Request ve Response Handling

@RequestBody ile istek gövdesini nesneye çevirmek, ResponseEntity ile yanıtı tam kontrol etmek, HTTP durum kodları ve content negotiation.

Orta 45 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/request-response-handling

---

Request ve Response Handling

Spring MVC Temelleri dersinde `@ResponseBody` / `@RestController`'ın bir nesneyi otomatik olarak JSON'a çevirdiğini görmüştük, Mapping Annotation'ları dersinde `consumes` / `produces`'ı tanımıştık. Bu derste bu mekanizmaların **perde arkasına** iniyoruz: `@RequestBody`'nin bir JSON gövdesini nasıl bir Java nesnesine çevirdiğini, `ResponseEntity` ile yanıtı tam olarak nasıl kontrol edeceğini, HTTP durum kodlarının her birinin ne zaman kullanılacağını, ve content negotiation'ın istemci ile sunucu arasında nasıl bir "anlaşma" olduğunu.

Request ve Response Handling Nedir?

Bir HTTP isteğinin ve yanıtının, path/query string/header'ların ötesinde bir de **gövdesi** (body) vardır -- genelde JSON, isteğe göre XML ya da başka bir format. `@RequestBody` bu gövdeyi okur, `@ResponseBody` / `ResponseEntity` gövdeyi yazar:

```
@PostMapping("/users")
public ResponseEntity<User> create(@RequestBody CreateUserRequest request) {
    // request, isteğin JSON gövdesinden otomatik doldurulduu
    User created = ...;
    return ResponseEntity.status(HttpStatus.CREATED).body(created);
    // created, yanıtın JSON gövdesine otomatik yazılacak
}
```

Neden Var?

Path variable'lar ve query parametreleri (bir önceki derste gördüğümüz gibi) tek tek, adlandırılmış değerler taşımak için idealdir -- ama karmaşık, iç içe yapılar (bir adres, birden fazla alan içeren bir sipariş) için pratik değildir; her alan için ayrı bir `@RequestParam` yazmak gerekirdi. Gövde, tüm bu veriyi **tek bir yapılandırılmış belge** olarak taşımanın yolu. Benzer şekilde, dönüş değerini olduğu gibi (200 OK, JSON) döndürmek çoğu durumda

yeterlidir, ama gerçek bir API'nin durum kodunu, header'larını (`Location`, özel header'lar) ve içerik türünü isteğe göre ayarlaması gerekir -- bu kontrolü `ResponseEntity` sağlar.

Tarihçe

Spring MVC Temelleri dersinin "Tarihçe" bölümünde bahsettiğimiz gibi, `@RequestBody` ve `@ResponseBody`, `@PathVariable` ile birlikte Spring 3.0'da (2009) geldi -- üçü de aynı hedefe hizmet ediyordu: REST tarzı, JSON tabanlı API'leri Spring MVC'nin ilk günlerindeki view-odaklı (HTML döndüren) modelin yanında ilk sınıf vatandaş yapmak. `ResponseEntity` de aynı dönemde eklendi -- yalnızca gövdeyi değil, durum kodunu ve header'ları da tek bir nesnede taşıyabilen bir sarmalayıcı olarak.

@RequestBody: İstek Gövdesini Nesneye Çevirmek

`@RequestBody`, isteğin **tüm** gövdesini okuyup bir Java nesnesine çevirir -- `@RequestParam` / `@PathVariable`'ın aksine, tek bir adlandırılmış değeri değil, gövdenin tamamını hedefler:

```
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;

// @RequestBody reads the ENTIRE HTTP request body and deserializes it into a Java
// object -- unlike @RequestParam/@PathVariable, which each read one named value, this
// reads the whole body at once.
@Controller
class UserCreationController {

    record CreateUserRequest(String name, String email) {
    }

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

İstek gövdesindeki `{"name": "...", "email": "..."}` JSON'ı, `CreateUserRequest` record'unun alanlarına isim eşleşmesiyle otomatik dolduruluyor. Bu dönüşümü kimin yaptığını "HttpMessageConverter: @RequestBody/@ResponseBody'nin Perde Arkası" bölümünde göreceğiz.

HttpMessageConverter: @RequestBody/@ResponseBody'nin Perde Arkası

`@RequestBody` / `@ResponseBody`, JSON dönüşümünü kendileri yapmaz -- bir `HttpMessageConverter`'a devrederler; JSON için bu converter, doğrudan burada kullandığımız Jackson `ObjectMapper`'ın kendisidir:

```
import com.fasterxml.jackson.databind.ObjectMapper;

