Pagination, Sorting, and Projections

"REST API Tasarımı"nın controller-seviyesinde çözdüğü Pageable/Sort'un, repository seviyesinde gerçekte nasıl LIMIT/OFFSET + count sorgusuna dönüştüğü, findAll(Sort) /derived metotlarla sıralama, ve bütün bir entity yerine yalnızca ihtiyaç duyulan alanları döndüren interface-tabanlı ve record/constructor-expression projection'lar -- bu projenin gerçek Topic/TopicTranslation sınıflarıyla. Spring Data JPA kategorisinin 4.'sü.

Orta 30 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/pagination-sorting-and-projections

---

"REST API Tasarımı", `Pageable` ve `Sort`'u query parametrelerinden çözen ve bir `Page<T>` döndüren bir controller gösterdi -- ama kendi örneği, bir yorumda, o `Page`'i "gerçek bir repository bunu veritabanında yapar" diyerek, zaten getirilmiş, bellek içi bir liste üzerinde inşa ettiğini itiraf ediyordu. Bu ders o diğer yarı: sayfalamayı, sıralamayı, ve sonuç şekillendirmeyi gerçek SQL'e dayanarak, repository seviyesinde gerçekten yapmak.

"REST API Tasarımı"nın Bitirmediği: Gerçek Sayfalama

Controller-tarafı hikaye zaten tamamlanmıştı: `Pageable` türünde bir `@RestController` metot parametresi, `?page=` / `?size=` / `?sort=`'den otomatik olarak çözülür, ve `Sort.by(...).and(...)`, programatik olarak çok-alanlı bir sıralama inşa eder. Bunların hiçbiri burada tekrarlanmıyor. Eksik olan, o `Pageable`'ın diğer ucunda ne olduğu -- onu gerçekten alan ve gerçek, verimli bir veritabanı sorgusuna çeviren repository metodu.

Sayfalanmış Bir Repository Metodu Bildirmek

Bir repository metodunu, bellekte zaten oturan bir liste üzerinde simüle etmek yerine gerçekten sayfalanmış yapmak, şaşırtıcı derecede az şey gerektiriyor.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

// "REST API Design"'s PaginationExample showed a controller resolving a
// Pageable from ?page=/?size= and returning a Page -- but it built the
// Page itself with PageImpl over an ALREADY-FETCHED, in-memory list,
// explicitly noting "a real repository does this in the database." This
// is that real repository method -- an extension to this project's own
// TopicRepository.
interface TopicPagingRepositoryExample extends JpaRepository<TopicExample, Long> {

    // Returning Page<TopicExample> (instead of List<TopicExample>) from a
    // method that takes a Pageable is all that's needed -- Spring Data JPA
    // generates a query with a real LIMIT/OFFSET, PLUS a second query that
    // counts the total matching rows, and packages both into the Page it
    // hands back.
    Page<TopicExample> findByCategoryId(Long categoryId, Pageable pageable);
}

class TopicExample {
}

class PagedRepositoryMethodExample {
    public static void main(String[] args) {
        // In a real application, this Pageable would already have been
        // resolved from ?page=0&size=2 by Spring MVC, exactly as "REST API
        // Design" covered -- constructed by hand here only to call the
        // repository method directly, outside of a running application.
        Pageable firstPage = PageRequest.of(0, 2);

        System.out.println(firstPage);
        // repository.findByCategoryId(5L, firstPage) would now run TWO real
        // queries against PostgreSQL: one SELECT ... LIMIT 2 OFFSET 0, and
        // one SELECT COUNT(*) -- see "What Actually Happens Underneath."
    }
}
```

`findByCategoryId(Long categoryId, Pageable pageable)`, "Query Methods and JPQL with @Query"deki derived query metotlarına neredeyse birebir benziyor -- tek değişiklik dönüş türü, `List<TopicExample>` yerine `Page<TopicExample>`. Bu tek değişiklik yeterli: Spring Data JPA, gerçek bir `LIMIT` / `OFFSET` içeren bir sorgu, artı eşleşen toplam satır sayısını sayan ikinci bir sorgu üretir, ve ikisini de döndürdüğü `Page`'e paketler.

Altında Gerçekte Ne Olur: LIMIT, OFFSET ve Bir Count Sorgusu

Bir `Page` döndüren metoda tek bir çağrı, sessizce bir değil İKİ sorgu çalıştırır:

```
findByCategoryId(5L, PageRequest.of(0, 2))
        |
        +--> SELECT * FROM topic WHERE category_id = 5 LIMIT 2 OFFSET 0
        |         (gerçek sayfa satırları)
        |
        +--> SELECT COUNT(*) FROM topic WHERE category_id = 5
                  (toplamda, her sayfa boyunca kaç satır var)
```

İlk sorgu yalnızca mevcut sayfanın satırlarını getirir -- asla bütün tabloyu değil. İkincisi, "REST API Tasarımı"nda controller seviyesinde zaten kullanılan `page.getTotalElements()` / `getTotalPages()`'ı mümkün kılan şeydir; o olmadan, kaç sayfa kaldığını bilmenin hiçbir yolu olmazdı. Her iki sorgu da, metodun kendi derived adından ya da `@Query`'sinden bir kez üretilen aynı `WHERE` koşulunu paylaşır.

Repository Seviyesinde Sıralama

Sayfalanmış (ya da sayfalanmamış) bir sonucu sıralamak yeni bir mekanizma gerektirmez -- zaten işlenen parçaları yeniden kullanır.

```
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

interface TopicSortingRepositoryExample extends JpaRepository<TopicSortExample, Long> {

    // findAll(Sort sort) isn't written here at all -- it comes directly
    // from PagingAndSortingRepository, exactly as covered in "Entities and
    // the Repository Abstraction." No new method needed to sort every
    // Topic by an arbitrary field.

    // A derived method can ALSO accept a Sort parameter, combining a
    // filter condition with caller-supplied ordering -- CategoryId narrows
    // the rows, Sort decides what order they come back in.
    List<TopicSortExample> findByCategoryId(Long categoryId, Sort sort);
}

class TopicSortExample {
}

class SortAtRepositoryLevelExample {
    public static void main(String[] args) {
        // Sort.by(...).and(...) construction itself was already covered in
        // "REST API Design" -- this is the exact same Sort object, now
        // handed to a repository method instead of just resolved from
        // query parameters.
        Sort bySortOrder = Sort.by(Sort.Direction.ASC, "sortOrder");

        System.out.println(bySortOrder);
        // repository.findAll(bySortOrder) -- every Topic, ordered by sortOrder
        // repository.findByCategoryId(5L, bySortOrder) -- only category 5's
        //     Topics, in the same order
    }
}
```

`findAll(Sort sort)`, bu interface'te hiçbir yerde yazılmıyor -- "Entities and the Repository Abstraction"ın zaten işlediği gibi, doğrudan `PagingAndSortingRepository`'den miras alınıyor; yalnızca her satırı rastgele bir alana göre sıralamak için yeni bir metoda gerek yok. Derived bir metot da, kendi olağan koşullarının yanı sıra doğrudan bir `Sort` parametresi kabul edebilir -- `findByCategoryId(Long categoryId, Sort sort)`, kategoriye göre filtreler VE sonucu sıralar, çağıran sıralamayı sağlar. `Sort` nesnesinin kendisi -- `Sort.by(...).and(...)` -- "REST API Tasarımı"ndan değişmedi; burada yeni olan yalnızca kime teslim edildiği (yalnızca controller'da çözülmek yerine bir repository metoduna).

Sayfalamayı, Sıralamayı ve Bir Filtre Koşulunu Birleştirmek

`Pageable` ve `Sort`, gerçekte jonglörlük yapılacak iki ayrı konu değildir -- bir `Pageable`, zaten kendi gömülü `Sort`'unu taşır.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.repository.JpaRepository;

interface TopicDifficultyPagingRepositoryExample extends
JpaRepository<TopicDifficultyExample, Long> {

    // A single Pageable already carries BOTH paging (page/size) AND
    // sorting information (Pageable has its own embedded Sort) -- no
    // separate Sort parameter is needed here at all.
    Page<TopicDifficultyExample> findByDifficulty(String difficulty, Pageable pageable);
}

class TopicDifficultyExample {
}

class PagedAndFilteredQueryExample {
    public static void main(String[] args) {
        // A Pageable built WITH a Sort attached -- PageRequest.of(page,
        // size, sort) -- carries paging and ordering together in one object.
        Pageable secondPageOrderedBySlug = PageRequest.of(1, 5, Sort.by("slug"));

        System.out.println(secondPageOrderedBySlug);
        // repository.findByDifficulty("INTERMEDIATE", secondPageOrderedBySlug)
        // generates one query filtering by difficulty, ordering by slug,
        // and applying LIMIT 5 OFFSET 5 -- all three concerns, from one
        // method call.
    }
}
```

