JAKARTA EE

Jakarta NoSQL

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

1.0.0-b5, December 23, 2022: Draft

Table of Contents

Specification: Jakarta NoSQL

Version: 1.0.0-b5

Status: Draft

Release: December 23, 2022

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.

The project has two layers that define communication with NoSQL databases through APIs. These are:

1. **Mapping Layer**: This layer is annotation-driven and uses technologies such as Jakarta Contexts and Dependency Injection and Jakarta Bean Validation, making it simple for developers to use. In the traditional RDBMS world, this layer may be compared to the Java Persistence API or object-relational mapping frameworks such as Hibernate.
2. **Communication Layer**: This layer contains four modules, one for each NoSQL database type: Key-Value, Column Family, Document and Graph. In the traditional the RDBMS world, this layer may be compared to the JDBC API.

This clearly helps to achieve very low application coupling with the underlying NoSQL technologies used in applications.

Jakarta NoSQL defines an API for each NoSQL database type. However, it uses the same annotations to map Java objects. Therefore, with just these annotations that look like JPA, 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;
 //...
}
```

Vendor lock-in is one of the things any Java project needs to consider when choosing NoSQL databases. If there is a need to switch out a database, other considerations include: 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 APIs of **Communication** and **Mapping** Layers.

Jakarta NoSQL also has template classes that apply the design pattern 'template method' to databases operations. And the `Repository` interface allows Java developers to create and extend

interfaces, with implementation automatically provided by a Jakarta NoSQL implementation: support method queries built by developers will automatically be implemented for them.

```
public interface DeityRepository extends Repository<Deity, String> {

    Optional<Deity> findById(String id);
    Optional<Deity> findByName(String name);
}
```

```
SeContainer container = SeContainerInitializer.newInstance().initialize()

Service service = container.select(Service.class).get();

DeityRepository repository = service.getDeityRepository();

Deity diana = Deity.builder()
        .withId("diana")
        .withName("Diana")
        .withPower("hunt")
        .build();

repository.save(diana);

Optional<Deity> idResult = repository.findById("diana");
Optional<Deity> nameResult = repository.findByName("Diana");
```

1.2. Beyond Jakarta Persistence (JPA)

The Jakarta Persistence specification is a good 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;
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

DocumentQuery query = select()
        .from(Deity.class)
        .where("name")
        .eq("Diana")
        .build(); // SELECT Deity WHERE name equals "Diana"

List<Deity> deities = template.select(query); // execute query

DocumentDeleteQuery delete = delete()
        .from("deity").where("name")
        .eq("Diana")
        .build(); // delete query

template.delete(delete);
```

1.4. Let's Not Reinvent the Wheel: Graph Database Type

The Communication Layer defines three new APIs: Key-Value, Document and Column Family. It does not have new Graph API, because a very good one already exists. Apache TinkerPop is a graph computing framework for both graph databases (OLTP) and graph analytic systems (OLAP). Using Apache TinkerPop as Communication API for Graph databases, the Mapping API has a tight integration with it.

1.5. Particular Behavior Matters in NoSQL Databases

Particular behavior matters. Even within the same type, each NoSQL database has a unique feature that may be a considerable factor when choosing one database over another. This "feature" might make it easier to develop, make it more scaleable or consistent from a configuration standpoint, have the desired consistency level or search engine, etc. Some examples include: Cassandra and its Cassandra Query Language and consistency level; OrientDB with live queries; ArangoDB and its Arango Query Language; Couchbase with N1QL; etc. Each NoSQL database has a specific behavior and this behavior matters, so Jakarta NoSQL was designed to be extensible enough to capture these substantially different feature elements.

```
public interface PersonRepository extends CouchbaseRepository {

    @N1QL("select * from Person")
    List<Person> findAll();

    @N1QL("select * from Person where name = $name")
    List<Person> findByName(@Param("name") String name);
}

Person person = // instance;
CassandraTemplate template = //instance;
ConsistencyLevel level = ConsistencyLevel.THREE;
template.save(person, level);
```

1.6. Key Features

- Simple APIs supporting all well-known NoSQL storage types - Column Family, Key-Value Pair, Graph and 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. The API is built on Java 8 features like Lambdas and Streams, and therefore fits perfectly with the functional features of Java 8+.

1.7. 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.

1.7.1. Project Lead

- Otavio Santana

1.7.2. Committers

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

- Werner Keil
- Michael Redlich

1.7.3. Historical Committer

- Leonardo Lima

1.7.4. Mentor

- Wayne Beaton

1.7.5. Contributors

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

Chapter 2. Introduction to the Communication API

With the strategy to divide and conquer on Jakarta NoSQL, the communication API was born. It has the goal to make the communication layer easy and extensible. The extensibility is more than important, that is entirely necessary once the API must support specific feature in each database. Nonetheless, the advantage of a common API in a change to another database provider has lesser than using the specific API.

To cover the three of the four database types, this API includes three packages, one for each database.

- `jakarta.nosql.column`
- `jakarta.nosql.document`
- `jakarta.nosql.keyvalue`

The package name might change on the Jakarta EE process.

A package for the Graph database type is not included in this API because we leverage the Graph communication API provided by Apache TinkerPop.

So, if a database is multi-model, i.e., has support for more than one database type, it will implement an API for each database it supports. Also, each API has the TCK to prove if the database is compatible with the API. Even from different NoSQL types, it tries to use the same nomenclature:

- Configuration: It is a function that reads from `Settings and then creates a manager factory instance.
- Manager's Factory: The manager factory instance creates a manager instance from the database name.
- Manager: The manager instance bridges the Jakarta NoSQL and the NoSQL vendor.
- Entity: The communication entity
- Value: the information unit

Structure	Key-value	Column	Document
Configuration	KeyValueConfiguration	ColumnConfiguration	DocumentConfiguration
Factory	BucketManagerFactory	ColumnManagerFactory	DocumentManagerFactory
Manager's Factory	BucketManager	ColumnManager	DocumentManager
Entity	KeyValueEntity	ColumnEntity	DocumentEntity

The codes below show several CRUD operations using the communication layer.

2.1. Document Communication sample

```
Settings settings = Settings.builder().put("credential", "value").build();
DocumentConfiguration configuration = new NoSQLDocumentProvider();
DocumentManagerFactory factory = configuration.apply(settings);
try (DocumentManager manager = factory.apply("database")) {
    DocumentEntity entity = DocumentEntity.of("entity");
    entity.add("id", 10L);
    entity.add("version", 0.001);
    entity.add("name", "Diana");
    manager.insert(entity);
    DocumentQuery query = select().from("entity").where("version").gte(0.001).build();

    Stream<DocumentEntity> entities = manager.select(query);
    DocumentDeleteQuery delete = delete().from("entity").where("id").eq(10L).build();

  manager.delete(delete);
}
```

2.2. Column Communication sample

```
Settings settings = Settings.builder().put("credential", "value").build();
ColumnConfiguration configuration = new NoSQLColumnProvider();
ColumnManagerFactory factory = configuration.apply(settings);
try (ColumnManager manager = factory.apply("database")) {
    ColumnEntity entity = ColumnEntity.of("entity");
    entity.add("id", 10L);
    entity.add("version", 0.001);
    entity.add("name", "Diana");

    manager.insert(entity);
    ColumnQuery query = select().from("entity").where("version").gte(0.001).build();
    Stream<ColumnEntity> entities = manager.select(query);

    ColumnDeleteQuery delete = delete().from("entity").where("id").eq(10L).build();

    manager.delete(delete);
}
```

2.3. Key-value Communication sample

```
Settings settings = Settings.builder().put("credential", "value").build();
KeyValueConfiguration configuration = new NoSQLKeyValueProvider();
BucketManagerFactory factory = configuration.apply(settings);
try (BucketManager manager = factory.apply("database")) {
     KeyValueEntity entity = KeyValueEntity.of(12, "Poliana");
```

```
        manager.put(entity);
        manager.delete(12);
    }
```

2.4. The API Structure

The communication has four projects:

- The **communication-core**: The Jakarta NoSQL API communication common to all database types.
- The **communication-key-value**: The Jakarta NoSQL communication API layer to a key-value database.
- The **communication-column**: The Jakarta NoSQL communication API layer to a column database.
- The **communication-document**: The Jakarta NoSQL communication API layer to a document database.

