

```

@Entity
@Table(name = "orders")
@Getter
@Setter
@Builder
@NoArgsConstructor(access = AccessLevel.PUBLIC)
@AllArgsConstructor(access = AccessLevel.PRIVATE)
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String customer;

    @OneToMany(mappedBy = "order", cascade = CascadeType.ALL,
orphanRemoval = true)
    private Set<OrderItem> items = new HashSet<>();
}

```

```

@Entity
@Getter
@Setter
@Builder
@NoArgsConstructor(access = AccessLevel.PUBLIC)
@AllArgsConstructor(access = AccessLevel.PRIVATE)
public class OrderItem {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String product;

    @ManyToOne(fetch = FetchType.LAZY)
    private Order order;

    @OneToMany(mappedBy = "orderItem", cascade =
CascadeType.ALL, orphanRemoval = true)

```

```

        private Set<ItemTag> tags = new HashSet<>();
    }
    @Entity
    @Getter
    @Setter
    @Builder
    @NoArgsConstructor(access = AccessLevel.PUBLIC)
    @AllArgsConstructor(access = AccessLevel.PRIVATE)
    public class ItemTag {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private String name;

        @ManyToOne(fetch = FetchType.LAZY)
        private OrderItem orderItem;

        @OneToMany(mappedBy = "itemTag", cascade =
CascadeType.ALL, orphanRemoval = true)
        private Set<TagAttribute> attributes = new HashSet<>();
    }

    @Entity
    @Getter
    @Setter
    @Builder
    @NoArgsConstructor(access = AccessLevel.PUBLIC)
    @AllArgsConstructor(access = AccessLevel.PRIVATE)
    public class TagAttribute {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        private String name;

        @ManyToOne(fetch = FetchType.LAZY)
        private ItemTag itemTag;
    }

```

I have these JPA Entities. Using spring 4 and java25 how can I organize springboot Specifications so that each specification lays in its own class but I should search Order entities according to deep nested attributes and same joins may not occur because similar specs are called more than once

What you want is **composable, reusable Specifications** that can:

1. Live in **separate classes**
2. Query **deeply nested associations**
3. **Avoid duplicate joins** when multiple specs are combined

This is a classic Spring Data JPA `Specification` problem, and it *is solvable cleanly* with a small infrastructure pattern.

## Key ideas (important)

1. Always reuse joins via `Root#getJoins()`
2. Encapsulate join-creation logic
3. Specifications should only express predicates, not structure
4. Use LEFT joins unless filtering requires INNER

## 1. Create a reusable Join Helper (critical)

This helper ensures **the same join is reused** instead of recreated.

java

```
public final class JoinUtils {

    private JoinUtils() {}

    @SuppressWarnings("unchecked")
    public static <R, J> Join<R, J> getOrCreateJoin(
        From<?, R> root,
        String attribute,
        JoinType joinType
    ) {
        return root.getJoins().stream()
            .filter(j -> j.getAttribute().getName().equals(attribute))
            .map(j -> (Join<R, J>) j)
    }
}
```

```
}  
}
```

[Copy code](#)

This is the **most important piece**.

## 2. Specification per concern (one class per spec)

### OrderSpecifications

java

[Copy code](#)

```
public final class OrderSpecifications {  
  
    private OrderSpecifications() {}  
  
    public static Specification<Order> customerEquals(String customer) {  
        return (root, query, cb) ->  
            cb.equal(root.get("customer"), customer);  
    }  
}
```

## 3. Nested specs that reuse joins

### OrderItemSpecifications


java

[Copy code](#)

```
public final class OrderItemSpecifications {  
  
    private OrderItemSpecifications() {}  
  
    public static Specification<Order> hasProduct(String product) {  
        return (root, query, cb) -> {  
            Join<Order, OrderItem> items =  
                JoinUtils.getOrCreateJoin(root, "items", JoinType.LEFT);  
  
            return cb.equal(items.get("product"), product);  
        };  
    }  
}
```

### ItemTagSpecifications

java

 Copy code

```
public final class ItemTagSpecifications {

    private ItemTagSpecifications() {}


    public static Specification<Order> hasTagName(String tagName) {
        return (root, query, cb) -> {
            Join<Order, OrderItem> items =
                JoinUtils.getOrCreateJoin(root, "items", JoinType.LEFT);

            Join<OrderItem, ItemTag> tags =
                JoinUtils.getOrCreateJoin(items, "tags", JoinType.LEFT);

            return cb.equal(tags.get("name"), tagName);
        };
    }
}
```

## TagAttributeSpecifications (deep nesting)

java

 Copy code

```
public final class TagAttributeSpecifications {

    private TagAttributeSpecifications() {}

    public static Specification<Order> hasTagAttribute(String attributeName) {
        return (root, query, cb) -> {
            Join<Order, OrderItem> items =
                JoinUtils.getOrCreateJoin(root, "items", JoinType.LEFT);

            Join<OrderItem, ItemTag> tags =
                JoinUtils.getOrCreateJoin(items, "tags", JoinType.LEFT);

            Join<ItemTag, TagAttribute> attributes =
                JoinUtils.getOrCreateJoin(tags, "attributes", JoinType.LEFT);

            return cb.equal(attributes.get("name"), attributeName);
        };
    }
}
```

```
}  
}
```

## 4. Combine specs safely (no duplicate joins)

java Copy code

```
Specification<Order> spec =  
    Specification.where(OrderSpecifications.customerEquals("John"))  
        .and(OrderItemSpecifications.hasProduct("Laptop"))  
        .and(ItemTagSpecifications.hasTagName("PROMO"))  
        .and(TagAttributeSpecifications.hasTagAttribute("COLOR"));
```

- ✓ Joins are reused
- ✓ No Cartesian explosion
- ✓ Each spec is independent
- ✓ Clean separation of responsibility

## 5. Important JPA details (don't skip)

### Avoid duplicate results

When joining collections, **always add**:

java Copy code

```
query.distinct(true);
```

You can do it in a base spec:

java Copy code

```
public static Specification<Order> distinct() {  
    return (root, query, cb) -> {  
        query.distinct(true);  
        return cb.conjunction();  
    };  
}
```

Then:

java Copy code

```
Specification<Order> spec =  
    Specification.where(OrderSpecifications.distinct())  
        .and(...);
```

## 6. Why this works

Problem	Solution
Duplicate joins	<code>getOrCreateJoin()</code>
Deep nesting	Layered specs
Reusability	One spec = one concern
Spring 4 compatible	Pure JPA Criteria
Java 25 ready	No deprecated APIs

## 7. Optional improvement (advanced)

If you want **even cleaner specs**, you can introduce a `JoinContext` object stored in `CriteriaQuery#getHints()` — but the above approach is **already production-grade** and widely used.

