Relationships, Fetching, and the N+1 Problem

@OneToMany/@ManyToMany mapping ("Transaction Management"in zaten kapsamlıca işlediği @ManyToOne'un ötesinde), cascade türleri (PERSIST/MERGE/REMOVE/ALL) ve orphanRemoval, N+1 probleminin somut gösterimi, ve @EntityGraph/batch fetching /projection'larla üç farklı düzeltme -- bu projenin gerçek Topic/Category/QuizQuestion sınıflarıyla. Spring Data JPA kategorisinin 6.'sı.

İleri 40 dk

Son güncelleme: 24 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/relationships-fetching-and-n-plus-1

Bu kategoride şimdiye kadar işlenen her entity, en fazla dışarı doğru işaret eden tek bir `@ManyToOne`'a sahipti -- `Topic.category`, `Category.course`. "Transaction Management", bu tek ilişkiye lazy olarak, bir transaction dışında erişildiğinde ne olduğunu zaten işledi. Bu ders aynı anda üç yönde daha ileri gidiyor: bir ilişkinin alabileceği DİĞER şekiller, bir entity'yi kaydetmenin ya da silmenin başka birine nasıl yayılabileceği (cascade), ve şimdiye kadar hiç adlandırılmamış bir performans sorunu -- tamamen doğru bir ilişki mapping'i kümesinin bile sessizce tetikleyebileceği.

"Transaction Management"in Zaten İşlediği, ve Bu Dersin Eklediği

`LazyInitializationException`, `join fetch`, ve `spring.jpa.open-in-view`, bu projenin gerçek `Topic` / `Category` / `Course` ilişkileri kullanılarak zaten tam olarak işlendi -- hiçbiri burada tekrarlanmıyor. Yeni olan: `@OneToMany` ve `@ManyToMany` (şimdiye kadar yalnızca `@ManyToOne` ortaya çıktı), cascade türleri ve `orphanRemoval`, ve N+1 problemi -- lazy loading'in (zaten işlenen) mümkün kıldığı ama kendi başına açıklamadığı, spesifik, somut bir performans tuzağı.

@ManyToOne'un Diğer Tarafı: @OneToMany

Bu projenin gerçek `Topic` entity'si, `Category`'ye işaret eden bir `@ManyToOne`'a sahip -- ama `Category`'de şu anda hiçbir şey geriye işaret etmiyor. `@OneToMany`, o eksik diğer taraf.

```
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;

import java.util.ArrayList;
import java.util.List;

// This project's real Topic already has @ManyToOne(fetch = LAZY) Category
// -- covered in "Entities and the Repository Abstraction" and in
// "Transaction Management"'s LazyInitializationException section. What's
// new here is the OTHER side of that same relationship: Category "has
// many" Topics.
@Entity
class CategoryWithTopicsExample {
    @Id
    private Long id;

    // mappedBy = "category" points at the FIELD on TopicExample that owns
    // this relationship -- @OneToMany is the "many" side's mirror image,
    // not a second, independent mapping. The foreign key (category_id)
    // still lives on the topic table, exactly as it already does in this
    // project's real schema; @OneToMany adds no column of its own.
    @OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
    private List<TopicExample> topics = new ArrayList<>();

    List<TopicExample> getTopics() {
        return topics;
    }
}

@Entity
class TopicExample {
    @Id
    private Long id;

    // This is the OWNING side -- the side with the actual foreign-key
    // column. This project's real Topic entity has exactly this field.
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id", nullable = false)
    private CategoryWithTopicsExample category;
}
```

`mappedBy = "category"`, bu ilişkiyi gerçekte sahiplenen `TopicExample`'daki ALANA işaret eder -- `@OneToMany`, var olan bir `@ManyToOne`'un ayna görüntüsüdür, kendi sütunu olan ikinci, bağımsız bir mapping değil. Foreign key (`category_id`), bu projenin gerçek şemasında zaten olduğu gibi, yalnızca `topic` tablosunda yaşamaya devam eder; `@OneToMany` hiçbir yere kendi sütununu eklemez.

