REST API Tasarımı

Entity'yi doğrudan dışarı vermenin riskleri ve DTO deseni; Pageable/Page/Sort ile sayfalama, sıralama ve query parametreleriyle filtreleme; URI versioning vs header versioning; idempotency ve Idempotency-Key header'ı ile POST'u idempotent yapmak; HATEOAS (kısa bakış).

İleri 42 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/rest-api-design

---

REST API Tasarımı

Advanced Spring MVC dersi, bir isteğin **etrafına** nasıl davranış eklendiğini (interceptor, CORS, multipart) ele aldı -- bu ders ise isteğin/yanıtın **kendi şekline** dönüyor. Request ve Response Handling'de `@RequestBody`, `ResponseEntity` ve HTTP durum kodlarını, Validation & Exception Handling'de de `ProblemDetail` ile standart hata gövdelerini görmüştük -- bu ders, bu araçların üzerine, gerçek dünya REST API'lerinin sıkça karşılaştığı beş somut tasarım sorununu ekliyor: entity'yi dışarı sızdırmadan veri taşımak (DTO), büyük koleksiyonları parça parça döndürmek (pagination/sorting/filtering), bir API'yi geriye dönük uyumluluğu bozmadan değiştirmek (versioning), bir isteğin istemeden iki kez işlenmesini önlemek (idempotency) ve yanıtın kendi navigasyonunu taşıması (HATEOAS).

REST API Tasarımı Nedir?

REST (Representational State Transfer), Request ve Response Handling ve Path Variable'lar ve Request Parametreleri derslerinde zaten kullandığımız ilkelerin (kaynaklar URL'lerle temsil edilir, HTTP metotları anlamlı bir sözleşme taşır, yanıtlar durum kodlarıyla konuşur) bir mimari stildir. Bu ders, o ilkeleri **tek bir endpoint'in** ötesine, bir API'nin **bütününün** nasıl tasarlanacağına taşıyor:

```
// "REST'e uygun" tek bir endpoint yeterli değil -- bir API'nin tümü tutarlı
// olmalı: aynı hata şekli, aynı sayfalama deseni, aynı versiyonlama stratejisi.
@GetMapping("/api/v1/topics")
ResponseEntity<PagedResponse<TopicSummary>> listTopics(Pageable pageable) { ... }
```

Neden Var?

Her endpoint kendi kuralını icat ederse (biri sayfalamayı `?page=` / `?offset=` diye farklı adlandırır, biri hatayı düz metin diğeri JSON döner), bir API'yi tüketen istemci her endpoint için ayrı bir zihinsel model kurmak zorunda kalır. Validation & Exception Handling'de `@RestControllerAdvice`'ın hata gövdesini **tek bir yere** topladığını görmüştük -- bu dersteki desenlerin (DTO, pagination shape, versioning stratejisi) hepsi aynı motivasyonu paylaşıyor: tutarlılığı endpoint'ler arasında merkezi ve öngörülebilir kılmak.

Tarihçe

REST terimi, Roy Fielding'in 2000 doktora tezinde tanımlandı -- HTTP'nin kendisiyle aynı yaşta bir mimari stil değil, HTTP'nin **doğru kullanımı** üzerine bir gözlem. HATEOAS, Fielding'in orijinal tezinin bir parçasıydı, ama pratikte en az benimsenen ilke oldu -- çoğu "REST API" aslında HATEOAS'sız, düz JSON döndüren bir HTTP API. Spring HATEOAS projesi 2012'de bu boşluğu doldurmak için başladı (bu projede kullanılmıyor, bkz. "HATEOAS Nedir? (Kısa Bakış)"). Idempotency-Key header deseni, Stripe'ın API'sinin 2017'de popülerleştirdiği, sonradan IETF taslağına dönüşen bir konvansiyon. API versioning stratejileri arasındaki URI vs header tartışması ise 2010'lardan beri süregelen, hâlâ kesin bir tek doğrusu olmayan bir tasarım tercihi.

Entity'yi Doğrudan Dışarı Vermenin Riskleri: Neden DTO?

Bir JPA entity'sini doğrudan `@RestController`'dan döndürmek cazip görünür -- Jackson zaten onu JSON'a çevirebilir. Ama bunun iki gerçek riski var:

```
// Returning a JPA entity directly from a @RestController -- letting Jackson serialize
// it as-is -- looks convenient, but couples your HTTP contract to your database
// schema and can leak things you never meant to expose.
class EntityLeakageRiskExample {

    // A typical entity: exactly what the database needs, nothing about what an API
    // consumer should see.
    static class UserEntity {
        Long id;
        String email;
        String passwordHash;       // never meant to leave the server
        String internalNotes;      // an admin-only field, added later by someone else
        java.util.List<String> roles; // in a real @Entity, this would be a LAZY collection

        UserEntity(Long id, String email, String passwordHash, String internalNotes, java.
util.List<String> roles) {
            this.id = id;
            this.email = email;
            this.passwordHash = passwordHash;
            this.internalNotes = internalNotes;
            this.roles = roles;
        }
    }

    // What Jackson would serialize if this entity were returned directly from a
    // @RestController method -- every field, by default, becomes a JSON property.
    static String naiveSerialize(UserEntity user) {
        return "{\"id\":" + user.id
                + ",\"email\":\"" + user.email + "\""
                + ",\"passwordHash\":\"" + user.passwordHash + "\""      // leaked
                + ",\"internalNotes\":\"" + user.internalNotes + "\""   // leaked
                + ",\"roles\":" + user.roles + "}";
    }

    public static void main(String[] args) {
        UserEntity user = new UserEntity(1L, "ada@example.com", "$2a$10$abcdef...",
                "flagged for review 2025-11-02", java.util.List.of("USER"));

        System.out.println(naiveSerialize(user));
        // {"id":1,"email":"ada@example.com","passwordHash":"$2a$10$abcdef...",
        //  "internalNotes":"flagged for review 2025-11-02","roles":[USER]}

        // Two separate problems bundled into one bad decision:
        // 1) passwordHash/internalNotes were never meant to be public API fields.
        // 2) In a REAL @Entity, "roles" would likely be a LAZY collection -- serializing
        //    it outside an open Hibernate session throws LazyInitializationException,
        //    which is exactly why this project's TopicController resolves associations
        //    with an explicit join fetch (see TopicRepository.
findBySlugWithCategoryAndCourse)
        //    instead of leaving them to be touched later, e.g. during serialization.
    }
}
```

