Path Variable'lar ve Request Parametreleri

@PathVariable, @RequestParam ve @RequestHeader ile bir HTTP isteğinin URL'sinden ve header'larından veri okumak; path variable ile query parametresi arasındaki fark.

Orta 40 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/path-variables-request-parameters

---

Path Variable'lar ve Request Parametreleri

Mapping Annotation'ları ve HTTP Metotları dersinde `@PathVariable`'ı yalnızca örneği gerçekçi tutmak için, ayrıntısına girmeden kullanmıştık. Bu derste bir isteğin **URL'sinden** (hem path'in kendisinden hem query string'inden) ve **header'larından** veri okumanın tüm yollarına giriyoruz -- Spring MVC Temelleri dersinin "HandlerMapping ve HandlerAdapter: DispatcherServlet'in İçinde Neler Oluyor?" bölümünde bıraktığımız sözü de burada tutuyoruz: gerçek `HandlerAdapter`'ın her parametre türünü isteğin doğru parçasından nasıl okuyup doldurduğunu, son mini projede kendi elimizle inşa edeceğiz.

URL Mapping Desenleri Nedir?

Bir URL'nin path kısmı iki tür bilgi taşıyabilir: **sabit (literal) segmentler** (`/products`) ve **değişken segmentler** (`{id}` gibi, süslü parantezle işaretli). Bu ikisinin birleşimi bir URL mapping deseni oluşturur:

```
/users              -- sabit, hiçbir değişken yok
/users/{id}         -- bir değişken segment
/users/search       -- sabit, {id} ile karışmıyor (bkz. bir önceki dersteki
                       "Sınıf ve Metot Seviyesinde @RequestMapping'i Birleştirmek")
```

Değişken segmentlerdeki değeri okumanın yolu `@PathVariable`; path'in **dışında**, `?` işaretinden sonra gelen kısımdaki (query string) değerleri okumanın yolu ise `@RequestParam`'dır -- bu derste ikisini de ayrıntısıyla göreceğiz.

Neden Var?

Path variable'lar ve request parametreleri olmasaydı, her farklı `id` için ayrı bir mapping tanımlaman gerekirdi -- `/users/1`, `/users/2`, `/users/3` için üç ayrı `@GetMapping`. Bunun yerine `{id}` gibi bir yer tutucu, **sonsuz sayıda** URL'yi tek bir mapping'e ve tek bir metoda bağlar; gerçek değer, method çağrıldığında parametre olarak elde edilir. Aynı mantık query string için de geçerli: `?page=1`, `?page=2`, `?page=3` için ayrı mapping yazmak yerine, `page` parametresi tek bir metotta okunur.

Tarihçe

Spring MVC Temelleri dersinin "Tarihçe" bölümünde bahsettiğimiz gibi, Spring 3.0 (2009), `@PathVariable` ve `@RequestBody` / `@ResponseBody` ile REST tarzı endpoint'leri standartlaştırdı. `@RequestParam` daha da öncesine, Spring 2.5'e (2007) kadar gider -- `@RequestMapping` / `@Controller` ile aynı sürümde, form-tabanlı web uygulamalarının (HTML formlarının `GET` / `POST` ile query string ya da form-encoded body göndermesi) ihtiyacına yanıt olarak geldi. `@RequestHeader` de aynı dönemde eklendi.

@PathVariable: URL'den Değer Okumak

En temel kullanım: path'teki bir `{placeholder}`'ı, aynı isimli bir metot parametresiyle eşlemek:

```
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;

// @PathVariable pulls a value straight out of the URL's own path -- the {id} segment
// in the mapping and the `id` parameter are linked by name.
@Controller
class ProductController {

    @GetMapping("/products/{id}")
    @ResponseBody
    public String getProduct(@PathVariable Long id) {
        return "Product #" + id;
    }
}
```

`getProduct(Long id)`'daki `id`, mapping'deki `{id}` ile isim eşleşmesiyle bağlanıyor -- Spring, path'ten okuduğu `String` değerini otomatik olarak `Long`'a çeviriyor; bu dönüşümün nasıl çalıştığını ve başarısız olduğunda ne olduğunu "Tip Dönüşümü ve Hatalı Değerler: 400 Bad Request" bölümünde göreceğiz.