@ManyToMany ve Join Table'lar

Bazı ilişkilerin doğal bir "sahip" tarafı hiç yoktur -- her iki tarafın tablosu da bir foreign key için doğru yer değildir, çünkü her iki taraf da diğerinin birçoğuyla ilişkilenebilir.

```
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.JoinTable;
import jakarta.persistence.ManyToMany;

import java.util.HashSet;
import java.util.Set;

// A plausible new relationship this project doesn't currently have: a
// Topic can carry several Tags, and a Tag can label several Topics --
// neither side "owns" the other the way a foreign key does in
// @ManyToOne, so the relationship needs its own JOIN TABLE.
@Entity
class TopicWithTagsExample {
    @Id
    private Long id;

    // @JoinTable describes that separate table explicitly -- "topic_tag",
    // with a topic_id column and a tag_id column, neither of which lives
    // on "topic" or "tag" themselves.
    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(
            name = "topic_tag",
            joinColumns = @JoinColumn(name = "topic_id"),
            inverseJoinColumns = @JoinColumn(name = "tag_id"))
    private Set<TagExample> tags = new HashSet<>();

    Set<TagExample> getTags() {
        return tags;
    }
}

@Entity
class TagExample {
    @Id
    private Long id;
    private String name;
}

// This project's REAL QuizQuestion is a different, often better, pattern
// for the same underlying idea (Quiz and Question relate to many of each
// other): instead of a raw @ManyToMany, it's a full ENTITY of its own,
// with its own "position" column -- a plain @ManyToMany join table has no
// room for extra data ABOUT the relationship itself, only the link.
// Reach for an explicit join entity, as this project already does,
// whenever the relationship needs to carry data beyond just "these two
// are linked."
```

`@JoinTable`, ayrı bir tabloyu (`topic_tag`) açıkça tarif eder, ne `topic`'te ne `tag`'de yaşayan bir `topic_id` sütunu ve bir `tag_id` sütunuyla. Bu projenin kendi gerçek `QuizQuestion`'ı, aynı temel fikir için (`Quiz` ve `Question`, birbirinin birçoğuyla ilişkilenir) ilişkili ama farklı, genelde daha iyi bir yaklaşım gösteriyor -- düz bir `@ManyToMany` yerine, kendi `position` sütunu olan tam bir entity. Düz bir `@ManyToMany` join table'ının, ilişkinin KENDİSİ HAKKINDA veriye yer yoktur, yalnızca bağlantıya -- ilişkinin bundan fazlasını taşıması gerektiğinde, bu projenin zaten yaptığı gibi, açık bir join entity'sine başvur.

Cascade Türleri ve orphanRemoval

Bir entity'yi kaydetmek ya da silmek, otomatik olarak ilişkili olduğu entity'lere dokunmaz -- `cascade` ve `orphanRemoval`, bunu bilinçli olarak yapan şeydir.

```
import jakarta.persistence.CascadeType;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;

import java.util.ArrayList;
import java.util.List;

@Entity
class CategoryCascadeExample {
    @Id
    private Long id;

    // Without "cascade", saving a new Category does NOT automatically
    // save the Topics added to its list -- each would need its own
    // explicit topicRepository.save(...) call. CascadeType.PERSIST makes
    // categoryRepository.save(category) also save every Topic in
    // "topics", in the same operation.
    //
    // CascadeType.REMOVE makes deleting a Category also delete every
    // Topic that still belongs to it -- appropriate here, since a Topic
    // genuinely can't exist without a Category (category_id is
    // nullable = false on this project's real schema). CascadeType.MERGE
    // extends that same "do it for the children too" idea to updates.
    // CascadeType.ALL is shorthand for PERSIST + MERGE + REMOVE (and two
    // less common ones) together.
    @OneToMany(mappedBy = "category", fetch = FetchType.LAZY,
            cascade = CascadeType.ALL, orphanRemoval = true)
    private List<TopicCascadeExample> topics = new ArrayList<>();

    void addTopic(TopicCascadeExample topic) {
        topics.add(topic);
        topic.setCategory(this);
    }

    // orphanRemoval = true covers a DIFFERENT case than CascadeType.REMOVE:
    // removing a Topic from THIS LIST (topics.remove(someTopic)) -- without
```

```
    // deleting the Category itself at all -- deletes that Topic from the
    // database too, because it no longer belongs to anything. Without
    // orphanRemoval, that Topic would simply become an orphaned row,
    // still present but disconnected.
    void removeTopic(TopicCascadeExample topic) {
        topics.remove(topic);
    }
}

@Entity
class TopicCascadeExample {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id", nullable = false)
    private CategoryCascadeExample category;

    void setCategory(CategoryCascadeExample category) {
        this.category = category;
    }
}
```

