



JNOSQL

Eclipse JNoSQL

Version 0.0.6-SNAPSHOT, June 14, 2018

Table of Contents

Eclipse JNoSQL	2
1. Let's talk about standard to NoSQL database in Java	3
1.1. Key-value	3
1.2. Document collection	3
1.3. Column Family	4
1.4. Graph	4
1.5. Multi-model database	5
1.6. Standard in SQL	5
2. The main idea behind the API	7
2.1. Eclipse JNoSQL	8
2.1.1. Communication API	8
2.1.2. Mapping API	9
3. Communication API Introduction	10
3.1. The API structure	10
3.2. Value	11
3.2.1. Make custom Writer and Reader	11
3.3. Element Entity	14
3.3.1. Document	14
3.3.2. Column	14
3.4. Entity	15
3.4.1. ColumnFamilyEntity	15
3.4.2. DocumentEntity	16
3.4.3. KeyValueEntity	16
3.5. Manager	16
3.5.1. Document Manager	16
DocumentCollectionManager	17
DocumentCollectionManagerAsync	17
Search information on a document collection	18
Removing information from Document Collection	18
3.5.2. Column Manager	19
ColumnFamilyManager	19
ColumnFamilyManagerAsync	19
Search information on a column family	20
Removing information from Column Family	21
3.5.3. BucketManager	21
Removing and retrieve information from a key-value database	22
3.5.4. Querying by text at Communication API	22
Key-Value	22

Column and Document	23
PreparedStatement and PreparedStatementAsync	24
WHERE	24
Conditions	24
Operators	25
The value	25
SKIP	26
LIMIT	26
ORDER BY	26
TTL	26
3.6. Factory	26
3.6.1. Column Family Manager Factory	26
3.6.2. Document Collection Factory	27
3.6.3. Bucket Manager Factory	27
3.7. Configuration	27
3.7.1. Settings	28
3.7.2. Document Configuration	28
3.7.3. Column Configuration	28
3.7.4. Key Value Configuration	29
3.8. The diversity on NoSQL database	29
4. Mapping API Introduction	31
4.1. The Mapping structure	31
4.2. Models Annotation	31
4.2.1. Annotation Models	32
Entity	32
Column	33
MappedSuperclass	34
Id	34
Embeddable	35
Convert	35
Collection	36
4.2.2. Qualifier annotation	38
4.2.3. ConfigurationUnit	38
JSON file structure	38
XML file structure	39
YAML file structure	39
Injection the code	40
4.3. Template classes	40
4.3.1. DocumentTemplate	41
4.3.2. DocumentTemplateAsync	43
4.3.3. ColumnTemplate	45

ColumnTemplateAsync	47
4.3.4. Key-Value template	49
4.3.5. Graph template	51
Create the Relationship Between Them (EdgeEntity)	52
Querying with traversal	52
4.3.6. Querying by text at Mapping API	54
Key-Value	54
Column-Family	54
Document Collection	54
Graph	55
4.4. Repository	55
4.4.1. Query by method	58
4.4.2. Using Repository as asynchronous way	59
4.4.3. Using Query annotation	60
4.4.4. How to Create Repository and RepositoryAsync implementation programmatically. .	60
4.5. Bean Validation	62
5. References	64

Specification: Eclipse JNoSQL

Version: 0.0.6-SNAPSHOT

Status: Draft

Release: June 14, 2018

Copyright (c) 2016-2018 Eclipse JNoSQL Contributors:
Otávio Santana

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.

Eclipse JNoSQL

Eclipse JNoSQL is a Java framework that streamlines the integration of Java applications with NoSQL databases. It defines a set of APIs to interact with NoSQL databases and provides a standard implementation for most NoSQL databases. This clearly helps to achieve very low coupling with the underlying NoSQL technologies used.

The project has two layers:

- **Communication Layer:** This is a set of APIs that defines communication with NoSQL databases. In traditional SQL/RDBMS world, these can be compared with the JDBC APIs. This API set contains four modules, with each one representing a NoSQL database storage type: Key-Value, Column Family, Document and Graph.
- **Mapping Layer:** These are the APIs that help developers to map Java objects to NoSQL databases. This layer is annotation driven and uses technologies like CDI and Bean Validations to make it simple for the developer. In traditional SQL/RDBMS world, this layer can be compared to JPA and ORM frameworks.

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
- Support for Asynchronous Queries
- Support for Asynchronous Write operations
- Easy-to-implement API Specification and Test Compatibility Kit (TCK) for NoSQL Vendors
- The API's focus is on simplicity and ease of use. Developers should only have to know a minimal set of artifacts to work with JNoSQL. The API is built on Java 8 features like Lambdas and Streams and therefore fits perfectly with the functional features of Java 8+.

Chapter 1. Let's talk about standard to NoSQL database in Java

The NoSQL DB is a database that provides a mechanism for storage and retrieval of data which is modeled by means other than the tabular relations used in relational databases. These databases have speed and high scalability. This kind of database has becoming more popular in several applications, that include financial one. As result of the increase, the number of a user the number of vendors is increasing too.

The NoSQL database is defined basically by its model of storage, and there are four types:

1.1. Key-value

This database has a structure that looks like a `java.util.Map` API, where we can storage any value from a key.

Examples:

- AmazonDynamo
- AmazonS3
- Redis
- Scalaris
- Voldemort

Table 1. Key-Value vs Relational structure

Relational structure	Key-value structure
Table	Bucket
Row	Key/value pair
Column	----
Relationship	----

1.2. Document collection

This model can store any document, without the need to predefine a structure. This document may be composed of numerous fields, with many kinds of data, including a document inside another document. This model works either with XML or JSON file.

Examples:

- AmazonSimpleDb
- ApacheCouchdb
- MongoDB
- Riak

Table 2. Document vs Relational structure

Relational structure	Document Collection structure
Table	Collection
Row	Document
Column	Key/value pair
Relationship	Link

1.3. Column Family

This model became popular with the BigTable's paper by Google, with the goal of being a distributed system storage, projected to have either a high scalability and volume.

Examples:

- Hbase
- Cassandra
- Scylla
- Clouddata
- SimpleDb
- DynamoDB

Table 3. Column Family vs Relational structure

Relational structure	Column Family structure
Table	Column Family
Row	Column
Column	Key/value pair
Relationship	not supported

1.4. Graph

In computing, a graph database is a database that uses graph structures for semantic queries with nodes, edges, and properties to represent and store data.

- **Property:** A small component in the structure; a tuple where the key is the name and the value is the information itself.
- **Vertex:** Looks like the table in a SQL technology that has an unlimited number of properties.
- **Edge:** The element that represents the relationship between vertices; it has a small similarity with a relationship in SQL. However, the edge has properties — so a connection deeper than relational technology.

The graph direction is another concept pretty important in a graph structure, i.e. you can know a person despite this person not knowing you.

- **Out:** Vertex > action > vertex; the "out" is the "action" direction, moreover the sample (i.e. I know the Ironman)
- **In:** Vertex > passive; the "in" is the reaction course; synonymously, the Ironman is known by me.
- **both:** Refers to either direction **IN** or **OUT**

Examples:

- Neo4j
- InfoGrid
- Sones
- HyperGraphDB

Table 4. Graph vs Relational structure

Relational Structure	Graph structure
Table	Vertex and Edge
Row	Vertex
Column	Vertex and Edge property
Relationship	Edge

1.5. Multi-model database

Some database has support for more than one kind of model storage this is the multi-model database.

Examples:

- OrientDB
- Couchbase

1.6. Standard in SQL

Looking to Java application that uses a relational database. It's a good practice have a layer to be a bridge between a Java application and relationship database: a DAO, the data access object. Talking more about relational databases, there are APIs such as JPA and JDBC that have some advantages to a Java developer:

- There isn't a lock-in vendor, in other words, with the standard a database change will happen easier and with transparency because we just need to change a simple driver.
- It isn't necessary to learn a new API for each new database, once there is a common database communication.
- There isn't an impact in a change from one vendor to another, in some moments is necessary to use a specific database resource, but in this case the not everything in DAO layer is lost.

Currently in NoSQL database hasn't standard so a Java developer has some issues:

- Lock-in vendor
- To each new database is necessary to learn a new API, any change to another database has a high impact and once all the communication layer will be lost there isn't a standard API. This happens even with the same kind of NoSQL database, for example, a change in a column to another column.