Birden Fazla Path Variable

Bir path, birden fazla değişken segment taşıyabilir -- her biri kendi metot parametresine bağlanır:

```
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;

// A path can carry more than one variable -- each {placeholder} becomes its own
// method parameter, matched by name.
@Controller
class OrderItemController {

    @GetMapping("/users/{userId}/orders/{orderId}")
    @ResponseBody
    public String getOrder(@PathVariable Long userId, @PathVariable Long orderId) {
        return "Order #" + orderId + " belonging to user #" + userId;
    }
}
```

`{userId}` ve `{orderId}`, sırasıyla `userId` ve `orderId` parametrelerine bağlanıyor -- eşleme isimle yapıldığı için parametrelerin metot imzasındaki sırası, path'teki sırasıyla aynı olmak zorunda değildir (yine de okunabilirlik için aynı sırada tutmak iyi bir alışkanlıktır).

Path Variable Adını Eşlemek: value Attribute'u

Metot parametresinin adı, `{placeholder}` ile birebir aynı olmak zorunda değil -- `@PathVariable`'ın `value` attribute'u hangi placeholder'a bağlanacağını açıkça belirtir:

```
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;

// When the method parameter's name doesn't match the {placeholder}, or when the code
// is compiled without the -parameters flag (so parameter names aren't available at
// runtime), @PathVariable's `value` tells Spring explicitly which placeholder to bind.
@Controller
class ArticleController {

    @GetMapping("/articles/{articleSlug}")
    @ResponseBody
    public String getArticle(@PathVariable("articleSlug") String slug) {
        return "Article: " + slug;
    }
}
```

`slug` parametresi, `{articleSlug}` placeholder'ına `@PathVariable("articleSlug")` ile açıkça bağlanıyor. Bu yalnızca isim tercihi meselesi değil -- kod `-parameters` derleyici bayrağı

olmadan derlenirse, metot parametrelerinin gerçek isimleri çalışma zamanında hiç mevcut olmaz; `value` bu durumda **zorunlu** hale gelir.

Path Variable mi, Query Parameter mı? Ne Zaman Hangisi

Path variable ile query parametresi arasındaki fark, sözdiziminden daha derinde bir anlam farkıdır: path variable bir kaynağı **kimliklendirir** (onsuz istek anlamsızdır), query parametresi zaten geçerli bir isteği **filtreler/daraltır**:

```
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.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

// A path variable IDENTIFIES a resource -- without it, there's no request to make.
// A query parameter FILTERS/MODIFIES a request that's already valid on its own.
@Controller
class ArticleListController {

    // No {id} here -- this endpoint is valid with zero query parameters too.
    @GetMapping("/articles")
    @ResponseBody
    public String list(@RequestParam(required = false) String category) {
        return category == null ? "All articles" : "Articles in category: " + category;
    }

    // {id} is required -- there is no "get one article" without knowing which one.
    @GetMapping("/articles/{id}")
    @ResponseBody
    public String getOne(@PathVariable Long id) {
        return "Article #" + id;
    }
}
```

`/articles/{id}` -- `id` olmadan "tek bir makaleyi getir" isteğinin hiçbir anlamı yok, bu yüzden path variable. `/articles?category=...` -- `category` olmadan da "tüm makaleleri listele" isteği geçerli, `category` yalnızca sonucu daraltıyor, bu yüzden query parametresi. Bu ayrımı doğru yapmak, URL'lerin okunabilir ve önbelleklenebilir kalmasını sağlar.

@RequestParam: Query String'den Değer Okumak

`@RequestParam`, `@PathVariable` ile aynı isim-eşleme mantığını, path yerine query string üzerinde uygular:

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

// @RequestParam reads a value from the query string -- ?page=2 becomes the `page`
// parameter, matched by name just like @PathVariable.
@Controller
class UserListController {

