JAKARTA EE

Jakarta NoSQL

1.0.0, January 18, 2025: Final

Table of Contents

Specification: Jakarta NoSQL

Version: 1.0.0

Status: Final

Release: January 18, 2025

Chapter 1. Introduction

In the ever-evolving landscape of modern application development, NoSQL databases have emerged as a vital component for handling vast amounts of unstructured data efficiently. Jakarta NoSQL bridges the gap between Java applications and NoSQL databases by offering a comprehensive set of APIs, annotations, and SPIs. These standardized tools empower developers to seamlessly integrate their Java applications with various types of NoSQL databases, ensuring flexibility and agility in database selection without compromising application logic.

At its core, Jakarta NoSQL aims to simplify the complexity associated with NoSQL database integration, providing a unified approach for developers to interact with different database systems. By abstracting away the intricacies of database-specific operations, such as data modeling, querying, and transaction management, Jakarta NoSQL fosters a consistent development experience across diverse NoSQL technologies. This abstraction layer not only streamlines development efforts but also future-proofs applications against changes in underlying database implementations, enabling smooth transitions between different NoSQL solutions as project requirements evolve.

Furthermore, Jakarta NoSQL fosters a vibrant ecosystem by encouraging collaboration between developers, database vendors, and the wider Java community. Its extensible architecture allows for the seamless integration of custom database implementations, ensuring compatibility with both established and emerging NoSQL technologies. With Jakarta NoSQL, developers can focus on building robust, scalable applications while leveraging the strengths of NoSQL databases, confident in the knowledge that their code remains portable, adaptable, and primed for future innovation.

1.1. Goal

The primary goal of Jakarta NoSQL is to streamline and enhance productivity in performing everyday NoSQL operations within Java applications. In the ever-expanding landscape of data management, NoSQL databases have emerged as powerful tools for handling diverse data structures and massive volumes of information. Jakarta NoSQL aims to facilitate seamless integration between Java applications and NoSQL databases, providing developers with a standardized and efficient approach to interacting with these databases.

1. **Increasing Productivity**: Jakarta NoSQL is designed to simplify the process of working with NoSQL databases, enabling developers to focus on application logic rather than the intricacies of database management. By offering standardized APIs, annotations, and query languages, Jakarta NoSQL reduces the learning curve associated with integrating and interacting with various NoSQL database systems.
2. **Rich Object Mapping**: One of the core features of Jakarta NoSQL is its support for rich object mapping. This feature allows developers to map Java objects directly to NoSQL database structures, eliminating the need for complex data transformation code. By providing a seamless mapping mechanism, Jakarta NoSQL enables developers to work with NoSQL databases using familiar object-oriented paradigms, enhancing productivity and code readability.
3. **Flexibility and Adaptability**: Jakarta NoSQL is designed to be flexible and adaptable, capable of working with a wide range of NoSQL database systems. Moreover, its extensible architecture allows for the easy integration of new database types and behaviors through extensions. It

ensures that Jakarta NoSQL remains relevant and up-to-date in the face of evolving database technologies and requirements.

1.2. Non-Goals

While Jakarta NoSQL aims to enhance productivity and simplify integration with NoSQL databases, it is essential to clarify its non-goals:

1. **ORM-like Features**: Jakarta NoSQL does not aim to replicate all Object-Relational Mapping (ORM) framework features. While it provides rich object mapping capabilities, it may offer a different level of abstraction and functionality than traditional ORM frameworks for relational databases.
2. **Full Compatibility with Every NoSQL Database**: Jakarta NoSQL aims to provide a standardized approach for working with NoSQL databases. However, it may offer partial compatibility with every NoSQL database on the market. Compatibility may vary based on the database type and specific features supported by each database.
3. **Replacing Database-specific Features**: Jakarta NoSQL does not intend to remove all database-specific features provided by individual NoSQL databases. While it offers a standard set of APIs and annotations, developers may still need to leverage database-specific features directly for certain advanced use cases.

1.3. Conventions

Throughout the Jakarta NoSQL specification, the terms "entity attribute" and "entity property" are used interchangeably to refer to the fields or properties defined within an entity class.

When demonstrating output samples, JSON format is commonly used to represent data structures. However, it's important to note that this does not imply that a NoSQL database must serialize data in JSON format. The JSON samples provided serve to demonstrate and exemplify the structure of the data.

It's crucial to understand that a Jakarta NoSQL provider and the underlying NoSQL database have the flexibility to define the serialization process according to their requirements. This may involve using user-defined types (UDTs), proprietary serialization formats, or other methods tailored to the specific database technology used.

1.4. Jakarta NoSQL Project Team

This specification is being developed as part of Jakarta NoSQL 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 Lead

- Otavio Santana

1.4.2. Contributors

- Ivar Grimstad
- Kevin Sutter
- Scott Stark

1.4.3. Committers

- Andres Galante
- Fred Rowe
- Gaurav Gupta
- Ivan Junckes Filho
- Jesse Gallagher
- Michael Redlich
- Nathan Rauh
- Otavio Santana
- Werner Keil

1.4.4. Historical Committer

- Leonardo Lima

1.4.5. Mentor

- Wayne Beaton

1.4.6. Full List of Contributors

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

Chapter 2. Entity Classes

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

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

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

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

- Each persistent instantiation of the schema is distinguishable by a unique *identifier*.
- Any persistent instantiation of the schema is representable by an instance of the entity class. In a given Java program, multiple entity class instances might represent the same persistent instance of the schema.

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

1. The **entity class** itself: An entity class is simple Java object equipped with fields or accessor methods designating each property of the entity. An entity class is identified by an annotation.
2. Its **data schema**: Some data storage technologies require an explicit schema defining the structure and properties of the data the entity represents.

2.1. Programming Model for Entity Classes

A *programming model for entity classes* specifies:

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

Jakarta NoSQL defines its programming model for entities explicitly. It relies on annotations provided by the specification. Jakarta NoSQL's programming model allows for seamless integration with custom annotations defined by Jakarta NoSQL providers or extensions.

This approach ensures flexibility and interoperability, enabling developers to leverage Jakarta NoSQL's standardized annotations alongside provider-specific annotations for fine-tuning entity

behavior and mapping. Additionally, Jakarta NoSQL facilitates integration with other Jakarta EE specifications, fostering a cohesive Java-based NoSQL application development ecosystem.

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

Every entity programming model specifies an *entity-defining annotation*, `jakarta.nosql.Entity`.

Furthermore, an entity programming model must define an annotation that identifies the field or property holding the unique identifier of an entity, the `Jakarta.nosql.Id`.

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

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

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

In either case, an entity programming model might restrict the visibility of an entity class's fields and property accessors.

An entity programming model might support inheritance between entities and provide support for retrieving entities in a polymorphic fashion. This specification does not require support for inheritance.

To ensure compatibility with Jakarta NoSQL, an entity programming model must adhere to the following constructor rules:

- Constructors must be `public` or `protected` with no parameters or with parameters annotated with `Jakarta.nosql.Column` or `Jakarta.nosql.Id`.
- Annotations at the constructor will build the entity and read information from the database, while field annotations are required to write information to the database.
- If both a non-args constructor and a constructor with annotated parameters exist, the constructor with annotations will be used to create the entity.
- Constructor parameters without annotations will be ignored, utilizing a non-arg constructor instead.
- Entities should not have multiple constructors using `jakarta.nosql.Id` or `jakarta.nosql.Column` annotations.

2.1.1. Persistent Fields

A field of an entity class may or may not represent state which is persistent in the datastore. A *persistent field* has some corresponding representation in the data schema of the entity.

Every programming model for entity classes must support *direct field access*, that is, access to the persistent fields of an entity class without triggering any intermediating user-written code such as JavaBeans-style property accessors.

