JAKARTA EE

Jakarta Data

1.1.0-M1, October 09, 2025: Draft

Table of Contents

Specification: Jakarta Data

Version: 1.1.0-M1

Status: Draft

Release: October 09, 2025

Copyright

Eclipse Foundation Specification License - v1.1

Disclaimers

Chapter 1. Introduction

The Jakarta Data specification provides an API to simplify data access. It enables the Java developer to focus on the data model, while delegating away the complexities of data persistence. To make this possible, Jakarta Data includes a variety of features such as pre-built interfaces for data access, offset and cursor based pagination strategies, and the ability to compose custom query methods that the framework implements.

Data is a primary concern of most applications, and dealing with a database presents one of the most significant challenges within software architecture. Beyond selecting from the various database options available in the market, it is necessary to consider the intricacies of persistence integrations. Jakarta Data simplifies the lives of Java developers by providing a solution that streamlines data access and manipulation.

In this context, a domain-centric approach refers to designing the application’s architecture primarily focusing on the domain model. It means that the application’s data and logic structure and organization revolve around the core domain concepts and business rules, ensuring that the domain model plays a central role in shaping the application’s structure.

1.1. Goals

Jakarta Data addresses a fundamental challenge in Java application development: the seamless integration of diverse data sources amid the dissimilarities in their respective programming models. Offering the Java developer a common, familiar starting point for data access is helpful for solutions involving multiple databases and storage technologies.

The primary problem Jakarta Data sets out to solve is the complexity and inconsistency that arises when Java applications encounter various database systems—relational, document, column, key-value, graph, and others. Managing these diverse data sources can be daunting, often requiring developers to write specialized code for each storage technology.

Jakarta Data combines the concept of a persistence agnostic API with a domain-centric approach. This approach enables developers to work with different databases and storage engines while aligning their data access strategies with the core principles of a domain-centric architecture, where the domain model plays a central role in shaping the application’s structure.

Jakarta Data is guided by a set of clear and well-defined objectives to simplify data integration and enhance data access for Java developers. These objectives serve as the pillars of its design philosophy, ensuring that it addresses real-world challenges and provides concrete advantages to developers:

Jakarta Data is engineered to tackle a fundamental problem

simplifying data access and manipulation within Java applications that interact with diverse databases and storage sources.

Jakarta Data is designed to be persistence agnostic

In this context, agnostic does not mean that you can switch the underlying persistence without changes but implies that Jakarta Data is not tied to a specific database technology. It offers a flexible, adaptable framework that allows you to work with the databases and storage sources that best suit your project’s needs. This agnostic approach ensures that Jakarta Data can cater to various use cases.

Enhancing a domain-centric approach

Jakarta Data enhances the concept of a persistence agnostic API by incorporating a domain-centric approach. It enables developers to align their data access strategies with the core principles of a domain-centric architecture, where the domain model plays a central role in shaping the application’s structure.

Unified API

Jakarta Data provides a unified and standardized API for interacting with various data sources. This consistency simplifies development by allowing developers to use the same tools and practices regardless of the underlying database technology.

Pluggable and extensible

Jakarta Data is designed to be pluggable and extensible. Even in cases where the API doesn't directly support a specific behavior of a storage engine, Jakarta Data aims to provide an extensible API to enable developers to customize and adapt as needed.

Simplified and domain-centric querying and database operations

Jakarta Data strongly emphasizes simplifying and aligning querying and database operations with your application's domain model. By offering domain-centric query capabilities through annotations, built-in repository interfaces, or query-by-method, Jakarta Data strives to be compatible with multiple databases and inherently closer to your application's domain logic. This approach ensures that your queries and operations are more versatile across various persistence engines, making working with different data sources easier while maintaining a cohesive and domain-focused codebase.

Seamless integration

Jakarta Data enables seamless integration between Java applications and various persistence layers, making it easier for developers to work with different databases and storage sources without extensive customization.

1.2. Non-goals

The following are not goals of Jakarta Data:

Specific features of Jakarta Persistence, Jakarta NoSQL, etc., and specializations

Jakarta Data does not intend to replicate or replace the specific features provided by other Jakarta specifications, such as Jakarta Persistence and Jakarta NoSQL, along with their associated specializations and extensions. These specifications have well-defined scopes and functionalities that cater to specific use cases. Jakarta Data operates with the understanding that it complements these specifications by providing a higher-level, agnostic API. It does not seek to duplicate their capabilities but aims to simplify data access and integration across diverse data sources.

Replacement of Jakarta Persistence or Jakarta NoSQL specifications

Jakarta Data's primary goal is not to replace or supersede the Jakarta Persistence or Jakarta NoSQL specifications. Instead, it works in harmony with these specifications, serving as an additional layer that abstracts the complexities of data access. Jakarta Data enhances the developer experience by offering a persistence-agnostic approach while leveraging the capabilities of Jakarta Persistence and Jakarta NoSQL. Its role is to complement and simplify, not replace, these established specifications.

1.3. Conventions

The term *entity attribute*, and in some places *persistent attribute*, is used through the Jakarta Data specification to refer to a field or property of an `Entity` that is persisted to a data store. The Jakarta Persistence specification also uses these terms, as well as the term *persistent fields and properties*, which has the same meaning. Additionally, the Jakarta Data specification sometimes refers to an entity attribute by its name. For example, the `price` attribute of the `Product` entity. In other cases, an entity attribute is referred to by what it represents. For example, an entity's `Id` attribute or `Version` attribute.

1.4. Jakarta Data project team

This specification is being developed as part of Jakarta Data project under the Jakarta EE Specification Process. It is the result of the collaborative work of the project committers and various contributors.

1.4.1. Project leads

- Nathan Rauh
- Otavio Santana

1.4.2. Committers

- Gavin King
- Denis Stepanov
- Dmitry Kornilov
- Emily Jiang
- Graeme Rocher
- James Krueger
- James Stephens
- Kyle Aure
- Mark Swatosh
- Michael Redlich
- Nathan Rauh
- Otavio Santana
- Werner Keil

1.4.3. Mentor

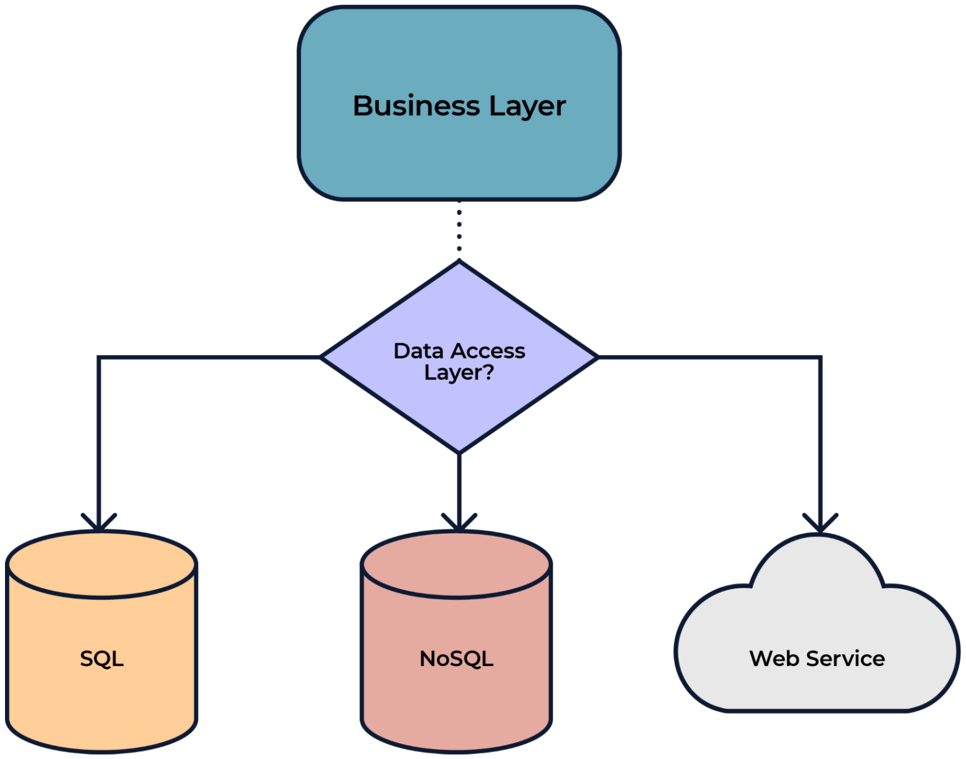
- Dmitry Kornilov

1.4.4. Contributors

The complete list of Jakarta Data contributors may be found here.

Chapter 2. Architecture

In the realm of software design, the repository pattern encapsulates the logic required to access data sources. This pattern consolidates data access functionality, offering improved maintainability and decoupling the infrastructure or technology used to access databases from the domain model layer.



The Repository pattern is a fundamental concept within Jakarta Data that plays a central role in data access and management. Essentially, a repository is a mediator between an application's domain logic and the underlying data storage, be it a relational database, NoSQL database, or any other data source.

In Jakarta Data, a Repository provides a structured and organized way to interact with data. It abstracts data storage and retrieval complexities, allowing you to work with domain-specific objects and perform common operations on data without writing low-level database queries.

As employed in Jakarta Data, the Repository pattern exhibits several key characteristics that make it a powerful tool for managing data access within Java applications. These characteristics collectively define how repositories function within Jakarta Data, providing a structured and domain-centric approach to working with data. These key characteristics offer insight into how repositories simplify data access and enhance the maintainability of code.

Abstraction

Repositories abstract the details of how data is stored, enabling the developer to focus on the application's domain logic without being tightly coupled to a specific database technology.

Structured data access

Jakarta Data repositories offer a structured and consistent way to perform data access operations. This structured approach ensures that the codebase remains organized and maintainable.

Domain-centric

Repositories are designed to be domain-centric, aligning with the application's domain model. It means that data access operations are closely tied to business entities, making code more intuitive and expressive.

In summary, the Repository pattern in Jakarta Data offers a structured and domain-centric approach to data access, providing a balance between abstraction and ease of use. It simplifies data access by encapsulating the details of the data source while aligning closely with the application's domain model. It makes it a valuable choice for many Java developers, especially in projects where a clean separation of concerns and maintainable codebase are essential.

2.1. Repositories in Jakarta Data

Within the context of Jakarta Data, the repository plays a pivotal role in simplifying data access for various persistence stores. The repository is a Java interface that acts as a gateway for accessing persistent data of one or more entity types. Repositories offer a streamlined approach to working with data by exposing operations for querying, retrieving, and modifying entity class instances that represent data in the persistent store.

Several characteristics define repositories:

Reduced boilerplate code

One of the primary goals of a repository abstraction is to significantly reduce the boilerplate code required to implement data access layers for diverse persistence stores. This reduction in repetitive code enhances code maintainability and developer productivity.

Jakarta Data annotations

In Jakarta Data, repositories are defined as interfaces and are annotated with the `@Repository` annotation. This annotation serves as a marker to indicate that the interface represents a repository.

Built-in interfaces

The Jakarta Data specification provides a set of built-in interfaces from which repositories can inherit. These built-in interfaces offer a convenient way to include a variety of pre-defined methods for common operations. They also declare the entity type to use for methods where the entity type cannot otherwise be inferred.

Data retrieval and modification

Repositories facilitate data retrieval and modification operations. This includes querying for persistent instances in the data store, creating new persistent instances in the data store, removing existing persistent instances, and modifying the state of persistent instances. Conventionally, these operations are named insert, update, save and delete for modifying operations and find, count, and exists for retrieval operations.

Subset of data

Repositories may expose only a subset of the full data set available in the data store, providing a focused and controlled access point to the data.

Entity associations

Entities within a repository may have associations between them, especially in the case of relational data access. However, this specification does not define the semantics of associations between entities belonging to different repositories.

Stateless and stateful repositories

Repositories are usually stateless. As an extension, the module `jakarta.data.stateful` defines support for stateful repositories backed by Jakarta Persistence-style persistence contexts.

Repositories in Jakarta Data serve as efficient gateways for managing and interacting with persistent data, offering a simplified and consistent approach to data access and modification within Java applications.

The application must provide the following when using repositories in Jakarta Data:

Entity classes and mappings

Developers define a set of entity classes and mappings tailored to a specific data store. These entities represent the data structure and schema, offering a powerful means to interact with the underlying data.

Repository interfaces

Jakarta Data enables the creation of one or more repository interfaces, following predefined rules that include the guidelines set forth by this specification. These interfaces are the gateways to accessing and manipulating the data, offering a structured and efficient way to perform data operations.

An implementation of Jakarta Data, specifically tailored to the chosen data store, assumes the responsibility of implementing each repository interface. This symbiotic relationship between developers and Jakarta Data ensures that data access and manipulation remain consistent, efficient, and aligned with best practices.

Jakarta Data empowers developers to shape their data access strategies by defining entity classes and repositories, with implementations seamlessly adapting to the chosen data store. This flexibility and Jakarta Data's persistence-agnostic approach promote robust data management within Java applications.

The Jakarta Data specification supports two basic ways to define a repository interface:

- by extending one of the generic repository supertype interfaces defined by Jakarta Data, or
- by annotating the methods of an interface which does *not* extend any built-in supertype.

A Java developer creates an interface, marks it with the `@Repository` annotation, and has the option to extend one or more built-in generic repository interfaces, or to annotate its lifecycle methods.

Jakarta Data allows applications to intermix the two approaches by defining methods annotated with `@Insert`, `@Update`, `@Delete`, or `@Save` on repositories which inherit the built-in supertypes.

2.1.1. Repositories with built-in supertypes

Jakarta Data defines a hierarchy of built-in interfaces which user-defined repositories may inherit. At the root of this hierarchy is the `DataRepository` interface. A repository is permitted to extend one or more of the members of the hierarchy, or none at all. When a repository extends a built-in interface, the implementation of every method inherited from the built-in interface must preserve the semantics specified by the built-in interface.

```
Failed to generate image: Could not load Ditaa. Either require 'asciidoctor-diagram-ditaamini' or specify the
location of the Ditaa JAR(s) using the 'DIAGRAM_DITAA_CLASSPATH' environment variable.
                          +----------------+
                          | DataRepository |
                          +----------------+
                                  ^
                                  |
                                  |
                          +-----------------+
                          | BasicRepository |
                          +-----------------+
                                  ^
                                  |
                                  |
                        +--------------------+
                        | CrudRepository     |
                        +--------------------+
```

A repository which extends a built-in supertype usually acts as a home for operations acting on a single entity type called the *primary entity type* of the repository. The primary entity type is determined by the argument to the first generic type variable of the generic supertype.

- The `BasicRepository` interface includes some of the most common operations applying to a single type of entity, including `save()`, `delete()`, and `findById()`.
- The `CrudRepository` interface inherits `BasicRepository`, adding `insert()` and `update()` methods corresponding to the Create and Update operations of the CRUD (Create, Read, Update, Delete) pattern.

Given a `Product` entity with ID of type `long`, the repository could be as simple as:

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

}
```

There is no nomenclature restriction requiring the `Repository` suffix. For example, a repository for the `Car` entity does not need to be named `CarRepository`. It could be named `Cars`, `Vehicles`, or even `Garage`.