    @GetMapping("/users")
    @ResponseBody
    public String list(@RequestParam int page) {
        return "Showing page " + page;
    }
}
```

`?page=2` isteği, `page` parametresine `2` (otomatik `int`'e çevrilmiş olarak) bağlanıyor. `@PathVariable`'dan farklı olarak, `@RequestParam` **varsayılan olarak zorunludur** -- `page` hiç gönderilmezse, controller metodu hiç çağrılmaz, istemci `400 Bad Request` alır.

Zorunlu, Opsiyonel ve Varsayılan Değerli Parametreler

Bir önceki bölümdeki "varsayılan olarak zorunlu" davranışı, `required` ve `defaultValue` ile değiştirilebilir:

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

// By default, @RequestParam is REQUIRED -- a missing query parameter is a 400 Bad
// Request, not a null value. `required = false` and `defaultValue` change that.
@Controller
class SearchController {

    @GetMapping("/search")
    @ResponseBody
    public String search(
            @RequestParam String query,
            @RequestParam(required = false) String sortBy,
            @RequestParam(defaultValue = "20") int limit) {
        return "Searching \"" + query + "\", sortBy=" + sortBy + ", limit=" + limit;
    }
}
```

`query` zorunlu (`?query=` olmadan `400`); `sortBy` , `required = false` sayesinde opsiyonel (verilmezse `null`); `limit` , `defaultValue = "20"` sayesinde hem opsiyonel hem de verilmediğinde `null` yerine anlamlı bir değere sahip. `defaultValue` verildiğinde `required`'ı ayrıca belirtmeye gerek yoktur -- bir varsayılan değeri olan bir parametre zaten örtük olarak opsiyoneldir.

Birden Fazla Değerli Parametreler: List ve Array

Bir query string, aynı anahtarı birden fazla kez taşıyabilir -- bunu bir `List`'e (ya da diziye) bağlamak, tek bir parametrenin birden fazla değer taşımasını sağlar:

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

import java.util.List;

// A query string can repeat the same key (?tag=java&tag=spring) -- binding that to a
// List lets a single parameter carry multiple values.
@Controller
class ArticleFilterController {

    @GetMapping("/articles/by-tag")
    @ResponseBody
    public String filterByTags(@RequestParam List<String> tag) {
        return "Filtering by tags: " + tag;
    }
}
```

`?tag=java&tag=spring` isteği, `tag` parametresine `["java", "spring"]` listesini bağlıyor. Bu, "Path Variable mi, Query Parameter mı? Ne Zaman Hangisi" bölümünde gördüğümüz filtreleme senaryosunun doğal bir uzantısı -- "yalnızca bu etiketlerden birine sahip olanları göster" gibi çoklu-seçim filtreleri için idealdir.

Tüm Query Parametrelerini Almak: Map<String, String>

Bazen parametre isimlerini önceden bilmek mümkün olmaz -- `@RequestParam`'ı bir `Map`'e bağlamak, istekte bulunan **her** query parametresini, ismi ne olursa olsun, yakalar:

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

import java.util.Map;

// Sometimes you don't know the query parameter names in advance -- binding to a Map
// captures every query parameter present on the request, whatever its name.
@Controller
class FlexibleFilterController {

    @GetMapping("/reports")
    @ResponseBody
    public String report(@RequestParam Map<String, String> allParams) {
        return "Received filters: " + allParams;
    }
}
```

`allParams`, `?status=active&region=eu` gibi önceden bilinmeyen sayıda ve isimde parametre içeren bir isteği tek bir `Map<String, String>`'e topluyor. Bu esneklik bir bedel karşılığında gelir: derleme zamanında hangi parametrelerin var olduğunu kontrol edemezsin, tip dönüşümü de (hepsi `String` olarak gelir) elle yapılmalıdır.

@RequestHeader: HTTP Header'larını Okumak

`@RequestParam`'ın query string için yaptığını, `@RequestHeader` HTTP header'ları için yapar:

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

// @RequestHeader reads a value from the HTTP request headers, the same way
// @RequestParam reads from the query string -- required by default, with the same
// `required`/`defaultValue` options.
@Controller
class ClientInfoController {