### Summary

- ✓ Put **each Specification** in its own class
- ✓ Use a **shared Join utility**
- ✓ Reuse joins to avoid duplication
- ✓ Combine specs freely
- ✓ Works with deep nested relationships

If you want, I can also:

- Refactor this to **metamodel-based (static metamodel)**
- Show a **generic recursive join builder**
- Add **dynamic filter building from REST parameters**

Just tell me.



Give me the optional improvement you suggested

Below is the **optional / advanced improvement** using a **JoinContext** stored in the **CriteriaQuery** so that:

- Joins are created **once per query**
- Specs do **not depend on Root structure**
- You can reuse joins **across all specifications**, even deeply nested
- This scales very well as your model grows

This is a **professional-grade pattern** used in large Spring Data JPA codebases.

## Concept

We store a **JoinContext** inside the **CriteriaQuery** using query hints.

Each **Specification** :

- Retrieves the context
- Asks for a join path
- Gets the same join instance every time

No duplicate joins. No tight coupling.

## 1. JoinContext

```
java                                                                    Copy code

public class JoinContext {

    private final Map<String, From<?, ?>> joins = new HashMap<>();

    @SuppressWarnings("unchecked")
    public <F, T> Join<F, T> getOrCreate(
        From<?, F> from,
        String attribute,
        JoinType joinType
    ) {
        String key = from.getJavaType().getName() + "." + attribute;

        return (Join<F, T>) joins.computeIfAbsent(
            key,
            k -> from.join(attribute, joinType)
        );
    }
}
```



```
}  
}
```

## 2. Query hint helper

```
java 📄 Copy code  
  
public final class JoinContextHolder {  
  
    private static final String JOIN_CONTEXT_KEY = "JOIN_CONTEXT";  
  
    private JoinContextHolder() {}  
  
    public static JoinContext get(CriteriaQuery<?> query) {  
        return (JoinContext) query.getHints()  
            .computeIfAbsent(JOIN_CONTEXT_KEY, k -> new JoinContext());  
    }  
}
```

## 3. Base specification (initializes the context)

You must add this spec once.

```
java 📄 Copy code  
  
public final class BaseSpecifications {  
  
    private BaseSpecifications() {}  
  
    public static <T> Specification<T> withJoinContext() {  
        return (root, query, cb) -> {  
            JoinContextHolder.get(query);  
            query.distinct(true);  
            return cb.conjunction();  
        };  
    }  
}
```

## 4. Deep specs using JoinContext

### OrderItem spec

```
java 📄 Copy code
```

```

public final class OrderItemSpecifications {

    private OrderItemSpecifications() {}

    public static Specification<Order> hasProduct(String product) {
        return (root, query, cb) -> {
            JoinContext ctx = JoinContextHolder.get(query);

            Join<Order, OrderItem> items =
                ctx.getOrCreate(root, "items", JoinType.LEFT);

            return cb.equal(items.get("product"), product);
        };
    }
}

```

## ItemTag spec

java Copy code

```

public final class ItemTagSpecifications {

    private ItemTagSpecifications() {}

    public static Specification<Order> hasTag(String tagName) {
        return (root, query, cb) -> {
            JoinContext ctx = JoinContextHolder.get(query);

            Join<Order, OrderItem> items =
                ctx.getOrCreate(root, "items", JoinType.LEFT);


            Join<OrderItem, ItemTag> tags =
                ctx.getOrCreate(items, "tags", JoinType.LEFT);

            return cb.equal(tags.get("name"), tagName);
        };
    }
}

```

## TagAttribute spec (very deep)

java

 Copy code

```
public final class TagAttributeSpecifications {

    private TagAttributeSpecifications() {}

    public static Specification<Order> hasAttribute(String attributeName) {
        return (root, query, cb) -> {
            JoinContext ctx = JoinContextHolder.get(query);

            Join<Order, OrderItem> items =
                ctx.getOrCreate(root, "items", JoinType.LEFT);

            Join<OrderItem, ItemTag> tags =
                ctx.getOrCreate(items, "tags", JoinType.LEFT);

            Join<ItemTag, TagAttribute> attributes =
                ctx.getOrCreate(tags, "attributes", JoinType.LEFT);

            return cb.equal(attributes.get("name"), attributeName);
        };
    }
}
```

## 5. Usage

java

 Copy code

```
Specification<Order> spec =
    Specification.where(BaseSpecifications.withJoinContext())
        .and(OrderSpecifications.customerEquals("John"))
        .and(OrderItemSpecifications.hasProduct("Laptop"))
        .and(ItemTagSpecifications.hasTag("PROMO"))
        .and(TagAttributeSpecifications.hasAttribute("COLOR"));

List<Order> orders = orderRepository.findAll(spec);
```

## 6. Why this is better than the simple approach

Aspect	Simple JoinUtils	JoinContext
Join reuse	Root-level only	Whole query

Aspect	Simple JoinUtils	JoinContext
Nested joins	Manual	Automatic
Spec independence	Medium	High
Large query safety	OK	Excellent
Maintenance	Simple	Slightly more code

## 7. Important notes

### ⚠️ Key uniqueness

We use:

```
java
from.getJavaType().getName() + "." + attribute
```

📄 Copy code

This avoids collisions when the same attribute name appears on different entities.

### ⚠️ INNER vs LEFT

If **any** spec requires INNER joins:

- You must standardize join type OR
- Promote LEFT → INNER manually

For safety, many teams **default to LEFT**.

## 8. When to use this pattern

Use this **JoinContext** pattern when:

- You have **dynamic search APIs**
- Specs are composed arbitrarily
- Entities are deeply nested
- Duplicate joins have already bitten you 😊

## Final takeaway

This turns your Specifications into:

- Stateless
- Composable
- Order-independent
- Join-safe

If you want, I can also:

- Convert this to **static metamodel** ( `Order_` )
- Add **path-expression specs**
- Show **dynamic REST filter** → **Specification mapping**

Just say the word.



Add new attributes to the entities and make this composable set up like this? Sometimes i need to search by all attributes but sometimes i need to search by per one attribute of each entities. How can i achieve this without compromising composable structure

This is a **very good question** and it goes to the heart of *why* you build composable Specifications in the first place.

Below is a **clean, scalable pattern** that lets you:

- ✓ Add **new attributes** freely
- ✓ Search by **any single attribute**
- ✓ Search by **all attributes together**
- ✓ Keep **each spec composable & reusable**
- ✓ Avoid "god specifications"
- ✓ Avoid duplicate joins

This builds directly on the **JoinContext pattern** you already have.