```
@Repository
public interface Garage extends BasicRepository<Car, String> {

}
```

2.1.2. Repositories without built-in supertypes

Alternatively, Jakarta Data allows a custom repository interface which does not extend any built-in type. This option:

- provides the developer with complete control over the operations available, and over their naming, and
- allows a single repository to declare operations acting on a family of related entities, instead of being limited to just one entity type.

In this approach, database operations involving fundamental data changes, such as insertion, update, and removal, are realized via the use of lifecycle annotations like `@Insert`, `@Update`, `@Delete`, and `@Save`. These annotations enable the crafting of expressive and contextually meaningful repository methods, resulting in a repository API that closely mirrors the semantics of the domain.

For instance, consider the `Garage` repository interface below:

```
@Repository
public interface Garage {

    @Insert
    Car park(Car car);

    @Delete
    void unpark(Car car);
}
```

Notice that the `@Insert` annotation is used to declare the `park()` method.

The previous example illustrates the design of a repository interface which captures some of the essence of the business domain. This approach fosters a shared understanding and more intuitive communication within the development team, with database operations named according to the language of the domain.

2.2. Querying in Jakarta Data

Jakarta Data provides two core ways to express queries:

- 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.

Newly written code should use these approaches. A typical application based on Jakarta Data uses a mix of both approaches, with the choice of approach depending on the complexity of the query.

As an extension to the core specification, Jakarta Data 1.0 offers a Query by Method Name facility to provide a migration path for existing applications written for repository frameworks which offer similar functionality. Query by Method name is described in a companion document to this specification.

A Jakarta Data provider is required to support the Query by Method Name extension in Jakarta Data 1.0. This requirement will be removed in a future version of Jakarta Data.

Chapter 3. Entity classes

The notion of an *entity* is the fundamental building block with which a data model may be constructed. Abstractly, an entity (or *entity type*) is a schema for data.

- The schema may be as simple as a tuple of types, as is typical in the relational model, or it might be structured, as in document data stores.
- The schema might be explicit, as in the case of SQL DDL declaring a relational table, or it might be implicit, as is commonplace in key/value stores.
- Either way, we assume that the entity is represented in Java as a class, which we call the *entity class*. [1]

> When there's no risk of confusion, we often use the word "entity" to mean the entity class, or even an instance of the entity class.

Data represented by an entity is persistent, that is, the data itself outlives any Java process which makes use of it. Thus, it is necessary to maintain an association between instances of Java entity classes and state held in a data store.

- Each persistent instantiation of the schema is distinguishable by a unique *identifier*. For example, a row of a relational database table is identifiable by the value of its primary key.
- Any persistent instantiation of the schema is representable by an instance of the entity class. In a given Java program, multiple entity class instances might represent the same persistent instance of the schema.

In Jakarta Data, the concrete definition of an entity may be understood to encompass the following aspects:

1. The **entity class** itself: An entity class is simple Java object equipped with fields or accessor methods designating each attribute of the entity. An entity class is identified by an annotation.
2. Its **data schema**: Some data storage technologies require an explicit schema defining the structure and properties of the data the entity represents. For example, a relational database requires that the schema be specified using SQL Data Definition Language (DDL) statements. The schema might be generated by the Jakarta Data provider, from the information available in the Java entity class, or it might be managed independently. When the data store itself does not require an explicit schema, the data schema is implicit.
3. Its **association with a repository**: Each entity class is associated with at least one repository, which exposes operations for retrieving and storing instances of the entity.

> A Jakarta Data provider might allow the state of a single Jakarta Data entity to be stored across multiple entities in the data store. For example, in Jakarta Persistence, the `@SecondaryTable` annotation allows the state of an entity to be mapped across more than one database table.

3.1. Programming model for entity classes

A *programming model for entity classes* specifies:

- a set of restrictions on the implementation of a Java class which allows it to be used as an entity class with a given Jakarta Data provider, and
- a set of annotations allowing the identification of a Java class as an entity class, and further specification of the schema of the entity.

Jakarta Data does not define its own programming model for entities, but instead:

- is compatible with the programming models defined by the Jakarta Persistence [2] and Jakarta NoSQL [3] specifications, and

* allows for vendor-specific entity programming models to be defined by Jakarta Data providers.

This section lays out the core requirements that an entity programming model must satisfy in order to be compatible with Jakarta Data, and for the defining provider to be considered a fully-compliant implementation of this specification.

Every entity programming model specifies an *entity-defining annotation*. For Jakarta Persistence, this is `jakarta.persistence.Entity`. For Jakarta NoSQL, it is `jakarta.nosql.Entity`. A Jakarta Data provider must provide repository implementations for entity classes bearing the entity-defining annotations it supports, and must ignore entity classes with entity-defining annotations it does not support.

> To maintain clarity and to disambiguate the desired Jakarta Data provider, a single entity class should not mix entity-defining annotations from different providers. For example, an entity class should not be annotated both `jakarta.persistence.Entity` and `jakarta.nosql.Entity`. This practice allows the entity-defining annotation to indicate the desired provider in programs where multiple Jakarta Data providers are available.

Furthermore, an entity programming model must define an annotation which identifies the attribute holding the unique identifier of an entity. For Jakarta Persistence, it is `jakarta.persistence.Id` or `jakarta.persistence.EmbeddedId`. For Jakarta NoSQL, it is `jakarta.nosql.Id`. Alternatively, an entity programming model might allow the identifier attribute to be identified via some convention.

Typically, an entity programming model specifies additional annotations which are used to make the schema of the entity explicit, for example, `jakarta.persistence.Id` and `jakarta.persistence.Column`, or `jakarta.nosql.Id` and `jakarta.nosql.Column`. The nature of such annotations is beyond the scope of this specification.

In a given entity programming model, entity classes might always be mutable, or might always be immutable, or, alternatively, the model might support a mix of mutable and immutable entity classes.

* A programming model which supports immutable entity classes may require that every mutable entity class declare a constructor with no parameters, and might place limits on the visibility of this constructor.
* A programming model which supports the use of immutable entity classes—ideally represented as Java `record` types—would not typically require the existence of such a constructor.

In either case, an entity programming model might place restrictions on the visibility of fields and property accessors of an entity class.

3.1.1. Entity inheritance

An entity programming model might support inheritance between entities. Two entities are related by inheritance if:

1. the entity classes are related by Java language inheritance, and
2. the Jakarta Data provider supports retrieving and querying the entities in a polymorphic fashion.

It’s possible for two entity classes to be related by Java inheritance, but *not* by entity inheritance in the sense defined here. An entity programming model specifies which Java inheritance relationships are interpreted as entity inheritance.

When entities are related by inheritance, a query method which returns the entity supertype might also return instances of its subtypes.

The Jakarta Data provider determines how entity classes which participate in an entity inheritance hierarchy "map" to the data schema. For example, in a relational datastore, all entities in the hierarchy might be stored together on one table, or each entity might have its own dedicated table.

Support for entity inheritance is not required by this specification.

3.1.2. Persistent attributes

An attribute of an entity class may or may not represent state which is persistent in the datastore. A *persistent attribute* has some corresponding representation in the data schema of the entity, for example, it might map to a column or columns in a relational database table. Any programming model for entity classes must provide well-defined rules for distinguishing attributes which are persistent in the datastore from attributes which are *transient*, having no persistent representation in the datastore. Furthermore, the programming model must specify how the Jakarta Data provider accesses the persistent attributes of an entity to read and write their values.

Every programming model for entity classes must support *direct field access*, that is, access to the persistent fields of an entity class without triggering any intermediating user-written code such as JavaBeans-style property accessors. When direct field access is used, every Java field marked with the Java language `transient` modifier must be treated as transient. A programming model might place constraints on the visibility of persistent fields. For example, Jakarta Persistence disallows `public` persistent fields. Every programming model must permit `private` persistent fields.

A programming model for entity classes might also support *property-based access*, that is, access to persistent attributes via JavaBeans-style property accessors, or, especially for Java `record` types, via accessor methods combined with constructor-based initialization. Such programming models should provide an annotation or other convention to distinguish transient properties. For example, Jakarta Persistence provides `jakarta.persistence.Transient`. When property-based access is supported, a programming model might place constraints on the visibility of property accessors. For example, Jakarta Persistence requires that property accessors be `public` or `protected`. Support for property-based access is not required by this specification.

Jakarta Data distinguishes three kinds of persistent attributes within entity classes.

- A *basic attribute* holds a value belonging to some fundamental data type supported natively by the Jakarta Data Provider. Support for the set of basic types enumerated in the next section below is mandatory for all Jakarta Data providers.
- An *embedded attribute* allows the inclusion of the state of a finer-grained Java class within the state of an entity. The type of an embedded attribute is often a user-written Java class. Support for embedded attributes varies depending on the Jakarta Data provider and the database type.
- An *association attribute* implements an association between entity types. Support for association attributes varies depending on the Jakarta Data provider and the database type.

3.1.3. Basic types

Every Jakarta Data provider must support the following basic types within its programming model:

Basic Data Type	**Description**
Primitive types and wrapper classes	All Java primitive types, such as `int`, `double`, `boolean`, etc., and their corresponding wrapper types from `java.lang` (e.g., `Integer`, `Double`, `Boolean`).
`java.lang.String`	Represents text data.
`LocalDate`, `LocalDateTime`, `LocalTime`, `Year`, `Instant` from `java.time`	Represent date and time-related data.
`java.util.UUID`	Universally Unique IDentifier for identifying entities.

Basic Data Type	Description
`BigInteger` and `BigDecimal` from `java.math`	Represent large integer and decimal numbers.
`byte[]`	Represents binary data.
User-defined `enum` types	Custom enumerated types defined by user-written code.

In this specification, "string" means `java.lang.String`, "numeric" means any primitive numeric type, wrapper for a primitive numeric type, `BigInteger`, or `BigDecimal`, and "date/time" means `LocalDate`, `LocalDateTime`, `LocalTime`, or `Instant`.

For example, the following entity class has five basic attributes:

```
@Entity
public class Person {
    @Id
    private UUID id;
    private String name;
    private long ssn;
    private LocalDate birthdate;
    private byte[] photo;
}
```

In addition to the types listed above, an entity programming model might support additional domain-specific basic types. This extended set of basic types might include types with a nontrivial internal structure. An entity programming model might even provide mechanisms to convert between user-written types and natively-supported basic types. For example, Jakarta Persistence defines the `AttributeConverter` interface.

Many key-value, wide-column, document, and relational databases feature native support for arrays or even associative arrays of these basic types. Unfortunately, the semantics of such types—along with their performance characteristics—are extremely nonuniform, and so support for such types is left undefined by the Jakarta Data specification.

3.1.4. Embedded attributes and embeddable classes

An *embeddable class* differs from an entity class in that:

- the embeddable class lacks its own persistent identity, and
- the state of an instance of the embeddable class can only be stored in the database when the instance is referenced directly or indirectly by a "parent" entity class instance.

An *embedded attribute* is an attribute whose type is an embeddable class.

Like entities, embeddable classes may have basic attributes, embeddable attributes, and association attributes, but, unlike entities, they do not have identifier attributes.

Like entities, a programming model for entity classes might support mutable embeddable classes, immutable embeddable classes, or both.

A programming model for entity classes might define an annotation that identifies a user-written class as an embeddable class. For example, Jakarta Persistence defines the annotation `jakarta.persistence.Embeddabe`. Alternatively, the programming model might define an annotation that identifies an attribute as an embedded attribute. For example, Jakarta Persistence defines the annotation `jakarta.persistence.Embedded`.

There are two natural ways that a Jakarta Data provider might store the state of an instance of an embedded class in a database:

- by *flattening* the attributes of the embeddable class into the data structure representing the parent entity, or
- by *grouping* the attributes of the embedded class into a fine-grained structured type (a UDT, for example).

In a flattened representation of an embedded attribute, the attributes of the embeddable class occur directly alongside the basic attributes of the entity class in the data schema of the entity. There is no representation of the embeddable class itself in the data schema.

For example, consider the following Java classes:

```java
@Embeddable
public class Address {
    private String street;
    private String city;
    private String postalCode;
}

@Entity
public class Person {
    @Id
    private Long id;
    private String name;
    private Address address;  // embedded attribute
}
```

In a document, wide-column, or graph database, the JSON representation of an instance of the `Person` entity might be as follows:

```json
{
  "id": 1,
  "name": "John Doe",
  "street": "123 Main St",
  "city": "Sampleville",
  "postalCode": "12345"
}
```

Or, in a relational database, the DDL for the `Person` table might look like this:

```sql
create table Person (
    id bigint primary key,
    name varchar,
    street varchar,
    city varchar,
    postalCode varchar
)
```

In a structured representation, the attributes of the embeddable class are somehow grouped together in the data schema.

For example, the JSON representation of `Person` might be:

```json
{
  "id": 1,
  "name": "John Doe",
  "address":
  {
    "street": "123 Main St",
    "city": "Sampleville",
    "postalCode": "12345"
    }
}
```

Or the SQL DDL could be:

```
create type Address as (
    street varchar,
    city varchar,
    postalCode varchar
)

create table Person (
    id bigint primary key,
    name varchar,
    address Address
)
```

Support for embeddable classes and embedded attributes is not required by this specification. However, every Jakarta Data provider is strongly encouraged to provide support for embeddable classes within its entity programming model.

3.1.5. Entity associations

An association attribute is an attribute of an entity class whose declared type is also an entity class. Given an instance of the first entity class, its association attribute holds a reference to an instance of a second entity class.

For example, consider the following Java classes:

```
@Entity
public class Author {
    @Id
    private UUID id;
    private String name;
    private List<Book> books;
}

@Entity
public class Book {
    @Id
    private Long id;
    private String title;
    private String category;
    private List<Author> authors;
}
```

In a relational database, these entities might map to the following data schema:

```
create table Author (
    uuid id primary key,
    name varchar,
)

create table BookAuthor(
    book bigint,
    author uuid,
    primary key (book, author),
    foreign key (author) references Author,
    foreign key (book) references Book
)