A programming model might place constraints on the visibility of persistent fields.

Jakarta NoSQL distinguishes three kinds of persistent field within entity classes.

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

2.1.2. Basic Types

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

Basic Data Type	Description
Primitive types and wrapper classes	All Java primitive types, such as `int`, `double`, `boolean`, etc., and their corresponding wrapper types from `java.lang` (e.g., `Integer`, `Double`, `Boolean`).
`java.lang.String`	Represents text data.
`LocalDate`, `LocalDateTime`, `LocalTime`, `Instant` from `java.time`	Represent date and time-related data.
`java.util.UUID`	Universally Unique IDentifier for identifying entities.
`BigInteger` and `BigDecimal` from `java.math`	Represent large integer and decimal numbers.
`byte[]`	Represents binary data.
User-defined `enum` types	Custom enumerated types defined by user-written code.

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

```
@Entity
```

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

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

Many key-value, wide-column, and document databases feature native support for arrays or even associative arrays of these basic types.

2.1.2.1. Enum Type

Enum types in Java represent a fixed set of constants. In Jakarta NoSQL, enums are considered basic types and are commonly used to represent data with a limited number of predefined values. By default, enums are stored as strings in the database, with the enum constant name being used as the stored value. The `name()` method of the enum class is typically used to retrieve the name of the enum constant.

For example, consider the following enum representing the days of the week:

```
public enum DayOfWeek {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

When using an enum type in an entity class, it can be annotated with the `@Column` annotation to specify the storage details. Here's an entity class `Meeting` that includes an enum field representing the day of the week:

```
@Entity
public class Meeting {
```

```
    @Id
    private String id;

    @Column
    private DayOfWeek day;

    @Column
    private List<String> attendees;
}
```

In this example, the `day` field of the `Meeting` entity is of type `DayOfWeek`, an enum type representing the days of the week. The `@Column` annotation indicates that this enum will be stored as a string in the database using the `name()` method to retrieve the enum constant's name.

The JSON representation of a `Meeting` entity might look like this:

```
{
  "id": "123456",
  "day": "MONDAY",
  "attendees": ["Alice", "Bob", "Charlie"]
}
```

2.1.3. Embedded Fields and Embeddable Classes

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

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

An *embedded field* is a field whose type is an embeddable class.

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

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

Jakarta NoSQL defines an annotation identifying a user-written class as an embeddable class: `jakarta.nosql.Embeddable`.

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

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

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

To ensure compatibility with Jakarta NoSQL, an embeddable class must adhere to the following constructor rules:

- Constructors must be `public` or `protected` with no parameters or parameters annotated with `jakarta.nosql.Column`.
- Annotations at the constructor will build the entity and read information from the database, while field annotations are required to write information to the database.
- If both a non-args constructor and a constructor with annotated parameters exist, the constructor with annotations will be used to create the entity.
- Constructor parameters without annotations will be ignored, utilizing a non-arg constructor instead.
- Embeddable classes should not have multiple constructors using `jakarta.nosql.Column` annotations.

For example, consider the following Java classes:

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

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

In a document, wide-column, or graph database, the JSON representation of an instance of the `Person` entity where the `Address` class is **flat** might be:

```
{
  "id": 1,
  "name": "John Doe",
  "street": "123 Main St",
```

```
  "city": "Sampleville",
  "postalCode": "12345"
}
```

In a structured representation, when the embeddable field is **grouping** it will be together in the data schema.

```
@Embeddable(GROUPING)
public class Address {
    @Column
    private String street;
    @Column
    private String city;
    @Column
    private String postalCode;
}
```

In a document, wide-column, or graph database, the JSON representation of an instance of the `Person` entity where the `Address` class is **grouping** might be:

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

When an embeddable class is used within an iterable field of an entity class, both embedding strategies, namely **flattening** and **grouping**, will function as **grouping**. This means that the fields of the embeddable class will be grouped together within the data schema, regardless of whether the embeddable class is marked for flattening or grouping.

For example, consider the following entity class `Driver` containing an iterable of `Car` instances:

```
@Entity
public class Driver {
    @Id
    private UUID id;
    @Column
    private String name;
    @Column
    private Iterable<Car> cars;
}
```

```
@Embeddable
public class Car {
    @Column
    private String plate;
    @Column
    private String category;
}
```

In this scenario, the `Car` embeddable class is used within the `cars` field, which is an iterable in the `Driver` entity class. As a result, the embedding strategy will behave as **grouping**, regardless of whether the `Car` class is marked with the `@Embeddable(GROUPING)` annotation.

The JSON representation of an instance of the `Driver` entity might appear as follows:

```
{
  "id": "123e4567-e89b-12d3-a456-426614174000",
  "name": "John Doe",
  "cars": [
    {
      "plate": "ABC123",
      "category": "Sedan"
    },
    {
      "plate": "XYZ789",
      "category": "SUV"
    }
  ]
}
```

In this JSON representation, the `cars` field contains an array of `Car` objects, each with its own `plate` and `category` fields. This structure reflects the **grouping** embedding strategy, where the fields of the `Car` embeddable class are grouped together within the `Driver` entity's data schema.

Additionally, it's important to note that support for embedding with a `Map` may vary by NoSQL database and Jakarta NoSQL provider. Different providers may have different approaches or limitations regarding the embedding of data structures such as maps with embeddable classes. Developers should consult the documentation of their chosen NoSQL database and Jakarta NoSQL provider for specific details and considerations regarding map embedding.

Support for grouping embeddable classes and embedded fields is not required by this specification. However, every Jakarta NoSQL provider is strongly encouraged to support embeddable classes within its entity programming model. Some databases might require the use of the `udt` attribute in the `@Column` annotation for embedded fields.

2.1.4. Array Support

Jakarta NoSQL implementations MUST support binding Java arrays of the basic types, as referenced in Basic Types, and arrays of entities and embedded classes.

Arrays of entities and embedded classes are supported and will function as embedded classes with **grouping**.

Consider an entity class `Library` with an array of `Book` entities and an array of `String` tags.

```
@Entity
public class Library {
    @Id
    private Long id;

    @Column
    private Book[] books;

    @Column
    private String[] tags;
}

@Entity
public class Book {
    @Id
    private Long id;

    @Column
    private String title;
}
```

In this example, the array of `Book` entities will be treated as an embedded collection within the `Library` entity, using **grouping** to represent the structure.

The JSON representation of an instance of the `Library` entity might be:

```
{
  "id": 1,
  "books": [
    {"id": 101, "title": "Java Programming"},
    {"id": 102, "title": "Introduction to NoSQL"}
  ],
  "tags": ["Programming", "NoSQL", "Java"]
}
```

2.1.5. Entity Associations

An association field is a field of an entity class whose declared type is also an entity class. Given an

instance of the first entity class, its association field references an instance of a second entity class.

For example, consider the following Java classes:

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

@Entity
public class Book {
    @Column
    private String title;
    @Column
    private String category;
}
```

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

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "John Smith",
  "books": [
    {
      "title": "Java Programming",
      "category": "Programming"
    },
    {
      "title": "Introduction to NoSQL",
      "category": "Database"
    }
  ]
}
```

In this scenario, the association between `Author` and `Book` is represented by the `books` field in the `Author` entity class. Since NoSQL databases do not support joins, the association field behaves as a **grouping** embedded field defined at Embedded Fields and Embeddable Classes . It means that the `books` field groups together instances of the `Book` entity within the `Author` entity's data schema.

This specification does not require support for entity associations. Some databases might require the use of the udt attribute in the @Column annotation for embedded fields.

2.1.6. Collections of Embeddable Classes and Basic Types

A persistent field or property of an entity or embeddable class may correspond to a collection of a basic type, embeddable, or entity class.

No action is required beyond including the `Column` annotation for a collection of basic types.

