Dynamic Queries with Specifications

"REST API Tasarımı"nın bellek içinde bıraktığı isteğe bağlı filtrelemeyi, JPA'nın Criteria API'si (Root/CriteriaQuery/CriteriaBuilder/Predicate) üzerine ince bir sarmalayıcı olan Specification interface'i, JpaSpecificationExecutor, Specification.where/and/or ile birleştirme, ve findAll(Specification, Pageable) ile gerçek sayfalamayla birleştirme -- bu projenin gerçek Topic entity'siyle. Spring Data JPA kategorisinin 5.'si.

İleri 35 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/dynamic-queries-with-specifications

---

"REST API Tasarımı"nın filtreleme örneği, Java'da isteğe bağlı koşullar inşa etti -- filtre başına bir `Predicate<Topic>`, kendi query parametresi yokken varsayılan olarak "her şeyle eşleş" olan, bellek-içi bir `Stream` üzerinde `.and(...)` ile birleştirilen. Kendi yorumu bu sınırlama konusunda açıktı: "gerçek bir repository bunu bir WHERE cümlesine... ya da bu kadar dinamik durumlar için bir JPA `Specification`'a iter." Bu ders, o gerçek repository-seviyesi implementasyon.

"REST API Tasarımı"nın Bitirmediği: Filtrelemeyi Veritabanına İtmek

Bellek-içi versiyon çalıştı, ama yalnızca örnek listesinde üç `Topic` olduğu için. Gerçek bir tabloya karşı bu şekilde filtrelemek, önce her satırı getirmek, sonra Java'da çoğunu atmak demektir -- veritabanı hiçbir zaman bir index kullanamaz, ve filtrelenmemiş her satır yine de ağdan geçmek zorundadır. Fikrin kendisi -- yokken hiçbir şey yapmayan, başkalarıyla birleşen isteğe bağlı bir koşul -- zaten doğruydu; eksik olan, aynı fikri bir `Stream.filter(...)` yerine gerçek, üretilmiş bir `WHERE` cümlesi olarak çalıştırmak.

Query Metotları ve @Query Burada Neden Yetmiyor

"Query Methods and JPQL with @Query", bir sorgu elde etmenin iki yolunu işledi: Spring Data JPA'nın ayrıştırdığı bir isim, ya da doğrudan yazdığın JPQL. İkisi de derleme zamanında SABİTTİR -- bir metodun adı koşullarını bir kez bildirir, ve bir `@Query` string'i her çalıştığında aynı metindir. İkisi de "kategoriye göre filtrele, ama yalnızca bir kategori gerçekten sağlandıysa, ve zorluğa göre de, ama o da öyleyse" ifade edemez -- aktif koşullar KÜMESİ, bir istek gerçekten gelene kadar bilinmez. Bu, gerçekten farklı bir sorun, ve gerçekten farklı bir araç gerektiriyor.

Specification Nedir?

Bir `Specification<T>`, tek bir WHERE koşulunu temsil eden küçük bir fonksiyonel interface'tir -- sabit bir JPQL string'i ya da bir metot adı olarak değil, onu inşa eden Java kodu olarak ifade edilir.

```
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Predicate;
import jakarta.persistence.criteria.Root;
import org.springframework.data.jpa.domain.Specification;

// A Specification<Topic> is a small functional interface -- Java code
// that BUILDS a Predicate (a WHERE condition) instead of a query being
// written out as text, the way JPQL and native SQL both are.
class SingleSpecificationExample {

    static class Topic {
        String difficulty;
    }

    // Root<Topic> is how the Criteria API refers to "the Topic being
    // queried" -- root.get("difficulty") is the Criteria API's way of
    // writing t.difficulty. CriteriaBuilder is what actually constructs a
    // Predicate -- here, an equality check -- from that path and a value.
    static Specification<Topic> hasDifficulty(String difficulty) {
        return (Root<Topic> root, CriteriaQuery<?> query, CriteriaBuilder cb) ->
                cb.equal(root.get("difficulty"), difficulty);
    }

    public static void main(String[] args) {
        Specification<Topic> spec = hasDifficulty("ADVANCED");
        System.out.println(spec != null);
        // This alone doesn't run anything yet -- a Specification only
        // BUILDS a Predicate; something still has to hand it to a
        // repository, covered next in "JpaSpecificationExecutor."
    }
}
```