create table Book (
    id bigint primary key,
    title varchar,
    category varchar
)
```

Support for entity associations is not required by this specification.

3.2. Entity names and persistent attribute names

Entities and their persistent attributes may be referenced by name in the query language defined in Jakarta Data Query Language.

3.2.1. Entity names

Each entity must be assigned an *entity name* by the provider. By default, this must be the unqualified Java class name of the entity class. A programming model for entity classes might provide a way to explicitly specify an entity name. For example, Jakarta Persistence allows the entity name to be specified via the `name` member of the `@Entity` annotation.

3.2.2. Persistent attribute names

Each persistent attribute of an entity, as defined above in Persistent attributes, or of an embeddable class, as defined in Embedded attributes and embeddable classes, must be assigned a name, allowing the persistent attribute to be referenced by an automatic query method, a Query by Method Name, or from a query specified within the `@Query` annotation.

- when direct field access is used, the name of a persistent attribute is simply the name of the Java field, but
- when property-based access is used, the name of the attribute is derived from the accessor methods.

Any programming model for entity classes which supports property-based access must also define a rule for assigning names to persistent attributes. Typically, a property with accessors named `getX` and `setX` is assigned a persistent attribute name obtained by calling `java.beans.Introspector.decapitalize("X")`.

Within a given entity class or embeddable class, names assigned to persistent attributes must be unique ignoring case. A Jakarta Data provider is permitted to reject an entity class if two persistent attributes would be assigned the same name.

Furthermore, within the context of a given entity, each persistent attribute of an embeddable class reachable by navigation from the entity class may be assigned a compound name. The compound name is obtained by concatenating the names assigned to each attribute traversed by navigation from the entity class to the persistent attribute of the embedded class, optionally joined by a delimiter.

The rule for concatenating compound names depends on the context, and is specified in Attribute Name Concatenation and Delimiters. The examples in the table assume an `Order` entity has an `address` of type `MailingAddress` with a `zipCode` of type `int`.

Table 1. Attribute Name Concatenation and Delimiters

Context	Type	Delimiter	Example
`@Find`	Parameter name	`_`	`@Find List<Person> find(int address_zipCode);`

Context	Type	Delimiter	Example
`@Query`	Path expression within query	.	`@Query("FROM Person WHERE address.zipCode = ?1")`
Query by Method Name	Method name	_	`List<Person> findByAddress_zipCode(int zip);`
`Sort`	String argument	. or _	`Sort.asc("address_zipCode")`
`@By` or `@OrderBy`	Annotation value	. or _	`@Find List<Person> find(@By("address.zipCode") int zip);`

Application programmers are strongly encouraged to follow Java's camel case naming standard for attributes of entities, relations, and embeddable classes, avoiding underscores in attribute names. The resolution algorithm for persistent attribute identification relies on the use of underscore as a delimiter. Adhering to the camel case naming convention ensures consistency and eliminates ambiguity.

3.3. Type-safe access to entity attributes

Jakarta Data provides a static metamodel that allows entity attributes to be accessed by applications in a type-safe manner.

For each entity class, the application developer or a compile-time annotation processor can define a corresponding metamodel class following a prescribed set of conventions.

- The metamodel class can be an interface or concrete class.
- The metamodel class must be annotated with `@StaticMetamodel`, specifying the entity class as its `value`.
- The metamodel class contains one or more `public static` fields corresponding to persistent attributes of the entity class.
- Each field must be either of type `java.lang.String` (used primarily for annotation-based configurations) or `jakarta.data.metamodel.Attribute`, including any of its subinterfaces.

The application can use the field values of the metamodel class to obtain artifacts relating to the entity attribute in a type-safe manner, for example, `_Book.title.asc()` or `Sort.asc(_Book.title.name())` or `Sort.asc(_Book.TITLE)` rather than `Sort.asc("title")`.

3.3.1. Application requirements for a metamodel class

When an application programmer writes a static metamodel class for an entity by hand:

- each field corresponding to a persistent attribute of an entity must have modifiers `public`, `static`, and `final` (these are implicit when the metamodel class is an interface), and
- the fields must be statically initialized.

The static metamodel class is not required to include a field for every persistent attribute of the entity.

A convenience implementation of each subinterface of `Attribute` is provided by the `.of` method of each subinterface.

3.3.2. Compile-time annotation processor requirements for a metamodel class

When an annotation processor generates a static metamodel class for an entity:

- the metamodel class must be annotated with `jakarta.annotation.Generated`,
- each field corresponding to a persistent attribute of an entity must have modifiers `public`, `static`, and either `final` or `volatile`,
- the name of each field, ignoring case, must match the name of an entity attribute, according to the conventions specified below in Conventions for metamodel fields, and with the `_` character in the field name delimiting the attribute names of hierarchical structures or relationships, such as embedded classes.

The fields may be statically initialized, or they may be initialized by the provider during system initialization.

3.3.3. Conventions for metamodel fields

The following are conventions for static metamodel classes:

- The name of the static metamodel class should consist of underscore (_) followed by the entity class name.
- Fields of type `String` should be named with all upper case.
- Fields of type `Attribute` (or a subinterface of `Attribute`) should be named in lower case or mixed case.
- Uninitialized fields should have modifiers `public`, `static`, and `volatile`.
- Initialized fields must have modifiers `public`, `static`, and `final`.
- Fields of type `String` must always be statically initialized, enabling their use in annotation values.

The following subinterfaces of `Attribute` are recommended to obtain the full benefit of the static metamodel:

- `TextAttribute` for entity attributes that represent text, typically of type `String`.
- `NumericAttribute` for entity attributes of numeric types, such as `long`, `Integer`, and `BigDecimal`.
- `TemporalAttribute` for entity attributes of temporal types, such as `LocalDateTime`, and `Instant`.
- `ComparableAttribute` for entity attributes that represent other comparable values, such as `boolean` and enumerations.
- `NavigableAttribute` for entity attributes that are embeddables or associations.
- `BasicAttribute` for other types of entity attributes, such as collections.

To assist developers in selecting the appropriate subinterface of `Attribute` when declaring static metamodel fields, the following table maps common Java types to the corresponding attribute interface. While multiple Java types may be compatible with the same subinterface, this guidance ensures consistent and type-safe usage across different entity models.

Java Type(s)	Attribute Interface
`String`	`TextAttribute`
Primitive numeric types (`byte`, `int`, `double`, etc.), their corresponding wrapper classes from `java.lang` (e.g., `Byte`, `Integer`, `Double`), and arbitrary-precision types like `BigInteger` and `BigDecimal` from `java.math`	`NumericAttribute`
`LocalDate`, `LocalDateTime`, `LocalTime`, `Instant`, `Year`	`TemporalAttribute`

Java Type(s)	**Attribute Interface**
Any `enum`, `boolean`, `UUID`, custom `Comparable`	`ComparableAttribute`
Embedded objects, associated entities	`NavigableAttribute`
`byte[]`	`SortableAttribute` for databases that support sorting `byte[]` values, otherwise `BasicAttribute`
`Optional<T>`	**(not supported)**
Other types not explicitly listed above	`BasicAttribute`

When modeling attributes backed by Java primitive types (e.g., `int`, `long`), the type parameter `V` in `BasicAttribute<T, V>` must always be the boxed type (`Integer`, `Long`, etc.) due to Java's generic type constraints. However, the third parameter passed to `BasicAttribute.of(…)` — the attribute's `Class` — must reference the primitive type. For example:

```
BasicAttribute<Book, Integer> pages = BasicAttribute.of(Book.class, "pages", int.class);
```

This pattern ensures both compile-time correctness and alignment with Jakarta Data's internal type handling. While slightly asymmetrical, this approach is consistent with Java's type system and allows Jakarta Data providers to correctly infer and handle attribute types.

3.3.4. Example metamodel class and usage

Example entity class:

```
@Entity
public class Product {
  public long id;
  public String name;
  public float price;
}
```

Example metamodel class for the entity:

```
@StaticMetamodel(Product.class)
public interface _Product {
  String ID = "id";
  String NAME = "name";
  String PRICE = "price";

  NumericAttribute<Product,Long> id = NumericAttribute.of(
          Product.class, ID, long.class);
  TextAttribute<Product> name = TextAttribute.of(
          Product.class, NAME);
  NumericAttribute<Product,Float> price = NumericAttribute.of(
          Product.class, PRICE, float.class);
}
```

Example usage:

```
List<Product> found = products.findAll(
    Restrict.all(
        _Product.name.contains(searchPattern),
        _Product.price.greaterThanEqual(minPrice),
        _Product.price.lessThanEqual(maxPrice)
    ),
    Order.by(_Product.price.desc(),
             _Product.name.asc(),
             _Product.id.asc())
);
```

[1] We will not consider generic programs which work with entity data via detyped representations.

[2] Jakarta Persistence 3.2, https://jakarta.ee/specifications/persistence/3.2/

[3] Jakarta NoSQL 1.0, https://jakarta.ee/specifications/nosql/1.0/

Chapter 4. Repository interfaces

A Jakarta Data repository is a Java interface annotated with `@Repository`. A repository interface may declare:

- *abstract* (non-`default`) methods, and
- *concrete* (`default`) methods.

A concrete method may call other methods of the repository, including abstract methods.

Every abstract method of the interface is usually either:

- an entity instance *lifecycle method*,
- an *annotated query method*,
- an *automatic query method* with parameter-based conditions or defined using the Query by Method Name extension, or
- a *resource accessor method*.

A repository may declare lifecycle methods for a single entity type, or for multiple related entity types. Similarly, a repository might have query methods which return different entity types.

A repository interface may inherit methods from a superinterface. A superinterface of a repository interface must either:

- be one of the built-in generic repository supertypes defined by this specification, `DataRepository`, `BasicRepository`, or `CrudRepository`, or
- be a non-generic toplevel interface with no type parameters, whose abstract methods likewise declare no type parameters, and which does not itself directly or indirectly inherit any generic interface or any interface whose abstract methods declare type parameters.

A Jakarta Data implementation must treat abstract methods inherited by a repository interface as if they were directly declared by the repository interface.

Repositories perform operations on entities. For repository methods that are annotated with `@Insert`, `@Update`, `@Save`, or `@Delete`, the entity type is determined from the method parameter type. For repository methods that are annotated with `@Find`, the entity type is determined by the annotation `value` member, if an entity type is explicitly specified. Otherwise, for `find` and `delete` methods where the return type is an entity, array of entity, or parameterized type such as `List<MyEntity>` or `Page<MyEntity>`, the entity type is determined from the method return type. For `count`, `exists`, and other `find` and `delete` methods that do not return the entity or accept the entity as a parameter, the entity type cannot be determined from the method signature and a *primary entity type* must be defined for the repository.

Users of Jakarta Data declare a primary entity type for a repository by inheriting from a built-in repository super interface, such as `BasicRepository`, and specifying the primary entity type as the first type variable. For repositories that do not inherit from a super interface with a type parameter to indicate the primary entity type, lifecycle methods on the repository determine the primary entity type. To do so, all lifecycle methods where the method parameter is a type, an array of type, or is parameterized with a type that is annotated as an entity, must correspond to the same entity type. The primary entity type is assumed for methods that do not otherwise specify an entity type, such as `countByPriceLessThan`. Methods that require a primary entity type raise `MappingException` if a primary entity type is not provided.

A Jakarta Data provider might go beyond what is required by this specification and support abstract methods which do not fall into any of the above categories. Such functionality is not defined by this specification, and so applications with repositories which declare such methods are not portable between providers.

The subsections below specify the rules that an abstract method declaration must observe so that the Jakarta Data implementation is able to provide an implementation of the abstract method.

- If every abstract method of a repository complies with the rules specified below, then the Jakarta Data implementation must provide an implementation of the repository.
- Otherwise, if a repository declares an abstract method which does not comply with the rules specified below, or makes use of functionality which is not supported by the Jakarta Data implementation, then an error might be produced by the Jakarta Data implementation at build time or at runtime.

The portability of a given repository interface between Jakarta Data implementations depends on the portability of the entity types it uses. If an entity class is not portable between given implementations, then any repository which uses the entity class is also unportable between those implementations.

Additional portability guarantees may be provided by specifications which extend this specification, specializing to a given class of datastore.

4.1. Lifecycle methods

A *lifecycle method* is an abstract method annotated with a *lifecycle annotation*. Lifecycle methods allow the program to make changes to persistent data in the data store.

A lifecycle method must be annotated with a lifecycle annotation. The method signature of the lifecycle method, including its return type, must follow the requirements that are specified by the Javadoc of the lifecycle annotation.

Lifecycle method signatures follow one of these generic patterns:

```
@Lifecycle
void lifecycle(Entity e);
```

```
@Lifecycle
Entity lifecycle(Entity e);
```

where `Lifecycle` is a lifecycle annotation, `lifecycle` is the arbitrary name of the method, and `Entity` is either `E`, `List<E>`, or `E[]`, where `E` is a concrete entity class. In this context, any variadic parameter declared `E…` is treated as if it were declared with type `E[]`.

This specification defines two sets of built-in lifecycle annotations:

- `@Insert`, `@Update`, `@Delete`, and `@Save` are provided for use with regular stateless repositories, and
- as an extension, `@Persist`, `@Merge`, `@Refresh`, `@Remove`, and `@Detach` are provided for use with stateful repositories.

The semantics of a lifecycle method annotated with one of these annotations are defined by the Javadoc of the annotation.

For example:

```
@Insert
void insertBook(Book book);
```

Lifecycle methods are not guaranteed to be portable between all providers.

Jakarta Data providers must support lifecycle methods to the extent that the data store is capable of the corresponding operation. If the data store is not capable of the operation, the Jakarta Data provider must raise `UnsupportedOperationException` when the operation is attempted, per the requirements of the Javadoc for the lifecycle annotation, or the Jakarta Data provider must report the error at compile time.

There is no special programming model for lifecycle annotations. The Jakarta Data implementation automatically recognizes the lifecycle annotations it supports.

> A Jakarta Data provider might extend this specification to define additional lifecycle annotations, or to support lifecycle methods with signatures other than the usual signatures defined above. For example, a provider might support "merge" methods declared as follows:
>
> ```
> @Merge
> Book mergeBook(Book book);
> ```
>
> Such lifecycle methods are not portable between Jakarta Data providers.

4.2. Annotated query methods

An *annotated query method* is an abstract method annotated by a *query annotation* type. The query annotation specifies a query in some datastore-native query language.

Each parameter of an annotated query method must either:

- have exactly the same name and type as a named parameter of the query,
- have exactly the same type and position within the parameter list of the method as a positional parameter of the query, or
- be of type `Limit`, `Order`, `PageRequest`, or `Sort`.

A repository with annotated query methods with named parameters must be compiled so that parameter names are preserved in the class file (for example, using `javac -parameters`), or the parameter names must be specified explicitly using the `@Param` annotation.

An annotated query method must not also be annotated with a lifecycle annotation.

The return type of the annotated query method must be consistent with the result type of the query specified by the query annotation.

> The result type of a query depends on datastore-native semantics, and so the return type of an annotated query method cannot be specified here. However, Jakarta Data implementations are strongly encouraged to support the following return types:
>
> - for a query which returns a single result of type `T`, the type `T` itself, or `Optional<T>`,
> - for a query which returns many results of type `T`, the types `List<T>`, `Page<T>`, and `T[]`.
>
> Furthermore, implementations are encouraged to support `void` as the return type for a query which never returns a result.

This specification defines the built-in `@Query` annotation, which may be used to specify a query written in the Jakarta Data Query Language defined in the next chapter.

For example, using a named parameter:

```
@Query("where title like :title order by title asc, id asc")
Page<Book> booksByTitle(String title, PageRequest pageRequest);
```

```
@Query("where p.name = :prodname")
```

```
Optional<Product> findByName(@Param("prodname") String name);
```

Or, using a positional parameter:

```
@Query("delete from Book where isbn = ?1")
void deleteBook(String isbn);
```

Programs which make use of annotated query methods are not in general portable between providers. However, when the `@Query` annotation specifies a query written in JDQL, the annotated query method is portable between providers to the extent to which its semantics can be implemented on the underlying data store.

A Jakarta Data provider might extend this specification to define its own query annotation types. For example, a provider might define a `@SQL` annotation for declaring queries written in SQL.

There is no special programming model for query annotations. The Jakarta Data implementation automatically recognizes the query annotations it supports.

4.3. Parameter-based automatic query methods

A *parameter-based automatic query method* is an abstract method annotated with an *automatic query annotation*.

Each automatic query method must be assigned an entity type. The rules for inferring the entity type depend on the semantics of the automatic query annotation. Typically:

- If the automatic query method returns an entity type, the method return type identifies the entity. For example, the return type might be `E`, `Optional<E>`, `E[]`, `Page<E>`, or `List<E>`, where `E` is an entity class. Then the automatic query method would be assigned the entity type `E`.
- If the query does not return an entity type, the entity assigned to the automatic query method is the primary entity type of the repository.

Jakarta Data infers a query based on the parameters of the method. Each parameter must either:

- have exactly the same type and name as a persistent attribute of the entity class, or
- be of type `Limit`, `Order`, `PageRequest`, or `Sort`.

Parameter names map parameters to persistent attributes. A repository with parameter-based automatic query methods must either:

- be compiled so that parameter names are preserved in the class file (for example, using `javac -parameters`), or
- explicitly specify the name of the persistent attribute mapped by each parameter of an automatic query method using the `@By` annotation.

The attribute name specified using `@By` may be a compound name, as specified below in Persistent attribute names.

This specification defines the built-in automatic query annotations `@Find` and `@Delete`. The semantics of these annotations are specified in their Javadoc. Note that `@Delete` is *both* a lifecycle annotation *and* an automatic query annotation. The signature of a repository method annotated `@Delete` must be used to disambiguate the interpretation of the `@Delete` annotation.

For example:

```
@Find
Book bookByIsbn(String isbn);

@Find
```

```
List<Book> booksByYear(Year year, Sort<Book> order, Limit limit);

