JAKARTA EE

Jakarta Data

Query by Method Name Extension

1.0, October 09, 2025: Draft

Table of Contents

Specification: Jakarta Data

Version: 1.0

Status: Draft

Release: October 09, 2025

Copyright

Eclipse Foundation Specification License - v1.1

Disclaimers

Chapter 1. Introduction

Query by Method Name is a query language suitable for embedding in the names of methods written in Java. As such, its syntax is limited to the use of legal identifier characters, so the text of a query must contain neither whitespace, nor punctuation characters, nor numeric operators, nor comparison operators.

Jakarta Data 1.0 offers a Query by Method Name facility as an extension to the specification, providing a migration path for existing applications written for repository frameworks which offer similar functionality.

```
@Repository
public interface ProductRepository extends BasicRepository<Product, Long> {

  List<Product> findByName(String name);

  @OrderBy("price")
  List<Product> findByNameLike(String namePattern);

  List<Product> findByNameLikeAndPriceLessThanOrderByPriceDesc(String namePattern, float priceBelow);

}
```

The functionality described here overlaps significantly with both:

- parameter-based automatic query methods, that is, `@Find` and `@Delete`, and
- annotated query methods, that is `@Query` and Jakarta Data Query Language or Jakarta Persistence Query Language.

Therefore, these alternative approaches are strongly preferred for newly-written code.

A Jakarta Data provider is required to support the Query by Method Name extension in Jakarta Data 1.0.

A Jakarta Data provider backed by a key-value or wide-column datastore is not required to support Query by Method Name.

This functionality is considered deprecated, and the requirement that a provider support the Query by Method Name extension will be removed in a future version of Jakarta Data.

Chapter 2. Query by Method Name

In Query by Method Name, a query is expressed via a set of method naming conventions.

A method name is formed by concatenating, in the following order:

- an *action*, which must be `find`, `delete`, `count`, or `exists`,
- an optional limit,
- a optional restriction, and
- an optional order.

2.1. Limits

A `find` query may have a *limit*, for example, `First` or `First10`. Other actions must not be combined with a limit.

The limit determines the maximum number of records which may be returned by the query. If the query has an order, then the records which are returned are those which occur first after sorting.

2.2. Restrictions

A *restriction* specifies the criteria used to filter records. It is formed by concatenating `By` with one or more conditions, delimited by `And` or `Or`, for example, `PriceLessThanAndNameLike`.

Each *condition* is formed by concatenating, in the following order:

- an attribute name, which may be a compound name, as specified below in Persistent attribute names in Query by Method Name,
- optionally, `IgnoreCase` (for text-typed attributes),
- optionally, `Not`,
- optionally, an operator such as `LessThan` or `Like`.

Absence of an operator implies the equality condition.

The conditions belonging to the restriction determine the parameters of the method, as specified below in Query by Method Name Conditions.

When using NoSQL databases, the support for restrictions varies depending on the database type:

Key-value databases

Support for the equals restriction is required for the `Id` attribute. There is no requirement to support other types of restrictions, the `And` and `Or` keywords, or restrictions on other entity attributes. Queries in key-value databases are typically limited to operations using the `Id` attribute only.

Wide-column databases

Wide-column databases must support the `AND` operator but are not required to support the `Or` operator. Restrictions must be supported for the key attribute that is annotated with `jakarta.nosql.Id`. Support for restrictions on other attributes is not required. Typically they can be used if they are indexed as secondary indexes, although support varies by database provider.

Graph and document databases

The `And` and `Or` keywords and all of the restrictions described in this section must be supported.

Precedence between `And` and `Or` operators is not guaranteed and may vary significantly based on the NoSQL provider.

2.3. Orders

A `find` query may have an *order*. The *order* specifies how records must be sorted. It is formed by concatenating `OrderBy` with one or more ordered pairs of an entity attribute name and a direction of sorting, `Asc` or `Desc`. The direction may be omitted if there is only one entity attribute, in which case `Asc` is implied.

The order is lexicographic, that is, ordered pairs occurring earlier take precedence. An ordered pair occurring later is only used to resolve "ties" between records which cannot be unambiguously ordered using only earlier ordered pairs.

If no order is specified, the records are not sorted.

When using NoSQL databases, sorting support varies by database type:

Key-value databases

Support for sorting of results is not required.

Wide-column databases

Support for sorting of results is not required. In general, sorting is not natively supported. When sorting is available, it is typically limited to:

- The key attribute, defined by an annotation such as `jakarta.nosql.Id`.
- Fields that are indexed as secondary indexes.