## 1 Extend entities (example)

Let's add a few realistic attributes.

java

 Copy code

```
@Entity
@Table(name = "orders")
public class Order {
    @Id
```

```


@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;

private String customer;
private LocalDate orderDate;
private OrderStatus status;

@OneToMany(mappedBy = "order", cascade = CascadeType.ALL, orphanRemoval = true)
private Set<OrderItem> items = new HashSet<>();
}

```

java

 Copy code

```

@Entity
public class OrderItem {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;


    private String product;
    private BigDecimal price;
    private Integer quantity;

    @ManyToOne(fetch = FetchType.LAZY)
    private Order order;

    @OneToMany(mappedBy = "orderItem", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<ItemTag> tags = new HashSet<>();
}

```

java

 Copy code

```

@Entity
public class ItemTag {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String category;
}

```

```

@ManyToOne(fetch = FetchType.LAZY)
private OrderItem orderItem;

@OneToMany(mappedBy = "itemTag", cascade = CascadeType.ALL, orphanRemoval = true)
private Set<TagAttribute> attributes = new HashSet<>();
}

```

java Copy code

```

@Entity
public class TagAttribute {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String value;

    @ManyToOne(fetch = FetchType.LAZY)
    private ItemTag itemTag;
}

```

## 2 Rule: one attribute = one specification

This is the key principle.

Never create "search-by-everything" specs.

## 3 Attribute-level specifications

### Order attributes

java Copy code

```

public final class OrderSpecs {

    private OrderSpecs() {}

    public static Specification<Order> customerEquals(String customer) {
        return (root, query, cb) ->
            customer == null ? cb.conjunction()

```

```

        : cb.equal(root.get("customer"), customer);
    }

    public static Specification<Order> statusEquals(OrderStatus status) {
        return (root, query, cb) ->
            status == null ? cb.conjunction()
                : cb.equal(root.get("status"), status);
    }

    public static Specification<Order> orderDateBetween(
        LocalDate from, LocalDate to) {

        return (root, query, cb) -> {
            if (from == null && to == null) {
                return cb.conjunction();
            }
            if (from != null && to != null) {
                return cb.between(root.get("orderDate"), from, to);
            }
            return from != null
                ? cb.greaterThanOrEqualTo(root.get("orderDate"), from)
                : cb.lessThanOrEqualTo(root.get("orderDate"), to);
        };
    }
}

```

## OrderItem attributes

```

java 📄 Copy code

public final class OrderItemSpecs {

    private OrderItemSpecs() {}

    public static Specification<Order> productEquals(String product) {
        return (root, query, cb) -> {
            if (product == null) return cb.conjunction();

            JoinContext ctx = JoinContextHolder.get(query);
            Join<Order, OrderItem> items =
                ctx.getOrCreate(root, "items", JoinType.LEFT);

```



```

        return cb.equal(items.get("product"), product);
    };
}

public static Specification<Order> priceBetween(
    BigDecimal min, BigDecimal max) {

    return (root, query, cb) -> {
        if (min == null && max == null) return cb.conjunction();

        JoinContext ctx = JoinContextHolder.get(query);
        Join<Order, OrderItem> items =
            ctx.getOrCreate(root, "items", JoinType.LEFT);

        if (min != null && max != null) {
            return cb.between(items.get("price"), min, max);
        }
        return min != null
            ? cb.greaterThanOrEqualTo(items.get("price"), min)
            : cb.lessThanOrEqualTo(items.get("price"), max);
    };
}
}

```

## ItemTag attributes

java Copy code

```

public final class ItemTagSpecs {

    private ItemTagSpecs() {}

    public static Specification<Order> tagNameEquals(String name) {
        return (root, query, cb) -> {
            if (name == null) return cb.conjunction();

            JoinContext ctx = JoinContextHolder.get(query);

            Join<Order, OrderItem> items =
                ctx.getOrCreate(root, "items", JoinType.LEFT);

```

```

        Join<OrderItem, ItemTag> tags =
            ctx.getOrCreate(items, "tags", JoinType.LEFT);

        return cb.equal(tags.get("name"), name);
    };
}

public static Specification<Order> tagCategoryEquals(String category) {
    return (root, query, cb) -> {
        if (category == null) return cb.conjunction();

        JoinContext ctx = JoinContextHolder.get(query);

        Join<Order, OrderItem> items =
            ctx.getOrCreate(root, "items", JoinType.LEFT);

        Join<OrderItem, ItemTag> tags =
            ctx.getOrCreate(items, "tags", JoinType.LEFT);

        return cb.equal(tags.get("category"), category);
    };
}
}

```

## TagAttribute attributes

java Copy code

```

public final class TagAttributeSpecs {

    private TagAttributeSpecs() {}

    public static Specification<Order> attributeNameEquals(String name) {
        return (root, query, cb) -> {
            if (name == null) return cb.conjunction();

            JoinContext ctx = JoinContextHolder.get(query);

            Join<Order, OrderItem> items =
                ctx.getOrCreate(root, "items", JoinType.LEFT);

```

```

        Join<OrderItem, ItemTag> tags =
            ctx.getOrCreate(items, "tags", JoinType.LEFT);

        Join<ItemTag, TagAttribute> attrs =
            ctx.getOrCreate(tags, "attributes", JoinType.LEFT);

        return cb.equal(attrs.get("name"), name);
    };
}

public static Specification<Order> attributeValueEquals(String value) {
    return (root, query, cb) -> {
        if (value == null) return cb.conjunction();

        JoinContext ctx = JoinContextHolder.get(query);

        Join<Order, OrderItem> items =
            ctx.getOrCreate(root, "items", JoinType.LEFT);

        Join<OrderItem, ItemTag> tags =
            ctx.getOrCreate(items, "tags", JoinType.LEFT);

        Join<ItemTag, TagAttribute> attrs =
            ctx.getOrCreate(tags, "attributes", JoinType.LEFT);

        return cb.equal(attrs.get("value"), value);
    };
}
}

```

## 4 Composing “search by all” vs “search by one”

### Search by one attribute

```

java                                                                    Copy code

Specification<Order> spec =
    Specification.where(BaseSpecifications.withJoinContext())
        .and(OrderItemSpecs.productEquals("Laptop"));

```

### Search by many attributes (all)

java

 Copy code

```
Specification<Order> spec =
    Specification.where(BaseSpecifications.withJoinContext())
        .and(OrderSpecs.customerEquals("John"))
        .and(OrderSpecs.statusEquals(OrderStatus.COMPLETED))
        .and(OrderItemSpecs.priceBetween(
            new BigDecimal("100"), new BigDecimal("500")))
        .and(ItemTagSpecs.tagCategoryEquals("PROMO"))
        .and(TagAttributeSpecs.attributeNameEquals("COLOR"));
```

## 5 Optional: Dynamic “search by everything” builder

This avoids manual `.and()` chaining.