@Find
Page<Book> find(@By("year") Year publishedIn,
                @By("genre") Category type,
                Order<Book> sortBy,
                PageRequest pageRequest);
```

Automatic query methods annotated with `@Find` or `@Delete` *are* portable between providers.

> A Jakarta Data provider might extend this specification to define its own automatic query annotation types. In this case, an automatic query method is *not* portable between providers.

4.4. Resource accessor methods

A *resource accessor method* is a method with no parameters which returns a type supported by the Jakarta Data provider. The purpose of this method is to provide the program with direct access to the data store.

For example, if the Jakarta Data provider is based on JDBC, the return type might be `java.sql.Connection` or `javax.sql.DataSource`. Or, if the Jakarta Data provider is backed by Jakarta Persistence, the return type might be `jakarta.persistence.EntityManager`.

The Jakarta Data provider recognizes the connection types it supports and implements the method such that it returns an instance of the type of resource. If the resource type implements `java.lang.AutoCloseable` and the resource is obtained within the scope of a default method of the repository, then the Jakarta Data provider automatically closes the resource upon completion of the default method. If the method for obtaining the resource is invoked outside the scope of a default method of the repository, then the user is responsible for closing the resource instance.

> A Jakarta Data implementation might allow a resource accessor method to be annotated with additional metadata providing information about the connection.

For example:

```
Connection connection();

default void cleanup() {
    try (Statement s = connection().createStatement()) {
        s.executeUpdate("truncate table books");
    }
}
```

A repository may have at most one resource accessor method.

A resource accessor method with return type `jakarta.persistence.EntityManager` should only be declared by a stateful repository.

4.5. Conflicting repository method annotations

Annotations like `@Find`, `@Query`, `@Insert`, `@Update`, `@Delete`, and `@Save` are mutually-exclusive. A given method of a repository interface may have at most one:

- `@Find` annotation,
- lifecycle annotation, or
- query annotation.

If a method of a repository interface has more than one such annotation, the annotated repository method must raise `UnsupportedOperationException` every time it is called. Alternatively, a Jakarta Data provider is permitted to reject such a method declaration at compile time.

4.6. Special parameters for limits, sorting, and pagination

An annotated, parameter-based, or Query by Method Name query method may have *special parameters* of type `Limit`, `Order`, `Sort`, or `PageRequest` if the method return type indicates that the method may return multiple entities, that is, if the return type is:

- an array type,
- `List` or `Stream`, or
- `Page` or `CursoredPage`.

A special parameter controls which query results are returned to the caller of a repository method, or in what order the results are returned:

- a `Limit` allows the query results to be limited to a given range defined in terms of an offset and maximum number of results,
- a `Sort` or `Order` allows the query results to be sorted by a given entity attribute or list of attributes, respectively, and
- a `PageRequest` splits results into pages. A parameter of this type must be declared when the repository method returns a `Page` of results, as specified below in Offset-based pagination, or a `CursoredPage`, as specified in Cursor-based pagination.

A repository method throws `NullPointerException` if an argument to a special parameter of the method is null.

A repository method must throw `UnsupportedOperationException` if it has:

- more than one parameter of type `PageRequest` or `Limit`,
- a parameter of type `PageRequest` and a parameter of type `Limit`,
- a `@First` annotation and a parameter of type `PageRequest` or `Limit`,
- a parameter of type `PageRequest` or `Limit`, in combination with the keyword `First`,
- a `@First` annotation, in combination with the keyword `First`,
- more than one parameter of type `Order`, or
- more than one parameter of type `Restriction`.

Alternatively, a Jakarta Data provider is permitted to reject such a repository method declaration at compile time.

A repository method must throw `DataException` if the database is incapable of ordering the query results using the given sort criteria.

The following example demonstrates the use of special parameters:

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

    @Find
    Page<Product> findByName(String name, PageRequest pageRequest, Order<Product> order);

    @Query("where name like :pattern")
    List<Product> findByNameLike(String pattern, Limit max, Sort<?>... sorts);

}
```

An instance of `Sort` may be obtained by specifying an entity attribute name:

```
Sort nameAscending = Sort.asc("name");
```

Even better, the static metamodel may be used to obtain an instance of `Sort` in a typesafe way:

```
Sort<Employee> nameAscending = _Employee.name.asc();
```

This `PageRequest` specifies a starting page and maximum page size:

```
PageRequest pageRequest = PageRequest.ofPage(1).size(20);
List<Product> first20 = products.findByName(name, pageRequest,
                            Order.by(_Product.price.desc(),
                                     _Product.id.asc()));
```

4.7. Precedence of sort criteria

The specification defines different ways of providing sort criteria on queries. This section discusses how these different mechanisms relate to each other.

4.7.1. Sort criteria within query language

Sort criteria can be hard-coded directly within query language by making use of the `@Query` annotation. A repository method that is annotated with `@Query` with a value that contains an `ORDER BY` clause (or query language equivalent) must not provide sort criteria via the other mechanisms.

A repository method that is annotated with `@Query` with a value that does not contain an `ORDER BY` clause and ends with a `WHERE` clause (or query language equivalents to these) can use other mechanisms that are defined by this specification for providing sort criteria.

4.7.2. Static mechanisms for sort criteria

Sort criteria are provided statically for a repository method by using the `OrderBy` keyword or by annotating the method with one or more `@OrderBy` annotations. The `OrderBy` keyword cannot be intermixed with the `@OrderBy` annotation or the `@Query` annotation. Static sort criteria takes precedence over dynamic sort criteria in that static sort criteria are evaluated first. When static sort criteria sorts entities to the same position, dynamic sort criteria are applied to further order those entities.

4.7.3. Dynamic mechanisms for sort criteria

Sort criteria are provided dynamically to repository methods either via `Sort` parameters or via a `Order` parameter that has one or more `Sort` values.

4.7.4. Examples of sort criteria precedence

In the following examples, the query results are sorted by `age`, using the dynamic sorting criteria passed to the `sorts` parameter to break ties between records with the same `age`.

```
@Query("WHERE u.age > ?1")
@OrderBy(_User.AGE)
Page<User> findByNamePrefix(String namePrefix,
                            PageRequest pagination,
                            Order<User> sorts);
```

```
@Query("WHERE u.age > ?1")
```

```
    @OrderBy(_User.AGE)
    List<User> findByNamePrefix(String namePrefix, Sort<?>... sorts);
```

4.8. Restrictions

A method annotated `@Find`, `@Delete`, or `@Query` may have a special parameter of type `Restriction<T>`. This parameter allows the caller of a repository method to programmatically express additional restrictions on the results returned by the repository method, often using a generated metamodel class for type safety.

For example:

```
@Repository
public interface ProductRepository {
    @Find
    List<Product> search(Restriction<Product> restriction);
}


List<Product> pencils = products.search(_Product.description.like("%pencil%"));
```

Programmatic restrictions passed to a parameter of type `Restriction` combine conjunctively with static restrictions:

- expressed via parameters of a method annotated `@Find`, or
- given in the where clause of a query specified by a `@Query` annotation.

For example:

```
@Repository
public interface ProductRepository {
   @Find
   List<Product> search(@By(_Product.DESCRIPTION) @Is(Like.class) description,
                        Restriction<Product> restriction);
}


List<Product> redPencils =
        products.search("%pencil%",
                        _Product.color.equalTo(Color.RED));

List<Product> expensivePencils =
        products.search("%pencil%",
                        _Product.price.greaterThan(BigDecimal.TEN));
```

The static methods `Restrict.all(…)` or `Restrict.any(…)` allow restrictions to be combined:

```
List<String> expensiveRedPencils = products.search(
    "%pencil%",
    Restrict.all(
        _Product.color.equalTo(Color.RED),
        _Product.price.greaterThan(BigDecimal.TEN)
    )
);
```

Instances of `Restriction<T>` are immutable and may be reused across multiple repository calls.

4.9. Pagination in Jakarta Data

Dividing up large sets of data into pages is a beneficial strategy for data access and retrieval in many applications, including those developed in Java. Pagination helps improve the efficiency of handling large datasets in a way that is also user-friendly. In Jakarta Data, APIs are provided to help Java developers efficiently manage and navigate through data.

Jakarta Data supports two types of pagination: offset-based and cursor-based. These approaches differ in how they manage and retrieve paginated data:

Offset pagination is the more traditional form based on position relative to the first record in the dataset. It is typically used with a fixed page size, where a specified number of records is retrieved starting from a given offset position.

Cursor-based pagination, also known as seek method or keyset pagination, uses a unique key or unique combination of values (referred to as the key) to navigate the dataset relative to the first or last record of the current page. Cursor-based pagination is typically used with fixed page sizes but can accommodate varying the page size if desired. It is more robust when dealing with datasets where the underlying data might change and offers the the potential for improved performance by avoiding the need to scan records prior to the cursor.

The critical differences between offset-based and cursor-based pagination lie in their retrieval methods:

- Offset-based pagination uses a fixed page size and retrieves data based on page number and size.
- Cursor-based pagination relies on a unique key or unique combination of values (the key) for an entity relative to which it determines the next page or previous page.

4.9.1. Offset-based pagination

Offset pagination is a popular method for managing and retrieving large datasets efficiently. It is based on dividing the dataset into pages containing a specified number of elements. This method allows developers to retrieve a subset of the dataset by identifying the page number and the maximum number of elements per page.

Offset pagination is motivated by the need to provide efficient navigation through large datasets. Loading an entire dataset into memory at once can be resource-intensive and lead to performance issues. By breaking the dataset into smaller, manageable pages, offset pagination improves performance, reduces resource consumption, and enhances the overall user experience.

Offset pagination offers several key features that make it a valuable approach for managing and retrieving large datasets in a controlled and efficient manner:

- *Page size:* The maximum number of elements to be included in each page is known as the page size. This parameter determines the subset of data retrieved with each pagination request.
- *Page number:* The page number indicates which subset of the dataset to retrieve. It typically starts from 1, representing the first page, and increments with each subsequent page.
- *Efficient navigation:* Offset pagination allows efficient dataset navigation. By specifying the desired page and page size, developers can control the data retrieved, optimizing memory usage and processing time.
- *Sequential order:* Elements are retrieved sequentially based on predefined criteria, such as ascending or descending order of a specific attribute, like an ID.

4.9.1.1. Requirements when using offset pagination

The following requirements must be met when using offset-based pagination:

- The repository method signature must return `Page`. A repository method with return type of `Page` must raise `UnsupportedOperationException` if the database is incapable of offset pagination.
- The repository method signature must accept a `PageRequest` parameter.
- Sort criteria must be provided and should be minimal.
- The combination of provided sort criteria must define a deterministic ordering of entities.

- The entities within each page must be ordered according to the provided sort criteria.
- If `PageRequest.requestTotal()` returns `true`, the `Page` should contain accurate information about the total number of pages and total number of elements across all pages. Otherwise, if `PageRequest.requestTotal()` returns `false`, the operations `Page.totalElements()` and `Page.totalPages()` throw `IllegalStateException`.
- Except for the highest numbered page, the Jakarta Data provider must return full pages consisting of the maximum page size number of entities.
- Page numbers for offset pagination are computed by taking the entity's 1-based offset after sorting, dividing it by the maximum page size, and rounding up. For example, the 52nd entity is on page 6 when the maximum page size is 10, because 52 / 10 rounded up is 6. Note that the first page number is always 1.

4.9.1.2. Scenario: Person entity and People repository

Consider a scenario with a `Person` entity and a corresponding `People` repository:

```
public class Person {
    private Long id;
    private String name;
}

@Repository
public interface People extends BasicRepository<Person, Long> {
}
```

The dataset contains the following elements:

```
[
   {"id":1, "name":"Lin Le Marchant"},
   {"id":2, "name":"Corri Davidou"},
   {"id":3, "name":"Alyse Dadson"},
   {"id":4, "name":"Orelle Roughey"},
   {"id":5, "name":"Jaquith Wealthall"},
   {"id":6, "name":"Boothe Martinson"},
   {"id":7, "name":"Patten Bedell"},
   {"id":8, "name":"Danita Pilipyak"},
   {"id":9, "name":"Harlene Branigan"},
   {"id":10, "name":"Boothe Martinson"}
]
```

Code Execution:

```
@Inject
People people;

Page<Person> page =
        people.findAll(PageRequest.ofPage(1).size(2),
                       Order.by(Sort.asc("id")));
```

Resulting Page Content:

```
[
   {"id":1, "name":"Lin Le Marchant"},
   {"id":2, "name":"Corri Davidou"}
]
```

Next Page Execution:

```
if (page.hasNext()) {
   PageRequest nextPageRequest = page.nextPageRequest();
   Page<Person> page2 = people.findAll(nextPageRequest,
                                       Order.by(Sort.asc("id")));
}
```

Resulting Page Content:

```
[
   {"id":3, "name":"Alyse Dadson"},
   {"id":4, "name":"Orelle Roughey"}
]
```

In this scenario, each page represents a subset of the dataset, and developers can navigate through the pages efficiently using offset pagination.

Offset pagination is a valuable tool for Java developers when dealing with large datasets, providing control, efficiency, and a seamless user experience.

4.9.2. Cursor-based pagination

Cursor-based pagination aims to reduce missed and duplicate results across pages by querying relative to the observed values of entity attributes that constitute the sorting criteria. Cursor-based pagination can also offer an improvement in performance because it avoids fetching and ordering results from prior pages by causing those results to be non-matching. A Jakarta Data provider appends additional conditions to the query and tracks cursor-based values automatically when `CursoredPage` is used as the repository method return type. The application invokes `nextPageRequest` or `previousPageRequest` on the `CursoredPage` to obtain a `PageRequest` which keeps track of the cursor-based values.

For example,

```
@Repository
public interface CustomerRepository extends BasicRepository<Customer, Long> {
    @Find
    @OrderBy(_Customer.LAST_NAME)
    @OrderBy(_Customer.FIRST_NAME)
    @OrderBy(_Customer.ID)
    CursoredPage<Customer> findByZipcode(int zipcode, PageRequest pageRequest);
}
```

You can obtain the initial page relative to an offset and subsequent pages relative to the last entity of the current page as follows,

```
PageRequest pageRequest = PageRequest.ofSize(50);
Page<Customer> page =
        customers.findByZipcode(55901, pageRequest);
if (page.hasNext()) {
  pageRequest = page.nextPageRequest();
  page = customers.findByZipcode(55901, pageRequest);
  ...
}
```

Or you can obtain the next (or previous) page relative to a known entity,

```
Customer c = ...
PageRequest p = PageRequest.ofPage(10)
                           .size(50)
                           .afterCursor(Cursor.forKey(c.lastName, c.firstName, c.id));
page = customers.findByZipcode(55902, p);
```

The sort criteria for a repository method that performs cursor-based pagination must uniquely identify each entity and must be provided by:

- the `@OrderBy` annotation or annotations of the repository method,
- `Order` or `Sort` parameters of the repository method, or
- an `OrderBy` in Query by Method Name.

The values of the entity attributes of the combined sort criteria define the cursor for cursor-based cursor based pagination. Within the cursor, each entity attribute has the same sorting and order of precedence that it has within the combined sort criteria.

4.9.2.1. Example of appending to queries for cursor-based pagination

Without cursor-based pagination, a Jakarta Data provider that is based on Jakarta Persistence might compose the following JPQL for the `findByZipcode()` repository method from the prior example:

```
FROM Customer
WHERE zipCode = ?1
ORDER BY lastName ASC, firstName ASC, id ASC
```

When cursor-based pagination is used, the keys values from the `Cursor` of the `PageRequest` are available as query parameters, allowing the Jakarta Data provider to append additional query conditions. For example,

```
FROM Customer
WHERE (zipCode = ?1)
  AND (
         lastName > ?2
      OR lastName = ?2 AND firstName > ?3
      OR lastName = ?2 AND firstName = ?3 AND id > ?4
  )
ORDER BY lastName ASC, firstName ASC, id ASC
```

4.9.2.2. Avoiding missed and duplicate results

Because searching for the next page of results is relative to a last known position, it is possible with cursor-based pagination to allow some types of updates to data while pages are being traversed without causing missed results or duplicates to appear. If you add entities to a prior position in the traversal of pages, the shift forward of numerical position of existing entities will not cause duplicates entities to appear in your continued traversal of subsequent pages because cursor-based pagination does not query based on a numerical position. If you remove entities from a prior position in the traversal of pages, the shift backward of numerical position of existing entities will not cause missed entities in your continued traversal of subsequent pages because keyset pagination does not query based on a numerical position.

