JAKARTA EE

Jakarta Data

1.0.0-M1, October 26, 2023: Draft

Table of Contents

Specification: Jakarta Data

Version: 1.0.0-M1

Status: Draft

Release: October 26, 2023

Copyright

Eclipse Foundation Specification License

Disclaimers

Jakarta Data

Chapter 1. Introduction

The Jakarta Data specification provides an API for easier data access. A Java developer can split the persistence from the model with several features, such as the ability to compose custom query methods on a Repository interface where the framework will implement it.

There is no doubt about the importance of data within the context of an application. We often discuss the concept of a stateless application, where we delegate the management of the application's state to the database.

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's essential 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 is a pivotal solution that addresses a fundamental challenge in Java application development: the seamless integration of diverse data sources. This integration is crucial for applications with many 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—some relational, some NoSQL, and others unique. 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, a driver, 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 among 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

In the development of any software component, important decisions are made to strike a balance between various considerations. It's equally important to define what Jakarta Data does not aim to achieve—its non-goals:

1. **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 its capabilities but aims to simplify data access and integration across diverse data sources. The specialized features, specializations, and extensions that address particular behaviors and use cases are crucial within the Jakarta ecosystem but distinct from Jakarta Data's core API. Jakarta Data's core primarily provides a persistence-agnostic data access and integration framework. It establishes a consistent foundation that abstracts the complexities of data sources and storage engines, offering developers a unified approach.
2. **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."

These clarifications emphasize Jakarta Data's role as a complementary and simplifying layer within the Jakarta ecosystem, highlighting that it does not aim to replicate existing functionality or replace established specifications.

1.3. Conventions

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

- Denis Stepanov
- Dmitry Kornilov
- Emily Jiang
- Graeme Rocher
- James Krueger
- James Stephens
- Michael Redlich
- Nathan Rauh
- Otavio Santana
- Werner Keil

1.4.3. Mentor

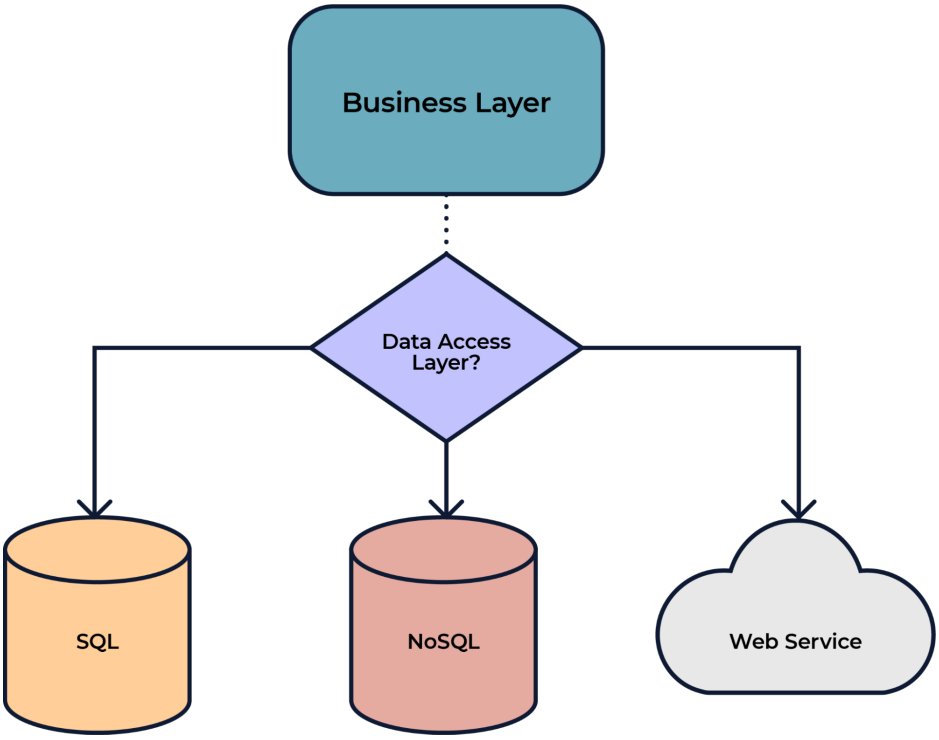
- Dmitry Kornilov

1.4.4. Contributors

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

Chapter 2. Repository

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 your 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 your data. It abstracts data storage and retrieval complexities, allowing you to work with domain-specific objects and perform everyday data operations 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 your Java applications. These characteristics collectively define how repositories function within Jakarta Data, providing a structured and domain-centric approach to working with data. By understanding these key characteristics, you'll gain insight into how repositories simplify data access and enhance the maintainability of your code.

- **Abstraction:** Repositories abstract the details of how data is stored, enabling you to focus on your 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 your codebase remains organized and maintainable.
- **Domain-Centric:** Repositories are designed to be domain-centric, aligning with your

application's domain model. It means that data access operations are closely tied to your business entities, making your 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 your 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 on Jakarta Data

Within the context of Jakarta Data, a repository plays a pivotal role in simplifying data access layers for various persistence stores. It 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 persistent instances of the entities they are associated with.

Several vital characteristics define repositories:

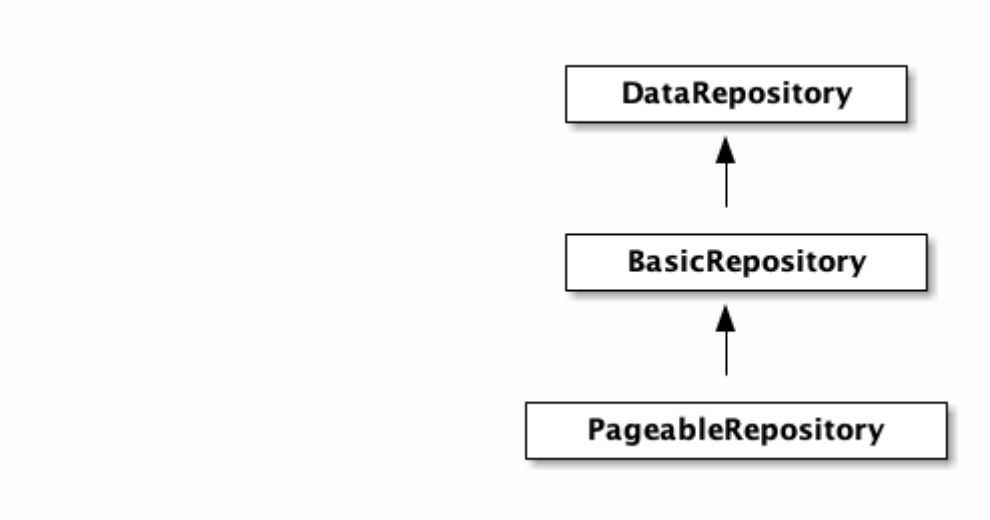
- **Reduced Boilerplate Code:** One of the primary goals of a repository abstraction is to significantly reduce the boilerplate code required to implement data access layers for diverse persistence stores. This reduction in repetitive code enhances code maintainability and developer productivity.
- **Jakarta 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 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 Repositories:** Repositories are typically designed to be stateless. However, it's important to note that this specification does not address the definition of repositories backed by Jakarta Persistence-style stateful 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 Jakarta Data specification supports two types of repositories.