There is a massive effort to create a common API to make the Java developers life easier, such as Spring Data, Hibernate ORM, and TopLink. The JPA is popular API in the Java world, this is why all solutions try to use it, however, this API is created to SQL and not to NoSQL, and it doesn't support all behavior in NoSQL database, many NoSQL hasn't a transaction, and many NoSQL database hasn't support to asynchronous insertion.

The solution for this case creates a specification that covers the four kinds of NoSQL database. The new API should look like the JPA, once the developer has familiarity with this API, besides adding new behavior and new exceptions, when a database has not support to a specific resource. Besides the API, another important point is integration with others Java specifications such as CDI and Bean Validation.

Chapter 2. The main idea behind the API

Once, we talked about the importance of the standard of a NoSQL database API; the next step is to discuss, in more details, about API. However, to make a natural explanation, first going to talk about both layer and tier. These structures level make the communication, maintenance, split the responsibility clearer. The new API proposal is going to be responsible for being a bridge between the logic tier and data tier, to do this, we need to create two APIs one to communication to a database and another one to be a high abstraction to Java application.

In software, the world is common that application has structures: tier, physical structure, and layer, logic one. The multi-tier application has three levels:

- **Presentation tier:** That has as primary duty translate the result, from below tiers, to the user a can understand.
- **Logic tier:** The tier where has all business rules, process, conditions, save the information, etc. This level moves and processes information between other levels.
- **Data tier:** Storage and retrieve information either database or a system file.

Talking more precisely about the physical layer, logic to separate responsibilities, there are layers

The logic tier, where the application and the business rule stay, it has layers:

- **Application layer:** The bridge between the view tier and logic tier, e.g., Convert an object into either JSON or HTML.
- **Service layer:** The service layer, this can be either Controller or a Resource.
- **Business Layer:** where the whole business and the model be.
- **Persistence Layer:** The platform between the logic tier and data tier. The layer has an integration such as DAO or repository.

Within a persistence layer, it has its layers: A Data Access Object, DAO, this structure connect business layer and persistence layer. Inside it has an API that does database. Currently, there is a difference between SQL and NoSQL database:

In the relational database there are two mechanisms, beyond DAO, JDBC, and JPA:

- **JDBC:** a deep layer with a database that has communications, basic transactions, basically it's a driver to a particular database.
- **JPA:** A high layer that has communication either JDBC and JPA. This layer has high mapping to Java; this place has annotations and an EntityManager. In general, a JPA has integration with other specifications such as CDI and Bean Validation.

A huge advantage of this strategy that one change, either JDBC or JPA, can happen quickly. When a developer changes a database, he just needs the switch to a respective driver by a database and done! Code ready to a new database changed.

In a NoSQL database, there isn't a strategy to save code or little impact for a change. All APIs are different and don't follow any one standard, so one change to new database results in a lot of work.

There are some solutions such as Spring Data, Hibernate OGM, TopLink NoSQL but it's at a high level. In other words, if this high-level API hasn't support to a particular database the result going to be either changing a high-level API or use the API from NoSQL database directly, so lost a lot of code. This solution has several issues:

- The database vendor need to be worried about the high-level mapping to Java world
- The solution provider needs to be concerned about the low level of communication with a particular database. The database vendor needs to “copy” this communication solutions to all Java vendors.
- To a Java developer there are two lock-in types: If a developer uses an API directly for a change, it will lose code. If a developer uses a high-level mapping, this developer has lock-in in a Java solution because if this high level hasn't support to a particular NoSQL database, the developer needs to change to either Java solution or use an API NoSQL directly.

The solve this problem the API should have two layers:

- The communication layer: the driver from a particular database that connects Java to an accurate database. This layer has four specializations, one to each NoSQL type.
- The mapping level: its duty is to high concept to Java developers, this layer has annotations and integration to other specializations.

These APIs are optional each other, in other words, a Java solution just needs to implement a great solution, and the database vendors need to implements the connection API.

2.1. Eclipse JNoSQL

The Eclipse JNoSQL is a several tool to make easy integration between the Java Application with the NoSQL. JNoSQL has a standard API. However, NoSQL has a diversity even when both are the same type. Eg. two column family databases, HBase and Cassandra, they have particular behavior and resource that make their individual such as Cassandra Query Language and consistency level that just does exist on Cassandra. So the API must be extensive and configurable to have support also to a specific database. To solve this problem the project gonna have two layers:

- **Communication API:** An API just to communicate with the database, exactly what JDBC does to SQL. This API is going to have four specializations, one for each kind of database.
- **Mapping API:** An API to do integration and do the best integration with the Java developer. That is going to be annotation drive and going to have integration with other technologies like Bean Validation, etc.

2.1.1. Communication API

The communication has a goal do the low-level API, in other words, communicate with the NoSQL databases. This project is going to work as a driver to NoSQL databases. At overall it has four APIs inside, one for each NoSQL kind, beyond it own TCK. A test compatibility kit, the TCK, are a test group that makes sure if an A NoSQL database does support a database.

2.1.2. Mapping API

Mapping API is the integration and mapping layer, in other words, it takes the communication level, and it does integration with others technologies such as Bean Validation and also with an entity model. It has a CDI engine.



Formula is elementary: Communication plus CDI equals to Mapping.

As communication have, mapping also has an API to each database flavor. Using CDI as the engine, each component is configurable, and it has features such as:

- Persist an object through annotation
- Make replaceable any component (reflections, entity conversions, cache, persistence lifecycle and more).
- Observer event a continued existence database lifecycle (each databases kind has an individual event).

An important point about CDI events is how easy is create and add new functionalities without change the core code. That is easy to use bean validation just listen to an event.

Chapter 3. Communication API Introduction

With the strategy to divide and conquer on JNoSQL. The communication API was born, it has the goal to be 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 is a change to another database provider has lesser than using the specific API.

To cover the three kinds of database, this API has three packages, one for each data bank.

- [org.jnosql.column](#)
- [org.jnosql.document](#)
- [org.jnosql.key](#)



The package name might change on the Jakarta EE process.

There isn't communication API because of the Graph API already does exist, that is [Apache TinkerPop](#).

So, if a database is multi-model, has support to more than one database, it will implement an API to each database which 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
- Factory
- Manager
- Entity
- Value

3.1. The API structure

The communication has four projects:

- The **communication-core**: The JNoSQL API communication commons to all types.
- The **communication-key-value**: The JNoSQL communication API layer to key-value database.
- The **communication-column**: The JNoSQL communication API layer to column database.
- The **communication-document**: The JNoSQL communication API layer to document database.

Each module works separately, thereby a NoSQL vendor just needs to implement the specific type, e.g., a key-value provider will apply a key-value API. If a NoSQL already has a driver, this API can work as an adapter with the current one. To multi-model NoSQL providers will implement the API whose they need.



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

3.2. Value

This interface represents the value that will store, that is a wrapper to be a bridge between the database and the application. Eg. If a database does not support a Java type, it may do the conversion with easily.

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

The Value interface has the methods:

- `Object get();` Returns the value as Object
- `<T> T get(Class<T> clazz);` Does the conversion process to the required type that is the safer way to do it. If the type required doesn't have support it will throw an exception, although, the API allows to create custom converters.
- `<T> T get(TypeSupplier<T> typeSupplier);` Similar to the previous method, it does the conversion process but using a structure that uses generics such as List, Map, Stream and Set.

```
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();
```

3.2.1. Make custom Writer and Reader

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

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

Both class implementations load from Java SE ServiceLoader resource. So, to Communication API learn a new type just register on ServiceLoader, e.g., Given a Money type:

```

public class Money {

    private final String currency;

    private final BigDecimal value;

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

    public String getCurrency() {
        return currency;
    }

    public BigDecimal getValue() {
        return value;
    }

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

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

```

Just to be more didactic the book creates a simple money representation. As everyone knows that is not a good practice reinventing the wheel, so in production the Java Developer must use mature Money APIS such as [moneta](#) that is the reference implementation of [JSR 354](#).