java

 Copy code

```
public class OrderSearchCriteria {
    String customer;
    OrderStatus status;
    LocalDate fromDate;
    LocalDate toDate;
    String product;
    BigDecimal minPrice;
    BigDecimal maxPrice;
    String tagName;
    String tagCategory;
    String attributeName;
    String attributeValue;
}
```

java

 Copy code

```
public class OrderSpecificationBuilder {

    public static Specification<Order> build(OrderSearchCriteria c) {
        return Specification.where(BaseSpecifications.withJoinContext())
            .and(OrderSpecs.customerEquals(c.customer))
            .and(OrderSpecs.statusEquals(c.status))
            .and(OrderSpecs.orderDateBetween(c.fromDate, c.toDate))
            .and(OrderItemSpecs.productEquals(c.product))
            .and(OrderItemSpecs.priceBetween(c.minPrice, c.maxPrice))
```

```

        .and(ItemTagSpecs.tagNameEquals(c.tagName))
        .and(ItemTagSpecs.tagCategoryEquals(c.tagCategory))
        .and(TagAttributeSpecs.attributeNameEquals(c.attributeName))
        .and(TagAttributeSpecs.attributeValueEquals(c.attributeValue));
    }
}

```

## 6 Why this does NOT compromise composability

Concern	Result
Single responsibility	✓ One spec = one attribute
Reusability	✓ Any combination
Join duplication	✓ JoinContext
Optional filters	✓ <code>cb.conjunction()</code>
Scalability	✓ Add attribute → add spec
Readability	✓ Explicit intent

## 7 Golden rule (memorize this)

You compose behavior, not queries.

You never ask:

“Which query do I need?”

You ask:

“Which predicates apply?”

## Final takeaway

This pattern gives you:

- ✓ Unlimited attributes
- ✓ Dynamic AND / OR composition
- ✓ Deep joins
- ✓ Zero duplicate joins
- ✓ Clean, testable code

If you want next, I can:

- Add **OR groups**
- Show **multi-value filters (IN)**
- Add **case-insensitive / LIKE** searching
- Convert this to **QueryDSL-style DSL**

Just tell me.



Can i group predicates under methods like searchfull() adds all predicates searchhalf() adds half of predicates only and use this methods to generate Specifications for search

Yes — **you absolutely can**, and if you do it **the right way**, it does **NOT** compromise composability.

The key is **what you group**.

- ✓ You may group **compositions of specs**
- ✗ You should NOT group **logic inside specs**

Below is the **correct, scalable pattern**.

## Core rule (very important)

Specifications stay atomic.

Grouping methods only compose them.

Think in layers:

css

 Copy code

[ Attribute Specs ] → [ Grouped Specs ] → [ Search Builders ]

### 1 Attribute specs remain unchanged

Example (already correct):

java

 Copy code

```
OrderSpecs.customerEquals(...)
OrderItemSpecs.productEquals(...)
ItemTagSpecs.tagCategoryEquals(...)
TagAttributeSpecs.attributeNameEquals(...)
```

These are **never** aware of grouping.

## 2 Grouped specifications (your `searchFull`, `searchHalf`)

These are just **compositions**.

java Copy code

```
public final class OrderSearchGroups {

    private OrderSearchGroups() {}

    public static Specification<Order> searchFull(OrderSearchCriteria c) {
        return Specification
            .where(OrderSpecs.customerEquals(c.customer))
            .and(OrderSpecs.statusEquals(c.status))
            .and(OrderSpecs.orderDateBetween(c.fromDate, c.toDate))
            .and(OrderItemSpecs.productEquals(c.product))
            .and(OrderItemSpecs.priceBetween(c.minPrice, c.maxPrice))
            .and(ItemTagSpecs.tagNameEquals(c.tagName))
            .and(ItemTagSpecs.tagCategoryEquals(c.tagCategory))
            .and(TagAttributeSpecs.attributeNameEquals(c.attributeName))
            .and(TagAttributeSpecs.attributeValueEquals(c.attributeValue));
    }

    public static Specification<Order> searchHalf(OrderSearchCriteria c) {
        return Specification
            .where(OrderSpecs.customerEquals(c.customer))
            .and(OrderItemSpecs.productEquals(c.product))
            .and(ItemTagSpecs.tagCategoryEquals(c.tagCategory))
            .and(TagAttributeSpecs.attributeNameEquals(c.attributeName));
    }
}
```

✓ No joins

✓ No predicates

✓ No JPA logic

Just composition.

### 3 Always wrap with base spec

```
java Copy code  
  
Specification<Order> spec =  
    Specification.where(BaseSpecifications.withJoinContext())  
        .and(OrderSearchGroups.searchFull(criteria));
```

Or:

```
java Copy code  
  
Specification<Order> spec =  
    BaseSpecifications.withJoinContext()  
        .and(OrderSearchGroups.searchHalf(criteria));
```

### 4 Why this works perfectly

Risk	Why it's safe
Hard-coded queries	✗ avoided
Duplicate joins	✗ JoinContext
Spec explosion	✗ grouping reduces noise
Loss of flexibility	✗ still composable
Testing difficulty	✗ groups testable

### 5 Advanced: conditional grouping

You can make groups **data-driven**.

```
java Copy code  
  
public static Specification<Order> searchMinimal(OrderSearchCriteria c) {  
    Specification<Order> spec = Specification.where(null);  
  
    if (c.customer != null) {
```



```

        spec = spec.and(OrderSpecs.customerEquals(c.customer));
    }
    if (c.product != null) {
        spec = spec.and(OrderItemSpecs.productEquals(c.product));
    }
    if (c.tagCategory != null) {
        spec = spec.and(ItemTagSpecs.tagCategoryEquals(c.tagCategory));
    }
    return spec;
}

```

This is useful for **role-based searching** or **API versions**.

## 6 Even better: enum-based grouping (clean API)

```

java Copy code

public enum SearchMode {
    FULL,
    HALF,
    MINIMAL
}

```

```

java Copy code

public static Specification<Order> byMode(
    SearchMode mode, OrderSearchCriteria c) {

    return switch (mode) {
        case FULL -> searchFull(c);
        case HALF -> searchHalf(c);
        case MINIMAL -> searchMinimal(c);
    };
}

```

## 7 What NOT to do (anti-pattern 🚫)

```

java Copy code

// ❌ BAD
public static Specification<Order> searchFull(...) {
    return (root, query, cb) -> {

```

```
// joins
// predicates
// conditions

};

}
```

Why bad?

- Not composable
- Not reusable
- Hard to test
- Breaks JoinContext pattern

## 8 Mental model (remember this)

Specifications describe “what”

Groups describe “when together”

## Final answer

- ✓ Yes, group predicates via **grouping methods**
- ✓ Keep attribute specs atomic
- ✓ Group only by composing specs
- ✓ Wrap everything with your base JoinContext spec

This gives you:

- Clean API
- Maximum flexibility
- Zero duplication
- Long-term maintainability

If you want next, I can show:

- OR-grouped searches ( (A AND B) OR (C AND D) )
- Role-based predicate groups
- Pagination + sorting with grouped specs

Just say it.



Great question — pagination + sorting is **exactly where grouped specs shine**, and Spring Data already gives you most of what you need.

