example.com/errors/insufficient-
stock"));
        problem.setTitle("Insufficient Stock");
        problem.setProperty("errorCode", "INSUFFICIENT_STOCK");
        problem.setProperty("productId", productId);
        problem.setProperty("requestedQuantity", requested);
        problem.setProperty("availableQuantity", available);
        problem.setProperty("timestamp", Instant.now());
        return problem;
    }

    public static void main(String[] args) {
        ProblemDetail problem = insufficientStock("SKU-42", 10, 3);

        System.out.println(problem.getStatus() + " " + problem.getTitle());
        System.out.println(problem.getProperties());
        // {errorCode=INSUFFICIENT_STOCK, productId=SKU-42, requestedQuantity=10,
        //  availableQuantity=3, timestamp=...}
    }
}
```

`setType(...)`, `setTitle(...)`, and repeated `setProperty(...)` calls build out a `ProblemDetail` with a machine-readable `errorCode`, the specific resource involved, and a `timestamp` — all still valid RFC 7807, since `ProblemDetail` is designed around exactly this kind of extension. A client can branch on `errorCode` reliably, in a way it never safely could on a human-readable message string.

Choosing the Right Status Code for a Domain Exception

Different business failures deserve different status codes — picking the right one communicates something specific, instead of forcing every client to inspect a response body just to know what category of problem occurred.

```
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;

// Three business exceptions, three DIFFERENT status codes -- picking the
// right one communicates something specific to the client, not just
// "something went wrong":
// 404 Not Found            -- the resource being asked about doesn't exist
// 409 Conflict              -- the request conflicts with the resource's current state
// 422 Unprocessable Entity  -- the request was well-formed and understood,
//                              but violates a business rule (as opposed to
//                              400, which means the request itself was malformed)
@RestControllerAdvice
class DomainExceptionAdvice {

    static class OrderNotFoundException extends RuntimeException {
        OrderNotFoundException(String orderId) {
            super("Order not found: " + orderId);
        }
    }

    static class DuplicateOrderException extends RuntimeException {
        DuplicateOrderException(String orderId) {
            super("Order already exists: " + orderId);
        }
    }

    static class InsufficientStockException extends RuntimeException {
        InsufficientStockException(String productId) {
            super("Not enough stock for: " + productId);
        }
    }

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

    @ExceptionHandler(DuplicateOrderException.class)
    @ResponseStatus(HttpStatus.CONFLICT)
    public String handleDuplicate(DuplicateOrderException e) {
        return e.getMessage();
    }

    @ExceptionHandler(InsufficientStockException.class)
    @ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY)
    public String handleInsufficientStock(InsufficientStockException e) {
        return e.getMessage();
    }
}
```

`404 Not Found` means the resource being asked about doesn't exist. `409 Conflict` means the request conflicts with the resource's current state (a duplicate, in this example). `422 Unprocessable Entity` means the request was well-formed and understood, but violates a

business rule — the key distinction from `400 Bad Request`, which means the request itself was malformed or failed validation, as covered in the sections above.

> A quick rule of thumb: if the request itself is broken (missing fields, wrong types), that's `400`. If the request is well-formed but the THING it's asking for doesn't exist, that's `404`. If it's well-formed but conflicts with existing state, that's `409`. If it's well-formed and the resource exists, but a business rule still rejects it, that's `422`.

Centralizing Framework Exceptions with ResponseEntityExceptionHandler

`@RestControllerAdvice` with individual `@ExceptionHandler` methods, from Spring MVC's lesson, centralizes handling for an application's OWN exceptions. `ResponseEntityExceptionHandler` does the equivalent for exceptions Spring MVC ITSELF throws.

```
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatusCode;
import org.springframework.http.ProblemDetail;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import org.springframework.web.context.request.WebRequest;
import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

// ResponseEntityExceptionHandler is Spring MVC's OWN base class for
// handling the framework's built-in exceptions (MethodArgumentNotValidException,
// HttpMessageNotReadableException, and many others) -- extending it and
// overriding one method CUSTOMIZES that specific case, while every other
// exception it already knows how to handle keeps its default behavior for
// free. This centralizes handling for framework-level exceptions the way
// a @RestControllerAdvice with individual @ExceptionHandler methods
// centralizes handling for an application's OWN exceptions.
@RestControllerAdvice
class GlobalMvcExceptionHandler extends ResponseEntityExceptionHandler {

    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex, HttpHeaders headers,
            HttpStatusCode status, WebRequest request) {