// @RequestBody and @ResponseBody don't do the JSON conversion themselves -- they
// delegate to an HttpMessageConverter, and for JSON that converter is backed by
// exactly the Jackson ObjectMapper used directly here. spring-boot-starter-web
// auto-configures one of these and registers it as a bean; this is what it does
// under the hood on every request/response.
class HttpMessageConverterExample {

    record Product(String name, double price) {
    }

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();

        // What happens to an incoming @RequestBody:
        String requestJson = "{\"name\":\"Keyboard\",\"price\":49.9}";
        Product product = mapper.readValue(requestJson, Product.class);
        System.out.println(product);
        // Product[name=Keyboard, price=49.9]

        // What happens to an outgoing @ResponseBody:
        String responseJson = mapper.writeValueAsString(product);
        System.out.println(responseJson);
        // {"name":"Keyboard","price":49.9}
    }
}
```

`spring-boot-starter-web`, bu converter'ı otomatik olarak yapılandırıp bir bean olarak kaydeder -- Spring Boot Auto-Configuration & Properties dersinde gördüğümüz auto-configuration mekanizmasının bir başka örneği. Gerçek bir istekte `mapper.readValue(...)`'ı sen çağırmazsın, DispatcherServlet senin adına çağırır; burada gördüğün, her isteğin/yanıtın arkasında gerçekleşenin birebir aynısı.

İç İçe Nesneler ve Listelerin Deserialize Edilmesi

`@RequestBody`, düz (flat) nesnelerle sınırlı değil -- Jackson, her seviyede eşleşen bir Java tipi olduğu sürece iç içe nesneleri ve listeleri de özyinelemeli olarak deserialize eder:

```
import com.fasterxml.jackson.databind.ObjectMapper;

import java.util.List;

// @RequestBody isn't limited to flat objects -- Jackson recursively deserializes
// nested objects and lists, as long as every level has a matching Java type.
class NestedObjectDeserializationExample {

    record Address(String city, String country) {
    }

    record OrderRequest(String customerName, Address shippingAddress, List<String> items) {
    }

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();

        String json = """
                {
                  "customerName": "Ayse",
                  "shippingAddress": { "city": "Istanbul", "country": "Turkey" },
                  "items": ["Keyboard", "Mouse"]
                }
                """;

        OrderRequest order = mapper.readValue(json, OrderRequest.class);
        System.out.println(order);
        // OrderRequest[customerName=Ayse, shippingAddress=Address[city=Istanbul,
country=Turkey], items=[Keyboard, Mouse]]
    }
}
```

`OrderRequest`'in içindeki `shippingAddress` (bir `Address` nesnesi) ve `items` (bir `List<String>`), tek bir `readValue(...)` çağrısıyla, hiçbir elle yazılmış dönüşüm kodu olmadan tam olarak doldurulyor -- Jackson, JSON'ın yapısını Java tipinin yapısıyla adım adım eşliyor.

Eksik ya da Fazla Alanlar: Jackson Nasıl Davranır?

JSON'da eksik olan bir alanla, JSON'da olup Java tarafında karşılığı olmayan fazla bir alan, Jackson'da çok farklı iki davranışa yol açar:

```
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.exc.UnrecognizedPropertyException;

// Two very different failure modes: a field the JSON is MISSING is usually silently
// null (harmless, unless a lower-level lesson like Validation adds a rule against
// it); a field the JSON has EXTRA that Java doesn't know about is rejected outright,
// by Jackson's default configuration.
class UnknownFieldsToleranceExample {

    record CreateUserRequest(String name, String email) {
    }

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();

        String missingField = "{\"name\":\"Ayse\"}";
        CreateUserRequest withMissingField = mapper.readValue(missingField,
CreateUserRequest.class);
        System.out.println(withMissingField);
        // CreateUserRequest[name=Ayse, email=null]