The first type consists of built-in interfaces that are parent interfaces from which repositories can inherit. At the root of this hierarchy is the `DataRepository` interface. These built-in interfaces are extensible, meaning a repository can extend one or more of them or none at all. When a repository extends a built-in interface, the method signatures copied from the built-in interfaces must retain the same behavior as defined in the built-in interfaces.



- Interface with basic operations on a repository for a specific type. This one we can see more often on several Java implementations.
- Interface with basic operations using the pagination feature.

From the Java developer perspective, create an interface that is annotated with the `@Repository` annotation and optionally extends one of the built-in repository interfaces.

So, given a `Product` entity where the ID is a `long` type, the repository would be:

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

}
```

There is no nomenclature restriction to make mandatory the `Repository` suffix. Such as, you might represent the repository of the Car's entity as a `Garage` instead of `CarRepository`.

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

}
```

Jakarta Data empowers developers to take control of their data access and management by providing the flexibility to define two essential components:

1. **Entity Classes and Mappings:** Developers can 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.
2. **Repository Interfaces:** Jakarta Data encourages 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.

Subsequently, 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.

Additionally, Jakarta Data allows for custom interfaces that do not extend any built-in interfaces. These non-built-in interfaces enable developers to define their repository structures and behavior.

Non-built-in interfaces play a pivotal role in Jakarta Data, offering a powerful mechanism for creating custom repository interfaces that align seamlessly with your specific domain requirements. These custom interfaces provide a means to define your domain's ubiquitous language precisely.

In this context, database operations involving fundamental data changes, such as insertion, update, and removal, are realized through the strategic utilization of annotations like `Insert`, `Update`, `Delete`, and `Save`. These annotations enable the crafting of expressive and contextually meaningful repository methods, resulting in a repository that closely mirrors the semantics of your domain.

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

```
@Repository
public interface Garage {

    @Insert
    Car park(Car car);

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

Here, the `@Insert` annotation is used for the `park` method, allowing you to design a repository interface that encapsulates the essence of your domain. This approach fosters a shared understanding and more intuitive communication within your development team, ensuring that your database operations are integral to your domain's language.

2.2. Repository interfaces

A Jakarta Data repository is a Java interface annotated `@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*, 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 Jakarta Data implementation must treat inherited abstract methods as if they were directly declared by the repository interface. For example, a repository interface may inherit the `CrudRepository` interface defined by this specification.

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 `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 `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. 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 programs 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.

2.2.1. 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 methods follow the general pattern:

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

where `lifecycle` is the arbitrary name of the method, `Entity` is a concrete entity class or an `Iterable` or array of this entity, `Lifecycle` is a lifecycle annotation, and `ReturnType` is a return type that is permitted by the lifecycle annotation JavaDoc.

This specification defines four built-in lifecycle annotations: `@Insert`, `@Update`, `@Delete`, and `@Save`.

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.

2.2.2. 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`, `Pageable`, 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 in an arbitrary query language understood by the Jakarta Data provider.

For example, using a named parameter:

```
@Query("where title like :title order by title")
Page<Book> booksByTitle(String title, Pageable page);
```

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 portable between providers.

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.

2.2.3. Automatic query methods

An *automatic query method* is an abstract method that either follows the Query by Method Name pattern where the Jakarta Data provider generates a query based on the name of the method, or follows a pattern for whereby the Jakarta Data provider automatically generates a find query based on the names of parameters that are supplied to the method, where the method has return type that identifies the entity, such as `E`, `Optional<E>`, `Page<E>`, or `List<E>`, where `E` is an entity class. Each parameter must either:

- have exactly the same type and name as a persistent field or property of the entity class, or
- be of type `Limit`, `Pageable`, or `Sort`.

A repository with automatic query methods that are based on parameters must either 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.

For example:

```
Book bookByIsbn(String isbn);

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

Automatic query methods *are* portable between providers.

2.2.4. Resource accessor method

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.

2.2.5. Additional examples

The following examples demonstrate the use of annotated and automatic query methods with `CrudRepository`.

```
@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {
  @Query("SELECT p FROM Product p WHERE p.name=?1")  // example in JPQL
  Optional<Product> findByName(String name);
}
```

```
@Repository
public interface ProductRepository extends CrudRepository<Product, Long> {
  @Query("SELECT p FROM Product p WHERE p.name=:name")  // example in JPQL
  Optional<Product> findByName(@Param("name") String name);
```

```
}
```

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

  // Assumes that the Product entity has attributes: yearProduced
  List<Product> findMadeIn(int yearProduced, Sort... sorts);

  @Query("SELECT count(p) FROM Product p WHERE p.name=?1 AND p.status=?2")
  int countWithStatus(String name, Status status);

