Query Methods and JPQL with @Query

Spring Data JPA'nın bir repository metodunun adından bir sorguyu nasıl çıkardığı (findBy /existsBy/countBy, And/Or/OrderBy), @Query ile JPQL yazmak ve isimli parametreler, toplu güncelleme/silme için @Modifying, join fetch, ve ne zaman bir native query'ye başvurulacağı -- bu projenin gerçek repository'leriyle. Spring Data JPA kategorisinin 3.'sü.

Orta 30 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/query-methods-and-jpql

---

"Entities and the Repository Abstraction", `TopicRepository.findBySlug(String slug)`'ın çalıştığını gösterdi -- hiçbir yerde hiçbir sorgu, SQL, JPQL yazılmadan -- ve bunun tam olarak nasıl mümkün olduğunu bu derse erteledi. İşte tam olarak burada cevaplanıyor: bir repository'ye ne getireceğini söylemenin iki farklı yolu, gerçekten ihtiyacın olmadıkça hiçbir zaman elle bir `SELECT` yazmadan.

Bir Repository'den Veri İstemenin İki Yolu

Bir repository metodu, sorgusunu iki yerden birinden alabilir: Spring Data JPA, metodun kendi adından otomatik olarak bir tane ÇIKARABİLİR, ya da `@Query` ile açıkça bir tane yazabilirsin. Bu projedeki her repository metodu ikisinden birini kullanır -- üçüncü bir seçenek yoktur, ve hiçbir metot tahmine bırakılmaz.

Derived Query Metotları: Bir Sorguyu Metot Adından Okumak

Spring Data JPA, uygulama başlangıcında bir metodun adını ayrıştırır, parçalarını entity'nin kendi özellikleriyle eşleştirir, ve bundan bir sorgu inşa eder -- uygulaman gerçek bir isteği hiç ele almadan önce.

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

import java.util.List;
import java.util.Optional;

// This project's own CodeExampleRepository (see src/main/java/com/cdurgun/
// learning/repository/CodeExampleRepository.java) -- two methods, zero
// SQL/JPQL written anywhere. Spring Data JPA parses each method's NAME at
// startup and builds a query from it.
interface CodeExampleRepositoryExample extends JpaRepository<CodeExampleExample, Long> {

    // "findBy" + "TopicId" + "OrderBy" + "SortOrder" + "Asc" is parsed as:
    //   SELECT * FROM code_example
    //   WHERE topic_id = ?
    //   ORDER BY sort_order ASC
    // "TopicId" resolves to the entity's "topic" relationship's id --
    // Spring Data JPA understands nested property paths, not just direct
    // fields.
    List<CodeExampleExample> findByTopicIdOrderBySortOrderAsc(Long topicId);

    // "findBy" + "TopicId" + "And" + "ExampleName" is parsed as:
    //   SELECT * FROM code_example
    //   WHERE topic_id = ? AND example_name = ?
    // Parameters are matched to conditions IN ORDER -- the first parameter
    // (topicId) binds to the first condition (TopicId), the second
    // (exampleName) to the second (ExampleName).
    Optional<CodeExampleExample> findByTopicIdAndExampleName(Long topicId, String
exampleName);
}