The first step is to create the converter to a custom type to a database, the `ValueWriter`. It has two methods:

- `boolean isCompatible(Class clazz)`: Check if the given class has support for this implementation.
- `S write(T object)`: Once the implementation supports the type, the next step converts a `T` instance to `S` type.


```

public class MoneyValueWriter implements ValueWriter<Money, String> {

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

    @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. This interface has two methods:

- `boolean isCompatible(Class clazz)`; Check if the given class has support for this implementation.
- `<T> T read(Class<T> clazz, Object value)`; Converts to the `T` type from `Object` instance.

```

public class MoneyValueReader implements ValueReader {

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

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

```

After all, the both implementation were done, the last step is to register them into two files:

- `META-INF/services/org.jnosql.ValueReader`
- `META-INF/services/org.jnosql.ValueWriter`

Each file will have the qualified of this respective implementation:

The file `org.jnosql.ValueReader` will have:

```
my.company.MoneyValueReader
```

The file `org.jnosql.ValueWriter` will have:

```
my.company.MoneyValueWriter
```

```
Value value = Value.of("BRL 10.0");  
Money money = value.get(Money.class);  
List<Money> list = value.get(new TypeReference<List<Money>>() {});  
Set<Money> set = value.get(new TypeReference<Set<Money>>() {});;
```

3.3. Element Entity

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

- **Document**
- **Column**

3.3.1. 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 an another document inside, the subdocument concept.

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

The way to storage information in subdocuments will also depend on each driver's implementation.

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();
```

3.3.2. 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 storage subcolumn will also depend on each driver implementation as well as the information.

To access the information from `Column` it has an alias method to `Value`, thus you can do a conversion directly from `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();
```

3.4. Entity

The Entity is the body of the information that goes to the database; each database has an Entity:

- `ColumnEntity`
- `DocumentEntity`
- `KeyValueEntity`

3.4.1. ColumnFamilyEntity

The `ColumnFamilyEntity` is an entity to 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("columnFamily");
entity.add(Column.of("id", Value.of(10L)));
entity.add(Column.of("version", 0.001));
entity.add(Column.of("name", "Diana"));
entity.add(Column.of("options", Arrays.asList(1, 2, 3)));

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

3.4.2. DocumentEntity

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

```
DocumentEntity entity = DocumentEntity.of("documentFamily");
String name = entity.getName();
entity.add(Document.of("id", Value.of(10L)));
entity.add(Document.of("version", 0.001));
entity.add(Document.of("name", "Diana"));
entity.add(Document.of("options", Arrays.asList(1, 2, 3)));

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

3.4.3. KeyValueEntity

The `KeyValueEntity` is the simplest structure; it has a tuple, 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);
```

3.5. Manager

The Manager is the class that pushes information to a database and retrieves it. The manager might have a synchronous and asynchronous implementation.

- `DocumentCollectionManager`
- `DocumentCollectionManagerAsync`
- `ColumnConfiguration`
- `ColumnConfigurationAsync`
- `BucketManager`

3.5.1. Document Manager

The manager class to a document type can be synchronous or asynchronous:

- `DocumentCollectionManager`: To do synchronous operations.
- `DocumentCollectionManagerAsync`: To do asynchronous operations.

DocumentCollectionManager

The `DocumentCollectionManager` 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);
DocumentCollectionManager manager = //instance;
//insert operations
manager.insert(entity);
manager.insert(entity, Duration.ofHours(2L)); //inserts with 2 hours of TTL
manager.insert(entities, Duration.ofHours(2L)); //inserts with 2 hours of TTL
//updates operations
manager.update(entity);
manager.update(entities);
```

DocumentCollectionManagerAsync

The `DocumentCollectionManagerAsync` is the class that manages the persistence on an asynchronous way to document collection.

```
DocumentEntity entity = DocumentEntity.of("collection");
Document diana = Document.of("name", "Diana");
entity.add(diana);

List<DocumentEntity> entities = Collections.singletonList(entity);
DocumentCollectionManagerAsync managerAsync = //instance

//insert operations
managerAsync.insert(entity);
managerAsync.insert(entity, Duration.ofHours(2L)); //inserts with 2 hours of TTL
managerAsync.insert(entities, Duration.ofHours(2L)); //inserts with 2 hours of TTL
//updates operations
managerAsync.update(entity);
managerAsync.update(entities);
```

Sometimes on an asynchronous process, is important to know when this process is over, so the `DocumentCollectionManagerAsync` also has callback support.

```
Consumer<DocumentEntity> callBack = e -> {};
managerAsync.insert(entity, callBack);
managerAsync.update(entity, callBack);
```

Search information on a document collection

Diana has support to retrieve information from both ways synchronous and asynchronous from the `DocumentQuery` class. The `DocumentQuery` has information such as sort type, document and also the condition to retrieve information.

The condition on `DocumentQuery` is given from `DocumentCondition`, which has the status and the document. Eg. The condition behind is to find a name equal "Ada".

```
DocumentCondition nameEqualsAda = DocumentCondition.eq(Document.of("name", "Ada"));
```

Also, the developer can use the aggregators such as **AND**, **OR** e **NOT**.

```
DocumentCondition nameEqualsAda = DocumentCondition.eq(Document.of("name", "Ada"));
DocumentCondition youngerThan2Years = DocumentCondition.lt(Document.of("age", 2));
DocumentCondition condition = nameEqualsAda.and(youngerThan2Years);
DocumentCondition nameNotEqualsAda = nameEqualsAda.negate();
```

If there isn't a condition in the query that means the query will try to retrieve all information from the database, similar to a "select * from database" in a relational database, just remembering that the return depends on the driver. It is important to say that not all NoSQL databases have support for this resource.

`DocumentQuery` also has pagination feature to define where the data start, and it limits.

```
DocumentCollectionManager manager = //instance;
DocumentCollectionManagerAsync managerAsync = //instance;
DocumentQuery query = DocumentQueryBuilder.select().from("collection").where("age").
lt(10).and("name").eq("Ada").orderBy("name").asc().limit(10).skip(2).build();
List<DocumentEntity> entities = manager.select(query);
Optional<DocumentEntity> entity = manager.singleResult(query);
Consumer<List<DocumentEntity>> callback = e -> {};
managerAsync.select(query, callback);
```

Removing information from Document Collection

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

It is smoother than `DocumentQuery` because there isn't pagination and sort feature, once this information is unnecessary to remove information from database.

```

DocumentCollectionManager manager = //instance;
DocumentCollectionManagerAsync managerAsync = //instance;
DocumentDeleteQuery query = DocumentQueryBuilder.delete().from("collection").where(
"age").gt(10).build();
manager.delete(query);
managerAsync.delete(query);
managerAsync.delete(query, v -> {});

```

3.5.2. Column Manager

The Manager class for the column family type can be synchronous or asynchronous:

- **ColumnFamilyManager**: To do synchronous operations.
- **ColumnFamilyManagerAsync**: To do asynchronous operations.

ColumnFamilyManager

The **ColumnFamilyManager** is the class that manages the persistence on the synchronous way to column family.

```

ColumnEntity entity = ColumnEntity.of("columnFamily");
Column diana = Column.of("name", "Diana");
entity.add(diana);

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

//inserts operations
manager.insert(entity);
manager.insert(entity, Duration.ofHours(2L)); //inserts with 2 hours of TTL
manager.insert(entities, Duration.ofHours(2L)); //inserts with 2 hours of TTL
//updates operations
manager.update(entity);
manager.update(entities);

```

ColumnFamilyManagerAsync

The **ColumnFamilyManagerAsync** is the class that manages the persistence on the asynchronous way to column family.

```

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

List<ColumnEntity> entities = Collections.singletonList(entity);
ColumnFamilyManagerAsync managerAsync = null;

//inserts operations
managerAsync.insert(entity);
managerAsync.insert(entity, Duration.ofHours(2L)); //inserts with 2 hours of TTL
managerAsync.insert(entities, Duration.ofHours(2L)); //inserts with 2 hours of TTL
//updates operations
managerAsync.update(entity);
managerAsync.update(entities);

```

Sometimes on an asynchronous process, is important to know when this process is over, so the `ColumnFamilyManagerAsync` also has callback support.

```

Consumer<ColumnEntity> callBack = e -> {};
managerAsync.insert(entity, callBack);
managerAsync.update(entity, callBack);

```

Search information on a column family

Diana has support to retrieve information from both ways synchronous and asynchronous from the `ColumnQuery` class. The `ColumnQuery` has information such as sort type, document and also the condition to retrieve information.