Graph and document databases

Support for sorting by a single entity attribute is required. Support for compound sorting (sorting by multiple entity attributes) is not required and may vary due to:

- Potential instability with tied values, where sorting for equivalent values may differ across queries.
- Schema flexibility and mixed data types.
- Dependence on indexes and internal storage order, requiring proper indexing to ensure predictable sorting.
- The distributed nature of sharded clusters, where sorting across shards may introduce additional complexity.

2.4. Example query methods

The following table displays some examples of legal method signatures.

`findByName(String name)`	Find entities by the `name` attribute.
`findByAgeGreaterThan(int age)`	Find entities where `age` is greater than the specified value.
`findByAuthorName(String authorName)`	Find entities by the `authorName` attribute of a related entity.
`findByCategoryNameAndPriceLessThan(String categoryName, double price)`	Find entities by `categoryName` and `price` attributes, applying an `And` condition.

`findByNameLikeOrderByPriceDescIdAsc`	Find entities by matching the `name` attribute against a pattern, sorting the results by `price` in descending order, and sorting results with the same `price` by the `id` in ascending order.

2.5. BNF Grammar for Query Methods

The rules for parsing an interpreting a method name are specified by the following grammar.

```
query : find | action
find : "find" limit? ignoredText? restriction? order?
action : ("delete" | "count" | "exists") ignoredText? restriction?
restriction : "By" predicate
limit : "First" max?
predicate : condition (("And" | "Or") condition)*
condition : attribute "IgnoreCase"? "Not"? operator?
operator
    : "Contains"
    | "EndsWith"
    | "StartsWith"
    | "LessThan"
    | "LessThanEqual"
    | "GreaterThan"
    | "GreaterThanEqual"
    | "Between"
    | "Like"
    | "In"
    | "Null"
    | "True"
    | "False"
attribute : identifier ("_" identifier)*
identifier : word
max : digit+
order : "OrderBy" (attribute | orderItem+)
orderItem : attribute ("Asc" | "Desc")
```

Quoted names are considered case-sensitive keywords.

Table 1. Explanation of the BNF elements

Rule name	Explanation
`query`	May be a `find` query, or a `delete`, `count`, or `exists` operation.
`find`	A `find` query has an optional limit and optional restriction on records to be retrieved, and optional sorting.
`action`	Any other kind of operation has only a restriction to a subset of records.
`restriction`	Restricts the records returned to those which satisfy a predicate
`limit`	Limits the records retrieved by a `find` query to a hardcoded maximum, such as `First10`.
`ignoredText`	Optional text that does not contain `By`, `All`, or `First`.
`predicate`	A filtering criteria, which may include multiple conditions separated by `And` or `Or`.
`condition`	An attribute of the queried entity and an operator.
`operator`	An operator belonging to a condition, for example, `Between` or `LessThan`. When absent, equality is implied.

Rule name	Explanation
`attribute`	An entity attribute name, which can include underscores for nested attributes.
`identifier`	A legal Java identifier, not containing an underscore.
`max`	A positive whole number.
`order`	Specifies that results of a `find` query should be sorted lexicographically, with respect to one or more order items.
`orderItem`	An entity attribute used to sort results, where `Asc` or `Desc` specifies the sorting direction.

2.6. Query by Method Name Keywords

An implementation of Query by Method Name must support the following types of operation.

Table 2. Query by Method Name Actions

Action	Description
`find`	Returns entity instances representing the records which satisfy the restriction, or representing all records if there is no restriction.
`delete`	Deletes every record which satisfy the restriction, or all records if there is no restriction, and returns either no result (`void`) or the number of records deleted.
`count`	Returns the number of records which satisfy the restriction, or the total number of records if there is no restriction.
`exists`	Returns `true` if at least one record satisfies the restriction or if there is at least one record in the database when there is no restriction.

An implementation of Query by Method Name must support the following keywords.

Table 3. Query by Method Name Keywords

Keyword	Description	Method signature example
`And`	The `And` operator requires both conditions to match.	`findByNameAndYear`
`Or`	The `Or` operator requires at least one of the conditions to match.	`findByNameOrYear`
`Not`	Negates the condition that immediately follows the `Not` keyword. When used without a subsequent keyword, means not equal to.	`findByNameNotLike`
`First`	For a query with ordered results, limits the quantity of results to the number following First, or if there is no subsequent number, to a single result.	`findFirst10By`

Keyword	Description	Method signature example
`OrderBy`	Specify a static sorting order followed by one or more ordered pairings of an entity attribute name and direction (`Asc` or `Desc`). The direction `Asc` can be omitted from the final attribute listed, in which case ascending order is implied for that attribute.	`findByAgeOrderByHeightDescIdAsc` `findByAgeOrderById`
`Desc`	Specify a static sorting order of descending.	`findByNameOrderByAgeDesc`
`Asc`	Specify a static sorting order of ascending.	`findByNameOrderByAgeAsc`

For relational databases, the logical operator `And` takes precedence over `Or`, meaning that `And` is evaluated on conditions before `Or` when both are specified on the same method. For other database types, the precedence is limited to the capabilities of the database. For example, some graph databases are limited to precedence in traversal order.

