JAKARTA EE

Jakarta NoSQL

Contributors to Jakarta NoSQL Specification (https://github.com/eclipse-ee4j/nosql/graphs/contributors)

1.0.0-SNAPSHOT, June 18, 2023: Draft

Table of Contents

Specification: Jakarta NoSQL

Version: 1.0.0-SNAPSHOT

Status: Draft

Release: June 18, 2023

Chapter 1. Introduction

1.1. One API, Multiple Databases

Jakarta NoSQL is a Java framework that streamlines the integration of Java applications with NoSQL databases.

It uses the same annotations to map Java objects. Therefore, with just these annotations, whose names match those from the Jakarta Persistence specification, there is support for several NoSQL database types.

```
@Entity
public class Book {

    @Id
    private String isbn;

    @Column
    private String title;

    @Column
    private String description;
 //...
}
```

Developers must consider low cognitive load when choosing a NoSQL database for their applications. For example, if there is a need to switch out a database, considerations include the time spent on the change; the learning curve of a new database API; the code that will be lost; the persistence layer that needs to be replaced, etc. Jakarta NoSQL avoids most of these issues through the API.

Jakarta NoSQL also provides `Template` classes that apply the **Template Method** design pattern to all database operations.

1.2. Beyond Jakarta Persistence (JPA)

The Jakarta Persistence specification is an excellent API for object-relational mapping and has established itself as a Jakarta EE standard. It would be ideal to use the same API for both SQL and NoSQL, but there are behaviors in NoSQL that SQL does not cover, such as time-to-live and asynchronous operations. Jakarta Persistence was simply not designed to handle those features.

```
Template template = // instance; a template to document NoSQL operations
Deity diana = Deity.builder()
        .withId("diana")
        .withName("Diana")
        .withPower("hunt")
        .build();
```

```
Duration ttl = Duration.ofSeconds(1);
template.insert(diana, ttl);
```

1.3. A Fluent API

Jakarta NoSQL provides a fluent API for Java developers to more easily create queries that either retrieve or delete information in a NoSQL database type. For example, given the scenario below:

```
Template template = // instance; a template to document NoSQL operations
Deity diana = Deity.builder()
        .withId("diana")
        .withName("Diana")
        .withPower("hunt")
        .build();

template.insert(diana);// insert an entity
```

It's possible to retrieve information by using the Fluent API which is a uniform way to define query criteria independent of each NoSQL database type:

```
List<Deity> deities = template.select(Deity.class)
        .where("name")
        .eq("Diana").result(); // SELECT Deity WHERE name equals Diana
```

And, with this uniform API, it's possible to perform deletions by providing a flexible criteria selection:

```
template.delete(Deity.class).where("name")
        .eq("Diana").execute();
```

1.4. Key Features

- Simple APIs that support all well-known NoSQL storage types;
- Specialized NoSQL types: Key-Value, Column Family, Document databases;
- Use of Convention Over Configuration;
- Easy-to-implement API Specification and Technology Compatibility Kit (TCK) for NoSQL Vendors;
- The API's focus is on simplicity and ease of use. Developers should only know minimal artifacts to work with Jakarta NoSQL.

Chapter 2. Jakarta NoSQL's API

Put differently, Jakarta NoSQL's API has the same goals as the JPA or ORM. In the NoSQL world, the **OxM** then converts the entity object to a communication model that goes to the database.

2.1. Annotations

Jakarta NoSQL is a popular specification that provides a standardized way for Java developers to work with non-relational databases. One of the key features of this specification is the use of Java annotations, which provide a simple and intuitive way for developers to map Java objects to NoSQL databases. These annotations allow developers to specify the structure of their data and the relationships between different objects without having to write complex code or queries.

Using Java annotations, developers can significantly simplify their workflow and reduce the time and effort required to work with NoSQL databases. Annotations also make it easier to maintain code and reduce the risk of errors since the database structure is defined clearly and concisely. Additionally, annotations provide high flexibility and customization, allowing developers to tailor the database schema to their specific needs.