Each module works separately such that a NoSQL vendor just needs to implement the specific type. For example, a key-value provider will apply a key-value API. If a NoSQL driver already has a driver, this API can work as an adapter with the current one. For a multi-model NoSQL database, providers will implement the APIs they need.

To the Graph communication API, there is the Apache TinkerPop that won't be covered in this documentation.

2.5. Value

This interface represents the value that will store, that is, a wrapper to serve as a bridge between the database and the application. For example, if a database does not support a Java type, it may do the conversion with ease.

```
Value value = Value.of(12);

String string = value.get(String.class);

List<Integer> list = value.get(new TypeReference<List<Integer>>() {});

Set<Long> set = value.get(new TypeReference<Set<Long>>() {});

Stream<Integer> stream = value.get(new TypeReference<Stream<Integer>>() {});

Object integer = value.get();
```

2.5.1. Create Custom Writer and Reader

As mentioned before, the `Value` interface is used to store the cost information into a database. The API already has support to the Java type such as primitive types, wrappers types, new Java 8 date/time. Furthermore, the developer can create a custom converter quickly and easily. It has two interfaces:

- `ValueWriter`: This interface represents an instance of `Value` to write in a database.
- `ValueReader`: This interface represents how the `Value` will convert to Java application. This interface will use the `<T> T get(Class<T> type)` and `<T> T get(TypeSupplier<T> typeSupplier)`.

Both class implementations load from the Java SE ServiceLoader resource. So for the Communication API to learn a new type, just register on ServiceLoader. Consider the following `Money` class:

```
import java.math.BigDecimal;
import java.util.Currency;
import java.util.Objects;

public class Money {

    private final Currency currency;

    private final BigDecimal value;

    private Money(Currency currency, BigDecimal value) {
        this.currency = currency;
        this.value = value;
    }

    public Currency getCurrency() {
        return currency;
    }

    public BigDecimal getValue() {
        return value;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }
        Money money = (Money) o;
        return Objects.equals(currency, money.currency) &&
                Objects.equals(value, money.value);
    }
```

```
    @Override
    public int hashCode() {
        return Objects.hash(currency, value);
    }

    @Override
    public String toString() {
        return currency.getCurrencyCode() + ' ' + value;
    }

    public static Money of(Currency currency, BigDecimal value) {
        return new Money(currency, value);
    }

    public static Money parse(String text) {
        String[] texts = text.split(" ");
        return new Money(Currency.getInstance(texts[0]),
                BigDecimal.valueOf(Double.valueOf(texts[1])));
    }
}
```

Just to be more didactic, the book creates a simple money representation. As everyone knows, reinventing the wheel is not a good practice. In a production environment, the Java developer should use mature Money APIs such as Moneta, the reference implementation of JSR 354.

The first step is to create the converter to a custom type database, the `ValueWriter`.

```
import jakarta.nosql.ValueWriter;

public class MoneyValueWriter implements ValueWriter<Money, String> {

    @Override
    public boolean isCompatible(Class type) {
        return Money.class.equals(type);
    }

    @Override
    public String write(Money money) {
        return money.toString();
    }
}
```

With the `MoneyValueWriter` created and the `Money` type will save as String, then the next step is read information to Java application. As can be seen, a `ValueReader` implementation.

```
import jakarta.nosql.ValueReader;
```

```
public class MoneyValueReader implements ValueReader {

    @Override
    public boolean isCompatible(Class type) {
        return Money.class.equals(type);
    }

    @Override
    public <T> T read(Class<T> type, Object value) {
        return (T) Money.parse(value.toString());
    }
}
```

After both implementations have been completed, the last step is to register them into two files:

- `META-INF/services/jakarta.nosql.ValueReader`
- `META-INF/services/jakarta.nosql.ValueWriter`

Each file will have the qualifier of its respective implementation:

The file `jakarta.nosql.ValueReader` will contain:

```
my.company.MoneyValueReader
```

The file `jakarta.nosql.ValueWriter` will contain:

```
my.company.MoneyValueWriter
```

```
Value value = Value.of("BRL 10.0");

Money money = value.get(Money.class);

List<Money> moneys = value.get(new TypeReference<List<Money>>() {});

Set<Money> moneys = value.get(new TypeReference<Set<Money>>() {});;
```

2.6. Element Entity

The **Element Entity** is a small piece of a body, except for the key-value structure type, once this structure is simple. For example, in the column family structure, the entity has columns, the element entity with column has a tuple where the key is the name, and the value is the information as an implementation of `Value`.

- **Document**

- **Column**

2.6.1. Document

The `Document` is a small piece of a Document entity. Each document has a tuple where the key is the document name, and the value is the information itself as `Value`.

```
Document document = Document.of("name", "value");

Value value = document.getValue();

String name = document.getName();
```

The document might have a nested document, that is, a sub-document.

```
Document subDocument = Document.of("subDocument", document);
```

The way to store information in sub-documents will also depend on the implementation of each database driver.

To access the information from `Document`, it has an alias method to `Value`. In other words, it does a conversion directly from `Document` *interface*.

```
Document age = Document.of("age", 29);

String ageString = age.get(String.class);

List<Integer> ages = age.get(new TypeReference<List<Integer>>() {});

Object ageObject = age.get();
```

2.6.2. Column

The Column is a small piece of the Column Family entity. Each column has a tuple where the name represents a key and the value itself as a `Value` implementation.

```
Column document = Column.of("name", "value");

Value value = document.getValue();

String name = document.getName();
```

With this interface, we may have a column inside a column.

```
Column subColumn = Column.of("subColumn", column);
```

The way to store a sub-column will also depend on each driver's implementation as well as the information.