Birincisi: entity, veritabanının ihtiyaç duyduğu **her** alanı taşır -- `passwordHash`, iç notlar gibi hiçbir istemcinin görmemesi gereken alanlar da dahil. İkincisi: gerçek bir entity'de lazy-loaded bir koleksiyon (`@ManyToOne(FetchType.LAZY)` gibi, bu projenin `TopicRepository`'sinde

de gördüğümüz), serialization sırasında dokunulursa `LazyInitializationException` fırlatabilir -- bu proje bu riski `findBySlugWithCategoryAndCourse`'daki join fetch ile baştan çözüyor, ama genel prensip aynı: entity'nin iç yapısını API sözleşmesine sızdırma.

DTO Deseni: Record ile İstek/Yanıt Ayrımı

Çözüm, API'nin gerçekten ihtiyaç duyduğu şekli **ayrıca** tanımlamak:

```
// A DTO (Data Transfer Object) is a shape designed for the API contract, not the
// database. Records (see the Record lesson) are a natural fit -- immutable,
// concise, and each one describes exactly one direction of the conversation.
class DtoRecordExample {

    // Request DTO: only what a client is allowed to send. No id (the server assigns
    // it), no passwordHash (the client sends a plain password, the server hashes it).
    record CreateUserRequest(String email, String password) {
    }

    // Response DTO: only what a client is allowed to see. No passwordHash, no
    // internalNotes -- compare with EntityLeakageRiskExample.UserEntity.
    record UserResponse(Long id, String email, java.util.List<String> roles) {
    }

    public static void main(String[] args) {
        CreateUserRequest request = new CreateUserRequest("ada@example.com", "s3cret!");
        System.out.println(request);
        // DtoRecordExample$CreateUserRequest[email=ada@example.com, password=s3cret!]

        UserResponse response = new UserResponse(1L, request.email(), java.util.List.of
("USER"));
        System.out.println(response);
        // DtoRecordExample$UserResponse[id=1, email=ada@example.com, roles=[USER]]

        // Two different shapes for two different moments -- CreateUserRequest never
        // has an id (it doesn't exist yet), UserResponse never has a password (it
        // should never come back out). A single shared "User" shape used for both
        // directions can't express either constraint.
    }
}
```

Record dersinde gördüğümüz gibi, bir record immutable ve özlü -- her biri tek bir yönü (istek ya da yanıt) tanımlar. `CreateUserRequest`'in `id` alanı yok (henüz yok çünkü), `UserResponse`'un `password` alanı yok (asla dışarı çıkmaması gerektiği için). Tek bir paylaşılan "User" şekli bu iki kısıtı aynı anda ifade edemezdi.

Entity ↔ DTO Dönüşümü: Elle Mapping

DTO deseni, bir şeyin ikisi arasında **dönüştürmesiyle** anlam kazanır:

```
// The DTO pattern only pays off once something actually converts between entity and
// DTO. The simplest version is a plain static method -- no mapping library needed
// for a shape this small (see this lesson's "Example Writing Principles" -- don't add
// infrastructure a small example doesn't need).
class EntityToDtoMappingExample {

    record TopicSummary(String slug, String title, String difficulty) {
    }

    // Stands in for this project's real Topic/TopicTranslation entities -- a
    // simplified shape, just enough to show the mapping.
    record TopicEntityStub(String slug, String difficulty, String translatedTitle) {
    }

    static TopicSummary toDto(TopicEntityStub entity) {
        return new TopicSummary(entity.slug(), entity.translatedTitle(), entity.
difficulty());
    }

    public static void main(String[] args) {
        TopicEntityStub entity = new TopicEntityStub("advanced-spring-mvc", "ADVANCED",
"Advanced Spring MVC");

        TopicSummary dto = toDto(entity);
        System.out.println(dto);
        // TopicSummary[slug=advanced-spring-mvc, title=Advanced Spring MVC,
difficulty=ADVANCED]

        // At this project's actual scale, a hand-written toDto(...) per entity is
        // perfectly maintainable. Larger codebases often reach for a mapping library
        // (MapStruct is the common choice -- it generates this exact kind of method
        // at compile time instead of by hand) once there are dozens of DTOs and
        // fields change often enough that keeping mappings in sync by hand gets error-
prone.
    }
}
```

Bu projenin ölçeğinde elle yazılmış bir `toDto(...)` metodu gayet sürdürülebilir. Daha büyük kod tabanlarında, DTO sayısı onlarcaya çıkıp alanlar sık değiştiğinde, bu dönüşümü elle senkron tutmak hataya açık hâle gelir -- MapStruct gibi bir mapping kütüphanesi (derleme zamanında aynı metodu otomatik üretir) genelde bu noktada devreye girer.