```
@Entity
public class BucketList {
    @Id
    private Long id;
    @Column
    private String name;
    @Column
    private List<String> tasks;
}
```

```
{
  "id": 123,
  "name": "Personal Goals",
  "tasks": ["Travel the world", "Learn a new language", "Write a book"]
}
```

The entity class will behave as an embeddable **grouping** class. This support may vary among NoSQL providers and might require a UDT name presentation in the case of embeddable or entity.

For key-value databases, the serialization will occur through a unique blob, a process outside the scope of the Jakarta NoSQL specification.

```
@Entity
public class Company {
    @Id
    private String name;

    @Column(udt= "headquarter")
    private Set<Headquarter> headquarters;
}

@Entity
// It could be Embedded, and the behavior won't change
public class Headquarter {

    @Column
    private String city;

    @Column
    private String country;
```

```
}
```

```
{
  "name": "Acme Inc.",
  "headquarters": [
    {"city": "New York", "country": "USA"},
    {"city": "London", "country": "UK"}
  ]
}
```

Collections within entities can accommodate various types of data, including basic types and complex structures like lists of strings. Jakarta NoSQL provides flexibility in handling such collections, ensuring seamless integration with the underlying NoSQL database.

2.1.7. Map Collections

Java `Map` collections offer a convenient way to represent associations and key-value pairs within entities in Jakarta NoSQL. Jakarta NoSQL handles map collections, allowing developers to manage complex data structures efficiently.

```
@Entity
public class Contact {
    @Id
    private String name;

    @Column
    private Map<String, String> socialMedia;
}
```

JSON representation:

```
{
  "name": "John Doe",
  "socialMedia": {
    "twitter": "@johndoe",
    "linkedin": "linkedin.com/in/johndoe"
  }
}
```

In the example above, the `Contact` entity includes a `socialMedia` field, represented as a `Map` where the key is a string representing the social media platform, and the value is the corresponding username or profile link.

The behavior of map collections remains consistent regardless of whether the map values are basic types, embeddable classes, or entity classes. However, for embeddable or entity classes used as map values, the `udt` attribute may be required in the `@Column` annotation to specify the user-defined type.

For instance, consider the following example:

```
@Entity
public class Computer {
    @Id
    private String name;

    @Column
    private Map<String, Program> programs;
}

@Embedded
public class Program {
    @Id
    private String name;

    @Column
    private Map<String, String> socialMedia;
}
```

JSON representation:

```
{
  "name": "My Computer",
  "programs": {
    "browser": {
      "socialMedia": {
        "twitter": "@browseruser",
        "instagram": "@browseruser"
      }
    },
    "editor": {
      "socialMedia": {
        "github": "github.com/editoruser",
        "linkedin": "linkedin.com/in/editoruser"
      }
    }
  }
}
```

The `Computer` entity includes a `programs` field, a map where the keys represent program names, and the values are instances of the `Program` embeddable class. Each `Program` instance contains its own `socialMedia` map, representing the social media profiles associated with that program.

It's important to note that support for map collections may vary depending on the NoSQL database and Jakarta NoSQL provider used. Developers should consult the documentation of their chosen provider for specific details and considerations regarding map collections.

2.1.8. 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, optionally joined by a delimiter.

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

Chapter 3. Annotations

Jakarta NoSQL introduces a comprehensive set of annotations tailored to streamline and simplify mapping Java entities to NoSQL databases. These annotations offer:

- A standardized approach for defining entity classes, marked with `@Entity`, to establish the structural blueprint of data entities within the NoSQL environment.
- Precise specification of primary keys using the `@Id` annotation, essential for uniquely identifying entities within the database.
- Flexible mapping of entity properties to database fields through the `@Column` annotation, ensuring seamless integration of Java objects with NoSQL data storage.
- Conversion of non-persistent object types to database-compatible formats facilitated by the `@Convert` annotation, enhancing compatibility and data manipulation capabilities.
- Embedding of objects within entity structures enabled by the `@Embeddable` annotation, allowing for efficient storage of complex data structures as part of owning entities.
- Abstraction of common properties and behaviors across multiple entity classes through the `@MappedSuperclass` annotation, promoting code reusability and maintainability.
- Specifying inheritance strategies using the `@Inheritance`, `@DiscriminatorValue`, and `@DiscriminatorColumn` annotations facilitates polymorphic data modeling within the NoSQL environment.

Jakarta NoSQL has support for those nine types:

1. @Entity
2. @Embeddable
3. @Id
4. @Column
5. @Convert
6. @MappedSuperclass
7. @Inheritance
8. @DiscriminatorColumn
9. @DiscriminatorValue

In the realm of Jakarta NoSQL, developers wield a powerful arsenal of annotations tailored to meet diverse data modeling needs:

- **@Entity**: The `@Entity` annotation signifies that a Java class represents a persistent entity with a lifecycle managed by the underlying data store. By annotating a class with `@Entity`, developers indicate that instances of this class are subject to CRUD (Create, Read, Update, Delete) operations within the NoSQL database. This annotation not only defines the entity's structure but also denotes its existence beyond the scope of a single Java application instance. In essence, the `@Entity` annotation encapsulates the notion of a domain object that persists beyond the lifetime of a Java process, ensuring consistency and durability in data management.

- **@Embeddable**: The `@Embeddable` annotation is a Java feature that identifies a class that can be embedded within another entity. It enables developers to create intricate data structures by combining reusable components. This technique makes it possible to represent finer-grained attributes of an entity by using an embeddable class, which encapsulates related data fields into a single logical unit. There are two types of embedding strategies that can be used with this annotation: flattening and grouping. In the flattening strategy, the fields of the embeddable class are directly added to the data schema of the parent entity. In contrast, in the grouping strategy, the fields are grouped within a structured type.
- **@Id**: Central to the entity model is the `@Id` annotation, which designates a field as the primary key. This annotation empowers developers to define the unique identifier for each entity, ensuring data integrity and facilitating efficient data retrieval operations.
- **@Column**: The `@Column` annotation provides fine-grained control over mapping entity properties to database fields. By annotating fields with `@Column`, developers customize the storage and retrieval of data, specifying attributes such as column names, types, and constraints.
- **@Convert**: With the `@Convert` annotation, developers can seamlessly transform entity attribute values between Java and database types. This annotation offers flexibility in data representation, allowing developers to adapt entity properties to suit the requirements of different database systems.
- **@MappedSuperclass**: The `@MappedSuperclass` annotation is used to define shared attributes and behaviors across multiple entity classes by denoting a superclass whose mappings are applied to its subclasses.
- **@Inheritance**: The @Inheritance annotation facilitates modeling inheritance hierarchies within entity classes. By default, Jakarta NoSQL supports a single inheritance strategy where subclass information is incorporated into the data structure as a field within the parent entity. In this default strategy, attributes of subclasses are represented as fields within the parent entity, maintaining a denormalized data structure. However, Jakarta NoSQL allows Jakarta Data providers to offer alternative inheritance strategies beyond the default specification.
- **@DiscriminatorColumn**: The `@DiscriminatorColumn` annotation configures the discriminator column used in single table inheritance mappings. By annotating a field with `@DiscriminatorColumn`, developers control the storage of discriminator values, ensuring accurate and efficient retrieval of entity subclasses.
- **@DiscriminatorValue**: When using inheritance strategies, the `@DiscriminatorValue` annotation specifies the discriminator value for entities in a single table inheritance hierarchy. This annotation enables database systems to differentiate between subclasses based on a discriminator column value.

In essence, Jakarta NoSQL annotations empower developers to craft sophisticated data models that seamlessly bridge the gap between Java entities and NoSQL databases. With a rich array of annotations at their disposal, developers can unlock the full potential of NoSQL technology, building scalable, efficient, and maintainable applications within the Jakarta EE ecosystem.

3.1. @Entity

The `@Entity` annotation is the cornerstone for defining persistent entities within Jakarta NoSQL. By annotating a Java class with `@Entity`, developers signify its role as a persistent entity, eligible for

storage and retrieval in a NoSQL database. This annotation encapsulates the lifecycle management of entities, facilitating seamless integration with various NoSQL data stores.

3.1.1. Entity Definition Reference

First, let's establish a reference for entity definition, denoted by Entity Classes. In Jakarta NoSQL, an entity class is typically annotated with `@Entity` to indicate its persistent nature.