        ProblemDetail problem = ProblemDetail.forStatusAndDetail(status, "Validation
failed");
        problem.setProperty("fieldErrors", ex.getBindingResult().getFieldErrors());

        return ResponseEntity.status(status).headers(headers).body(problem);
        // Every OTHER exception ResponseEntityExceptionHandler already
        // understands -- a malformed JSON body, an unsupported media type,
        // a missing request parameter -- is still handled by its default
        // logic, with no code written here for any of them.
    }
}
```

Extending it and overriding one method — `handleMethodArgumentNotValid(...)` here — customizes exactly that one case, while every other framework exception it already knows how to handle (a malformed JSON body, an unsupported media type, a missing parameter, and many more) keeps its sensible default behavior automatically, with no code written for any of them.

Keeping Error Responses Safe

An exception's message or stack trace often contains information that was never meant to leave the server — a database hostname, an internal file path, a library version.

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.logging.Level;
import java.util.logging.Logger;

@RestControllerAdvice
class SafeErrorHandlingAdvice {

    private static final Logger log = Logger.getLogger(SafeErrorHandlingAdvice.class.
getName());

    // UNSAFE (shown only as a comment -- never write this): returning
    // e.getMessage() or a stack trace directly to the client can leak a
    // database column name, an internal file path, or a library version --
    // information an attacker can use, and information a client never
    // needed in the first place.
    //
    // @ExceptionHandler(Exception.class)
    // public ProblemDetail handleUnsafe(Exception e) {
    //     return ProblemDetail.forStatusAndDetail(HttpStatus.INTERNAL_SERVER_ERROR, e.
toString());
    // }

    // SAFE: the FULL exception is logged where only the team can see it --
    // stack trace, message, everything -- while the client receives a
    // generic, constant message that reveals nothing about the failure's
    // internal cause.
    @ExceptionHandler(Exception.class)
    public ProblemDetail handleUnexpected(Exception e) {
        log.log(Level.SEVERE, "Unhandled exception", e);
        return ProblemDetail.forStatusAndDetail(
                HttpStatus.INTERNAL_SERVER_ERROR,
                "An unexpected error occurred. Please try again later.");
    }

    public static void main(String[] args) {
        SafeErrorHandlingAdvice advice = new SafeErrorHandlingAdvice();
        ProblemDetail problem = advice.handleUnexpected(
                new RuntimeException("Connection to db-primary-7.internal:5432 refused"));

        System.out.println(problem.getDetail());
        // An unexpected error occurred. Please try again later.
        // -- the real message, with its internal hostname, only ever reached the log.
    }
}
```

The unsafe version — returning `e.toString()` or a message straight from the exception — hands exactly that information to whoever sent the request, attacker or not. The safe version logs the FULL exception where only the team can see it, and returns a generic, constant message to the client — the two audiences (an engineer debugging a log, a client reading a response) get exactly the information each one should have, and no more.

A Practical, End-to-End Example

Combining everything above into one realistic endpoint shows how these pieces fit together in practice.

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ProblemDetail;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.stream.Collectors;

// A single, realistic REST API endpoint plus the centralized advice that
// handles everything it can fail with -- validation errors, a specific
// business exception, and an unanticipated failure -- combining every
// technique covered in this lesson into one practical example.
class OrderApi {

    @RestController
    static class OrderController {

        record PlaceOrderRequest(String productId, int quantity) {
        }

        static class OutOfStockException extends RuntimeException {
            OutOfStockException(String productId) {
                super("Product is out of stock: " + productId);
            }
        }

        @PostMapping("/orders")
        public String placeOrder(@jakarta.validation.Valid @RequestBody PlaceOrderRequest
request) {
            if (request.quantity() > 100) {
                throw new OutOfStockException(request.productId());
            }
            return "Order placed for " + request.productId();
        }
    }

    @RestControllerAdvice
    static class OrderExceptionAdvice {

        private static final Logger log = Logger.getLogger(OrderExceptionAdvice.class.
getName());

        // 1. Validation failures -- 400, with per-field detail.
        @ExceptionHandler(MethodArgumentNotValidException.class)
        public ProblemDetail handleValidation(MethodArgumentNotValidException e) {
            ProblemDetail problem = ProblemDetail.forStatusAndDetail(
                    HttpStatus.BAD_REQUEST, "Validation failed");
```

```
            problem.setProperty("fieldErrors", e.getBindingResult().getFieldErrors().
stream()
                    .map(fe -> fe.getField() + ": " + fe.getDefaultMessage())
                    .collect(Collectors.toList()));
            return problem;
        }

        // 2. A specific business rule -- 422, since the request was
        // well-formed but conflicts with real-world stock levels.
        @ExceptionHandler(OrderController.OutOfStockException.class)
        public ProblemDetail handleOutOfStock(OrderController.OutOfStockException e) {
            ProblemDetail problem = ProblemDetail.forStatusAndDetail(
                    HttpStatus.UNPROCESSABLE_ENTITY, e.getMessage());
            problem.setProperty("errorCode", "OUT_OF_STOCK");
            return problem;
        }

