Exception Handling

Spring MVC'nin "Validation & Exception Handling" ve bu kategorinin "Java Bean Validation" derslerinin üzerine inşa edilir -- MethodArgumentNotValidException'ın gerçekte ne taşıdığı, validasyon hatalarını ProblemDetail'e dönüştürmek (çapraz-alan hatalar dahil), custom ProblemDetail özellikleri, domain exception'ları doğru HTTP durum koduna (400/404/409/422/500) eşlemek, ResponseEntityExceptionHandler ile merkezileştirme, ve internal detay sızdırmayan güvenli hata yanıtları. Advanced Spring serisinin 2. ve son dersi.

İleri 35 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/exception-handling

---

Spring MVC'deki "Validation & Exception Handling", `@ExceptionHandler`, `@RestControllerAdvice`, ve RFC 7807 `ProblemDetail`'e ilk bakışı zaten işledi. Bu kategorinin daha önceki dersi "Java Bean Validation", artık daha çeşitli şekillerde başarısız olabilen -- custom, çapraz alanlı bir tanesi dahil -- çok daha zengin bir kısıt kümesi ekledi. Bu ders ikisini birbirine bağlıyor: validasyon başarısız olduğunda gerçekte ne olur, gerçek bir REST API'nin hataları için genel olarak durum kodlarını ve response gövdelerini nasıl tasarlarsın, ve bir client'ın görmemesi gereken hiçbir şeyi sızdırmadan hepsini nasıl merkezileştirirsin.

Neden Her Hata Genel Bir 500 Olmamalı

Her hata için `500 Internal Server Error` döndürmek yazması kolay ama client için neredeyse işe yaramazdır -- "geçersiz veri gönderdin"i "veritabanımız çöktü"den, o da "bu kaynak yok"tan ayırt edemez, üç tamamen farklı client tepkisi gerektiren üç durum. İyi tasarlanmış bir API, CLIENT hatalarını (geçersiz girdi, bir iş kuralı ihlali, eksik bir kaynak) -- client'ın potansiyel olarak düzeltip yeniden deneyebileceği -- gerçek SUNUCU hatalarından -- client'ın düzeltemeyeceği -- ayırt eder. Bu dersteki her şey, bu ayrımı somutlaştırmakla ilgili.

MethodArgumentNotValidException: @Valid Gerçekte Ne Fırlatır

"Validation & Exception Handling", `@Valid`'in validasyonu tetiklediğini gösterdi, ama bir `@RequestBody` üzerinde başarısız olduğunda gerçekte ne olduğunu adlandırmadı: Spring, controller metodunun gövdesi hiç çalışmadan önce bir `MethodArgumentNotValidException` fırlatır.

```
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
    }
}
```

Exception, Spring MVC'nin geleneksel form binding için kullandığı TAM OLARAK AYNI tür olan bir `BindingResult` taşır -- başarısız her kısıt için bir `FieldError`, her biri kendi alanını ve kısıtın mesajını adlandırır. `exception.getBindingResult().getFieldErrors()`, bir handler'ın yalnızca ilkini değil, her tek hatayı aynı anda okumasının yoludur.

Validasyon Hatalarını Bir ProblemDetail'e Dönüştürmek

`MethodArgumentNotValidException`'ı yakalayan bir `@RestControllerAdvice` metodu, o `BindingResult`'ı bir yanıta dönüştürür -- ve yalnızca alan-bazlı hatalardan fazlasını okuması gerekir.

```
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()`, sıradan tek-alan hatalarını (`@NotBlank`, `@Positive` ve geri kalanı) kapsar. `getGlobalErrors()`, "Validation & Exception Handling"in hiç ihtiyaç duymadığı bir şeyi kapsar: "Java Bean Validation"daki `@ValidDateRange` gibi tek bir alana bağlı olmayan, bu yüzden bir `FieldError` yerine bir `ObjectError` olarak ortaya çıkan sınıf-seviyesi bir custom

kısıt. Yalnızca `getFieldErrors()`'ı okuyan bir handler, başarısız bir çapraz-alan kuralını yanıtından sessizce tamamen düşürürdü.

Custom ProblemDetail Özellikleri

Spring MVC'nin dersi, bir `ProblemDetail`'e tek bir `"errors"` özelliği ekledi. Pratikte, gerçek bir API'nin hata gövdesi genelde bundan fazlasını gerektirir.