The trick is to **keep pagination & sorting outside the specs**, while still supporting **deep-join sorting**.

Below is a **clean, production-grade setup** that works with:

- ✓ grouped specifications ( `searchFull` , `searchHalf` )
- ✓ deep joins (items → tags → attributes)
- ✓ pagination
- ✓ sorting (root + nested fields)
- ✓ zero duplicate joins

## 1 Repository setup

```
java                                                                    Copy code

public interface OrderRepository
    extends JpaRepository<Order, Long>,
        JpaSpecificationExecutor<Order> {
}
```

This is required.

## 2 Rule: specs never know about paging

Specifications filter.  
Pageable paginates.  
Sort orders.

This separation keeps everything composable.

## 3 Simple pagination + sorting (root fields)

```
java                                                                    Copy code

Pageable pageable =
    PageRequest.of(
```

- ✓ Works out of the box
- ✓ Uses grouped specs
- ✓ No extra code

Spring cannot automatically sort across joins in `Specification`.

## 5 SortSpec (composable ordering)

## Sort request model

Examples:

- ## Sort specification

java

 Copy code

```

public final class OrderSortSpecification {

    private OrderSortSpecification() {}

    public static Specification<Order> applySort(List<OrderSort> sorts) {
        return (root, query, cb) -> {
            if (sorts == null || sorts.isEmpty()) {
                return cb.conjunction();
            }

            JoinContext ctx = JoinContextHolder.get(query);
            List<Order> orders = new ArrayList<>();

            for (OrderSort sort : sorts) {
                Path<?> path = resolvePath(root, ctx, sort.property());
                orders.add(
                    sort.direction() == Sort.Direction.ASC
                        ? cb.asc(path)
                        : cb.desc(path)
                );
            }

            query.orderBy(orders);
            return cb.conjunction();
        };
    }

    private static Path<?> resolvePath(
        Root<Order> root,
        JoinContext ctx,
        String propertyPath
    ) {
        String[] parts = propertyPath.split("\\.");
        From<?, ?> from = root;

        for (int i = 0; i < parts.length - 1; i++) {
            from = ctx.getOrCreate(from, parts[i], JoinType.LEFT);
        }
        return from.get(parts[parts.length - 1]);
    }
}

```

```
}  
}
```

- ✓ Reuses joins
- ✓ Unlimited depth
- ✓ Works with grouped specs

## 6 Pagination + deep sorting together

```
java                                                                    Copy code  
  
Specification<Order> spec =  
    BaseSpecifications.withJoinContext()  
        .and(OrderSearchGroups.searchHalf(criteria))  
        .and(OrderSortSpecification.applySort(  
            List.of(  
                new OrderSort("orderDate", Sort.Direction.DESC),  
                new OrderSort("items.product", Sort.Direction.ASC),  
                new OrderSort("items.tags.name", Sort.Direction.ASC)  
            )  
        ));  
  
Pageable pageable = PageRequest.of(0, 20);  
  
Page<Order> page = orderRepository.findAll(spec, pageable);
```

📌 Note:

- When ordering is applied inside the spec, **do NOT** pass `Sort` to `Pageable`
- `Pageable` still controls **limit / offset**

## 7 Mixed mode (root sort via Pageable, deep sort via spec)

Allowed — but be explicit.

```
java                                                                    Copy code  
  
Pageable pageable =  
    PageRequest.of(  
        0,  
        20,  
        Sort.by("customer") // root field only  
    );
```

```
Specification<Order> spec =
    BaseSpecifications.withJoinContext()
        .and(OrderSearchGroups.searchFull(criteria))
        .and(OrderSortSpecification.applySort(
            List.of(
                new OrderSort("items.price", Sort.Direction.DESC)
            )
        ));
```

## 8 Avoid common pitfalls 🚨

### 1. Missing `distinct(true)`

Always ensure:

```
java                                                                    Copy code

query.distinct(true);
```

Handled already in `BaseSpecifications.withJoinContext()`.

### 2. Sorting on collections

Sorting by collection attributes:

- Can duplicate rows internally
- Works correctly **only with** `distinct`

### 3. Pageable sort + spec sort conflict

If both are provided:

- JPA provider may ignore one
- Prefer **spec-based ordering** for nested paths

## 9 Controller example (clean API)

```
java                                                                    Copy code

@GetMapping("/orders")
public Page<Order> search(
    OrderSearchCriteria criteria,
    @RequestParam SearchMode mode,
    @RequestParam int page,
    @RequestParam int size,
    @RequestParam List<String> sort
) {
```

```

List<OrderSort> sorts = sort.stream()
    .map(s -> {
        String[] p = s.split(",");
        return new OrderSort(p[0], Sort.Direction.fromString(p[1]));
    })
    .toList();


Specification<Order> spec =
    BaseSpecifications.withJoinContext()
        .and(OrderSearchGroups.byMode(mode, criteria))
        .and(OrderSortSpecification.applySort(sorts));

return orderRepository.findAll(
    spec,
    PageRequest.of(page, size)
);
}

```

Example request:

pgsql

 Copy code

```

/orders?mode=FULL&page=0&size=20
&sort=orderDate,DESC
&sort=items.tags.name,ASC

```

## Mental model (remember this)

sql

 Copy code

```

Specifications  → filtering + ordering
Pageable       → paging only
Grouping      → reuse
JoinContext    → safety

```

## Final takeaway

- ✓ Grouped specs work perfectly with pagination
- ✓ Deep sorting is solved via a dedicated SortSpec
- ✓ Paging remains clean & independent
- ✓ No loss of composability



If you want next, I can show:

- Multi-column OR sorting
- NULLS FIRST / LAST
- Case-insensitive sorting
- Cursor-based pagination with specs

Just tell me.



With this code does not hibernate throws HHH000104:  
firstResult/maxResults specified with collection fetch; applying in  
memory

Short answer: **yes, with the code as shown you *can* trigger**

sql

 Copy code

HHH000104: firstResult/maxResults specified with collection fetch; applying in memory

**but only under specific conditions** — and it's fixable without giving up grouped specs or deep sorting.

Let's break it down precisely and then fix it properly.

## Why this warning happens (exact reason)


Hibernate throws **HHH000104** when **ALL** of the following are true:

1. Pagination is applied ( `setFirstResult` / `setMaxResults` )
2. A collection is fetched ( `join fetch` or implicit fetch join )
3. The collection is **to-many** ( `@OneToMany` , `@ManyToMany` )


Hibernate **cannot paginate SQL rows correctly** when a collection fetch join multiplies rows, so it:

- Fetches everything
- Applies pagination in memory
- Emits the warning

## Important clarification (your current code)

 Your specs use `join()` , NOT `fetch()`

java

 Copy code

```
ctx.getOrCreate(root, "items", JoinType.LEFT);
```

This is a **normal join**, not a fetch join.