```
@Entity
public class Person {

@Id
private UUID id;
@Column
private String name;
}
```

In this example, the `Person` class is defined as an entity with an `id` field annotated with `@Id`, which designates it as the primary key, and a `name` field annotated with `@Column`, indicating it as a persistent attribute.

One of the notable features of Jakarta NoSQL is its support for immutable and mutable entity classes. For immutable classes, Jakarta NoSQL provides compatibility with Java records, allowing developers to define compact and immutable entity structures concisely.

```
@Entity
public record Person(@Id private UUID id, @Column private String name) {
}
```

In this sample, the `Person` class is defined as a record, capturing its immutable nature. The `@Id` and `@Column` annotations are applied directly to the constructor parameters, indicating the primary key and persistent attributes.

The serialization method of entity classes may vary depending on the NoSQL vendor and configuration. Here's a sample JSON structure representing a `Person` entity:

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "John Doe"
}
```

This JSON structure captures the serialized representation of a `Person` entity with its `id` and `name` attributes. The specific serialization method may differ based on the chosen NoSQL vendor and its corresponding serialization mechanisms.

3.1.2. Associating with Other Entities

One of the powerful features of Jakarta NoSQL is its ability to associate entities with each other, enabling the creation of complex data structures. When an entity is related to another entity, it is incorporated as an embeddable group within the parent entity, as defined by Embedded Fields and Embeddable Classes . Let's consider an example where a `Person` entity is associated with an `Address` entity:

```
@Entity
public class Person {

    @Id
    private Long id;

    @Column
    private String name;

    @Column
    private Address address;
}

@Entity
public class Address {

    @Column
    private String street;

    @Column
    private String city;
}
```

In this example, the `Person` entity includes a field `address` of type `Address`, indicating the association between the two entities. The `address` field will be represented as a nested structure within the `Person` entity when serialized.

The serialization method of entity classes may vary depending on the NoSQL vendor. Here's a sample JSON structure representing a `Person` entity with an associated `Address`:

```
{
   "_id":10,
   "name":"Ada Lovelace",
   "address":{
      "city":"São Paulo",
      "street":"Av Nove de Julho"
   }
}
```

This JSON structure represents a serialized `Person` entity with `id`, `name`, and `address` attributes. The

`address` field is a nested structure that includes `city` and `street` attributes from the associated `Address` entity.

Entities can also be associated using collection types like `Iterable`, such as `List` or `Set`. Let's consider an example where an `Owner` entity is associated with multiple `Car` entities:

```
@Entity
public class Owner {
    @Id
    private String name;
   @Column
    private List<Car> cars;
}

@Entity
public class Car {
    @Column
    private String make;
    @Column
    private String model;
}
```

In this example, the `Owner` entity includes a field `cars` of type `List<Car>`, indicating an association between the two entities. The `cars` field will hold a collection of `Car` entities associated with the `Owner`.

The serialization method of entity classes may vary depending on the NoSQL vendor. Here's a sample JSON structure representing an `Owner` entity with associated `Car` entities:

```
{
   "name": "marie Curie",
   "cars": [
      {
         "make": "Toyota",
         "model": "Camry"
      },
      {
         "make": "Honda",
         "model": "Accord"
      }
   ]
}
```

This JSON structure represents a serialized `Owner` entity with `name` and `cars` attributes. The `cars` field is an array containing nested structures representing associated `Car` entities.

It's important to note that not all NoSQL databases support entity associations. Developers should verify the compatibility of association features with their

chosen database technology.

3.2. @Embeddable

The `@Embeddable` annotation in Jakarta NoSQL marks a class as embeddable, as defined by Embedded Fields and Embeddable Classes . An embeddable class is a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

By default, the embedding strategy is FLAT, where the fields of the embedded class are directly mapped to columns in the owning entity's table.

In the example below:

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

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

Here's the JSON sample for **FLAT** embedding:

```
{
   "id": 1,
   "name": "John Doe",
   "address_street": "123 Main St",
   "address_city": "Sampleville",
   "address_postalCode": "12345"
}
```

In addition to **FLAT**, the **GROUPING** embedding strategy is also available. In GROUPING embedding, the fields of the embedded class are grouped together within a structured type in the owning entity's table.

To specify GROUPING embedding, use `@Embeddable(GROUPING)`.

Here's the JSON sample for GROUPING embedding:

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

3.3. @Id

The `@Id` annotation in Jakarta NoSQL defines the unique identifier for entities within a database. For any entity, it is mandatory to have a field annotated with `@Id` to identify each instance uniquely. According to the specification, the field annotated with `@Id` must be of a basic type defined by Basic Types.

While the `@Id` annotation allows for a native name, it's essential to note that some NoSQL databases have reserved keywords for their keys. In such cases, if the native name conflicts with a database keyword, the database might ignore it.

Jakarta NoSQL does not provide any strategy for autoincrement when the field annotated with `@Id` is not filled up. The behavior varies with the NoSQL database. For instance, in a Key-Value NoSQL database, a null value for an ID will generate a `NullPointerException`. In other NoSQL databases, it might vary from generating a value to returning an exception error.

Let's consider an example:

```
@Entity
public class User {

    @Id
    private String userName;

    @Column
    private String name;

    @Column
    private List<String> phones;
}
```

In this example, the `User` entity includes a field `userName` annotated with `@Id`, indicating that `userName` serves as the unique identifier for instances of the `User` entity.

Here's a generated JSON sample of this structure:

```
{
   "userName": "john_doe",
   "name": "John Doe",
   "phones": [
      "123456789",
      "987654321"
   ]
}
```

This JSON structure represents a serialized `User` entity with `userName`, `name`, and `phones` attributes. The `userName` field acts as the unique identifier for the entity.

3.4. @Column

The `@Column` annotation in Jakarta NoSQL marks fields that should be mapped to database columns within an entity, similar to the `@Id` annotation.

In the example below:

```
@Entity
public class Person {
    @Column
    private String nickname;

    @Column
    private String name;

    @Column
    private List<String> phones;

    // ignored for Jakarta NoSQL
    private String address;
}
```

In this example, the `Person` entity includes fields `nickname`, `name`, and `phones` annotated with `@Column`. These fields are marked for persistence, indicating that they should be mapped to database columns.

It's important to note that fields without the `@Column` annotation, such as `address` in the example, will be ignored for Jakarta NoSQL mapping.

Here's a generated JSON sample of this structure:

```
{
   "nickname": "john_doe",
```

```
   "name": "John Doe",
   "phones": [
      "123456789",
      "987654321"
   ]
}
```

This JSON structure represents a serialized `Person` entity with `nickname`, `name`, and `phones` attributes. The `nickname`, `name`, and `phones` fields are mapped to corresponding database columns.

When using the `@Column` annotation, it's important to remember that Key-Value databases usually only require the `@Id` annotation to identify unique entities. The Jakarta NoSQL provider will determine how to serialize the object for storage, which can be combined with other annotations, such as Jakarta JSON Binding, to customize the serialization process. This flexibility allows developers to adapt the serialization process to the specific requirements of their database.