Other types of updates to data, however, will cause duplicate or missed results. If you modify entity attributes which are used as the sort criteria, cursor-based pagination cannot prevent the same entity from appearing again or never appearing due to the altered values. If you add an entity that you previously removed, whether with different values or the same values, cursor-based pagination cannot prevent the entity from being missed or possibly appearing a second time due to its changed values.

4.9.2.3. Restrictions on use of cursor-based pagination

- The repository method signature must return `CursoredPage`. A repository method with return type of `CursoredPage` must raise `UnsupportedOperationException` if the database is incapable of cursor-based pagination.
- The contents of the `CursoredPage` returned by the repository method must be entities, not entity attributes, records containing a subset of entity attributes, or any other values that are not the entity itself.

- The repository method signature must accept a `PageRequest` parameter.
- Sort criteria must be provided and should be minimal.
- The combination of provided sort criteria must uniquely identify each entity such that the sort criteria defines a deterministic ordering of entities.
- The entities within each page must be ordered according to the provided sort criteria.
- Page numbers for cursor-based pagination are estimated relative to prior page requests or the observed absence of further results and are not accurate. Page numbers must not be relied upon when using cursor-based pagination.
- Page totals and result totals are not accurate for cursor-based pagination and must not be relied upon.
- A next or previous page can end up being empty. You cannot obtain a next or previous `PageRequest` from an empty page because there are no key values relative to which to query.
- A repository method that is annotated with `@Query` and performs cursor-based pagination must omit the `ORDER BY` clause from the provided query and instead must supply the sort criteria via `@OrderBy` annotations or `Sort` criteria of `PageRequest`. The provided query must end with a `WHERE` clause to which additional conditions can be appended by the Jakarta Data provider. The Jakarta Data provider is not expected to parse query text that is provided by the application.

4.9.2.4. Cursor-based pagination example with sorts

Here is an example where an application uses `@Query` to provide a partial query to which the Jakarta Data provider can generate and append additional query conditions and an `ORDER BY` clause.

```
@Repository
public interface CustomerRepository extends BasicRepository<Customer, Long> {
    @Query("WHERE totalSpent / totalPurchases > ?1")
    CursoredPage<Customer> withAveragePurchaseAbove(float minimum,
                                                    PageRequest pageRequest,
                                                    Order<Customer> sorts);
}
```

Example traversal of pages:

```
Order<Customer> order =
        Order.by(_Customer.yearBorn.desc(),
                 _Customer.name.asc(),
                 _Customer.id.asc());
PageRequest pageRequest = PageRequest.ofSize(25);
do {
    page = customers.withAveragePurchaseAbove(50.0f, pageRequest, order);
    ...
    if (page.hasNext()) {
        pageRequest = page.nextPageRequest();
    }
}
while (page.hasNext());
```

4.9.2.5. Example with before/after cursor

In this example, the application uses a cursor to request pages in forward and previous direction from a specific value, which is the price for a matching product.

```
@Repository
public interface Products extends CrudRepository<Product, Long> {
    @Query("where name like ?1")
    CursoredPage<Product> findByNameLike(String namePattern,
                                         PageRequest pageRequest,
                                         Order<Product> sorts);
}
```

Obtaining the next 10 products that cost $50.00 or more:

```
float priceMidpoint = 50.0f;
Order<Product> order =
        Order.by(_Product.price.asc(),
                 _Product.id.asc());
PageRequest pageRequest =
        PageRequest.ofPage(5)
                   .size(10)
                   .afterCursor(Cursor.forKey(priceMidpoint, 0L));
CursoredPage<Product> moreExpensive =
        products.findByNameLike(pattern, pageRequest, order);
```

Obtaining the previous 10 products:

```
pageRequest =
        moreExpensive.hasContent() && moreExpensive.hasPrevious()
                ? moreExpensive.previousPageRequest()
                : pageRequest.beforeCursor(Cursor.forKey(priceMidpoint, 1L));
CursoredPage<Product> lessExpensive =
        products.findByNameLike(pattern, pageRequest, order);
```

4.9.2.6. Example with combined sort criteria

In this example, the application uses `OrderBy` to define a subset of the sort criteria during development time, but also uses `Sort` to dynamically determine more fine-grained sorting when all of the static sort criteria matches. In this case the repository query is written to always order `Car` entities with a vehicle condition of `VehicleCondition.NEW` ahead of those with `VehicleCondition.USED`.

```
@Repository
public interface Products extends CrudRepository<Product, Long> {
    @Find
    @OrderBy(_Car.VEHICLE_CONDITION)
    CursoredPage<Car> find(@By(_Car.MAKE) String manufacturer,
                           @By(_Car.MODEL) String model,
                           PageRequest pageRequest,
                           Order<Car> sorts);
}
```

The above criteria does not uniquely identify `Car` entities. After sorting on the vehicle condition, finer grained sorting is provided dynamically by the `Order`, in this case the vehicle price followed by the unique Vehicle Identification Number (VIN). It is a good practice for the final sort criterion to be a unique identifier of the entity to ensure a deterministic ordering.

```
Order<Car> order = Order.by(_Car.price.desc(),
                            _Car.vin.asc())
PageRequest page1Request = PageRequest.ofSize(25);
CursoredPage<Car> page1 =
        cars.find(make, model, page1Request, order);
```

The query results are ordered first by vehicle condition. All resulting entities with the same vehicle condition are subsequently ordered by their price in descending order. All resulting entities with the same vehicle condition and price are ordered alphabetically by their VIN. The end user requests the next page of results. If the application still has access to the page at this point, it can use `page.nextPageRequest()` to obtain a request for the next page of results. In this case, the Jakarta Data provider computes the cursor from the vehicle condition, price, and VIN of the final `Car` entity of the page and includes the cursor in the resulting `PageRequest` instance. Alternatively, the application does not need access to the page if it obtained the cursor or the vehicle condition, price, and VIN values that make up the cursor. In this case, it can construct a new `PageRequest`,

```
PageRequest page2Request = PageRequest
             .ofPage(2) // cosmetic when using a cursor
             .size(25)
             .afterCursor(Cursor.forKey(lastCar.vehicleCondition,
                                        lastCar.price,
                                        lastCar.vin));
CursoredPage<Car> page2 =
        cars.find(make, model, page2Request, order);
```

4.9.2.7. Scenario: Person Entity and People Repository

This cursor-based pagination scenario uses the same `Person` entity and example dataset from the offset-based pagination scenario, but orders it by `name` and then by `id`,

```
[
   {"id":3, "name":"Alyse Dadson"},
   {"id":6, "name":"Boothe Martinson"},
   {"id":10, "name":"Boothe Martinson"},
   {"id":2, "name":"Corri Davidou"},
   {"id":8, "name":"Danita Pilipyak"},
   {"id":9, "name":"Harlene Branigan"},
   {"id":5, "name":"Jaquith Wealthall"},
   {"id":1, "name":"Lin Le Marchant"},
   {"id":4, "name":"Orelle Roughey"},
   {"id":7, "name":"Patten Bedell"}
]
```

```
@Repository
public interface People extends BasicRepository<Person, Long> {
    @Find
    CursoredPage<Person> findAll(PageRequest pagination,
                                 Order<Person> sorts);
}
```

Code Execution:

```
@Inject
People people;

Order<Person> order = Order.by(Sort.asc("name"),
                               Sort.asc("id"));
PageRequest firstPageRequest = PageRequest.ofSize(4);
CursoredPage<Person> page =
        people.findAll(firstPageRequest, order);
```

Resulting Page Content:

```
[
   {"id":3, "name":"Alyse Dadson"},
   {"id":6, "name":"Boothe Martinson"},
   {"id":10, "name":"Boothe Martinson"},
   {"id":2, "name":"Corri Davidou"}
]
```

Deletion of an Entity:

```
// The user decides to remove one of the entities that has the same name,
people.deleteById(10);
```

Next Page Execution:

```
if (page.hasNext()) {
   PageRequest nextPageRequest = page.nextPageRequest();
   CursoredPage<Person> page2 = people.findAll(nextPageRequest, order);
}
```

Resulting Page Content:

```
[
   {"id":8, "name":"Danita Pilipyak"},
   {"id":9, "name":"Harlene Branigan"},
   {"id":5, "name":"Jaquith Wealthall"},
   {"id":1, "name":"Lin Le Marchant"}
]
```

It should be noted, the above result is different than what would be retrieved with offset-based pagination, where the removal of an entity from the first page shifts the offset for entries 5 through 8 to start from `{"id":9, "name":"Harlene Branigan"}`, skipping over `{"id":8, "name":"Danita Pilipyak"}` that becomes offset position 4 after the removal. Cursor-based pagination does not skip the entity because it queries relative to a cursor position, starting from the next entity after `{"id":2, "name":"Corri Davidou"}`.

4.10. Precedence of repository methods

The following order, with the lower number having higher precedence, is used when interpreting the meaning of repository methods.

1. If the method is a Java `default` method, then its provided implementation is used.
2. If the method has a *resource accessor method* return type recognized by the Jakarta Data provider, then the method is implemented as a resource accessor method.
3. If the method is annotated with a query annotation recognized by the Jakarta Data provider, such as `@Query`, then the method is implemented to execute the query specified by the query annotation.
4. If the method is annotated with an automatic query annotation, such as `@Find`, or with a lifecycle annotation declaring the type of operation, for example, with `@Insert`, `@Update`, `@Save`, or `@Delete`, and the provider recognizes the annotation, then the annotation determines how the method is implemented, possibly with the help of other annotations present on the method parameters, for example, any `@By` annotations of the parameters.
5. If the method is named according to the conventions of *Query by Method Name*, then the method is implemented according to the Query by Method Name extension to this specification.

A repository method that does not fit any of the above patterns and is not handled as a vendor-specific extension to the specification must either result in an error at build time or raise `UnsupportedOperationException` at runtime.

4.11. Null arguments to repository methods

When a repository method is called with a null value as an argument to one of its parameters, the repository implementation might throw an exception:

- when a lifecycle method is called with a null entity instance, the repository implementation must throw `NullPointerException`, or
- when an annotated or parameter-based query method is called with a null argument, the repository implementation is permitted, but not required, to throw an appropriate exception type, unless the null argument

occurs as an argument to a special parameter, in which case the repository implementation is required to throw `NullPointerException`.

> The behavior of a query method when the method is called with a null argument is not defined by this specification, and is not portable between Jakarta Data providers.

4.12. Asynchronous repositories

An *asynchronous repository method* is a repository method which returns an object representing a value which will eventually be obtained from the database, but which might not yet be available. An asynchronous repository method is permitted to return such an object immediately, and access the database asynchronously.

An asynchronous repository method has a signature of form:

```
F<R> m(P1 p1, P2 p2, ...)
```

where:

- `R m(P1 p1, P2 p2, …)` is a legal repository method signature according to the previous sections of this chapter, or, in the special case that `R` is the type `java.lang.Void`, `void m(P1 p1, P2 p2, …)` is a legal repository method signature according to the previous sections of this chapter, and
- `F` is a parameterized type representing a value which might not yet have been computed and which is supported by the Jakarta Data provider.

> Every Jakarta Data provider is encouraged, but not required, to support asynchronous repository methods returning `java.util.concurrent.CompletionStage`.

> A repository method annotated with the `@Asynchronous` annotation from the Jakarta Concurrency specification is permitted to declare the return type `java.util.concurrent.CompletionStage`. In this case, the Jakarta Data provider must synchronously return an already-completed `CompletionStage` so that the Jakarta Concurrency provider is able to control the asynchronous behavior.

For example, the following is an asynchronous parameter-based query method that relies on the Jakarta Concurrency `@Asynchronous` interceptor to control the asynchronous behavior:

```
@Asynchronous
@Find
CompletionStage<Book> bookByIsbn(String isbn);
```

This method is an asynchronous lifecycle method that relies on the Jakarta Data provider to control the asynchronous behavior:

```
@Insert
CompletionStage<Void> insertBook(Book book);
```

An *asynchronous repository* is a repository which declares asynchronous repository methods. A repository may declare a mixture of synchronous and asynchronous repository methods if every asynchronous method is annotated with the `@Asynchronous` annotation, so that Jakarta Concurrency provides the asynchronous behavior. Otherwise, the Jakarta Data provider is not required to support mixing synchronous and asynchronous repository methods within the same repository interface. The `@Asynchronous` annotation must not be used on repositories implemented using reactive streams.

An asynchronous repository might be backed by a thread pool, or it might be implemented using reactive streams. Such implementation details are concerns of the Jakarta Data provider, and are beyond the scope of this specification.

4.13. Stateful repositories

Repositories in Jakarta Data are usually stateless. That is, the repository does not maintain any relationship with any entity instance across invocations of repository methods. Any instance of an entity class may be passed to a method of a stateless repository which accepts the entity type.

A *stateful repository* is a repository which maintains an association with a set of *managed* entity instances. Support for stateful repositories is defined in the dedicated module `jakarta.data.stateful`.

4.13.1. Persistence contexts

A stateful repository is backed by a *persistence context*, a set of managed entity instances in which at most one instance represents a given record in the database.

- An entity instance never belongs to multiple persistence contexts.
- The Jakarta Data implementation must ensure that a given persistence context never contains more than one entity instance representing the same record.
- Multiple repositories might share a persistence context, especially if they share a datastore.
- A persistence context is never shared across transactions.

A query method of a stateful repository which returns an entity type always returns managed instances belonging to the persistence context associated with the repository.

If an application program accesses an entity instance belonging to a persistence context associated with one transaction from within the scope of a second different transaction, the behavior is undefined by this specification.

This specification does not define the lifecycle of a persistence context, nor how a persistence context is propagated across repositories.

Implementations of Jakarta Data are encouraged to propagate a single persistence context within a given transaction across repositories which share a given datasource. For example, a Jakarta Data provider backed by an implementation of Jakarta Persistence might take advantage of standard Jakarta EE transaction-scoped persistence context propagation.

4.13.2. Stateful repository lifecycle operations

A lifecycle method of a stateful repository must be annotated with a lifecycle annotation specific to the stateful model.

The annotations `@Save`, `@Insert`, `@Update`, `@Delete` are used to define stateless repositories and must not be used to declare lifecycle methods of a stateful repository. Instead, this specification defines the special lifecycle annotations `@Persist`, `@Merge`, `@Refresh`, `@Remove`, and `@Detach` for declaring lifecycle methods of stateful repositories.

A lifecycle method annotated `@Remove` or `@Refresh` only accepts managed entities associated with the persistence context underlying the repository. On the other hand, a method annotated `@Persist`, `@Merge`, or `@Detach` also accepts unmanaged entities.

4.13.3. Automatic change detection

Any modification to the persistent state of a managed entity must be automatically detected by the Jakarta Data implementation, and the corresponding database record must be automatically updated. The application program is not required to explicitly call a repository method after modifying a managed entity.

Invocation of a lifecycle method or modification of the state of a managed entity might result in one or more records being inserted, updated, or deleted in the database. Such changes to the database do not typically happen synchronously with invocation of the lifecycle method, or immediately after modification of the managed entity. Instead, such changes are made when the persistence context is periodically *flushed*.

> This specification does not prescribe the timing of flush operations, but a flush typically happens before execution of a query or during the before completion phase of transaction commit. An implementation of Jakarta Data should flush as needed to ensure that query results are consistent with the current state of the persistence context.

Chapter 5. Jakarta Data Query Language

The Jakarta Data Query Language (JDQL) is a simple language designed to be used inside the `@Query` annotation to specify the semantics of query methods of Jakarta Data repositories. The language is in essence a subset of the widely-used Jakarta Persistence Query Language (JPQL), and thus a dialect of SQL. But, consistent with the goals of Jakarta Data, it is sufficiently limited in functionality that it is easily implementable across a wide variety of data storage technologies. Thus, the language defined in this chapter excludes features of JPQL which, while useful when the target datasource is a relational database, cannot be easily implemented on all non-relational datastores. In particular, the `from` clause of a Jakarta Data query may contain only a single entity.