Sayfalama (Pagination): Pageable ve Page

Büyük bir koleksiyonu tek seferde döndürmek yerine, parça parça vermek:

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;

import java.util.List;

// Pageable/Page are the same Spring Data types that back JpaRepository (see how
// TopicRepository extends it) -- when a @RestController method takes a Pageable
// parameter, Spring resolves it from ?page=/?size=/?sort= query parameters
// automatically, no manual parsing needed.
class PaginationExample {

    record Topic(String slug, String title) {
    }

    static Page<Topic> findTopics(List<Topic> allTopics, Pageable pageable) {
        int start = (int) pageable.getOffset();
        int end = Math.min(start + pageable.getPageSize(), allTopics.size());
        List<Topic> pageContent = start >= allTopics.size() ? List.of() : allTopics.subList
(start, end);
        // A real repository does this in the database (LIMIT/OFFSET); PageImpl here
        // just wraps an already-fetched in-memory list to demonstrate the shape.
        return new PageImpl<>(pageContent, pageable, allTopics.size());
    }

    public static void main(String[] args) {
        List<Topic> allTopics = List.of(
                new Topic("spring-mvc-fundamentals", "Spring MVC Fundamentals"),
                new Topic("mapping-annotations-http-methods", "Mapping Annotations and HTTP
Methods"),
                new Topic("path-variables-request-parameters", "Path Variables and Request
Parameters"),
                new Topic("request-response-handling", "Request and Response Handling"),
                new Topic("validation-exception-handling", "Validation and Exception
Handling"));

        // ?page=0&size=2 -- Spring resolves this into a Pageable automatically when
        // a controller method takes one as a parameter.
        Pageable firstPage = PageRequest.of(0, 2);
        Page<Topic> page1 = findTopics(allTopics, firstPage);

        System.out.println(page1.getContent());
        // [Topic[slug=spring-mvc-fundamentals, ...], Topic[slug=mapping-annotations-http-
methods, ...]]
        System.out.println("totalElements=" + page1.getTotalElements() + ", totalPages=" +
page1.getTotalPages());
        // totalElements=5, totalPages=3

        Pageable lastPage = PageRequest.of(2, 2);
        Page<Topic> page3 = findTopics(allTopics, lastPage);
        System.out.println(page3.getContent() + ", isLast=" + page3.isLast());
        // [Topic[slug=validation-exception-handling, ...]], isLast=true
    }
}
```

`Pageable` / `Page`, bu projenin `TopicRepository`'sinin de miras aldığı `JpaRepository` ailesinden -- `@RestController` metodun bir parametresi `Pageable` tipindeyse, Spring bunu `?page=` / `?size=` / `?sort=` query parametrelerinden otomatik çözer, elle parse etmeye gerek kalmaz. `page.getTotalElements()` / `getTotalPages()`, istemcinin kaç sayfa daha olduğunu bilmesini sağlar.

Sıralama (Sorting): Sort ile Çoklu Alan

`Sort`, `Pageable` ile birlikte çalışır ve birden fazla alanla zincirlenebilir:

```
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;

// Sort composes with Pageable -- a client can ask for ?sort=difficulty,asc&sort=title,desc
// and Spring resolves it into exactly the Sort object built here by hand.
class SortingExample {

    public static void main(String[] args) {
        Sort byDifficultyThenTitle = Sort.by(Sort.Direction.ASC, "difficulty")
                .and(Sort.by(Sort.Direction.DESC, "title"));

        System.out.println(byDifficultyThenTitle);
        // difficulty: ASC,title: DESC

        // Combined with paging into a single Pageable, exactly what a
        // @RestController parameter of type Pageable resolves to from query
        // parameters like ?page=0&size=10&sort=difficulty,asc&sort=title,desc:
        Pageable pageable = PageRequest.of(0, 10, byDifficultyThenTitle);
        System.out.println("page=" + pageable.getPageNumber() + ", sort=" + pageable.
getSort());
        // page=0, sort=difficulty: ASC,title: DESC

        // A shorthand for a single field:
        Pageable simpleSort = PageRequest.of(0, 10, Sort.by("title"));
        System.out.println(simpleSort.getSort());
        // title: ASC  -- Sort.by(String...) defaults to ascending
    }
}
```

`Sort.by(Sort.Direction.ASC, "difficulty").and(Sort.by(Sort.Direction.DESC, "title"))`, istemcinin `?sort=difficulty,asc&sort=title,desc` ile isteyeceği sıralamanın sunucu tarafındaki karşılığı -- Spring bu query parametrelerini otomatik olarak tam da bu `Sort` nesnesine çözer.

Filtreleme: Query Parametreleriyle Dinamik Sorgu

Path Variable'lar ve Request Parametreleri'nde `@RequestParam`'ın opsiyonel olabileceğini görmüştük -- filtreleme de tam olarak bunun üzerine kurulu:

```
import java.util.List;
import java.util.Optional;
import java.util.function.Predicate;

// Query parameters like ?difficulty=ADVANCED&category=spring-mvc filter a collection --
// each present parameter narrows the result, each absent one is simply skipped.
// A real repository would push this down into a WHERE clause (or a JPA Specification
// for cases this dynamic); this example keeps the filtering logic itself visible by
// building it as a chain of optional Predicates over an in-memory list.
class DynamicFilterExample {

    record Topic(String slug, String category, String difficulty) {
    }

    static List<Topic> filter(List<Topic> topics, String category, String difficulty) {
        Predicate<Topic> byCategory = Optional.ofNullable(category)
                .<Predicate<Topic>>map(c -> t -> t.category().equals(c))
                .orElse(t -> true);
        Predicate<Topic> byDifficulty = Optional.ofNullable(difficulty)
                .<Predicate<Topic>>map(d -> t -> t.difficulty().equals(d))
                .orElse(t -> true);

        return topics.stream().filter(byCategory.and(byDifficulty)).toList();
    }

    public static void main(String[] args) {
        List<Topic> topics = List.of(
                new Topic("advanced-spring-mvc", "spring-mvc", "ADVANCED"),
                new Topic("spring-mvc-fundamentals", "spring-mvc", "INTERMEDIATE"),
                new Topic("threads", "concurrency", "ADVANCED"));