class CodeExampleExample {
}
```

`findByTopicIdOrderBySortOrderAsc(Long topicId)`, parça parça okunur: `findBy` bir koşul başlatır, `TopicId`, `WHERE topic_id = ?` olur, ve `OrderBySortOrderAsc`, `ORDER BY sort_order ASC` olur -- özellikle, `TopicId`, yalnızca doğrudan bir alan değil, entity'nin `topic` ilişkisi üzerinden onun `id`'sine kadar çözülür. `findByTopicIdAndExampleName(...)`, tek bir metotta iki koşulu gösterir: parametreler koşullarla SIRAYLA eşleştirilir, bu yüzden ilk parametre ilk koşula, ikincisi ikinciye bağlanır.

Koşulları Birleştirmek ve Sıralamak

Aynı adlandırma kuralları, birkaç koşul ve anahtar kelimenin zincirlendiği metotlara da ölçeklenir -- bu projenin gerçek `QuizRepository`'si bunun en yoğun örneğine sahip.

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

import java.util.Optional;

// This project's own QuizRepository has the single most keyword-dense
// derived query method in the codebase -- a good specimen for seeing
// several naming keywords used together.
interface QuizRepositoryExample extends JpaRepository<QuizExample, Long> {

    // findFirstByTopicIdAndLanguageAndActiveTrueOrderByIdAsc breaks down as:
    //
    //   findFirst        -> LIMIT 1 (return one result, not a List)
    //   By                -> begins the condition clause
    //   TopicId           -> WHERE topic_id = ?          (1st parameter)
    //   And               -> combine with AND
    //   Language          -> AND language = ?             (2nd parameter)
    //   And               -> combine with AND
    //   ActiveTrue        -> AND active = true             (NO parameter --
    //                        "True"/"False" are LITERAL values, not
    //                        placeholders, for a boolean property)
    //   OrderByIdAsc      -> ORDER BY id ASC
    //
    // Two parameters go in (topicId, language) even though the method
    // name mentions three conditions -- ActiveTrue supplies its own value.
    Optional<QuizExample> findFirstByTopicIdAndLanguageAndActiveTrueOrderByIdAsc(Long
topicId, String language);

    // Two more derived prefixes this project doesn't happen to use, but
    // follow the identical parsing rules -- notice each returns a
    // different, purpose-built type instead of the entity itself:
    //
    //   existsByTopicIdAndLanguage(Long topicId, String language) -> boolean
    //     (a single SELECT EXISTS(...) query -- avoids loading a whole
    //     entity just to check whether one is present)
    //
    //   countByTopicId(Long topicId) -> long
    //     (a single SELECT COUNT(*) query -- avoids loading any rows at
    //     all just to know how many there are)
}

class QuizExample {
}
```

`findFirstByTopicIdAndLanguageAndActiveTrueOrderByIdAsc(...)`, beş parçaya ayrılır: `findFirst`, sonucu bir liste yerine tek bir satırla sınırlar; `TopicId` ve `Language` (`And` ile birleşmiş), her biri bir metot parametresi tüketir; `ActiveTrue`, ÜÇÜNCÜ bir koşul ekler -- `WHERE active = true` -- ama HİÇ parametre tüketmez, çünkü `True` / `False`, boolean bir özellik için kendi literal değerini sağlar; `OrderByIdAsc`, sonucu sıralar. Üç koşul, yalnızca iki parametre -- tam olarak bir bakışta yanlış sayılması kolay olduğu için fark edilmeye değer.

Diğer Derived Önekler: findFirstBy, existsBy ve countBy

`findBy`, Spring Data JPA'nın anladığı tek önek değildir -- `existsBy` ve `countBy`, aynı ayrıştırma kurallarını izler, ama temelde farklı, daha verimli bir cevap şekli döndürür. `existsByTopicIdAndLanguage(...)`, tek bir `SELECT EXISTS(...)` sorgusundan düz bir `boolean` döndürür -- bir şeyin var olup olmadığını, öğrenmek için bütün bir entity'yi yüklemeden kontrol eder. `countByTopicId(...)`, tek bir `SELECT COUNT(*)` sorgusundan bir `long` döndürür -- satırları hiç getirmeden onları sayar. Yalnızca boolean'a ya da sayıya ihtiyaç duyduğunda, gerçek veriye değil, `findBy...().isPresent()` ya da `findBy...().size()` yerine bunlara başvur.

Bir Derived İsim Yetmediğinde: @Query ve JPQL

Derived bir isim, tek bir entity üzerindeki basit koşullar için iyi çalışır -- bir sorgu ilişkiler arasında gezinmesi gerektiğinde, ya da bir metot adının temiz bir şekilde ifade edemeyeceği bir şeyi ifade etmesi gerektiğinde doğru araç olmaktan çıkar.