        String extraField = "{\"name\":\"Ayse\",\"email\":\"ayse@example.com\",\"age\":30}";
        try {
            mapper.readValue(extraField, CreateUserRequest.class);
        } catch (UnrecognizedPropertyException e) {
            System.out.println("Rejected unknown field: " + e.getPropertyName());
            // Rejected unknown field: age
        }
    }
}
```

`email` alanı eksik olduğunda, hiçbir hata olmadan sessizce `null` atanıyor. `age` diye bilinmeyen bir alan geldiğinde ise `UnrecognizedPropertyException` fırlatılıyor -- Jackson'ın varsayılan ayarı, tanımadığı alanları **reddetmek**tir. Eksik alanlar için hiçbir "zorunlu" kontrolü yoktur -- bunu Bean Validation sağlıyor, bir sonraki derste (Validation & Exception Handling) ele alacağımız konu.

ResponseEntity: Yanıtı Tam Kontrol Etmek

Düz bir nesne döndürmek her zaman `200 OK` gönderir. `ResponseEntity`, gövdeyle birlikte durum kodunun tam kontrolünü sağlar:

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

import java.util.Map;

// Returning a plain object always sends 200 OK. ResponseEntity gives full control
// over the status code (and, as we'll see next, headers) alongside the body.
@Controller
class ProductLookupController {
    private final Map<Long, String> products = Map.of(1L, "Keyboard");

    @GetMapping("/products/{id}")
    @ResponseBody
    public ResponseEntity<String> getProduct(@PathVariable Long id) {
        String name = products.get(id);
        if (name == null) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
        }
        return ResponseEntity.ok(name);
    }
}
```

Ürün bulunduğunda `ResponseEntity.ok(name)` ile `200`, bulunamadığında `ResponseEntity.status(HttpStatus.NOT_FOUND).build()` ile `404` dönüyor -- aynı metot, koşula göre iki farklı durum kodu üretebiliyor; bu, düz bir dönüş değeriyle mümkün olmayan bir esneklik.

ResponseEntity ile Header Eklemek

`ResponseEntity`'nin builder'ı, durum koduyla birlikte header da ekleyebilir -- en yaygın örnek, yeni oluşturulan bir kaynağın adresini bildiren `Location`:

```
import org.springframework.http.ResponseEntity;
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;

import java.net.URI;

// ResponseEntity.BodyBuilder can also set headers -- the most common case being
// Location, telling the client where the resource it just created now lives.
@Controller
class ArticleCreationController {

    record CreateArticleRequest(String title) {
    }

    @PostMapping("/articles")
    @ResponseBody
    public ResponseEntity<Void> create(@RequestBody CreateArticleRequest request) {
        long newId = 42; // pretend this came from a real save operation
        URI location = URI.create("/articles/" + newId);

        return ResponseEntity.created(location)
                .header("X-Created-By", "learning-platform")
                .build();
    }
}
```

`ResponseEntity.created(location)`, hem `201 Created` durumunu hem de `Location` header'ını tek satırda ayarlıyor; `.header(...)` ile ek, özel header'lar da eklenebiliyor. İstemci, `Location` header'ından yeni kaynağın URL'sini okuyup doğrudan ona gidebilir.

HTTP Durum Kodları: Ne Zaman Hangisi

Durum kodları rastgele seçilmez -- her biri istemciye belirli, standartlaşmış bir anlam iletir:

- **2xx**: istek başarıyla işlendi (bkz. "2xx Başarı Kodları: 200, 201, 204")
- **4xx**: istekte bir sorun var -- istemci bir şeyi düzeltmeli (bkz. "4xx İstemci Hataları: 400, 401, 403, 404, 409")
- **5xx**: istek geçerliydi ama sunucu tarafında bir hata oluştu (bkz. "5xx Sunucu Hataları: 500")

Bu üç kategoriyi ayırt etmek önemlidir: bir istemci 4xx aldığında isteğini değiştirmeden tekrar denemenin anlamı yoktur (aynı hata tekrar oluşur); 5xx aldığında ise -- isteğin kendisi geçerli olduğu için -- bir süre sonra tekrar denemek makul olabilir.

2xx Başarı Kodları: 200, 201, 204

En sık karşılaşılan üç başarı kodu, farklı senaryolara karşılık gelir:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.ResponseBody;

// The three 2xx codes that come up constantly: 200 (a normal successful read/update
// with a body), 201 (a new resource was created, usually with a Location header, see
// "Adding Headers with ResponseEntity"), 204 (successful, but there's nothing to send
// back).
@Controller
class NoteController {

    @GetMapping("/notes/1")
    @ResponseBody
    public ResponseEntity<String> get() {
        return ResponseEntity.status(HttpStatus.OK).body("Buy milk");
    }

    @PostMapping("/notes")
    @ResponseBody
    public ResponseEntity<String> create() {
        return ResponseEntity.status(HttpStatus.CREATED).body("Note created");
    }

    @DeleteMapping("/notes/1")
    @ResponseBody
    public ResponseEntity<Void> delete() {
        return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
    }
}

class StatusCode2xxExample {
    public static void main(String[] args) {
        NoteController controller = new NoteController();