> A Jakarta Data provider backed by access to a relational database might choose to allow the use of a much larger subset of JPQL—or even the whole language—via the `@Query` annotation. Such extensions are not required by this specification.

5.1. Type system

Every expression in a JDQL query is assigned a Java type. An implementation of JDQL is required to support the Java types listed in Basic types, that is: primitive types, `String`, `LocalDate`, `LocalDateTime`, `LocalTime`, `Year`, and `Instant`, `java.util.UUID`, `java.math.BigInteger` and `java.math.BigDecimal`, `byte[]`, and `enum` types.

> An implementation of JDQL is permitted and encouraged to support additional types. Use of such types is not guaranteed to be portable between implementations.

The interpretation of an operator expression or literal expression of a given type is given by the interpretation of the equivalent expression in Java. However, the precise behavior of some queries might vary depending on the native semantics of queries on the underlying datastore. For example, numeric precision and overflow, string collation, and integer division are permitted to depart from the semantics of the Java language.

> This specification should not be interpreted to mandate an inefficient implementation of query language constructs in cases where the native behavior of the database varies from Java in such minor ways. That said, portability between Jakarta Data providers is maximized when their behavior is closest to the Java language.

Since an attribute of an entity may be null, a JDQL expression may evaluate to a null value.

5.2. Lexical structure

Lexical analysis requires recognition of the following token types:

- keywords (reserved identifiers),
- regular identifiers,
- named and ordinal parameters,
- operators and punctuation characters,
- literal strings, and
- integer and decimal number literals.

5.2.1. Identifiers and keywords

An *identifier* is any legal Java identifier which is not a keyword. Identifiers are case-sensitive: `hello`, `Hello`, and `HELLO` are distinct identifiers.

In the JDQL grammar, identifiers are labelled with the `IDENTIFIER` token type.

The following identifiers are *keywords*: `select`, `update`, `set`, `delete`, `from`, `where`, `order`, `by`, `asc`, `desc`, `not`, `and`, `or`, `between`, `like`, `in`, `null`, `local`, `true`, `false`. In addition, every reserved identifier listed in section 4.4.1 of the Jakarta Persistence specification version 3.2 is also considered a reserved identifier. Keywords and other reserved identifiers are case-insensitive: `null`, `Null`, and `NULL` are three ways to write the same keyword.

Use of a reserved identifier as a regular identifier in JDQL might be accepted by a given Jakarta Data provider, but such usage is not guaranteed to be portable between providers.

5.2.2. Parameters

A *named parameter* is a legal Java identifier prefixed with the `:` character, for example, `:name`.

An *ordinal parameter* is a decimal integer prefixed with the `?` character, for example, `?1`.

Ordinal parameters are numbered sequentially, starting with `?1`.

5.2.3. Operators and punctuation

The character sequences `+`, `-`, `*`, `/`, `||`, `=`, `<`, `>`, `<>`, `<=`, `>=` are *operators*.

The characters `(`, `)`, and `,` are *punctuation characters*.

When working with NoSQL databases, the support for arithmetic operations and support of parentheses for precedence might vary significantly:

Key-value databases

Arithmetic operations (`+`, `-`, `*`, `/`) are not supported. These databases are designed for simple key-based lookups and lack query capabilities for complex operations.

Wide-column databases

Arithmetic operations are not required to be supported. Some wide-column databases might offer limited support, which might require secondary indexing even for basic querying.

Document Databases

Support of arithmetic operations and support of parenthesis for precedence are not required, although databases typically offer these capabilities. Behavior and extent of support can vary significantly between providers.

Graph Databases

Support for arithmetic operations and parentheses for precedence are not required but is typically offered by databases. Behavior and extent of support can vary significantly between providers.

Due to the diversity of NoSQL database types and their querying capabilities, there is no guarantee that all NoSQL providers will support punctuation characters such as parentheses `(`, `)` for defining operation precedence. It is recommended to consult your NoSQL provider’s documentation to confirm the supported query features and their behavior.

5.2.4. String literals

A *literal string* is a character sequence quoted using the character `'`.

A single literal `'` character may be included within a string literal by self-escaping it, that is, by writing `''`. For example, the string literal `'Furry''s theorem has nothing to do with furries.'` evaluates to the string `Furry's theorem has nothing to do with furries.`.

In the grammar, literal strings are labelled with the `STRING` token type.

5.2.5. Numeric literals

Numeric literals come in two flavors:

- any legal Java decimal literal of type `int` or `long` is an *integer literal*, and
- any legal Java literal of type `float` or `double` is a *decimal literal*.

In the grammar, integer and decimal literals are labelled with the `INTEGER` and `DOUBLE` token types respectively.

JDQL does not require support for literals written in octal or hexadecimal.

5.2.6. Whitespace

The characters Space, Horizontal Tab, Line Feed, Form Feed, and Carriage Return are considered whitespace characters and make no contribution to the token stream.

As usual, token recognition is "greedy". Therefore, whitespace must be placed between two tokens when:

- a keyword directly follows an identifier or named parameter,
- an identifier directly follows a keyword or named parameter, or
- a numeric literal directly follows an identifier, keyword, or parameter.

5.3. Expressions

An expression is a sequence of tokens to which a Java type can be assigned, and which evaluates to a well-defined value when the query is executed. In JDQL, expressions may be categorized as:

- literals,
- special values,
- parameters,
- enum literals,
- paths,
- function calls, and
- operator expressions.

5.3.1. Literal expressions

A string, integer, or decimal literal is assigned the type it would be assigned in Java. So, for example, `'Hello'` is assigned the type `java.lang.String`, `123` is assigned the type `int`, `1e4` is assigned the type `double`, and `1.23f` is assigned the type `float`.

The syntax for literal expressions is given by the `literal` grammar rule, and in the previous section titled Lexical structure.

When executed, a literal expression evaluates to its literal value.

5.3.2. Special values

The special values `true` and `false` are assigned the type `boolean`, and evaluate to their literal values.

The special values `local date`, `local time`, and `local datetime` are assigned the types `java.time.LocalDate`, `java.time.LocalTime`, and `java.time.LocalDateTime`, and evaluate to the current date and current datetime of the database server, respectively.

The syntax for special values is given by the `special_expression` grammar rule.

5.3.3. Parameter expressions

A parameter expression, with syntax given by `input_parameter`, is assigned the type of the repository method parameter it matches. For example, the parameter `:titlePattern` is assigned the type `java.lang.String`:

```
@Query("where title like :titlePattern")
List<Book> booksMatchingTitle(String titlePattern);
```

When executed, a parameter expression evaluates to the argument supplied to the parameter of the repository method.

Positional and named parameters must not be mixed in a single query.

5.3.4. Enum literals

An *enum literal expression* is a Java identifier, with syntax specified by `enum_literal`, and may only occur as the right operand of a `set` assignment or `=`/`<>` equality comparison. It is assigned the type of the left operand of the assignment or comparison. The type must be a Java `enum` type, and the identifier must be the name of an enumerated value of the `enum` type including the fully qualified Java enum class name. For example, `day <> java.time.DayOfWeek.MONDAY` is a legal comparison expression.

When executed, an enum expression evaluates to the named member of the Java `enum` type.

5.3.5. Path expressions

A *path expression* is a period-separated list of Java identifiers, with syntax specified by `state_field_path_expression`. Each identifier is interpreted as the name of an attribute of an entity or embeddable class. Each prefix of the list is assigned a Java type:

- the first element of the list is assigned the type of the named attribute of the entity being queried, and
- each subsequent element is assigned the type of the named attribute of the type assigned to the previous element.

The type of the whole path expression is the type of the last element of the list. For example, `pages` is assigned the type `int`, `address` is assigned the type `org.example.Address`, and `address.street` is assigned the type `java.lang.String`.

Typically, the last element of a path expression is assigned a basic type. Non-terminal path elements are usually assigned an embeddable type, if the element references an embedded attribute, or an entity type, if the element references an association attribute. However, since a Jakarta Data provider is not required to support embedded attributes or associations, a JDQL implementation is not required to support compound path expressions.

When a path expression is executed, each element of the path is evaluated in turn:

- the first element of the path expression is evaluated in the context of a given record of the queried entity type, and evaluates to the value of the named entity attribute of the given record, and then
- each subsequent element is evaluated in the context of the result produced the previous element (typically, and embeddable class or associated entity class), and evaluates to the value of the named attribute of the result.

If any element of a path expression evaluates to a null value, the whole path expression evaluates to a null value.

5.3.6. Identifier expressions

An *identifier expression*, with syntax given by `id_expression`, is assigned the type of the unique identifier of the queried entity and evaluates to the unique identifier of a given record. An identifier expression is a synonym for a path expression with one element matching the identifier attribute of the queried entity type. An identifier expression may occur in the `select` clause, in the `order` clause, or as a scalar expression in the `where` clause.

5.3.7. Function calls

A *function call* is the name of a JDQL function, followed by a parenthesized list of argument expressions, with syntax given by `function_expression`.

- The `abs()` function is assigned the type of its numeric argument, and evaluates to the absolute value of the numeric value to which its argument evaluates. Its argument must be of numeric type.
- The `length()` function is assigned the type `java.lang.Integer`, and evaluates to the length of string to which its argument evaluates. Its argument must be of type `java.lang.String`.
- The `lower()` function is assigned the type `java.lang.String`, and evaluates to the lowercase form of the string to which its argument evaluates. Its argument must be of type `java.lang.String`.
- The `upper()` function is assigned the type `java.lang.String`, and evaluates to the uppercase form of the string to which its argument evaluates. Its argument must be of type `java.lang.String`.
- The `left()` function is assigned the type `java.lang.String`, and evaluates to a prefix of the string to which its first argument evaluates. The length of the prefix is given by the integer value to which its second argument evaluates. The first argument must be of type `java.lang.String`, and the second argument must be of integral numeric type.
- The `right()` function is assigned the type `java.lang.String`, and evaluates to a suffix of the string to which its first argument evaluates. The length of the suffix is given by the integer value to which its second argument evaluates. The first argument must be of type `java.lang.String`, and the second argument must be of integral numeric type.

When any argument expression of any function call evaluates to a null value, the whole function call evaluates to null.

These functions cannot be emulated on every datastore. When a function cannot be reasonably emulated via the native query capabilities of the database, a JDQL implementation is not required to provide the function.

If the JDQL implementation does not support a standard function explicitly listed above, it must throw `UnsupportedOperationException` when the function name occurs in a query. Alternatively, the Jakarta Data provider is permitted to reject a repository method declaration at compilation time if its `@Query` annotation uses an unsupported function.

On the other hand, an implementation of JDQL might provide additional built-in functions, and might even allow invocation of user-defined functions. Section 4.7 of the Jakarta Persistence specification defines a set of functions that all JPQL implementations are required to provide, including `concat`, `substring`, `trim`, `locate`, `ceiling`, `floor`, `exp`, `ln`, `mod`, `power`, `round`, `sign`, `sqrt`, `cast`, `extract`, `coalesce`, and `nullif`.

> JDQL implementations are encouraged to support any of these functions which are reasonably implementable.

5.3.8. Operator expressions

The syntax of an *operator expression* is given by the `scalar_expression` rule. Within an operator expression, parentheses indicate grouping.

All binary infix operators are left-associative. The relative precedence, from highest to lowest precedence, is given by:

1. `*` and `/`,
2. `+` and `-`,
3. `||`.

The unary prefix operators `+` and `-` have higher precedence than the binary infix operators. Thus, `2 * -3 + 5` means `(2 * (-3)) + 5` and evaluates to `-1`.

The concatenation operator `||` is assigned the type `java.lang.String`. Its operand expressions must also be of type `java.lang.String`. When executed, a concatenation operator expression evaluates to a new string concatenating the strings to which its arguments evaluate.

The numeric operators `+`, `-`, `*`, and `/` have the same meaning for primitive numeric types they have in Java, and operator expression involving these operators are assigned the types they would be assigned in Java.

> As an exception, when the operands of `/` are both integers, a JDQL implementation is not required to interpret the operator expression as integer division if that is not the native semantics of the database. However, portability is maximized when Jakarta Data providers *do* interpret such an expression as integer division.

The four numeric operators may also be applied to an operand of wrapper type, for example, to `java.lang.Integer` or `java.lang.Double`. In this case, the operator expression is assigned a wrapper type, and evaluates to a null value when either of its operands evaluates to a null value. When both operands are non-null, the semantics are identical to the semantics of an operator expression involving the corresponding primitive types.

The four numeric operators may also be applied to operands of type `java.math.BigInteger` or `java.math.BigDecimal`.

A numeric operator expression is evaluated according to the native semantics of the database. In translating an operator expression to the native query language of the database, a Jakarta Data provider is encouraged, but not required, to apply reasonable transformations so that evaluation of the expression more closely mimics the semantics of the Java language.

5.3.9. Numeric types and numeric type promotion

The type assigned to an operator expression depends on the types of its operand expression, which need not be identical. The rules for numeric promotion are given in section 4.7 of the Jakarta Persistence specification version 3.2:

> - *If there is an operand of type* `Double` *or* `double`*, the expression is of type* `Double`*;*
> - *otherwise, if there is an operand of type* `Float` *or* `float`*, the expression is of type* `Float`*;*
> - *otherwise, if there is an operand of type* `BigDecimal`*, the expression is of type* `BigDecimal`*;*
> - *otherwise, if there is an operand of type* `BigInteger`*, the expression is of type* `BigInteger`*, unless the operator is* `/` *(division), in which case the expression type is not defined here;*

- *otherwise, if there is an operand of type* `Long` *or* `long`*, the expression is of type* `Long`*, unless the operator is* `/` *(division), in which case the expression type is not defined here;*
- *otherwise, if there is an operand of integral type, the expression is of type* `Integer`*, unless the operator is* `/` *(division), in which case the expression type is not defined here.*

5.4. Conditional expressions

A *conditional expression* is a sequence of tokens which specifies a condition which, for a given record, might be *satisfied* or *unsatisfied*. Unlike the scalar Expressions defined in the previous section, a conditional expression is not considered to have a well-defined type.

JPQL defines the result of a conditional expression in terms of ternary logic. JDQL does not specify that a conditional expression evaluates to well-defined value, only the effect of the conditional expression when it is used as a restriction. The "value" of a conditional expression is not considered observable by the application program.

Conditional expressions may be categorized as:

- `null` comparisons,
- `in` expressions,
- `between` expressions,
- `like` expressions,
- equality and inequality operator expressions, and
- logical operator expressions.

The syntax for conditional expressions is given by the `conditional_expression` rule. Within a conditional expression, parentheses indicate grouping.

5.4.1. Null comparisons

A `null` comparison, with syntax given by `null_comparison_expression` is satisfied when:

- the `not` keyword is missing, and its operand evaluates to a null value, or
- the `not` keyword occurs, and its operand evaluates to any non-null value.

5.4.2. In expressions

An `in` expression, with syntax given by `in_expression` is satisfied when its leftmost operand evaluates to a non-null value, and:

- the `not` keyword is missing, and any one of its parenthesized operands evaluates to the same value as its leftmost operand, or
- the `not` keyword occurs, and none of its parenthesized operands evaluate to the same value as its leftmost operand.

All operands must have the same type.

5.4.3. Between expressions

A `between` expression, with syntax given by `between_expression` is satisfied when its operands all evaluate to non-null values, and, if the `not` keyword is missing, its left operand evaluates to a value which is:

- larger than or equal to the value taken by its middle operand, and
- smaller than or equal to the value taken by its right operand.

Or, if the `not` keyword occurs, the left operand must evaluate to a value which is:

- strictly smaller than to the value taken by its middle operand, or
- strictly larger than the value taken by its right operand.

All three operands must have the same type.