The condition on `ColumnQuery` is given from `ColumnCondition`, whose it has the status and the column. Eg. The condition behind is to find a name equal "Ada".

```

ColumnCondition nameEqualsAda = ColumnCondition.eq(Column.of("name", "Ada"));

```

Also, the developer can use the aggregators such as **AND**, **OR** e **NOT**.

```

ColumnCondition nameEqualsAda = ColumnCondition.eq(Column.of("name", "Ada"));
ColumnCondition youngerThan2Years = ColumnCondition.lt(Column.of("age", 2));
ColumnCondition condition = nameEqualsAda.and(youngerThan2Years);
ColumnCondition nameNotEqualsAda = nameEqualsAda.negate();

```

If there isn't condition at the query that means the query will try to retrieve all information from the database, look like a "select * from database" in a relational database, just to remember the return depends on from driver. It is important to say that not all NoSQL databases have support for this resource.

`ColumnQuery` also has pagination feature to define where the data start, and it limits.


```

ColumnFamilyManager manager = //instance;
ColumnFamilyManagerAsync managerAsync = //instance;
ColumnQuery query = ColumnQueryBuilder.select().from("collection")
    .where("age").lt(10).and("name").eq("Ada").orderBy("name").asc().limit(10).skip(2).build();

List<ColumnEntity> entities = manager.select(query);
Optional<ColumnEntity> entity = manager.singleResult(query);

Consumer<List<ColumnEntity>> callback = e -> {};
managerAsync.select(query, callback);

```

Removing information from Column Family

Such as `ColumnQuery` there is a class to remove information from the column database type: A `ColumnDeleteQuery` type.

It is smoother than `ColumnQuery` because there isn't pagination and sort feature, once this information is unnecessary to remove information from database.

```

ColumnFamilyManager manager = //instance;
ColumnFamilyManagerAsync managerAsync = //instance;

ColumnDeleteQuery query = ColumnQueryBuilder.delete()
    .from("collection").where("age").gt(10).build();

manager.delete(query);

managerAsync.delete(query);
managerAsync.delete(query, v -> {});

```

3.5.3. BucketManager

The `BucketManager` is the class which saves the `KeyValueEntity` on a synchronous way in key-value database.

```

BucketManager bucketManager= null;
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)); //two hours TTL
bucketManager.put(entity, Duration.ofHours(2)); //two hours TTL

```

Removing and retrieve information from a key-value database

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"));
```

3.5.4. Querying by text at Communication API

The communication API also has a query as text. These queries will convert to an operation that already exists in the Manager interface from the `query` method, thereby, these operations might return an `UnsupportedOperationException` if a NoSQL has not support to that procedure.

This query has basic principles:

- All instructions end with a break like `\n`
- It is case sensitive
- All keywords must be in lowercase
- The goal is to look like SQL, however simpler
- Even passing in the syntax and parser the query, a specific implementation may not support an operation. E.g., Column family may not support query in a different field that is not the ID field.

Key-Value

The key-value has three operations, **put**, **remove** and **get**.

Get

Retrieving data for an entity is done using a GET statement:

```
get_statement ::= GET ID (',' ID)*
//sample
get "Apollo" //to return an element where the id is 'Apollo'
get "Diana" "Artemis" //to return a list of values from the ids
```

Remove

To delete one or more entities, use the remove statement

```
del_statement ::= GET ID (',' ID)*
//sample
remove "Apollo"
remove "Diana" "Artemis"
```

Put

To either insert or override values from a key-value database, use the put statement.

```
put_statement ::= PUT {KEY, VALUE, [TTL]}
//sample
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
```

Column and Document

Both have sample syntax that looks like a SQL query, however, remember it has a limitation and does not support joins. Document types are usually more queriable than a column type. They have four operations: insert, update, delete, and select.

Insert

Inserting data for an entity is done using an INSERT statement:

```
insert_statement ::= INSERT entity_name (name = value, (`,` name = value) *) [ TTL ]
//sample
insert God (name = "Diana", age = 10)
insert God (name = "Diana", age = 10, power = {"sun", "god"})
insert God (name = "Diana", age = 10, power = {"sun", "god"}) 1 day
```

Update

Updating an entity is done using an update statement:

```
update_statement ::= UPDATE entity_name (name = value, (`,` name = value) *)
//sample
update God (name = "Diana", age = 10)
update God (name = "Diana", age = 10, power = {"hunt", "moon"})
```

Delete

Deleting either an entity or fields uses the delete statement

```
delete_statement ::= DELETE [ simple_selection (`,` simple_selection) ]
                    FROM entity_name
                    WHERE where_clause
//sample
delete from God
delete name, age ,adress.age from God where id = "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 for corresponding to the query.

```
select_statement ::= SELECT ( select_clause | '*' )
                  FROM entity_name
                  [ WHERE where_clause ]
                  [ SKIP (integer) ]
                  [ LIMIT (integer) ]
                  [ ORDER BY ordering_clause ]

//sample
select * from God
select name, age ,adress.age from God order by name desc age desc
select * from God where birthday between "01-09-1988" and "01-09-1988" and salary =
12
select name, age ,adress.age from God skip 20 limit 10 order by name desc age desc
```

PreparedStatement and PreparedStatementAsync

To run a query dynamically, use the `prepare` method in the manager 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 = documetManager.prepare("select * from Person
where name = @name");
preparedStatement.bind("name", "Ada");
List<DocumentEntity> adas = preparedStatement.getResultList();
```

```
PreparedStatementAsync preparedStatement = documetManagerAsync.prepare("select * from
Person where name = @name");
preparedStatement.bind("name", "Ada");
Consumer<List<DocumentEntity>> callback = ...;
preparedStatement.getResultList(callback);
```

WHERE

The **WHERE** clause specifies a filter to the result. These filters are booleans operations that are composed of one or more conditions appended with the and (**AND**) and or (**OR**) operators.

Conditions

Condition performs different computations or actions depending on whether a boolean query condition evaluates to **true** or **false**. The conditions are composed of three elements:

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

3. **Value**, that data that receives the operation.

Operators

The Operators are:

Table 5. 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

The value

The value is the last element in a condition, and it defines what it'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 two double quotes ". E.g.: `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"}`

SKIP

The **SKIP** option to a **SELECT** statement defines where the query should start,

LIMIT

The **LIMIT** option to a **SELECT** statement limits the number of rows returned by a query,

ORDER BY

The **ORDER BY** clause allows selecting the order of the returned results. It takes as argument a list of column names along with the order for the column (**ASC** for ascendant and **DESC** for the descendant, omitting the order being equivalent to **ASC**).

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`



To graph API check [Gremlin](#)

3.6. Factory

The factory class creates the **Managers**.

- **ColumnFamilyManagerAsyncFactory**
- **ColumnFamilyManagerFactory**
- **BucketManagerFactory**
- **DocumentCollectionManagerFactory**
- **DocumentCollectionManagerAsyncFactory**

3.6.1. Column Family Manager Factory

The factory classes have the duty to create the column family manager.

- **ColumnFamilyManagerAsyncFactory**
- **ColumnFamilyManagerFactory**

The `ColumnFamilyManagerAsyncFactory` and `ColumnFamilyManagerFactory` creates the manager synchronously and asynchronously respectively.

```
ColumnFamilyManagerFactory factory = //instance
ColumnFamilyManagerAsyncFactory asyncFactory = //instance
ColumnFamilyManager manager = factory.get("database");
ColumnFamilyManagerAsync managerAsync = asyncFactory.getAsync("database");
```

The factories were separated intentionally, as not all databases support synchronous and asynchronous operations.

3.6.2. Document Collection Factory

The factory classes have the duty to create the document collection manager.

- **DocumentCollectionManagerFactory**
- **DocumentCollectionManagerAsyncFactory**

The `DocumentCollectionManagerAsyncFactory` and `DocumentCollectionManagerFactory` creates the manager synchronously and asynchronously respectively.