        System.out.println(controller.get().getStatusCode());
        // 200 OK
        System.out.println(controller.create().getStatusCode());
        // 201 CREATED
        System.out.println(controller.delete().getStatusCode());
        // 204 NO_CONTENT
    }
}
```

`200 OK`, normal bir başarılı okuma/güncelleme (gövdeli). `201 Created`, yeni bir kaynak oluşturuldu (genelde "ResponseEntity ile Header Eklemek" bölümünde gördüğümüz `Location` header'ıyla birlikte). `204 No Content`, işlem başarılı ama geri gönderilecek bir gövde yok -- Mapping Annotation'ları dersindeki "DELETE ve Idempotency" bölümünde de gördüğümüz kod.

4xx İstemci Hataları: 400, 401, 403, 404, 409

Beş yaygın 4xx kodu, `ResponseStatusException` ile (bu projenin kendi `TopicController`'ının da kullandığı sınıf) fırlatılıyor:

```
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.DeleteMapping;
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.server.ResponseStatusException;

import java.util.Map;

// ResponseStatusException -- the same class this project's own TopicController uses
// (see "This Project's Own Mappings" in an earlier lesson) -- is the simplest way to
// signal a 4xx from anywhere in a controller: throw it, and DispatcherServlet turns
// it into the right HTTP response, no manual ResponseEntity needed.
@Controller
class AccountController {
    private final Map<Long, String> accounts = Map.of(1L, "checking");
    private final Map<Long, String> owners = Map.of(1L, "ayse");

    record TransferRequest(Long fromAccountId, Double amount) {
    }

    @PostMapping("/transfers")
    @ResponseBody
    public String transfer(@RequestBody TransferRequest request) {
        if (request.amount() == null || request.amount() <= 0) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "amount must be
positive"); // 400
        }
        return "Transfer accepted";
    }

    @GetMapping("/accounts/{id}")
    @ResponseBody
    public String getAccount(@PathVariable Long id, String currentUser) {
        if (currentUser == null) {
            throw new ResponseStatusException(HttpStatus.UNAUTHORIZED, "login required");
// 401
        }
        if (!owners.getOrDefault(id, "").equals(currentUser)) {
            throw new ResponseStatusException(HttpStatus.FORBIDDEN, "not your account"); //
403
        }
        String account = accounts.get(id);
        if (account == null) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, "account not found");
// 404
        }
```

```
        return account;
    }

    @DeleteMapping("/accounts/{id}")
    @ResponseBody
    public String closeAccount(@PathVariable Long id) {
        String account = accounts.get(id);
        if ("checking".equals(account)) {
            // Business rule violated: checking accounts with a balance can't be
            // closed. The request is well-formed, but conflicts with server state.
            throw new ResponseStatusException(HttpStatus.CONFLICT, "account has a
balance"); // 409
        }
        return "Account closed";
    }
}
```

`400 Bad Request`: gövde/parametre geçersiz (`amount` negatif). `401 Unauthorized`: istemci hiç kimlik doğrulamamış. `403 Forbidden`: istemci kimliği belli ama bu kaynağa erişim yetkisi yok -- 401'den farkı, "kim olduğunu biliyoruz, yine de izin yok" olması. `404 Not Found`: kaynak yok. `409 Conflict`: istek biçim olarak geçerli, ama sunucudaki mevcut durumla çelişiyor (bakiyesi olan bir hesabı kapatmaya çalışmak gibi).

5xx Sunucu Hataları: 500

4xx'in aksine, `500` genelde **kasıtlı olarak** döndürülmez -- kimsenin yakalamadığı bir exception'a Spring'in verdiği varsayılan yanıttır:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

// Unlike a 4xx, a 500 usually isn't something you return on purpose -- it's Spring's
// DEFAULT response when a controller method throws an exception nobody handled.
@Controller
class ReportController {

    @GetMapping("/reports/summary")
    @ResponseBody
    public String summary() {
        int result = 1 / computeDivisor(); // bug: divisor can be 0, throws
ArithmeticException
        return "Result: " + result;
        // With no ResponseStatusException and no @ExceptionHandler (the next lesson,
        // Validation & Exception Handling, covers those) to catch it,
        // DispatcherServlet's default error handling turns the uncaught
        // ArithmeticException into a generic 500 Internal Server Error -- the
        // exception's details are logged server-side but never exposed to the client.
    }

    private int computeDivisor() {
        return 0;
    }
}
```

`ArithmeticException`, ne bir `ResponseStatusException` ne de (bir sonraki derste göreceğimiz) bir `@ExceptionHandler` tarafından yakalanıyor -- DispatcherServlet'in varsayılan hata işleyicisi devreye girip istemciye genel bir `500 Internal Server Error` döndürüyor; exception'ın ayrıntıları yalnızca sunucu loglarında kalıyor, istemciye hiç sızmıyor.