        System.out.println(filter(topics, "spring-mvc", null));
        // [Topic[advanced-spring-mvc,...], Topic[spring-mvc-fundamentals,...]] -- category
only

        System.out.println(filter(topics, "spring-mvc", "ADVANCED"));
        // [Topic[advanced-spring-mvc,...]] -- both filters applied

        System.out.println(filter(topics, null, null));
        // all three -- no filters means every predicate defaults to "true"
    }
}
```

Her filtre parametresi opsiyonel: mevcutsa sonucu daraltır, yoksa hiç etkilemez (`true` dönen bir predicate ile). Gerçek bir repository'de bu mantık genelde veritabanına, bir `WHERE` cümlesine ya da (çok sayıda opsiyonel alan için) bir JPA `Specification`'a taşınır -- ama temel fikir aynı: her filtre, sağlanmamışsa "hiçbir şeyi eleme" davranışına düşer.

Sayfalanmış Yanıtın Şekli: content, totalElements, totalPages

Bir `Page<T>`'i doğrudan controller'dan döndürmek çalışır, ama Spring Data'nın kendisi bunu önermiyor -- `PageImpl`'in iç alanları belgeli, kararlı bir sözleşme değil ve varsayılan JSON şekli sürümler arasında değişebiliyor:

```
import org.springframework.data.domain.Page;

import java.util.List;

// Returning a Page<T> directly from a @RestController works, but Spring Data itself
// warns against it: PageImpl's internal fields aren't a stable, documented API
// contract, and its default JSON shape has changed across Spring Data versions.
// The recommended fix is the same idea as the DTO pattern -- wrap the page in a
// shape YOU control and document, not one an internal class happens to produce.
class PagedResponseShapeExample {

    record TopicSummary(String slug, String title) {
    }

    // A stable, project-owned response shape -- pulls only what a client actually
    // needs out of Page<T>, in field names this project's own API docs can commit to.
    record PagedResponse<T>(List<T> content, int page, int size, long totalElements, int
totalPages) {
        static <T> PagedResponse<T> from(Page<T> springDataPage) {
            return new PagedResponse<>(
                    springDataPage.getContent(),
                    springDataPage.getNumber(),
                    springDataPage.getSize(),
                    springDataPage.getTotalElements(),
                    springDataPage.getTotalPages());
        }
    }

    public static void main(String[] args) {
        Page<TopicSummary> springDataPage = new org.springframework.data.domain.PageImpl<>(
                List.of(new TopicSummary("advanced-spring-mvc", "Advanced Spring MVC")),
                org.springframework.data.domain.PageRequest.of(0, 2),
                5);

        PagedResponse<TopicSummary> response = PagedResponse.from(springDataPage);
        System.out.println(response);
        // PagedResponse[content=[TopicSummary[slug=advanced-spring-mvc, ...]], page=0,
        //   size=2, totalElements=5, totalPages=3]

        // Whatever Page<T>'s own serialization looks like in a given Spring Data
        // version, this record's shape doesn't change unless this project changes it.
    }
}
```

Çözüm, DTO desenindeki fikrin aynısı: `Page<T>`'i, bu projenin **kendi** belgeleyebileceği, alan adlarını **kendi** kontrol ettiği bir `PagedResponse<T>`'e sarmalamak. `Page<T>`'in dahili serialization'ı hangi Spring Data sürümünde nasıl değişirse değişsin, bu record'un şekli yalnızca bu proje değiştirirse değişir.

API Versioning: URI Versioning vs Header Versioning

Bir API zamanla değişir -- eski istemcileri kırmadan yeni bir şekil sunmanın iki yaygın yolu:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

// Two common ways to version a REST API. Neither needs a new mechanism -- both
// reuse tools this project already knows: @GetMapping's path (Mapping Annotations
// and HTTP Methods) for URI versioning, @RequestHeader (Path Variables and Request
// Parameters) for header versioning.
@RestController
class ApiVersioningExample {

    // URI versioning: the version is part of the path itself -- impossible to miss,
    // easy to route differently, but "v1"/"v2" leak into every client's URLs forever.
    @GetMapping("/api/v1/topics/{slug}")
    public String getTopicV1(String slug) {
        return "{\"slug\":\"" + slug + "\"}"; // v1 shape: flat
    }

    @GetMapping("/api/v2/topics/{slug}")
    public String getTopicV2(String slug) {
        return "{\"slug\":\"" + slug + "\",\"links\":{}}"; // v2 shape: adds a field
    }

    // Header versioning: the URL never changes -- one @GetMapping, the version comes
    // from a request header instead.
    @GetMapping("/api/topics/{slug}")
    public String getTopic(String slug, @RequestHeader(name = "Api-Version", defaultValue =
"1") int apiVersion) {
        return apiVersion >= 2
                ? "{\"slug\":\"" + slug + "\",\"links\":{}}"
                : "{\"slug\":\"" + slug + "\"}";
    }

    public static void main(String[] args) {
        ApiVersioningExample controller = new ApiVersioningExample();

        System.out.println(controller.getTopicV1("advanced-spring-mvc"));
        // {"slug":"advanced-spring-mvc"}
        System.out.println(controller.getTopicV2("advanced-spring-mvc"));
        // {"slug":"advanced-spring-mvc","links":{}}

        System.out.println(controller.getTopic("advanced-spring-mvc", 1));
        // {"slug":"advanced-spring-mvc"}
        System.out.println(controller.getTopic("advanced-spring-mvc", 2));
        // {"slug":"advanced-spring-mvc","links":{}}
    }
}
```

URI versioning (`/api/v1/...` vs `/api/v2/...`), Mapping Annotation'ları ve HTTP Metotları dersinde gördüğümüz `@GetMapping`'in path'inin bir parçası -- gözden kaçırılması imkansız, ama "v1"/"v2" istemcinin her URL'sine sonsuza kadar sızar. Header versioning (`Api-Version: 2`), Path Variable'lar ve Request Parametreleri dersindeki `@RequestHeader`'ı kullanır -- URL hiç değişmez, ama versiyon artık URL'e bakarak görünmez, dokümantasyona bağımlı hâle gelir.