 **Normal joins do NOT trigger HHH000104**

So **by default**, the setup I gave you **does NOT** cause the warning.

## When you **WILL** see HHH000104

You will get the warning if **anywhere** in the query you do this:

java

 Copy code

```
root.fetch("items", JoinType.LEFT);
```

or

java

 Copy code

```
items.fetch("tags", JoinType.LEFT);
```

or Hibernate decides to fetch because of:

- `@EntityGraph` with collections
- `FetchType.EAGER` on collections
- Custom repository method with `JOIN FETCH`

## Typical mistake that causes it

People often “optimize” like this:

java

 Copy code

```
Join<Order, OrderItem> items =  
    (Join<Order, OrderItem>) root.fetch("items", JoinType.LEFT);
```

 This immediately breaks pagination.

## The golden rule (memorize this)

Never use `FETCH JOIN` on collections in paginated queries

## Correct architecture (production-grade)

## Step 1: Page IDs only (no fetch joins)

Your current specification setup is already correct for this.

```
java                                                                    Copy code

Page<Order> page = orderRepository.findAll(spec, pageable);
```

This query:

- Filters
- Sorts
- Paginates
- Uses joins, not fetch joins
- ☒ No HHH000104

## Step 2: Fetch associations in a SECOND query

After pagination, fetch the graph by IDs.

### Option A: `@EntityGraph` (recommended)

```
java                                                                    Copy code

@EntityGraph(attributePaths = {
    "items",
    "items.tags",
    "items.tags.attributes"
})
List<Order> findByIdIn(Collection<Long> ids);
```

Usage:

```
java                                                                    Copy code

Page<Order> page = orderRepository.findAll(spec, pageable);

List<Order> content =
    orderRepository.findByIdIn(
        page.getContent()
            .stream()
            .map(Order::getId)
            .toList()
    );
```

- ✓ Pagination safe
- ✓ No warning
- ✓ One extra query
- ✓ Correct object graph

### Option B: Batch fetching (simpler)

```
java                                                                    Copy code

@OneToMany(mappedBy = "order")
@BatchSize(size = 50)
private Set<OrderItem> items;
```

Hibernate will fetch collections in batches **without fetch joins**.

- ✓ No warning
- ✓ Less SQL
- ✓ Works well for read-heavy systems

### What about sorting by collection attributes?

Sorting **does NOT** require fetch joins.

This is safe:

```
java                                                                    Copy code

LEFT JOIN items
ORDER BY items.product
```

Hibernate only fetches root entities.

### Checklist: how to guarantee NO HHH000104

#### ✓ Do this

- Use `join()` only
- Keep `distinct(true)`
- Paginate only root entity
- Fetch collections separately

#### ✗ Never do this

- `fetch()` on `@OneToMany` with pagination

- `@EntityGraph` on paged query
- `FetchType.EAGER` on collections

## Sanity check for your current design

Feature	Safe
Grouped specs	✓
JoinContext	✓
Deep filtering	✓
Deep sorting	✓
Pagination	✓
Fetch joins	✗ (don't use)

## Final verdict

👉 Your composable specification + pagination + sorting design is correct and safe

You only get HHH000104 if you:

- add fetch joins
- or use entity graphs on paged queries

If you want, I can:

- Refactor your repo to a **2-phase fetch pattern**
- Show a **DTO projection approach**
- Explain **why Hibernate can't fix this internally**

Just tell me which one you want.



Refactor my repo to a 2-phase fetch pattern transparently


Great, this is the **right fix** 👍

Below is a **clean, transparent 2-phase fetch refactor** that:

- ✔ avoids `HHHH000104`
- ✔ keeps your existing Specification API unchanged
- ✔ supports grouped specs, deep joins, sorting, pagination
- ✔ is transparent to callers (service calls one method)

## Goal

You want callers to do **this**:

```
java                                                                     Copy code
```

```
Page<Order> page = orderRepository.search(spec, pageable);
```

...and **never think about:**

- pagination limitations
- fetch joins
- entity graphs
- Hibernate warnings

## 1 Split repository responsibilities

## 1.1 Base repository (unchanged)

```
java
```

```
public interface OrderRepository
    extends JpaRepository<Order, Long>,
        JpaSpecificationExecutor<Order> {
}
```

```
Copy code
```

## 1.2 Custom repository API (transparent search)

```
java
```

Copy code

```
public interface OrderSearchRepository {  
    Page<Order> search(Specification<Order> spec, Pageable pageable);  
}
```

## 2 Custom repository implementation

Spring Data naming rule is **critical**:

php-template

 Copy code

<OrderRepository>Impl

## OrderRepositoryImpl

java  Copy code

```
@Repository
@RequiredArgsConstructor
public class OrderRepositoryImpl implements OrderSearchRepository {

    private final EntityManager em;
    private final OrderRepository orderRepository;

    @Override
    public Page<Order> search(
        Specification<Order> spec,
        Pageable pageable
    ) {

        // ----- Phase 1: page ONLY root entities -----
        Page<Order> page = orderRepository.findAll(spec, pageable);

        if (page.isEmpty()) {
            return page;
        }

        // ----- Phase 2: fetch associations by IDs -----
        List<Long> ids = page.getContent()
            .stream()
            .map(Order::getId)
            .toList();

        List<Order> fetched =
            em.createQuery("""
                select distinct o
                from Order o
                left join fetch o.items i
                left join fetch i.tags t
                left join fetch t.attributes
                where o.id in :ids
            """)
                .setParameter("ids", ids)
                .getResultList();
    }
}
```

```

        "", Order.class)
        .setParameter("ids", ids)
        .getResultList();

// ----- Preserve pagination order -----
Map<Long, Order> byId = fetched.stream()
    .collect(Collectors.toMap(Order::getId, Function.identity()));

List<Order> ordered = page.getContent()
    .stream()
    .map(o -> byId.get(o.getId()))
    .toList();

return new PageImpl<>(ordered, pageable, page.getTotalElements());
}
}

```

### 3 Wire it transparently

Update your main repository to **extend both interfaces**:

```

java Copy code

public interface OrderRepository
    extends JpaRepository<Order, Long>,
        JpaSpecificationExecutor<Order>,
        OrderSearchRepository {
}

```

Now Spring will:

- Use `OrderRepositoryImpl` automatically
- Expose `search()` on `OrderRepository`

### 4 Service layer (zero changes for callers)

```

java Copy code

@Service
@RequiredArgsConstructor
public class OrderService {

    private final OrderRepository orderRepository;
}

```



```

public Page<Order> searchOrders(
    OrderSearchCriteria criteria,
    SearchMode mode,
    Pageable pageable
) {
    Specification<Order> spec =
        BaseSpecifications.withJoinContext()
            .and(OrderSearchGroups.byMode(mode, criteria));

    return orderRepository.search(spec, pageable);
}