Content Negotiation: Accept ile Temsil Seçmek

Content negotiation, istemci (`Accept` header'ıyla) ve sunucunun (`produces` attribute'uyla), aynı kaynağın **hangi temsilini** değiş tokuş edeceği konusunda anlaşmasıdır:

```
import org.springframework.http.MediaType;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

// Content negotiation is the client (via the Accept header) and the server (via
// `produces`) agreeing on which REPRESENTATION of the same resource to exchange.
// Both mappings below serve the same underlying data, in different formats.
@Controller
class ProductRepresentationController {

    @GetMapping(path = "/products/1", produces = MediaType.APPLICATION_JSON_VALUE)
    @ResponseBody
    public String asJson() {
        return "{\"name\":\"Keyboard\",\"price\":49.9}";
    }

    @GetMapping(path = "/products/1", produces = MediaType.APPLICATION_XML_VALUE)
    @ResponseBody
    public String asXml() {
        return "<product><name>Keyboard</name><price>49.9</price></product>";
    }
    // A request with "Accept: application/json" matches asJson(); "Accept:
    // application/xml" matches asXml(). A request with "Accept: text/csv" -- a
    // representation neither method produces -- matches neither, and DispatcherServlet
    // responds with 406 Not Acceptable before either method is ever called.
}
```

Aynı path (`/products/1`) iki farklı `produces` değeriyle iki kez tanımlı -- `Accept: application/json` gönderen bir istemci `asJson()` 'a, `Accept: application/xml` gönderen bir istemci `asXml()` 'e yönlendiriliyor. Bu, Mapping Annotation'ları dersindeki `consumes` / `produces` bölümünün bir uzantısı -- orada `consumes` , giden isteğin türünü; burada `produces` + `Accept` , dönen yanıtın türünü belirliyor.

Desteklenmeyen Bir Temsil İstendiğinde: 406 Not Acceptable

Bir istemci, sunucunun **hiçbir mapping'inin üretmediği** bir temsil isterse (`Accept: text/csv` gibi, önceki bölümün örneğinde), DispatcherServlet `404` değil `406 Not Acceptable` döner -- path var, ama istenen temsil hiçbir mapping'de yok. Bu, Mapping Annotation'ları dersindeki "Desteklenmeyen Bir HTTP Metodu İstendiğinde: 405 Method Not Allowed" bölümündeki mantığın bir başka boyutu: 404 (path yok), 405 (path var, metot yok), 406 (path ve metot var, temsil yok) -- üçü de "bir şey eksik ama tam olarak ne" sorusuna farklı, spesifik bir cevap verir.

Bu Projenin Kendi Response'ları: Gerçek Bir Örnek

Bu dersteki mekanizmaları, `TopicController` 'ın kendi kodunda görebilirsin -- proje şu an salt-okunur bir HTML sitesi olduğu için hâlâ `@RequestBody` kullanmıyor, ama artık iki farklı metotta hem `ResponseStatusException` 'ı ("4xx İstemci Hataları: 400, 401, 403, 404, 409"

bölümü) hem de `ResponseEntity`'yi ("ResponseEntity ile Header Eklemek" bölümü) gerçekten kullanıyor:

```
Topic topic = topicRepository.findBySlugWithCategoryAndCourse(slug)
        .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Konu
bulunamadı: " + slug));
// ...
return ResponseEntity.status(HttpStatus.MOVED_PERMANENTLY)
        .location(URI.create("/" + language.getCode() + "/topics/" + slug))
        .build();
```

İlki, path variable'la (bir önceki dersin "Path Variable mi, Query Parameter mı? Ne Zaman Hangisi" ayrımını hatırlarsak) kimliklendirilen bir kaynak bulunamadığında `404` döndürüyor. İkincisi ise elle kurulmuş, `ResponseEntity` ile gerçek bir `301 Moved Permanently` ve bir `Location` header'ı -- dil URL path'ine taşınmadan önceki eski `/topics/{slug}?lang=..` adreslerinin bugün hâlâ çalışmasının sebebi bu: kırılmak yerine yeni adrese yönlendiriliyorlar.