`CascadeType.PERSIST`, bir `Category`'yi kaydetmenin, listesine yeni eklenen her `Topic`'i de aynı işlemde kaydetmesini sağlar. `CascadeType.REMOVE`, bir `Category`'yi silmenin, ona hâlâ bağlı her `Topic`'i de silmesini sağlar -- burada makul bir seçim, çünkü bu projenin gerçek şeması zaten `category_id`'yi `nullable = false` yapıyor, yani bir `Topic` gerçekten birisiz var olamaz. `CascadeType.ALL`, `PERSIST` / `MERGE` / `REMOVE`'un (ve iki daha az yaygın cascade türünün) birlikte kısaltmasıdır. `orphanRemoval = true`, tamamen farklı bir durumu kapsar: `Category`'yi hiç silmeden, bir `Topic`'i `topics` LİSTESİNDEN çıkarmak, o `Topic`'i veritabanından da siler, çünkü artık hiçbir şeye ait değildir; o olmadan, o `Topic` basitçe bağlantısız, sahipsiz bir satır hâline gelirdi.

> "Çok" tarafının gerçekten kendi bağımsız yaşam döngüsü olduğu bir ilişkide `CascadeType.REMOVE` / `ALL` (`Category`'nin silmeyi `Topic`'in kendi `CodeExample`'larına yaydığını hayal et, bazı tasarımlarda bunlar tartışmalı olarak tek bir topic'ten daha uzun yaşayabilmeli) istenenden çok daha fazlasını silebilir. Buna bilinçli olarak, ilişkili entity'nin gerçekten ebeveyni olmadan var olup olamayacağına göre başvur -- her `@OneToMany`'de bir varsayılan olarak değil.

N+1 Problemi, Gösterilmiş

Bu, yukarıdaki mapping mekaniğinin mümkün kıldığı, somut, kaçırması kolay performans tuzağı.

```
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
```

```
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

// This is the concrete shape of the N+1 problem -- a correct, perfectly
// legal piece of code that still runs far more queries than it looks like
// it should.
interface CategoryNPlusOneRepositoryExample extends JpaRepository<CategoryNPlusOneExample,
Long> {
}

@Entity
class CategoryNPlusOneExample {
    @Id
    private Long id;

    @OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
    private List<TopicNPlusOneExample> topics;

    List<TopicNPlusOneExample> getTopics() {
        return topics;
    }
}

@Entity
class TopicNPlusOneExample {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    private CategoryNPlusOneExample category;
}

class NPlusOneProblemExample {

    static void printAllTopicCounts(CategoryNPlusOneRepositoryExample repository) {
        // Query #1: one SELECT fetching every Category -- say there are 7.
        List<CategoryNPlusOneExample> categories = repository.findAll();

        for (CategoryNPlusOneExample category : categories) {
            // category.getTopics() is LAZY -- accessing it here, inside the
            // loop, triggers a SEPARATE query, right now, for THIS one
            // category's topics alone.
            //
            // Query #2, #3, #4, #5, #6, #7, #8: one MORE SELECT, per
            // category, run one at a time as the loop reaches each one.
            System.out.println(category.getTopics().size());
        }
        // Total: 1 query to fetch the categories ("N" categories), PLUS
        // one query PER category to fetch its topics -- "N+1" queries for
        // what looks, at a glance, like a single loop over already-fetched
        // data. With 7 categories, that's 8 queries; with 100, it's 101.
    }
}
```

`printAllTopicCounts(...)`, tamamen sıradan görünüyor -- her `Category`'yi getir, üzerlerinde döngü kur, her birinin konularını oku. Ama `category.getTopics()` lazy'dir, ve döngü içinde ona erişmek, tam o anda, yalnızca o kategori için AYRI bir sorguyu tetikler. Bir sorgu kategorileri getirir; döngü içinde kategori başına bir sorgu DAHA çalışır -- 7 kategoriyle, bu tek bir getirme gibi okunan şey için toplam 8 sorgudur; 100'le, 101'dir. Buradaki hiçbir şey sıradan anlamda bir hata değildir -- her satır, tek başına doğrudur -- sorun tamamen doğru kodun sonunda veritabanına kaç gidiş-geliş yaptığıyla ilgilidir.