If you use NoSQL databases that serialize information to JSON, A Jakarta NoSQL provider can integrate them with Jakarta JSON Binding annotations. The Jakarta NoSQL provider will define the integration process, ensuring a smooth and efficient serialization of data to the JSON format.

3.5. @Convert

The `@Convert` annotation is used in Jakarta NoSQL to declare that a specific field in an entity class requires conversion using a specified converter. This annotation is useful for converting non-persistent object types to formats that are compatible with the database, which expands data storage and manipulation capabilities.

When you use the `@Convert` annotation, you must provide a converter class as an argument, which specifies the type of conversion to be performed. The converter class must implement the `AttributeConverter` interface and define methods for converting the object type from and to its database representation.

For example, consider the following entity class `Employee`:

```
@Entity
public class Employee {

    @Column
    private String name;

    @Column
    private Job job;

    @Column("money")
    @Convert(MoneyConverter.class)
    private MonetaryAmount salary;
```

```
}
```

In this example, the `Employee' entity's' salary' field is annotated with `@Convert`, specifying the `MoneyConverter` class as the converter. The `MoneyConverter` class implements the `AttributeConverter` interface to convert `MonetaryAmount` objects to and from their database representation.

Here's a simplified implementation of the `MoneyConverter` class:

```
public class MoneyConverter implements AttributeConverter<MonetaryAmount, String> {

    @Override
    public String convertToDatabaseColumn(MonetaryAmount appValue) {
        return appValue.toString();
    }

    @Override
    public MonetaryAmount convertToEntityAttribute(String dbValue) {
        return MonetaryAmount.parse(dbValue);
    }
}
```

Additionally, let's generate a JSON format for this code:

```
{
   "name": "John Doe",
   "job": "Software Engineer",
   "money": "USD 5000.00"
}
```

In this JSON representation, the `money` field is stored in a database-compatible format after conversion by the `MoneyConverter` class, ensuring seamless integration with the NoSQL database.

3.6. Inheritance

In Jakarta NoSQL, entities support inheritance, enabling the creation of hierarchies of classes where subclasses inherit attributes and behaviors from their superclass. This feature allows for the modeling of complex data structures and relationships within NoSQL databases.

Entities in Jakarta NoSQL can be both abstract and concrete classes. Abstract classes can be annotated with the `@Entity` annotation and mapped as entities, allowing them to participate in entity inheritance hierarchies and be queried for as entities. Similarly, concrete classes can also be annotated with `@Entity` and serve as entities in the inheritance hierarchy.

It's important to note that entities can extend both entity and non-entity classes. This flexibility allows for the construction of diverse inheritance structures, accommodating various data modeling requirements.

Inheritance in Jakarta NoSQL facilitates polymorphic associations and queries, allowing for more flexible and expressive data manipulation and retrieval operations. Subclasses inherit attributes and behaviors from their superclass, providing a mechanism for code reuse and organizational structuring within the data model.

Overall, the support for inheritance in Jakarta NoSQL contributes to the creation of robust and adaptable data models, empowering developers to effectively represent complex domain structures in NoSQL databases.

Some NoSQL databases, particularly key-value databases, might not fully support inheritance due to their schemaless nature and limited querying capabilities. Developers should carefully consider the compatibility of their chosen database with inheritance features when designing their data models.

3.6.1. Abstract Entity Classes

In Jakarta NoSQL, an abstract class can be specified as an entity, providing a mechanism for defining common attributes and behaviors that are shared among multiple concrete subclasses. An abstract entity class is annotated with the `@Entity` annotation, indicating its role as a mapped entity within the data model.

Abstract entity classes in Jakarta NoSQL differ from concrete entities primarily in their inability to be directly instantiated. Instead, abstract entities serve as templates or blueprints for concrete subclasses, encapsulating shared functionality and defining a common interface for their subclasses to implement.

Despite being abstract, abstract entity classes are fully mapped as entities and can participate in data manipulation operations such as queries. Queries targeting abstract entity classes will operate over and/or retrieve instances of their concrete subclasses, allowing for polymorphic queries that span the entire inheritance hierarchy.

By leveraging abstract entity classes, developers can effectively organize and structure their data model, promoting code reuse, maintainability, and scalability. Abstract entities encapsulate common attributes and behaviors, fostering a modular and extensible design approach within Jakarta NoSQL applications.

3.6.2. @MappedSuperclass

The `@MappedSuperclass` annotation designates a class whose mapping information is applied to the entities that inherit from it. Unlike regular entities, a mapped superclass does not imply the existence of a separate storage structure such as tables in relational databases.

In NoSQL databases, where data is often stored in a schema-less or schema-flexible manner, the concept of inheritance may not directly correspond to table inheritance as seen in relational databases. However, the `@MappedSuperclass` annotation serves a similar purpose by allowing common mappings to be defined in a superclass and inherited by its subclasses.

For example, consider a mapped superclass `Animal`:

```
@Entity
public class Dog extends Animal {

    @Column
    private String name;
}

@MappedSuperclass
public class Animal {

    @Column
    private String breed;

    @Column
    private Integer age;
}
```

In this example, the `Animal` class serves as a mapped superclass where common attributes like `breed` and `age` are defined. Subclasses, such as `Dog`, can then inherit these mappings, enabling a consistent data model across entities while accommodating the flexibility of NoSQL database structures.

Here's a JSON sample demonstrating the usage of `Animal` in a subclass `Dog`:

```
{
   "breed": "Golden Retriever",
   "age": 3,
   "name": "Buddy"
}
```

In this JSON representation, the attributes `breed` and `age` from the `Animal` superclass are inherited by the `Dog` entity, showcasing the application of mapped superclass mappings to its subclasses.

3.6.3. @Inheritance

The `@Inheritance` annotation in Jakarta NoSQL enables the use of inheritance strategies within the data model, allowing for the creation of class hierarchies where subclasses inherit attributes and behaviors from their superclass. By applying `@Inheritance` to the superclass, developers can define the inheritance strategy to be used for mapping the class hierarchy to the underlying NoSQL database.

Consider the following example:

```
@Entity
@Inheritance
public abstract class Notification {
    @Id
    private Long id;
```

```
    @Column
    private String name;

    @Column
    private LocalDate createdOn;

    public abstract void send();
}
```

In Jakarta NoSQL, a unique strategy is employed where a single data structure per class hierarchy is utilized. In this strategy, all classes in the hierarchy are mapped to a single data structure in the database. The structure includes a "discriminator column," which serves to identify the specific subclass to which each instance belongs. This approach facilitates polymorphic relationships between entities and enables queries that span the entire class hierarchy.

However, this strategy does have a drawback: it requires that fields corresponding to subclass-specific state be nullable. Despite this limitation, the `@Inheritance` annotation provides a powerful mechanism for organizing and structuring data within Jakarta NoSQL applications, supporting code reuse, maintainability, and flexibility in data modeling.

It's important to note that the `@Inheritance` annotation works in conjunction with other annotations such as `@DiscriminatorValue` and `@DiscriminatorColumn`, which further refine the inheritance mapping strategy and specify how subclass instances are differentiated within the single data structure.

It's crucial to note that the `@Inheritance` annotation may not be fully supported in all NoSQL databases, particularly in Key-value databases. Due to the schemaless nature of these databases, the concept of class inheritance and table inheritance, as seen in traditional relational databases, may not apply. Therefore, the use of `@Inheritance` in such environments might lead to unexpected behavior or errors, and the Key-value database may outright ignore this annotation.

Additionally, in cases where serialization is performed via JSON, certain NoSQL databases may not inherently support polymorphism and inheritance. However, Jakarta NoSQL providers may implement solutions using features from Jakarta JSON Binding to handle polymorphic entities effectively. Developers should be aware of these limitations and ensure compatibility with their chosen NoSQL database and serialization mechanisms when employing the `@Inheritance` annotation.