To access the information from `Column`, it has an alias method to `Value`. Thus, you can convert directly from a `Column` *interface*.

```
Column age = Column.of("age", 29);

String ageString = age.get(String.class);

List<Integer> ages = age.get(new TypeReference<List<Integer>>() {});

Object ageObject = age.get();
```

2.7. Entity

The Entity is the body of the information that goes to the database. Each database has an Entity:

- ColumnEntity
- DocumentEntity
- KeyValueEntity

2.7.1. ColumnEntity

The `ColumnEntity` is an entity to the Column Family database type. It is composed of one or more columns. As a result, the `Column` is a tuple of name and value.

```
ColumnEntity entity = ColumnEntity.of("entity");

entity.add("id", 10L);

entity.add("version", 0.001);

entity.add("name", "Diana");

entity.add("options", Arrays.asList(1, 2, 3));

List<Column> columns = entity.getColumns();

Optional<Column> id = entity.find("id");
```

2.7.2. DocumentEntity

The `DocumentEntity` is an entity to Document collection database type. It is composed of one or more documents. As a result, the `Document` is a tuple of name and value.

```
DocumentEntity entity = DocumentEntity.of("documentFamily");

String name = entity.getName();

entity.add("id", 10L);

entity.add("version", 0.001);

entity.add("name", "Diana");

entity.add("options", Arrays.asList(1, 2, 3));

List<Document> documents = entity.getDocuments();
Optional<Document> id = entity.find("id");
entity.remove("options");
```

2.7.3. KeyValueEntity

The `KeyValueEntity` is the simplest structure. It has a tuple and a key-value structure. As the previous entity, it has direct access to information using alias method to `Value`.

```
KeyValueEntity<String> entity = KeyValueEntity.of("key", Value.of(123));

KeyValueEntity<Integer> entity2 = KeyValueEntity.of(12, "Text");

String key = entity.getKey();

Value value = entity.getValue();

Integer integer = entity.get(Integer.class);
```

2.8. Manager

The `Manager` is the class that pushes information to a database and retrieves it.

- **DocumentManager**
- **ColumnConfiguration**
- **BucketManager**

2.8.1. Document Manager

The `DocumentManager` is the class that manages the persistence on the synchronous way to document collection.

```
DocumentEntity entity = DocumentEntity.of("collection");

Document diana = Document.of("name", "Diana");

entity.add(diana);

List<DocumentEntity> entities = Collections.singletonList(entity);

DocumentManager manager = // instance;

// Insert operations
manager.insert(entity);

manager.insert(entity, Duration.ofHours(2L)); // inserts with two hours of TTL

manager.insert(entities, Duration.ofHours(2L)); // inserts with two hours of TTL

// Update operations
manager.update(entity);

manager.update(entities);
```

2.8.1.1. Search information

The Document Communication API supports retrieving information from a `DocumentQuery` instance.

By default, there are two ways to create a `DocumentQuery` instance that are available as a static method in the same class:

1. **The select methods** follow the fluent-API principle; thus, it is a safe way to create a query using a DSL code. Therefore, each action will only show the reliability option as a menu.
2. **The builder methods** follow the builder pattern; it is not more intelligent and safer than the previous one. However, it allows for running more complex queries and combinations.

Both methods should guarantee the validity and consistency`DocumentQuery` instance.

In the next step, there are a couple of query creation samples using both select and builder methods.

- Select all fields from the document collection Person.

Using the select method:

```
DocumentQuery query = DocumentQuery.select().from("Person").build();
```

```
//static imports
DocumentQuery query = select().from("Person").build();
```

Using the builder method:

```
DocumentQuery query = DocumentQuery.builder().from("Person").build();
//static imports
DocumentQuery query = builder().from("Person").build();
```

- Select all fields where the "name" equals "Ada Lovelace" from the document collection Person.

Using the select method:

```
 DocumentQuery query = DocumentQuery.select()
                .from("Person").where("name").eq("Ada Lovelace")
                .build();
//static imports
 DocumentQuery query = select()
                .from("Person").where("name").eq("Ada Lovelace")
                .build();
```

Using the builder method:

```
DocumentQuery query = DocumentQuery.builder()
                .from("Person").where(DocumentCondition.eq("name", "Ada Lovelace"))
                .build();
//static imports
DocumentQuery query = builder().from("Person")
                               .where(eq("name", "Ada Lovelace"))
                               .build();
```

- Select the field name where the "name" equals "Ada Lovelace" from the document collection Person.

Using the select method:

```
DocumentQuery query = DocumentQuery.select("name")
                .from("Person").where("name").eq("Ada Lovelace")
                .build();
//static imports
DocumentQuery query = select("name")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .build();
```

Using the builder method:

```
DocumentQuery query = DocumentQuery.builder("name")
              .from("Person").where(DocumentCondition.eq("name", "Ada Lovelace"))
              .build();
//static imports

DocumentQuery query = builder("name")
              .from("Person").where(eq("name", "Ada Lovelace"))
              .build();
```

- Select the fields name and age where the "name" is "Ada Lovelace" and the "age" is greater than twenty from the document collection Person.

Using the select method:

```
DocumentQuery query = DocumentQuery.select("name", "age")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .and("age").gt(20)
              .build();
//static imports
DocumentQuery query = select("name", "age")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .and("age").gt(20)
              .build();
```

Using the builder method:

```
DocumentQuery query = DocumentQuery.builder("name", "age")
             .from("Person")
             .where(DocumentCondition.and(DocumentCondition.eq("name", "Ada Lovelace"
),
             DocumentCondition.gt("age", 20)))
             .build();

//static imports

DocumentQuery query = builder("name", "age")
             .from("Person")
             .where(and(eq("name", "Ada Lovelace"),
             gt("age", 20)))
             .build();
```

- Select the fields name and age where the "name" is "Ada Lovelace" or the "age" is greater than twenty from the document collection Person.

Using the select method:

```
DocumentQuery query = DocumentQuery.select("name", "age")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .or("age").gt(20)
              .build();
//static imports
DocumentQuery query = select("name", "age")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .or("age").gt(20)
              .build();
```

Using the builder method:

```
DocumentQuery query = DocumentQuery.builder("name", "age")
             .from("Person")
             .where(DocumentCondition.or(DocumentCondition.eq("name", "Ada Lovelace"),
             DocumentCondition.gt("age", 20)))
             .build();

//static imports

DocumentQuery query = builder("name", "age")
             .from("Person")
             .where(or(eq("name", "Ada Lovelace"),
             gt("age", 20)))
             .build();
```

- Select the fields name and age where the "name" is "Ada Lovelace" or the "age" is greater than twenty; skip the first element, and the max return is two from the document collection Person.

Using the select method:

```
DocumentQuery query = DocumentQuery.select("name", "age")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .or("age").gt(20)
              .skip(1)
              .limit(2)
              .build();
//static imports
DocumentQuery query = select("name", "age")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .or("age").gt(20)
              .skip(1)
              .limit(2)
```

```
                .build();
```

Using the builder method:

```
DocumentQuery query = DocumentQuery.builder("name", "age")
                .from("Person")
                .where(DocumentCondition.or(DocumentCondition.eq("name", "Ada Lovelace"
),
                          DocumentCondition.gt("age", 20)))
                .skip(1).limit(2)
                .build();

//static imports

DocumentQuery query = builder("name", "age")
                  .from("Person")
                  .where(or(eq("name", "Ada Lovelace"),
                          gt("age", 20)))
                  .skip(1).limit(2)
                  .build();
```

- Select the fields name and age where the "name" is "Ada Lovelace" or the "age" is greater than twenty; skip the first element, and the max return is two sorts ascending by name and descending by age from the document collection Person.