```
DocumentCollectionManagerFactory factory = //instance
DocumentCollectionManagerAsyncFactory asyncFactory = //instance
DocumentCollectionManager manager = factory.get("database");
DocumentCollectionManagerAsync managerAsync = asyncFactory.getAsnc("database");
```

The factories were separated intentionally, as not all databases support synchronous and asynchronous operations.

3.6.3. Bucket Manager Factory

The factory classes have the duty to create the bucket manager.

```
BucketManagerFactory bucketManager= //instance
BucketManager bucket = bucketManager.getBucketManager("bucket");
```

Beyond the `BucketManager`, some databases have support for particular structure represented in the Java world such as `List`, `Set`, `Queue` e `Map`.

```
List<String> list = bucketManager.getList("list", String.class);
Set<String> set = bucketManager.getSet("set", String.class);
Queue<String> queue = bucketManager.getQueue("queue", String.class);
Map<String, String> map = bucketManager.getMap("map", String.class, String.class);
```

These methods may return an `java.lang.UnsupportedOperationException` if the database does not support any of structures.

3.7. Configuration

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

Once there are a large diversity configuration flavors on such as P2P, master/slave, thrift communication, HTTP, etc. The implementation may be different, however, 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.

3.7.1. Settings

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

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

3.7.2. Document Configuration

On the document collection configuration, there are two classes, `DocumentConfiguration` and `DocumentConfigurationAsync` to `DocumentCollectionManagerFactory` and `DocumentCollectionManagerAsyncFactory` respectively.

```
DocumentConfiguration configuration = //instance
DocumentConfigurationAsync configurationAsync = //instance
DocumentCollectionManagerFactory managerFactory = configuration.get();
DocumentCollectionManagerAsyncFactory managerAsyncFactory = configurationAsync
    .getAsync();
```

If a database has support to both synchronous and asynchronous, it may use `UnaryDocumentConfiguration` that implement both document configuration.

```
UnaryDocumentConfiguration unaryDocumentConfiguration = //instance
DocumentCollectionManagerFactory managerFactory = unaryDocumentConfiguration.get();
DocumentCollectionManagerAsyncFactory managerAsyncFactory =
    unaryDocumentConfiguration.getAsync();
```

3.7.3. Column Configuration

On the column family configuration, there are two classes, `ColumnConfiguration` and `ColumnConfigurationAsync` to `ColumnFamilyManagerFactory` and `ColumnFamilyManagerAsyncFactory` respective.

```
ColumnConfiguration configuration = //instance
ColumnConfigurationAsync configurationAsync = //instance
ColumnFamilyManagerFactory managerFactory = configuration.get();
ColumnFamilyManagerAsyncFactory managerAsyncFactory = configurationAsync.getAsync();
```


If a database has support to both synchronous and asynchronous, it may use `UnaryColumnConfiguration` that implement both document configuration.

```
UnaryColumnConfiguration unaryDocumentConfiguration = //instance
ColumnFamilyManagerFactory managerFactory = unaryDocumentConfiguration.get();
ColumnFamilyManagerAsyncFactory managerAsyncFactory = unaryDocumentConfiguration
.getAsync();
```

3.7.4. Key Value Configuration

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

```
KeyValueConfiguration configuration = //instance
BucketManagerFactory managerFactory = configuration.get();
```

3.8. The diversity on NoSQL database

On NoSQL world beyond the several types, it's trivial a particular database has features that do exist on this provider. When there is a change among the types, column family, and document collection, there is a considerable change. Notably, with there a switch to the same kind such as column family to column family, e.g., Cassandra to HBase, there is the same problem once Cassandra has featured such as Cassandra query language and consistency level. The communication API allows looking the variety on NoSQL database. The configurations classes, and entity factory return specialist class from a provider.

```
public interface ColumnFamilyManagerFactory<SYNC extends ColumnFamilyManager>
extends AutoCloseable {
    SYNC get(String database);
}
```

A `ColumnFamilyManagerFactory` return a class the implements `ColumnFamilyManager`. E.g: Using a particular resource from Cassandra driver.

```

CassandraConfiguration condition = new CassandraConfiguration();
try(CassandraDocumentEntityManagerFactory managerFactory = condition.get()) {
    CassandraColumnFamilyManager columnEntityManager = managerFactory.get(KEY_SPACE);
    ColumnEntity entity = ColumnEntity.of(COLUMN_FAMILY);
    Column id = Column.of("id", 10L);
    entity.add(id);
    entity.add(Column.of("version", 0.001));
    entity.add(Column.of("name", "Diana"));
    entity.add(Column.of("options", Arrays.asList(1, 2, 3)));
    columnEntityManager.save(entity);
    //common implementation
    ColumnQuery query = ColumnQuery.of(COLUMN_FAMILY);
    query.and(ColumnCondition.eq(id));
    Optional<ColumnEntity> result = columnEntityManager.singleResult(query);
    //cassandra implementation
    columnEntityManager.save(entity, ConsistencyLevel.THREE);
    List<ColumnEntity> entities = columnEntityManager.cql("select * from
newKeySpace.newColumnFamily");
    System.out.println(entities);
}

```

Chapter 4. Mapping API Introduction

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

This level is in charge to do integration among technologies such as Bean Validation. The Mapping API has annotations that make the Java developer life easier. As communication project, it must be extensible and configurable to keep the diversity on NoSQL database.

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

- `org.jnosql.persistence.column`
- `org.jnosql.persistence.document`
- `org.jnosql.persistence.graph`
- `org.jnosql.persistence.key`



The package name might change on the Jakarta EE process.

4.1. The Mapping structure

The mapping API has six parts:

- The **persistence-core**: The mapping commons project.
- The **persistence-configuration**: The configuration in mapper.
- The **persistence-column**: The mapping to column NoSQL database.
- The **persistence-document**: The mapping to document NoSQL database.
- The **persistence-key-value**: The mapping to key-value NoSQL database.
- The **persistence-validation**: The support to Bean Validation



Each module works separately as Communication API.



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

4.2. Models Annotation

As mentioned previously, the Mapping API has annotations that make the Java developer life easier; these annotations have two categories:

- Annotation Models
- Qualifier annotation

4.2.1. Annotation Models

The annotation model is to convert the entity model to the entity on communication, the communication entity:

- Entity
- Column
- MappedSuperclass
- Id
- Embeddable
- Convert

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

Entity

This annotation maps the class to Eclipse JNoSQL. It has an unique attribute called `name` . This attribute is to inform either the column family name or the document collection name, etc. The default value is the simple name of a class, for example, given the `org.jnosql.demo.Person` class the default name will `Person`.

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

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

An entity as a field will incorporate as a sub entity. E.g., In a document, this entity field will convert to a subdocument.

```

@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"
  }
}

```

Column

This annotation is used to define which fields on an Entity will be persisted. It also has a unique attribute name to specify that name on Database, and the default value is the field name.

```

@Entity
public class Person {
    @Column
    private String nickname;
    @Column
    private String name;
    @Column
    private List<String> phones;
    //ignored
    private String address;
}

```

MappedSuperclass

If this annotation is on the parent class, it will persist its information as well. So beyond the son class, it will store any field that is in Parent class with Column annotation.

```

@Entity
public class Dog extends Animal {

    @Column
    private String name;

}

@MappedSuperclass
public class Animal {

    @Column
    private String race;

    @Column
    private Integer age;

}

```

On this sample above, when saves a **Dog** instance, it saves the **Animal** case too, explicitly will save the fields **name**, **race** and **age**.

Id

It shows which attribute is the id, or the key in key-value types, thus the value will be the remaining information. The way of storing the class will depend on the database driver.

```

@Entity
public class User implements Serializable {

    @Id
    private String userName;

    private String name;

    private List<String> phones;
}

```

Embeddable

Defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the object.

```

@Entity
public class Book {

    @Column
    private String name;

    @Column
    private Author author;

}

@Embeddable
public class Author {

    @Column
    private String name;

    @Column
    private Integer age;

}

```

Convert

As Communication, the Mapping API has a converter at abstraction level. This feature is useful, e.g., to cipher a field, String to String, or just to do a converter to a custom type using annotation. The **Converter** annotation has a parameter, and an `AttributeConverter` implementation class can be used. Eg. The sample bellow shows how to create a converter to a custom Money class.