@EntityGraph ile Düzeltmek

Bir düzeltme: ilişkiyi, kullanıldığı her yerde fetch türünü değiştirmeden, özellikle bu TEK sorgu için eagerly join etmek.

```
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;
import org.springframework.data.jpa.repository.EntityGraph;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.List;

// @EntityGraph fixes the N+1 problem from NPlusOneProblemExample for THIS
// one query, without changing the relationship's fetch type globally
// (which would affect every other query using this entity too).
interface CategoryEntityGraphRepositoryExample extends JpaRepository<CategoryGraphExample,
Long> {

    // attributePaths names exactly which lazy relationship(s) to pull back
    // eagerly, for this method only -- the equivalent of "join fetch t.topics"
    // in JPQL (already covered for a @ManyToOne relationship in "Transaction
    // Management"), expressed as an annotation instead.
    @EntityGraph(attributePaths = "topics")
    List<CategoryGraphExample> findAll();
}

@Entity
class CategoryGraphExample {
    @Id
    private Long id;

    @OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
    private List<TopicGraphExample> topics;
}

@Entity
class TopicGraphExample {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    private CategoryGraphExample category;
}

// With this @EntityGraph in place, calling findAll() now runs ONE query
// (fetching every Category with its Topics already joined in), instead of
// the 1+N queries NPlusOneProblemExample triggered -- the loop over
// category.getTopics() no longer touches the database at all, because the
// data is already there.
```

`@EntityGraph(attributePaths = "topics")`, JPQL'deki `join fetch t.topics`'in ("Transaction Management"in bir `@ManyToOne` ilişkisi için zaten kullandığı aynı teknik, şimdi bir `@OneToMany` 'ye uygulanmış) annotation eşdeğeridir -- `findAll()`, artık önceki bölümdeki 1+N sorgu yerine, `topics`'in zaten birleştirilmiş olduğu TEK bir sorgu çalıştırır.

Batch Fetching ile Düzeltmek

Aynı sorun için farklı bir düzeltme, birçok farklı sorgunun aynı ilişkiye dokunduğu ve her birine `@EntityGraph` eklemenin tekrarlı olacağı durumlarda yararlı.

```
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.OneToMany;
import org.hibernate.annotations.BatchSize;

import java.util.List;

// A DIFFERENT fix for the same N+1 problem -- useful specifically when
// @EntityGraph isn't a good fit (for instance, when many different
// queries touch this relationship, and adding @EntityGraph to every one
// of them would be repetitive).
@Entity
class CategoryBatchFetchExample {
    @Id
    private Long id;

    // @BatchSize(size = 20) doesn't eliminate the extra queries the way
    // @EntityGraph does -- it groups them. Instead of one query PER
    // category (the 1+N shape), Hibernate fetches topics for up to 20
    // categories' worth of ids AT ONCE, in a single "WHERE category_id IN
    // (?, ?, ..., up to 20 values)" query.
    @OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
    @BatchSize(size = 20)
    private List<TopicBatchFetchExample> topics;
}

@Entity
class TopicBatchFetchExample {
    @Id
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id")
    private CategoryBatchFetchExample category;
}

// With 7 categories and batch size 20 (larger than the category count),
// NPlusOneProblemExample's 1+7 queries become just 1+1: one query for the
// categories, and ONE batched query -- "WHERE category_id IN (1,2,3,4,5,6,7)"
// -- covering every category's topics at once, instead of seven separate
// single-category queries.
```

`@BatchSize(size = 20)`, ekstra sorguları ORTADAN KALDIRMAZ -- onları GRUPLAR. Kategori başına bir sorgu yerine, Hibernate, tek bir `WHERE category_id IN (?, ?, ...)` sorgusuyla, en

fazla 20 kategorinin id'si için bir kerede konuları getirir. 7 kategori ve 20'lik bir batch boyutuyla, N+1 örneğindeki 1+7 sorgu, yalnızca 1+1 olur -- kategoriler için bir sorgu, ve her kategorinin konularını birlikte kapsayan tek bir batch sorgusu.