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

import java.util.Optional;

// This project's own QuizRepository -- a query no method name could
// derive cleanly (three conditions across two joined entities, one of
// them a relationship's own field). @Query switches from a generated
// query to one written directly in JPQL.
interface QuizJpqlRepositoryExample extends JpaRepository<QuizJpqlExample, Long> {

    // JPQL looks like SQL, but queries ENTITIES and their fields (Quiz,
    // q.topic, t.slug), not tables and columns -- Hibernate translates
    // this into real SQL against "quiz"/"topic" underneath, exactly the
    // same translation step covered in "JPA, Hibernate, and Spring Data
    // JPA."
    //
    // The method's own parameter NAMES (topicSlug, language, quizSlug)
    // bind directly to the query's :topicSlug/:language/:quizSlug --
    // Spring Data JPA matches them by name, with no separate @Param
    // annotation required here.
    @Query("select q from Quiz q join fetch q.topic t where t.slug = :topicSlug " +
            "and q.language = :language and q.slug = :quizSlug and q.active = true")
    Optional<QuizJpqlExample> findByTopicSlugAndLanguageAndSlugAndActiveTrue(
            String topicSlug, String language, String quizSlug);
}

class QuizJpqlExample {
}
```

`@Query`, üretilen bir sorgudan, doğrudan JPQL'de -- Jakarta Persistence Query Language -- yazılmış bir taneye geçer. JPQL, SQL'e benzer, ama doğrudan tabloları ve sütunları değil,

ENTITY'LERİ ve onların alanlarını (`Quiz` , `q.topic` , `t.slug`) sorgular; Hibernate onu altında gerçek SQL'e çevirir, "JPA, Hibernate ve Spring Data JPA"nın tanıttığı tam olarak aynı çeviri adımı. Metodun kendi parametre adları (`topicSlug` , `language` , `quizSlug`), sorgunun `:topicSlug` / `:language` / `:quizSlug` yer tutucularına doğrudan bağlanır -- Spring Data JPA bunları isme göre eşleştirir, bu durumda ayrı bir annotation gerekmeden.

> Bir metot parametresinin adını eşleştirerek isimli-parametre bağlaması, yalnızca proje parametre adlarını koruyarak derlendiğinde çalışır (`-parameters` derleyici bayrağı, Spring Boot projelerinin varsayılan olarak açtığı). `@Param("name")` , bu bağlamayı ne olursa olsun açık hale getirir, ve emin olmak istediğinde, ya da bir parametrenin Java adıyla sorgunun yer tutucu adının farklı olması gerektiğinde eklemeye değer.

join fetch ile İlişkili Entity'leri Birleştirmek

Bir ilişkiye dokunan bir sorgu, "Transaction Management"in zaten derinlemesine işlediği tam olarak aynı sorunu -- ilişki bir transaction dışında erişilirse bir `LazyInitializationException` 'ı -- riske atar. `join fetch` , ilgili veriyi AYNI sorguda geri getirerek bunu atlatmanın bir yoludur.

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

import java.util.List;

// Two real join fetch queries from this project -- TopicTranslationRepository
// and QuizQuestionRepository. "join fetch" itself, and the
// LazyInitializationException it avoids, is already covered in full in
// "Transaction Management" -- this only looks at the JPQL syntax itself.
interface JoinFetchRepositoryExample extends JpaRepository<TopicTranslationExample, Long> {

    // A single "join fetch" pulls the related Topic back in the SAME
    // query, instead of a separate query per row later -- exactly the
    // technique "Transaction Management" uses to sidestep
    // LazyInitializationException for this project's sitemap generation.
    @Query("select tt from TopicTranslationExample tt join fetch tt.topic where tt.
published = true")
    List<TopicTranslationExample> findAllPublishedWithTopic();
}

interface QuizQuestionRepositoryExample extends JpaRepository<QuizQuestionExample, Long> {

    // Multiple "join fetch" clauses chain together -- this one pulls back
    // QuizQuestion, its Question, AND that Question's own Topic, all in
    // one query. Chaining joins like this to avoid running one query per
    // relationship, per row, is exactly the shape of problem "Relationships,
    // Fetching, and the N+1 Problem," later in this category, covers in
    // full -- this lesson only needs the JPQL syntax itself.
    @Query("select qq from QuizQuestionExample qq join fetch qq.question q join fetch q.
topic " +
            "where qq.quiz.id = :quizId order by qq.position asc")
    List<QuizQuestionExample> findByQuizIdOrderByPositionAsc(Long quizId);
}

class TopicTranslationExample {
}

class QuizQuestionExample {
}
```