Best Practices

- **Gerçekten yeni bir kaynak oluşturduğunda `201` + `Location` header'ını kullan, yalnızca `200` ile yetinme** -- "ResponseEntity ile Header Eklemek" bölümünde gördüğümüz gibi, istemcinin yeni kaynağın adresini elle inşa etmesine gerek kalmaz.
- **401 ile 403'ü karıştırma** -- "4xx İstemci Hataları: 400, 401, 403, 404, 409" bölümünde gördüğümüz gibi, biri kimlik doğrulamanın hiç yapılmadığını, diğeri kimliğin bilindiğini ama yetkinin olmadığını ifade eder; bu ayrım istemcinin doğru aksiyonu almasını sağlar (401'de giriş yap, 403'te farklı bir hesap dene).
- **500'ü asla kasıtlı olarak döndürme** -- "5xx Sunucu Hataları: 500" bölümünde gördüğümüz gibi, bu kod "beklenmeyen bir hata oldu" anlamına gelir; beklenen her hata durumu bir `ResponseStatusException` (ya da bir sonraki derste göreceğimiz `@ExceptionHandler`) ile ele alınmalıdır.
- **`@RequestBody` ile gelen veriye asla körü körüne güvenme** -- "Eksik ya da Fazla Alanlar: Jackson Nasıl Davranır?" bölümünde gördüğümüz gibi, Jackson yalnızca **biçimi** doğrular (JSON geçerli mi, tipler uyuşuyor mu); iş kurallarını (pozitif miktar, boş olmayan isim gibi) doğrulamak sana kalır -- bir sonraki ders bu doğrulamayı otomatikleştiren `@Valid`'i tanıtacak.

Yaygın Hatalar

1. `@RequestBody`'nin, `@RequestParam` gibi tek bir alanı okuduğunu sanmak. `@RequestBody` **tüm** gövdeyi tek bir nesneye çevirir -- birden fazla `@RequestBody` parametresi aynı metotta olamaz, çünkü gövde yalnızca bir kez okunabilir (bkz. "@RequestBody: İstek Gövdesini Nesneye Çevirmek").

2. Jackson'ın bilinmeyen JSON alanlarını sessizce yok sayacağını varsaymak. Varsayılan davranış tam tersi -- fazladan bir alan, isteği tamamen reddeder (bkz. "Eksik ya

da Fazla Alanlar: Jackson Nasıl Davranır?"). Bu varsayımla yazılmış bir istemci kodu, sunucu tarafında yeni bir alan eklendiğinde beklenmedik `400`'lerle karşılaşabilir.

3. Her başarı durumunda düşünmeden `200` dönmek, `201`/`204`'ü hiç kullanmamak. "2xx Başarı Kodları: 200, 201, 204" bölümünde gördüğümüz ayrım, istemcinin (özellikle otomatikleştirilmiş bir istemcinin) yanıtı doğru yorumlamasını sağlar -- bir oluşturma isteğinin `200` mü `201` mi döndüğü, istemcinin davranışını değiştirebilir.

4. Bir iş kuralı ihlalini (örn. bakiyesi olan hesabı kapatma) `400 Bad Request` ile karıştırmak. İstek biçim olarak tamamen geçerliyse ama sunucudaki mevcut durumla çelişiyorsa, doğru kod `409 Conflict`'tir -- `400`, isteğin **kendisinin** hatalı olduğu durumlar içindir (bkz. "4xx İstemci Hataları: 400, 401, 403, 404, 409").

5. `Accept` header'ını yok sayıp her zaman aynı formatı (örn. sadece JSON) döndürmek, sonra bir istemci XML beklediğinde neden `406` aldığını anlamamak. "Content Negotiation: Accept ile Temsil Seçmek" ve "Desteklenmeyen Bir Temsil İstendiğinde: 406 Not Acceptable" bölümlerinde gördüğümüz gibi, `406`, sunucunun o path için hiçbir `produces`'ının istenen `Accept` ile eşleşmediği anlamına gelir.

Özet, Cheat Sheet ve Terimler Sözlüğü

Request ve response handling, bir HTTP isteğinin/yanıtının gövdesini okuma/yazma (`@RequestBody` / `ResponseEntity`), hangi durum kodunun ne zaman kullanılacağı ve istemci-sunucu arasındaki temsil anlaşmasıdır (content negotiation). Öne çıkan noktalar:

- `@RequestBody`: isteğin tüm gövdesini bir Java nesnesine çevirir, bir `HttpMessageConverter` (JSON için Jackson `ObjectMapper`) aracılığıyla
- `ResponseEntity`: durum kodu + header'lar + gövdeyi tek bir nesnede taşır; `.ok()`, `.status(...)`, `.created(uri)`, `.noContent()` gibi builder metotları var
- Jackson, bilinmeyen JSON alanlarını **varsayılan olarak reddeder**, eksik alanlara ise sessizce `null` atar
- 2xx: başarı (200 okuma/güncelleme, 201 oluşturma, 204 gövdesiz başarı)
- 4xx: istemci hatası (400 geçersiz istek, 401 kimlik doğrulanmamış, 403 yetkisiz, 404 bulunamadı, 409 çelişki)
- 5xx: sunucu hatası, genelde kasıtsız (500, yakalanmamış exception'ların varsayılanı)
- 406 Not Acceptable: path var ama `Accept` ile eşleşen bir `produces` yok