Using the select method:

```
DocumentQuery query = DocumentQuery.select("name", "age")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .or("age").gt(20)
              .orderBy("name").asc()
              .orderBy("desc").desc()
              .build();

//static imports

DocumentQuery query = select("name", "age")
              .from("Person")
              .where("name").eq("Ada Lovelace")
              .or("age").gt(20)
              .orderBy("name").asc()
              .orderBy("desc").desc()
              .build();
```

Using the builder method:

```
 DocumentQuery query = DocumentQuery.builder("name", "age")
         .from("Person")
         .where(DocumentCondition.or(DocumentCondition.eq("name", "Ada Lovelace"),
                DocumentCondition.gt("age", 20)))
         .sort(Sort.asc("name"), Sort.desc("age"))
         .build();

//static imports

DocumentQuery query = builder("name", "age")
         .from("Person")
         .where(or(eq("name", "Ada Lovelace"),
                gt("age", 20)))
         .sort(asc("name"), desc("age"))
         .build();
```

2.8.1.2. Removing information

Similar to `DocumentQuery`, there is a class to remove information from the document database type: A `DocumentDeleteQuery` type.

It is more efficient than `DocumentQuery` because there is no pagination and sort feature as this information is unnecessary to remove information from database.

It follows the same principle of the query where it has the build and select methods.

```
DocumentManager manager = // instance;
DocumentDeleteQuery query = DocumentQueryBuilder.delete().from("collection")
                                                .where("age").gt(10).build();
manager.delete(query);
//using builder
DocumentDeleteQuery query = DocumentQueryBuilder.builder().from("collection")
                                                .where(DocumentCondition.gt("age", 10
).build();
```

The `DocumentCondition` has support for both `DocumentQuery` and `DocumentDeleteQuery` on fluent and builder patterns.

The main difference is that you'll combine all the options manually on the builder instead of being transparent as the fluent way does.

Thus, it is worth checking the DocumentCondition to see all the filter options.

2.8.2. Column Manager

The `ColumnManager` is the class that manages the persistence on the synchronous way to a Column Family database.

```
ColumnEntity entity = ColumnEntity.of("entity");

Column diana = Column.of("name", "Diana");

entity.add(diana);
```

```
List<ColumnEntity> entities = Collections.singletonList(entity);
ColumnManager manager = // instance;

// Insert operations
manager.insert(entity);

manager.insert(entity, Duration.ofHours(2L)); // inserts with two hours of TTL

manager.insert(entities, Duration.ofHours(2L)); // inserts with two hours of TTL

// Update operations
manager.update(entity);

manager.update(entities);
```

The Column Communication API supports retrieving information from a `ColumnQuery` instance.

By default, there are two ways to create a `ColumnQuery` instance that are available as a static method in the same class:

1. **The select methods** follow the fluent-API principle; thus, it is a safe way to create a query using a DSL code. Therefore, each action will only show the reliability option as a menu.
2. **The builder methods** follow the builder pattern; it is not more intelligent and safer than the previous one. However, it allows for running more complex queries and combinations.

Both methods should guarantee the validity and consistency`ColumnQuery` instance.

In the next step, there are a couple of query creation samples using both select and builder methods.

- Select all fields from the column family Person.

Using the select method:

```
ColumnQuery query = ColumnQuery.select().from("Person").build();
//static imports
ColumnQuery query = select().from("Person").build();
```

Using the builder method:

```
ColumnQuery query = ColumnQuery.builder().from("Person").build();
//static imports
ColumnQuery query = builder().from("Person").build();
```

- Select all fields where the "name" equals "Ada Lovelace" from the column family Person.

Using the select method:

```
ColumnQuery query = ColumnQuery.select()
            .from("Person").where("name").eq("Ada Lovelace")
            .build();
//static imports
ColumnQuery query = select()
            .from("Person").where("name").eq("Ada Lovelace")
            .build();
```

Using the builder method:

```
ColumnQuery query = ColumnQuery.builder()
            .from("Person").where(ColumnCondition.eq("name", "Ada Lovelace"))
            .build();
//static imports
ColumnQuery query = builder().from("Person")
            .where(eq("name", "Ada Lovelace"))
            .build();
```

- Select the field name where the "name" equals "Ada Lovelace" from the column family Person.

Using the select method:

```
ColumnQuery query = ColumnQuery.select("name")
            .from("Person").where("name").eq("Ada Lovelace")
            .build();
//static imports
ColumnQuery query = select("name")
            .from("Person")
            .where("name").eq("Ada Lovelace")
            .build();
```

Using the builder method:

```
ColumnQuery query = ColumnQuery.builder("name")
            .from("Person").where(ColumnCondition.eq("name", "Ada Lovelace"))
            .build();
//static imports
```

```
ColumnQuery query = builder("name")
            .from("Person").where(eq("name", "Ada Lovelace"))
            .build();
```

- Select the fields name and age where the "name" is "Ada Lovelace" and the "age" is greater than twenty from the column family Person.

Using the select method:

```
ColumnQuery query = ColumnQuery.select("name", "age")
            .from("Person")
            .where("name").eq("Ada Lovelace")
            .and("age").gt(20)
            .build();
//static imports
ColumnQuery query = select("name", "age")
            .from("Person")
            .where("name").eq("Ada Lovelace")
            .and("age").gt(20)
            .build();
```

Using the builder method:

```
ColumnQuery query = ColumnQuery.builder("name", "age")
            .from("Person")
            .where(ColumnCondition.and(ColumnCondition.eq("name", "Ada Lovelace"),
             DocumentCondition.gt("age", 20)))
             .build();

//static imports

ColumnQuery query = builder("name", "age")
            .from("Person")
            .where(and(eq("name", "Ada Lovelace"),
             gt("age", 20)))
            .build();
```

- Select the fields name and age where the "name" is "Ada Lovelace" or the "age" is greater than twenty from the column family Person.

Using the select method:

```
ColumnQuery query = ColumnQuery.select("name", "age")
            .from("Person")
            .where("name").eq("Ada Lovelace")
            .or("age").gt(20)
            .build();
//static imports
```

```
ColumnQuery query = select("name", "age")
            .from("Person")
            .where("name").eq("Ada Lovelace")
            .or("age").gt(20)
            .build();
```

Using the builder method:

```
ColumnQuery query = ColumnQuery.builder("name", "age")
            .from("Person")
            .where(ColumnCondition.or(ColumnCondition.eq("name", "Ada Lovelace"),
             ColumnCondition.gt("age", 20)))
            .build();

//static imports

ColumnQuery query = builder("name", "age")
            .from("Person")
            .where(or(eq("name", "Ada Lovelace"),
             gt("age", 20)))
            .build();
```

- Select the fields name and age where the "name" is "Ada Lovelace" or the "age" is greater than twenty; skip the first element, and the max return is two from the column family Person.

Using the select method:

```
ColumnQuery query = ColumnQuery.select("name", "age")
            .from("Person")
            .where("name").eq("Ada Lovelace")
            .or("age").gt(20)
            .skip(1)
            .limit(2)
            .build();
//static imports
ColumnQuery query = select("name", "age")
            .from("Person")
            .where("name").eq("Ada Lovelace")
            .or("age").gt(20)
            .skip(1)
            .limit(2)
            .build();
```

Using the builder method:

```
ColumnQuery query = ColumnQuery.builder("name", "age")
            .from("Person")
```

```
            .where(ColumnCondition.or(ColumnCondition.eq("name", "Ada Lovelace"),
                        ColumnCondition.gt("age", 20)))
            .skip(1).limit(2)
            .build();

//static imports

ColumnQuery query = builder("name", "age")
                .from("Person")
                .where(or(eq("name", "Ada Lovelace"),
                        gt("age", 20)))
                .skip(1).limit(2)
                .build();
```

- Select the fields name and age where the "name" is "Ada Lovelace" or the "age" is greater than twenty; skip the first element, and the max return is two sorts ascending by name and descending by age from the column family Person.