İlişkiyi Hiç Getirmeyerek Düzeltmek: Projection'lar

Üçüncü bir seçenek, sorunu daha verimli çözmek yerine tamamen atlatır: çağıran hiç ihtiyaç duymadıysa, ilişkiyi hiç getirme. Bu kategoride daha önceki "Pagination, Sorting, and Projections", tam olarak bunu zaten işledi -- bir interface ya da record projection, yalnızca bir sorgunun gerçekten ihtiyaç duyduğu alanları seçer, ve bu asla lazy bir ilişkiyi içermiyorsa, hiçbir N+1 problemi oluşamaz, çünkü ilişkiye baştan hiç dokunulmaz.

Düzeltmeler Arasında Seçim Yapmak

Şimdi dört araç, aynı temel sorunun örtüşen versiyonlarını çözüyor, her biri farklı bir duruma uyuyor. `join fetch` ("Transaction Management"te zaten işlendi), ilişkiye her zaman ihtiyaç duyan tek, spesifik bir sorguya uyar. `@EntityGraph`, aynı duruma daha az JPQL yazarak uyar, özellikle bir `join fetch` cümlesi eklenecek bir `@Query`'si olmayan derived bir query metodu için. Batch fetching, birçok farklı sorgunun dokunduğu bir ilişkiye uyar, her yere `@EntityGraph` eklemenin tekrarlı olacağı durumda. Bir projection, baştan ilişkinin verisine hiç ihtiyaç duymayan bir sorguya uyar -- uygulandığında en ucuz düzeltme.

Yaygın Yanlış Anlamalar

"N+1, bir şeyin bozuk olduğu anlamına gelir." İlgili her tek sorgu tamamen doğrudur -- sorun tamamen gidiş-geliş SAYISIYLA ilgilidir, doğrulukla değil. **"`@OneToMany`'nin kendi foreign-key sütununa ihtiyacı vardır."** Yoktur -- foreign key tamamen sahip olan `@ManyToOne` tarafında yaşar; `mappedBy` ile `@OneToMany`, kendi sütununu eklemez. **"`cascade` ve `orphanRemoval` aynı şeyi yapar."** Yapmazlar -- `cascade`, ebeveyn üzerinde yapılan açık bir kaydetme/silme işlemini çocuklarına yayar; `orphanRemoval`, bir çocuğu, ebeveyninin koleksiyonundan çıkarıldığı için, çocuk üzerinde hiç açık bir silme olmadan siler.

Sırada Ne Var

Bu dersteki her düzeltme, lazy loading ve N+1 problemini tek bir isteğin sınırlarının DIŞINDAN aştı. Bu kategoride sıradaki "Persistence Context ve Locking", bir seviye daha derine iniyor -- "managed," "detached," ve persistence context'in kendi first-level cache'inin bir entity'nin yaşam döngüsü için gerçekte ne anlama geldiğine, ve iki transaction aynı satıra aynı anda dokunduğunda ne olduğuna.

Best Practices

- `cascade` / `orphanRemoval`'a yalnızca ilişkili entity gerçekten ebeveyninden bağımsız var olamadığında (ya da olmaması gerektiğinde) başvur -- her `@OneToMany`'de bir varsayılan olarak değil.
- İlişki kendi verisini taşıması gerektiği anda, düz bir `@ManyToMany` yerine açık bir join entity'sini (bu projenin gerçek `QuizQuestion`'ı gibi) tercih et.
- Her zaman bir ilişkiye ihtiyaç duyan tek bir sorgu için `@EntityGraph`'a, birçok sorgunun dokunduğu bir ilişki için batch fetching'e, ve ilişkinin verisine gerçekte hiç ihtiyaç duyulmadığında bir projection'a başvur.
- `findAll()` şeklinde bir sorgudan hemen sonra bir koleksiyon üzerinde bir döngüye dikkat et -- bir N+1 probleminin gizlendiği en yaygın tek yer.