3.6.4. @DiscriminatorColumn

The `@DiscriminatorColumn` annotation is utilized to specify the discriminator column for the inheritance mapping strategy within an entity class hierarchy. This annotation is typically applied at the root of the hierarchy, although it can also be used within subhierarchies if a different inheritance strategy is applied.

When using inheritance mapping, the discriminator column serves as a means to differentiate

between different subclasses of the entity. This column holds a value that identifies the specific subclass represented by each entity instance in the database.

If the `@DiscriminatorColumn` annotation is not explicitly provided, and a discriminator column is required, the default name for the column is typically "DTYPE" or "dtype", depending on the NoSQL provider. The discriminator type is set to STRING.

It's essential to note that the `@DiscriminatorColumn` annotation can be applied to both concrete and abstract entity classes. This flexibility allows for precise control over the inheritance mapping strategy within the entity hierarchy.

Let's illustrate the usage of the `@DiscriminatorColumn` annotation with an example:

```
@Entity
@Inheritance
@DiscriminatorColumn("type")
public abstract class Notification {
    @Id
    private Long id;

    @Column
    private String name;

    @Column
    private LocalDate createdOn;

    public abstract void send();
}
```

In this example, the `Notification` class is marked as abstract and serves as the root of an inheritance hierarchy. The `@DiscriminatorColumn("type")` annotation specifies that the discriminator column for this hierarchy will be named "type". This column will hold values indicating the specific subclass type for each entity instance.

3.6.5. @DiscriminatorValue

This annotation specifies the value of the discriminator column for entities of the given type.

The DiscriminatorValue annotation can only be specified on a concrete entity class. If the DiscriminatorValue annotation is not specified, a provider-specific function will be used to generate a value representing the entity type. By default, the discriminator value is derived from the `Class.getSimpleName()`.

The inheritance strategy and the discriminator column are only specified in the root of an entity class hierarchy or subhierarchy in which a different inheritance strategy is applied. The discriminator value, if not defaulted, should be specified for each entity class in the hierarchy.

```
@Entity
@DiscriminatorValue("SMS")
```

```
public class SmsNotification extends Notification {

    @Column
    private String phoneNumber;

    @Override
    public void send() {
        System.out.println("Sending message to sms: " + phoneNumber);
    }
}

@Entity
@DiscriminatorValue("Email")
public class EmailNotification extends Notification {

    @Column
    private String emailAddress;

    @Override
    public void send() {
        System.out.println("Sending message to email: " + emailAddress);
    }
}

@Entity
// the discriminator value is SocialMediaNotification
public class SocialMediaNotification extends Notification {
    @Column
    private String username;

    @Override
    public void send() {
        System.out.println("Sending a post to: " + username);
    }
}
```

This JSON structure represents three different types of notifications: SMS, Email, and Social Media. Each notification has a unique ID, a name, a creation date, and type-specific attributes such as phone number or username. The discriminator value `"type"` indicates the specific subclass of the `Notification` entity.

```
[
  {
    "id": 1,
    "name": "Notification 1",
    "createdOn": "2024-02-14",
    "type": "SMS",
    "phoneNumber": "+1234567890"
  },
  {
```

```
    "id": 2,
    "name": "Notification 2",
    "createdOn": "2024-02-14",
    "type": "Email",
    "phoneNumber": "user@example.com"
  },
  {
    "id": 3,
    "name": "Notification 3",
    "createdOn": "2024-02-14",
    "type": "SocialMediaNotification",
    "username": "socialmedia_user"
  }
]
```

In case of querying an entity using the `@Inheritance` annotation as defined by @Inheritance, the Jakarta NoSQL provider must automatically include the condition where the value from `@DiscriminatorColumn` equals the value of `@DiscriminatorValue`.

For example, given the sample code from @Inheritance and executing the query for `SmsNotification`, the generated query should include a condition such as `type = 'SMS'` or its equivalent in the respective NoSQL database.

Chapter 4. Template Classes

The DAO (Data Access Object) pattern in Jakarta NoSQL simplifies the implementation of common database operations by providing a basic API to the underlying persistence engine. This pattern encapsulates the logic for interacting with the database, promoting a clean separation between the application's business logic and its data access code.

In the DAO pattern, each entity in the application typically has a corresponding DAO class responsible for handling database operations related to that entity. These DAO classes abstract away the complexity of database interactions, providing a simplified interface for performing CRUD (Create, Read, Update, Delete) operations.

Jakarta NoSQL's DAO feature follows this pattern closely, offering a set of template classes that serve as the foundation for implementing DAOs. These template classes provide pre-defined methods for common database operations, such as saving, updating, querying, and deleting entities.

Developers can extend these template classes to create custom DAOs for their application entities. By doing so, they can focus on implementing the specific logic required for their application, while the underlying Jakarta NoSQL framework handles the low-level database interactions.

Overall, the DAO pattern in Jakarta NoSQL promotes modularity, reusability, and maintainability in application development by abstracting away database access details. By adhering to this pattern, developers can create robust and scalable applications with ease, without having to deal with the complexities of database interaction at the application level.

The provided code snippet demonstrates the usage of the `Template` class in Jakarta NoSQL to perform basic CRUD operations on entities in the database using the DAO pattern.

1. **Create Book Entity**: An instance of the `Book` entity is created using the builder pattern. This entity represents a book with attributes such as title, author, publication year, and edition.
2. **Insert Operation**: The `insert` method of the `Template` class is invoked to insert the `Book` entity into the database. This method takes the entity as a parameter and stores it in the underlying database.
3. **Find Operation**: The `find` method of the `Template` class is called to retrieve the `Book` entity from the database based on its ID (`id`). This method returns an `Optional` object containing the retrieved entity, if it exists.
4. **Print Result**: The retrieved `Book` entity is printed to the console using `System.out.println`. If the entity exists in the database, it will be printed; otherwise, the output will indicate that the entity was not found.
5. **Delete Operation**: Finally, the `delete` method of the `Template` class is used to delete the `Book` entity from the database based on its ID. This method removes the entity from the database.

Overall, this code snippet demonstrates how to use the `Template` class in Jakarta NoSQL to interact with the database, abstracting away the low-level details of database operations and providing a simplified interface for performing CRUD operations on entities.

```
@Inject
```

```
Template template;

//1. Create Book Entity
Book book = Book.builder()
        .id(id)
        .title("Java Concurrency in Practice")
        .author("Brian Goetz")
        .year(Year.of(2006))
        .edition(1)
        .build();
//2. Insert Operation
template.insert(book);
//3. Find Operation
Optional<Book> optional = template.find(Book.class, id);
//4. Print Result
System.out.println("The result " + optional);
//5. Delete Operation
template.delete(Book.class, id);
```

The `Template` class in Jakarta NoSQL simplifies CRUD (Create, Read, Update, Delete) operations by providing a fluent API for interacting with the underlying NoSQL database. This API allows developers to perform advanced queries and deletion operations beyond the basic ID attribute.

```
@Inject
Template template;

List<Book> books = template.select(Book.class)
        .where("author")
        .eq("Joshua Bloch")
        .and("edition")
        .gt(3)
        .result();

template.delete(Book.class)
        .where("author")
        .eq("Joshua Bloch")
        .and("edition")
        .gt(3)
        .execute();
```

The fluent API feature for searching and removing entities provided by the `Template` class in Jakarta NoSQL offers excellent flexibility and convenience for CRUD operations. However, it's essential to note that this feature may only be fully supported for some types of NoSQL databases, as the capabilities of the underlying database technology may limit certain operations.