Idempotency Nedir? Doğal Olarak Idempotent Metotlar

Bir işlem, bir kez çağrılmasıyla N kez çağrılması **aynı sonucu** üretiyorsa idempotent'tir:

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

// An operation is idempotent when calling it once has the same effect as calling it
// N times. Mapping Annotations and HTTP Methods already introduced idempotent
// as a property of GET/PUT/DELETE -- this example proves it by actually calling each
// operation twice and checking the store ends up in the same state either way.
class IdempotentMethodsExample {

    static final Map<String, String> store = new HashMap<>();

    static void put(String key, String value) {
        store.put(key, value); // PUT: replaces whatever was there -- same result every time
    }

    static void delete(String key) {
        store.remove(key); // DELETE: removing something already gone is still "gone" --
same result
    }

    static String post(String value) {
        // POST: creates a NEW resource every time -- calling it twice is NOT the same
        // as calling it once.
        String id = "id-" + (store.size() + 1);
        store.put(id, value);
        return id;
    }

    public static void main(String[] args) {
        store.clear();

        put("topic-1", "Advanced Spring MVC");
        put("topic-1", "Advanced Spring MVC"); // calling PUT again
        System.out.println(store);
        // {topic-1=Advanced Spring MVC} -- calling it twice left the exact same state

        delete("topic-1");
        delete("topic-1"); // calling DELETE on something already gone
        System.out.println(store.containsKey("topic-1"));
        // false either way -- both calls end in the same state

        store.clear();
        String firstId = post("REST API Design");
        String secondId = post("REST API Design"); // calling POST again
        System.out.println(firstId + " != " + secondId + " -> " + store);
        // id-1 != id-2 -> {id-1=REST API Design, id-2=REST API Design}
        // two calls created two resources -- POST is NOT idempotent by default
    }
}
```

`PUT` ve `DELETE`, doğaları gereği idempotent -- aynı `PUT`'u iki kez göndermek, kaynağı aynı son duruma getirir; aynı `DELETE`'i iki kez göndermek, kaynağı "yok" durumunda bırakır (ikinci çağrı hiçbir şey değiştirmez). `POST` ise **değil** -- her çağrı, tanımı gereği yeni bir kaynak yaratır. Bu ayrım, Mapping Annotation'ları ve HTTP Metotları dersinin "HTTP Metotları: Safe ve Idempotent Kavramları" bölümünde tanıtılmıştı; burada gerçekten çalıştırıp doğruluyoruz.

Idempotency-Key Header'ı ile POST'u Idempotent Yapmak

`POST`'un idempotent olmaması gerçek bir sorun yaratır: bir istemci zaman aşımından sonra isteği tekrar denediğinde, sunucunun isteği ilk kez mi işlediği belirsizdir. Çözüm, istemcinin ürettiği bir anahtarla sunucunun "bunu daha önce gördüm" diyebilmesi:

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

// IdempotentMethodsExample showed POST creating a new resource on every call -- a
// real problem when a client retries a request after a timeout, unsure whether the
// first attempt actually succeeded. The fix: the client generates a unique
// Idempotency-Key per logical operation and sends it with every retry; the server
// remembers which keys it has already processed and returns the SAME result instead
// of creating a duplicate.
class IdempotencyKeyExample {

    record OrderResult(String orderId, String status) {
    }

    static final Map<String, OrderResult> processedKeys = new HashMap<>();
    static int nextOrderNumber = 1;

    static OrderResult createOrder(String idempotencyKey, String item) {
        OrderResult existing = processedKeys.get(idempotencyKey);
        if (existing != null) {
            return existing; // same key seen before -- return the original result, create
nothing
        }

        OrderResult result = new OrderResult("order-" + nextOrderNumber++, "CREATED: " +
item);
        processedKeys.put(idempotencyKey, result);
        return result;
    }

    public static void main(String[] args) {
        String key = "a1b2c3-client-generated-uuid";

        OrderResult first = createOrder(key, "Java Mug");
        System.out.println(first);
        // OrderResult[orderId=order-1, status=CREATED: Java Mug]

        // The client didn't get a response in time (network blip) and retries with
        // the SAME key:
        OrderResult retry = createOrder(key, "Java Mug");
        System.out.println(retry);
        // OrderResult[orderId=order-1, status=CREATED: Java Mug] -- identical, no
duplicate order

        // A genuinely new order uses a fresh key, and does create a new resource:
        OrderResult secondOrder = createOrder("d4e5f6-different-uuid", "Mechanical
Keyboard");
        System.out.println(secondOrder);
        // OrderResult[orderId=order-2, status=CREATED: Mechanical Keyboard]
    }
}
```

İstemci, mantıksal bir işlem için tek bir `Idempotency-Key` üretir (genelde bir UUID) ve her denemede aynı anahtarı gönderir. Sunucu, o anahtarı daha önce işlediyse **yeni bir kaynak yaratmadan** ilk sonucu döndürür -- ikinci çağrının etkisi, ilk çağrının etkisiyle birebir aynı, yani `POST` artık efektif olarak idempotent.