5.4.4. Like expressions

A `like` expression is satisfied when its left operand evaluates to a non-null value and:

- the `not` keyword is missing, and this value matches the pattern, or
- the `not` keyword occurs, and the value does not match the pattern.

The left operand must have type `java.lang.String`.

Within the pattern, `_` matches any single character, and `%` matches any sequence of characters.

5.4.5. Equality and inequality operators

The equality and inequality operators are `=`, `<>`, `<`, `>`, `<=`, `>=`.

- For primitive types, these operators have the same meaning they have in Java, except for `<>` which has the same meaning that `!=` has in Java. Such an operator expression is satisfied when the equivalent operator expression would evaluate to `true` in Java.
- For wrapper types, these operators are satisfied if both operands evaluate to non-null values, and the equivalent operator expression involving primitives would be satisfied.
- For other types, these operators are evaluated according to the native semantics of the database.

The operands of an equality or inequality operator must have the same type.

Portability is maximized when Jakarta Data providers interpret equality and inequality operators in a manner consistent with the implementation of `Object.equals()` or `Comparable.compareTo()` for the assigned Java type.

When using NoSQL databases, there are limitations to the support of equality and inequality operators:

1. **Key-Value Databases**: Support for the equality restriction on the key attribute is required. The key attribute is defined by the annotation `jakarta.nosql.Id`. Key-value databases are not required to support any other restrictions.
2. **Wide-Column Databases**: Support for equality restriction and the inequality restriction on the `Id` attribute is required. Support for restrictions on other entity attributes is not required. These operations typically work only with the `Id` by default but might be compatible for other entity attributes if secondary indexes are configured in the database schema.
3. **Graph and Document Databases**: Support for all equality and inequality operators is required.

5.4.6. Ordering

Every basic type can, in principle, be equipped with a total order. An order for a type determines the result of inequality comparisons, and the effect of the Order clause.

For numeric types, and for date, time, and datetime types, the total order is unique and completely determined by the semantics of the type. JDQL implementations must sort these types according to their natural order, that is, the order in JDQL must agree with the order defined by Java.

Boolean values must be ordered so that `false < true` is satisfied.

For other types, there is at least some freedom in the choice of order. Usually, the order is determined by the native semantics of the database. Note that:

- Textual data is represented in JDQL as the type `java.lang.String`. Strings are in general ordered lexicographically, but the ordering also depends on the character set and collation used by the database server. Applications must not assume that the order agrees with the `compareTo()` method of `java.lang.String`. In evaluating an inequality involving string operands, an implementation of JDQL is not required to emulate Java collation.
- Binary data is represented in JDQL as the type `byte[]`. Binary data is in general ordered lexicographically with respect to the constituent bytes. However, since this ordering is rarely meaningful, this specification does not require implementations of JDQL to respect it.
- This specification does not define an order for the sorting of Java `enum` values, which is provider-dependent. A programming model for entity classes might allow control over the order of `enum` values. For example, Jakarta Persistence allows this via the `@Enumerated` annotation.
- This specification does not define an order for UUID values, which is provider-dependent.

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

Key-value databases

Sorting of results is not supported.

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.

5.4.7. Logical operators

The logical operators are `and`, `or`, and `not`.

- An `and` operator expression is satisfied if and only if both its operands are satisfied.

- An `or` operator expression is satisfied if at least one of its operands is satisfied.
- A `not` operator expression is never satisfied if its operand *is* satisfied.

This specification leaves undefined the interpretation of the `not` operator when its operand *is not* satisfied.

A compliant implementation of JDQL might feature SQL/JPQL-style ternary logic, where `not n > 0` is an unsatisfied logical expression when `n` evaluates to null, or it might feature binary logic where the same expression is considered satisfied. Application programmers should take great care when using the `not` operator with scalar expressions involving `null` values.

Syntactically, logical operators are parsed with lower precedence than equality and inequality operators and other conditional expressions listed above. The `not` operator has higher precedence than `and` and `or`. The `and` operator has higher precedence than `or`.

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

Key-value databases

Support for the equality restriction is required for the `Id` attribute. There is no requirement to support other types of restrictions or restrictions on other entity attributes.

Wide-column databases

Wide-column databases are not required to support the `AND` operator or 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` operators 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.

5.5. Clauses

Each JDQL statement is built from a sequence of *clauses*. The beginning of a clause is identified by a keyword: `from`, `where`, `select`, `set`, or `order`.

There is a logical ordering of clauses, reflecting the order in which their effect must be computed by the datastore:

1. `from`
2. `where`,
3. `select` or `set`,
4. `order`.

The interpretation and effect of each clause in this list is influenced by clauses occurring earlier in the list, but not by clauses occurring later in the list.

5.5.1. From clause

The `from` clause, with syntax given by `from_clause`, specifies an *entity name* which identifies the queried entity. Path expressions occurring in later clauses are interpreted with respect to this entity. That is, the first element of each path expression in the query must be a persistent attribute of the entity named in the `from` clause. The entity name is a Java identifier, usually the unqualified name of the entity class, as specified in Entity names.

> The syntax of the `update` statement is irregular, with the `from` keyword implied. That is, the syntax *should* be `update from Entity`, but for historical reasons it is simply `update Entity`.

The `from` clause is optional in `select` statements. When it is missing, the queried entity is determined by the return type of the repository method, or, if the return type is not an entity type, by the primary entity type of the repository.

For example, this repository method declaration:

```
@Query("where title like :title")
List<Book> booksByType(String title);
```

is equivalent to:

```
@Query("from Book where title like :title")
List<Book> booksByType(String title);
```

5.5.2. Where clause

The `where` clause, with syntax given by `where_clause`, specifies a conditional expression used to restrict the records returned, deleted, or updated by the query. Only records for which the conditional expression is satisfied are returned, deleted, or updated.

The `where` clause is always optional. When it is missing, there is no restriction, and all records of the queried entity type are returned, deleted, or updated.

5.5.3. Select clause

The `select` clause, with syntax given by `select_clause`, specifies one or more path expressions which determine the values returned by the query. Each path expression is evaluated for each record which satisfies the restriction imposed by the `where` clause, as specified in Path expressions, and a tuple containing the resulting values is added to the query results.

> When a `select` clause contains more than one item, the query return type must be a Java `record` type, and the elements of the tuple are repackaged as an instance of the query return type by calling a constructor of the `record`, passing the elements in the same order they occur in the `select` list. When the `select` clause contains only one item, the query directly returns the values of the path expression.

Alternatively, the `select` clause may contain either:

- a single `count(this)` aggregate expression, which evaluates to the number of records which satisfy the restriction, or
- a single identifier expression, which evaluates to the unique identifier of each record.

```
A query beginning with `select count(this)` always returns a single result of type `Long`, no matter how many
records satisfy the conditional expression in the `where` clause.
```

> If a datastore does not natively provide the ability to count query results, the Jakarta Data provider is strongly encouraged, but not required, to implement this operation by counting the query results in Java.

If the JDQL implementation does not support `count(this)`, it must throw `UnsupportedOperationException` when this aggregate expression occurs in a query. Alternatively, the Jakarta Data provider is permitted to reject a repository method declaration at compilation time if its `@Query` annotation uses the unsupported aggregate expression.

The `select` clause is optional in `select` statements. When it is missing, the query returns the queried entity.

> When working with NoSQL databases, the `select` clause behavior may vary depending on the database structure and capabilities:
>
> **Key-value databases**
>
> These databases generally do not support `select` clauses beyond retrieving values by their keys. Support for complex path expressions and aggregate functions like `count(this)` is not required.
>
> **Wide-column databases**
>
> The ability to use a `select` clause may depend on the presence of secondary indexes. Without secondary indexes, selection is often restricted to key-based operations. Support for `count(this)` is not required.
>
> **Graph and document databases**
>
> Support for flexible `select` clauses, including path expressions and aggregate functions like `count(this)` is required. Performance might vary based on the size and indexing of the dataset.
>
> For `count(this)` in particular, if the NoSQL datastore does not natively support counting query results, the Jakarta Data provider is encouraged to implement this operation in Java. However, providers are not required to do so. If `count(this)` is unsupported, an `UnsupportedOperationException` must be thrown during query execution, or repository methods using this expression may be rejected at compilation time.
>
> It is advisable to review your NoSQL provider's documentation to confirm the support and performance implications of `select` clauses and aggregate functions in your queries.

5.5.4. Set clause

The `set` clause, with syntax given by `set_clause`, specifies a list of updates to attributes of the queried entity. For each record which satisfies the restriction imposed by the `where` clause, and for each element of the list, the scalar expression is evaluated and assigned to the entity attribute identified by the path expression.

5.5.5. Order clause

The `order` clause (or `order by` clause), with syntax given by `orderby_clause`, specifies a lexicographic order for the query results, that is, a list of entity attributes used to sort the records which satisfy the restriction imposed by the `where` clause. The keywords `asc` and `desc` specify that a given attribute should be sorted in ascending or descending order respectively; when neither is specified, ascending order is the default.

> An implementation of JDQL is not required to support sorting by entity attributes which are not returned by the query. If a query returns the queried entity, then any sortable attribute of the queried entity may occur in the `order` clause. Otherwise, if the query has an explicit `select` clause, a provider might require that an attribute which occurs in the `order` also occurs in the `select`.

Entity attributes occurring earlier in the `order by` clause take precedence. That is, an attribute occurring later in the `order by` clause is only used to resolve "ties" between records which cannot be unambiguously ordered using only earlier attributes.

This specification does not define how null values are ordered with respect to non-null values. The ordering of null values may vary between data stores and between Jakarta Data providers.

The `order` clause is always optional. When it is missing, and when no sort criteria are given as arguments to a parameter of the repository method, the order of the query results is undefined, and might not be deterministic.

If a datastore does not natively provide the ability to sort query results, the Jakarta Data provider is strongly encouraged, but not required, to sort the query results in Java before returning the results to the client.

If the Jakarta Data provider cannot satisfy a request for sorted query results, it must throw `DataException`.

5.6. Statements

Finally, there are three kinds of *statement*:

- `select` statements,
- `update` statements, and
- `delete` statements.

The clauses which can appear in a statement are given by the grammar for each kind of statement.

5.6.1. Select statements

A `select` statement, with syntax given by `select_statement`, returns data to the client. For each record which satisfies the restriction imposed by the `where` clause, a result is returned containing the value obtained by evaluating the path expression in the `select` clause. Alternatively, for the case of `select count(this)`, the query returns the number of records which satisfied the restriction.

5.6.2. Update statements

An `update` statement, with syntax given by `update_statement`, updates each record which satisfies the restriction imposed by the `where` clause, and returns the number of updated records to the client.

If a NoSQL database is not capable of conditional updates or cannot determine the number of matching records reliably for an `update` operation that returns an `int` or `long`, the `update` operation must throw an `UnsupportedOperationException`.

Additionally, in databases with **append-only semantics**—such as many time-series and wide-column databases—the `update` operation may behave more like an `insert`, and repeated updates to the same record might not overwrite previous values.

5.6.3. Delete statements

A `delete` statement, with syntax given by `delete_statement`, deletes each record which satisfies the restriction imposed by the `where` clause, and returns the number of deleted records to the client.

If a NoSQL database is not capable of the execution of conditional deletes or cannot determine the number of deleted records reliably for a `delete` operation that returns an `int` or `long`, the `delete` operation must throw an `UnsupportedOperationException`.

5.7. Syntax

The following grammar defines the syntax of JDQL, via ANTLR4-style BNF.

```
grammar JDQL;

statement : select_statement | update_statement | delete_statement;

select_statement : select_clause? from_clause? where_clause? orderby_clause?;
update_statement : 'UPDATE' entity_name set_clause where_clause?;
delete_statement : 'DELETE' from_clause where_clause?;

from_clause : 'FROM' entity_name;

where_clause : 'WHERE' conditional_expression;

set_clause : 'SET' update_item (',' update_item)*;
update_item : state_field_path_expression '=' (scalar_expression | 'NULL');

select_clause : 'SELECT' (select_item | select_items);
select_item
    : state_field_path_expression
    | id_expression
    | aggregate_expression
    ;
select_items
    : state_field_path_expression (',' state_field_path_expression)+
    ;

orderby_clause : 'ORDER' 'BY' orderby_item (',' orderby_item)*;
orderby_item : (state_field_path_expression | id_expression) ('ASC' | 'DESC');

conditional_expression
    // highest to lowest precedence
    : '(' conditional_expression ')'
    | null_comparison_expression
    | in_expression
    | between_expression
    | like_expression
    | comparison_expression
    | 'NOT' conditional_expression
    | conditional_expression 'AND' conditional_expression
    | conditional_expression 'OR' conditional_expression
    ;

comparison_expression : scalar_expression ('=' | '>' | '>=' | '<' | '<=' | '<>') scalar_expression;
between_expression : scalar_expression 'NOT'? 'BETWEEN' scalar_expression 'AND' scalar_expression;
like_expression : scalar_expression 'NOT'? 'LIKE' STRING;

in_expression : state_field_path_expression 'NOT'? 'IN' '(' in_item (',' in_item)* ')';
in_item : literal | enum_literal | input_parameter;

null_comparison_expression : state_field_path_expression 'IS' 'NOT'? 'NULL';

scalar_expression
    // highest to lowest precedence
    : '(' scalar_expression ')'
    | primary_expression
    | ('+' | '-') scalar_expression
    | scalar_expression ('*' | '/') scalar_expression
    | scalar_expression ('+' | '-') scalar_expression
    | scalar_expression '||' scalar_expression
    ;

primary_expression
    : function_expression
    | special_expression
    | id_expression
    | state_field_path_expression
    | enum_literal
    | input_parameter
    | literal
    ;

id_expression : 'ID' '(' 'THIS' ')' ;

aggregate_expression : 'COUNT' '(' 'THIS' ')';

function_expression
    : 'ABS' '(' scalar_expression ')'
    | 'LENGTH' '(' scalar_expression ')'
    | 'LOWER' '(' scalar_expression ')'
    | 'UPPER' '(' scalar_expression ')'
    | 'LEFT' '(' scalar_expression ',' scalar_expression ')'
    | 'RIGHT' '(' scalar_expression ',' scalar_expression ')'
    ;

special_expression
    : 'LOCAL' 'DATE'
    | 'LOCAL' 'DATETIME'
    | 'LOCAL' 'TIME'
    | 'TRUE'
    | 'FALSE'
    ;

state_field_path_expression : IDENTIFIER ('.' IDENTIFIER)*;

entity_name : IDENTIFIER; // no ambiguity

enum_literal : IDENTIFIER ('.' IDENTIFIER)*; // ambiguity with state_field_path_expression resolvable semantically

input_parameter : ':' IDENTIFIER | '?' INTEGER;

literal : STRING | INTEGER | DOUBLE;
```

Chapter 6. Jakarta Data providers

A Jakarta Data provider might come as an integrated component of a Jakarta EE container, or it might be a separate component that integrates with the Jakarta EE container via standard or proprietary SPIs. For example, a Jakarta Data provider might use a CDI portable extension to integrate with dependency injection. The role of the Jakarta Data provider is to supply implementations of application-defined repositories.

6.1. Providers and repositories

A Jakarta Data provider must supply an implementation of each repository interface which either:

- explicitly specifies the name of the provider via the `provider` member of the `@Repository` annotation, or
- does not explicitly specify a `provider`, but whose entity classes are annotated with an entity-defining annotation type supported by the provider.

The repository implementation and the Jakarta Data provider are responsible for mediating interaction between the application program and the datastore, often taking advantage of other Jakarta EE services such as datasource provisioning and transaction management.

An application usually obtains a reference to an implementation of a repository via dependency injection. Therefore, the provider must make its repository implementations available to the bean container responsible for dependency injection.

Multiple Jakarta Data providers may coexist within a single program and provide repository implementations. It is the responsibility of the application developer to ensure that each repository is eligible for implementation by exactly one provider, according to the rules specified below.