Using the select method:

```
ColumnQuery query = ColumnQuery.select("name", "age")
            .from("Person")
            .where("name").eq("Ada Lovelace")
            .or("age").gt(20)
            .orderBy("name").asc()
            .orderBy("desc").desc()
            .build();
```

Using the builder method:

```
 ColumnQuery query = ColumnQuery.builder("name", "age")
         .from("Person")
         .where(DocumentCondition.or(DocumentCondition.eq("name", "Ada Lovelace"),
                DocumentCondition.gt("age", 20)))
         .sort(Sort.asc("name"), Sort.desc("age"))
         .build();

//static imports

ColumnQuery query = builder("name", "age")
         .from("Person")
         .where(or(eq("name", "Ada Lovelace"),
                gt("age", 20)))
         .sort(asc("name"), desc("age"))
         .build();
```

2.8.2.1. Removing information

Similar to `ColumnQuery`, there is a class to remove information from the document database type: A `ColumnDeleteQuery` type.

It is more efficient than `ColumnQuery` because there is no pagination and sort feature as this information is unnecessary to remove information from database.

It follows the same principle of the query where it has the build and select methods.

```
ColumnManager manager = // instance;
ColumnDeleteQuery query = ColumnDeleteQuery.delete().from("collection")
                                                    .where("age").gt(10).build();
manager.delete(query);
//using builder
ColumnDeleteQuery query = ColumnDeleteQuery.builder().from("collection")
                                                    .where(DocumentCondition.gt("age", 10
).build();
```

The `ColumnCondition` has support for both `ColumnQuery` and `ColumnDeleteQuery` on fluent and builder patterns.

The main difference is that you'll combine all the options manually on the builder instead of being transparent as the fluent way does.

Thus, it is worth checking the ColumnCondition to see all the filter options.

2.8.3. BucketManager

The `BucketManager` is the class which saves the `KeyValueEntity` in a synchronous way in Key-Value database.

```
BucketManager bucketManager = //instance;
KeyValueEntity<String> entity = KeyValueEntity.of("key", 1201);

Set<KeyValueEntity<String>> entities = Collections.singleton(entity);

bucketManager.put("key", "value");

bucketManager.put(entity);

bucketManager.put(entities);

bucketManager.put(entities, Duration.ofHours(2)); // inserts with two hours TTL

bucketManager.put(entity, Duration.ofHours(2)); // inserts with two hours TTL
```

2.8.3.1. Remove and Retrieve information

With a simple structure, the bucket needs a key to both retrieve and delete information from the database.

```
Optional<Value> value = bucketManager.get("key");

Iterable<Value> values = bucketManager.get(Collections.singletonList("key"));

bucketManager.remove("key");

bucketManager.remove(Collections.singletonList("key"));
```

2.8.4. Querying by Text with the Communication API

The Communication API allows queries to be text. These queries are converted to an operation that already exists in the Manager interface from the `query` method. An `UnsupportedOperationException` is thrown if a NoSQL database doesn't have support for that procedure.

Queries follow these rules:

- All instructions end with a like break `\n`
- It is case-sensitive
- All keywords must be in lowercase
- The goal is to look like SQL, however simpler
- Even if a query has valid sintax a specific implementation may not support an operation. For example, a Column family database may not support queries in a different field that is not the ID field.

2.8.4.1. Key-Value Database Types

Key-Value databases support three operations: `get`, `del` and `put`.

`get`

Use the `get` statement to retrie data related to a key

```
get_statement ::=  get ID (',' ID)*

//examples
get "Apollo" //to return an element where the id is 'Apollo'
get "Diana" "Artemis" //to return a list of values from the keys
```

`del`

Use the `del` statement to delete one or more entities

```
del_statement ::=  del ID (',' ID)*

//examples
del "Apollo"
del "Diana" "Artemis"
```

put

Use the `put` statement to either insert or override values

```
put_statement ::=  put {KEY, VALUE [, TTL]}

//examples
put {"Diana" , "The goddess of hunt"} //adds key 'diana' and value 'The goddess of
hunt'
put {"Diana" , "The goddess of hunt", 10 second} //also defines a TTL of 10 seconds
```

2.8.4.2. Column-Family and Document Database Types

The queries have syntax similar to SQL queries. But keep in mind that it has a limitation: joins are not supported.

They have four operations: `insert`, `update`, `delete`, and `select`.

insert

Use the `insert` statement to store data for an entity

```
insert_statement ::=  insert ENTITY_NAME (NAME = VALUE, (`,` NAME = VALUE) *) || JSON
[ TTL ]

//examples
insert Deity (name = "Diana", age = 10)
insert Deity (name = "Diana", age = 10, powers = {"sun", "moon"})
insert Deity (name = "Diana", age = 10, powers = {"sun", "moon"}) 1 day
insert Deity {"name": "Diana", "age": 10, "powers": ["hunt", "moon"]}
insert Deity {"name": "Diana", "age": 10, "powers": ["hunt", "moon"]} 1 day
```

update

Use the `update` statement to update the values of an entity

```
update_statement ::= update ENTITY_NAME (NAME = VALUE, (`,` NAME = VALUE) *) || JSON

 //examples
update Deity (name = "Diana", age = 10)
update Deity (name = "Diana", age = 10, power = {"hunt", "moon"})
```

```
update Deity {"name": "Diana", "age": 10, "power": ["hunt", "moon"]}
```

`delete`

Use the `delete` statement to remove fields or entities

```
delete_statement ::=  delete [ simple_selection ( ',' simple_selection ) ]
                      from ENTITY_NAME
                      [ where WHERE_CLAUSE ]
//examples
delete from Deity
delete power, age from Deity where name = "Diana"
```

`select`

The `select` statement reads one or more fields for one or more entities. It returns a result-set of the entities matching the request, where each entity contains the fields corresponding to the query.

```
select_statement ::=  select ( SELECT_CLAUSE | '*' )
                      from ENTITY_NAME
                      [ where WHERE_CLAUSE ]
                      [ skip (INTEGER) ]
                      [ limit (INTEGER) ]
                      [ order by ORDERING_CLAUSE ]
//examples
select * from Deity
select name, age, adress.age from Deity order by name desc age desc
select * from Deity where birthday between "01-09-1988" and "01-09-1988" and salary =
12
select name, age, adress.age from Deity skip 20 limit 10 order by name desc age desc
```

2.8.4.3. `where`

The `where` keyword specifies a filter (`WHERE_CLAUSE`) to the query. A filter is composed of boolean statements called `conditions` that are combined using `and` or `or` operators.

```
WHERE_CLAUSE ::= CONDITION ([and | or] CONDITION)*
```

2.8.4.4. Conditions

Conditions are boolean statements that operate on data being queried. They are composed of three elements:

1. **Name**: the data source, or target, to apply the operator
2. **Operator**, defines comparing process between the name and the value.
3. **Value**, that data that receives the operation.