  @Query("DELETE FROM Product p WHERE p.yearProduced=?1")
  void deleteOutdated(int yearProduced);
}
```

2.3. Entity Classes

In Jakarta Data, an entity refers to a fundamental data representation and management building block. It can be conceptually understood in several aspects:

1. **Entity Classes**: Entity classes are simple Java objects equipped with fields or accessor methods that designate each property of the entity. Depending on your data storage needs, you may use annotations from the Jakarta Persistence specification, such as `jakarta.persistence.Entity`, `jakarta.persistence.Id`, and `jakarta.persistence.Column`, to define and customize entities for relational databases. Alternatively, for NoSQL databases, you can use annotations from the Jakarta NoSQL specification, including `jakarta.nosql.Entity`, `jakarta.nosql.Id`, and `jakarta.nosql.Column`.
2. **Data Schema**: Abstractly, an entity or entity type serves as a schema for data. It defines the structure and properties of the data it represents. This schema can be as simple as a set of typed fields, similar to the relational model, or more structured, as found in document data stores. The schema can be explicitly defined, as in the case of SQL Data Definition Language (DDL) declarations for relational tables, or it can be implicit, common in key/value stores.
3. **Persistence and Representation**: Entities are associated with persistent data, meaning the data outlives any specific Java process utilizing it. 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. In Java, these entities are represented as classes, referred to as entity classes. It's important to note that multiple instances of the entity class within a Java program can represent a single persistent instance of the schema.
4. **Provider Differentiation**: To maintain clarity and specify the desired provider when using Jakarta Data, it's recommended that applications do not mix Entity annotations from different models. This practice allows the Entity annotation to indicate the desired provider, especially in cases where multiple types of Jakarta Data providers are available. A Jakarta Data provider must provide implementation of repositories for Entity types having the Entity annotations that it supports, ignoring Entity types only having identifiable Entity annotations that the Jakarta Data provider does not support. The latter are to be handled by other Jakarta Data providers

that do support the other types of Entity annotations.

An entity within Jakarta Data encompasses the Java class representing the data and the schema, persistence characteristics, and provider-specific annotations, all working together to simplify data access and management within Java applications. While a Jakarta Data provider might require a default constructor and work primarily with mutable entities, Jakarta Data allows for the use of immutable entity classes, which are best represented as Java records.

2.3.1. Programming Model for Entity Data in Jakarta Data

Jakarta Data does not define an entity model of its own, instead borrowing the entity models of other Jakarta standards (Jakarta Persistence and Jakarta NoSQL) and allowing for vendor-specific entity models to be used. This section defines core concepts that entity models must follow in order to be used with Jakarta Data. The entity model allows the user to define Java classes that represent data entities. These entities can be stored, retrieved, and manipulated in various databases, including key-value, wide-column, document, graph, and relational databases. A programming model for entities ensures that entity classes are well-defined and can be seamlessly integrated with different database technologies.

Jakarta Data places requirements on two types of fields within entity classes: basic fields and relation fields. Basic fields represent fundamental data types natively supported by Jakarta Data Providers. Support for basic types is mandatory for all Jakarta Data providers. On the other hand, Domain-Relation fields allow entities to interact with other domain classes or types, enriching data structures, but also making them more complex. Support for Domain-Relation fields varies depending on the Jakarta Data provider and the database type. Jakarta Data does not require support for Domain-Relation entity fields when using Graph databases.

2.3.1.1. Basic Types

A variety of basic types can be used for fields or properties of entity classes. The basic types include:

Basic Data Type	Description
Primitive Types and Wrappers	All Java primitive types, such as int, double, boolean, etc., and their corresponding java.lang wrapper types (e.g., Integer, Double, Boolean).
String	Represents text data.
LocalDate, LocalDateTime, LocalTime, Instant	Represent date and time-related data.
UUID	Universally Unique Identifier for identifying entities.
BigInteger, BigDecimal	Represent large integer and decimal numbers.
byte[]	Represents binary data.
Enum Types	Custom enumerated types defined by developers.

Every entity in Jakarta Data must have a unique identifier composed of one or more supported

basic types. This unique identifier is crucial for distinguishing individual entities in the database. Entity models that are used with Jakarta Data must define a way for developers to specify the unique identifier. Typically this is done with an `@Id` annotation, but other means are permitted, such as `@EmbeddedId` in Jakarta Persistence which defines a compound unique identifier based on an embeddable class, or by naming convention (for example, considering a property to be the unique identifier if it is named id or ends in Id).

It's important to note that key-value, wide-column, document, and relational databases support Collection specializations and Maps of the basic types. However, these databases may have different serialization processes, impacting performance and causing impedance mismatch. Developers should consider these side effects when working with collections and maps in their entity models.

In addition to the basic types, entity models might also choose to support additional types that represent data in a domain-specific manner. These custom types may include complex data structures and objects. Entity models can choose to provide mechanisms to convert these custom types to the supported basic types.

2.3.1.2. Domain-Relation Fields in Jakarta Data

In Jakarta Data, the concept of "Domain-Relation" fields encompasses two distinct types: "component" and "association" fields. These fields enable developers to establish relationships between entities and other domain concepts, enriching the complexity and structure of data entities.

- **Component Fields**: A component field represents a relationship where one entity is treated as a component of another entity. It implies that the component does not have a life cycle outside the entity to which it belongs. It is an embedded object that exists solely within the context of the owning entity.
- **Association Fields**: Association fields represent semantically weak relationships, often called semantic dependencies, between objects that may not have a direct or strong connection. Unlike component fields, association fields may exist between entities that are otherwise unrelated. Associations can further specialize into aggregation, a specific form of association in which each object involved has its life cycle, yet a notion of ownership also exists.

Definition and implementation of Domain-Relation fields may vary across different Jakarta Data providers. Providers can create annotations, define conventions, or leverage standards like Jakarta Persistence to establish these relationships effectively.

The topic of serialization of Domain-Relation fields within databases is crucial for maintaining data consistency and integrity. The next section discussed how Jakarta Data providers handle the persistence and serialization of Domain-Relation fields for various database types.

In Jakarta Data, the serialization of Domain-Relation fields, such as components, can be achieved in two ways.

1. **Merging Fields Directly in the Entity (Component Embedding):**

In this approach, a component merges its fields directly within the entity. From the perspective of

the persistence layer, the component itself doesn't exist as a separate table or document. Instead, it becomes part of the entity's structure. This approach leads to a flat representation in the database.

Example:

```
public class Address {

    private String street;
    private String city;
    private String postalCode;
}

//the entity
public class Person {
    private Long id;

    private String name;
    private Address address;  // This is a component field
}
```

The structure of the entity in a Document, Wide-Column, and Graph database will be like the JSON representation below. The JSON representation is for illustration; the actual representation is an implementation detail of the database. The JSON and the following SQL table are representations resulting from the Java classes.

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

id	name	street	city	postalCode
1	John Doe	123 Main St	Sampleville	12345

This approach allows for a flat and denormalized structure in the database, making it suitable for a variety of database types.

1. * Second Approach: Storing Components in Separate Tables (Relational Databases) or as Subdocuments/UDTs (NoSQL Databases)*

This approach is typically employed for more complex relationships and associations within the domain model, allowing for greater flexibility and scalability. It involves storing components, such as the `Passport` in the example, in separate tables for relational databases or as subdocuments or User-Defined Types (UDTs) for NoSQL databases. This method is suitable for scenarios where an association exists between one entity and another.

For instance, consider the `Citizen` and `Passport` classes. In a relational database, this approach results in two separate tables, each representing an entity with its associated persistence context. In contrast, for a NoSQL database like a Document database, the `Passport` can still act as a component within the `Citizen` entity, signifying that a `Passport` is closely tied to a `Citizen`. However, the modeling may vary depending on the specific NoSQL database and its capabilities.

```
public class Passport {
    private Long id;
    private String passportNumber;
    private LocalDate expirationDate;
}

public class Citizen {
    private Long id;
    private String name;
    private Passport passport; // One-to-One relationship with Passport for relational
database
}
```

Here are some possible representations of the `Citizen` and `Passport` entities in table and JSON formats:

Table 1. Citizen Table:

id (Primary Key)	name	passport_id (Foreign Key)
1	John Doe	1

Table 2. Passport Table:

id (Primary Key)	passportNumber	expirationDate
1	A123456	2023-12-31

These tables represent the data from the JSON structure you provided earlier. In a relational database, the `Citizen` and `Passport` entities are stored in separate tables, connected by a foreign key relationship.

```
{
  "id": 1,
  "name": "John Doe",
  "passport": {
    "passportNumber": "A123456",
    "expirationDate": "2023-12-31"
  }
}
```

Entities in a domain model often have relationships with other entities. In some cases, an entity may have a collection of another type of entity, creating a one-to-many or many-to-many relationship. This scenario explores how such relationships are represented and managed in the

context of Jakarta Data.

Consider the example of an `Author` entity associated with multiple `Book` entities. This relationship allows an author to be linked to multiple books they have authored. While this structure remains relatively unchanged for NoSQL databases, it introduces specific considerations in relational databases, where it typically generates auxiliary tables to manage the relationship. We'll explore these representations in both JSON and relational database formats.

