JAKARTA EE

Jakarta NoSQL

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

1.0.0-b6, February 10, 2023: Draft

Table of Contents

Specification: Jakarta NoSQL

Version: 1.0.0-b6

Status: Draft

Release: February 10, 2023

Chapter 1. Introduction

1.1. One Mapping API, Multiple Databases

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

Jakarta NoSQL defines an API for each NoSQL database type:

- Key-Value
- Column Family
- Document

However, 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 more than twenty NoSQL databases.

```
@Entity
public class Deity {

    @Id
    private String id;

    @Column
    private String name;

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

Developers need to consider vendor lock-in when choosing a NoSQL database for their application. 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 **Mapping** 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.

```
ColumnTemplate 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 is a fluent API for Java developers to more easily create queries that either retrieve or delete information in a `Document` database type. For example:

```
DocumentTemplate 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

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

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

1.4. Key Features

- Simple APIs that support all well-known NoSQL storage 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 APIs focus is on simplicity and ease-of-use. Developers should only have to know a minimal set of artifacts to work with Jakarta NoSQL.

1.5. 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.

1.5.1. Project Lead

- Otavio Santana

1.5.2. Contributors

- Ivar Grimstad
- Kevin Sutter
- Scott Stark

1.5.3. Committers

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

1.5.4. Historical Committer

- Leonardo Lima

1.5.5. Mentor

- Wayne Beaton

1.5.6. Full List of Contributors

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

Chapter 2. Introduction to the Mapping API

The mapping level, to put it differently, has the same goals as either the JPA or ORM. In the NoSQL world, the **OxM** then converts the entity object to a communication model.

This level is responsible to perform integration among technologies such as Bean Validation. The Mapping API provides annotations that make the Java developer's life easier. As a communication project, it must be extensible and configurable to keep the diversity of NoSQL databases.

2.1. Models Annotation

As previously mentioned, the Mapping API provides annotations that make the Java developer's life easier. These annotations can be categorized in two categories:

- Annotation Models
- Qualifier Annotation

2.1.1. Annotation Models

The annotation model converts the entity model into the entity on communication, the communication entity:

- @Entity
- @Column
- @Id

The Jakarta NoSQL Mapping API does not require getter and setter methods to fields. However, the Entity class must have a non-private constructor with no parameters.

2.1.1.1. @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 {
}
```

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

An entity that is a field will be incorporated as a sub-entity. For example, in a Document, the entity

field will be converted to a sub-document.

```
@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;
}
```

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

2.1.1.2. @Column

This annotation defines which fields that belong to an Entity will be persisted. There is a single attribute that specifies that name in Database with a default value that is the field name as declared in the class. This annotation is mandatory for non-Key-Value database types. In Key-Value types, only the Key needs to be identified with the `@Key` annotation. All other fields are stored as a single BLOB.

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

    @Column("personName")
```

```
    private String name;

    @Column
    private List<String> phones;

    // ignored
    private String address;
}
```

2.1.1.3. @Id

This annotation defines which attribute is the entity's ID, or the Key in Key-Value databases. In such a case, the Value is the remaining information. It has a single attribute (like `@Column`) to define the native name. Unlike `@Column`, the default value is `_id`.

```
@Entity
public class User {

    @Id
    private String userName;

    @Column
    private String name;

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

2.2. Template Classes

The Template classes offer convenient creation, update, delete, and query operations for databases. The `Template` instance is the root implementation for all types. So, each database type will support this instance.

```
@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 two queries, 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

This template has the responsibility to serve as the persistence of an entity in a key-value database.

The `KeyValueTemplate` is the template for synchronous tasks.

```
@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.

To use a key-value template, just follow the CDI style and precede the field with the `@Inject` annotation.

```
@Inject
private KeyValueTemplate template;
```

2.2.2. ColumnTemplate

This template has the responsibility to serve as a bridge between the entity model and the communication to a column family NoSQL database type.

The `ColumnTemplate` is the column template for the synchronous tasks.

```
@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);
```

To remove and retrieve information from document collection, there are `select` and `delete` methods.

```
@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();

// 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();
```

To use a column template, just follow the CDI style and precede the field with the `@Inject` annotation.

```
@Inject
private ColumnTemplate template;
```

2.2.3. DocumentTemplate

This template has the responsibility to serve as a bridge between the entity model and the communication to a column family NoSQL database type.

The `DocumentTemplate` is the document template for the synchronous tasks.

```
@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);
```

To remove and retrieve information from document collection, there are `select` and `delete` methods.

```
@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();
}
```

To use a document template, just follow the CDI style and place an `@Inject` annotation on the field.

```
@Inject
private DocumentTemplate template;
```

2.2.4. Querying by Text with the Mapping API

Jakarta NoSQL provides query by text that you can execute. Furthermore, there is the option to explore a prepared statement query. Jakarta NoSQL does not provide any query support. Thus, any vendor might have diverse queries.

2.2.4.1. Key-Value Database Types

```
KeyValueTemplate template = // instance;
Stream<User> users = template.query("get \"Diana\"");
template.query("remove \"Diana\"");
```

2.2.4.2. Column-Family Database Types

```
ColumnTemplate template = // instance;
Stream<Person> result = template.query("select * from Person where id = 1");
```

2.2.4.3. Document Database Types

```
DocumentTemplate template = // instance;
Stream<Person> result = template.query("select * from Person where age > 10");
```

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