2.8.4.5. Operators

The Operators are:

Table 1. Operators in a query

Operator	Description
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
⇐	Less than or equal to
BETWEEN	TRUE if the operand is within the range of comparisons
NOT	Displays a record if the condition(s) is NOT TRUE
AND	TRUE if all the conditions separated by AND is TRUE
OR	TRUE if any of the conditions separated by OR is TRUE
LIKE	TRUE if the operand matches a pattern
IN	TRUE if the operand is equal to one of a list of expressions

2.8.4.6. The Value

The value is the last element in a condition, and it defines what'll go to be used, with an operator, in a field target.

There are six types:

- Number is a mathematical object used to count, measure and also label, where if it is a decimal, will become **double**, otherwise, **long**. E.g.: `age = 20`, `salary = 12.12`
- String: one or more characters among either two double quotes, `"`, or single quotes, `'`. E.g.: `name = "Ada Lovelace"`, `name = 'Ada Lovelace'`
- Convert: convert is a function where given the first value parameter as number or string, it will convert to the class type of the second one. E.g.: `birthday = convert("03-01-1988", java.time.LocalDate)`
- Parameter: the parameter is a dynamic value, which means it does not define the query; it'll replace in the execution time. The parameter is at `@` followed by a name. E.g.: `age = @age`
- Array: A sequence of elements that can be either number or string that is between braces `{ }`. E.g.: `power = {"Sun", "hunt"}`
- JSON: JavaScript Object Notation is a lightweight data-interchange format. E.g.: `siblings = {"apollo": "brother", "zeus": "father"}`

2.8.4.7. `skip`

The `skip` option in a `select` statement defines where the query results should start.

2.8.4.8. `limit`

The `limit` option in a `select` statement limits the number of rows returned by a query.

2.8.4.9. `order by`

The `order by` option allows defining the order of the returned results. It takes as argument (ORDERING_CLAUSE) a list of column names along with the ordering for the column (`asc` for ascendant, which is the default, and `desc` for the descendant).

```
ORDERING_CLAUSE ::= NAME [asc | desc] ( NAME [asc | desc])*
```

2.8.4.10. TTL

Both the **INSERT** and **PUT** commands support setting a time for data in an entity to expire. It defines the time to live of an object that is composed of the integer value and then the unit that might be `day`, `hour`, `minute`, `second`, `millisecond`, `nanosecond`. E.g.: `ttl 10 second`

2.8.4.11. `PreparedStatement` and `PreparedStatementAsync`

To dynamically run a query, use the `prepare` method in the manager for instance. It will return a `PreparedStatement` interface. To define a parameter to key-value, document, and column query, use the "@" in front of the name.

```
PreparedStatement preparedStatement = documentManager
        .prepare("select * from Person where name = @name");

preparedStatement.bind("name", "Ada");

Stream<DocumentEntity> adas = preparedStatement.getResult();
```

```
PreparedStatementAsync preparedStatement = documentManagerAsync
        .prepare("select * from Person where name = @name");

preparedStatement.bind("name", "Ada");

Consumer<Stream<DocumentEntity>> callback = //instance;

preparedStatement.getResult(callback);
```

For more information on Apache TinkerPop and the Gremlin API, please visit this website.

2.9. Factory

The factory class creates the **Managers**.

- **BucketManagerFactory**: The factory classes have the responsibility to create the `BucketManager`.
- **ColumnManagerFactory**: The factory classes have the duty to create the Column manager.
- **DocumentManagerFactory**: The factory classes have the duty to create the document collection manager.

2.10. Configuration

The configuration classes create a Manager Factory. This class has all the configuration to build the database connection.

There are a large number of diversity configuration flavors such as P2P, master/slave, thrift communication, HTTP, etc. The implementation may be different, however, but they have a method to return a Manager Factory. It is recommended that all database driver providers have a properties file to read this startup information.

2.10.1. Settings

The `Settings` interface represents the settings used in a configuration. It extends looks like a `Map<String, Object>`; for this reason, gives a key that can set any value as configuration.

```
Settings settings = Settings.builder()
    .put("key", "value")
    .build();
Map<String, Object> map = //instance;

Settings settings = Settings.of(map);
```

2.10.2. Document Configuration

For the Document collection configuration, `DocumentConfiguration` configures and creates `DocumentManagerFactory`.

```
Settings settings = Settings.builder()
    .put("key", "value")
    .build();
DocumentConfiguration configuration = //instance;
DocumentManagerFactory managerFactory = configuration.get(settings);
```

2.10.3. Column Configuration

For the Column Family configuration, `ColumnConfiguration` creates and configures `ColumnManagerFactory`.

```
Settings settings = Settings.builder()
    .put("key", "value")
    .build();
ColumnConfiguration configuration = //instance;
ColumnManagerFactory managerFactory = configuration.get(settings);
```

2.10.4. Key Value Configuration

For the key-value configuration, there is `KeyValueConfiguration` to `BucketManagerFactory`.

```
Settings settings = Settings.builder()
    .put("key", "value")
    .build();
KeyValueConfiguration configuration = //instance;
BucketManagerFactory managerFactory = configuration.get();
```

Chapter 3. 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 in charge to perform integration among technologies such as Bean Validation. The Mapping API has 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 database.

To go straight and cover the four NoSQL types, this API has four domains:

- `jakarta.nosql.mapping.column`
- `jakarta.nosql.mapping.document`
- `jakarta.nosql.mapping.graph`
- `jakarta.nosql.mapping.keyvalue`

The package name might change on the Jakarta EE process.

3.1. The Mapping structure

The mapping API has five parts:

- The **persistence-core**: The mapping common project where there are annotations commons among the NoSQL types APIs.
- The **persistence-key-value**: The mapping to key-value NoSQL database.
- The **persistence-column**: The mapping to column NoSQL database.
- The **persistence-document**: The mapping to document NoSQL database.
- The **persistence-graph**: The mapping to Graph NoSQL database.

Each module works separately as a Communication API.

Similar to the communication API, there is a support for database diversity. This project has extensions for each database types on the database mapping level.

3.2. 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

3.2.1. Annotation Models

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

- Entity
- Column
- Id
- Embeddable
- Convert
- MappedSuperclass
- Inheritance
- DiscriminatorColumn
- DiscriminatorValue
- Database

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

3.2.1.1. @Entity

This annotation maps the class to Jakarta NoSQL. There is a single value attribute. This attribute specifies the column family name, or 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"
   }
}
```

3.2.1.2. @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. It is 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 `@Key` - 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;
}
```

3.2.1.3. @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;
}
```

3.2.1.4. @Embeddable

This annotation defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of that object. The behaviour is similar to `@MappedSuperclass`, but this is used on composition instead of inheritance.

```
@Entity
public class Book {

    @Column
    private String title;

    @Column
    private Author author;
}

@Embeddable
public class Author {

    @Column
    private String author;

    @Column
    private Integer age;
}
```

In this example, there is a single instance in the database with columns `title`, `author` and `age`.

```
{
```

```
    "title":"Effective Java",
    "author":"Joshua Bloch",
    "age": 2019
}
```

3.2.1.5. @Convert

This annotation allows value conversions when mapping the value that came from the Communication API. This is useful for cases such as to cipher a field (String to String conversion), or to convert to a custom type. The Converter annotation has a single, mandatory parameter: a class that inherits from `AttributeConverter` that will be used to perform the conversion. The example below shows how to create a converter to a custom `Money` class.

