Mapping Annotation'ları ve HTTP Metotları

@RequestMapping ve beş kısayolu (@GetMapping/@PostMapping/@PutMapping /@PatchMapping/@DeleteMapping), consumes/produces ve HTTP metotlarının safe /idempotent semantiği.

Orta 35 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/mapping-annotations-http-methods

---

Mapping Annotation'ları ve HTTP Metotları

Spring MVC Temelleri dersinde `@GetMapping`'i yalnızca yüzeysel gördük -- bir path'i bir metoda bağlayan tek bir annotation olarak. Bu derste mapping annotation'larının tüm ailesine (`@RequestMapping` ve beş kısayoluna), her birinin hangi HTTP metoduna karşılık geldiğine ve HTTP metotlarının kendi anlamsal kurallarına (safe, idempotent) giriyoruz. "Aynı Path, Farklı HTTP Metotları" bölümünde, Fundamentals dersindeki `RequestRouterSimulation` mini projesinin uyarısında bıraktığımız eksiği -- HTTP metodu ayrımı -- de tamamlayacağız.

Mapping Annotation'ları Nedir?

Mapping annotation'ları, bir controller metodunun **hangi HTTP isteğine** yanıt vereceğini bildiren annotation'lardır -- bir path (`"/users"`), isteğe bağlı bir HTTP metodu, ve isteğe bağlı diğer koşullar (content type, header'lar) tanımlarlar:

```
@Controller
class UserController {
    @GetMapping("/users")       // path: /users, HTTP metodu: GET
    public String list() { ... }
}
```

Bu tanım, Spring MVC Temelleri dersinde gördüğümüz `HandlerMapping`'in okuduğu tam olarak bu bilgi -- "HandlerMapping ve HandlerAdapter: DispatcherServlet'in İçinde Neler Oluyor?" bölümündeki `buildHandlerMapping` simülasyonumuzun, gerçek Spring'de karşılık geldiği mekanizma.

Neden Var?

Mapping annotation'ları olmasaydı, DispatcherServlet'e "bu path'e, bu HTTP metoduyla gelen istek, şu metoda gitsin" bilgisini başka bir yerden (XML, elle yazılmış bir routing tablosu) vermek gerekirdi -- Spring MVC Temelleri dersinin "Tarihçe" bölümünde bahsettiğimiz, 2004'teki XML tabanlı `<bean>` eşlemeleri tam olarak buydu. Annotation'lar bu bilgiyi, metodun **kendi üzerinde**, kodun yanında tutar -- yeni bir endpoint eklemek, yeni bir metot yazıp üzerine annotation koymaktan ibarettir.

Tarihçe

Spring MVC Temelleri dersinin "Tarihçe" bölümünde bu ailenin genel zaman çizelgesini görmüştük: `@RequestMapping`, Spring 2.5 (2007) ile geldi; `@GetMapping` gibi kısayollar ise çok daha sonra, Spring 4.3 (2016) ile eklendi. Aradaki dokuz yılda geliştiriciler `@RequestMapping(method = RequestMethod.GET)` yazmak zorundaydı -- tekrarlayıcı ve `method` parametresini unutmaya açık (unutulduğunda mapping, **her** HTTP metodunu kabul eder, bu da "Desteklenmeyen Bir HTTP Metodu İstendiğinde: 405 Method Not Allowed" bölümünde göreceğimiz hatayı gizler). Spring 4.3, beş HTTP metodu için beş kısayol (`@GetMapping`, `@PostMapping`, `@PutMapping`, `@PatchMapping`, `@DeleteMapping`) ekleyerek bu tekrarı ortadan kaldırdı.

@RequestMapping: Temel Mapping Annotation'ı

Aile ağacının kökü `@RequestMapping`'dir -- `method` attribute'u ile herhangi bir HTTP metodunu (ya da hiçbirini belirtmeyip hepsini) eşleyebilir:

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

// @RequestMapping is the original, most general mapping annotation -- it can match
// any HTTP method (or several at once) via its `method` attribute. Every shortcut
// annotation we'll see next (@GetMapping, @PostMapping...) is built on top of this one.
@Controller
class RawRequestMappingController {

    @RequestMapping(path = "/ping", method = RequestMethod.GET)
    @ResponseBody
    public String ping() {
        return "pong";
    }