`hasDifficulty(...)`, bir `Specification<Topic>` döndürür -- bir `Root<Topic>`, bir `CriteriaQuery`, ve bir `CriteriaBuilder` verildiğinde bir `Predicate` üreten bir lambda. Bu noktada hiçbir şey henüz çalışmaz; bir `Specification`, yalnızca bir koşulun NASIL İNŞA EDİLECEĞİNİ tarif eder. Bir şeyin bir şey yapması için onu hâlâ bir repository'ye teslim etmesi gerekir.

Bir Specification'ın Altındaki Criteria API

`Root`, `CriteriaQuery`, `CriteriaBuilder`, ve `Predicate`, JPA'nın kendi Criteria API'sinden gelir -- sorguları query metni yerine Java nesnelerinden inşa etmek için ("JPA, Hibernate ve Spring Data JPA"da işlenen, JPA-spesifikasyon anlamındaki) spesifikasyon. `Root<Topic>`, "sorgulanan `Topic`"i ifade eder -- `root.get("difficulty")`, Criteria API'nin `t.difficulty` yazmanın yolu. `CriteriaBuilder`, gerçekte bir yoldan ve bir değerden bir `Predicate` (`cb.`

`equal(...)`, ve diğer karşılaştırmalar için birçok benzer metot) inşa eden şeydir. Spring Data JPA'nın `Specification`'ı, bu API'nin üzerinde ince, kullanışlı bir sarmalayıcıdır -- onun yerini almaz, tam olarak "JPA, Hibernate ve Spring Data JPA"nın genel olarak Spring Data JPA'nın JPA ya da Hibernate'in yerini almadığını işlediği aynı ilişki.

JpaSpecificationExecutor: Bir Repository'nin Specification Kabul Etmesini Sağlamak

Bir `Specification`, bir koşulu tarif eder, ama bir repository'nin bir taneyi kabul etmeyi açıkça seçmesi gerekir.

```
import org.springframework.data.jpa.domain.Specification;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;

import java.util.List;

// Extending JpaSpecificationExecutor<Topic> ALONGSIDE JpaRepository<Topic, Long>
// is what actually gives this project's own TopicRepository the ability to
// accept a Specification at all -- without it, a Specification has
// nothing to be run against.
interface TopicSpecificationRepositoryExample
        extends JpaRepository<TopicSpecExample, Long>,
JpaSpecificationExecutor<TopicSpecExample> {

    // No new methods are declared here -- JpaSpecificationExecutor already
    // contributes findAll(Specification), findOne(Specification),
    // count(Specification), and more, the same way CrudRepository already
    // contributed save/findById/findAll in "Entities and the Repository
    // Abstraction."
}

class TopicSpecExample {
}

class JpaSpecificationExecutorExample {
    public static void main(String[] args) {
        // repository.findAll(hasDifficulty("ADVANCED")) would now generate
        // a real SELECT ... WHERE difficulty = 'ADVANCED' -- the
        // Specification from SingleSpecificationExample, finally given
        // something to run against.
        System.out.println("see SingleSpecificationExample for the Specification itself");
    }
}
```

`JpaRepository<Topic, Long>`'un yanı sıra `JpaSpecificationExecutor<Topic>`'i genişletmek, bir repository'ye gerçekte `findAll(Specification)`, `findOne(Specification)`, `count(Specification)`, ve daha fazlasını verir -- "Entities and the Repository Abstraction"ın `CrudRepository`'nin `save` / `findById` / `findAll` kattığını işlediğiyle tam olarak aynı şekilde, bedavaya miras alınır. `JpaSpecificationExecutor` olmadan, bir `Specification`'ın gerçekte karşı çalışacağı hiçbir şey yoktur.