HATEOAS Nedir? (Kısa Bakış)

HATEOAS, bir yanıtın yalnızca veri değil, istemcinin **sıradaki adımlarını** da taşımasıdır:

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

// HATEOAS (Hypermedia as the Engine of Application State): a response includes not
// just data, but the LINKS a client can follow next -- the API guides the client,
// instead of the client having to hard-code every URL it might ever need. This
// project doesn't use the real `spring-hateoas` library (it isn't a dependency
// here), so this example hand-builds the same shape a real HATEOAS response has,
// to show the idea without adding a library this project doesn't otherwise need.
class HateoasConceptExample {

    record TopicResponse(String slug, String title, Map<String, String> links) {
    }

    static TopicResponse toResponseWithLinks(String slug, String title, String
previousSlug, String nextSlug) {
        Map<String, String> links = new LinkedHashMap<>();
        links.put("self", "/api/topics/" + slug);
        if (previousSlug != null) {
            links.put("previous", "/api/topics/" + previousSlug);
        }
        if (nextSlug != null) {
            links.put("next", "/api/topics/" + nextSlug);
        }
        return new TopicResponse(slug, title, links);
    }

    public static void main(String[] args) {
        TopicResponse response = toResponseWithLinks(
                "advanced-spring-mvc", "Advanced Spring MVC",
                "spring-mvc-views-thymeleaf", "rest-api-design");

        System.out.println(response);
        // TopicResponse[slug=advanced-spring-mvc, title=Advanced Spring MVC,
        //   links={self=/api/topics/advanced-spring-mvc,
        //          previous=/api/topics/spring-mvc-views-thymeleaf,
        //          next=/api/topics/rest-api-design}]

        // A client following "next" never needs to know this project's URL scheme
        // (/api/topics/{slug}) -- it just follows the link the server gave it. This
        // project's own topic.html does the conceptual equivalent server-side
        // (previousTopic/nextTopic in TopicController), just rendered as HTML
        // <a> tags instead of a JSON "links" map.
    }
}
```

İstemci, `next` linkini takip ederken bu projenin URL şemasını (`/api/topics/{slug}`) hiç bilmek zorunda kalmaz -- sunucunun verdiği linki izler. Bu proje gerçek `spring-hateoas` kütüphanesini kullanmıyor (proje bağımlılıklarında yok), bu yüzden yukarıdaki örnek elle kurulmuş bir `links` map'i -- ama fikir, bu projenin kendi `topic.html`'inin `previousTopic`/`nextTopic` ile yaptığının (bkz. Spring MVC Views ve Thymeleaf dersi) JSON karşılığı: sunucu, "önceki"/"sonraki" neresi biliyor, istemcinin bilmesine gerek yok.

Best Practices

- **API'yi tüketen istemcinin hiç görmemesi gereken alanları (şifre hash'i, iç notlar, dahili ID'ler) bir DTO ile açıkça filtrele** -- entity'yi doğrudan döndürmek bunu unutmayı kolaylaştırır (bkz. "Entity'yi Doğrudan Dışarı Vermenin Riskleri: Neden DTO?").
- **Sayfalanmış yanıtları kendi kontrolündeki bir DTO'yla sarmala, `Page<T>`'i doğrudan döndürme** -- Spring Data'nın kendisi bunu önerir (bkz. "Sayfalanmış Yanıtın Şekli: content, totalElements, totalPages").
- **Versioning stratejini API'nin başından (ya da en azından ilk kırıcı değişiklikten önce) seç ve tutarlı uygula** -- yarı yolda URI'den header'a (ya da tersine) geçmek, mevcut tüm istemcileri kırar (bkz. "API Versioning: URI Versioning vs Header Versioning").
- **Yan etkisi olan (ödeme, sipariş oluşturma gibi) `POST` endpoint'lerinde `Idempotency-Key`'i ciddiye al** -- ağ zaman aşımları gerçek ve sık; bu desen olmadan bir tekrar deneme, çift ücretlendirme gibi somut kullanıcı zararlarına yol açabilir (bkz. "Idempotency-Key Header'ı ile POST'u Idempotent Yapmak").

Yaygın Hatalar

1. Entity'yi "şimdilik" doğrudan döndürüp DTO'yu sonraya bırakmak. Bir kez istemciler entity'nin şekline bağımlı hâle geldikten sonra, aradan bir DTO eklemek geriye dönük uyumluluğu bozan bir değişikliğe dönüşür -- DTO'yu en başından kurmak, sonradan eklemekten çok daha ucuzdur (bkz. "Entity'yi Doğrudan Dışarı Vermenin Riskleri: Neden DTO?").

2. Sayfalama parametrelerini (`page`, `size`, `sort`) elle `@RequestParam` ile okuyup `Pageable`'ı hiç kullanmamak. Bu, Spring'in zaten çözdüğü bir sorunu yeniden çözmek demektir -- `Pageable` parametresi aynı işi, doğrulama ve varsayılan değerlerle birlikte, tek satırda yapar (bkz. "Sayfalama (Pagination): Pageable ve Page").

3. Filtre parametrelerinin `null` olabileceğini unutup doğrudan `.equals(...)` çağırmak. `category.equals(t.category())` gibi bir kod, `category` sağlanmadığında `NullPointerException` fırlatır -- her opsiyonel filtre, "sağlanmadıysa etkisiz" davranışını açıkça ifade etmeli (bkz. "Filtreleme: Query Parametreleriyle Dinamik Sorgu").

4. URI versioning ile header versioning'i aynı API içinde karıştırmak. Bazı endpoint'ler `/api/v1/...`, bazıları `Api-Version` header'ı kullanırsa, istemci için hangi stratejinin geçerli olduğunu tahmin etmek zorlaşır -- bir API tek bir stratejide tutarlı kalmalı (bkz. "API Versioning: URI Versioning vs Header Versioning").

5. `Idempotency-Key`'i sunucu tarafında süresiz saklamak. Gerçek bir uygulamada anahtarlar bir süre sonra (örneğin 24 saat) temizlenmeli -- aksi hâlde bellek/depolama sınırsız büyür; bu örnekteki `Map`, süre sonu mantığı olmadan yalnızca fikri gösteriyor (bkz. "Idempotency-Key Header'ı ile POST'u Idempotent Yapmak").

6. HATEOAS linklerini yalnızca dokümantasyonda tanımlayıp yanıta hiç koymamak. HATEOAS'ın bütün amacı, istemcinin dokümantasyona değil **yanıtın kendisine** bakarak sıradaki adımı bulabilmesi -- yalnızca dokümante edilmiş ama yanıtta olmayan bir link, HATEOAS değil, sıradan bir API sözleşmesidir (bkz. "HATEOAS Nedir? (Kısa Bakış)").

Özet, Cheat Sheet ve Terimler Sözlüğü

REST API tasarımı, tek bir endpoint'in doğru çalışmasının ötesinde, bir API'nin **bütününün** tutarlı, öngörülebilir ve geriye dönük uyumlu kalmasıyla ilgili. Öne çıkan noktalar:

- DTO: API sözleşmesini entity'nin iç yapısından ayıran, istek/yanıt için ayrı şekiller tanımlayan desen
- `Pageable` / `Page<T>` / `Sort`: sayfalama ve sıralamanın Spring Data'daki karşılığı, query parametrelerinden otomatik çözülür
- Filtreleme: her query parametresi opsiyonel, sağlanmadığında sonucu etkilemeyen bir predicate
- `PagedResponse<T>`: `Page<T>`'in kararsız iç yapısı yerine, projenin kendi kontrolündeki sayfalama şekli
- URI versioning: versiyon path'te (`/api/v1/...`) -- görünür ama kalıcı
- Header versioning: versiyon bir header'da (`Api-Version`) -- URL sabit kalır ama versiyon görünmez
- Idempotent: bir kez ya da N kez çağrılması aynı sonucu üreten işlem (`GET` / `PUT` / `DELETE` doğal olarak, `POST` değil)
- `Idempotency-Key`: istemcinin ürettiği, sunucunun "bu isteği daha önce işledim" diyebilmesini sağlayan header
- HATEOAS: yanıtın veri yanında istemcinin sıradaki adımlarını (linkler) de taşıması

Hızlı referans:

```
@RestController
class TopicApiController {

    @GetMapping("/api/v1/topics")
    PagedResponse<TopicSummary> list(
            @RequestParam(required = false) String category,
            Pageable pageable) {
        // filtrele -> sayfala -> DTO'ya sarmala
        return PagedResponse.from(repository.findAll(pageable));
    }

    @PostMapping("/api/v1/orders")
    ResponseEntity<OrderResponse> createOrder(
            @RequestHeader("Idempotency-Key") String key,
            @RequestBody CreateOrderRequest request) {
        OrderResponse existing = seenKeys.get(key);
        if (existing != null) return ResponseEntity.ok(existing);
        // ... yeni kaynağı oluştur, seenKeys'e ekle ...
        return ResponseEntity.status(HttpStatus.CREATED).body(created);
    }
}
```

Terimler Sözlüğü

DTO (Data Transfer Object) — API sözleşmesi için tasarlanmış, veritabanı entity'sinden bağımsız veri şekli.

`Pageable` — Sayfa numarası, boyutu ve sıralamayı taşıyan, query parametrelerinden otomatik çözülen Spring Data arayüzü.

`Page<T>` — Bir sayfanın içeriğini (`content`) ve toplam eleman/sayfa sayısını birlikte taşıyan Spring Data arayüzü.

`Sort` — Bir ya da daha fazla alana göre, yön (`ASC` / `DESC`) belirtilerek sıralama tanımlayan Spring Data tipi.

URI versioning — API versiyonunun URL path'inin bir parçası olduğu versiyonlama stratejisi.

Header versioning — API versiyonunun bir HTTP header'ıyla belirtildiği, URL'in sabit kaldığı versiyonlama stratejisi.

Idempotent — Bir kez çağrılmasıyla N kez çağrılması aynı sonucu üreten işlem.

`Idempotency-Key` — İstemcinin ürettiği, sunucunun bir isteği daha önce işleyip işlemediğini anlamasını sağlayan HTTP header'ı.

HATEOAS (Hypermedia as the Engine of Application State) — Bir API yanıtının, veri yanında istemcinin izleyebileceği linkleri de taşıması gerektiğini söyleyen REST ilkesi.

Ek: Mini Proje — Sayfalanmış ve Filtrelenmiş Konu Kataloğu API'si

Bu dersin üç veri şekillendirme mekaniğini (filtreleme, sayfalama/sıralama, kararlı bir yanıt şekli) tek bir katalog endpoint'inde birleştiriyoruz:

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageImpl;
import org.springframework.data.domain.Pageable;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;
import java.util.Optional;
import java.util.function.Predicate;

// Mini project, part 1/2: combines this lesson's three data-shaping mechanics --
// filtering (?category=), pagination and sorting (Pageable), and a stable response
// shape (PagedResponseShapeExample's pattern) -- into a single catalog endpoint.
@RestController
class PaginatedCatalogController {

    record TopicSummary(String slug, String title, String category, String difficulty) {
    }

    record PagedResponse<T>(List<T> content, int page, int size, long totalElements, int
totalPages) {
        static <T> PagedResponse<T> from(Page<T> springDataPage) {
            return new PagedResponse<>(springDataPage.getContent(), springDataPage.
getNumber(),
                    springDataPage.getSize(), springDataPage.getTotalElements(),
springDataPage.getTotalPages());
        }
    }

    private final List<TopicSummary> allTopics;

    PaginatedCatalogController(List<TopicSummary> allTopics) {
        this.allTopics = allTopics;
    }

    @GetMapping("/api/topics")
    public PagedResponse<TopicSummary> listTopics(
            @RequestParam(required = false) String category,
            Pageable pageable) {

        Predicate<TopicSummary> matchesCategory = Optional.ofNullable(category)
                .<Predicate<TopicSummary>>map(c -> t -> t.category().equals(c))
                .orElse(t -> true);

        List<TopicSummary> filtered = allTopics.stream().filter(matchesCategory).toList();

        int start = Math.min((int) pageable.getOffset(), filtered.size());
        int end = Math.min(start + pageable.getPageSize(), filtered.size());

        Page<TopicSummary> page = new PageImpl<>(filtered.subList(start, end), pageable,
filtered.size());
        return PagedResponse.from(page);
    }
}
```

```
import org.springframework.data.domain.PageRequest;

import java.util.List;

// Mini project, part 2/2: drives PaginatedCatalogController with a small in-memory
// catalog -- one call with just paging, one adding a category filter, showing the
// filter narrows the total BEFORE paging is applied (totalElements reflects the
// filtered count, not the full catalog).
class PaginatedCatalogDemo {

    public static void main(String[] args) {
        List<PaginatedCatalogController.TopicSummary> catalog = List.of(
                new PaginatedCatalogController.TopicSummary(
                        "spring-mvc-fundamentals", "Spring MVC Fundamentals", "spring-mvc",
"INTERMEDIATE"),
                new PaginatedCatalogController.TopicSummary(
                        "advanced-spring-mvc", "Advanced Spring MVC", "spring-mvc",
"ADVANCED"),
                new PaginatedCatalogController.TopicSummary(
                        "threads", "Threads", "concurrency", "ADVANCED"));

        PaginatedCatalogController controller = new PaginatedCatalogController(catalog);

        System.out.println(controller.listTopics(null, PageRequest.of(0, 2)));
        // PagedResponse[content=[...2 topics...], page=0, size=2, totalElements=3,
totalPages=2]

        System.out.println(controller.listTopics("spring-mvc", PageRequest.of(0, 2)));
        // PagedResponse[content=[...2 spring-mvc topics...], page=0, size=2,
totalElements=2, totalPages=1]
        // -- totalElements is 2, not 3: it reflects the filtered set, not the whole catalog
    }
}
```