```
// Entity
public class Author {

    private UUID id;

    private String name;

    private List<Book> books;
}

// If in a relational database, Book might also be an entity
public class Book {

    // This field might not be required for some NoSQL database modeling
    private Long id;

    private String title;

    private String category;

    // Other fields and methods
}
```

```
{
  "id": "6f6d665d-5585-46cd-8b9b-61a559de0e13",
  "name": "John Doe",
  "books": [
    {
      "title": "Sample Book 1",
      "category": "Fiction"
    },
    {
      "title": "Sample Book 2",
      "category": "Non-fiction"
    }
  ]
}
```

In the JSON representation, an `Author` entity can be associated with multiple `Book` entities within an array.

Here are the tables with content based on the JSON data structure:

Table 3. Author Table:

id (Primary Key)	name
1	John Doe
2	Jane Smith

Table 4. Book Table:

id (Primary Key)	title	category
1	Sample Book 1	Fiction
2	Sample Book 2	Non-fiction

Table 5. Author_Book (Auxiliary) Table:

author_id (Foreign Key)	book_id (Foreign Key)
1	1
1	2
2	2

These tables represent the data from the JSON structure you provided earlier, illustrating a many-to-many relationship between authors and books using an auxiliary table.

In a relational database, this relationship typically generates three tables: `Author`, `Book`, and an auxiliary table `Author_Book` to manage the many-to-many relationship between authors and books.

In some scenarios, books can have multiple authors, and authors can contribute to several books, resulting in a many-to-many (N-N) cardinality relationship. Jakarta Data offers flexibility in representing and managing such complex relationships.

This N-N relationship typically generates a dedicated table to manage the associations between books and authors in a relational database. However, NoSQL databases may take a different approach, especially in cases where denormalization and data duplication are favored for query-driven designs.

The key-value database might support these fields, generating a single BLOB value. However, the serialization process for such fields may vary depending on the Jakarta Data provider and the specific key-value database used.

The Graph database is not required to support Domain-Relation types, but it might be used for aggregate query returns or as a read-only field.

2.3.1.3. Recursion at Domain-Relation Fields in Jakarta Data

In the context of Jakarta Data, the term "recursion" pertains to the ability to manage hierarchical or nested relationships between entities. This capability is essential when dealing with complex domain models involving associations, aggregations, or compositions between various entity types.

Jakarta Data ensures these relationships are correctly mapped and maintained within the database, enabling consistent data retrieval and manipulation.

For relational databases, Jakarta Data requires support of recursive relationships. If one entity type is associated with or contains another entity type, the Jakarta Data provider for relational databases must establish the required table structures and foreign key constraints to uphold these relationships. This approach guarantees data integrity and consistency when working with the database.

For other types of databases, Jakarta Data does not require explicit support for recursive relationships. In NoSQL databases, data is often stored in a denormalized or nested document format, making it more challenging to enforce strict hierarchical relationships. Instead, NoSQL databases may emphasize query-driven design rather than explicit mapping or management of recursive relationships.

In instances where a Jakarta Data provider for NoSQL databases encounters a recursive relationship that it cannot support due to the specific characteristics of the database, it must throw a `jakarta.data.exceptions.MappingException` or an appropriate subclass of `MappingException`. This exception notifies developers that the database does not support the relationship.

2.3.2. 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 can choose to define a separate class, referred to as the metamodel class, following a prescribed set of conventions. The application developer annotates the metamodel class with `@StaticMetamodel`, specifying the entity class as its `value`. The metamodel class must contain one or more fields of type `jakarta.data.model.Attribute` with modifiers `public`, `static`, `final`, with each field named after an entity attribute. The application sets the value of each field to an uninitialized instance obtained from the `Attribute.get` method. A Jakarta Data provider that provides a repository for the entity class must initialize each `Attribute` field for which the field name corresponds to an entity attribute name.

2.3.2.1. Application Requirements for a Metamodel Class

For each entity class for which the application wishes to access the metamodel,

- The application defines a class (the metamodel class) and annotates it with the `@StaticMetamodel` annotation.
- The application specifies the `value` of the `@StaticMetamodel` annotation to be an entity class that the application uses in a repository as the result type of a find method or the parameter type of an insert, update, save, or delete method.

For each field of the metamodel class,

- The field type must be `jakarta.data.model.Attribute`.
- The field must have the `public` modifier.
- The field must have the `static` modifier.

- The field must have the `final` modifier.
- The name of the field, ignoring case, must match the name of an entity attribute, with the `_` character in the field name delimiting the attribute names of hierarchical structures or relationships, such as embedded classes.
- The value of the field must be a unique instance obtained from the `Attribute.get` method.

If the entity class has a unique identifier that is represented by a single entity attribute, then the application can also include a field that follows the above criteria, except that it has a field name of `id`.

The application is not required to include fields for all entity attributes.

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())` rather than `Sort.asc("title")`.

If the application defines repositories for the same entity class across multiple Jakarta Data providers, no guarantee is made of the order in which the fields of the metamodel class are assigned by the Jakarta Data providers.

2.3.2.2. Jakarta Data Provider Requirements for a Metamodel Class

The Jakarta Data provider observes classes that are annotated with the `@StaticMetamodel` annotation. If the `value` of the `@StaticMetamodel` annotation is an entity class for which the Jakarta Data provides a repository implementation, then the Jakarta Data provider must initialize, via the `Attribute.init` method, the value of each field meeting the criteria that is defined in the "Application Requirements for a Metamodel Class" above with the corresponding entity attribute name. The value of the field that is named `id`, if present, must be initialized by the Jakarta Data provider with the attribute information for the unique identifier if the unique identifier is a single entity attribute.

2.3.2.3. 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 class Product_ {
  public static final Attribute id = Attribute.get();
  public static final Attribute name = Attribute.get();
```

```
  public static final Attribute price = Attribute.get();
}
```

Example usage:

```
List<Product> found = products.findByNameContains(searchPattern,
                                                  Product_.price.desc(),
                                                  Product_.name.asc(),
                                                  Product_.id.asc());
```

2.4. Query by Method Name

The Query by method mechanism allows for creating query commands by naming convention.