`findAllPublishedWithTopic()`, her `TopicTranslation`'ın `Topic`'ini, daha sonra satır başına ayrı bir sorgu tetiklemek yerine, tek bir sorguda geri getirmek için bir `join fetch` kullanır -- "Transaction Management"in bu projenin kendi sitemap üretimi için zaten gösterdiği tam olarak aynı teknik. `findByQuizIdOrderByPositionAsc(...)`, iki `join fetch` cümlesini zincirler, bir `QuizQuestion`'ı, onun `Question`'ını, VE o `Question`'ın kendi `Topic`'ini, hepsini tek seferde geri getirir. Join'leri özellikle satır başına, ilişki başına bir sorgu çalıştırmaktan kaçınmak için böyle zincirlemek, bu kategoride ilerideki "İlişkiler, Fetching ve N+1 Problemi"nin tam olarak işlediği sorun şekli -- bu ders yalnızca JPQL sözdizimine ihtiyaç duydu.

Veriyi Değiştirmek: @Modifying

Şimdiye kadarki her sorgu bir okumaydı. Birçok satırı aynı anda değiştirmek -- her birini Java'ya yükleyip bir alanı değiştirip tek tek geri kaydetmeden -- bir annotation daha gerektirir.

```
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;

// A plausible extension to this project's real QuestionRepository: bulk-
// rejecting AI-submitted questions that have sat in PENDING_REVIEW too
// long, instead of loading every one of them into Java, changing a field,
// and saving each back individually.
interface QuestionModifyingRepositoryExample extends JpaRepository<QuestionExample, Long> {

    // @Query here is an UPDATE statement, not a SELECT -- @Modifying is
    // REQUIRED to tell Spring Data JPA "this isn't a normal read, execute
    // it as a bulk update/delete instead." Without @Modifying, Spring Data
    // JPA would try to treat the result as a list of entities and fail.
    //
    // @Transactional is also required: a modifying query runs directly
    // against the database, bypassing the persistence context's usual
    // change-tracking entirely -- it needs an active transaction the same
    // way any other write does, exactly as covered in "Transaction
    // Management."
    @Modifying
    @Transactional
    @Query("update QuestionExample q set q.status = 'REJECTED' " +
            "where q.status = 'PENDING_REVIEW' and q.createdAt < :cutoff")
    int rejectStalePendingReview(@Param("cutoff") LocalDateTime cutoff);
}

class QuestionExample {
}
```

Buradaki `@Query`, bir `SELECT` değil bir `UPDATE` ifadesi taşıyor -- `@Modifying`, Spring Data JPA'ya bunun sıradan bir okuma olmadığını, bunun yerine toplu bir güncelleme olarak çalıştırılması gerektiğini söylemek için GEREKLİDİR; olmadan, Spring Data JPA sonucu entity'lere eşlemeye çalışır ve başarısız olur. `@Transactional` de gereklidir: değiştiren bir sorgu, persistence context'in olağan değişiklik izlemesini tamamen atlayarak doğrudan veritabanına karşı çalışır, ve "Transaction Management"in zaten işlediği gibi, başka herhangi bir yazma gibi aktif bir transaction'a ihtiyaç duyar.

Native Query'ler Üzerine Bir Not

`@Query`'nin hiç JPQL içermesi gerekmez -- `nativeQuery = true`, entity modeline karşı değil, gerçek tabloya ve gerçek sütunlarına karşı sorgulanan gerçek SQL'e geçer.