Specification'ları Birleştirmek: where, and, or

Bir `Specification`'ın gerçek değeri, birkaçı tek, daha büyük bir koşulda birleştiğinde ortaya çıkar.

```
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Predicate;
import jakarta.persistence.criteria.Root;
import org.springframework.data.jpa.domain.Specification;

class CombiningSpecificationsExample {

    static class Topic {
        String category;
        String difficulty;
    }

    static Specification<Topic> hasCategory(String category) {
        return (Root<Topic> root, CriteriaQuery<?> query, CriteriaBuilder cb) ->
                cb.equal(root.get("category"), category);
    }

    static Specification<Topic> hasDifficulty(String difficulty) {
        return (Root<Topic> root, CriteriaQuery<?> query, CriteriaBuilder cb) ->
                cb.equal(root.get("difficulty"), difficulty);
    }

    public static void main(String[] args) {
        // Specification.where(...) starts a chain; .and(...)/.or(...)
        // combine two Specifications into a single, larger one -- exactly
        // the way DynamicFilterExample ("REST API Design") combined two
        // Predicates with .and(...), except this builds a real SQL WHERE
        // clause instead of filtering an in-memory Stream.
        Specification<Topic> spec = Specification
                .where(hasCategory("spring-mvc"))
                .and(hasDifficulty("ADVANCED"));
        // Generates: WHERE category = 'spring-mvc' AND difficulty = 'ADVANCED'

        Specification<Topic> either = Specification
                .where(hasCategory("spring-mvc"))
                .or(hasDifficulty("ADVANCED"));
        // Generates: WHERE category = 'spring-mvc' OR difficulty = 'ADVANCED'

        System.out.println(spec != null && either != null);
    }
}
```

`Specification.where(...)`, bir zincir başlatır; `.and(...)` / `.or(...)`, iki `Specification`'ı tek, daha büyük bir tanede birleştirir -- "REST API Tasarımı"ndaki `DynamicFilterExample`'ın `Predicate.and(...)` ile zaten kullandığı tam olarak aynı şekil, yalnızca bu, bellek-içi bir `Stream`'i filtrelemek yerine gerçek bir SQL `WHERE` cümlesi üretir.

Veritabanına İtilmiş İsteğe Bağlı Filtreler

Bu, "REST API Tasarımı"nın ertelenmiş vaadini gerçekten yerine getiren parça -- aynı isteğe bağlı-filtre şekli, artık gerçek SQL üretiyor.

```
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Predicate;
import jakarta.persistence.criteria.Root;
import org.springframework.data.jpa.domain.Specification;

// This is the SAME shape of problem "REST API Design"'s DynamicFilterExample
// solved with an in-memory Predicate chain -- category and difficulty are
// each OPTIONAL, and an absent filter should exclude nothing. Here, the
// exact same idea is pushed into the database instead.
class OptionalFiltersSpecificationExample {

    static class Topic {
        String category;
        String difficulty;
    }

    // Specification.where(null) is a genuinely useful starting point -- it
    // behaves as a no-op "match everything" Specification, exactly the
    // way DynamicFilterExample's absent filters defaulted to "t -> true."
    static Specification<Topic> search(String category, String difficulty) {
        Specification<Topic> spec = Specification.where(null);

        if (category != null) {
            spec = spec.and((Root<Topic> root, CriteriaQuery<?> query, CriteriaBuilder cb) -
>
                    cb.equal(root.get("category"), category));
        }
        if (difficulty != null) {
            spec = spec.and((Root<Topic> root, CriteriaQuery<?> query, CriteriaBuilder cb) -
>
                    cb.equal(root.get("difficulty"), difficulty));
        }

        return spec;
        // category=null, difficulty=null   -> WHERE 1=1 (no filtering at all)
        // category="spring-mvc", diff=null -> WHERE category = 'spring-mvc'
        // both supplied                    -> WHERE category = ... AND difficulty = ...
    }

    public static void main(String[] args) {
        System.out.println(search("spring-mvc", null) != null);
        System.out.println(search(null, null) != null);
    }
}
```

`Specification.where(null)`, gerçekten yararlı bir başlangıç noktasıdır -- `DynamicFilterExample`'ın yokluğundaki filtrelerinin `t -> true`'ya varsayılan olarak oynadığı role tam olarak eşdeğer, bir no-op, "her şeyle eşleş" `Specification`'ı gibi davranır. Her `if (category != null)` / `if (difficulty != null)` kontrolü, yalnızca o filtre gerçekten sağlandığında bir `.and`

`(...)` daha ekler -- hiçbiri sağlanmadığında, üretilen sorgu hiçbir şeyi filtrelemez; ikisi de sağlandığında, ikisini de filtreler.