```

@Entity
public class Worker {
    @Column
    private String name;
    @Column
    private Job job;
    @Column("money")
    @Convert(MoneyConverter.class)
    private Money salary;
}

public class MoneyConverter implements AttributeConverter<Money, String>{
    @Override
    public String convertToDatabaseColumn(Money attribute) {
        return attribute.toString();
    }
    @Override
    public Money convertToEntityAttribute(String dbData) {
        return Money.parse(dbData);
    }
}

public class Money {
    private final String currency;

    private final BigDecimal value;

    //.....
}

```

Collection

The Mapping layer has support for `java.util.Collection` to both simple elements such as `String`, `Integer` that will send to the communication API the exact value and class that has fields inside once the class has either `Entity` or `Embedded` annotation otherwise will post as the first scenario, like `String` or any amount without converter process.

It has support to:

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


```

@Entity
public class Person {

    @Id
    private Long id;

    @Column
    private String name;

    @Column
    private List<String> phones;

    @Column
    private List<Address> address;
}

```

```

@Embeddable
public class Address {

    @Column
    private String street;

    @Column
    private String city;
}

```

```

{
  "_id": 10,
  "address": [
    {
      "city": "São Paulo",
      "street": "Av nove de julho"
    },
    {
      "city": "Salvador",
      "street": "Rua Engenheiro Jose Anasoh"
    }
  ],
  "name": "Name",
  "phones": [
    "234",
    "432"
  ]
}

```

4.2.2. Qualifier annotation

That is important to work with more than one type of the same application.

```
@Inject
private DocumentRepository repositoryA;
@Inject
private DocumentRepository repositoryB;
```

Two injections with the same interface, CDI throws an ambiguous exception. There is the **Database** qualifier to fix this problem. It has two attributes:

- **DatabaseType**: The database type, key-value, document, column, graph.
- **provider**: The provider database name, eg. "cassandra", "hbase", "mongodb". So using the **Database** qualifier:

```
@Inject
@Database(value = DatabaseType.DOCUMENT, provider = "databaseA")
private DocumentRepository repositoryA;
@Inject
@Database(value = DatabaseType.DOCUMENT, provider = "databaseB")
private DocumentRepository repositoryB;
```

Beyond this annotation, the producer method with the entity manager is required.

The benefits using this qualifier instead of creating a new one is that if the Manager Entity is produced using **Database** as a qualifier, It will create classes such as DocumentRepository, ColumnRepository, etc. automatically.

4.2.3. ConfigurationUnit

Storage the database's configuration such as password and user outside the code is important, Eclipse JNoSQL has the ConfigurationUnit annotation. That reads the configuration from a file such as XML, YAML, and JSON file and inject to create a factory. The default configuration structure is within either **META-INF** or **WEB-INF** folder.

JSON file structure

```
[
  {
    "description":"that is the description",
    "name":"name",
    "provider":"class",
    "settings":{
      "key":"value"
    }
  },
  {
    "description":"that is the description",
    "name":"name-2",
    "provider":"class",
    "settings":{
      "key":"value"
    }
  }
]
```

XML file structure

```
<?xml version="1.0" encoding="UTF-8"?>
<configurations>
  <configuration>
    <description>that is the description</description>
    <name>name</name>
    <provider>class</provider>
    <settings>
      <entry>
        <key>key2</key>
        <value>value2</value>
      </entry>
      <entry>
        <key>key</key>
        <value>value</value>
      </entry>
    </settings>
  </configuration>
</configurations>
```

YAML file structure

```

configurations:
  - description: that is the description
    name: name
    provider: class
    settings:
      key: value
      key2: value2

```

Injection the code

With the configuration file, the next step is to inject the dependency in the application. The default behavior supports the following classes:

- BucketManagerFactory
- DocumentCollectionManagerAsyncFactory
- DocumentCollectionManagerAsyncFactory
- ColumnFamilyManagerAsyncFactory
- ColumnFamilyManagerAsyncFactory

```

@Inject
@ConfigurationUnit(fileName = "column.xml", name = "name")
private ColumnFamilyManagerFactory<?> factoryA;

@Inject
@ConfigurationUnit(fileName = "document.json", name = "name-2")
private DocumentCollectionManagerFactory factoryB;

@Inject
@ConfigurationUnit
private BucketManagerFactory factoryB;

```

4.3. Template classes

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

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

The default workflow has six events:

1. **firePreEntity:** The Object received from mapping.

2. **firePreEntityDataBaseType**: Just like the previous event, however, to a specific database, in other words, each database has a particular event.
3. **firePreAPI**: The object converted to a communication layer.
4. **firePostAPI**: The entity connection as a response from the database.
5. **firePostEntity**: The entity model from the API low level from the `firePostAPI`.
6. **firePostEntityDataBaseType**: Just like the previous event, however, to a specific database, in other words, each database has a particular event.

4.3.1. DocumentTemplate

This template has the duty to be a bridge between the entity model and communication API to document collection. It has two classes `DocumentTemplate` and `DocumentTemplateAsync`, one for the synchronous and the other for the asynchronous work.

The `DocumentTemplate` is the document template for the synchronous tasks. It has three components:

- **DocumentEntityConverter**: That converts an entity to communication API, e.g., The Person to DocumentEntity.
- **DocumentCollectionManager**: The document collection entity manager.
- **DocumentWorkflow**: The workflow to update and insert methods.

```
DocumentTemplate template = //instance

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 do both remove and retrieve information from document collection there are **DocumentQuery** and **DocumentDeleteQuery**.

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

List<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);
```

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

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

The next step is to produce a **DocumentCollectionManager**:

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

To work with more than one Document Template, there are two approaches:

1) Using qualifiers:

```

@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 DocumentCollectionManager getManagerA() {
    DocumentCollectionManager manager = //instance
    return manager;
}

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

```

2) Using the **DocumentTemplateProducer** class

```

@Inject
private DocumentTemplateProducer producer;

public void sample() {
    DocumentCollectionManager managerA = //instance;
    DocumentCollectionManager managerB = //instance
    DocumentTemplate templateA = producer.get(managerA);
    DocumentTemplate templateB = producer.get(managerB);
}

```

4.3.2. DocumentTemplateAsync

The **DocumentTemplateAsync** is the document template for the asynchronous tasks. It has two components:

- **DocumentEntityConverter:** That converts an entity to communication API, e.g., The Person to DocumentEntity.
- **DocumentCollectionManagerAsync:** The document collection entity manager asynchronous.

```

DocumentTemplateAsync templateAsync = //instance

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);

Consumer<Person> callback = p -> {};
templateAsync.insert(person);
templateAsync.insert(person, Duration.ofHours(1L));
templateAsync.insert(person, callback);
templateAsync.insert(people);

templateAsync.update(person);
templateAsync.update(person, callback);
templateAsync.update(people);

```

For information removal and retrieval there are **DocumentQuery** and **DocumentDeleteQuery**, respectively, also the callback method can be used.

```

Consumer<List<Person>> callBackPeople = p -> {};
Consumer<Void> voidCallBack = v ->{};
templateAsync.find(query, callBackPeople);
templateAsync.delete(deleteQuery);
templateAsync.delete(deleteQuery, voidCallBack);

```

To use a document template just follow the CDI style and put an **@Inject** on the field.

```

@Inject
private DocumentTemplateAsync template;

```

The next step is produced a **DocumentCollectionManagerAsync**:

```

@Produces
public DocumentCollectionManagerAsync getManager() {
    DocumentCollectionManagerAsync managerAsync = //instance
    return manager;
}

```

To work with more than one Document Template, there are two approaches:

1) Using qualifiers:


```

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

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

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

@Produces
@Database(value = DatabaseType.DOCUMENT, provider = "databaseB")
public DocumentCollectionManagerAsync getManagerB() {
    DocumentCollectionManager manager = //instance
    return manager;
}

```

2) Using the **DocumentTemplateAsyncProducer**

```

@Inject
private DocumentTemplateAsyncProducer producer;

public void sample() {
    DocumentCollectionManagerAsync managerA = //instance;
    DocumentCollectionManagerAsync managerB = //instance
    DocumentTemplateAsync templateA = producer.get(managerA);
    DocumentTemplateAsync templateB = producer.get(managerB);
}

```

4.3.3. ColumnTemplate

This template has the duty to be a bridge between the entity model and the communication to a column family. It has two classes **ColumnTemplate** and **ColumnTemplateAsync**, one for the synchronous and the other for the asynchronous work.

The **ColumnTemplate** is the column template for the synchronous tasks. It has three components:

- **ColumnEntityConverter**: That converts an entity to communication API, e.g., The Person to ColumnFamilyEntity.
- **ColumnCollectionManager**: The communication column family entity manager.

- **ColumnWorkflow**: The workflow to update and insert methods.