Yaygın Hatalar

- İlişkili entity'nin gerçekten bağımsız var olup olamayacağını kontrol etmeden `CascadeType.REMOVE` / `ALL` eklemek -- istenenden çok daha fazlasını silmek.
- İlişki gerçekte kendi verisine ihtiyaç duyarken, açık bir join entity'si yerine düz bir `@ManyToMany`'ye başvurmak.
- Ürettiği 1+N sorguyu fark etmeden, her iterasyonda lazy bir koleksiyona erişen bir döngü yazmak.
- Aynı ilişki başka bir yerde onsuz sorgulanmaya devam ederken, `@EntityGraph`'ı tek bir sorguda N+1'i düzeltmek için uygulayıp sorunu yalnızca kısmen çözülmüş bırakmak.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- `@OneToMany` (`mappedBy` ile), var olan bir `@ManyToOne`'un ayna görüntüsüdür -- kendi sütununu eklemez; foreign key sahip olan tarafta kalır.
- `@ManyToMany`'nin kendi join table'ına (`@JoinTable`) ihtiyacı vardır; ilişki kendi verisini taşıması gerektiğinde açık bir join entity'si genelde daha iyi bir seçimdir.
- `cascade`, ebeveynden çocuklarına açık bir kaydetme/silmeyi yayar; `orphanRemoval`, bir çocuğu, ebeveyninin koleksiyonundan ayrıldığı için siler.
- N+1 problemi, bir listeyi getirmek için 1 sorgu, artı o liste üzerindeki bir döngüde lazy bir ilişkiye erişildiğinde öğe BAŞINA bir sorgu daha demektir.
- `@EntityGraph`, batch fetching, ve projection'lar, N+1 için üç farklı düzeltmedir, her biri farklı bir duruma uygun -- "Transaction Management"te zaten işlenen `join fetch`'in yanı sıra.

Cheat Sheet

```
// @ManyToOne'un diğer tarafı
@OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
private List<Topic> topics;

// Kendi join table'ıyla @ManyToMany
@ManyToMany
@JoinTable(name = "topic_tag",
        joinColumns = @JoinColumn(name = "topic_id"),
        inverseJoinColumns = @JoinColumn(name = "tag_id"))
private Set<Tag> tags;

// Cascade + orphanRemoval
@OneToMany(mappedBy = "category", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Topic> topics;

// N+1: bir sorgu, sonra döngü iterasyonu başına bir sorgu daha
for (Category c : categoryRepository.findAll()) {
    c.getTopics().size(); // her seferinde ayrı bir sorguyu tetikler
}

// Düzeltme 1: @EntityGraph
@EntityGraph(attributePaths = "topics")
List<Category> findAll();

// Düzeltme 2: batch fetching
@BatchSize(size = 20)
private List<Topic> topics;

// Düzeltme 3: hiç getirme -- bir projection (bkz. "Pagination, Sorting, and Projections")
interface CategorySummary { String getName(); }
```

Terimler Sözlüğü

- **@OneToMany**: foreign key'i diğer (`@ManyToOne`) entity'de yaşayan bir ilişkinin, `mappedBy` ile bildirilen "çok" tarafının mapping'i.
- **@ManyToMany**: hiçbir tarafın tablosunun bir foreign key için doğal yer olmadığı, kendi join table'ına ihtiyaç duyan bir ilişki.
- **Cascade**: bir ebeveyn entity üzerinde yapılan açık bir işlemi (kaydetme, silme) ilişkili çocuklarına yaymak.
- **orphanRemoval**: bir çocuk entity'yi, `cascade`'den bağımsız olarak, ebeveyninin koleksiyonundan çıkarıldığı için silmek.
- **N+1 problemi**: bir listeyi getiren bir sorgu, artı her birindeki lazy bir ilişkiye bir döngüde erişildiğinde öğe başına bir ekstra sorgu.
- **@EntityGraph / batch fetching**: N+1 için iki farklı düzeltme -- bir sorgu için bir ilişkiyi eagerly join etmek, ile birçok öğe-başına sorguyu daha az, daha büyük sorguya gruplamak.