`findByDifficulty(String difficulty, Pageable pageable)`, hiç ayrı bir `Sort` parametresine ihtiyaç duymaz, çünkü `PageRequest.of(page, size, sort)` zaten sayfalamayı ve sıralamayı tek bir `Pageable`'a paketler. Tek bir metot çağrısı, tek bir `Pageable` argümanı, ve üretilen sorgu, hepsini aynı anda -- filtreler, sıralar, ve sayfalar -- tek, sıradan bir metot imzasıyla ele alınan üç konu.

Bütün Bir Entity Getirmek Yerine Neden Proje(kte) Al?

Şimdiye kadarki her sorgu, çağıranın ihtiyacı olsun olmasın, eşlenen her alanla, bütün bir entity döndürdü. Bir PROJECTION, yalnızca belirli bir sorgunun gerçekten ihtiyaç duyduğu alanları döndürür, hem SQL'in kendisini daraltır (daha az sütun seçilir) hem de yalnızca okunacak, asla onun üzerinden değiştirilmeyecek veri için bütün bir entity'yi yönetmenin ek yükünden kaçınır.

Interface-Tabanlı Projection'lar

En basit projection, bir entity'nin özelliklerinin bir alt kümesiyle eşleşen getter'ları olan basitçe bir interface'tir.

```
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

// An INTERFACE PROJECTION: instead of the full Topic entity (id, category,
// slug, difficulty, estimatedMinutes, sortOrder, and any relationships it
// carries), only THREE fields are actually needed here. Declaring an
// interface with getters matching a subset of the entity's property names
// is enough -- no implementation is written for it.
interface TopicSummary {
    String getSlug();
    String getDifficulty();
    Integer getEstimatedMinutes();
}

interface TopicProjectionRepositoryExample extends JpaRepository<TopicProjectionExample,
Long> {

    // Spring Data JPA generates a proxy implementing TopicSummary at
    // runtime, and -- this is the real benefit, not just less Java code --
    // generates a SQL SELECT that names only slug/difficulty/
    // estimated_minutes, not every column the full entity would require.
    List<TopicSummary> findByCategoryId(Long categoryId);
}

class TopicProjectionExample {
}
```

`TopicSummary`, üç getter bildirir -- `getSlug()`, `getDifficulty()`, `getEstimatedMinutes()` -- ve hiçbiri elle implement edilmez. Spring Data JPA, çalışma zamanında onu implement eden bir proxy üretir, ve -- gerçek fayda, yalnızca daha az Java yazmak değil -- bütün `Topic` entity'sinin gerektireceği her sütunu değil, yalnızca o üç sütunu adlandıran bir SQL `SELECT` üretir.

DTO/Record Projection'ları

Bir record projection, tam olarak aynı daralmayı ister, ama sorgu, Spring Data JPA'nın getter adlarından çıkarması yerine, sonucun tam olarak nasıl inşa edileceğini söylemek zorundadır.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import java.util.List;

// A DTO/record projection -- as "Record" already covers, using a record
// as a JPQL "constructor expression" target. Unlike TopicSummary's
// interface projection (Spring Data JPA proxies it automatically), a
// record projection requires the query to say EXACTLY how to build one,
// via "select new fully.qualified.RecordName(...)".
record TopicTitleView(String slug, String title) {
}

interface TopicTranslationProjectionRepositoryExample extends org.springframework.data.jpa.
repository.JpaRepository<TopicTranslationProjectionExample, Long> {