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

  List<Product> findByName(String name);

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

  @OrderBy(value = "price", descending = true)
  List<Product> findByNameLikeAndPriceLessThan(String namePattern, float priceBelow);

}
```

The parsing of query method names follows a specific format:

- The method name consists of the subject, the predicate, and optionally the order clause.
- The subject, defines the action (such as `find` or `delete`) , optionally followed by an expression (for example, `First10`), followed by `By`
- The predicate defines the query's condition or filtering criteria, where multiple conditions are delimited by `And` or `Or`. For example, `PriceLessThanAndNameLike`.
- The order clause, which is optional, begins with `OrderBy` and consists of an ordered collection of entity attributes by which to sort results, delimited by `Asc` or `Desc` to specify the sort direction of the preceding attribute.
- The method name is formed by combining the subject, predicate, and order clause, in that order.

Queries can also handle entities with related classes by specifying the relationship using dot notation.

Example query methods:

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

2.4.1. BNF Grammar for Query Methods

Query methods allow developers to create database queries using method naming conventions. These methods consist of a subject, predicate, and optional order clause. This BNF notation provides a structured representation for understanding and implementing these powerful querying techniques in your applications.

```
<query-method> ::= <subject> <predicate> [<order-clause>]
<subject> ::= (<action> | "find" <find-expression>) "By"
<action> ::= "find" | "delete" | "update" | "count" | "exists"
<find-expression> ::= "First" [<positive-integer>]
<predicate> ::= <condition> { ("And" | "Or") <condition> }
<condition> ::= <property> ["IgnoreCase"] ["Not"] [<operator>]
<operator> ::= "Contains" | "EndsWith" | "StartsWith" | "LessThan"| "LessThanEqual" |
"GreaterThan" | "GreaterThanEqual" | "Between" | "Empty" | "Like" | "In" | "Null" |
"True" | "False"
<property> ::= <identifier> | <identifier> "_" <property>
<identifier> ::= <word>
<positive-integer> ::= <digit> { <digit> }
<order-clause> ::= "OrderBy" { <order-item> } ( <order-item> | <property> )
<order-item> ::= <property> ("Asc" | "Desc")
```

Explanation of the BNF elements:

- `<query-method>`: Represents a query method, which consists of a subject, a predicate, and an optional order clause.
- `<subject>`: Defines the action (e.g., "find" or "delete") followed by an optional expression and "By."
- `<action>`: Specifies the action, such as "find" or "delete."
- `<find-expression>`: Represents an optional expression for find operations, such as "First10."
- `<predicate>`: Represents the query's condition or filtering criteria, which can include multiple conditions separated by "And" or "Or."
- `<condition>`: Specifies a property and an operator for the condition.
- `<operator>`: Defines the operator for the condition, like "Between" or "LessThan."
- `<property>`: Represents a property name, which can include underscores for nested properties.
- `<identifier>`: Represents a word (e.g., property names, action names, etc.).

- `<positive-integer>`: Represents a whole number greater than zero.
- `<order-clause>`: Specifies the optional order clause, starting with "OrderBy" and followed by one or more order items.
- `<order-item>`: Represents an ordered collection of entity attributes by which to sort results, including an optional "Asc" or "Desc" to specify the sort direction.

2.4.2. Entity Property Names

Within an entity, property names must be unique ignoring case. For simple entity properties, the field or accessor method name serves as the entity property name. In the case of embedded classes, entity property names are computed by concatenating the field or accessor method names at each level.

Assume an Order entity has an Address with a ZipCode. In that case, the access is `order.address.zipCode`. This form is used within annotations, such as `@Query`.

```
@Repository
public interface OrderRepository extends BasicRepository<Order, Long> {

  @Query("SELECT order FROM Order order WHERE order.address.zipCode=?1")
  List<Order> withZipCode(ZipCode zipCode);

}
```

The resolution algorithm for identifying properties in query methods by method name, with manual traversal points, is defined as follows:

1. **Method Name Parsing**:: The query method's name is parsed to identify the property or properties being referenced. Method names in query methods typically follow a pattern of "findBy[Property]", where "[Property]" represents the name of the property you want to query by.
2. **Property Extraction**:: The property name is extracted from the method name by removing the "findBy" prefix. For example, in the query method `findByAddressZipCode`, the property name extracted is `AddressZipCode`.
3. **Property Name Capitalization**:: The extracted property name is treated as is, with its original capitalization. For example, if the property name is `AddressZipCode`, it remains in camel case.
4. **Manual Traversal Points**:: To resolve ambiguity or to specify traversal through nested properties, underscores (`_`) can be used within the method name. Each underscore represents a traversal point to access nested properties. For example, `findByAddress_ZipCode` explicitly indicates traversal to the `Address` object's `ZipCode` property.
5. **Domain Class Property Lookup**: The framework checks the domain class associated with the repository for a property with the same name as the extracted property name (uncapitalized) in a case-insensitive manner. If the domain class has a property named `addressZipCode` or `addresszipcode`, this is considered a successful resolution.
6. **Nested Property Handling**:: If the extracted property name includes underscores (`_`) indicating

nested traversal, the framework follows the specified path to resolve the property.

7. **Resolution Outcome**:: If the framework successfully identifies a property in the domain class or along the specified traversal path that matches the extracted property name, it uses that property in the query to filter data.

Users are encouraged to follow Java's naming standards in formalizing Jakarta Data queries using name conventions, avoiding underscores in field names. The resolution algorithm for property identification relies on "findBy[Property]" naming, allowing manual traversal with underscores. Adhering to the camel case for property names ensures consistency and seamless query method naming in Jakarta Data, enabling effective data filtering and retrieval from domain classes.

Scenario 1: Person Repository with findByAddressZipCode(ZipCode zipCode)

In this scenario, we have the following data models:

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

class Address {
  private Zipcode zipcode;
}
```

- The query method `findByAddressZipCode` takes a `ZipCode` object as a parameter.
- The Property Resolution Algorithm will parse the method name and extract `AddressZipCode`.
- It will then attempt to resolve the property named `addressZipCode` in the `Person` class, following automatic class splitting by camel case.
- Since the `Person` class has an `Address` property, it will recursively follow the path to the `Address` class.
- In the `Address` class, it will identify the `zipcode` property and filter `Person` records based on the provided `Zipcode` object within the `Address` object.

Scenario 2: People Repository with findByAddressZipCode(String addressZipCode)

In this scenario, we have the following data model:

```
class Person {
  private Long id;
  private String addressZipCode;
}
```

- The query method `findByAddressZipCode` takes a `String` parameter named `addressZipCode`.
- The Property Resolution Algorithm will parse the method name and extract `AddressZipCode`.
- It will then attempt to resolve the property named `addressZipCode` in the `Person` class, following

automatic class splitting by camel case.

- If a property named `addressZipCode` of type `String` exists in the `Person` class or its nested objects, the query will filter `Person` records based on the provided `addressZipCode` string.

Scenario 3: OrderRepository` Repository with `findByAddress_ZipCode(ZipCode zipCode)

In this scenario, we have the following data models:

```
class Order {
  private Long id;
  private String addressZipCode;
  private Address address;
}

class Address {
  private Zipcode zipcode;
}
```

- The query method `findByAddress_ZipCode` takes a `Zipcode` object as a parameter.
- The method name includes an underscore (_) indicating manual traversal points.
- The Property Resolution Algorithm will parse the method name and extract `Address_ZipCode`, recognizing the underscore as a traversal point.
- It will then attempt to resolve the property named `Address` within the `Order` class, followed by the `zipcode` property within the `Address` class, following manual traversal points.
- If properties `Address` and `ZipCode` are found in the appropriate classes or their nested objects, the query will filter `Order` records based on the provided `Zipcode` object within the `Address` object.

