Query Methods and JPQL with @Query

How Spring Data JPA derives a query from a repository method's name (findBy/existsBy /countBy, And/Or/OrderBy), writing JPQL with @Query and named parameters, @Modifying for bulk updates/deletes, join fetch, and when to reach for a native query -- using this project's own real repositories. The 3rd lesson in the Spring Data JPA category.

Intermediate 30 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/query-methods-and-jpql

---

"Entities and the Repository Abstraction" showed `TopicRepository.findBySlug(String slug)` working — no query, no SQL, no JPQL written anywhere — and deferred exactly how that's possible to this lesson. This is where that gets answered: two different ways to tell a repository what to fetch, without ever hand-writing a `SELECT` yourself unless you genuinely need to.

Two Ways to Ask a Repository for Data

A repository method can get its query from one of two places: Spring Data JPA can DERIVE one automatically from the method's own name, or you can write one explicitly with `@Query` . Every repository method in this project uses one or the other — there's no third option, and no method is ever left to guess.

Derived Query Methods: Reading a Query From a Method's Name

Spring Data JPA parses a method's name at application startup, matches its pieces against the entity's own properties, and builds a query from that — before your application ever handles a real request.

```
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)` is read piece by piece: `findBy` starts a condition, `TopicId` becomes `WHERE topic_id = ?`, and `OrderBySortOrderAsc` becomes `ORDER BY sort_order ASC` — notably, `TopicId` resolves through the entity's `topic` relationship down to its `id`, not just a direct field. `findByTopicIdAndExampleName(...)` shows two conditions in one method: parameters are matched to conditions IN ORDER, so the first parameter binds to the first condition, the second to the second.

Combining and Ordering Conditions

The same naming rules scale to methods with several conditions and keywords chained together — this project's real `QuizRepository` has the densest example of that.

```
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(...)` breaks down into five pieces: `findFirst` limits the result to one row instead of a list; `TopicId` and `Language` (joined by `And`) each consume one method parameter; `ActiveTrue` adds a THIRD condition — `WHERE active = true` — but consumes NO parameter at all, since `True` / `False` supply their own literal value for a boolean property; `OrderByIdAsc` sorts the result. Three conditions, only two parameters — worth noticing precisely because it's easy to miscount at a glance.

Other Derived Prefixes: findFirstBy, existsBy, and countBy

`findBy` isn't the only prefix Spring Data JPA understands — `existsBy` and `countBy` follow the identical parsing rules, but return a fundamentally different, more efficient shape of answer. `existsByTopicIdAndLanguage(...)` returns a plain `boolean` from a single `SELECT EXISTS(...)` query — checking whether something is present without loading a whole entity just to find out. `countByTopicId(...)` returns a `long` from a single `SELECT COUNT(*)` query — counting rows without fetching any of them at all. Reach for these instead of `findBy...().isPresent()` or `findBy...().size()` whenever you only need the boolean or the number, not the actual data.

When a Derived Name Isn't Enough: @Query and JPQL

A derived name works well for straightforward conditions on one entity — it stops being the right tool once a query needs to reach across relationships or express something a method name simply can't spell out cleanly.

```
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` switches from a generated query to one written directly in JPQL — Jakarta Persistence Query Language. JPQL looks like SQL, but it queries ENTITIES and their fields (

`Quiz`, `q.topic`, `t.slug`), not tables and columns directly; Hibernate translates it into real SQL underneath, the exact same translation step "JPA, Hibernate, and Spring Data JPA" introduced. The method's own parameter names (`topicSlug`, `language`, `quizSlug`) bind directly to the query's `:topicSlug` / `:language` / `:quizSlug` placeholders — Spring Data JPA matches them by name, with no separate annotation required in this case.

> Named-parameter binding by matching a method parameter's name only works when the project is compiled with parameter names retained (the `-parameters` compiler flag, which Spring Boot projects enable by default). `@Param("name")` makes that binding explicit regardless, and is worth adding whenever you want to be certain, or when a parameter's Java name and the query's placeholder name need to differ.

Joining Related Entities with join fetch

A query that touches a relationship risks the exact problem "Transaction Management" already covers in depth — a `LazyInitializationException` if that relationship gets accessed outside a transaction. `join fetch` is one way to sidestep it, by pulling the related data back in the SAME query.

```
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()` uses one `join fetch` to bring back each `TopicTranslation`'s `Topic` in a single query, instead of triggering a separate query per row later — exactly the technique "Transaction Management" already showed for this project's own sitemap generation. `findByQuizIdOrderByPositionAsc(...)` chains two `join fetch` clauses, pulling back a `QuizQuestion`, its `Question`, and that `Question`'s own `Topic`, all at once. Chaining joins like this specifically to avoid running one query per relationship, per row, is the exact shape of problem "Relationships, Fetching, and the N+1 Problem," later in this category, covers in full — this lesson only needed the JPQL syntax itself.