    // Without a `method`, @RequestMapping matches EVERY HTTP method on this path --
    // rarely what you want, but useful to know it's the default.
    @RequestMapping(path = "/any-method")
    @ResponseBody
    public String anyMethod() {
        return "matched regardless of HTTP method";
    }
}
```

`method` verilmediğinde, `@RequestMapping` path'e gelen **her** HTTP metodunu kabul eder -- `anyMethod()` hem `GET` hem `POST` hem `DELETE` isteğine yanıt verir. Bu nadiren istenen bir davranıştır; "HTTP Metodu ile CRUD İşlemleri Arasındaki Eşleme" bölümünde göreceğimiz gibi, her HTTP metodunun kendine has bir anlamı vardır ve bunu belirsiz bırakmak genelde bir tasarım hatasıdır.

@GetMapping, @PostMapping ve Diğer Kısayollar

Beş kısayol annotation'ı, `@RequestMapping`'in üzerine kurulu birer meta-annotation'dır -- her biri `method` attribute'unu senin için önceden doldurur:

```
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.PatchMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.ResponseBody;

// Each shortcut is a meta-annotation: @GetMapping is exactly
// @RequestMapping(method = RequestMethod.GET), just shorter and more readable at a
// glance. A typical resource controller uses one of each, one per operation.
@Controller
@ResponseBody
class UserShortcutController {

    @GetMapping("/users")
    public String listUsers() {
        return "list of users";
    }

    @PostMapping("/users")
    public String createUser() {
        return "user created";
    }

    @PutMapping("/users/1")
    public String replaceUser() {
        return "user replaced";
    }

    @PatchMapping("/users/1")
    public String updateUser() {
        return "user partially updated";
    }

    @DeleteMapping("/users/1")
    public String deleteUser() {
        return "user deleted";
    }
}
```

`@GetMapping("/users")`, tam olarak `@RequestMapping(path = "/users", method = RequestMethod.GET)` ile aynı şeydir -- yalnızca daha kısa ve niyeti ilk bakışta netleştiriyor. Gerçek bir kaynak (`resource`) controller'ı, tipik olarak bu beşinden birer tane taşır -- listeleme, oluşturma, tam güncelleme, kısmi güncelleme, silme.

Sınıf ve Metot Seviyesinde @RequestMapping'i Birleştirmek

`@RequestMapping`, sınıf seviyesinde kullanıldığında bir **ortak önek** tanımlar -- sınıf içindeki her metot, kendi path'ini bu önekin devamı olarak tanımlar:

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

// @RequestMapping at the class level sets a COMMON PREFIX for every method inside --
// exactly the pattern this project's own TopicController used to follow
// (@RequestMapping("/topics") on the class, @GetMapping("/{slug}") on the method),
// before it later grew a second mapping shape that no longer shared that prefix
// (see "This Project's Own Mappings: A Real Example" in the parent lesson).
// @PathVariable is used here just to keep the example realistic; we'll cover it in
// full in the next lesson (Path Variables & Request Parameters).
@Controller
@RequestMapping("/users")
class UserController {

    @GetMapping
    @ResponseBody
    public String list() {
        return "GET /users";
    }

    @GetMapping("/{id}")
    @ResponseBody
    public String getOne(@PathVariable Long id) {
        return "GET /users/" + id;
    }

    @GetMapping("/search")
    @ResponseBody
    public String search() {
        return "GET /users/search";
    }
}
```

Bu, bu projenin kendi `TopicController`'ının kullandığı desenin ta kendisi -- Spring MVC Temelleri dersinin "Bu Projenin Kendi Controller'ları: Gerçek Bir Spring MVC Örneği" bölümünde bıraktığımız sözü burada tutuyoruz; ayrıntısını "Bu Projenin Kendi Mapping'leri: Gerçek Bir Örnek" bölümünde göreceğiz. `search()` metodunun path'i (`/users/search`), `getOne()`'ın path variable'lı path'iyle (`/users/{id}`) çakışmaz -- Spring'in path eşleştirmesi, **sabit (literal) segmentleri her zaman değişken segmentlerden daha spesifik sayar** , tanımlama sırası önemli değildir.

Content Type Belirtmek: consumes ve produces

Bir mapping, yalnızca path ve HTTP metoduyla değil, **hangi içerik türünü kabul ettiği /ürettiği** ile de daraltılabilir:

```
import org.springframework.http.MediaType;
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;

// consumes/produces narrow a mapping to specific Content-Type/Accept headers -- the
// SAME path and HTTP method can be mapped twice, once per representation. @RequestBody
// is used here just to keep the example realistic; it's covered in full in a later
// lesson (Request & Response Handling).
@Controller
class ContentNegotiatingController {

    @PostMapping(path = "/orders", consumes = MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
    @ResponseBody
    public String createFromJson(@RequestBody String body) {
        return "{\"status\":\"created from JSON\"}";
    }

    @PostMapping(path = "/orders", consumes = MediaType.APPLICATION_XML_VALUE, produces =
MediaType.APPLICATION_XML_VALUE)
    @ResponseBody
    public String createFromXml(@RequestBody String body) {
        return "<status>created from XML</status>";
    }
}
```

Aynı path (`/orders`) ve aynı HTTP metodu (`POST`) burada **iki kez** tanımlı -- `consumes` / `produces` sayesinde çakışmıyorlar, çünkü DispatcherServlet, isteğin `Content-Type` / `Accept` header'larına bakarak hangisinin devreye gireceğine karar verir. `@RequestBody` burada örneği gerçekçi tutmak için kullanıldı; kendisini ayrıntısıyla ileriki bir derste (Request & Response Handling) göreceğiz.