In cases where the underlying NoSQL database does not support advanced querying or deletion beyond the basic ID attribute, attempting to use these features with the `Template` class will result in an `UnsupportedOperationException` being thrown by Jakarta NoSQL. This exception indicates that the current database type does not support the requested operation.

Some NoSQL databases may not support all filter operations, such as logical OR operations in the fluent API. In such cases, attempting to use unsupported operations with the `Template` class will result in an `UnsupportedOperationException` being thrown by Jakarta NoSQL. This exception indicates that the current Jakarta NoSQL provider does not support the requested operation due to limitations imposed by the underlying NoSQL database technology.

Developers should be aware that while Jakarta NoSQL aims to provide a unified API across different NoSQL databases, there may be variations in support for certain operations depending on the capabilities of the specific database provider. When encountering limitations or unsupported operations, developers may need to adjust their application logic or consider alternative approaches to achieve the desired functionality within the constraints of the chosen NoSQL database technology.

4.1. Template and Inheritance classes

In case of querying an entity using the `@Inheritance` annotation as defined by @Inheritance, the Jakarta NoSQL provider must automatically include the condition where the value from `@DiscriminatorColumn` equals the value of `@DiscriminatorValue`.

For example, given the sample code from @Inheritance and executing the query for `SmsNotification`, the generated query should include a condition such as `type = 'SMS'` or its equivalent in the respective NoSQL database.

```
List<SmsNotification> notifications = template.select(SmsNotification.class);
```

It ensures that only entities of type `SmsNotification`, as indicated by the discriminator value, are retrieved from the database.

4.2. Fluent API Query

The `Template` class in Jakarta NoSQL provides a fluent API for querying and deleting entities from the underlying NoSQL database. This fluent API offers a convenient and expressive way for Java developers to interact with their data, allowing them to construct complex queries efficiently and perform deletion operations.

4.2.1. Importance of Fluent API Query

The fluent API query is essential for Java developers as it simplifies retrieving and manipulating data from the NoSQL database. By providing a fluent interface, Jakarta NoSQL enables developers to express their query logic concisely and readably, making it easier to understand and maintain the codebase.

The fluent API query also allows developers to build dynamic queries at runtime by chaining together various methods and conditions. This flexibility enables applications to adapt to changing requirements and user inputs, providing a more robust and responsive user experience.

Furthermore, the fluent API query promotes code reuse and modularity by encapsulating query

logic within reusable components. Developers can define and combine reusable query fragments to construct more complex queries, reducing duplication and improving code maintainability.

4.2.2. Limitations in Key-Value Databases

It is worth noting that the `select` and `delete` methods of the `Template` class may not be fully compatible with key-value databases. This limitation arises because key-value databases primarily rely on key-value pairs for data retrieval and deletion rather than complex query predicates.

The primary data access mode in key-value databases is through direct lookup by key. It is challenging to support complex query operations like those provided by the fluent API query. As a result, attempts to use the `select` and `delete` methods with key-value databases may throw an `UnsupportedOperationException` by Jakarta NoSQL, indicating that the underlying database technology does not support the operation.

4.2.3. Supported Methods in Other NoSQL Databases

The fluent API query offers a wide range of supported methods through the `QueryMapper` class for other types of NoSQL databases, such as document-oriented or column-family databases. These methods may include filtering, sorting, and basic querying capabilities, providing developers with flexible data retrieval and manipulation tools.

However, it's essential to consider that the availability of certain query methods may vary depending on the specific NoSQL database being used. NoSQL databases that do not support certain operations can raise `UnsupportedOperationException`.

Attempting to use unsupported operations with the fluent API query may result in runtime exceptions or unexpected behavior. Developers should consult the documentation of their chosen NoSQL database to understand its query capabilities and limitations and adjust their application logic accordingly.

4.2.4. Query Navigation Hierarchy

In Jakarta NoSQL, the query navigation hierarchy refers to navigating through the properties of entities and their associated classes when constructing queries. Within an entity, property names must be unique, ignoring cases. For simple entity properties, the field or accessor method name serves as the entity property name. In the case of embedded and association classes, entity property names are computed by concatenating the field or accessor method names at each level, optionally joined by a dot or period, `.`, delimiter.

Within a given entity or embeddable class, names assigned to persistent fields must be unique, ignoring cases.

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

For example, consider the following data model:

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

class MailingAddress {
  private String zipcode;
  private String city;
}
```

In this scenario, querying for records based on the zip code of the `MailingAddress` class requires accessing the `address` field of `Person` and the `zipcode` property of `MailingAddress`.

```
@Inject
Template template;

List<Book> books = template.select(Person.class)
        .where("address.zipcode")
        .eq("402-775")
        .orderBy("address.city")
        .asc()
        .result();

template.delete(Person.class)
     .where("address.zipcode")
     .eq("402-775")
     .execute();
```

In the above example, the fluent API query navigates through the properties of the `Person` entity to access the `zipcode` property of the `MailingAddress` embedded class. The `where` clause specifies the path to the `zipcode` property using dot notation (`address.zipcode`). The `orderBy` clause similarly specifies the path to the `city` property for sorting the results by city in ascending order.

This query navigation hierarchy enables developers to construct complex queries traverse multiple levels of entity properties, facilitating flexible and precise data retrieval and manipulation in Jakarta NoSQL.

4.3. TTL (Time-To-Live) Support

TTL (Time-To-Live) is a feature provided by many NoSQL databases that allows developers to set an expiration time for data stored in the database. When data reaches its TTL, it is automatically removed from the database, freeing up resources and ensuring that it remains efficient and clutter-free.

For Java developers, TTL support is essential for managing data lifecycle and optimizing resource usage. It enables developers to implement caching strategies, manage temporary data, and enforce data retention policies effectively.

While TTL support is valuable, not all NoSQL databases provide native support for TTL. In cases where TTL is not supported, attempting to set a TTL on data may result in an `UnsupportedOperationException` being thrown by the Jakarta NoSQL provider.

Additionally, some NoSQL providers may have limitations on the granularity of TTL values, such as supporting only TTL values specified in certain units (e.g., hours) or rounding TTL values to the nearest supported unit. In such cases, attempting to set a TTL value that does not align with the provider's limitations may result in unexpected behavior or no TTL being applied.

For example, suppose a NoSQL database only supports TTL values specified in hours. If a developer attempts to set a TTL of 10 seconds, the Jakarta NoSQL provider may throw an UnsupportedOperationException. Similarly, if the developer attempts to set a TTL of 3660 seconds (which is more than one hour), the TTL value may be rounded to the nearest supported unit (i.e., one hour) by the provider.

```
@Inject
Template template;

// UnsupportedOperationException: TTL granularity not supported
template.insert(entity, Duration.ofSeconds(10L));

// Inserting data with a TTL of one hour
template.insert(entity, Duration.ofSeconds(3600));

