Dynamic Queries with Specifications

Delivers on the optional filtering "REST API Design" left in memory, pushing it into the database -- the Specification interface as a thin wrapper around JPA's own Criteria API (Root/CriteriaQuery/CriteriaBuilder/Predicate), JpaSpecificationExecutor, combining conditions with Specification.where/and/or, and combining with real pagination via findAll(Specification, Pageable) -- using this project's own Topic entity. The 5th lesson in the Spring Data JPA category.

Advanced 35 min

Last updated: August 24, 2026
Read online: https://www.learnforgex.com/en/topics/dynamic-queries-with-specifications

---

"REST API Design"'s filtering example built optional conditions in Java — a `Predicate<Topic>` per filter, each defaulting to "match everything" when its query parameter was absent, combined with `.and(...)` over an in-memory `Stream`. Its own comment was explicit about the limitation: "a real repository would push this down into a WHERE clause... or a JPA `Specification` for cases this dynamic." This lesson is that real repository-level implementation.

What "REST API Design" Left Unfinished: Pushing Filtering Into the Database

The in-memory version worked, but only because its example list had three `Topic`s in it. Filtering that way against a real table means fetching every row first, then discarding most of them in Java — the database never gets to use an index, and every unfiltered row still has to cross the network. The idea itself — an optional condition that does nothing when absent, combined with others — was already right; what's missing is running that same idea as a real, generated `WHERE` clause instead of a `Stream.filter(...)`.

Why Query Methods and @Query Aren't Enough Here

"Query Methods and JPQL with @Query" covered two ways to get a query: a name Spring Data JPA parses, or JPQL you write directly. Both are FIXED at compile time — a method's name declares its conditions once, and a `@Query` string is the same text every time it runs. Neither can express "filter by category, but only if a category was actually supplied, and by difficulty, but only if that was too" — the SET of active conditions isn't known until a request actually arrives. That's a genuinely different problem, and it needs a genuinely different tool.

What Is a Specification?

A `Specification<T>` is a small functional interface representing one WHERE condition, expressed as Java code that builds it, rather than as a fixed string of JPQL or a method name.

```
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(...)` returns a `Specification<Topic>` — a lambda that, given a `Root<Topic>`, a `CriteriaQuery`, and a `CriteriaBuilder`, produces a `Predicate`. Nothing runs yet at this point; a `Specification` only describes how to BUILD a condition. Something still has to hand it to a repository before it does anything.

The Criteria API Underneath a Specification

`Root`, `CriteriaQuery`, `CriteriaBuilder`, and `Predicate` come from JPA's own Criteria API — the specification (in the JPA-the-spec sense covered in "JPA, Hibernate, and Spring Data JPA") for building queries out of Java objects instead of query text. `Root<Topic>` 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` (`cb.equal(...)`, and many similar methods for other comparisons) from a path and a value. Spring Data JPA's

`Specification` is a thin, convenient wrapper around this API — it doesn't replace it, the same way "JPA, Hibernate, and Spring Data JPA" covered Spring Data JPA not replacing JPA or Hibernate more generally.

JpaSpecificationExecutor: Letting a Repository Accept Specifications

A `Specification` describes a condition, but a repository needs to explicitly opt into accepting one.

```
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");
    }
}
```

Extending `JpaSpecificationExecutor<Topic>` alongside `JpaRepository<Topic, Long>` is what actually gives a repository `findAll(Specification)`, `findOne(Specification)`, `count(Specification)`, and more — inherited for free, the exact same way "Entities and the Repository Abstraction" covered `CrudRepository` contributing `save` / `findById` / `findAll`. Without `JpaSpecificationExecutor`, a `Specification` has nothing to actually run against.

Combining Specifications: where, and, or

The real value of a `Specification` shows up once several of them combine into one larger condition.

```
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(...)` starts a chain; `.and(...)` / `.or(...)` combine two `Specification`s into a single, larger one — exactly the shape `DynamicFilterExample` in "REST API Design" already used with `Predicate.and(...)`, except this generates a real SQL `WHERE` clause instead of filtering an in-memory `Stream`.

Optional Filters, Pushed Into the Database

This is the piece that actually delivers on "REST API Design"'s deferred promise — the same optional-filter shape, now generating real SQL.

```
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)` is a genuinely useful starting point — it behaves as a no-op, "match everything" `Specification`, exactly the role `DynamicFilterExample`'s absent filters played by defaulting to `t -> true`. Each `if (category != null)` / `if (difficulty != null)`

check adds one more `.and(...)` only when that filter was actually supplied — with none supplied, the generated query filters nothing at all; with both supplied, it filters on both.

Specifications Together with Pageable

Dynamic filtering and real pagination aren't separate mechanisms — they combine into a single repository call.