HTTP Metotları: Safe ve Idempotent Kavramları

HTTP spesifikasyonu, her metoda iki önemli özellik atfeder: **safe** (sunucu durumunu değiştirmemeli) ve **idempotent** (bir kez ya da yüz kez çağırmak, sonucu aynı bırakmalı). `GET` 'in ikisi de olması **zorunludur**:

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

// "Safe" means an HTTP method must not change server state; "idempotent" means
// calling it once or a hundred times leaves the server in the same state as calling
// it exactly once. GET is required to be both; POST is neither.
@Controller
class ViewCounterController {
    private int views = 0;

    @GetMapping("/article")
    @ResponseBody
    public String viewArticle() {
        // Safe: reading the article never changes `views`.
        return "Article content (viewed " + views + " times so far)";
    }

    @PostMapping("/article/views")
    @ResponseBody
    public String recordView() {
        // Not safe, not idempotent: every call increments the counter further.
        views++;
        return "Recorded. Total views: " + views;
    }
}

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

        System.out.println(controller.viewArticle());
        // Article content (viewed 0 times so far)
        System.out.println(controller.viewArticle());
        // Article content (viewed 0 times so far) -- safe: calling GET changed nothing

        System.out.println(controller.recordView());
        // Recorded. Total views: 1
        System.out.println(controller.recordView());
        // Recorded. Total views: 2 -- not idempotent: state changed again
    }
}
```

`viewArticle()` (GET) kaç kez çağrılırsa çağrılsın `views` değişmiyor -- safe. `recordView()` (POST) ise her çağrıda durumu değiştiriyor -- ne safe ne idempotent. Bu ayrım, "PUT vs PATCH: Tam Güncelleme vs Kısmi Güncelleme" ve "DELETE ve Idempotency" bölümlerinde her metodu tek tek değerlendirirken referans noktamız olacak.

Aynı Path, Farklı HTTP Metotları

Spring MVC Temelleri dersinin son mini projesindeki uyarıyı hatırla: `RequestRouterSimulation`, yalnızca path'e bakıyordu, aynı path'e farklı HTTP metotlarıyla gelen istekleri ayıramıyordu. Bunu şimdi düzeltiyoruz:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMethod;

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

// The Fundamentals lesson's FrontControllerSimulationExample only kept a
// path -> method registry -- it couldn't tell a GET handler apart from a POST handler
// on the same path. This version adds the HTTP method into the registry key, exactly
// what real Spring's HandlerMapping does.
class ArticleHandlers {
    @GetMapping("/article")
    public String view() {
        return "Viewing article";
    }

    @PostMapping("/article")
    public String publish() {
        return "Publishing article";
    }
}

class HttpMethodDisambiguationExample {

    private record RouteKey(String path, RequestMethod method) {
    }

    static Map<RouteKey, Method> buildRegistry(Object handler) {
        Map<RouteKey, Method> registry = new HashMap<>();
        for (Method method : handler.getClass().getDeclaredMethods()) {
            GetMapping get = method.getAnnotation(GetMapping.class);
            if (get != null) {
                registry.put(new RouteKey(get.value()[0], RequestMethod.GET), method);
            }
            PostMapping post = method.getAnnotation(PostMapping.class);
            if (post != null) {
                registry.put(new RouteKey(post.value()[0], RequestMethod.POST), method);
            }
        }
        return registry;
    }

    static String dispatch(String path, RequestMethod httpMethod, Object handler,
Map<RouteKey, Method> registry) throws Exception {
        Method method = registry.get(new RouteKey(path, httpMethod));
        if (method == null) {
            return "405 Method Not Allowed: " + httpMethod + " " + path;
```

```
        }
        return (String) method.invoke(handler);
    }

    public static void main(String[] args) throws Exception {
        ArticleHandlers handler = new ArticleHandlers();
        Map<RouteKey, Method> registry = buildRegistry(handler);

        System.out.println(dispatch("/article", RequestMethod.GET, handler, registry));
        // Viewing article
        System.out.println(dispatch("/article", RequestMethod.POST, handler, registry));
        // Publishing article
        System.out.println(dispatch("/article", RequestMethod.DELETE, handler, registry));
        // 405 Method Not Allowed: DELETE /article
    }
}
```