Using Java annotations in Jakarta NoSQL represents a significant step forward for developers working with non-relational databases. By providing a standardized approach to mapping Java objects to NoSQL databases, this specification makes working with these robust data storage solutions more effortless and efficient.

Jakarta NoSQL has support for those three types:

- @Entity
- @Column
- @Id

2.1.1. @Entity

This annotation maps the class to Jakarta NoSQL. There is a single value attribute that specifies the column family name, the document collection name, etc. The default value is the simple name of the class. For example, given the `org.jakarta.nosql.demo.Person` class, the default name will be `Person`.

```
@Entity
public class Person {
}
```

In the case of name customization, it just needs to set the value of the @Entity annotation with the desired name as like below:

```
@Entity("ThePerson")
public class Person {
```

```
}
```

You can include one or multiple entities without requiring additional annotations like `OneToOne` or `OneToMany` in JPA when using the API. However, it's essential to remember that NoSQL databases have varying behaviors. For instance, in a Document database, these entities may be converted into a subdocument, while on a Key-value, it will be the value.

The sample below shows two entities, Person and Address, where a person has an address.

```
@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;
}
```

The serialization method may differ depending on the NoSQL vendor.

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

2.1.2. @Column

This annotation defines which fields that belong to an Entity will be persisted. The field name specifies the column name by default.

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

    @Column
    private String name;

    @Column
    private List<String> phones;

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

If any customization is needed, it just set the single attribute of the annotation to specify the desired name:

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

    @Column("personName")
    private String name;

    @Column
    private List<String> phones;

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

2.1.3. @Id

This annotation defines which attribute is the entity's ID, or the Key in Key-Value databases. Unlike `@Column`, the default value is `_id`.

```
@Entity
public class User {

    @Id
    private String userName;

    @Column
    private String name;
```

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

And, like `@Column`, if the ID's name requires customization, it just set the single attribute of the annotation to specify the desired name:

```
@Entity
public class User {

    @Id("userId")
    private String userName;

    @Column
    private String name;

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

2.2. Template Classes

The Template feature in Jakarta NoSQL simplifies the implementation of common database operations by providing a basic API to the underlying persistence engine. It follows the standard template pattern, a common design pattern used in software development.

The Template pattern involves creating a skeletal structure for an algorithm, with some steps implemented and others left to be implemented by subclasses. Similarly, the Template feature in Jakarta NoSQL makes a skeleton around NoSQL database operations, allowing developers to focus on implementing the specific logic required for their application.

The Template feature can be related to the Template Method design pattern, a variation of the Template pattern. In the Template Method pattern, an abstract class defines a template method that encapsulates a series of steps required to perform a task, with some steps implemented and others left to be implemented by subclasses.

The Template feature in Jakarta NoSQL also defines a template method for performing NoSQL database operations, with some steps implemented and others left to be implemented by the developer.

Overall, the Template feature in Jakarta NoSQL provides a simple and efficient way to implement common database operations while following established design patterns like the Template Method. By using the Template feature, developers can save time and effort in implementing their NoSQL database operations, allowing them to focus on other aspects of their application.

```
@Inject
Template template;
```

```
Book book = Book.builder()
        .id(id)
        .title("Java Concurrency in Practice")
        .author("Brian Goetz")
        .year(Year.of(2006))
        .edition(1)
        .build();

template.insert(book);
Optional<Book> optional = template.find(Book.class, id);
System.out.println("The result " + optional);
template.delete(Book.class, id);
```

Furthermore, in CRUD operations, Template provides a fluent-API for either select or delete entities. Thus, Template offers the capability for search and remove beyond the 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();
```

2.2.1. Key-Value Template

The `KeyValueTemplate` class is a type of `Template` made to make key-value databases easier to use. These databases store data as key-value pairs, where each key represents a unique identifier for a piece of data.