```
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`'s `findAll` also accepts a `Pageable`, the exact same type "Pagination, Sorting, and Projections" already covered. `repository.findAll(spec, pageable)` generates a filtered, paged query PLUS a filtered count query — the identical two-query shape from that lesson, now with a dynamic `WHERE` clause instead of a fixed one.

Common Misconceptions

"A `Specification` is a query." It's a description of one condition — nothing runs until it's handed to a repository that extends `JpaSpecificationExecutor`. **"`Specification` is a completely separate mechanism from JPA."** It's a thin wrapper around JPA's own Criteria API (`Root` / `CriteriaQuery` / `CriteriaBuilder` / `Predicate`) — the same relationship Spring Data JPA has to JPA everywhere else. **"Dynamic filtering always needs Specifications."** It doesn't — a query with a small, fixed set of optional conditions can sometimes be expressed with a single JPQL `@Query` using the `:param IS NULL OR ...` pattern (used in this project's own `QuestionRepository`); `Specification` earns its place once the number or shape of conditions genuinely varies per request.

What Comes Next

Every query covered in this category so far — derived methods, JPQL, projections, `Specification`s — has been a straightforward read within a single request. "Relationships, Fetching, and the N+1 Problem," next in this category, moves from WHAT a query returns to HOW an entity's own relationships get loaded — including a performance problem (N+1) that a perfectly correct query can still trigger.

Best Practices

- Reach for a `Specification` once the SET of active filter conditions genuinely isn't known until a request arrives — not as a default replacement for query methods or `@Query`.
- Start an optional-filter chain with `Specification.where(null)`, and add one `.and(...)` per filter only when that filter's value is actually present.
- Keep individual `Specification`s small and named for what they check (`hasCategory`, `hasDifficulty`) — combine them with `.and(...)` / `.or(...)` rather than writing one large, monolithic `Specification`.
- Remember `findAll(Specification, Pageable)` exists — dynamic filtering and pagination combine into one call, not two separate steps.

Common Mistakes

- Reaching for `Specification` for a query with a small, truly fixed set of conditions, when a derived method or a plain `@Query` would say the same thing more directly.
- Forgetting `JpaSpecificationExecutor` entirely, and being surprised a repository has no `findAll(Specification)` to call.

- Building a fresh `Specification` chain without starting from `Specification.where(null)`, and having to special-case the "no filters at all" scenario separately.
- Treating a `Specification` as something that runs on its own, rather than something that still needs a `Root` / `CriteriaBuilder` (supplied by the repository at query time) to actually produce a `Predicate`.

Summary, Cheat Sheet, and Glossary

Summary

- A `Specification<T>` is Java code describing one `WHERE` condition, built from JPA's own Criteria API (`Root`, `CriteriaQuery`, `CriteriaBuilder`, `Predicate`).
- A repository must extend `JpaSpecificationExecutor<T>` (alongside `JpaRepository<T, ID>`) to accept a `Specification` at all.
- `Specification.where(...).and(...)/.or(...)` combines multiple conditions into one, the same shape as combining `Predicate`s in memory, now generating real SQL.
- `Specification.where(null)` is a genuinely useful no-op starting point for building up optional filters one `.and(...)` at a time.
- `findAll(Specification, Pageable)` combines dynamic filtering with real pagination in a single repository call.

Cheat Sheet

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

// A repository that accepts Specifications
interface TopicRepository extends JpaRepository<Topic, Long>,
JpaSpecificationExecutor<Topic> {}

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

// Optional filters, pushed into the database
Specification<Topic> spec = Specification.where(null);
if (category != null)   spec = spec.and(hasCategory(category));
if (difficulty != null) spec = spec.and(hasDifficulty(difficulty));

// Dynamic filtering + real pagination together
Page<Topic> page = repository.findAll(spec, PageRequest.of(0, 10));
```

Glossary

- **Specification<T>**: a functional interface representing one query condition, built with JPA's Criteria API rather than a fixed query string.

- **Criteria API**: JPA's own API (`Root`, `CriteriaQuery`, `CriteriaBuilder`, `Predicate`) for building queries out of Java objects instead of query text.
- **JpaSpecificationExecutor**: the interface a repository must extend, alongside `JpaRepository`, to accept `Specification`s.
- **Specification.where(null)**: a no-op, "match everything" starting `Specification`, useful as the base of an optional-filter chain.