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

import java.util.List;

// This project's own QuestionRepository.findRandomPublishedPool -- real,
// running code behind the Practice feature's random-question selection.
interface QuestionNativeRepositoryExample extends JpaRepository<QuestionNativeExample,
Long> {

    // nativeQuery = true switches from JPQL (which queries entities and
    // their fields) to REAL SQL (which queries the actual "question"
    // table and its actual columns) -- used here because JPQL has no
    // portable RANDOM() function, and this query specifically needs
    // database-level random ordering. The trade-off: a native query is
    // tied to the actual database schema and to PostgreSQL's own SQL
    // dialect, not just to the entity model -- reach for one only when a
    // JPQL query genuinely can't express what's needed, as here.
    @Query(value = "SELECT * FROM question q " +
            "WHERE q.status = 'PUBLISHED' " +
            "AND q.language = :language " +
            "AND (:topicId IS NULL OR q.topic_id = :topicId) " +
            "ORDER BY RANDOM() " +
            "LIMIT :count",
            nativeQuery = true)
    List<QuestionNativeExample> findRandomPublishedPool(@Param("topicId") Long topicId,
                                                        @Param("language") String
language,
                                                        @Param("count") int count);
}

class QuestionNativeExample {
}
```

Bu projenin kendi `findRandomPublishedPool(...)`'u, özellikle JPQL'de taşınabilir bir `RANDOM()` fonksiyonu olmadığı ve bu belirli sorgunun Practice modunun soru seçimi için veritabanı-seviyesi rastgele sıralamaya ihtiyaç duyduğu için bir native query kullanıyor. Ödünleşim gerçektir: bir native query, kodu yalnızca entity modeline değil, gerçek şemaya ve PostgreSQL'in kendi SQL diyalektine bağlar -- yalnızca bir JPQL sorgusunun gerçekten ihtiyaç duyulanı ifade edemediği, burada olduğu gibi bir durumda buna başvur.

Yaygın Yanlış Anlamalar

"Derived query metot adları arkasında gerçek bir mekanizma olmayan, yalnızca uyman gereken bir konvansiyon." Başlangıçta ayrıştırılır ve gerçek bir sorguya

derlenirler -- geçersiz ya da ayrıştırılamayan bir metot adı, sessizce değil, uygulamayı hemen başarısız kılar. **"`@Query` her zaman SQL yazmak demektir."** Varsayılan olarak JPQL demektir, ki bu tabloları ve sütunları değil, entity'leri ve onların alanlarını sorgular -- gerçek SQL'e geçen `nativeQuery = true`'dur, ve bu istisnadır, kural değil. **"Değiştiren bir sorgu diğer herhangi bir repository metodu gibi çalışır."** Çalışmaz -- `@Modifying` olmadan, Spring Data JPA bunu bir okuma yerine toplu bir güncelleme/silme olarak ele alması gerektiğini bilmez.

Sırada Ne Var

Bu dersteki her sorgu, ya bütün bir entity, ya onların bir listesini, ya bir `boolean`'ı, ya da bir `long`'u döndürdü -- büyük bir sonuç kümesini bir client için şekillendirmek, sayfalamak ya da sıralamakla ilgili hiçbir şey işlenmedi. Bu kategoride sıradaki "Sayfalama, Sıralama ve Projeksiyonlar", tam olarak orada devam ediyor: repository seviyesinde `Page<T>` ve `Sort`-farkında sonuçlar döndürmek ("REST API Tasarımı"nın hiç öğretmediği resmin yarısı), ve bir sorgunun buna ihtiyacı olmadığında bütün bir entity'den daha dar bir şey döndürmek.

Best Practices

- Basit, tek-entity koşulları için derived bir query metodunu tercih et -- yalnızca derived bir isim hantal olacağında ya da sorguyu hiç ifade edemeyeceğinde `@Query`'ye başvur.
- Gerçekten yalnızca bir boolean ya da bir sayıya ihtiyaç olduğunda, `findBy...().isPresent()` / `.size()` yerine `existsBy...` / `countBy...` kullan.
- `@Modifying`'i her zaman `@Transactional`'la eşleştir -- değiştiren bir sorgu, diğer herhangi bir yazma gibi aktif bir transaction'a ihtiyaç duyar.
- Yalnızca JPQL bir şeyi gerçekten ifade edemediğinde (`RANDOM()`'da olduğu gibi) bir native query'ye başvur -- bu, o yeteneği entity-modeli bağımsızlığıyla takas eder.

Yaygın Hatalar

- Entity'nin özelliklerinin gerçekte desteklemediği bir derived metot adı yazıp, bir çalışma zamanı hatası yerine bir başlangıç hatasına şaşırmak.
- Derived bir metot adında parametreleri koşullara karşı yanlış saymak -- `ActiveTrue` gibi boolean bir koşul hiç parametre tüketmez.
- Bir `UPDATE` / `DELETE` `@Query`'de `@Modifying`'i, ya da onunla birlikte `@Transactional`'ı unutmak.
- Hiçbir gerçek neden olmadan, JPQL'in sağladığı entity-modeli bağımsızlığını kaybederek, varsayılan olarak JPQL yerine bir native query'ye başvurmak.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bir repository metodunun sorgusu iki yerden birinden gelir: adından otomatik olarak çıkarılır, ya da `@Query` ile açıkça yazılır.
- Derived bir isim parça parça ayrıştırılır -- `findBy` / `existsBy` / `countBy`, `And` / `Or` ile birleşen koşullar, `ActiveTrue` gibi boolean literaller, ve `OrderBy` -- ve uygulama başlangıcında doğrulanır.
- `@Query`, varsayılan olarak JPQL'e geçer -- entity'leri ve alanlarını sorgular, altında Hibernate tarafından SQL'e çevrilir -- ya da `nativeQuery = true` ile gerçek SQL'e.
- JPQL'de `join fetch`, bir ilişkiyi aynı sorguda geri getirir, sonradan bir `LazyInitializationException`'ı ve (daha büyük ölçekte) N+1 problemini önler.
- `@Modifying` (`@Transactional`'la eşleştirilmiş), satırları okumak yerine toplu güncelleyen ya da silen bir `@Query` için gereklidir.

Cheat Sheet