```
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(...)`, ve tekrarlanan `setProperty(...)` çağrıları, makine tarafından okunabilir bir `errorCode`, ilgili spesifik kaynak, ve bir `timestamp` ile donatılmış bir `ProblemDetail` inşa eder -- hepsi hâlâ geçerli RFC 7807'dir, çünkü `ProblemDetail` tam olarak bu tür bir genişletme için tasarlanmıştır. Bir client, `errorCode` üzerinden, okunabilir bir mesaj string'i üzerinde asla güvenle yapamayacağı şekilde güvenilir biçimde dallanabilir.

Bir Domain Exception İçin Doğru Durum Kodunu Seçmek

Farklı iş hataları farklı durum kodlarını hak eder -- doğru olanı seçmek, her client'ı hangi kategoride bir sorun olduğunu anlamak için response gövdesini incelemeye zorlamak yerine, spesifik bir şey iletir.

```
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`, hakkında soru sorulan kaynağın var olmadığı anlamına gelir. `409 Conflict`, isteğin kaynağın mevcut durumuyla çakıştığı anlamına gelir (bu örnekte bir yineleme). `422 Unprocessable Entity`, isteğin iyi biçimlendirilmiş ve anlaşılmış olduğu ama bir iş kuralını ihlal

ettiği anlamına gelir -- yukarıdaki bölümlerde işlenen, isteğin kendisinin bozuk olduğu ya da validasyondan geçemediği anlamına gelen `400 Bad Request`'ten temel ayrım.

> Hızlı bir kural: isteğin kendisi bozuksa (eksik alanlar, yanlış türler), bu `400`'dür. İstek iyi biçimlendirilmiş ama sorduğu ŞEY yoksa, bu `404`'tür. İyi biçimlendirilmiş ama mevcut durumla çakışıyorsa, bu `409`'dur. İyi biçimlendirilmiş ve kaynak var, ama bir iş kuralı yine de reddediyorsa, bu `422`'dir.

ResponseEntityExceptionHandler ile Framework Exception'larını Merkezileştirmek

Spring MVC'nin dersindeki, ayrı `@ExceptionHandler` metotlarıyla `@RestControllerAdvice`, bir uygulamanın KENDİ exception'larının ele alınmasını merkezileştirir. `ResponseEntityExceptionHandler`, Spring MVC'nin KENDİSİNİN fırlattığı exception'lar için eşdeğerini yapar.

```
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.
    }
}
```

Onu genişletmek ve tek bir metodu -- burada `handleMethodArgumentNotValid(...)` -- override etmek, tam olarak o tek durumu özelleştirir, zaten ele almayı bildiği diğer her framework exception'ı (bozuk bir JSON gövdesi, desteklenmeyen bir medya türü, eksik bir parametre, ve daha fazlası) ise, hiçbiri için kod yazmadan, otomatik olarak makul varsayılan davranışını korur.

Hata Yanıtlarını Güvenli Tutmak

Bir exception'ın mesajı ya da stack trace'i, genelde sunucudan asla çıkmaması gereken bilgiler içerir -- bir veritabanı hostname'i, bir iç dosya yolu, bir kütüphane sürümü.

```
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.
    }
}
```

Güvensiz versiyon -- `e.toString()`'i ya da exception'dan doğrudan bir mesajı döndürmek -- tam olarak bu bilgiyi, saldırgan olsun olmasın, isteği gönderen kişiye teslim eder. Güvenli versiyon, TAM exception'ı yalnızca ekibin görebileceği bir yerde loglar, ve client'a genel, sabit bir mesaj döndürür -- iki hedef kitle de (bir log'u debug eden bir mühendis, bir yanıtı okuyan bir client) tam olarak sahip olması gereken bilgiyi alır, fazlasını değil.

Pratik, Uçtan Uca Bir Örnek

Yukarıdakilerin hepsini tek, gerçekçi bir endpoint'te birleştirmek, bu parçaların pratikte nasıl bir araya geldiğini gösterir.

```
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(...)`, üç farklı şekilde başarısız olabilir, ve `OrderExceptionAdvice`, her birini gerçekten gerektirdiği teknikle ele alır: bir `MethodArgumentNotValidException`, alan-bazlı detayla bir `400`'e dönüşür, bir `OutOfStockException`, makine tarafından okunabilir bir `errorCode`'la bir `422`'ye dönüşür, ve diğer her şey tam olarak loglanıp güvenli, genel bir `500`'e indirgenir -- tek bir merkezi sınıf, bütün bir controller'ın gerçekçi başarısızlık modlarını kapsıyor.