Specification'lar Pageable ile Birlikte

Dinamik filtreleme ve gerçek sayfalama ayrı mekanizmalar değildir -- tek bir repository çağrısında birleşirler.

```
import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Predicate;
import jakarta.persistence.criteria.Root;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;

// JpaSpecificationExecutor's findAll ALSO accepts a Pageable, exactly the
// same Page<T> type covered in "Pagination, Sorting, and Projections" --
// dynamic filtering and real pagination aren't separate mechanisms, they
// combine into a single call.
interface TopicSearchRepositoryExample
        extends JpaRepository<TopicSearchExample, Long>,
JpaSpecificationExecutor<TopicSearchExample> {
}

class TopicSearchExample {
    String category;
    String difficulty;
}

class SpecificationWithPageableExample {

    static Specification<TopicSearchExample> hasCategory(String category) {
        return (Root<TopicSearchExample> root, CriteriaQuery<?> query, CriteriaBuilder cb) -
>
                cb.equal(root.get("category"), category);
    }

    public static void main(String[] args) {
        Specification<TopicSearchExample> spec = Specification.where(hasCategory("spring-
mvc"));
        Pageable pageable = PageRequest.of(0, 10);

        // repository.findAll(spec, pageable) would generate a filtered,
        // paged query PLUS a filtered count query -- the same two-query
        // shape "Pagination, Sorting, and Projections" already covered,
        // now with a dynamic WHERE clause instead of a fixed one.
        System.out.println(spec != null && pageable != null);
    }
}
```

`JpaSpecificationExecutor`'ın `findAll`'ı da bir `Pageable` kabul eder, "Pagination, Sorting, and Projections"in zaten işlediği tam olarak aynı tür. `repository.findAll(spec, pageable)`, filtrelenmiş, sayfalanmış bir sorgu ARTI filtrelenmiş bir count sorgusu üretir -- o dersteki birebir aynı iki-sorgu şekli, artık sabit bir `WHERE` cümlesi yerine dinamik bir taneyle.

Yaygın Yanlış Anlamalar

"Bir `Specification`, bir sorgudur." Tek bir koşulun bir tarifidir -- `JpaSpecificationExecutor`'ı genişleten bir repository'ye teslim edilene kadar hiçbir şey çalışmaz. **"`Specification`, JPA'dan tamamen ayrı bir mekanizma."** JPA'nın kendi Criteria API'sinin (`Root` / `CriteriaQuery` / `CriteriaBuilder` / `Predicate`) üzerinde ince bir sarmalayıcıdır -- Spring Data JPA'nın her yerde JPA'yla olan aynı ilişkisi. **"Dinamik filtreleme her zaman Specification'lara ihtiyaç duyar."** Duymaz -- küçük, sabit bir isteğe bağlı koşul kümesine sahip bir sorgu bazen `:param IS NULL OR ...` deseniyle (bu projenin kendi `QuestionRepository`'sinde kullanılan) tek bir JPQL `@Query` ile ifade edilebilir; `Specification`, koşulların sayısı ya da şekli istek başına gerçekten değiştiğinde yerini hak eder.

Sırada Ne Var

Bu kategoride şimdiye kadar işlenen her sorgu -- derived metotlar, JPQL, projection'lar, `Specification`'lar -- tek bir istek içinde basit bir okumaydı. Bu kategoride sıradaki "İlişkiler, Fetching ve N+1 Problemi", bir sorgunun NE döndürdüğünden, bir entity'nin kendi ilişkilerinin NASIL yüklendiğine geçiyor -- tamamen doğru bir sorgunun bile hâlâ tetikleyebileceği bir performans sorunu (N+1) dahil.

