Jakarta NoSQL

Version 1.0.0-b4, June 04, 2022: Draft

Table of Contents

Specification: Jakarta NoSQL

Version: 1.0.0-b4

Status: Draft

Release: June 04, 2022

Copyright (c) 2020 Jakarta NoSQL Contributors:
Eclipse Foundation, Otavio Santana, Leonardo de Moura Rocha Lima, Roan Brasil Monteiro

This program and the accompanying materials are made available under the
terms of the Eclipse Public License v. 2.0 which is available at
http://www.eclipse.org/legal/epl-2.0.

Jakarta NoSQL

Chapter 1. One Mapping API, multiple databases

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

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

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

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

Jakarta NoSQL defines an API for each NoSQL database type. However, it uses the same annotations to map Java objects. Therefore, with just these annotations that look like JPA, there is support for more than twenty NoSQL databases.

```
@Entity
public class Deity {

    @Id
    private String id;

    @Column
    private String name;

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

Vendor lock-in is one of the things any Java project needs to consider when choosing NoSQL databases. If there is a need to switch out a database, other considerations include: time spent on the change; the learning curve of a new database API; the code that will be lost; the persistence layer that needs to be replaced, etc. Jakarta NoSQL avoids most of these issues through the APIs of **Communication** and **Mapping** Layers.

Jakarta NoSQL also has template classes that apply the design pattern 'template method' to databases operations. And the `Repository` interface allows Java developers to create and extend interfaces, with implementation automatically provided by a Jakarta NoSQL implementation:

support method queries built by developers will automatically be implemented for them.

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

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

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

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

DeityRepository repository = service.getDeityRepository();

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

repository.save(diana);

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

1.1. Beyond JPA

JPA is a good API for object-relational mapping and has established itself as a standard in the Java world defined in JSRs. It would be ideal to use the same API for both SQL and NoSQL, but there are behaviors in NoSQL that SQL does not cover, such as time to live and asynchronous operations. JPA was simply not designed to handle those features.

```
ColumnTemplate template = //instance;
Deity diana = Deity.builder()
        .withId("diana")
        .withName("Diana")
        .withPower("hunt")
        .build();
Duration ttl = Duration.ofSeconds(1);
template.insert(diana, ttl);
```

1.2. A Fluent API

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

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

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

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

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

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

template.delete(delete);
```

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

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

1.4. Particular Behavior Matters in NoSQL Databases

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

```
public interface PersonRepository extends CouchbaseRepository {
```

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

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

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

1.5. Key Features

- Simple APIs supporting all well-known NoSQL storage types - Column Family, Key-Value Pair, Graph and Document databases.
- Use of Convention Over Configuration
- Easy-to-implement API Specification and Technology Compatibility Kit (TCK) for NoSQL Vendors
- The APIs focus is on simplicity and ease of use. Developers should only have to know a minimal set of artifacts to work with Jakarta NoSQL. The API is built on Java 8 features like Lambdas and Streams, and therefore fits perfectly with the functional features of Java 8+.

Chapter 2. Let's Talk About Standard to NoSQL Databases in Java

NoSQL databases provide mechanisms for storage and retrieval of unstructured data (non-relational) in stark contrast of the tabular relations used in relational databases. NoSQL databases, compared to relational databases, have better performance and high scalability. They are becoming more popular in several industry verticals, such as finance and streaming. As a result of this increased usage, the number of users and database vendors are increasing.

A NoSQL database is basically defined by a model of storage. There are four types:

2.1. Key-Value

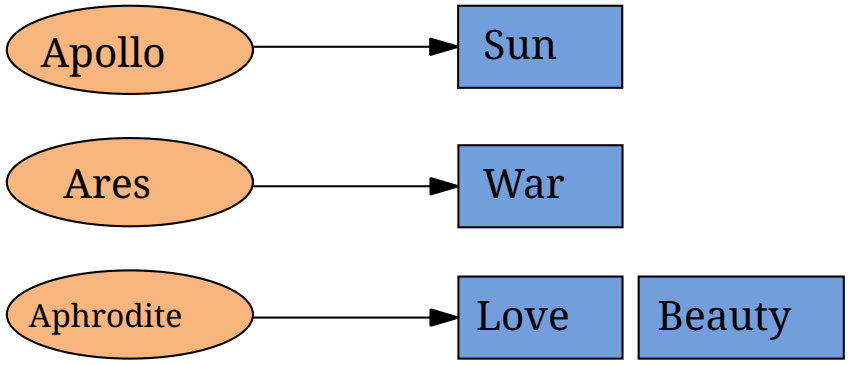


Figure 1. Key-Value structure

This database type has a structure that looks like a java.util.Map API, values are mapped to keys.