The `KeyValueTemplate` class offers developers a straightforward method to handle NoSQL databases and streamline dealing with intricate data structures. This class is based on the Template, which provides a versatile and adaptable approach to define regular tasks that can be executed on a database.

One of the key benefits of the `KeyValueTemplate` class is that it is designed to be highly customizable. Vendors who wish to explore particular NoSQL databases can use the KeyValue class as a starting point and then build on top of it to add support for additional features or behaviors specific to their

particular database.

Because the behavior of NoSQL databases can vary widely depending on the specific implementation, it is essential to have a flexible API that allows for specialization. The KeyValue class provides this flexibility and enables developers to build robust and efficient applications that take advantage of the unique features of key-value databases.

It is important to note that key-value databases are primarily oriented toward working with the key and must be better suited for performing complex queries or data analysis. As a result, key-value databases might not support some methods and may return an `UnsupportedOperationException`.

```
@Inject
KeyValueTemplate template;
...

User user = new User();
user.setNickname("ada");
user.setAge(10);
user.setName("Ada Lovelace");
List<User> users = Collections.singletonList(user);

template.put(user);
template.put(users);

Optional<Person> ada = template.get("ada", Person.class);
Iterable<Person> usersFound = template.get(Collections.singletonList("ada"), Person
.class);
```

In key-value templates, both the `@Entity` and `@Id` annotations are required. The `@Id` identifies the key, and the whole entity will be the value. The API won't cover how the value persists this entity.

2.2.2. ColumnTemplate

The `ColumnTemplate` class is a type of Template that makes it easier to work with Column-Family or Wide-Column databases. These NoSQL databases store data in a table-like structure, where each column contains a particular attribute or piece of information.

The `ColumnTemplate` class is designed to help developers quickly work with NoSQL databases and manage complex data structures. It uses the `Template`, allowing flexible and customizable joint operations on the database.

The `ColumnTemplate` class offers excellent flexibility as it can be easily customized according to the needs of different vendors. It allows vendors to use it as a foundation for their NoSQL databases and add specific features or behaviors unique to their databases.

To effectively utilize the specific features of Column-Family or Wild-Column databases, developers require a flexible API that accommodates the variations in the behavior of NoSQL databases. The `ColumnTemplate` class offers this adaptability, allowing the creation of durable and high-performing

applications.

```
@Inject
ColumnTemplate template;
...
Person person = new Person();
person.setAddress("Olympus");
person.setName("Artemis Good");
person.setPhones(Arrays.asList("55 11 94320121", "55 11 94320121"));
person.setNickname("artemis");

List<Person> people = Collections.singletonList(person);

Person personUpdated = template.insert(person);
template.insert(people);
template.insert(person, Duration.ofHours(1L));

template.update(person);
template.update(people);
```

The `select` and `delete` methods are two key features of the fluent API provided by the NoSQL Template. These methods enable developers to easily find and delete information from a NoSQL database using a simple and intuitive syntax.

The select method allows developers to query a database and return results that match a specified set of criteria. This method typically takes one or more parameters defining the query's standards, such as a key, value, or other attribute. The query results are usually returned as a collection of objects matching the specified criteria, making it easy for developers to iterate over the results and extract the necessary information.

On the other hand, the delete method allows developers to remove one or more items from a database based on a specified set of criteria. This method typically takes one or more parameters defining the deletion criteria, such as a key, value, or other attribute. The items that match the specified criteria are then removed from the database, making it easy for developers to clean up their data and maintain the integrity of their database.

Both the select and delete methods are designed to be used as part of a fluent API, meaning they can be chained together with other ways to create powerful and flexible queries. This allows developers to build complex queries that are tailored to the specific needs of their application without having to worry about the underlying details of the database.

```
@Entity
public class Person {

    @Id("native_id")
    private long id;

    @Column
    private String name;
```

```
    @Column
    private int age;
}
```

```
@Inject
ColumnTemplate template;
...
List<Person> people = template.select(Person.class)
        .where("id")
        .gte(10)
        .result();

// returning from Person entity where the ID is greater than ten as List

template.delete(Person.class)
        .where("id")
        .eq("20")
        .execute();

//deleting from Person entity where the ID is equals to twenty
```

2.2.3. DocumentTemplate

The `DocumentTemplate` class is a type of `Template` explicitly made for simplifying document-oriented tasks on NoSQL databases. These databases store data as documents, individual data sets that can be grouped into collections.