```

- ✓ No warnings
- ✓ Fully initialized graph
- ✓ Pagination correct
- ✓ Sorting preserved

## 5 Why this works (important)

Phase	Purpose
Phase 1	DB-level pagination
Phase 2	Graph fetching
No fetch join in phase 1	Avoids HHH000104
ID-based fetch	Stable + efficient
Reordering	Pagination correctness

## 6 Performance notes (real-world)

- ✓ SQL count
  - 2 queries total
  - Predictable
  - Cache-friendly
- ✓ Large pages

Works well even with `size=100`

## ✓ Index usage

Phase 1 uses indexes for filters + sort

## 7 Optional improvements

### 7.1 EntityGraph variant (cleaner JPQL)

```
java 📄 Copy code  
  
@EntityGraph(attributePaths = {  
    "items",  
    "items.tags",  
    "items.tags.attributes"  
})  
@Query("select o from Order o where o.id in :ids")  
List<Order> fetchGraph(@Param("ids") List<Long> ids);
```

Replace Phase 2 with:

```
java 📄 Copy code  
  
List<Order> fetched = orderRepository.fetchGraph(ids);
```

### 7.2 Batch fetching (if graph is huge)

```
java 📄 Copy code  
  
@BatchSize(size = 50)  
@OneToMany(mappedBy = "order")  
private Set<OrderItem> items;
```


Use when collections are optional.

## 8 What NOT to do 🚫

```
java 📄 Copy code  
  
@EntityGraph(...) // ❌ on paged query  
Page<Order> findAll(Specification spec, Pageable pageable);
```

or


java

 Copy code

```
root.fetch("items") // ❌ in specs
```

## 9 Final mental model

ini

 Copy code

```
SEARCH = filter + sort + page  
FETCH  = separate concern
```

You solved this **the correct way**.

### Final result

- ✓ Transparent API
- ✓ No Hibernate warnings
- ✓ Clean architecture
- ✓ Future-proof

If you want next, I can:

- Generalize this pattern for **any aggregate**
- Convert it into a **base repository**
- Show a **DTO + projection version**

Just say which one.



Don't use jql or sql. Refactor second fetch to use repository capabilities


Perfect constraint 👍

We'll keep **everything** inside Spring Data repository capabilities, no JPQL / no SQL, and still keep the 2-phase fetch transparent.

The clean solution here is `EntityGraph + findAllById` (or a derived query) in phase 2.

## Target API (unchanged for callers)

java


 Copy code

```
Page<Order> page = orderRepository.search(spec, pageable);
```

## 1 Repository refactor (no JPQL, no SQL)

### 1.1 Main repository

java

 Copy code

```
public interface OrderRepository
    extends JpaRepository<Order, Long>,
        JpaSpecificationExecutor<Order>,
        OrderSearchRepository {
}
```

### 1.2 EntityGraph-enabled fetch (Spring Data only)

java

 Copy code

```
public interface OrderGraphRepository {

    @EntityGraph(attributePaths = {
        "items",
        "items.tags",
        "items.tags.attributes"
    })
    List<Order> findAllById(Iterable<Long> ids);
}
```

#### Important

This is **NOT** used with pagination — only for phase 2.

### 1.3 Compose repositories (Spring magic)

Spring allows repository **interface composition**.

java

 Copy code

```
public interface OrderRepository
    extends JpaRepository<Order, Long>,
        JpaSpecificationExecutor<Order>,
        OrderSearchRepository,
```

```
OrderGraphRepository {
```

```
}
```

## 2 Custom repository implementation (transparent 2-phase fetch)

### OrderRepositoryImpl

java  Copy code

```
@Repository
@RequiredArgsConstructor
public class OrderRepositoryImpl implements OrderSearchRepository {

    private final OrderRepository delegate;

    @Override
    public Page<Order> search(
        Specification<Order> spec,
        Pageable pageable
    ) {
        // ----- Phase 1: paged root query -----
        Page<Order> page = delegate.findAll(spec, pageable);

        if (page.isEmpty()) {
            return page;
        }

        // ----- Phase 2: fetch graph by IDs -----
        List<Long> ids = page.getContent()
            .stream()
            .map(Order::getId)
            .toList();

        List<Order> fetched = delegate.findAllById(ids);

        // ----- Preserve pagination order -----
        Map<Long, Order> byId = fetched.stream()
            .collect(Collectors.toMap(Order::getId, Function.identity()));

        List<Order> ordered = page.getContent()
            .stream()
```

```

        .map(o -> byId.get(o.getId()))
        .toList();

    return new PageImpl<>(ordered, pageable, page.getTotalElements());
}
}

```

- ✓ No JPQL
- ✓ No SQL
- ✓ No Hibernate warning
- ✓ Fully initialized graph

### 3 Why this is safe

Aspect	Why
Pagination	Only in phase 1
EntityGraph	Only in phase 2
Collection fetch	Not paged
Spring Data only	✓
Composable specs	✓

### 4 Important EntityGraph rules

#### ✓ Allowed

```

java                                                                    Copy code

@EntityGraph
List<Order> findAllById(Iterable<Long> ids);

```

#### ✗ Forbidden

```

java                                                                    Copy code

@EntityGraph
Page<Order> findAll(Specification spec, Pageable pageable);

```

This would reintroduce `HHH000104` .

## 5 Optional: dynamic graph selection

If you want different graphs per use case:

```
java                                                                    Copy code

@NamedEntityGraph(
    name = "order.full",
    attributeNodes = @NamedAttributeNode(value = "items", subgraph = "items"),
    subgraphs = {
        @NamedSubgraph(
            name = "items",
            attributeNodes = @NamedAttributeNode(value = "tags", subgraph = "tags")
        ),
        @NamedSubgraph(
            name = "tags",
            attributeNodes = @NamedAttributeNode("attributes")
        )
    }
)

@Entity
public class Order { ... }
```

```
java                                                                    Copy code

@EntityGraph("order.full")
List<Order> findAllById(Iterable<Long> ids);
```

## 6 Zero changes to specs / grouping / sorting


Your existing:

- JoinContext
- Grouped specs
- Sorting spec
- Pagination

remain unchanged.

## 7 Final mental model

CSS

 Copy code

Spec + Pageable → IDs  
EntityGraph → **Object** graph  
Reorder → Correct **page**

## Final takeaway

You now have a **fully transparent, Spring-only, warning-free, scalable 2-phase fetch** that:

- ✓ respects Hibernate limits
- ✓ keeps composability
- ✓ avoids JPQL/SQL entirely

If you want next, I can:

- Generalize this into a **BaseSearchRepository**
- Add **conditional graphs**
- Show a **DTO-based alternative**

Just say the word.



Generalize this into a **BaseSearchRepository** and add conditional graphs

Excellent — this is the **final form** of the pattern.