Scenario 4: People Repository with findByAddressZipCode(String addressZipCode)

In this scenario, we have the following data model:

```
class Person {
  private Long id;
  private String addressZipcode;
}
```

- The query method `findByAddressZipCode` takes a `String` parameter named `addressZipCode`.
- The Property Resolution Algorithm will parse the method name and extract `AddressZipCode`.
- It will then attempt to resolve the property named `addressZipcode` in the `Person` class, following automatic class splitting by case-insensitive.
- If a property named `addressZipCode` of type `String` exists in the `Person` class or its nested objects, the query will filter `Person` records based on the provided `addressZipCode` string.

Define as a priority following standard Java naming conventions, camel case,

> using underscore as the last resort.

In queries by method name, `Id` is an alias for the entity property that is designated as the id. Entity property names that are used in queries by method name must not contain reserved words.

2.4.3. Query by Method Name Keywords

The following table lists the query-by-method keywords that must be supported by Jakarta Data providers, except where explicitly indicated for a type of database.

Keyword	Description	Not Required For
findBy	General query method returning the repository type.	Key-value, Wide-Column
deleteBy	Delete query method returning either no result (void) or the delete count.	Key-value, Wide-Column
countBy	Count projection returning a numeric result.	Key-value, Wide-Column
existsBy	Exists projection, returning as a `boolean` result.	Key-value, Wide-Column

Note

- The "Not Required For" column indicates the database types for which the respective keyword is not required or applicable.

Jakarta Data implementations must support the following list of query-by-method keywords, except where indicated for a database type. A repository method must raise `java.lang.UnsupportedOperationException` or a more specific subclass of the exception if the database does not provide the requested functionality.

Keyword	Description	Method signature Sample	Not Required For
And	The `and` operator.	findByNameAndYear	Key-value, Wide-Column
Or	The `or` operator.	findByNameOrYear	Key-value, Wide-Column
Not	Negates the condition that immediately follows the `Not` keyword. When used without a subsequent keyword, means not equal to.	findByNameNotLike	Key-value, Wide-Column

Keyword	Description	Method signature Sample	Not Required For
Between	Find results where the property is between the given values	findByDateBetween	Key-value, Wide-Column
Contains	For Collection attributes, matches if the collection includes the value. For String attributes, a substring of the String must match the value, which can be a pattern.	findByPhoneNumbersContains	Key-value, Wide-Column, Document
Empty	Find results where the property is an empty collection or has a null value.	deleteByPendingTasksEmpty	Key-value, Wide-Column, Document, Graph
EndsWith	Matches String values with the given ending, which can be a pattern.	findByProductNameEndsWith	Key-value, Wide-Column, Document, Graph
First	For a query with ordered results, limits the quantity of results to the number following First, or if there is no subsequent number, to a single result.	findFirst10By	Key-value, Wide-Column, Document, Graph
LessThan	Find results where the property is less than the given value	findByAgeLessThan	Key-value, Wide-Column
GreaterThan	Find results where the property is greater than the given value	findByAgeGreaterThan	Key-value, Wide-Column
LessThanEqual	Find results where the property is less than or equal to the given value	findByAgeLessThanEqual	Key-value, Wide-Column
GreaterThanEqual	Find results where the property is greater than or equal to the given value	findByAgeGreaterThanEqual	Key-value, Wide-Column

Keyword	Description	Method signature Sample	Not Required For
Like	Matches String values against the given pattern.	findByTitleLike	Key-value, Wide-Column, Document, Graph
IgnoreCase	Requests that string values be compared independent of case for query conditions and ordering.	findByStreetNameIgnoreCaseLike	Key-value, Wide-Column, Document, Graph
In	Find results where the property is one of the values that are contained within the given list	findByIdIn	Key-value, Wide-Column, Document, Graph
Null	Finds results where the property has a null value.	findByYearRetiredNull	Key-value, Wide-Column, Document, Graph
StartsWith	Matches String values with the given beginning, which can be a pattern.	findByFirstNameStartsWith	Key-value, Wide-Column, Document, Graph
True	Finds results where the property has a boolean value of true.	findBySalariedTrue	Key-value, Wide-Column
False	Finds results where the property has a boolean value of false.	findByCompletedFalse	Key-value, Wide-Column
OrderBy	Specify a static sorting order followed by the property path and direction of ascending.	findByNameOrderByAge	Key-value, Wide-Column
OrderBy___Desc	Specify a static sorting order followed by the property path and direction of descending.	findByNameOrderByAgeDesc	Key-value, Wide-Column
OrderBy___Asc	Specify a static sorting order followed by the property path and direction of ascending.	findByNameOrderByAgeAsc	Key-value, Wide-Column
OrderBy___(Asc|Desc)*(Asc|Desc)	Specify several static sorting orders	findByNameOrderByAgeAscNameDescYearAsc	Key-value, Wide-Column

Note:

- The "Not Required For" column indicates the database types for which the respective keyword is not required or applicable.

2.4.3.1. Patterns

Wildcard characters for patterns are determined by the data access provider. For relational databases, `_` matches any one character and `%` matches 0 or more characters.

2.4.3.2. Logical Operator Precedence

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

2.5. Special Parameter Handling

Jakarta Data also supports particular parameters to define pagination and sorting.

Jakarta Data recognizes, when specified on a repository method after the query parameters, specific types, like `Limit`, `Pageable`, and `Sort`, to dynamically apply limits, pagination, and sorting to queries. The following example demonstrates these features:

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

  List<Product> findByName(String name, Pageable pageable);

  List<Product> findByNameLike(String pattern, Limit max, Sort... sorts);

}
```

You can define simple sorting expressions by using property names.

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

You can combine sorting with a starting page and maximum page size by using property names.

```
Pageable pageable = Pageable.ofSize(20).page(1).sortBy(Sort.desc("price"));
first20 = products.findByNameLike(name, pageable);
```

Refer to the Jakarta Data module JavaDoc section on "Return Types for Repository Methods" for a listing of valid return types for methods with entity parameters.

2.6. 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.

2.6.1. 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.

2.6.2. Static Mechanisms for Sort Criteria

Sort criteria is 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 is evaluated first. When static sort criteria sorts entities to the same position, dynamic sort criteria is applied to further order those entities.

2.6.3. Dynamic Mechanisms for Sort Criteria

Sort criteria is provided dynamically to repository methods either via `Sort` parameters or via a `Pageable` parameter that has one or more `Sort` values. `Sort` and `Pageable` containing `Sort` must not both be provided to the same method.

2.6.4. Examples of Sort Criteria Precedence

The following examples work through scenarios where static and dynamic sort criteria are provided to the same method.