Best Practices

- Aktif filtre koşulları KÜMESİ, bir istek gelene kadar gerçekten bilinmediğinde bir `Specification`'a başvur -- query metotları ya da `@Query`'nin varsayılan bir yerine geçeni olarak değil.
- İsteğe bağlı bir filtre zincirini `Specification.where(null)` ile başlat, ve yalnızca o filtrenin değeri gerçekten mevcut olduğunda filtre başına bir `.and(...)` ekle.
- Tek tek `Specification`'ları küçük ve neyi kontrol ettiklerine göre adlandırılmış tut (`hasCategory`, `hasDifficulty`) -- tek, büyük, monolitik bir `Specification` yazmak yerine onları `.and(...)` / `.or(...)` ile birleştir.
- `findAll(Specification, Pageable)`'ın var olduğunu hatırla -- dinamik filtreleme ve sayfalama, iki ayrı adım değil, tek bir çağrıda birleşir.

Yaygın Hatalar

- Bir derived metodun ya da düz bir `@Query`'nin aynı şeyi daha doğrudan söyleyeceği, küçük, gerçekten sabit bir koşul kümesine sahip bir sorgu için `Specification`'a başvurmak.

- `JpaSpecificationExecutor`'ı tamamen unutup, bir repository'nin çağıracak bir `findAll(Specification)`'ı olmamasına şaşırmak.
- `Specification.where(null)`'dan başlamadan yeni bir `Specification` zinciri inşa etmek, ve "hiç filtre yok" senaryosunu ayrıca özel olarak ele almak zorunda kalmak.
- Bir `Specification`'ı kendi başına çalışan bir şey gibi ele almak, gerçekte bir `Predicate` üretmek için hâlâ (sorgu zamanında repository tarafından sağlanan) bir `Root` / `CriteriaBuilder`'a ihtiyaç duyan bir şey yerine.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bir `Specification<T>`, JPA'nın kendi Criteria API'sinden (`Root`, `CriteriaQuery`, `CriteriaBuilder`, `Predicate`) inşa edilen, tek bir `WHERE` koşulunu tarif eden Java kodudur.
- Bir repository'nin bir `Specification`'ı kabul etmesi için (`JpaRepository<T, ID>`'nin yanı sıra) `JpaSpecificationExecutor<T>`'ı genişletmesi gerekir.
- `Specification.where(...).and(...)/.or(...)`, birden fazla koşulu tek birinde birleştirir, bellekte `Predicate`'leri birleştirmekle aynı şekil, artık gerçek SQL üretiyor.
- `Specification.where(null)`, isteğe bağlı filtreleri bir seferde bir `.and(...)` inşa etmek için gerçekten yararlı bir no-op başlangıç noktasıdır.
- `findAll(Specification, Pageable)`, dinamik filtrelemeyi tek bir repository çağrısında gerçek sayfalamayla birleştirir.

Cheat Sheet

```
// Tek bir Specification
static Specification<Topic> hasDifficulty(String difficulty) {
    return (root, query, cb) -> cb.equal(root.get("difficulty"), difficulty);
}

// Specification kabul eden bir repository
interface TopicRepository extends JpaRepository<Topic, Long>,
JpaSpecificationExecutor<Topic> {}

// Specification'ları birleştirmek
Specification<Topic> spec = Specification.where(hasCategory("spring-mvc")).and(hasDifficulty
("ADVANCED"));

// Veritabanına itilmiş isteğe bağlı filtreler
Specification<Topic> spec = Specification.where(null);
if (category != null)   spec = spec.and(hasCategory(category));
if (difficulty != null) spec = spec.and(hasDifficulty(difficulty));

// Dinamik filtreleme + gerçek sayfalama birlikte
Page<Topic> page = repository.findAll(spec, PageRequest.of(0, 10));
```

Terimler Sözlüğü

- **Specification<T>**: sabit bir sorgu string'i yerine JPA'nın Criteria API'siyle inşa edilen, tek bir sorgu koşulunu temsil eden bir fonksiyonel interface.
- **Criteria API**: sorguları query metni yerine Java nesnelerinden inşa etmek için JPA'nın kendi API'si (`Root`, `CriteriaQuery`, `CriteriaBuilder`, `Predicate`).
- **JpaSpecificationExecutor**: bir repository'nin, `Specification`'ları kabul etmek için `JpaRepository`'nin yanı sıra genişletmesi gereken interface.
- **Specification.where(null)**: isteğe bağlı bir filtre zincirinin temeli olarak yararlı, no-op, "her şeyle eşleş" başlangıç `Specification`'ı.