`RouteKey` artık yalnızca path değil, `(path, method)` **çiftini** anahtar olarak kullanıyor -- `/article`'a gelen bir `GET`, `view()`'a; aynı path'e gelen bir `POST`, `publish()`'e gidiyor. Eşleşen bir `(path, method)` çifti yoksa (`DELETE /article` gibi), gerçek Spring'in de döneceği yanıt tam olarak bunun karşılığı: **405 Method Not Allowed** -- 404 değil, çünkü path'in kendisi var, sadece o HTTP metodunda değil.

PUT vs PATCH: Tam Güncelleme vs Kısmi Güncelleme

İkisi de "güncelleme" anlamına gelir, ama farklı sözleşmelerle: `PUT`, kaynağın **tamamını** yeni haliyle değiştirir (gönderilmeyen alanlar kaybolur); `PATCH`, yalnızca gönderilen alanları günceller:

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PatchMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.LinkedHashMap;
import java.util.Map;

// PUT replaces the ENTIRE resource -- every field must be supplied, missing fields
// are lost. PATCH updates only the fields that were actually sent, leaving the rest
// untouched.
@Controller
class UserProfileController {
    private final Map<String, String> profile = new LinkedHashMap<>();

    UserProfileController() {
        profile.put("name", "Ayse");
        profile.put("city", "Istanbul");
    }

    @PutMapping("/profile")
    @ResponseBody
    public Map<String, String> replace(@RequestBody Map<String, String> newProfile) {
        profile.clear();
        profile.putAll(newProfile); // anything not in newProfile is gone
        return new LinkedHashMap<>(profile); // a copy that keeps insertion order for
display
    }

    @PatchMapping("/profile")
    @ResponseBody
    public Map<String, String> update(@RequestBody Map<String, String> changes) {
        profile.putAll(changes); // only overwrites the given keys
        return new LinkedHashMap<>(profile); // a copy that keeps insertion order for
display
    }
}

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

        System.out.println(controller.update(Map.of("city", "Ankara")));
        // {name=Ayse, city=Ankara} -- PATCH: only "city" changed, "name" untouched

        System.out.println(controller.replace(Map.of("city", "Izmir")));
        // {city=Izmir} -- PUT: "name" is GONE, it wasn't in the replacement body
    }
}
```

`update()` (PATCH) yalnızca `city`'yi değiştiriyor, `name` dokunulmadan kalıyor. `replace()` (PUT) ise `profile.clear()` ile önce her şeyi siliyor, sonra yalnızca gönderilen alanları geri

koyuyor -- `name` gönderilmediği için tamamen kayboluyor. Bu karışıklık, API tasarımında en sık karşılaşılan hata kaynaklarından biri: PATCH isteği bekleyen bir istemcinin, yanlışlıkla PUT çağırıp diğer alanları silmesi.

DELETE ve Idempotency

`DELETE`, "HTTP Metotları: Safe ve Idempotent Kavramları" bölümündeki tanıma göre idempotent olmalıdır -- ama bunun ne anlama geldiği ilk bakışta göründüğünden daha ince bir noktadır:

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

import java.util.HashMap;
import java.util.Map;

// DELETE is idempotent in the sense that matters: no matter how many times you call
// it, the END STATE is the same (the resource is gone) -- even though the HTTP status
// code of the second call differs from the first.
@Controller
class BookDeletionController {
    private final Map<Long, String> books = new HashMap<>(Map.of(1L, "Effective Java"));

    @DeleteMapping("/books/{id}")
    @ResponseBody
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        if (books.remove(id) != null) {
            return ResponseEntity.noContent().build(); // 204: it was there, now it's gone
        }
        return ResponseEntity.status(HttpStatus.NOT_FOUND).build(); // 404: already gone
    }
}

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

        System.out.println(controller.delete(1L).getStatusCode());
        // 204 NO_CONTENT
        System.out.println(controller.delete(1L).getStatusCode());
        // 404 NOT_FOUND -- different status, but the end state (book 1 is gone) is
        // identical after either call, which is exactly what idempotency means.
    }
}
```

İlk `delete(1L)` çağrısı `204 No Content` döner (kitap gerçekten silindi), ikinci çağrı `404 Not Found` döner (kitap zaten yok) -- **iki farklı HTTP durum kodu**. Yine de idempotent'tir, çünkü idempotency HTTP durum kodunun aynı kalmasını değil, **sunucudaki nihai durumun** aynı kalmasını gerektirir -- her iki çağrıdan sonra da kitap 1 veritabanında yok.