    // This project's Topic entity carries no title itself -- title lives
    // on TopicTranslation, one per language (see "JPA, Hibernate, and
    // Spring Data JPA"'s and "Entities and the Repository Abstraction"'s
    // real Topic/TopicTranslation classes). This query joins across that
    // relationship and constructs a TopicTitleView directly -- pulling
    // back exactly two columns, from two tables, with no intermediate
    // entity ever loaded.
    @Query("select new com.cdurgun.learning.example.TopicTitleView(tt.topic.slug, tt.title)
" +
            "from TopicTranslationProjectionExample tt " +
            "where tt.language = :language and tt.published = true")
    List<TopicTitleView> findAllPublishedTitles(String language);
}

class TopicTranslationProjectionExample {
}
```

"Record"un tam olarak bunun için önerdiği türden bir `record` olan `TopicTitleView`, JPQL'in içinde doğrudan `select new ...TopicTitleView(tt.topic.slug, tt.title)` ile, bir "constructor expression" ile inşa edilir. Bu sorgu, `TopicTranslation`'ın `Topic`'e ilişkisi üzerinden birleşir ve iki tablodan tam olarak iki sütunu geri getirir, ara bir `Topic` ya da `TopicTranslation` entity'si hiç tam olarak belleğe yüklenmeden.

> Bir sorgunun TEK bir entity'nin kendi alanlarının basit bir alt kümesine ihtiyacı olduğunda interface projection'ı tercih et -- hiçbir sorgu değişikliği gerektirmez. Bir projection'ın burada `TopicTitleView`'ın yaptığı gibi bir ilişki BOYUNCA alan çekmesi gerektiğinde, ya da düz bir getter'ın ifade edemeyeceği herhangi bir hesaplanmış değere ihtiyacı olduğunda, bir record/constructor-expression projection'a başvur.

Yaygın Yanlış Anlamalar

"`Page<T>` ve `List<T>` temelde aynı şey, yalnızca ekstra metadata'yla." Gerçekten farklı sorgulardan gelirler -- `List<T>` döndüren bir metot tek bir sorgu çalıştırır; `Page<T>` döndüren bir metot iki tane çalıştırır (sayfanın satırları, ve ayrı bir count). **"Bir projection**

yalnızca daha az Java yazmakla ilgili." Gerçek fayda daha dar bir SQL `SELECT`'tir -- veritabanından getirilen daha az sütun, yalnızca sonucu tutan daha küçük bir Java türü değil. **"`Pageable` ve `Sort`, etrafta taşınacak iki ayrı şey."** Bir `Pageable`, içeride zaten kendi `Sort`'unu taşır -- `PageRequest.of(page, size, sort)` ile bir tane inşa etmek genelde yeterlidir, yanında ayrı bir `Sort` parametresine gerek kalmadan.

Sırada Ne Var

Bu dersteki her sorgu, sabit, bilinen bir koşulda -- `categoryId`, `difficulty` -- derleme zamanında, metodun kendi adı ya da `@Query`'si tarafından karara bağlanarak filtreledi. Bu kategoride sıradaki "Specifications ile Dinamik Sorgular", "REST API Tasarımı"nın filtreleme bölümünün yalnızca geçerken adını andığı şeyi işliyor: bir sorgunun koşullarını, aktif filtreler kümesi bir istek gerçekten gelene kadar bilinmediğinde, ÇALIŞMA ZAMANINDA inşa etmek.

Best Practices

- Sayfalanmış bir API endpoint'inin çağıracağı herhangi bir repository metodundan `List<T>` değil `Page<T>` döndür -- ekstra count sorgusu, `totalElements` / `totalPages`'ı baştan mümkün kılan şeydir.
- Bir sorgunun çağıranı bir entity'nin alanlarının yalnızca bir alt kümesine ihtiyaç duyduğunda, bütünü getirip (ve bedelini ödemek) yerine bir projection'a -- interface ya da record -- başvur.
- Yanında ayrı bir `Sort` parametresiyle jonglörlük yapmak yerine `PageRequest.of(page, size, sort)` ile bir `Pageable` inşa et.
- Tek bir entity'nin alanlarının basit bir alt kümesi için interface projection kullan; bir ilişki ya da hesaplanmış bir değer devreye girdiğinde bir record/constructor-expression projection'a başvur.

Yaygın Hatalar

- Gerçekten `Page` döndüren bir repository metodu bildirmek yerine, zaten tam olarak getirilmiş bir `List` üzerinde sayfalamayı simüle etmek (tam olarak "REST API Tasarımı"nın kendi örneğinin gerçekte yapmaktan bilinçli olarak kaçındığı şey).
- Bütün bir entity getirip sonradan birkaç alanı elle bir DTO'ya kopyalamak, bir projection'ın sorgunun kendisini daraltmasına izin vermek yerine.
- Aynı zamanda bir `Pageable` de alan bir metoda ayrı bir `Sort` parametresi geçirmek, `Pageable`'ın sıralamayı zaten taşıyabildiğini fark etmeden.
- Bir interface projection'ın, tek bir entity'nin kendi getter'larının basit bir alt kümesinin ötesinde herhangi bir şey için çalışmasını beklemek -- bir ilişkiye yayılan ya da hesaplanmış bir sonuç, bunun yerine bir constructor-expression projection'a ihtiyaç duyar.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- `List<T>` yerine `Page<T>` döndüren bir repository metodu, gerçek bir `LIMIT` / `OFFSET` sorgusu artı ayrı bir count sorgusu üretir, birlikte paketlenmiş.
- `findAll(Sort)`, `PagingAndSortingRepository`'den bedavaya miras alınır; derived bir metot da doğrudan bir `Sort` parametresi alabilir.
- Bir `Pageable`, zaten kendi gömülü `Sort`'unu taşır -- `PageRequest.of(page, size, sort)`, sayfalamayı ve sıralamayı birlikte ele alır.
- Bir projection, yalnızca bir sorgunun gerçekten ihtiyaç duyduğu alanları döndürür, üretilen SQL'in kendisini daraltır, yalnızca sonucu tutan Java türünü değil.
- Bir interface projection, yalnızca bir entity'nin özelliklerinin bir alt kümesiyle eşleşen getter'lara ihtiyaç duyar; bir sorgu bir ilişkiye yayıldığında ya da bir değer hesapladığında bir record/constructor-expression projection gerekir.

Cheat Sheet