An implementation of Query by Method Name backed by a document or graph database is not required to support the `First` keyword. A repository method must raise `java.lang.UnsupportedOperationException` or a more specific subclass of the exception if the database does not support this functionality.

2.7. Query by Method Name Conditions

In addition to equality conditions, Query by Method Name defines the following kinds of condition.

Table 4. Query by Method Name Conditions

Keyword	Attribute type	Parameters	Description	Method signature example
`Between`	Any sortable type	2	Find results where the attribute is between (inclusive of) two given values, with the first value being the inclusive minimum and the second value being the inclusive maximum.	`findByDateBetween`
`Contains`	`String`	1	Matches string values with the given substring, which can be a pattern.	`findByProductNameContains`
`EndsWith`	`String`	1	Matches String values with the given ending, which can be a pattern.	`findByProductNameEndsWith`
`LessThan`	Any sortable type	1	Find results where the attribute is less than the given value	`findByAgeLessThan`
`GreaterThan`	Any sortable type	1	Find results where the attribute is greater than the given value	`findByAgeGreaterThan`
`LessThanEqual`	Any sortable type	1	Find results where the attribute is less than or equal to the given value	`findByAgeLessThanEqual`

Keyword	Attribute type	Parameters	Description	Method signature example
`GreaterThanEqual`	Any sortable type	1	Find results where the attribute is greater than or equal to the given value	`findByAgeGreaterThanEqual`
`Like`	`String`	1	Matches string values against the given pattern.	`findByTitleLike`
`IgnoreCase`	`String`		Requests that string values be compared independent of case for query conditions and ordering.	`findByStreetNameIgnoreCaseLike`
`In`	Any type	1 `Set`	Find results where the attribute is one of the values that are contained within the given `Set`.	`findByIdIn`
`Null`	Any type	0	Finds results where the attribute has a null value.	`findByYearRetiredNull`
`StartsWith`	`String`	1	Matches String values with the given beginning, which can be a pattern.	`findByFirstNameStartsWith`
`True`	`Boolean` or `boolean`	0	Finds results where the attribute has a boolean value of `true`.	`findBySalariedTrue`
`False`	`Boolean` or `boolean`	0	Finds results where the attribute has a boolean value of `false`.	`findByCompletedFalse`

Most *Query by Method Name* conditions require a single repository method parameter. The `Between` condition requires two parameters. `Null`, `True`, and `False` require none. An `In` condition requires a parameter of type `Set<T>` where `T` is the type of the entity attribute. The repository method parameters used for *Query by Method Name* conditions follow the order in which the *Query by Method Name* conditions appear within the method name.

Wildcard characters for patterns are determined by the data store. For relational databases, `_` matches any one character and `%` matches zero or more characters.

An implementation of Query by Method Name backed by a document or graph database is not required to support `Contains`, `EndsWith`, `StartsWith`, `Like`, `IgnoreCase`, `In`, or `Null`. A repository method must raise `java.lang.UnsupportedOperationException` or a more specific subclass of the exception if the database does not provide the requested functionality.

In the following example the value of the first parameter, `namePattern`, is used for `NameLike`, the values of the second and third parameters, `minYear` and `maxYear`, are used for `YearMadeBetween`, and the value of the fourth parameter, `maxPrice`, is used for `PriceLessThan`.

```
List<Product> findByNameLikeAndYearMadeBetweenAndPriceLessThan(String namePattern,
                                                               int minYear,
                                                               int maxYear,
                                                               float maxPrice,
                                                               Limit limit,
                                                               Order<Product> sortBy)
```

When using NoSQL databases, support for the listed method conditions may vary significantly:

Textual conditions

The `Contains`, `EndsWith`, `IgnoreCase`, `Like`, and `StartsWith` restrictions depend on text processing capabilities and are not required to be supported by Key-Value, Wide-Column, Document and Graph databases. The ability to handle these operations varies widely and depends on the database's support for text-based queries.

Null conditions

The handling of `Null` values differs across NoSQL databases. In schemaless databases, a field's absence is often treated as non-existent rather than `Null`. This distinction can affect query behavior because some databases might not support `Null` checks or might interpret them differently. This specification does not make any requirements on how `Null` and non-existent fields are handled for Key-Value, Wide-Column, Document and Graph databases.

Key-value databases

Support is required for the `=` (equals) restriction when the attribute is the key. The key attribute is defined by `jakarta.nosql.Id`.