    @GetMapping("/whoami")
    @ResponseBody
    public String whoAmI(
            @RequestHeader("User-Agent") String userAgent,
            @RequestHeader(value = "X-Request-Id", required = false) String requestId) {
        return "User-Agent: " + userAgent + ", X-Request-Id: " + requestId;
    }
}
```

`User-Agent` zorunlu (her tarayıcı/istemci zaten gönderir), `X-Request-Id` ise `required = false` ile opsiyonel bırakıldı -- özel bir header her istemcide bulunmayabilir. Header isimleri (`"User-Agent"` gibi) genelde tire içerdiği için `value` attribute'u burada neredeyse her zaman zorunludur; Java metot parametre isimlerinde tire kullanılamaz.

Tip Dönüşümü ve Hatalı Değerler: 400 Bad Request

`@PathVariable` / `@RequestParam` / `@RequestHeader`'ın hepsi, HTTP isteğinden ham bir `String` olarak gelir -- `Long`, `int`, `boolean` gibi bir tipe dönüştürülmesi Spring'in `ConversionService`'i tarafından yapılır:

```
import org.springframework.core.convert.ConversionException;
import org.springframework.core.convert.support.DefaultConversionService;

// Every @PathVariable/@RequestParam value arrives as a String -- Spring converts it to
// the declared parameter type (Long, int, boolean...) using the same kind of
// ConversionService machinery shown here directly. When conversion fails, real Spring
// MVC turns it into a 400 Bad Request before your controller method is ever called.
class TypeConversionErrorExample {
    public static void main(String[] args) {
        DefaultConversionService conversionService = new DefaultConversionService();

        Long id = conversionService.convert("42", Long.class);
        System.out.println("Converted: " + id);
        // Converted: 42

        try {
            conversionService.convert("abc", Long.class);
        } catch (ConversionException e) {
            System.out.println("Conversion failed, just like a real request to /products
/abc would fail");
            // Conversion failed, just like a real request to /products/abc would fail
        }
    }
}
```

`"42"` sorunsuz `Long`'a çevriliyor. `"abc"` ise çevrilemiyor ve bir `ConversionException` fırlatıyor -- gerçek bir Spring MVC isteğinde bu tam olarak `GET /products/abc` gibi bir isteğin (bkz. "@PathVariable: URL'den Değer Okumak") neden `400 Bad Request` ile sonuçlandığının nedeni: dönüşüm, controller metodun çağrılmasından **önce**, DispatcherServlet katmanında başarısız olur -- metodun kendisi hiç çalışmaz.

Bu Projenin Kendi Path Variable'ı ve Query Parametresi: Gerçek Bir Örnek

Bu dersteki mekanizmaları, projenin kendi `TopicController`'ında görebilirsin -- ve gerçek kodu, bu ayrımın zamanla nasıl değiştiğinin de güzel bir örneği:

```
@GetMapping("/{lang:en|tr}/topics/{slug}")
public String show(@PathVariable String lang, @PathVariable String slug, Model model) {
    ...
}