HTTP Metodu ile CRUD İşlemleri Arasındaki Eşleme

Gördüğümüz beş HTTP metodu, CRUD (Create/Read/Update/Delete) işlemleriyle şu şekilde eşlenir:

- **GET** → Read (safe + idempotent) -- "HTTP Metotları: Safe ve Idempotent Kavramları" bölümünde gördüğümüz `viewArticle()`
- **POST** → Create (ne safe ne idempotent) -- her çağrı yeni bir kaynak yaratır ya da durumu değiştirir
- **PUT** → Update, tam değiştirme (idempotent, safe değil) -- aynı `PUT` isteğini iki kez göndermek, kaynağı ilk seferkiyle aynı son duruma getirir
- **PATCH** → Update, kısmi değiştirme (genelde idempotent kabul edilir, ama HTTP spesifikasyonu bunu garanti etmez -- "alanı 1 artır" gibi bir PATCH idempotent olmaz)
- **DELETE** → Delete (idempotent, safe değil) -- "DELETE ve Idempotency" bölümünde gördüğümüz gibi, durum kodu değişse de nihai durum sabit kalır

Desteklenmeyen Bir HTTP Metodu İstendiğinde: 405 Method Not Allowed

"Aynı Path, Farklı HTTP Metotları" bölümündeki simülasyonumuzun ürettiği `"405 Method Not Allowed"` mesajı, uydurma bir davranış değil -- gerçek DispatcherServlet'in yaptığı tam olarak bu: bir path için **en az bir** mapping bulunuyorsa ama istenen HTTP metoduyla eşleşen yoksa, 404 (path yok) değil, **405** (path var, bu metotla değil) döner. Bu ayrım önemlidir -- bir istemci 405 aldığında, path'i doğru yazdığını ama yanlış HTTP metodunu kullandığını anlayabilir; 404'te bu bilgiyi kaybeder.

Bu Projenin Kendi Mapping'leri: Gerçek Bir Örnek

Bu dersteki mekanizmaları, projenin kendi kaynak kodunda görebilirsin. `HomeController`, sınıf seviyesinde hiçbir `@RequestMapping` taşımaz -- iki endpoint'i (`@GetMapping("/")` ve `@GetMapping("/{lang:en|tr}")`) ortak bir literal önek paylaşmadığı için zaten paylaşılacak bir şey yok. `TopicController` ise bir zamanlar "Sınıf ve Metot Seviyesinde @RequestMapping'i Birleştirmek" bölümünün ders kitabı örneğiydi -- sınıf seviyesinde `@RequestMapping("/topics")`, metot seviyesinde `@GetMapping("/{slug}")` -- ta ki ikinci bir mapping şekli (eski URL'leri yönlendiren `/topics/{slug}`, gerçek `/{lang:en|tr}/topics/{slug}`'in yanına) eklenip artık ilkiyle tek bir sınıf-seviyesi önek paylaşamayana kadar. O noktada sınıf seviyesindeki `@RequestMapping` kaldırıldı, her metot artık kendi tam path'ini tanımlıyor -- paylaşılan bir önekin ne zaman kendini amorti etmekten çıktığına dair gerçek, küçük bir örnek. Her iki controller de yalnızca `GET` isteklerine yanıt verir -- bu proje şu an salt-okunur bir içerik sitesi olduğu için `POST` / `PUT` / `PATCH` / `DELETE` hiç kullanılmıyor; bu kategorinin sonraki konularında (Request & Response Handling, REST API Design) bu diğer metotları gerektirecek bir JSON API senaryosunu ele alacağız.

Best Practices

- **Her zaman en spesifik kısayolu kullan, çıplak `@RequestMapping`'i yalnızca gerçekten birden fazla HTTP metodu kabul etmen gerektiğinde tercih et** -- `method` belirtmeden bırakılan bir `@RequestMapping`, "Tarihçe" bölümünde bahsettiğimiz gibi her metodu sessizce kabul eder, bu genelde istenmeyen bir davranıştır.
- **HTTP metodunun anlamına sadık kal: GET'te veri değiştirme, DELETE'te idempotent ol** -- "HTTP Metotları: Safe ve Idempotent Kavramları" bölümündeki kurallara uymayan bir API, önbellekleme/retry gibi HTTP altyapısının varsaydığı davranışları bozar.
- **PUT ile PATCH'i birbirinin yerine kullanma** -- "PUT vs PATCH: Tam Güncelleme vs Kısmi Güncelleme" bölümünde gördüğümüz gibi, yanlış seçim istemeden veri kaybına yol açabilir.
- **Ortak bir path öneki olan endpoint'lerde sınıf seviyesinde `@RequestMapping` kullan** -- bu projenin kendi `TopicController`'ının yaptığı gibi (bkz. "Bu Projenin Kendi Mapping'leri: Gerçek Bir Örnek"), her metotta öneki tekrar yazmak yerine.