Hızlı referans:

```
@PostMapping("/resource")
ResponseEntity<Void> create(@RequestBody CreateRequest request) {
    // ... doğrulama, kaydetme ...
    return ResponseEntity.created(URI.create("/resource/" + id)).build();  // 201
}

@GetMapping("/resource/{id}")
ResponseEntity<Resource> getOne(@PathVariable Long id) {
    Resource found = ...;
    return found != null
        ? ResponseEntity.ok(found)                                          // 200
        : ResponseEntity.status(HttpStatus.NOT_FOUND).build();              // 404
}

@DeleteMapping("/resource/{id}")
ResponseEntity<Void> delete(@PathVariable Long id) {
    // ...
    return ResponseEntity.noContent().build();                              // 204
}

// Herhangi bir yerden 4xx fırlatmak:
throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "sebep");
throw new ResponseStatusException(HttpStatus.CONFLICT, "sebep");
```

Terimler Sözlüğü

`@RequestBody` — Bir HTTP isteğinin tüm gövdesini bir Java nesnesine deserialize eden annotation.

`ResponseEntity` — Durum kodu, header'lar ve gövdeyi tek bir nesnede taşıyan, yanıt üzerinde tam kontrol sağlayan sarmalayıcı sınıf.

`HttpMessageConverter` — `@RequestBody` / `@ResponseBody`'nin, gövde ile Java nesnesi arasındaki gerçek dönüşümü devrettiği bileşen (JSON için Jackson `ObjectMapper` tabanlı).

`ResponseStatusException` — Bir controller metodunun herhangi bir yerinden fırlatılarak DispatcherServlet'e belirli bir HTTP durum kodu döndürtmesini sağlayan sınıf.

Content negotiation — İstemci (`Accept` header'ı) ve sunucunun (`produces`), aynı kaynağın hangi temsilinin değiş tokuş edileceği konusunda anlaşması.

404 Not Found — İstenen kaynağın (path'in kendisinin) bulunamadığı durumda dönen HTTP durum kodu.

406 Not Acceptable — Path var ama istemcinin `Accept` header'ıyla eşleşen bir `produces` temsili olmadığında dönen HTTP durum kodu.

409 Conflict — İstek biçim olarak geçerli ama sunucudaki mevcut durumla çeliştiğinde dönen HTTP durum kodu.

500 Internal Server Error — Hiçbir yerde yakalanmamış bir exception için Spring'in döndürdüğü varsayılan HTTP durum kodu.

Ek: Mini Proje — Sipariş Oluşturma API'si

Bu dersteki her mekanizmayı, gerçekçi bir sipariş oluşturma/okuma API'sinde bir araya getiriyoruz:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
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.server.ResponseStatusException;

import java.net.URI;
import java.util.LinkedHashMap;
import java.util.Map;

// An order-creation endpoint using every mechanism from this lesson: @RequestBody to
// read the order, manual validation (Bean Validation arrives in the next lesson) that
// throws a ResponseStatusException for bad input, and a ResponseEntity with a
// Location header and 201 Created for success.
@Controller
class OrderApiController {
    private final Map<Long, String> orders = new LinkedHashMap<>();
    private long nextId = 1;

    record CreateOrderRequest(String item, Integer quantity) {
    }

    @PostMapping("/api/orders")
    @ResponseBody
    public ResponseEntity<Void> create(@RequestBody CreateOrderRequest request) {
        if (request.item() == null || request.item().isBlank()) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "item is required");
        }
        if (request.quantity() == null || request.quantity() <= 0) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "quantity must be
positive");
        }

        long id = nextId++;
        orders.put(id, request.quantity() + "x " + request.item());

        return ResponseEntity.created(URI.create("/api/orders/" + id)).build();
    }

    @GetMapping("/api/orders/{id}")
    @ResponseBody
    public ResponseEntity<String> getOne(@PathVariable Long id) {
        String order = orders.get(id);
        return order != null ? ResponseEntity.ok(order) : ResponseEntity.status(HttpStatus.
NOT_FOUND).build();
    }
}
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.server.ResponseStatusException;