@GetMapping("/topics/{slug}")
public ResponseEntity<Void> legacyRedirect(@PathVariable String slug,
                                           @RequestParam(required = false) String lang) {
    ...
}
```

`slug`, her iki metotta da "Path Variable mi, Query Parameter mı? Ne Zaman Hangisi" bölümündeki ayrımın birebir örneği -- `slug` olmadan "bu konuyu göster" isteğinin anlamı yok, bu yüzden her zaman path variable. `lang` ise daha ilginç: `show(...)`'da artık **o da** bir path variable, çünkü SEO gerekçesiyle her sayfanın dil başına kararlı, taranabilir bir URL'e ihtiyacı var (`/en/topics/{slug}` ve `/tr/topics/{slug}`, üstüne isteğe bağlı bir değiştirici eklenmiş TEK bir sayfa değil, BAĞIMSIZ olarak indexlenebilen iki ayrı sayfa) -- bu da `lang`'i artık kaynağın kimliğinin bir parçası yapıyor, üzerine eklenen bir filtre değil. `lang`'in hâlâ gerçekten isteğe bağlı bir `@RequestParam` olduğu tek yer `legacyRedirect(...)` -- bu metot yalnızca sitenin eski `/topics/{slug}?lang=..` URL'lerini yeni path'e 301 ile yönlendirmek için var; orada `lang` olmadan da istek hâlâ tamamen anlamlı (sadece İngilizce'ye düşüyor), yani ayrımın tarif ettiği "isteğe bağlı filtre/değiştirici" durumunun ta kendisi. `HomeController` da aynı ayrımı iki metodu arasında yansıtıyor: gerçekten bir sayfa render eden `index(...)` (`/{lang:en|tr}` mapping'i) yalnızca `@PathVariable String lang` alıyor; çıplak `/`'i `/en` ya da `/tr`'ye 302 ile yönlendiren `root(...)` ise `legacyRedirect(...)` ile aynı gerekçeyle aynı isteğe bağlı `@RequestParam(required = false) String lang`'i alıyor -- eski `/?lang=..` yer imlerine bir nezaket, isteğin kesinlikle ihtiyaç duyduğu bir şey değil.

Best Practices

- **Bir değer, kaynağın kimliğinin parçasıysa path variable, isteğe bağlı bir filtre /değiştiriciyse query parametresi olsun** -- "Path Variable mi, Query Parameter mı? Ne Zaman Hangisi" bölümündeki ayrımı tutarlı uygulamak, API'nin URL'lerini okunabilir ve önbelleklenebilir tutar.
- **Gerçekten opsiyonel olan her parametrede `required = false` ya da `defaultValue` kullan** -- aksi halde, "Zorunlu, Opsiyonel ve Varsayılan Değerli Parametreler" bölümünde gördüğümüz gibi, istemcinin göndermeyi unuttuğu her parametre bir `400`'e dönüşür.
- **`@RequestParam Map<String, String>`'i yalnızca gerçekten dinamik/önceden bilinmeyen parametreler için kullan** -- bilinen parametreleri ayrı ayrı bildirmek (bkz. "Zorunlu, Opsiyonel ve Varsayılan Değerli Parametreler"), tip güvenliği ve okunabilirlik sağlar.
- **Path variable isimlerini `{placeholder}` ile birebir aynı tutmayı tercih et, farklıysa her zaman `value` attribute'unu açıkça yaz** -- "Path Variable Adını Eşlemek: value Attribute'u" bölümünde gördüğümüz gibi, `-parameters` bayrağına güvenmek kırılgan bir varsayımdır.

Yaygın Hatalar

1. `@RequestParam`'ın da `@PathVariable` gibi varsayılan olarak opsiyonel olduğunu sanmak. Tam tersi -- `@RequestParam` varsayılan olarak **zorunludur**; opsiyonel olması için açıkça `required = false` ya da `defaultValue` gerekir (bkz. "@RequestParam: Query String'den Değer Okumak").

2. Bir path variable ile bir query parametresini birbirinin yerine kullanmak (örn. `/articles?id=5` yazıp `/articles/5` yerine). İkisi teknik olarak da çalışabilir, ama "Path Variable mi, Query Parameter mı? Ne Zaman Hangisi" bölümündeki anlam ayrımını çiğner -- kaynağı kimliklendiren bir değer path'te olmalıdır.

3. Tip dönüşümü hatasını (`400 Bad Request`) bir uygulama hatası sanıp controller'ın içine try/catch eklemeye çalışmak. Dönüşüm, controller metodu hiç çağrılmadan, DispatcherServlet katmanında gerçekleşir -- metodun içine bir try/catch koymak bu hatayı asla yakalamaz (bkz. "Tip Dönüşümü ve Hatalı Değerler: 400 Bad Request").

4. Header isimlerini Java metot parametre adı gibi yazıp `value` belirtmeyi unutmak (örn. `@RequestHeader String userAgent`, `"User-Agent"` yerine). Header isimlerindeki tire, Java tanımlayıcılarında geçerli değildir -- `value` olmadan Spring, `userAgent` adlı bir header arar, bulamaz ve (varsayılan olarak zorunlu olduğu için) `400` döner (bkz. "@RequestHeader: HTTP Header'larını Okumak").

5. `@RequestParam List<String>` beklerken istemcinin `?tag=java,spring` gibi tek bir parametrede virgülle ayrılmış değer göndermesini beklemek. Spring'in `List` bağlaması, aynı anahtarın **tekrarlanmasını** (`?tag=java&tag=spring`) bekler, tek bir değerin virgülle bölünmesini değil (bkz. "Birden Fazla Değerli Parametreler: List ve Array").

Özet, Cheat Sheet ve Terimler Sözlüğü

Path variable'lar ve request parametreleri, bir HTTP isteğinin URL'sinden (path ve query string) ve header'larından veri okumanın annotation tabanlı yoludur. Öne çıkan noktalar:

- `@PathVariable`: path'teki bir `{placeholder}`'ı okur; kaynağı kimliklendiren değerler için kullanılır
- `@RequestParam`: query string'den bir değeri okur; varsayılan olarak zorunludur (`required = false` / `defaultValue` ile opsiyonel yapılabilir)
- `@RequestHeader`: bir HTTP header'ını okur; aynı zorunlu/opsiyonel kurallarına sahiptir
- `List` / `Map` bağlama: `@RequestParam List<String>` tekrarlanan anahtarları, `@RequestParam Map<String, String>` bilinmeyen sayıda parametreyi toplar
- Tip dönüşümü, `ConversionService` tarafından, controller metodu çağrılmadan **önce** yapılır -- başarısız olursa `400 Bad Request`
- Path variable = kaynağı kimliklendirir (zorunlu); query parametresi = isteği filtreler (genelde opsiyonel)

Hızlı referans:

```
@GetMapping("/users/{id}")
String getOne(@PathVariable Long id) { ... }                    // path variable

@GetMapping("/articles/{articleSlug}")
String getArticle(@PathVariable("articleSlug") String slug) { ... }  // isim eşleme

@GetMapping("/search")
String search(
    @RequestParam String query,                                  // zorunlu
    @RequestParam(required = false) String sortBy,               // opsiyonel
    @RequestParam(defaultValue = "20") int limit,                // varsayılan değerli
    @RequestParam(required = false) List<String> tag,            // çoklu değer
    @RequestParam Map<String, String> allParams,                 // tüm parametreler
    @RequestHeader("User-Agent") String userAgent                // header
) { ... }
```

Terimler Sözlüğü

Path variable — Bir URL path'indeki `{placeholder}` segmentinden okunan, bir kaynağı kimliklendiren değer.

Query parametresi — Bir URL'nin `?` sonrasındaki kısmından (query string) okunan, isteği filtreleyen/değiştiren değer.

`@PathVariable` — Bir metot parametresini, path'teki bir `{placeholder}`'a bağlayan annotation.

`@RequestParam` — Bir metot parametresini, query string'deki (ya da form body'sindeki) bir değere bağlayan, varsayılan olarak zorunlu annotation.

`@RequestHeader` — Bir metot parametresini bir HTTP header değerine bağlayan annotation.

`ConversionService` — Bir `String`'i hedef Java tipine (Long, int, boolean...) çeviren Spring bileşeni; `@PathVariable` / `@RequestParam` / `@RequestHeader`'ın hepsi bunu kullanır.

400 Bad Request — Zorunlu bir parametre eksik olduğunda ya da tip dönüşümü başarısız olduğunda dönen HTTP durum kodu.

Ek: Mini Proje — Katalog Arama API'si

Bu dersteki her mekanizmayı, gerçekçi tek bir arama endpoint'inde bir araya getiriyoruz:

```
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.RequestHeader;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.List;

// A realistic search endpoint over a single category (path variable, identifies the
// resource collection), narrowed by optional query parameters (filters) and aware of
// the requesting client (header) -- every mechanism from this lesson, together.
@Controller
class CatalogSearchController {

    @GetMapping("/catalog/{category}/search")
    @ResponseBody
    public String search(
            @PathVariable String category,
            @RequestParam String query,
            @RequestParam(required = false) List<String> tag,
            @RequestParam(defaultValue = "10") int limit,
            @RequestHeader(value = "Accept-Language", required = false) String language) {
        return "Searching \"" + query + "\" in category=" + category
                + ", tags=" + tag + ", limit=" + limit + ", language=" + language;
    }
}
```

```
import java.util.List;

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

        System.out.println(controller.search("books", "spring", List.of("java", "web"), 10,
"en"));
        // Searching "spring" in category=books, tags=[java, web], limit=10, language=en