```
// Derived query metotları
List<CodeExample> findByTopicIdOrderBySortOrderAsc(Long topicId);
Optional<CodeExample> findByTopicIdAndExampleName(Long topicId, String name);
Optional<Quiz> findFirstByTopicIdAndLanguageAndActiveTrueOrderByIdAsc(Long topicId, String
language);
boolean existsByTopicIdAndLanguage(Long topicId, String language);
long countByTopicId(Long topicId);

// JPQL ile @Query, metot parametre adına göre bağlanan isimli parametreler
@Query("select q from Quiz q join fetch q.topic t where t.slug = :topicSlug")
Optional<Quiz> findByTopicSlug(String topicSlug);

// Değiştiren sorgu
@Modifying
@Transactional
@Query("update Question q set q.status = 'REJECTED' where q.status = 'PENDING_REVIEW'")
int rejectAllPendingReview();

// Native query
@Query(value = "SELECT * FROM question WHERE status = 'PUBLISHED' ORDER BY RANDOM() LIMIT :
count",
       nativeQuery = true)
List<Question> findRandomPublished(@Param("count") int count);
```

Terimler Sözlüğü

- **Derived query metodu**: Spring Data JPA'nın adını ayrıştırarak sorgusunu otomatik olarak inşa ettiği bir repository metodu.
- **JPQL (Jakarta Persistence Query Language)**: SQL'e benzeyen ama doğrudan tabloları ve sütunları değil, entity'leri ve onların alanlarını hedefleyen bir sorgu dili.
- **@Query**: bir repository metodu için açık bir JPQL (ya da `nativeQuery = true` ile native SQL) sorgusu sağlayan annotation.
- **join fetch**: ilişkili bir entity'yi aynı sorguda getiren, o ilişki için daha sonra ayrı bir sorgudan kaçınan bir JPQL cümlesi.

- **@Modifying**: bir okuma yerine toplu bir `UPDATE` / `DELETE` yapan bir `@Query` üzerinde gerekli olan annotation.
- **Native query**: entity modeline karşı JPQL'de değil, gerçek şemaya karşı gerçek SQL'de yazılmış bir `@Query`.