Wide-column databases

Support for restrictions on attributes other than the `Id` attribute is not required. Some databases might allow support for restrictions on other entity attributes if a secondary index is configured. Handling of `Null` values is not required to be supported and might vary in behavior because some wide-column stores might treat missing columns as `Null`, while others might not store the column at all.

Graph and document databases

These databases are generally more feature-rich. Support for the `Contains`, `EndsWith`, `IgnoreCase`, `Like`, and `StartsWith` restrictions is not required. Similarly, `Null` values are not required to be supported. Handling of `Null` in these databases may vary, especially when dealing with omitted fields versus explicitly set `Null` values.

Developers are encouraged to consult the documentation of their chosen NoSQL database provider to confirm the supported operations and clarify the behavior of `Null` versus non-existent fields.

2.8. Return Types

The return type of a Query by Method Name is determined as indicated in the following table, where `E` is the queried entity type.

Table 5. Repository Method Return Types

Operation	Return type	Notes
`count`	`long`	
`delete`	`void`,`long`,`int`	
`exists`	`boolean`	
`find`	`E` or `Optional<E>`	For queries returning a single item (or none)

Operation	Return type	Notes
`find`	`E[]` or `List<E>`	For queries where it is possible to return more than one item
`find`	`Stream<E>`	The caller must call `java.util.stream.BaseStream.close()` for every stream returned by the repository method
`find` accepting a `PageRequest`	`Page<E>` or `CursoredPage<E>`	For use with pagination

When working with NoSQL databases, the `count` operation might not be supported, depending on the database structure and capabilities:

Key-value databases

Support for count is not required. The majority of Key-Value databases do not support counting. Their design focuses on retrieving values by keys rather than aggregating data.

Wide-column databases

Support for count is not required. For databases that are capable of returning a count, it might require additional configuration, such as enabling secondary indexes or performing costly full table scans, which can affect performance.

Graph and document databases

Support for count is required. Performance might vary based on factors like indexing and query complexity.

2.9. Persistent attribute names in Query by Method Name

Section 3.2 of the Jakarta Data specification describes how names are assigned to persistent attributes of an entity.

For Query by Method Name, the use of delimiters within a compound name is optional. Delimiters may be omitted entirely from a compound name when they are not needed to disambiguate the persistent attribute to which the name refers. But for a given name, delimiter usage must be consistent: either the delimiter must be used between every pair of persistent attribute names within the compound name, or it must not occur within the compound name.

Resolution of a persistent attribute involves the following steps:

1. A persistent attribute name is extracted from the method name according to the BNF Grammar for Query Methods. For example, if the query method name is `findByAddressZipCode`, the extracted attribute name is `AddressZipCode`.
2. The extracted name is matched against the attributes of the entity class. If the name assigned to a persistent attribute of the entity class matches the extracted name, ignoring case, then the extracted name resolves to that attribute.
3. Otherwise, if no match is found among the attributes of the entity, the extracted name is matched against the attributes of entity classes and embedded classes reachable from the entity class, interpreting the extracted name as a compound name, as outlined in the previous section, both with and without the optional delimiter. If the compound name assigned to a persistent attribute matches the extracted name, also interpreted as a compound name, and ignoring case, then the extracted name resolves to that attribute.
4. If no matching persistent attribute is found in either of the previous steps, the provider is permitted to reject the query method or to throw `UnsupportedOperationException` when the method is called.

A persistent attribute name used in a *Query by Method Name* must not contain a keyword reserved by the grammar.

2.9.1. Scenario 1: Person Repository with Unambiguous Resolution

In this scenario, we have the following data model:

```
class Person {
  private Long id;
  private MailingAddress address;
}

class MailingAddress {
  private int zipcode;
}
```

The `Person` entity does not have an `addressZipCode` attribute, so use of the delimiter is optional. It is valid to write both of the following repository methods, which have the same meaning,

```
List<Person> findByAddressZipCode(int zipCode);
List<Person> findByAddress_zipcode(int zipCode);
```

2.9.2. Scenario 2: Customer Repository with Resolution that requires a Delimiter

In this scenario, we have the following data model:

```
class Customer {
  private Long id;
  private String addressZipCode;
  private MailingAddress address;
}

class MailingAddress {
  private int zipcode;
}
```

The `Customer` entity has an `addressZipCode` attribute, as well as an `address` attribute for an embeddable class with a `zipcode` attribute. The method name `findByAddressZipCode` points to the `addressZipCode` attribute and cannot be used to navigate to the embedded class. To navigate to the `zipcode` attribute of the embedded class, the delimiter must be used:

```
List<Customer> findByAddress_zipcode(int zipCode);
```