Relationships, Fetching, and the N+1 Problem

@OneToMany/@ManyToMany mapping (beyond the @ManyToOne already covered in full in "Transaction Management"), cascade types (PERSIST/MERGE/REMOVE/ALL) and orphanRemoval, a concrete demonstration of the N+1 problem, and three different fixes with @EntityGraph/batch fetching/projections -- using this project's own Topic/Category /QuizQuestion classes. The 6th lesson in the Spring Data JPA category.

Advanced 40 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/relationships-fetching-and-n-plus-1

Every entity covered in this category so far has had, at most, a single `@ManyToOne` pointing outward — `Topic.category`, `Category.course`. "Transaction Management" already covered what happens when that single relationship is accessed lazily, outside a transaction. This lesson goes further in three directions at once: the OTHER shapes a relationship can take, how deleting or saving one entity can cascade to another, and a performance problem — never named until now — that a perfectly correct set of relationship mappings can still quietly trigger.

What "Transaction Management" Already Covered, and What This Lesson Adds

`LazyInitializationException`, `join fetch`, and `spring.jpa.open-in-view` are already covered in full, using this project's own real `Topic` / `Category` / `Course` relationships — none of that is repeated here. What's new: `@OneToMany` and `@ManyToMany` (only `@ManyToOne` has come up so far), cascade types and `orphanRemoval`, and the N+1 problem — a specific, concrete performance trap that lazy loading (already covered) makes possible but doesn't, by itself, explain.

The Other Side of @ManyToOne: @OneToMany

This project's real `Topic` entity owns a `@ManyToOne` pointing at `Category` — but nothing on `Category` currently points back. `@OneToMany` is that missing other side.

```
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"` points at the FIELD on `TopicExample` that actually owns this relationship — `@OneToMany` is the mirror image of an existing `@ManyToOne`, not a second, independent mapping with its own column. The foreign key (`category_id`) still lives only on the `topic` table, exactly as it already does in this project's real schema; `@OneToMany` adds no column of its own anywhere.

@ManyToMany and Join Tables

Some relationships don't have a natural "owning" side at all — neither entity's table is the right place for a foreign key, because either side can relate to many of the other.

```
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` describes a separate table (`topic_tag`) explicitly, with a `topic_id` column and a `tag_id` column that live on neither `topic` nor `tag` . This project's own real `QuizQuestion` shows a related but different, often better approach 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 a `position` column. A plain `@ManyToMany` join table has no room for 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 more than that.

Cascade Types and orphanRemoval

Saving or deleting one entity doesn't automatically touch entities it's related to — `cascade` and `orphanRemoval` are what make it do so, deliberately.

```
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` makes saving a `Category` also save every `Topic` newly added to its list, in the same operation. `CascadeType.REMOVE` makes deleting a `Category` also delete every `Topic` still attached to it — a reasonable choice here, since this project's real schema already makes `category_id` `nullable = false`, meaning a `Topic` genuinely can't exist without one. `CascadeType.ALL` is shorthand for `PERSIST` / `MERGE` / `REMOVE` (and two less common cascade types) together. `orphanRemoval = true` covers a different case entirely: removing a `Topic` from the `topics` LIST — without deleting the `Category` at all — deletes that `Topic` from the database too, because it no longer belongs to anything; without it, that `Topic` would simply become a disconnected, orphaned row.

> `CascadeType.REMOVE` / `ALL` on a relationship where the "many" side genuinely has its own independent lifecycle (imagine `Category` cascading delete to `Topic`'s own `CodeExample`s, which arguably should be able to outlive a single topic in some designs) can delete far more than intended. Reach for it deliberately, based on whether the related entity can genuinely exist without its parent — not as a default on every `@OneToMany`.

The N+1 Problem, Demonstrated

This is the concrete, easy-to-miss performance trap the mapping mechanics above make possible.

```
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(...)` looks entirely ordinary — fetch every `Category`, loop over them, read each one's topics. But `category.getTopics()` is lazy, and accessing it inside the loop triggers a SEPARATE query, right then, for that one category alone. One query fetches the categories; one MORE query runs per category inside the loop — with 7 categories, that's 8 queries total for what reads like a single fetch; with 100, it's 101. Nothing here is a bug in the ordinary sense — every line is correct, individually — the problem is purely about how many round trips to the database the correct code ends up making.

Fixing It with @EntityGraph

One fix: eagerly join the relationship for this ONE query specifically, without changing its fetch type everywhere else it's used.

```
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")` is the annotation equivalent of `join fetch t.topics` in JPQL (the same technique "Transaction Management" already used for a `@ManyToOne` relationship, now applied to a `@OneToMany`) — `findAll()` now runs ONE query, with `topics` already joined in, instead of the 1+N queries from the previous section.

Fixing It with Batch Fetching

A different fix, useful when many different queries touch the same relationship, and adding `@EntityGraph` to every one of them would be repetitive.

```
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)` doesn't eliminate the extra queries — it GROUPS them. Instead of one query per category, Hibernate fetches topics for up to 20 categories' worth of ids at

once, with a single `WHERE category_id IN (?, ?, ...)` query. With 7 categories and a batch size of 20, the 1+7 queries from the N+1 example become just 1+1 — one query for the categories, one batched query covering every category's topics together.