```
// Sorts first by type. When type is the same, applies the Pageable's sort criteria
Page<User> findByNameStartsWithOrderByType(String namePrefix, Pageable pagination);

// Sorts first by type. When type is the same, applies the criteria in the Sorts
List<User> findByNameStartsWithOrderByType(String namePrefix, Sort... sorts);

// Sorts first by age. When age is the same, applies the Pageable's sort criteria
@OrderBy("age")
Page<User> findByNameStartsWith(String namePrefix, Pageable pagination);

// Sorts first by age. When age is the same, applies the criteria in the Sorts
@OrderBy("age")
List<User> findByNameStartsWith(String namePrefix, Sort... sorts);
```

```
// Sorts first by name. When name is the same, applies the Pageable's sort criteria
@Query("SELECT u FROM User u WHERE (u.age > ?1)")
@OrderBy("name")
KeysetAwarePage<User> olderThan(int age, Pageable pagination);
```

2.7. Keyset Pagination

Keyset pagination aims to reduce missed and duplicate results across pages by querying relative to the observed values of entity properties that constitute the sorting criteria. Keyset 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 keyset values automatically when `KeysetAwareSlice` or `KeysetAwarePage` are used as the repository method return type. The application invokes `nextPageable` or `previousPageable` on the keyset aware slice or page to obtain a `Pageable` which keeps track of the keyset values.

For example,

```
@Repository
public interface CustomerRepository extends BasicRepository<Customer, Long> {
  KeysetAwareSlice<Customer> findByZipcodeOrderByLastNameAscFirstNameAscIdAsc(
                                int zipcode, Pageable pageable);
}
```

You can obtain the next page with,

```
for (Pageable p = Pageable.ofSize(50); p != null; ) {
  page = customers.findByZipcodeOrderByLastNameAscFirstNameAscIdAsc(55901, p);
  ...
  p = page.nextPageable();
}
```

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

```
Customer c = ...
Pageable p = Pageable.ofSize(50).afterKeyset(c.lastName, c.firstName, c.id);
page = customers.findByZipcodeOrderByLastNameAscFirstNameAscIdAsc(55902, p);
```

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

- `OrderBy` name pattern of the repository method (as in the examples above) or `@OrderBy` annotation(s) on the repository method.
- `Sort` parameters of the `Pageable` that is supplied to the repository method.

2.7.1. Example of Appending to Queries for Keyset Pagination

Without keyset pagination, a Jakarta Data provider that is based on Jakarta Persistence might compose the following JPQL for the `findByZipcodeOrderByLastNameAscFirstNameAscIdAsc` repository method from the prior example:

```
SELECT o FROM Customer o WHERE (o.zipCode = ?1)
                         ORDER BY o.lastName ASC, o.firstName ASC, o.id ASC
```

When keyset pagination is used, the keyset values from the `Cursor` of the `Pageable` are available as query parameters, allowing the Jakarta Data provider to append additional query conditions. For example,

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

2.7.2. Avoiding Missed and Duplicate Results

Because searching for the next page of results is relative to a last known position, it is possible with keyset 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 keyset 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 properties which are used as the sort criteria, keyset 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, keyset pagination cannot prevent the entity from being missed or possibly appearing a second time due to its changed values.

2.7.3. Restrictions on use of Keyset Pagination

- The repository method signature must return `KeysetAwareSlice` or `KeysetAwarePage`. A repository method with return type of `KeysetAwareSlice` or `KeysetAwarePage` must raise `UnsupportedOperationException` if the database is incapable of keyset pagination.
- The repository method signature must accept a `Pageable` parameter.
- Sort criteria must be provided and should be minimal.

- The combination of provided sort criteria must uniquely identify each entity.
- Page numbers for keyset 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 keyset pagination.
- Page totals and result totals are not accurate for keyset pagination and must not be relied upon.
- A next or previous page can end up being empty. You cannot obtain a next or previous `Pageable` from an empty page because there are no keyset values relative to which to query.
- A repository method that is annotated with `@Query` and performs keyset pagination must omit the `ORDER BY` clause from the provided query and instead must supply the sort criteria via `@OrderBy` annotations or `Sort` parameters of `Pageable`. 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.

2.7.4. Keyset 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("SELECT o FROM Customer o WHERE (o.totalSpent / o.totalPurchases > ?1)")
  KeysetAwareSlice<Customer> withAveragePurchaseAbove(float minimum, Pageable
pagination);
}
```

Example traversal of pages:

```
for (Pageable p = Pageable.ofSize(25).sortBy(Sort.desc("yearBorn"), Sort.asc("name"),
Sort.asc("id")));
     p != null; ) {
  page = customers.withAveragePurchaseAbove(50.0f, p);
  ...
  p = page.nextPageable();
}
```

Chapter 3. Interoperability with other Jakarta EE Specifications

In this section, we will delve into the robust capabilities of Jakarta EE Data and its seamless integration with other Jakarta EE specifications. This integration offers a comprehensive data persistence and management solution in Java applications. When operating within a Jakarta EE product, the availability of other Jakarta EE Technologies depends on the profile. This section outlines how related technologies from other Jakarta EE Specifications work together with Jakarta Data to achieve interoperability.

3.1. Jakarta Contexts and Dependency Injection

Contexts and Dependency Injection (CDI) is a core specification in Jakarta EE that provides a powerful and flexible dependency injection framework for Java applications. CDI lets you decouple components and manage their lifecycle through dependency injection, enabling loose coupling and promoting modular and reusable code.

One of the critical aspects of CDI integration with Jakarta EE Data is the ability to create repository instances using the `@Inject` annotation effortlessly. Repositories are interfaces that define data access and manipulation operations for a specific domain entity. Let's take an example to illustrate this integration:

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

  List<Car> findByType(CarType type);

  Optional<Car> findByName(String name);

}
```

In the above example, we have a `CarRepository` interface that extends the `BasicRepository` interface provided by Jakarta EE Data. The `BasicRepository` interface offers a set of basic operations for entities.

By annotating the `CarRepository` interface with `@Repository`, we indicate that it is a repository component managed by the Jakarta EE Data framework. It allows the Jakarta Data provider to automatically generate the necessary implementations for the defined methods, saving us from writing boilerplate code.

Now, with CDI and the `@Inject` annotation, we can easily inject the CarRepository instance into our code and utilize its methods:

```
@Inject
CarRepository repository;
```

```
// ...