`listTopics`, `@RequestParam(required = false) String category` ile opsiyonel bir filtre, `Pageable` ile sayfalama/sıralama alıyor, ve sonucu `PagedResponseShapeExample`'daki gibi kararlı bir `PagedResponse<T>`'e sarmalıyor. `PaginatedCatalogDemo`, filtre uygulanmadan ve `category` filtresiyle iki farklı çağrı yaparak, `totalElements`'in filtrelenmiş kümeyi yansıttığını (tüm katalogu değil) gösteriyor.

Ek: Mini Proje — Idempotency Key Destekli Sipariş Oluşturma

Son mini proje, DTO desenini `Idempotency-Key` mekanizmasıyla gerçek bir `@PostMapping` / `ResponseEntity` üzerinde birleştiriyor:

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RestController;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

// Mini project, part 1/2: combines the DTO pattern (a request/response shape
// separate from any entity) with IdempotencyKeyExample's mechanism, wired through a
// real @PostMapping/@RequestHeader/ResponseEntity -- the same building blocks from
// Request ve Response Handling, applied to this lesson's idempotency problem.
@RestController
class IdempotentOrderController {

    record CreateOrderRequest(String item) {
    }

    record OrderResponse(String orderId, String item) {
    }

    private final Map<String, OrderResponse> processedKeys = new ConcurrentHashMap<>();
    private int nextOrderNumber = 1;

    @PostMapping("/api/orders")
    public ResponseEntity<OrderResponse> createOrder(
            @RequestHeader("Idempotency-Key") String idempotencyKey,
            @RequestBody CreateOrderRequest request) {

        OrderResponse existing = processedKeys.get(idempotencyKey);
        if (existing != null) {
            return ResponseEntity.ok(existing); // already processed -- 200, not a new 201
        }

        OrderResponse created = new OrderResponse("order-" + nextOrderNumber++, request.
item());
        processedKeys.put(idempotencyKey, created);
        return ResponseEntity.status(HttpStatus.CREATED).body(created);
    }
}
```

```
// Mini project, part 2/2: calls IdempotentOrderController.createOrder directly --
// once, then a "retry" with the same Idempotency-Key -- and shows the status code
// difference (201 vs. 200) alongside the identical order id.
class IdempotentOrderDemo {

    public static void main(String[] args) {
        IdempotentOrderController controller = new IdempotentOrderController();
        var request = new IdempotentOrderController.CreateOrderRequest("Java Mug");

        var first = controller.createOrder("a1b2c3-client-generated-uuid", request);
        System.out.println(first.getStatusCode() + " " + first.getBody());
        // 201 CREATED OrderResponse[orderId=order-1, item=Java Mug]

        var retry = controller.createOrder("a1b2c3-client-generated-uuid", request);
        System.out.println(retry.getStatusCode() + " " + retry.getBody());
        // 200 OK OrderResponse[orderId=order-1, item=Java Mug]  -- same order, not a
duplicate

        var secondOrder = controller.createOrder("d4e5f6-different-uuid",
                new IdempotentOrderController.CreateOrderRequest("Mechanical Keyboard"));
        System.out.println(secondOrder.getStatusCode() + " " + secondOrder.getBody());
        // 201 CREATED OrderResponse[orderId=order-2, item=Mechanical Keyboard]
    }
}
```

`createOrder`, `CreateOrderRequest` / `OrderResponse` DTO çiftini kullanıyor, `@RequestHeader("Idempotency-Key")` ile istemcinin anahtarını okuyor, ve daha önce görülmemiş bir anahtar için `201 Created`, daha önce görülmüş bir anahtar için `200 OK` (aynı gövdeyle) dönüyor. `IdempotentOrderDemo`, aynı anahtarla yapılan bir "retry"ın aynı sipariş ID'sini döndürdüğünü, farklı bir anahtarın ise gerçekten yeni bir sipariş yarattığını gösteriyor.