> If the application developer fails to disambiguate the provider for every repository in the application, the behavior is undefined by this specification.

6.2. Provider support for entities

A Jakarta Data provider typically supports one entity-defining annotation type, but it may support multiple entity-defining annotation types. A provider may even have no entity-defining annotation and feature a programming model for entity classes where the entity classes are unannotated.

In particular:

- The `jakarta.persistence.Entity` annotation from the Jakarta Persistence specification is an entity-defining annotation for Jakarta Data providers backed by a Jakarta Persistence provider. Other Jakarta Data providers must not support the use of `jakarta.persistence.Entity` as an entity-defining annotation.
- The `jakarta.nosql.Entity` annotation from the Jakarta NoSQL specification is an entity-defining annotation for Jakarta Data providers backed by NoSQL databases. Other Jakarta Data providers must not support the use of `jakarta.nosql.Entity` as an entity-defining annotation.

A Jakarta Data provider might define a custom entity-defining annotation. Custom entity-defining annotations must be marked with `@EntityDefining`. This allows other Jakarta Data providers and tools to recognize and process entities declared with custom entity-defining annotations.

The lifecycle methods and query methods of a repository operate on a set of entities associated with that repository. All entities associated with a given repository should be declared using the same entity-defining annotation type. If the entities associated with a given repository are declared using a mix of annotation types, the behavior is undefined by this specification.

A Jakarta Data provider must ignore every repository associated with entity classes declared using an entity-defining annotation which is available at runtime, but not recognized by the provider, allowing another Jakarta Data provider to supply the implementation.

On the other hand, if an entity class is annotated with an entity-defining annotation which is *not* available at runtime, the Jakarta Data provider must completely ignore the annotation, and treat the entity as if that entity-defining annotation were missing.

> This allows an entity class to be compiled with multiple entity-defining annotations, each targeting a different Jakarta Data provider, and have the provider automatically disambiguated at runtime based on the availability of the entity-defining annotations.

6.3. Provider name

The entity-defining annotation type is the preferred way to avoid conflicts between Jakarta Data providers. But when multiple Jakarta Data providers support the same entity-defining annotation, the application must disambiguate the Jakarta Data provider for a repository by explicitly specifying the `provider` attribute of the `Repository` annotation.

A Jakarta Data provider must ignore every repository which specifies the name of a different `provider` in its `@Repository` annotation.

Chapter 7. Interoperability with other Jakarta EE specifications

This section discusses interoperability with related Jakarta EE [1] specifications. When operating within a Jakarta EE product, the availability of other Jakarta EE technologies depends on whether the Jakarta EE Core profile, Jakarta EE Web profile, or Jakarta EE Platform is used.

7.1. Jakarta Contexts and Dependency Injection

Contexts and Dependency Injection [2] (CDI) is a specification in the Jakarta EE Core profile that provides a powerful and flexible dependency injection framework for Java applications. CDI provides a programming model based around decoupled components with container-managed lifecycles and container-injected dependencies, enabling loose coupling and promoting modular and reusable code.

7.1.1. CDI dependency injection

In the Jakarta EE environment, CDI allows implementations of Jakarta Data repositories to be made available for injection via the `@Inject` annotation.

The following example illustrates this integration:

```
@Repository
public interface CarRepository extends BasicRepository<Car, Long> {

  List<Car> findByType(CarType type);

  Optional<Car> findByName(String name);

}
```

Here, a `CarRepository` interface extends the `BasicRepository` interface provided by Jakarta Data, which pre-defines a set of basic operations for entities, as described in Repositories with built-in supertypes.

The `@Repository` annotation instructs the Jakarta Data provider to:

- generate an implementation of the repository interface, as defined in Repository interfaces, and
- make the implementation available as a CDI bean, either by registering the implementation class itself as a bean, or by registering a producer or producer factory.

The repository implementation bean must have:

- qualifier type `@Default`, and
- the repository interface as a bean type.

Thus, the implementation is eligible for injection to unqualified injection points typed to the repository interface, as defined by section 2.4 of the CDI specification, version 4.0.

This specification does not restrict the scope of the repository implementation bean.

In our example, the repository implementation is eligible for injection to unqualified injection points of type `CarRepository`.

```
@Inject
CarRepository repository;

// ...
```

```
List<Car> cars = repository.findByType(CarType.SPORT);
```

This fragment shows how the application might request injection of a `CarRepository` instance using the `@Inject` annotation, and then invoke various data access methods declared or inherited by the `CarRepository` interface, such as `save()`, `findByType()`, and `findByName()`.

This integration between CDI and Jakarta Data allows for seamless management of repository instances within Jakarta EE applications.

7.1.2. CDI events

A repository implementation may raise CDI events. In the Jakarta EE environment, the repository implementation is required to raise the event types defined in the package `jakarta.data.event` when lifecycle methods annotated `@Insert`, `@Update`, or `@Delete` are invoked, as specified by the API documentation of these annotations.

7.1.3. CDI extensions for Jakarta Data providers

In environments where CDI Full or CDI Lite is available, Jakarta Data providers can make use of a CDI extension—an implementation of `jakarta.enterprise.inject.spi.Extension` or `jakarta.enterprise.inject.build.compatible.spi.BuildCompatibleExtension`--to discover interfaces annotated with `@Repository` and make their implementations available for injection.

Jakarta Data does not mandate the use of a specific kind of CDI extension but places the general requirement on the Jakarta Data provider to arrange for injection of the provided repository implementation into injection points typed to the repository interface and having no qualifiers (other than `Default` or `Any`), as described above.

CDI Lite (corresponding to Jakarta Core profile) does not include a requirement to support `jakarta.enterprise.inject.spi.Extension`, which is part of CDI Full (Jakarta Web profile and Jakarta Platform). The `jakarta.enterprise.inject.build.compatible.spi.BuildCompatibleExtension` applies to both CDI Lite and CDI Full.

Jakarta Data providers that wish to provide both extensions can use CDI's `@SkipIfPortableExtensionPresent` to prevent the `BuildCompatibleExtension` from colliding with the portable `Extension` when running in the Jakarta Web Profile or Jakarta Platform where CDI Full is present.

7.2. Jakarta Interceptors

A repository interface or method of a repository interface may be annotated with an interceptor binding annotation. In the Jakarta EE environment—or in any other environment where Jakarta Interceptors [3] is available and integrated with Jakarta CDI—if the repository implementation is instantiated by the CDI bean container then the interceptor binding annotation is inherited by the repository implementation. That is, the interceptor binding annotation must be treated as if it were placed directly on the repository implementation bean. The interceptors bound to the annotation are applied automatically by the implementation of Jakarta Interceptors.

7.3. Jakarta Transactions

Work performed by a repository might occur within the scope of a transaction managed by Jakarta Transactions.[4] This is usually transparent to the code acting as client of the repository. When:

1. Jakarta Transactions is available,
2. a global transaction is active on the thread of execution in which a repository operation is called, and
3. the data source backing the repository is capable of transaction enlistment,

then the repository operation must be performed within the context of the global transaction. That is, the data source resources involved in the operation must be enlisted as participants in the transaction.

> In the Jakarta EE environment, such enlistment usually happens automatically when the repository implementation makes use of a data source which is under the management of the Jakarta EE container.

The repository operation must not commit or roll back a transaction which was already associated with the thread in which the repository operation was called, but it might cause the transaction to be marked for rollback if the repository operation fails, that is, it may set the transaction status to `jakarta.transaction.Status.STATUS_MARKED_ROLLBACK`.

A repository interface or method of a repository interface may be marked with the annotation `jakarta.transaction.Transactional`. When a repository operation marked `@Transactional` is called in an environment where both Jakarta Transactions and Jakarta CDI are available, the semantics of this annotation must be observed during execution of the repository operation.

> In the Jakarta EE environment, the `@Transactional` annotation is automatically inherited by the repository implementation from the user-written repository interface, and the semantics of the `@Transactional` annotation are applied automatically by the implementation of Jakarta Interceptors supplied by the Jakarta EE container.

7.4. Jakarta Persistence

Integration with Jakarta Persistence is left undefined in this first release of Jakarta Data.

7.5. Jakarta NoSQL

When integrating Jakarta Data with Jakarta NoSQL, developers can use the NoSQL annotations to define the mapping of entities in repositories. Entities in Jakarta NoSQL are typically annotated with `jakarta.nosql.Entity` to indicate their suitability for persistence in NoSQL databases.

A Jakarta Data provider that supports Jakarta NoSQL will scan classes marked with the `jakarta.nosql.Entity` annotation.

By supporting Jakarta NoSQL annotations, Jakarta Data providers enable Java developers to utilize familiar and standardized mapping techniques when defining entities in repositories, ensuring compatibility and interoperability with the respective technologies.

7.6. Jakarta Bean Validation

Integrating with Jakarta Bean Validation [5] ensures data consistency within the Java layer. By applying validation rules to the data, developers can enforce constraints and business rules, preventing invalid or inconsistent information from being processed or persisted.

Using Jakarta Validation brings several advantages. It helps maintain data integrity, improves data quality, and enhances the reliability of the application. Catching validation errors early in the Java layer can identify and resolve potential issues before further processing or persistence occurs. Additionally, Jakarta Validation allows for declarative validation rules, simplifying the validation logic and promoting cleaner and more maintainable code.

In Jakarta Data, repository methods participate in method validation as defined by the section "Method and constructor validation" of the Jakarta Validation specification. Method validation includes validation of constraints on method parameters and results. The `jakarta.validation.Valid` annotation is used to opt in to cascading validation that validates constraints that are found on an object that is supplied as a parameter or returned as a result.

The following code snippet demonstrates the usage of Jakarta Validation annotations in the `Student` entity class:

```java
@Entity
public class Student {

    @Id
    private String id;

    @Column
    @NotBlank
    private String name;

    @Positive
    @Min(18)
    @Column
    private int age;
}
```

In this example, the `name` field is annotated with `@NotBlank`, indicating that it must not be blank. The `age` field is annotated with both `@Positive` and `@Min(18)`, ensuring it is a positive integer greater than or equal to 18.

The `School` repository interface, shown below, uses the `jakarta.validation.Valid` annotation to cause the constraints from the `Student` entity to be validated during the `save` operation, whereas the validation constraints are not applied to the `Student` entities returned as a result of the `findByAgeLessThanEqual` operation because the `findByAgeLessThanEqual` method does not include a `jakarta.validation.Valid` annotation that applies to the return value.

```java
@Repository
public interface School extends DataRepository<Student, String> {
    @Save
    void save(@Valid Student s);

    List<Student> findByAgeLessThanEqual(@Min(18) int age);
}
```

[1] Jakarta EE Platform 11, https://jakarta.ee/specifications/platform/11/

[2] Jakarta Contexts and Dependency Injection 4.1, https://jakarta.ee/specifications/cdi/4.1/

[3] Jakarta Interceptors 2.2, https://jakarta.ee/specifications/interceptors/2.2/

[4] Jakarta Transactions 2.0, https://jakarta.ee/specifications/transactions/2.0/

[5] Jakarta Bean Validation 3.1, https://jakarta.ee/specifications/bean-validation/3.1/

Chapter 8. Portability in Jakarta Data

Jakarta Data offers varying degrees of portability depending on the database and capabilities used. A subset of function is standardized across all database types, while other subsets of function are standardized only for the specific types of databases to which the capability pertains. These requirements are explicitly called out in documentation throughout the specification, such as the "Unavailable In" column of the tables of repository keywords in the Jakarta Data module Javadoc. The Jakarta Data specification levies requirements against applications and Jakarta Data providers, but not against databases. The Jakarta Data specification requires the Jakarta Data provider to offer function to the extent that the database is capable and to raise an exception where the database is not capable. It is not the aim of Jakarta Data to offer the ability to switch between different databases, but to standardize a common starting point for data access from which capability that is specific to the various types of databases is able to build upon.

The portability that is offered by Jakarta Data pertains to usage of the Jakarta Data API by application code, enabling application code that restricts itself to the Jakarta Data API to remain the same when used with any Jakarta Data provider running against the same database. Jakarta Data relies on external persistence specifications such as Jakarta Persistence and Jakarta NoSQL to standardize entity models. Jakarta Data does not place any requirements on the format of data to make persisted data and other database artifacts portable across providers. Jakarta Data does not offer any means to migrate data that is persisted by one provider to a form that is usable by another provider.

8.1. Portability for relational databases

A Jakarta Data provider backed by access to relational data must support the built-in lifecycle annotations `@Insert`, `@Update`, and `@Delete`, along with the built-in repository types `BasicRepository` and `CrudRepository`. It must also fully support Jakarta Data query methods, including pagination, ordering, and limiting, subject to the caveat that follows.

> ℹ The SQL dialect and the database set limits on what operations are implementable. A Jakarta Data provider is not required to supply an implementation of a repository which declares query methods mapping to operations which are not supported by the database itself.

If the provider is backed by JDBC, it should support resource accessor methods of type `java.sql.Connection`.

8.2. Portability for NoSQL databases

Portability in Jakarta Data extends to various NoSQL databases, each presenting unique challenges and capabilities. Jakarta Data aims to provide a consistent experience across these NoSQL database types. This section covers the key portability aspects for four categories of NoSQL databases: key-value, wide-column, document, and graph databases.

8.2.1. Key-value databases

Key-value databases resemble dictionaries or Maps in Java, where data is primarily accessed using a key. In such databases, queries unrelated to keys are typically limited. To ensure a minimum level of support, Jakarta Data mandates the implementation of `BasicRepository` built-in methods that require an identifier or a key. However, the deleteAll and count methods are not required. Methods relying on complex queries, which are defined as queries that do not use the Key or identifier, raise `java.lang.UnsupportedOperationException` due to the fundamental nature of key-value databases.

> ❗ For any NoSQL database type not covered here, such as time series databases, the Key-value support serves as the minimum required level of compatibility.

8.2.2. Wide-column databases

Wide-column databases offer more query flexibility, even allowing the use of secondary indexes, albeit potentially impacting performance. When interacting with wide-column databases, Jakarta Data requires the implementation of the `BasicRepository` along with all of its methods. However, developers should be mindful that certain query keywords, such as "And" or "Or," may not be universally supported in these databases. The full set of required keywords is documented in the section of the Jakarta Data module Javadoc that is titled "Reserved Keywords for Query by Method Name".

8.2.3. Document databases

Document databases provide query flexibility akin to relational databases, offering robust query capabilities. They encourage denormalization for performance optimization. When interfacing with document databases, Jakarta Data goes a step further by supporting both built-in repositories: `BasicRepository` and `CrudRepository`. Additionally, Query by Method Name is available, though developers should be aware that some keywords may not be universally supported. The full set of required keywords is documented in the section of the Jakarta Data module Javadoc that is titled "Reserved Keywords for Query by Method Name".

These portability considerations reflect Jakarta Data’s commitment to providing a consistent data access experience across diverse NoSQL database types. While specific capabilities and query support may vary, Jakarta Data aims to simplify data access, promoting flexibility and compatibility in NoSQL database interactions.

8.2.4. Graph databases

A Graph database, a specialized NoSQL variant, excels in managing intricate data relationships, rivaling traditional relational databases. Its unique strength lies in its ability to handle both directed and undirected edges (or relationships) between vertices (or nodes) and store entity attributes on both vertices and edges.

Graph databases excel at answering queries that return rows containing flat objects, collections, or a combination of flat objects and connections. However, portability is only guaranteed when mapping rows to classes, and when queries specified via annotations or other supported means are used. It should be noted that queries derived from keywords and combinations of mapped classes/attributes will be translated into vendor-specific queries.

It is important to note that in Jakarta Data the Graph database supports the built-in repositories: `BasicRepository` and `DataRepository`. Additionally, Query by Method Name is available, though developers should be aware that some keywords may not be universally supported. The full set of required keywords is documented in the section of the Jakarta Data module Javadoc that is titled "Reserved Keywords for Query by Method Name".