Developers can easily work with document-oriented NoSQL databases and streamline complex data structures using the `DocumentTemplate` class. This class is built on the `Template`, offering a flexible and extensible approach to defining common database operations.

The `DocumentTemplate` class offers excellent flexibility as it can be easily customized according to specific needs. Vendors can use it as a foundation to create support for additional features and behaviors that are unique to their NoSQL databases.

Because the behavior of document-oriented NoSQL databases can vary widely depending on the specific implementation, it is crucial to have a flexible API that allows for specialization. The `DocumentTemplate` class provides this flexibility and enables developers to build robust and efficient applications that take advantage of the unique features of document-oriented NoSQL databases.

One of the primary advantages of document-oriented NoSQL databases is their ability to handle semi-structured or unstructured data, which can be challenging to store and query in traditional relational databases. The DocumentTemplate class is designed to take advantage of this flexibility, providing a simple and intuitive way to work with documents, collections, and queries.

```
@Inject
DocumentTemplate template;
```

```
...

Person person = new Person();
person.setAddress("Olympus");
person.setName("Artemis Good");
person.setPhones(Arrays.asList("55 11 94320121", "55 11 94320121"));
person.setNickname("artemis");

List<Person> people = Collections.singletonList(person);

Person personUpdated = template.insert(person);
template.insert(people);
template.insert(person, Duration.ofHours(1L));

template.update(person);
template.update(people);
```

The `select` and `delete` methods are two key features of the fluent API provided by the NoSQL Template. These methods enable developers to easily find and delete information from a NoSQL database using a simple and intuitive syntax.

The select method allows developers to query a database and return results that match a specified set of criteria. This method typically takes one or more parameters defining the query's standards, such as a key, value, or other attribute. The query results are usually returned as a collection of objects matching the specified criteria, making it easy for developers to iterate over the results and extract the necessary information.

On the other hand, the delete method allows developers to remove one or more items from a database based on a specified set of criteria. This method typically takes one or more parameters defining the deletion criteria, such as a key, value, or other attribute. The items that match the specified criteria are then removed from the database, making it easy for developers to clean up their data and maintain the integrity of their database.

Both the select and delete methods are designed to be used as part of a fluent API, meaning they can be chained together with other ways to create powerful and flexible queries. This allows developers to build complex queries that are tailored to the specific needs of their application without having to worry about the underlying details of the database.

```
@Entity
public class Person {

    @Id("native_id")
    private long id;

    @Column
    private String name;

    @Column
    private int age;
```

```
}
```

```
@Inject
private DocumentTemplate template;

public void mapper() {
List<Person> people = template.select(Person.class)
        .where("id")
        .gte(10)
        .result();

// translating: select().from("Person").where("native_id").gte(10L).build();

template.delete(Person.class)
        .where("id")
        .eq("20")
        .execute();

// translating: delete().from("Person").where("native_id").gte(10L).build();
}
```

2.2.4. Querying by Text with the Mapping API

Query by text is a powerful feature in the Jakarta NoSQL API that allows Java developers to perform CRUD operations on NoSQL databases using a text-based query language. Although the syntax may vary between different NoSQL vendors, the Jakarta NoSQL API provides a standard set of interfaces and methods that simplify performing CRUD operations through text queries.

Apart from supporting static queries, the Jakarta NoSQL API offers an API for dynamic queries. This feature enables developers to bind parameters to a query at runtime, making the queries more flexible and adaptable to changing requirements.