```
@Entity
public class Employee {

    @Column
    private String name;

    @Column
    private Job job;

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

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);
    }
}

public class MonetaryAmount {
    private final String currency;

    private final BigDecimal value;

    public String toString() {
        // specific implementation
    }
```

```
    public static MonetaryAmount parse(String string) {
        // specific implementation
    }
}
```

3.2.1.6. Collections

The Mapping layer supports `java.util.Collection` (and subclasses as defined below) mapping to simple elements such as `String` and `Integer` (that will be sent to the communication API as-is), and mapping to `Entity` or `Embedded` entities.

The following collections are supported:

- `java.util.Deque`
- `java.util.Queue`
- `java.util.List`
- `java.util.Iterable`
- `java.util.NavigableSet`
- `java.util.SortedSet`
- `java.util.Collection`

```
@Entity
public class Person {

    @Id
    private Long id;

    @Column
    private String name;

    @Column
    private List<String> phones;

    @Column
    private List<Address> addresses;
}

@Embeddable
public class Address {

    @Column
    private String street;

    @Column
    private String city;
}
```

The above classes are mapped to:

```
{
   "_id":10,
   "addresses":[
      {
         "city":"São Paulo",
         "street":"Av Nove de Julho"
      },
      {
         "city":"Salvador",
         "street":"Rua Engenheiro Jose Anasoh"
      }
   ],
   "name":"Name",
   "phones":[
      "234",
      "432"
   ]
}
```

3.2.1.7. @MappedSuperclass

The class with the `@MapperSuperclass` annotation will have all attributes considered as an extension of this subclass with an `@Entity` annotation. In this case, all attributes are going to be stored, even the attributes inside the super class.

Using the MappedSuperclass strategy, inheritance is only evident in the class but not the entity model.

This means, that this annotation causes fields annotated with `@Column` in a parent class to be persisted together with the child class' fields.

```
@Entity
public class Dog extends Animal {

    @Column
    private String name;
}

@MappedSuperclass
public class Animal {

    @Column
    private String race;

    @Column
    private Integer age;
```

```
}
```

Notice that the `Animal` doest not have an @Entity annotation, as it won't be persisted in the database by itself.

On the example above, when saving a Dog instance, Animal class' fields are saved too: `name`, `race`, and `age` are saved in a single instance.

3.2.1.8. @Inheritance

The strategy to work with inheritance with NoSQL, you can active it by adding the @Inheritance annotation to the superclass.

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

    @Column
    private String name;

    @Column
    private LocalDate createdOn;

    public abstract void send();
}
```

3.2.1.9. @DiscriminatorColumn

This annotation specifies the discriminator column for the inheritance mapping strategy. The strategy and the discriminator column are only specified in the root of an entity class hierarchy.

If the DiscriminatorColumn annotation is missing, and a discriminator column is required, the name of the discriminator column defaults is "type".

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

    @Column
    private String name;

    @Column
    private LocalDate createdOn;
```

```
    public abstract void send();
}
```

3.2.1.10. @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, the discriminator value default is the `Class.getSimpleName()`.

```
@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 phoneNumber;

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

@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);
    }
```

```
}
```

3.2.1.11. @Database

This annotation allows programmers to specialize `@Inject` annotations to choose which specific resource should be injected.

For example, when working with multiple `DocumentTemplate`, the following statement are ambiguous:

```
@Inject
DocumentTemplate templateA;

@Inject
DocumentTemplate templateB;
```

`@Database` has two attributes to help specify what resource should be injected:

- **DatabaseType**: The database type (key-value, document, column, graph);
- **provider**: The provider's database name

Applying the annotation to the example above, the result is:

```
@Inject
@Database(value = DatabaseType.DOCUMENT, provider = "databaseA")
private DocumentTemplate templateA;

@Inject
@Database(value = DatabaseType.DOCUMENT, provider = "databaseB")
private DocumentTemplate templateB;
```

A producer method annotated with the same `@Database` values must exist as well.

3.3. Template classes

The template offers convenient operations to create, update, delete, query, and provides a mapping among your domain objects and communication API. The templates classes have a goal to persist an Entity Model through a communication API. It has three components:

- **Converter**: That converts the Entity to a communication level API.
- **EntityManager**: The EntityManager for communication.
- **Workflow**: That defines the workflow when either you save or update an entity. These events are useful when you, e.g., want to validate data before being saved. See the following picture:

3.3.1. 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;
```

You can work with several key-value database instances through CDI qualifier. To identify each database instance, make a `BucketManager` visible for CDI by putting the `@Produces` and the `@Database` annotations in the method.

```
@Inject
@Database(value = DatabaseType.KEY_VALUE, provider = "databaseA")
private KeyValueTemplate templateA;

@Inject
@Database(value = DatabaseType.KEY_VALUE, provider = "databaseB")
private KeyValueTemplate templateB;

// producers methods
@Produces
```

```
@Database(value = DatabaseType.KEY_VALUE, provider = "databaseA")
public BucketManager getManagerA() {
    BucketManager manager = // instance;
    return manager;
}

@Produces
@Database(value = DatabaseType.KEY_VALUE, provider = "databaseB")
public BucketManager getManagerB() {
    BucketManager manager = // instance;
    return manager;
}
```

3.3.2. 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);
```

For information removal and retrieval, there are `ColumnQuery` and `ColumnDeleteQuery` classes, respectively. Also, the callback method may be used.

```
ColumnQuery query = select().from("Person").where("address").eq("Olympus").build();

Stream<Person> peopleWhoLiveOnOlympus = template.select(query);
Optional<Person> artemis = template.singleResult(select().from("Person").where(
"nickname").eq("artemis").build());

ColumnDeleteQuery deleteQuery = delete().from("Person").where("address").eq("Olympus"
```

```
).build();
template.delete(deleteQuery);
```

Both `ColumnQuery` and `ColumnDeleteQuery` won't convert the object to native fields. However, `ColumnQueryMapperBuilder` creates both query types, reads the class, then switches to the native fields through annotations.

```
@Entity
public class Person {

    @Id("native_id")
    private long id;

    @Column
    private String name;

    @Column
    private int age;
}
```

```
@Inject
ColumnQueryMapperBuilder mapperBuilder;
...
ColumnQuery query = mapperBuilder.selectFrom(Person.class).where("id").gte(10).build(
);
  // translating: select().from("Person").where("native_id").gte(10L).build();
 ColumnDeleteQuery deleteQuery = mapperBuilder.deleteFrom(Person.class)
                                              .where("id").eq("20").build();
 // 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;
```

The next step is to produce a `ColumnManager`:

```
@Produces
public ColumnManager getManager() {
    ColumnManager manager = // instance;
    return manager;
}
```

You can work with several column database instances through CDI qualifier. To identify each

database instance, make a `ColumnManager` visible for CDI by putting the `@Produces` and the `@Database` annotations in the method.