```
// Gerçekten sayfalanmış bir repository metodu
Page<Topic> findByCategoryId(Long categoryId, Pageable pageable);

// Sıralama: bedavaya miras alınır, ya da derived-metot parametresi olarak
List<Topic> findAll(Sort sort); // PagingAndSortingRepository'den
List<Topic> findByCategoryId(Long categoryId, Sort sort);

// Sayfalama + sıralama + filtreleme birlikte
Page<Topic> findByDifficulty(String difficulty, Pageable pageable);
Pageable pageable = PageRequest.of(1, 5, Sort.by("slug"));

// Interface projection
interface TopicSummary {
    String getSlug();
    Integer getEstimatedMinutes();
}
List<TopicSummary> findByCategoryId(Long categoryId);

// Record / constructor-expression projection
record TopicTitleView(String slug, String title) {}

@Query("select new com.example.TopicTitleView(tt.topic.slug, tt.title) " +
       "from TopicTranslation tt where tt.language = :language")
List<TopicTitleView> findAllTitles(String language);
```

Terimler Sözlüğü

- **Page<T>**: bir sonuç sayfasını artı metadata'yı (toplam eleman, toplam sayfa) temsil eden, iki gerçek sorguya dayanan bir Spring Data türü.
- **Pageable**: hangi sayfanın ve boyutun getirileceğini, kendi gömülü `Sort`'uyla birlikte tarif eden bir nesne.

- **Projection**: bir entity'nin alanlarının bir alt kümesine daraltılmış bir sorgu sonucu, üretilen SQL'in kendi `SELECT` listesini küçültür.
- **Interface projection**: bir entity'nin özelliklerinin bir alt kümesiyle eşleşen getter'ları olan, Spring Data JPA tarafından otomatik olarak implement edilen bir interface olarak tanımlanmış bir projection.
- **Constructor-expression (record/DTO) projection**: alanlar bir ilişkiye yayıldığında ya da hesaplandığında gereken, JPQL'in içinde `select new ...SomeType(...)` ile açıkça inşa edilen bir projection.