Car ferrari = Car.id(10L).name("Ferrari").type(CarType.SPORT);
repository.save(ferrari);
```

In the above snippet, we inject the `CarRepository` instance into our code using the `@Inject` annotation. Once injected, we can use the repository object to invoke various data access methods provided by the `CarRepository` interface, such as `save()`, `findByType()`, and `findByName()`. This integration between CDI and Jakarta EE Data allows for seamless management of repository instances and enables easy data access and manipulation within your Jakarta EE applications.

3.2. Jakarta Data Providers

Jakarta Data providers implement repository interfaces, performing queries and other operations related to entities per the rules of the Jakarta Data specification. A Jakarta Data provider makes the repository implementation available to the application via dependency injection. The Jakarta Data specification defines the rules by which Jakarta Data providers must abide to ensure that multiple Jakarta Data providers are able to coexist in a system without interfering or overlapping on the same injection points.

3.2.1. Using CDI Extensions

In environments where CDI Full or CDI Lite is available, Jakarta Data providers can make use of CDI extensions - `jakarta.enterprise.inject.spi.Extension` and `jakarta.enterprise.inject.build.compatible.spi.BuildCompatibleExtension` - to discover interfaces that are annotated with `@Repository` and provide their implementations to be injected into injection points within the application. Jakarta Data does not mandate the use of a specific type 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 having a type that is the repository interface and having no qualifiers (other than `Default` or `Any`).

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.

3.2.2. Mapping an Entity

A Jakarta Data provider supplies an implementation of repository interfaces in Jakarta Data for one or more types of entities. An entity refers to a class that represents objects in a storage engine, such as SQL or NoSQL databases.

The `jakarta.persistence.Entity` annotation from the Jakarta Persistence specification can be used by repository entity classes for Jakarta Data providers that are backed by a Jakarta Persistence provider. Other Jakarta Data providers must not support the `jakarta.persistence.Entity` annotation.

The `jakarta.nosql.Entity` annotation from the Jakarta NoSQL specification can be used by repository entity classes for Jakarta Data providers that are backed by NoSQL databases. Other Jakarta Data providers must not support the `jakarta.nosql.Entity` annotation.

Jakarta Data providers that define custom entity annotations must follow the convention that the class name of all supported entity annotation types ends with `Entity`. This enables Jakarta Data providers to identify if a repository entity class contains entity annotations from different Jakarta Data providers so that the corresponding `Repository` can be ignored by Jakarta Data providers that should not provide it.

Jakarta Data providers must ignore all `Repository` annotations where annotations for the corresponding entity are available at run time and none of the entity annotations are supported by the Jakarta Data provider. Ignoring these `Repository` annotations allows other Jakarta Data providers to handle them.

3.2.3. Jakarta Data Provider Name

The entity annotation class is the primary strategy to avoid conflicts between Jakarta Data providers. In most cases, it is sufficient to differentiate between providers. In situations where multiple Jakarta Data providers support the same entity annotation class, the application can specify the name of the desired Jakarta Data provider using the optional `provider` attribute of the `Repository` annotation.

To ensure compatibility and prevent conflicts, Jakarta Data providers must disregard all `Repository` annotations that specify a different provider's name through the `Repository.provider()` attribute. By ignoring these annotations, Jakarta Data providers allow other Jakarta Data providers to handle them.

3.3. Jakarta Transactions Usage

When running in an environment where Jakarta Transactions is available and a global transaction is active on the thread of execution for a repository operation and the data source backing the repository is capable of transaction enlistment, the repository operation enlists the data source resource as a participant in the transaction. The repository operation does not commit or roll back the transaction that was already present on the thread, but it might cause the transaction to be marked as rollback only (`jakarta.transaction.Status.STATUS_MARKED_ROLLBACK`) if the repository operation fails.

When running in an environment where Jakarta Transactions and Jakarta CDI are available, a repository method can be annotated with the `jakarta.transaction.Transactional` annotation, which is applied to the execution of the repository method.

3.4. Interceptor Annotations on Repository Methods

When a repository method is annotated with an interceptor binding annotation, the interceptor is bound to the repository bean according to the interceptor binding annotation of the repository interface method, causing the bound interceptor to be invoked around the repository method when it runs. This enables the use of interceptors such as `jakarta.transaction.Transactional` on repository methods when running in an environment where the Jakarta EE technology that provides the interceptor is available.

3.5. Jakarta Persistence

When integrating Jakarta Data with Jakarta Persistence, developers can leverage the JPA annotations to define the mapping of entities in repositories. Entities in Jakarta Persistence are typically annotated with `jakarta.persistence.Entity` to indicate their persistence capability.

A Jakarta Data provider that supports Jakarta Persistence allows you to define repositories for classes marked with the `jakarta.persistence.Entity` annotation.

By supporting Jakarta Persistence 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.

3.6. 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.

3.7. Jakarta Bean Validation

Integrating with Jakarta Validation 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:

```
@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.

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

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

3.7.1. Avoiding Overlap with Validation from Jakarta Persistence

Jakarta Data providers that are built using Jakarta Persistence might require the user to define persistence units for repositories or might handle the details of defining the persistence units internally. A user that defines the persistence unit for a Jakarta Data repository must specify the `validation-mode` as `NONE` per the "Enabling Automatic Validation" section of the Jakarta Persistence

specification to avoid duplicate validation of entities. Similarly, the Jakarta Data provider must specify either the `validation-mode` of `NONE` or the `jakarta.persistence.validation.mode` map key with value of `none` that is defined in the "Enabling Automatic Validation" section of the Jakarta Persistence specification to avoid duplicate validation of entities. == Portability in Jakarta Data

Portability is a critical aspect of Jakarta Data, ensuring flexibility and adaptability across different data stores while maintaining consistent functionality. Jakarta Data offers a comprehensive approach to portability, which is discussed separately for relational databases and NoSQL databases.

3.8. Portability in Relational Databases

All functionality defined by Jakarta Data must be supported when using relational databases.

1. **Support for Jakarta Persistence Annotations:** Jakarta Data, when used in conjunction with a Jakarta Persistence provider, ensures comprehensive support for all Jakarta Persistence entity annotations. This support covers a wide range of functionality, including entity definitions, primary keys, and relationships.
2. **Built-In Repositories:** Jakarta Data's built-in repositories, such as `PageableRepository` and `BasicRepository`, are designed to offer consistent and well-defined methods compatible with relational databases. Developers can rely on these repositories to perform common data access tasks.
3. **Query Methods:** Jakarta Data's support for query methods, including pagination, ordering, and limiting, is designed to work seamlessly with relational databases.

By aligning Jakarta Data closely with relational databases, developers can expect high portability and compatibility. This approach ensures that Jakarta Data remains a powerful tool for simplifying data access, irrespective of the specific relational database used.

3.9. Portability in 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. Here, we delve into the key portability aspects for four primary NoSQL database categories: key-value, wide-column, document, and graph databases.

3.9.1. 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.

3.9.2. 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, including query by method. 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 "Query Methods Keywords".

3.9.3. 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 `PageableRepository`. Additionally, method by query is implemented, though developers should be aware that some keywords may not be universally supported. The full set of required keywords is documented in the section "Query Methods Keywords".

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.

3.9.4. 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 properties 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/properties will be translated into vendor-specific queries.

It's important to note that in Jakarta Data the Graph database supports both built-in repositories: `BasicRepository` and `PageableRepository`. Additionally, query-by-method is implemented, though developers should be aware that some keywords may not be universally supported. The full set of required keywords is documented in the section "Query Methods Keywords."