        System.out.println(controller.search("electronics", "headphones", null, 10, null));
        // Searching "headphones" in category=electronics, tags=null, limit=10,
language=null
    }
}
```

`category`, "Path Variable mi, Query Parameter mı? Ne Zaman Hangisi" bölümündeki ayrımı izleyerek path variable (kategori olmadan arama isteğinin bağlamı yok); `query` / `tag` / `limit` ise query parametresi (aynı kategori içinde farklı şekillerde daraltılabilir isteğin kendisi zaten geçerli). `Accept-Language` header'ı, istemcinin dil tercihini -- tıpkı bu projenin kendi `lang` query parametresi gibi, ama HTTP'nin standart mekanizmasıyla -- taşıyor.

Ek: Mini Proje — Elle Yazılmış Bir Argüman Bağlayıcı (Argument Resolver) Simülasyonu

Son mini proje, Spring MVC Temelleri dersinde bıraktığımız sözü tutuyor: gerçek `HandlerAdapter`'ın, bir metodun her parametresini annotation'ına bakarak isteğin doğru parçasından nasıl doldurduğunu, elle inşa ediyoruz:

```
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestParam;

import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
import java.util.Map;

// Spring MVC Fundamentals' HandlerAdapter simulations could only invoke parameterless
// methods. This one does what a real HandlerAdapter's argument resolvers do: look at
// each parameter's annotation, pull the matching value out of the request, and pass it
// along when invoking the method.
class GreetingHandler {
    public String greet(@PathVariable("name") String name,
                        @RequestParam(defaultValue = "en") String lang,
                        @RequestHeader(value = "X-Client", required = false) String
client) {
        return "Hello " + name + " (lang=" + lang + ", client=" + client + ")";
    }
}

class RequestBinderSimulation {

    static Object invoke(Object handler, Method method,
                         Map<String, String> pathVariables,
                         Map<String, String> queryParams,
                         Map<String, String> headers) throws Exception {
        Parameter[] parameters = method.getParameters();
        Object[] args = new Object[parameters.length];

        for (int i = 0; i < parameters.length; i++) {
            Parameter parameter = parameters[i];

            PathVariable pathVar = parameter.getAnnotation(PathVariable.class);
            if (pathVar != null) {
                args[i] = pathVariables.get(pathVar.value());
                continue;
            }

            RequestParam requestParam = parameter.getAnnotation(RequestParam.class);
            if (requestParam != null) {
                String value = queryParams.get(parameter.getName());
                args[i] = value != null ? value : requestParam.defaultValue();
                continue;
            }

            RequestHeader requestHeader = parameter.getAnnotation(RequestHeader.class);
            if (requestHeader != null) {
                args[i] = headers.get(requestHeader.value());
            }
        }

        return method.invoke(handler, args);
    }
}
```

```
import java.lang.reflect.Method;
import java.util.Map;

class RequestBinderDemo {
    public static void main(String[] args) throws Exception {
        GreetingHandler handler = new GreetingHandler();
        Method method = GreetingHandler.class.getMethod("greet", String.class, String.
class, String.class);

        Object result = RequestBinderSimulation.invoke(
                handler,
                method,
                Map.of("name", "Ayse"),
                Map.of(),
                Map.of("X-Client", "web"));

        System.out.println(result);
        // Hello Ayse (lang=en, client=web)
    }
}
```

`RequestBinderSimulation.invoke(...)`, `greet(...)` metodunun her parametresini reflection ile dolaşıyor -- `@PathVariable` işaretliyse `pathVariables` map'inden, `@RequestParam` işaretliyse `queryParams`'tan (yoksa `defaultValue()`'dan), `@RequestHeader` işaretliyse `headers`'tan değeri okuyor, sonra metodu bu değerlerle çağırıyor. Bu, "Tip Dönüşümü ve Hatalı Değerler: 400 Bad Request" bölümünde gördüğümüz `ConversionService` adımı hariç, gerçek Spring'in her istekte arka planda yaptığı işin tam bir modeli.

> `RequestBinderSimulation`, `@RequestParam`'ın hangi query parametresine bağlanacağını `parameter.getName()` ile (Java'nın kendi reflection API'siyle) buluyor -- bu, yalnızca kod `-parameters` derleyici bayrağıyla derlendiğinde güvenilir çalışır; aksi halde parametre adları `arg0`, `arg1` gibi jenerik isimlere döner. "Path Variable Adını Eşlemek: value Attribute'u" bölümünde gördüğümüz gibi, gerçek kodda bu belirsizliğe güvenmek yerine her zaman `value` yazmak daha güvenlidir.