It is important to note that the Jakarta NoSQL specification does not specify the query syntax. Instead, each NoSQL vendor can define their query syntax and return types, allowing them to tailor their syntax to their platform's unique features and capabilities. This flexibility enables vendors to provide robust and efficient query capabilities that meet the needs of their users.

Overall, the Jakarta NoSQL API's support for query by text provides Java developers with a flexible and intuitive way to perform CRUD operations on NoSQL databases. Although the query syntax is vendor-specific, the common interfaces and methods provided by the Jakarta NoSQL API make it easy to work with any NoSQL database that supports queries by text. The ability to dynamically bind parameters to a query at runtime adds extra flexibility, allowing developers to adapt their queries to changing requirements.

2.2.4.1. Key-Value Database Types

```
@Inject
KeyValueTemplate template;
```

```
Stream<User> users = template.query("QUERY BY PROVIDER");
```

2.2.4.2. Column-Family Database Types

```
@Inject
ColumnTemplate template;

Stream<Person> result = template.query("QUERY BY PROVIDER");
```

2.2.4.3. Document Database Types

```
@Inject
DocumentTemplate template;

Stream<Person> result = template.query("QUERY BY PROVIDER");
```

Chapter 3. References

3.1. Frameworks

Spring Data

http://projects.spring.io/spring-data/

Hibernate OGM

http://hibernate.org/ogm/

Eclipselink

http://www.eclipse.org/eclipselink/

Jdbc-json

https://github.com/jdbc-json/jdbc-cb

Simba

http://www.simba.com/drivers/

Apache Tinkerpop

http://tinkerpop.apache.org/

Apache Gora

http://gora.apache.org/about.html

3.2. Databases

ArangoDB

https://www.arangodb.com/

Blazegraph

https://www.blazegraph.com/

Cassandra

http://cassandra.apache.org/

CosmosDB

https://docs.microsoft.com/en-us/azure/cosmos-db/introduction

Couchbase

https://www.couchbase.com/

Elastic Search

https://www.elastic.co/

Grakn

https://grakn.ai/

Hazelcast

https://hazelcast.com/

Hbase

https://hbase.apache.org/

Infinispan

http://infinispan.org/

JanusGraph IBM

https://www.ibm.com/cloud/compose/janusgraph

Janusgraph

http://janusgraph.org/

Linkurio

https://linkurio.us/

Keylines

https://cambridge-intelligence.com/keylines/

MongoDB

https://www.mongodb.com/

Neo4J

https://neo4j.com/

OriendDB

https://orientdb.com/why-orientdb/

RavenDB

https://ravendb.net/

Redis

https://redis.io/

Riak

http://basho.com/

Scylladb

https://www.scylladb.com/

Stardog

https://www.stardog.com/

TitanDB

http://titan.thinkaurelius.com/

Memcached

https://memcached.org/

3.3. Articles

Graph Databases for Beginners: ACID vs. BASE Explained

https://neo4j.com/blog/acid-vs-base-consistency-models-explained/

Base: An Acid Alternative

https://queue.acm.org/detail.cfm?id=1394128

Understanding the CAP Theorem

https://dzone.com/articles/understanding-the-cap-theorem

Wikipedia CAP theorem

https://en.wikipedia.org/wiki/CAP_theorem

List of NoSQL databases

http://nosql-database.org/

Data access object Wiki

https://en.wikipedia.org/wiki/Data_access_object

CAP Theorem and Distributed Database Management Systems

https://towardsdatascience.com/cap-theorem-and-distributed-database-management-systems-5c2be977950e

Oracle Java EE 9 NoSQL view

https://javaee.github.io/javaee-spec/download/JavaEE9.pdf

3.4. 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.

3.4.1. Project Lead

- Otavio Santana

3.4.2. Contributors

- Ivar Grimstad
- Kevin Sutter

- Scott Stark

3.4.3. Committers

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

3.4.4. Historical Committer

- Leonardo Lima

3.4.5. Mentor

- Wayne Beaton

3.4.6. Full List of Contributors

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