Fixing It by Not Fetching the Relationship at All: Projections

A third option sidesteps the problem entirely, rather than solving it more efficiently: don't fetch the relationship at all, if the caller never actually needed it. "Pagination, Sorting, and Projections," earlier in this category, already covered exactly this — an interface or record projection selects only the fields a query genuinely needs, and if that never includes a lazy relationship, no N+1 problem can occur, because the relationship is never touched in the first place.

Choosing Between the Fixes

Four tools now solve overlapping versions of the same underlying problem, each fitting a different situation. `join fetch` (already covered in "Transaction Management") suits a single, specific query that always needs the relationship. `@EntityGraph` suits the same situation with less JPQL to write, especially for a derived query method that has no `@Query` to add a `join fetch` clause to. Batch fetching suits a relationship touched by many different queries, where adding `@EntityGraph` everywhere would be repetitive. A projection suits a query that never needed the relationship's data to begin with — the cheapest fix, when it applies.

Common Misconceptions

"N+1 means something is broken." Every individual query involved is completely correct — the problem is purely about the NUMBER of round trips, not correctness. **"`@OneToMany` needs its own foreign-key column."** It doesn't — the foreign key lives entirely on the owning `@ManyToOne` side; `@OneToMany` with `mappedBy` adds no column of its own. **"`cascade` and `orphanRemoval` do the same thing."** They don't — `cascade` propagates an explicit save/delete operation on the parent to its children; `orphanRemoval` deletes a child specifically because it was removed from its parent's collection, with no explicit delete on the child at all.

What Comes Next

Every fix in this lesson worked around lazy loading and the N+1 problem from OUTSIDE a single request's boundaries. "The Persistence Context and Locking," next in this category, goes a level deeper — into what "managed," "detached," and the persistence context's own first-level cache actually mean for an entity's lifecycle, and what happens when two transactions touch the same row at once.

Best Practices

- Reach for `cascade` / `orphanRemoval` only when the related entity genuinely can't (or shouldn't) exist independently of its parent — not as a default on every `@OneToMany`.
- Prefer an explicit join entity (like this project's real `QuizQuestion`) over a plain `@ManyToMany` the moment the relationship needs to carry any data of its own.
- Reach for `@EntityGraph` for a single query that always needs a relationship, batch fetching for a relationship touched by many queries, and a projection when the relationship's data was never actually needed.
- Watch for a loop over a collection immediately after a `findAll()`-shaped query — it's the single most common place an N+1 problem hides.

Common Mistakes

- Adding `CascadeType.REMOVE` / `ALL` without checking whether the related entity can genuinely exist independently — deleting far more than intended.
- Reaching for a plain `@ManyToMany` when the relationship actually needs its own data, instead of an explicit join entity.
- Writing a loop that accesses a lazy collection per iteration, without noticing the 1+N queries it generates.
- Applying `@EntityGraph` to fix N+1 in one query while the same relationship is still queried elsewhere without it, leaving the problem only partially solved.

Summary, Cheat Sheet, and Glossary

Summary

- `@OneToMany` (with `mappedBy`) is the mirror image of an existing `@ManyToOne` — it adds no column of its own; the foreign key stays on the owning side.
- `@ManyToMany` needs its own join table (`@JoinTable`); an explicit join entity is often the better choice once the relationship needs to carry its own data.
- `cascade` propagates an explicit save/delete from a parent to its children; `orphanRemoval` deletes a child specifically because it left its parent's collection.
- The N+1 problem is 1 query to fetch a list, plus one more query PER item when a lazy relationship is accessed in a loop over that list.
- `@EntityGraph`, batch fetching, and projections are three different fixes for N+1, each suited to a different situation — alongside `join fetch`, already covered in "Transaction Management."

Cheat Sheet

```
// The other side of @ManyToOne
@OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
private List<Topic> topics;

// @ManyToMany with its own join table
@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: one query, then one more per loop iteration
for (Category c : categoryRepository.findAll()) {
    c.getTopics().size(); // triggers a separate query, each time
}

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

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

// Fix 3: don't fetch it at all -- a projection (see "Pagination, Sorting, and Projections")
interface CategorySummary { String getName(); }
```

Glossary

- **@OneToMany**: the "many" side's mapping of a relationship whose foreign key lives on the other (`@ManyToOne`) entity, declared with `mappedBy`.
- **@ManyToMany**: a relationship needing its own join table, since neither side's table is the natural place for a foreign key.
- **Cascade**: propagating an explicit operation (save, delete) performed on a parent entity to its related children.
- **orphanRemoval**: deleting a child entity specifically because it was removed from its parent's collection, independent of `cascade`.
- **N+1 problem**: one query fetching a list, plus one additional query per item when a lazy relationship on each is accessed in a loop.
- **@EntityGraph / batch fetching**: two different fixes for N+1 — eagerly joining a relationship for one query, versus grouping many per-item queries into fewer, larger ones.