We'll build a **generic, reusable BaseSearchRepository** with **conditional EntityGraphs**, while keeping:

- ✓ Spring Data only (no JPQL / SQL)
- ✓ Specifications untouched
- ✓ Pagination safe
- ✓ Fetch strategy configurable per use case
- ✓ Transparent to callers

## 1 Design goals (so the solution makes sense)

We want:

java



 Copy code


```
Page<Order> page =  
    orderRepository.search(spec, pageable, FetchProfile.FULL);
```

And we want the **same base infrastructure** to work for:

- `Order`
- any other aggregate root
- different fetch depths ( `LIGHT` , `FULL` , etc.)

## 2 Fetch profile abstraction (conditional graphs)

### 2.1 FetchProfile enum

```
java  Copy code
```

```
public enum FetchProfile {  
    NONE,          // no second fetch  
    LIGHT,         // shallow graph  
    FULL           // deep graph  
}
```

### 2.2 Graph selector interface (entity-specific)

Each aggregate decides which graph to use.

```
java  Copy code
```

```
public interface GraphSelector<T, ID> {  
  
    FetchProfile defaultProfile();  
  
    List<T> fetch(  
        FetchProfile profile,  
        Iterable<ID> ids  
    );  
}
```

This keeps the **base repository** generic.

## 3 BaseSearchRepository (generic contract)

```
java  Copy code
```

```
public interface BaseSearchRepository<T, ID> {

    Page<T> search(
        Specification<T> spec,
        Pageable pageable,
        FetchProfile profile
    );
}
```

## 4 BaseSearchRepositoryImpl (generic implementation)

This is the core reusable engine.

```
java 📄 Copy code

@RequiredArgsConstructor
public abstract class BaseSearchRepositoryImpl<T, ID>
    implements BaseSearchRepository<T, ID> {

    private final JpaSpecificationExecutor<T> specRepo;
    private final GraphSelector<T, ID> graphSelector;

    @Override
    public Page<T> search(
        Specification<T> spec,
        Pageable pageable,
        FetchProfile profile
    ) {
        // ----- Phase 1: paged root query -----
        Page<T> page = specRepo.findAll(spec, pageable);

        if (page.isEmpty() || profile == FetchProfile.NONE) {
            return page;
        }

        // ----- Phase 2: fetch graph -----
        List<ID> ids = page.getContent()
            .stream()
            .map(this::getId)
            .toList();
```

```

FetchProfile effectiveProfile =
    profile != null
        ? profile
        : graphSelector.defaultProfile();

List<T> fetched =
    graphSelector.fetch(effectiveProfile, ids);

// ----- Preserve pagination order -----
Map<ID, T> byId = fetched.stream()
    .collect(Collectors.toMap(this::getId, Function.identity()));

List<T> ordered = page.getContent()
    .stream()
    .map(e -> byId.get(this.getId(e)))
    .toList();

return new PageImpl<>(ordered, pageable, page.getTotalElements());
}

/**
 * Each entity defines how to extract its ID
 */
protected abstract ID getId(T entity);
}

```

- ✓ No entity knowledge
- ✓ No JPQL
- ✓ No SQL
- ✓ Fully reusable

## 5 Order-specific graph configuration

### 5.1 Named EntityGraphs on entity

```

java 📄 Copy code

@NamedEntityGraph(
    name = "order.light",
    attributeNodes = @NamedAttributeNode("items")
)

```

```

@NamedEntityGraph(
    name = "order.full",
    attributeNodes = @NamedAttributeNode(
        value = "items",
        subgraph = "items"
    ),
    subgraphs = {
        @NamedSubgraph(
            name = "items",
            attributeNodes = @NamedAttributeNode(
                value = "tags",
                subgraph = "tags"
            )
        ),
        @NamedSubgraph(
            name = "tags",
            attributeNodes = @NamedAttributeNode("attributes")
        )
    }
)
@Entity
public class Order {
    ...
}

```

## 5.2 OrderGraphSelector (entity-specific)

```

java Copy code

@Component
@RequiredArgsConstructor
public class OrderGraphSelector
    implements GraphSelector<Order, Long> {

    private final OrderRepository repo;

    @Override
    public FetchProfile defaultProfile() {
        return FetchProfile.LIGHT;
    }
}

```

```

@Override
public List<Order> fetch(
    FetchProfile profile,
    Iterable<Long> ids
) {
    return switch (profile) {
        case LIGHT -> repo.findAllById(ids); // default graph
        case FULL -> repo.findAllWithFullGraph(ids);
        default -> repo.findAllById(ids);
    };
}
}

```

## 5.3 Repository methods (Spring Data only)

java Copy code

```

public interface OrderRepository
    extends JpaRepository<Order, Long>,
        JpaSpecificationExecutor<Order>,
        BaseSearchRepository<Order, Long> {

    @EntityGraph("order.full")
    List<Order> findAllWithFullGraph(Iterable<Long> ids);
}

```

⚠ No pagination here — safe.

## 6 OrderSearchRepositoryImpl (glue code)

java Copy code

```

@Repository
public class OrderSearchRepositoryImpl
    extends BaseSearchRepositoryImpl<Order, Long> {

    public OrderSearchRepositoryImpl(
        OrderRepository repo,
        OrderGraphSelector selector
    ) {
        super(repo, selector);
    }
}

```

```

@Override
protected Long getId(Order entity) {
    return entity.getId();
}
}

```

## 7 Service usage (clean & explicit)

```

java                                                                    Copy code

Specification<Order> spec =
    BaseSpecifications.withJoinContext()
        .and(OrderSearchGroups.byMode(mode, criteria));

Page<Order> page =
    orderRepository.search(
        spec,
        pageable,
        FetchProfile.FULL
    );

```

Or lightweight:

```

java                                                                    Copy code

orderRepository.search(spec, pageable, FetchProfile.NONE);

```

## 8 Why this architecture scales

Concern	Solved
Pagination safety	✓
Conditional fetching	✓
Composable specs	✓
No Hibernate warnings	✓
Entity-specific graphs	✓

Reusable core



## 9 How to add another entity

For `Customer`, you only add:

1. `@NamedEntityGraph s`
2. `CustomerGraphSelector`
3. `CustomerSearchRepositoryImpl`

Zero changes to base logic.



## Final mental model

sql

 Copy code

<code>BaseSearchRepository</code>	→ infrastructure
<code>GraphSelector</code>	→ entity-specific fetch
<code>Specification</code>	→ filtering
<code>Pageable</code>	→ paging

## Final takeaway

You now have:

- ✓ A generic, reusable, production-grade search engine
- ✓ Safe pagination
- ✓ Conditional deep fetch
- ✓ Clean separation of concerns
- ✓ Spring Data only

If you want next, I can:

- Add multi-aggregate search
- Show DTO projections with the same base
- Add role-based graph selection

Just say the word.