```
@Inject
@Database(value = DatabaseType.COLUMN, provider = "databaseA")
private ColumnTemplate templateA;

@Inject
@Database(value = DatabaseType.COLUMN, provider = "databaseB")
private ColumnTemplate templateB;

// producers methods
@Produces
@Database(value = DatabaseType.COLUMN, provider = "databaseA")
public ColumnManager getManagerA() {
    return manager;
}

@Produces
@Database(value = DatabaseType.COLUMN, provider = "databaseB")
public ColumnManager getManagerB() {
    return manager;
}
```

3.3.3. 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 `DocumentQuery` and `DocumentDeleteQuery` classes.

```
DocumentQuery query = select().from("Person").where("address").eq("Olympus").build();

Stream<Person> peopleWhoLiveOnOlympus = template.find(query);
Optional<Person> artemis = template.singleResult(select().from("Person").where(
"nickname").eq("artemis").build());

DocumentDeleteQuery deleteQuery = delete().from("Person").where("address").eq("
Olympus").build();
template.delete(deleteQuery);
```

Both `DocumentQuery` and `DocumentDeleteQuery` query won't convert the Object to native fields. However, `DocumentQueryMapperBuilder` creates both query types, reads the class, then switches to the native fields through annotations.

```
@Entity
public class Person {

    @Id("native_id")
    private long id;

    @Column
    private String name;

    @Column
    private int age;
}
```

```
@Inject
private DocumentQueryMapperBuilder mapperBuilder;

public void mapper() {
  DocumentQuery query = mapperBuilder.selectFrom(Person.class).where("id")
                                     .gte(10).build();
  // translating: select().from("Person").where("native_id").gte(10L).build();
  DocumentDeleteQuery deleteQuery = mapperBuilder.deleteFrom(Person.class)
                                                 .where("id").eq("20").build();
  // 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;
```

You can work with several document database instances through CDI qualifier. To identify each database instance, make a `DocumentManager` visible for CDI by putting the `@Produces` and the `@Database` annotations in the method.

```
@Inject
@Database(value = DatabaseType.DOCUMENT, provider = "databaseA")
private DocumentTemplate templateA;

@Inject
@Database(value = DatabaseType.DOCUMENT, provider = "databaseB")
private DocumentTemplate templateB;

// producers methods
@Produces
@Database(value = DatabaseType.DOCUMENT, provider = "databaseA")
public DocumentManager getManagerA() {
    return manager;
}

@Produces
@Database(value = DatabaseType.DOCUMENT, provider = "databaseB")
public DocumentManager getManagerB() {
    return manager;
}
```

3.3.4. Graph template

This template has the responsibility to serve as the persistence of an entity in a Graph database using Apache Tinkerpop.

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

```
@Inject
GraphTemplate 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);
```

3.3.4.1. Create the Relationship Between Them (EdgeEntity)

```
Person poliana = // instance;
Book shack = // instance;
EdgeEntity edge = graphTemplate.edge(poliana, "reads", shack);
reads.add("where", "Brazil");
Person out = edge.getOutgoing();
Book in = edge.getIncoming();
```

3.3.4.2. Querying with Traversal

Traversals in Gremlin are spawned from a `TraversalSource`. The `GraphTraversalSource` is the typical "graph-oriented" DSL used throughout the documentation and will most likely be the most used DSL in a TinkerPop application.

To run a query in Graph with Gremlin, there are traversal interfaces. These interfaces are lazy; in other words, they just run after any finalizing method.

For example, In this scenario, there is a marketing campaign, and the target is:

- An engineer
- The salary is higher than $3,000
- The age is between 20 and 25 years old

```
List<Person> developers = graph.getTraversalVertex()
       .has("salary", gte(3_000D))
       .has("age", between(20, 25))
       .has("occupation", "Developer")
       .<Person>stream().collect(toList());
```

The next step is to return the engineer's friends.

```
List<Person> developers = graph.getTraversalVertex()
        .has("salary", gte(3_000D))
        .has("age", between(20, 25))
        .has("occupation", "Developer")
        .<Person>stream().out("knows").collect(toList());
```

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

```
@Inject
private GraphTemplate template;
```

You can work with several graph database instances through CDI qualifier. To identify each database instance, make a `Graph` visible for CDI by putting the `@Produces` and the `@Database` annotations in the method.

```
@Inject
@Database(value = DatabaseType.GRAPH, provider = "databaseA")
private GraphTemplate templateA;

@Inject
@Database(value = DatabaseType.GRAPH, provider = "databaseB")
private GraphTemplate templateB;

// producers methods
@Produces
@Database(value = DatabaseType.GRAPH, provider = "databaseA")
public Graph getManagerA() {
    return graph;
}

@Produces
@Database(value = DatabaseType.GRAPH, provider = "databaseB")
public Graph getManagerB() {
    return graph;
}
```

3.3.5. Querying by Text with the Mapping API

Similar to the Communication layer, the Mapping layer has query by text. Both Communication and Mapping have the `query` and `prepare` methods, however, the Mapping API will convert the fields and entities to native names from the Entity and Column annotations.

3.3.5.1. Key-Value Database Types

In the Key-Value database, a `KeyValueTemplate` is used in this NoSQL storage technology. Usually, all the operations are defined by the ID. Therefore, it has a smooth query.

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

3.3.5.2. Column-Family Database Types

The Column-Family database has a more complex structure; however, a search from the key is still recommended. For example, both Cassandra and HBase have a secondary index, yet, neither have a

guarantee about performance, and they usually recommend having a second table whose row key is the "secondary index" and is only being used to find the row key needed for the actual table. Given a `Person` class as an entity, we would like to operate from the field ID, which is the entity from the Entity.

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

The main difference to run using a template instead of in a manager instance as the template will be a mapper as **ColumnQueryMapperBuilder** does.

3.3.5.3. Document Database Types

The Document database allows for more complex queries, so with more complex entities within a Document database, a developer can more easily and naturally find from different fields. Also, there are Document databases that support an aggregations query. However, Jakarta NoSQL does not yet support this. From the Jakarta NoSQL API perspective, the Document and Column-Family types are pretty similar, but with the Document database type, a Java developer might initiate a query from a field that isn't a key, and neither returns an unsupported operation exception or adds a secondary index for this. So, given the same `Person` class as an entity with the Document database type, a developer can do more with queries, such as "person" between "age."

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

The main difference to run using a template instead of in a manager instance as the template will be a mapper as **DocumentQueryMapperBuilder** does.

3.3.5.4. Graph Database Types

If an application needs a recommendation engine or a full detail about the relationship between two entities in your system, it requires a Graph database type. A graph database contains a vertex and an edge. The edge is an object that holds the relationship information about the edges and has direction and properties that make it perfect for maps or human relationship. For the Graph API, Jakarta NoSQL uses the Apache Tinkerpop. Likewise, the `GraphTemplate` is a wrapper to convert a Java entity to a `Vertex` in TinkerPop.

```
GraphTemplate template = // instance;
Stream<City> cities = template.query("g.V().hasLabel('City')");
```

```
PreparedStatement preparedStatement = documentTemplate
        .prepare("select * from Person where name = @name");

preparedStatement.bind("name", "Ada");
```

```
Stream<Person> adas = preparedStatement.getResult();

// Keep using gremlin for Graph databases
PreparedStatement prepare = graphTemplate().prepare("g.V().hasLabel(param)");

prepare.bind("param", "Person");

Stream<Person> people = preparedStatement.getResult();
```

Chapter 4. References

4.1. 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

4.2. 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/

4.3. 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