Best Practices

- Bir durum kodunu alışkanlıkla ya da kolaylıkla değil, gerçekte neyin ters gittiğine göre (bozuk istek, eksik kaynak, çakışan durum, reddedilen iş kuralı, gerçek sunucu hatası) seç.
- Bir `BindingResult`'tan hem `getFieldErrors()`'ı hem `getGlobalErrors()`'ı oku -- sınıf-seviyesi bir custom kısıtın hatası yalnızca ikincisinde görünür.
- Bir client'ın spesifik hataya göre dallanması gerekebileceği her durumda, yalnızca bir mesaj göstermek yerine makine tarafından okunabilir bir `errorCode`'u custom bir `ProblemDetail` özelliği olarak ekle.
- Beklenmeyen her şey için tam exception'ı içeride logla ve dışarıda genel bir mesaj döndür -- `e.getMessage()`'ın ya da bir stack trace'in asla doğrudan bir client'a ulaşmasına izin verme.

Yaygın Hatalar

- Bir validasyon hatası ya da bir iş kuralı reddi için `500` döndürmek -- ikisi de client-kaynaklıdır, ve ikisi de client'ın harekete geçebileceği bir `4xx` durumunu hak eder.
- Yalnızca `getFieldErrors()`'ı okuyup, sınıf-seviyesi bir kısıtın hatasını -- yalnızca `getGlobalErrors()`'da göründüğü için -- tamamen kaçırmak.
- `409 Conflict` ve `422 Unprocessable Entity`'yi birbirinin yerine kullanmak -- bir conflict mevcut durumla ilgilidir; unprocessable bir entity, herhangi bir conflict'ten bağımsız, bir iş kuralıyla ilgilidir.
- Bir exception'ın ham mesajını ya da stack trace'ini bir response gövdesinde açığa çıkarmak, bir client'ın (ya da saldırganın) asla görmemesi gereken implementasyon detaylarını sızdırmak.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bir `@RequestBody` üzerinde `@Valid`'in başarısız olması, alan-bazlı ve sınıf-seviyesi hataları taşıyan bir `BindingResult` içeren `MethodArgumentNotValidException` fırlatır.
- `getFieldErrors()`, tek-alan hatalarını kapsar; `getGlobalErrors()`, bir çapraz-alan kuralı gibi sınıf-seviyesi custom kısıtları kapsar.
- `ProblemDetail`, tek bir hata listesinin ötesinde -- bir hata kodu, bir kaynak id'si, bir timestamp -- bir API'nin ihtiyaç duyduğu kadar custom özelliği destekler.
- Farklı domain hataları farklı durum kodlarını hak eder: `400` bozuk, `404` eksik, `409` çakışan durum, `422` reddedilen iş kuralı, `500` gerçek sunucu hatası.
- `ResponseEntityExceptionHandler`, `@RestControllerAdvice`'ın bir uygulamanın kendi exception'larının ele alınmasını merkezileştirmesi gibi, Spring MVC'nin kendi exception'larının ele alınmasını merkezileştirir.
- Güvenli bir hata yanıtı, tam exception'ı içeride loglar ve dışarıda yalnızca genel bir mesaj döndürür.

Cheat Sheet

```
// İki tür validasyon hatasını da okumak
ex.getBindingResult().getFieldErrors();   // alan-bazlı
ex.getBindingResult().getGlobalErrors();  // sınıf-seviyesi custom kısıtlar

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

// Domain exception'lar için durum kodları
@ExceptionHandler(NotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND) // 404
@ExceptionHandler(ConflictException.class)
@ResponseStatus(HttpStatus.CONFLICT) // 409
@ExceptionHandler(BusinessRuleException.class)
@ResponseStatus(HttpStatus.UNPROCESSABLE_ENTITY) // 422

// Framework exception'larını merkezileştirmek
class GlobalMvcExceptionHandler extends ResponseEntityExceptionHandler {
    @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(...) { ... }
}

// Güvenli 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.");
}
```

Terimler Sözlüğü

- **MethodArgumentNotValidException**: `@Valid`, bir `@RequestBody` üzerinde başarısız olduğunda Spring MVC'nin fırlattığı, bir `BindingResult` taşıyan exception.
- **Field error vs. object error**: bir `FieldError`, başarısız olan tek bir alanı raporlar; bir `ObjectError` (`getGlobalErrors()`'dan), tek bir alana bağlı olmayan sınıf-seviyesi bir hatayı raporlar.
- **422 Unprocessable Entity**: iyi biçimlendirilmiş, anlaşılmış ama yine de bir iş kuralını ihlal eden bir istek için durum kodu.
- **ResponseEntityExceptionHandler**: Spring MVC'nin kendi hazır exception'larını merkezi olarak ele almak için base sınıfı.
- **Güvenli hata yanıtı (safe error response)**: tam hata detayını içeride loglayan ama dışarıda yalnızca genel bir mesaj açığa çıkaran bir yanıt.