Yaygın Hatalar

1. `@RequestMapping`'e `method` yazmayı unutup, mapping'in yalnızca beklenen HTTP metoduna yanıt verdiğini sanmak. `method` verilmezse **her** HTTP metodu kabul edilir -- bu, yanlışlıkla bir `DELETE` isteğinin bir "salt okunur" endpoint'e ulaşmasına izin verebilir (bkz. "@RequestMapping: Temel Mapping Annotation'ı").

2. `/users/{id}` ile `/users/search` gibi bir literal path'in çakışacağını düşünüp, tanımlama sırasını değiştirerek "düzeltmeye" çalışmak. Spring, literal segmentleri her zaman değişken segmentlerden daha spesifik sayar -- sıralama hiç önemli değildir (bkz. "Sınıf ve Metot Seviyesinde @RequestMapping'i Birleştirmek").

3. GET ile veri değiştiren bir endpoint yazmak ("kolay test edilsin" diye tarayıcıdan tıklanabilir bir silme linki gibi). Bu, GET'in safe olması gerektiği kuralını çiğner -- bir önbellek, bir bot ya da bir tarayıcı ön-yükleme özelliği bu GET isteğini beklenmedik şekilde tekrar tetikleyebilir (bkz. "HTTP Metotları: Safe ve Idempotent Kavramları").

4. PATCH isteği gönderirken PUT semantiğini beklemek (yani gönderilmeyen alanların korunacağını değil, silineceğini sanmak) ya da tam tersi. İkisinin sözleşmesi kasıtlı olarak farklıdır -- hangisinin çağrıldığı, gönderilmeyen alanların akıbetini belirler (bkz. "PUT vs PATCH: Tam Güncelleme vs Kısmi Güncelleme").

5. DELETE'in idempotent olmasını, "ikinci çağrı da aynı durum kodunu döner" diye yanlış yorumlamak. İdempotency, durum kodunun değil, **sunucudaki nihai durumun** aynı kalmasıyla ilgilidir -- ilk çağrı 204, ikincisi 404 dönebilir, ikisi de idempotent'tir (bkz. "DELETE ve Idempotency").

6. Desteklenmeyen bir HTTP metoduyla gelen isteğe 404 dönmesini beklemek. Path gerçekten mevcutsa ama o HTTP metoduyla eşleşen bir mapping yoksa, doğru yanıt 405'tir -- 404, path'in kendisinin hiç bulunamadığı durumlar içindir (bkz. "Desteklenmeyen Bir HTTP Metodu İstendiğinde: 405 Method Not Allowed").

Özet, Cheat Sheet ve Terimler Sözlüğü

Mapping annotation'ları, bir controller metodunu path + HTTP metodu (+ isteğe bağlı content type) kombinasyonuna bağlar; her HTTP metodunun kendine has safe/idempotent kuralları vardır. Öne çıkan noktalar:

- `@RequestMapping`: temel annotation, `method` verilmezse her HTTP metodunu kabul eder
- `@GetMapping` / `@PostMapping` / `@PutMapping` / `@PatchMapping` / `@DeleteMapping`: beş HTTP metodu için kısayollar, `@RequestMapping(method=...)`'in meta-annotation'ları
- Sınıf seviyesinde `@RequestMapping`: ortak path öneki, metot seviyesindeki path'lerle birleşir
- `consumes` / `produces`: aynı path + HTTP metodu kombinasyonunu, content type'a göre ayrıştırır
- Safe: sunucu durumunu değiştirmez (yalnızca GET zorunlu)
- Idempotent: N kez çağırmak, 1 kez çağırmakla aynı **nihai durumu** üretir (GET, PUT, DELETE zorunlu; POST değil; PATCH garantili değil)
- 405 Method Not Allowed: path var ama bu HTTP metoduyla mapping yok (404'ten farklı)

Hızlı referans:

```
@RequestMapping(path = "/x", method = RequestMethod.GET)  // temel form
@GetMapping("/x")           // kısayolu -- ikisi eşdeğer

@RequestMapping("/users")   // sınıf seviyesinde ortak önek
class UserController {
    @GetMapping                    // GET /users
    @GetMapping("/{id}")           // GET /users/{id}
    @PostMapping                   // POST /users
    @PutMapping("/{id}")           // PUT /users/{id}     -- tam değiştirme
    @PatchMapping("/{id}")         // PATCH /users/{id}   -- kısmi değiştirme
    @DeleteMapping("/{id}")        // DELETE /users/{id}  -- idempotent
}

@PostMapping(path = "/orders", consumes = MediaType.APPLICATION_JSON_VALUE)
// yalnızca Content-Type: application/json olan isteklerle eşleşir
```

Terimler Sözlüğü

Mapping annotation — Bir controller metodunu, bir path + HTTP metodu (+ isteğe bağlı diğer koşullar) kombinasyonuna bağlayan annotation ailesi.