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

        ResponseEntity<Void> created = controller.create(new OrderApiController.
CreateOrderRequest("Keyboard", 2));
        System.out.println(created.getStatusCode() + " Location=" + created.getHeaders().
getLocation());
        // 201 CREATED Location=/api/orders/1

        System.out.println(controller.getOne(1L).getBody());
        // 2x Keyboard

        try {
            controller.create(new OrderApiController.CreateOrderRequest("Mouse", 0));
        } catch (ResponseStatusException e) {
            System.out.println(e.getStatusCode() + ": " + e.getReason());
            // 400 BAD_REQUEST: quantity must be positive
        }
    }
}
```

`create(...)`, "Eksik ya da Fazla Alanlar: Jackson Nasıl Davranır?" bölümünde bahsettiğimiz manuel doğrulamayı (`item` / `quantity` kontrolleri) yapıyor, geçersizse "4xx İstemci Hataları: 400, 401, 403, 404, 409" bölümündeki `ResponseStatusException` ile `400` fırlatıyor; geçerliyse "ResponseEntity ile Header Eklemek" bölümündeki desenle `201` + `Location` döndürüyor. `OrderApiDemo`, hem başarı hem hata yollarını, gerçek bir DispatcherServlet olmadan doğrudan metot çağrısıyla çalıştırıyor.

Ek: Mini Proje — Elle Yazılmış Bir HttpMessageConverter Zinciri Simülasyonu

Son mini proje, "HttpMessageConverter: @RequestBody/@ResponseBody'nin Perde Arkası" ve "Content Negotiation: Accept ile Temsil Seçmek" bölümlerini tek bir mekanizmada birleştiriyor -- birden fazla converter'ın, `Accept`'e göre seçilmesi:

```
import com.fasterxml.jackson.databind.ObjectMapper;

import java.util.Map;
import java.util.function.Function;

// A miniature model of Spring's HttpMessageConverter chain: several converters, each
// claiming a media type, picked based on what the request/response needs -- the same
// idea behind both @RequestBody deserialization ("HttpMessageConverter: The Mechanism
// Behind @RequestBody/@ResponseBody") and content negotiation ("Content Negotiation:
// Choosing a Representation with Accept").
class MessageConverterSimulation {

    record Product(String name, double price) {
    }

    private final ObjectMapper jsonMapper = new ObjectMapper();

    private final Map<String, Function<Product, String>> writers = Map.of(
            "application/json", this::toJson,
            "application/xml", this::toXml
    );

    String write(Product product, String acceptHeader) {
        Function<Product, String> writer = writers.get(acceptHeader);
        if (writer == null) {
            return "406 Not Acceptable: " + acceptHeader;
        }
        return writer.apply(product);
    }

    Product read(String json) throws Exception {
        return jsonMapper.readValue(json, Product.class);
    }

    private String toJson(Product product) {
        try {
            return jsonMapper.writeValueAsString(product);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private String toXml(Product product) {
        return "<product><name>" + product.name() + "</name><price>" + product.price() + "<
/price></product>";
    }
}
```

```
class MessageConverterDemo {
    public static void main(String[] args) throws Exception {
        MessageConverterSimulation converters = new MessageConverterSimulation();

        MessageConverterSimulation.Product product = converters.read("{\"name\":\"
Keyboard\",\"price\":49.9}");
        System.out.println(product);
        // Product[name=Keyboard, price=49.9]

        System.out.println(converters.write(product, "application/json"));
        // {"name":"Keyboard","price":49.9}
        System.out.println(converters.write(product, "application/xml"));
        // <product><name>Keyboard</name><price>49.9</price></product>
        System.out.println(converters.write(product, "text/csv"));
        // 406 Not Acceptable: text/csv
    }
}
```

`writers` haritası, gerçek Spring'in `List<HttpMessageConverter<?>>`'ının minik bir modeli -- her biri bir media type'ı "iddia ediyor". `write(...)`, `acceptHeader`'a uyan bir converter bulamazsa "Desteklenmeyen Bir Temsil İstendiğinde: 406 Not Acceptable" bölümünde gördüğümüz kodu üretiyor -- gerçek DispatcherServlet'in yaptığı seçimin, elle yazılmış hâli.

> `read(...)`'in `HttpMessageConverterExample`'daki `ObjectMapper` kullanımıyla, `write(...)`'ın da aynı örnekteki JSON dalıyla birebir aynı olduğuna dikkat et -- bu mini proje yeni bir mekanizma icat etmiyor, dersin başından beri gördüğümüz parçaları tek bir "converter seçimi" akışında birleştiriyor.