```

ColumnTemplate template = //instance

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**, respectively, also the callback method can be used.

```

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

List<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);

```

To use a column template just follow the CDI style and put an **@Inject** on the field.

```

@Inject
private ColumnTemplate template;

```

The next step is produced a **ColumnFamilyManager**:

```

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

```

To work with more than one Column Template, there are two approaches:

1) Using qualifiers:

```
@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 ColumnFamilyManager getManagerA() {
    ColumnFamilyManager manager = //instance
    return manager;
}

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

2) Using the **ColumnTemplateProducer** class

```
@Inject
private ColumnTemplateProducer producer;

public void sample() {
    ColumnFamilyManager managerA = //instance;
    ColumnFamilyManager managerB = //instance
    ColumnTemplate templateA = producer.get(managerA);
    ColumnTemplate templateB = producer.get(managerB);
}
```

ColumnTemplateAsync

The **ColumnTemplateAsync** is the document template for the asynchronous tasks. It has two components:

- **ColumnEntityConverter:** That converts an entity to communication API, e.g., The Person to ColumnFamilyEntity.
- **ColumnFamilyManagerAsync:** The communication column family entity manager

asynchronous.

```
ColumnTemplateAsync templateAsync = //instance

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);

Consumer<Person> callback = p -> {};
templateAsync.insert(person);
templateAsync.insert(person, Duration.ofHours(1L));
templateAsync.insert(person, callback);
templateAsync.insert(people);

templateAsync.update(person);
templateAsync.update(person, callback);
templateAsync.update(people);
```

For information removal and retrieval there **ColumnQuery** and **ColumnDeleteQuery**, respectively, also the callback method can be used.

```
Consumer<List<Person>> callBackPeople = p -> {};
Consumer<Void> voidCallBack = v ->{};
templateAsync.select(query, callBackPeople);
templateAsync.delete(deleteQuery);
templateAsync.delete(deleteQuery, voidCallBack);
```

To use a column template just follow the CDI style and put an **@Inject** on the field.

```
@Inject
private ColumnTemplateAsync template;
```

The next step is to produce a **ColumnFamilyManagerAsync**:

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

To work with more than one Column Template, there are two approaches:

1) Using qualifiers:

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

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

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

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

2) Using the **ColumnTemplateAsyncProducer**

```
@Inject
private ColumnTemplateAsyncProducer producer;

public void sample() {
    ColumnFamilyManagerAsync managerA = //instance;
    ColumnFamilyManagerAsync managerB = //instance
    ColumnTemplateAsync templateA = producer.get(managerA);
    ColumnTemplateAsync templateB = producer.get(managerB);
}
```

4.3.4. Key-Value template

The **KeyValueTemplate** is the template to synchronous tasks. It has three components: The **KeyValueTemplate** is responsible for persistence of an entity in a key-value database. It is composed basically for three components.

- **KeyValueEntityConverter**: That converts an entity to communication API, e.g., The Person to **KeyValueEntity**.
- **BucketManager**: The key-value entity manager.

- **KeyValueWorkflow**: The workflow to put method.

```
KeyValueTemplate template = null;
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);
```



To key-value templates both Entity and @Id are required. The @Id identifies the key and the whole entity will be the value. The API won't cover how this value persists this entity at NoSQL database.

To use a key-value template just follows the CDI style and put an @Inject on the field.

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

The next step is to produce a **BucketManager**:

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

To work with more than one key-value Template, there are two approaches: 1) Using qualifiers:

```

@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() {
    DocumentCollectionManager manager = //instance
    return manager;
}

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

```

2) Using the **KeyValueTemplateProducer** class

```

@Inject
private KeyValueTemplateProducer producer;

public void sample() {
    BucketManager managerA = //instance;
    BucketManager managerB = //instance
    KeyValueTemplate templateA = producer.get(managerA);
    KeyValueTemplate templateB = producer.get(managerB);
}

```

4.3.5. Graph template

The **GraphTemplate** is the column template to synchronous tasks. It has three components: The **GraphTemplate** is responsible for persistence of an entity in a Graph database using **Apache Tinkerpop**. It is composed basically for three components.

- **GraphConverter**: That converts an entity to communication API, e.g., The Person to Vertex.
- **Graph**: A Graph is a container object for a collection of Vertex, Edge, VertexProperty, and Property objects.
- **GraphWorkflow**: The workflow to update and insert methods.

```

GraphTemplate template = //instance

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);

```

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();

```

Querying with traversal

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

E.g.:

In the 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_000))
    .has("age", between(20, 25))
    .has("occupation", "Developer")
    .<Person>stream().collect(toList());

```

The next step is to return the engineer 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 put an `@Inject` on the field.

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

The next step: make a `*Graph*` instance eligible to CDI applying a produces method:

```
@Produces
public Graph getManager() {
    Graph graph = //instance
    return graph;
}
```

To work with more than one Graph Template, there are two approaches:

1) Using qualifiers:

```
@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() {
    Graph manager = //instance
    return graph;
}

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

2) Using the **GraphTemplateProducer** class

```
@Inject
private GraphTemplateProducer producer;

public void sample() {
    Graph graphA = //instance;
    Graph graphB = //instance
    GraphTemplate templateA = producer.get(graphA);
    GraphTemplate templateB = producer.get(graphB);
}
```

4.3.6. Querying by text at Mapping API

As in communication layer, the Mapping has a query by text. Both communication and Mapping has the **query** and **prepare** methods, however, at the Mapping API, it will convert the fields and entities to native names from the Entity and Column annotations.

Key-Value

In the Key-value database, there is a **KeyValueTemplate** in this NoSQL storage technology. Usually, all the operations are defined by the ID, therefore, it has a smooth query.

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

Column-Family

The column family has a little more complex structure, however, the search from the key is still recommended. E.g.: Both Cassandra and HBase have a secondary index, however, neither have a guarantee about performance, and they usually recommend to have a second table whose rowkey is your "secondary index" and is only being used to find the rowkey needed for the actual table. Given Person as an entity and then we would like to operate from the field ID, which is the entity from the Entity.

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

Document Collection

The document types allow more complex queries, so more complex entities, with a document type a developer can find from different fields than id easily and naturally. Also, there is NoSQL documents type that supports aggregations query, however, Eclipse JNoSQL does not support this yet. At the Eclipse JNoSQL API perspective, the document and column type is pretty similar, but with the document, a Java developer might do a query from a field that isn't a key and neither

returns an unported operation exception or adds a secondary index for this. So, given the same Person entity with document NoSQL type, a developer can do more with queries, such as "person" between "age."

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

Graph

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. A graph database has the vertex and the edge. The edge is an object that holds the relationship information about the edges and has direction and properties that make it perfect to maps or human relationship. To the Graph API, Eclipse JNoSQL uses the Apache Tinkerpop. Likewise, the GraphTemplate is a wrapper to convert a Java entity to Vertex in TinkerPop.

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

4.4. Repository

In addition to a template class, the Mapping API has the Repository. This interface helps the Entity repository to save, update, delete and retrieve information. To use Repository, just need to create a new interface that extends the **Repository**.

```
interface PersonRepository extends Repository<Person, String> {  
  
}
```

The qualifier is mandatory to define the database type that will use at the injection point moment.

```
@Inject  
@Database(DatabaseType.DOCUMENT)  
private PersonRepository documentRepository;  
@Inject  
@Database(DatabaseType.COLUMN)  
private PersonRepository columnRepository;  
@Inject  
@Database(DatabaseType.KEY_VALUE)  
private PersonRepository keyValueRepository;  
@Inject  
@Database(DatabaseType.GRAPH)  
private PersonRepository graphRepository;
```

And then, make any manager class (**ColumnFamilyManager**, **DocumentCollectionManager**,

BucketManager and **Graph**) eligible to CDI defining a method with Produces annotation.