`@RequestMapping` — Ailenin temel annotation'ı; `method` attribute'uyla herhangi bir HTTP metodunu eşleyebilir, verilmezse hepsini kabul eder.

Meta-annotation — Başka bir annotation'ın üzerine kurulu, onu belirli bir attribute değeriyle önceden yapılandıran annotation (`@GetMapping` , `@RequestMapping(method=GET)` 'in meta-annotation'ıdır).

Safe (HTTP metodu) — Çağrıldığında sunucu durumunu değiştirmeyen HTTP metodu özelliği; yalnızca GET (ve HEAD/OPTIONS) için zorunludur.

Idempotent (HTTP metodu) — N kez çağrıldığında, sunucudaki nihai durumun 1 kez çağrılmışçasına aynı kalmasını garanti eden HTTP metodu özelliği.

405 Method Not Allowed — Path'in var olduğu ama istenen HTTP metoduyla eşleşen bir mapping bulunamadığı durumda dönen HTTP durum kodu.

`consumes` / `produces` — Bir mapping'i, isteğin `Content-Type` / `Accept` header'larına göre daraltan mapping annotation attribute'ları.

Ek: Mini Proje — Basit Bir Kitap CRUD API'si

Bu dersteki her annotation'ı, tek bir controller'da, gerçek bir kaynak üzerinde bir araya getiriyoruz:

```
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.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import java.util.LinkedHashMap;
import java.util.List;
import java.util.Map;

// A small, complete CRUD controller -- every mapping annotation from this lesson in
// one place, the same shape this project would use for a real /api/books endpoint.
@Controller
@RequestMapping("/api/books")
class BookCrudController {
    private final Map<Long, String> books = new LinkedHashMap<>();
    private long nextId = 1;

    @GetMapping
    @ResponseBody
    public List<String> list() {
        return List.copyOf(books.values());
    }

    @GetMapping("/{id}")
    @ResponseBody
```

```
    public ResponseEntity<String> getOne(@PathVariable Long id) {
        String title = books.get(id);
        return title != null ? ResponseEntity.ok(title) : ResponseEntity.status(HttpStatus.
NOT_FOUND).build();
    }

    @PostMapping
    @ResponseBody
    public ResponseEntity<Long> create(@RequestBody String title) {
        long id = nextId++;
        books.put(id, title);
        return ResponseEntity.status(HttpStatus.CREATED).body(id);
    }

    @PutMapping("/{id}")
    @ResponseBody
    public ResponseEntity<Void> replace(@PathVariable Long id, @RequestBody String title) {
        if (!books.containsKey(id)) {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
        }
        books.put(id, title);
        return ResponseEntity.noContent().build();
    }

    @DeleteMapping("/{id}")
    @ResponseBody
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        books.remove(id);
        return ResponseEntity.noContent().build(); // idempotent: same 204 whether or not
it existed
    }
}
```

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

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

        ResponseEntity<Long> created = controller.create("Effective Java");
        System.out.println(created.getStatusCode() + " id=" + created.getBody());
        // 201 CREATED id=1

        System.out.println(controller.list());
        // [Effective Java]

        System.out.println(controller.getOne(1L).getBody());
        // Effective Java

        controller.replace(1L, "Effective Java (3rd Edition)");
        System.out.println(controller.getOne(1L).getBody());
        // Effective Java (3rd Edition)