// Inserting data with a TTL of one hour (rounded from 3660 seconds)
template.insert(entity, Duration.ofSeconds(3660));
```

Chapter 5. Jakarta NoSQL Providers

A Jakarta NoSQL provider might come as an integrated component of a Jakarta EE container or a separate component that integrates with the Jakarta EE container via standard or proprietary SPIs. For example, a Jakarta NoSQL provider might use a CDI portable extension to integrate with dependency injection.

Jakarta NoSQL providers play a crucial role in the ecosystem by interpreting the annotations provided by developers and implementing the corresponding Template interfaces. These providers handle operations related to entities according to the rules outlined in the Jakarta NoSQL specification.

By adhering to these rules, Jakarta NoSQL providers ensure seamless integration with the application via dependency injection. This integration allows developers to access the functionality provided by the Template interfaces without concerning themselves with the underlying database implementation details.

The Jakarta NoSQL specification sets clear guidelines for Jakarta NoSQL providers, ensuring consistency and compatibility across different providers. These rules enable multiple Jakarta Data providers to coexist within a system without interfering or overlapping at the same injection points. This level of standardization fosters interoperability and flexibility, empowering developers to choose the provider that best suits their project requirements.

5.1. Configuration and Credentials

Configuration and credentials for NoSQL databases are not standardized within the Jakarta NoSQL specification. Each Jakarta NoSQL provider is responsible for providing its own configuration mechanism, allowing developers to configure the connection to the NoSQL database according to their specific requirements.

The Jakarta NoSQL specification highly recommends following the Twelve-Factor App methodology, particularly the "Store config in the environment" principle. This approach advocates for storing configuration details such as database credentials, connection URLs, and other settings as environment variables. This practice promotes portability, scalability, and security by separating configuration from code and ensuring consistency across different environments.

5.2. Schema Generation

The process of creating a schema for a database is called schema generation. This process is not included in the Jakarta NoSQL specification. While some NoSQL databases allow schema definition and enforcement, many NoSQL databases are schemaless, which means developers can store data without defining a schema beforehand.

Therefore, the ability to generate a schema may differ depending on the NoSQL database and the Jakarta NoSQL provider being used. Some Jakarta NoSQL providers offer schema generation capabilities, while others do not.

For developers working with schemaless NoSQL databases, schema generation may not be

necessary. This is because the database dynamically adapts to the structure of the data being stored. In such cases, the focus should be on organizing data to suit the application's requirements best, rather than defining a rigid schema.

In cases where schema generation is supported, developers should be aware that the process may vary between NoSQL databases and Jakarta NoSQL providers. Different databases may have unique requirements or conventions for defining schemas, and Jakarta NoSQL providers may offer different approaches or tools for schema generation.

Whether schema generation is necessary or beneficial depends on the specific use case, the NoSQL database being used, and the development team's preferences. Developers should consult the documentation of their chosen NoSQL database and Jakarta NoSQL provider for guidance on schema generation practices and considerations.

5.3. Jakarta NoSQL Providers Extensions

Jakarta NoSQL providers and NoSQL databases have the flexibility to extend the API according to their specific requirements. This extensibility allows providers to create new annotations or develop specialized versions of the Template API tailored to their unique features or functionalities.

However, it's essential to note that these extensions are specific to the respective provider or database and may not be compatible with others. As a result, there is no guarantee of compatibility between extensions developed by different Jakarta NoSQL providers or NoSQL databases.

Despite the lack of cross-compatibility, this extensibility empowers providers to effectively innovate and address specific use cases or requirements. By leveraging extensions, developers can harness the full potential of Jakarta NoSQL while benefiting from the diverse capabilities offered by different providers and databases.

5.4. Persistent Fields

A Jakarta NoSQL provider can read the annotation via runtime, for example, using reflection, or via build-time, for example, Java Annotation Processor.

Jakarta NoSQL provider runtime accesses the persistent state of an entity via either:

- property access using style property accessors defined by the Jakarta NoSQL provider for its respective field or
- field access, that is, direct access to instance variables.

When property access is used, the Jakarta NoSQL provider must define the conversion; the recommendation is the method signature convention for JavaBeans read/write properties, as determined by the JavaBeans `Introspector` class.

The `Column` annotation should be at the field in both access types.

Chapter 6. Interoperability with other Jakarta EE Specifications

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

6.1. Jakarta Contexts and Dependency Injection

Contexts and Dependency Injection (CDI) is a foundational specification within the Jakarta EE Core profile, offering a robust dependency injection framework for Java applications. CDI facilitates the decoupling of components and manages their lifecycle through dependency injection, promoting loose coupling and enabling the creation of modular, reusable code.

CDI is crucial in integrating the Jakarta NoSQL template seamlessly into applications through the `@Inject` annotation in Jakarta EE environments. This integration allows developers to inject instances of the `Template` class directly into their application components, enabling straightforward access to its methods and functionalities.

With CDI and the `@Inject` annotation, developers can inject the `Template` instance and utilize its methods effortlessly, as illustrated in the following example:

```
@Inject
Template template;

// ...

List<Car> cars = template.select(Car.class).where("type").eq(CarType.SPORT).result();
```

The Jakarta NoSQL provider should also provide CDI qualifiers to work with multiple NoSQL databases through CDI. Developers can use these qualifiers to specify which database instance they want to inject, enabling flexibility and compatibility with different NoSQL data stores. Jakarta NoSQL providers typically supply annotations like `@DatabaseQualifier` to annotate the injection points.

For example:

```
@Inject
@DatabaseQualifier
Template template;

@Inject
@DatabaseQualifier("another")
Template anotherTemplate;
```

The template implementation bean must have:

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

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

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

6.1.1. CDI Extensions for Jakarta Data Providers

In environments where CDI Full or CDI Lite is available, Jakarta NoSQL providers can leverage CDI extensions to enhance the integration and discovery of entities or implementations. While Jakarta NoSQL does not prescribe a specific type of CDI extension, it does require Jakarta NoSQL providers to ensure that template implementations are injected into appropriate injection points, typically interfaces, without additional qualifiers.

It's important to note the distinction between CDI Full and CDI Lite: CDI Full, part of the Jakarta Web profile and Jakarta Platform, includes support for `jakarta.enterprise.inject.spi.Extension`, whereas CDI Lite (Jakarta Core profile) does not. However, both CDI Full and CDI Lite support `jakarta.enterprise.inject.build.compatible.spi.BuildCompatibleExtension`.

To avoid conflicts between the `BuildCompatibleExtension` and portable `Extension` in CDI Full environments, Jakarta NoSQL providers can utilize CDI's `@SkipIfPortableExtensionPresent` annotation.

CDI provides a robust foundation for integrating Jakarta NoSQL repositories into Jakarta EE applications, offering flexibility and extensibility to meet diverse requirements and use cases.

6.2. Jakarta Bean Validation

Integrating Jakarta Bean Validation into Jakarta NoSQL ensures data consistency within the Java layer by enforcing validation rules and constraints on data objects. By applying validation rules, developers can maintain data integrity, improve data quality, and enhance the reliability of their applications.

Jakarta Validation offers several advantages for Jakarta NoSQL applications:

1. It helps identify and prevent invalid or inconsistent data from being processed or persisted, reducing the risk of data corruption.
2. Catching validation errors early in the Java layer allows developers to identify and resolve potential issues before further processing or persistence occurs, leading to more robust and reliable applications.
3. Jakarta Validation supports declarative validation rules, simplifying the validation logic and promoting cleaner, more maintainable code.

In Jakarta NoSQL, template implementations are subject to method validation as specified in the "Method and constructor validation" section of the Jakarta Validation specification. This validation includes checking for constraints on method parameters and results. Automatic validation using

these constraints is done by delegating validation to the Bean Validation implementation when inserting, updating, or deleting data through the methods.

Let's consider an example demonstrating the usage of Jakarta Bean 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.

To execute validation before inserting data using Jakarta NoSQL templates, developers can simply invoke the `insert` and `update` methods on the template instance:

```
@Inject
private Template template;
...
// Execute the validation before inserting the data.
template.insert(student);
```

6.3. Jakarta Data

Developers can seamlessly incorporate common data patterns, such as repositories, into their codebase by integrating Jakarta NoSQL with Jakarta Data.

Jakarta NoSQL providers that support Jakarta Data typically scan interfaces marked with the `jakarta.data.repository.Repository` annotation. This annotation serves as a marker for repositories, providing a standardized way to define repository interfaces.

By embracing Jakarta Data, Jakarta NoSQL providers enable Java developers to use standardized data patterns and techniques when defining entities and repositories. This compatibility ensures interoperability with other technologies and frameworks, fostering a cohesive and streamlined

development experience.