Modifying Data: @Modifying

Every query so far has been a read. Changing many rows at once — without loading each one into Java, changing a field, and saving it back individually — needs one more annotation.

```
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 {
}
```

`@Query` here holds an `UPDATE` statement, not a `SELECT` — `@Modifying` is REQUIRED to tell Spring Data JPA this isn't an ordinary read and should run as a bulk update instead; without it, Spring Data JPA would try to map the result onto entities and fail. `@Transactional` is required too: a modifying query runs directly against the database, bypassing the persistence context's usual change tracking entirely, and needs an active transaction the same way any other write does — exactly as "Transaction Management" already covers.

A Note on Native Queries

`@Query` doesn't have to contain JPQL at all — `nativeQuery = true` switches to real SQL, queried against the actual table and its actual columns rather than the entity model.

```
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 {
}
```

This project's own `findRandomPublishedPool(...)` uses a native query specifically because JPQL has no portable `RANDOM()` function, and this particular query needs database-level random ordering for Practice mode's question selection. The trade-off is real: a native query ties the code to the actual 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 it couldn't here.

Common Misconceptions

"Derived query method names are just a convention I have to follow, with no real mechanism behind them." They're parsed and compiled into a real query at startup — an invalid or unparseable method name fails the application immediately, not silently. **"`@Query` always means writing SQL."** By default it means JPQL, which queries entities and their fields, not tables and columns — `nativeQuery = true` is what switches to real SQL, and it's

the exception, not the rule. **"A modifying query works like any other repository method."** It doesn't — without `@Modifying`, Spring Data JPA doesn't know to treat it as a bulk update/delete rather than a read.

What Comes Next

Every query in this lesson returned either a whole entity, a list of them, a `boolean`, or a `long` — nothing about shaping, paging, or sorting a large result set for a client was covered. "Pagination, Sorting, and Projections," next in this category, picks up exactly there: returning `Page<T>` and `Sort`-aware results at the repository level (the half of the picture "REST API Design" never taught), and returning something narrower than a whole entity when a query doesn't need one.

Best Practices

- Prefer a derived query method for straightforward, single-entity conditions — reach for `@Query` only once a derived name would be unwieldy or can't express the query at all.
- Use `existsBy...` / `countBy...` instead of `findBy...().isPresent()` / `.size()` whenever a boolean or a count is genuinely all that's needed.
- Always pair `@Modifying` with `@Transactional` — a modifying query needs an active transaction exactly like any other write.
- Reach for a native query only when JPQL genuinely can't express something (as with `RANDOM()`) — it trades entity-model independence for that capability.

Common Mistakes

- Writing a derived method name the entity's properties don't actually support, and being surprised by a startup failure rather than a runtime one.
- Miscounting parameters against conditions in a derived method name — a boolean condition like `ActiveTrue` consumes no parameter at all.
- Forgetting `@Modifying` on an `UPDATE` / `DELETE` `@Query`, or forgetting `@Transactional` alongside it.
- Reaching for a native query by default instead of JPQL, losing the entity-model independence JPQL provides for no real reason.

Summary, Cheat Sheet, and Glossary

Summary

- A repository method's query comes from one of two places: derived automatically from its name, or written explicitly with `@Query`.

- A derived name is parsed piece by piece — `findBy` / `existsBy` / `countBy`, conditions joined with `And` / `Or`, boolean literals like `ActiveTrue`, and `OrderBy` — and validated at application startup.
- `@Query` switches to JPQL by default — querying entities and their fields, translated to SQL by Hibernate underneath — or to real SQL with `nativeQuery = true`.
- `join fetch` in JPQL pulls a relationship back in the same query, avoiding a later `LazyInitializationException` and (at larger scale) the N+1 problem.
- `@Modifying` (paired with `@Transactional`) is required for an `@Query` that updates or deletes rows in bulk, rather than reading them.

Cheat Sheet

```
// Derived query methods
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);

// @Query with JPQL, named parameters bound by method parameter name
@Query("select q from Quiz q join fetch q.topic t where t.slug = :topicSlug")
Optional<Quiz> findByTopicSlug(String topicSlug);

// Modifying query
@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);
```

Glossary

- **Derived query method**: a repository method whose query Spring Data JPA builds automatically by parsing its name.
- **JPQL (Jakarta Persistence Query Language)**: a query language resembling SQL but targeting entities and their fields rather than tables and columns.
- **@Query**: an annotation supplying an explicit JPQL (or, with `nativeQuery = true`, native SQL) query for a repository method.
- **join fetch**: a JPQL clause that retrieves a related entity in the same query, avoiding a separate query for that relationship later.
- **@Modifying**: an annotation required on an `@Query` that performs a bulk `UPDATE` / `DELETE` rather than a read.

- **Native query**: a `@Query` written in real SQL against the actual schema, rather than in JPQL against the entity model.