        System.out.println(controller.delete(1L).getStatusCode());
        // 204 NO_CONTENT
        System.out.println(controller.getOne(1L).getStatusCode());
        // 404 NOT_FOUND
    }
}
```

`BookCrudController`, "Sınıf ve Metot Seviyesinde @RequestMapping'i Birleştirmek" bölümünde gördüğümüz sınıf seviyesi önek deseniyle (`@RequestMapping("/api/books")`) başlıyor; `list()` / `getOne()` / `create()` / `replace()` / `delete()` , sırasıyla `@GetMapping` / `@GetMapping("/{id}")` / `@PostMapping` / `@PutMapping("/{id}")` / `@DeleteMapping("/{id}")` ile beş CRUD işlemini kapsıyor. `BookCrudDemo`, "Spring MVC Temelleri" dersindeki `ProductCatalogDemo` 'da yaptığımız gibi, gerçek bir DispatcherServlet olmadan controller metotlarını doğrudan çağırarak tüm akışı (oluştur → listele → güncelle → sil → tekrar sorgula) uçtan uca çalıştırıyor.

Ek: Mini Proje — HTTP Metodu Duyarlı Bir Router Simülasyonu

Son mini proje, Spring MVC Temelleri dersindeki `RequestRouterSimulation` 'ı, "Aynı Path, Farklı HTTP Metotları" bölümünde tanıttığımız `(path, method)` anahtarıyla birleştiriyor:

```
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.RequestMethod;

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

// Fundamentals' RequestRouterSimulation registered multiple handlers but could only
// key by path. This version keys by (path, HTTP method) across MULTIPLE handler
// classes -- the closest our hand-rolled simulations get to real Spring routing.
```

```
class ArticleApiHandlers {
    @GetMapping("/articles")
    public String list() {
        return "Listing articles";
    }

    @PostMapping("/articles")
    public String create() {
        return "Creating an article";
    }
}

class CommentApiHandlers {
    @GetMapping("/comments")
    public String list() {
        return "Listing comments";
    }

    @DeleteMapping("/comments")
    public String deleteAll() {
        return "Deleting all comments";
    }
}

class RouterWithMethodSimulation {
    private final Map<RouteKey, HandlerEntry> registry = new HashMap<>();

    private record RouteKey(String path, RequestMethod method) {
    }

    private record HandlerEntry(Object instance, Method method) {
    }

    void register(Object handler) {
        for (Method method : handler.getClass().getDeclaredMethods()) {
            GetMapping get = method.getAnnotation(GetMapping.class);
            if (get != null) {
                registry.put(new RouteKey(get.value()[0], RequestMethod.GET), new
HandlerEntry(handler, method));
            }
            PostMapping post = method.getAnnotation(PostMapping.class);
            if (post != null) {
                registry.put(new RouteKey(post.value()[0], RequestMethod.POST), new
HandlerEntry(handler, method));
            }
            DeleteMapping delete = method.getAnnotation(DeleteMapping.class);
            if (delete != null) {
                registry.put(new RouteKey(delete.value()[0], RequestMethod.DELETE), new
HandlerEntry(handler, method));
            }
        }
    }

    String dispatch(String path, RequestMethod httpMethod) {
        HandlerEntry entry = registry.get(new RouteKey(path, httpMethod));
        if (entry == null) {
            return "405 Method Not Allowed: " + httpMethod + " " + path;
        }
```

```
        try {
            return (String) entry.method().invoke(entry.instance());
        } catch (ReflectiveOperationException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
import org.springframework.web.bind.annotation.RequestMethod;

class RouterWithMethodDemo {
    public static void main(String[] args) {
        RouterWithMethodSimulation router = new RouterWithMethodSimulation();
        router.register(new ArticleApiHandlers());
        router.register(new CommentApiHandlers());

        System.out.println(router.dispatch("/articles", RequestMethod.GET));
        // Listing articles
        System.out.println(router.dispatch("/articles", RequestMethod.POST));
        // Creating an article
        System.out.println(router.dispatch("/articles", RequestMethod.DELETE));
        // 405 Method Not Allowed: DELETE /articles
        System.out.println(router.dispatch("/comments", RequestMethod.DELETE));
        // Deleting all comments
    }
}
```

`RouterWithMethodSimulation.register(...)`, artık `@GetMapping`, `@PostMapping` ve `@DeleteMapping`'in **üçünü de** okuyup aynı registry'ye, `RouteKey(path, method)` anahtarıyla ekliyor. `ArticleApiHandlers` ve `CommentApiHandlers` birbirinden habersiz iki ayrı "controller", ama `dispatch(...)` ikisini de tek bir yerden, hem path hem HTTP metoduna göre doğru şekilde buluyor -- Fundamentals dersindeki mini projenin bıraktığı eksiğin tam çözümü.

> `dispatch("/articles", RequestMethod.DELETE)` çağrısının `"405 Method Not Allowed"` dönmesine dikkat et -- `/articles` path'i registry'de var (`GET` ve `POST` için), ama `DELETE` için yok. Bu, "Desteklenmeyen Bir HTTP Metodu İstendiğinde: 405 Method Not Allowed" bölümünde gördüğümüz 404/405 ayrımının, elle yazılmış bir simülasyonda bile doğal olarak ortaya çıktığını gösteriyor.