        // 3. Everything else -- 500, logged in full, revealed to the
        // client only as a generic, safe message.
        @ExceptionHandler(Exception.class)
        public ProblemDetail handleUnexpected(Exception e) {
            log.log(Level.SEVERE, "Unhandled exception in OrderController", e);
            return ProblemDetail.forStatusAndDetail(
                    HttpStatus.INTERNAL_SERVER_ERROR,
                    "An unexpected error occurred. Please try again later.");
        }
    }
}
```

`OrderController.placeOrder(...)` can fail three distinct ways, and `OrderExceptionAdvice` handles each with the technique it actually calls for: a `MethodArgumentNotValidException` becomes a `400` with per-field detail, an `OutOfStockException` becomes a `422` with a machine-readable `errorCode`, and anything else is logged in full and reduced to a safe, generic `500` — one centralized class covering an entire controller's realistic failure modes.

Best Practices

- Pick a status code by what actually went wrong (malformed request, missing resource, conflicting state, rejected business rule, genuine server failure), not by habit or convenience.
- Read both `getFieldErrors()` and `getGlobalErrors()` from a `BindingResult` — a class-level custom constraint's failure only shows up in the second one.
- Attach a machine-readable `errorCode` as a custom `ProblemDetail` property whenever a client might need to branch on the specific failure, not just display a message.
- Log the full exception internally and return a generic message externally for anything unanticipated — never let `e.getMessage()` or a stack trace reach a client directly.

Common Mistakes

- Returning `500` for a validation failure or a business rule rejection — both are client-caused, and both deserve a `4xx` status the client can act on.
- Reading only `getFieldErrors()` and missing a class-level constraint's failure entirely, since it only appears in `getGlobalErrors()`.
- Using `409 Conflict` and `422 Unprocessable Entity` interchangeably — a conflict is about existing state; an unprocessable entity is about a business rule, independent of any conflict.
- Exposing an exception's raw message or stack trace in a response body, leaking implementation details a client (or attacker) was never meant to see.

Summary, Cheat Sheet, and Glossary

Summary

- `@Valid` failing on a `@RequestBody` throws `MethodArgumentNotValidException`, carrying a `BindingResult` with per-field and class-level errors.
- `getFieldErrors()` covers single-field failures; `getGlobalErrors()` covers class-level custom constraints like a cross-field rule.
- `ProblemDetail` supports as many custom properties as an API needs — an error code, a resource id, a timestamp — beyond a single error list.
- Different domain failures deserve different status codes: `400` malformed, `404` missing, `409` conflicting state, `422` rejected business rule, `500` genuine server failure.
- `ResponseEntityExceptionHandler` centralizes handling for Spring MVC's own exceptions, the way `@RestControllerAdvice` centralizes handling for an application's own.
- A safe error response logs the full exception internally and returns only a generic message externally.

Cheat Sheet

```
// Reading both kinds of validation errors
ex.getBindingResult().getFieldErrors();   // per-field
ex.getBindingResult().getGlobalErrors();  // class-level custom constraints

// Custom ProblemDetail properties
ProblemDetail problem = ProblemDetail.forStatusAndDetail(status, detail);
problem.setProperty("errorCode", "OUT_OF_STOCK");

// Status codes for domain exceptions
@ExceptionHandler(NotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
@ExceptionHandler(ConflictException.class)
@ResponseStatus(HttpStatus.CONFLICT) // 409
@ExceptionHandler(BusinessRuleException.class)
@ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY) // 422

// Centralizing framework exceptions
class GlobalMvcExceptionHandler extends ResponseEntityExceptionHandler {
    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(...) { ... }
}

// Safe fallback
@ExceptionHandler(Exception.class)
public ProblemDetail handleUnexpected(Exception e) {
    log.error("Unhandled exception", e);
    return ProblemDetail.forStatusAndDetail(HttpStatus.INTERNAL_SERVER_ERROR, "An
unexpected error occurred.");
}
```

Glossary

- **MethodArgumentNotValidException**: the exception Spring MVC throws when `@Valid` fails on a `@RequestBody`, carrying a `BindingResult`.
- **Field error vs. object error**: a `FieldError` reports a single failed field; an `ObjectError` (from `getGlobalErrors()`) reports a class-level failure not tied to one field.
- **422 Unprocessable Entity**: the status for a well-formed, understood request that still violates a business rule.
- **ResponseEntityExceptionHandler**: Spring MVC's base class for centrally handling the framework's own built-in exceptions.
- **Safe error response**: a response that logs full failure detail internally but reveals only a generic message externally.