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

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

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

@Produces
public Graph getGraph() {
    Graph graph = //instance
    return graph;
}
```

To work with multiple database you can use qualifiers:

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

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

@Inject
@Database(value = DatabaseType.COLUMN, provider = "databaseA")
private PersonRepository columnRepositoryA;

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

@Inject
@Database(value = DatabaseType.KEY_VALUE, provider = "databaseA")
private UserRepository userRepositoryA;
@Inject
@Database(value = DatabaseType.KEY_VALUE, provider = "databaseB")
```

```

private UserRepository userRepositoryB;

@Inject
@Database(value = DatabaseType.GRAPH, provider = "databaseA")
private PersonRepository graphRepositoryA;

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

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

@Produces
@Database(value = DatabaseType.COLUMN, provider = "databaseB")
public ColumnFamilyManager getColumnFamilyManagerB() {
    ColumnFamilyManager manager = //instance
    return manager;
}

@Produces
@Database(value = DatabaseType.DOCUMENT, provider = "databaseA")
public DocumentCollectionManager getDocumentCollectionManagerA() {
    DocumentCollectionManager manager = //instance
    return manager;
}

@Produces
@Database(value = DatabaseType.DOCUMENT, provider = "databaseB")
public DocumentCollectionManager DocumentCollectionManagerB() {
    DocumentCollectionManager manager = //instance
    return manager;
}

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

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

```

```

@Produces
@Database(value = DatabaseType.GRAPH, provider = "databaseA")
public Graph getGraph() {
    Graph graph = //instance
    return graph;
}
@Produces
@Database(value = DatabaseType.GRAPH, provider = "databaseB")
public Graph getGraphB() {
    Graph graph = //instance
    return graph;
}

```

So, Eclipse JNoSQL will inject automatically.

```

PersonRepository repository = //instance

Person person = new Person();
person.setNickname("diana");
person.setName("Diana Goodness");

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

repository.save(person);
repository.save(people);

```

4.4.1. Query by method

The Repository also has a method query from the method name. These are the keywords:

- **findBy**: The prefix to find some information
- **deleteBy**: The prefix to delete some information

Also the operators:

- And
- Or
- Between
- LessThan
- GreaterThan
- LessThanEqual
- GreaterThanEqual
- Like
- OrderBy

- OrderBy_\\\\Desc
- OrderBy_\\\\ASC

```
interface PersonRepository extends Repository<Person, Long> {
    List<Person> findByAddress(String address);
    Stream<Person> findByName(String name);
    Stream<Person> findByNameOrderByNameAsc(String name);
    Optional<Person> findByNickname(String nickname);
    void deleteByNickName(String nickname);
}
```

Using these keywords, Mapping will create the queries.

4.4.2. Using Repository as asynchronous way

The RepositoryAsync interface works similarly as Repository but with asynchronous work.

```
@Inject
@Database(DatabaseType.DOCUMENT)
private PersonRepositoryAsync documentRepositoryAsync;

@Inject
@Database(DatabaseType.COLUMN)
private PersonRepositoryAsync columnRepositoryAsync;
```

In other words, just inject and then create an Entity Manager async with producers method.

```
PersonRepositoryAsync repositoryAsync = //instance

Person person = new Person();
person.setNickname("diana");
person.setName("Diana Goodness");

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

repositoryAsync.save(person);
repositoryAsync.save(people);
```

Also, delete and retrieve information with a callback.

```
interface PersonRepositoryAsync extends RepositoryAsync<Person, Long> {

    void findByNickname(String nickname, Consumer<List<Person>> callback);

    void deleteByNickName(String nickname);

    void deleteByNickName(String nickname, Consumer<Void> callback);
}
```



In the key-value resource, the **Repository** does not support method query resource; this database type has key oriented operations.

4.4.3. Using Query annotation

The Repository interface contains all the trivial methods shared among the NoSQL implementations that a developer does not need to care. Also, there is query method that does a query based on the method name. Equally important there are two new annotations: the Query and param that defines the statement and set the values in the query respectively.

```
public interface PersonRepository extends Repository<Person, Long> {
    @Query("select * from Person")
    Optional<Person> findByQuery();

    @Query("select * from Person where id = @id")
    Optional<Person> findByQuery(@Param("id") String id);
}
```



Remember, when a developer defines who that repository will be implemented from the CDI qualifier, the query will be executed to that defined type, given that, gremlin to Graph, JNoSQL key to key-value and so on.

4.4.4. How to Create Repository and RepositoryAsync implementation programmatically

The Mapping API has support to create Repository programmatically to each NoSQL type, so there are **ColumnRepositoryProducer**, **DocumentRepositoryProducer**, **KeyValueRepositoryProducer**, **GraphRepositoryProducer** to column, document, key-value, graph repository implementation respectively. Each producer needs both the repository class and the manager instance to return a repository instance. The **ColumnRepositoryAsyncProducer** and **DocumentRepositoryAsyncProducer** have a method to create a RepositoryAsync instance that needs both an interface that extends RepositoryAsync and the manager async.

Graph repository producer

```
@Inject
private GraphRepositoryProducer producer;

public void anyMethod() {
    Graph graph = ...;//instance
    PersonRepository personRepository = producer.get(PersonRepository.class, graph);
}
```

Key-value repository producer

```
@Inject
private KeyValueRepositoryProducer producer;

public void anyMethod() {
    BucketManager manager = ...;//instance
    PersonRepository personRepository = producer.get(PersonRepository.class, manager);
}
```

Column repository producer

```
@Inject
private ColumnRepositoryProducer producer;

@Inject
private ColumnRepositoryAsyncProducer producerAsync;

public void anyMethod() {
    DocumentCollectionManager manager = ...;//instance
    DocumentCollectionManagerAsync managerAsync = ...;//instance
    PersonRepository personRepository = producer.get(PersonRepository.class, graph);
    PersonRepositoryAsync personRepositoryAsync = producerAsync.get
(PersonRepositoryAsync.class, graph);
}
```

```
@Inject
private DocumentRepositoryProducer producer;

@Inject
private DocumentRepositoryAsyncProducer producerAsync;

public void anyMethod() {
    DocumentCollectionManager manager = ...;//instance
    DocumentCollectionManagerAsync managerAsync = ...;//instance
    PersonRepository personRepository = producer.get(PersonRepository.class, graph);
    PersonRepositoryAsync personRepositoryAsync =
producerAsync.get(PersonRepositoryAsync.class, graph);
}
```

```
@Inject
private ColumnRepositoryProducer producer;

@Inject
private ColumnRepositoryAsyncProducer producerAsync;

public void anyMethod() {
    ColumnFamilyManager manager = ...;//instance
    ColumnFamilyManagerAsync managerAsync = ...;//instance
    PersonRepository personRepository = producer.get(PersonRepository.class, manager);
    PersonRepositoryAsync personRepositoryAsync = producerAsync.get
(PersonRepositoryAsync.class, managerAsync);
}
```

```
@Inject
private DocumentRepositoryProducer producer;

@Inject
private DocumentRepositoryAsyncProducer producerAsync;

public void anyMethod() {
    DocumentCollectionManager manager = ...;//instance
    DocumentCollectionManagerAsync managerAsync = ...;//instance
    PersonRepository personRepository = producer.get(PersonRepository.class, manager);
    PersonRepositoryAsync personRepositoryAsync = producerAsync.get
(PersonRepositoryAsync.class, managerAsync);
}
```

4.5. Bean Validation

The Mapping has support to use [Bean Validation](#) (BV), which supports a plugin that, basically,

listens to an event from **preEntity** and executes the BV.

```
@Entity
public class Person {

    @Key
    @NotNull
    @Column
    private String name;

    @Min(21)
    @NotNull
    @Column
    private Integer age;

    @DecimalMax("100")
    @NotNull
    @Column
    private BigDecimal salary;

    @Size(min = 1, max = 3)
    @NotNull
    @Column
    private List<String> phones;
}
```

In case of a validation problem in the project, a **ConstraintViolationException** will be thrown.

```
Person person = Person.builder()
    .withAge(10)
    .withName("Ada")
    .withSalary(BigDecimal.ONE)
    .withPhones(singletonList("123131231"))
    .build();
repository.save(person); //throws a ConstraintViolationException
```

Chapter 5. References

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>