Examples:

- Amazon DynamoDB
- Redis
- Scalaris
- Voldemort

Table 1. Key-Value vs Relational structure

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

2.2. Document

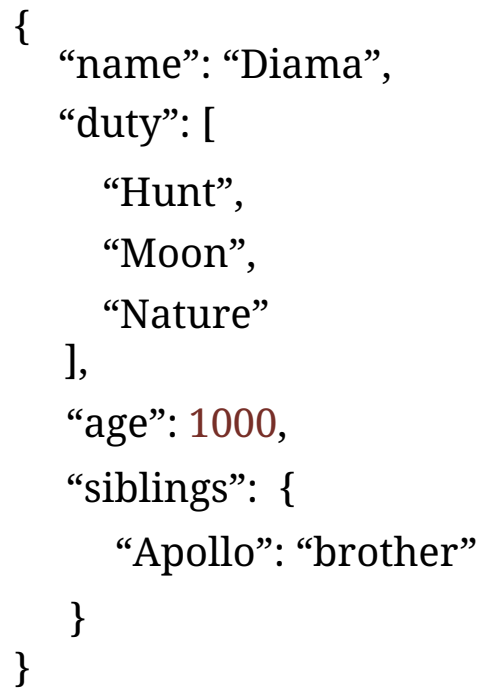


Figure 2. Document Structure

This model can store documents without a predefined structure. A document may be composed of numerous fields with different kinds of data, including a document inside another document. This model works either with XML or JSON file.

Examples:

- Amazon SimpleDB
- Apache CouchDB
- MongoDB

Table 2. Document vs Relational structure

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

2.3. Column Family

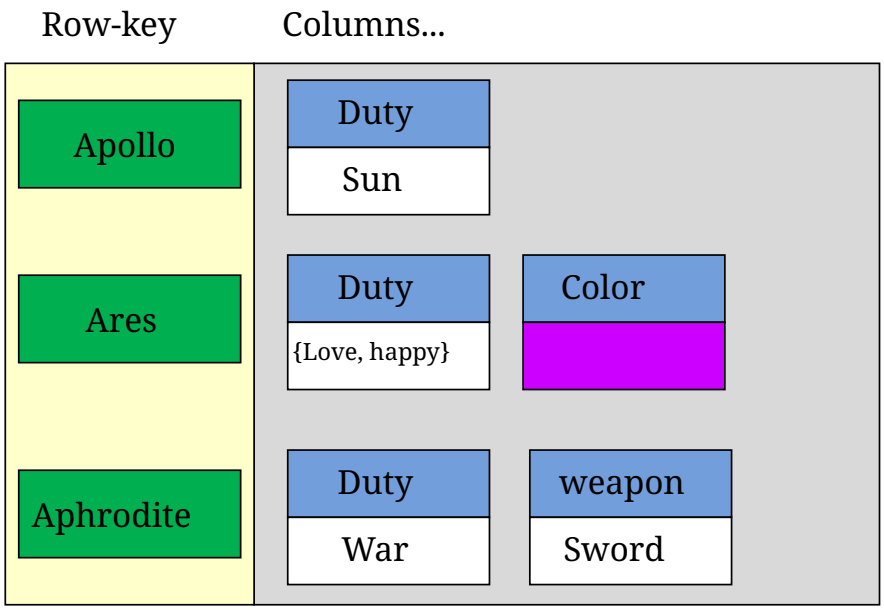


Figure 3. Column Family Structure

This model became popular with the Bigtable paper by Google, with the goal of being a distributed storage system for structured data, projected to have either high scalability or volume.

Examples:

- HBase
- Cassandra
- Scylla
- Cloud Data
- SimpleDB

Table 3. Column Family vs Relational structure

Relational structure	**Column Family structure**
Table	Column Family
Row	Column
Column	Key/value pair
Relationship	----

2.4. Graph

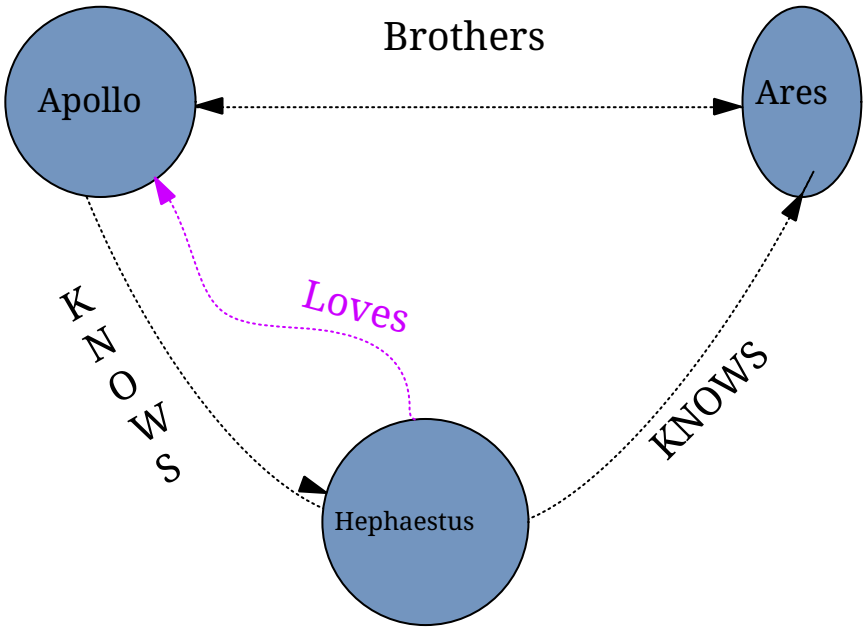


Figure 4. Graph Structure

A graph database uses graph structures for semantic queries with nodes, edges and properties to represent and store data.

- **Vertex**: A node in the graph. It stores data like the table in SQL or a Document in a Document database;
- **Edge**: An element that establishes the relationship between vertices;
- **Property**: A key-value pair that defines an edge's properties.

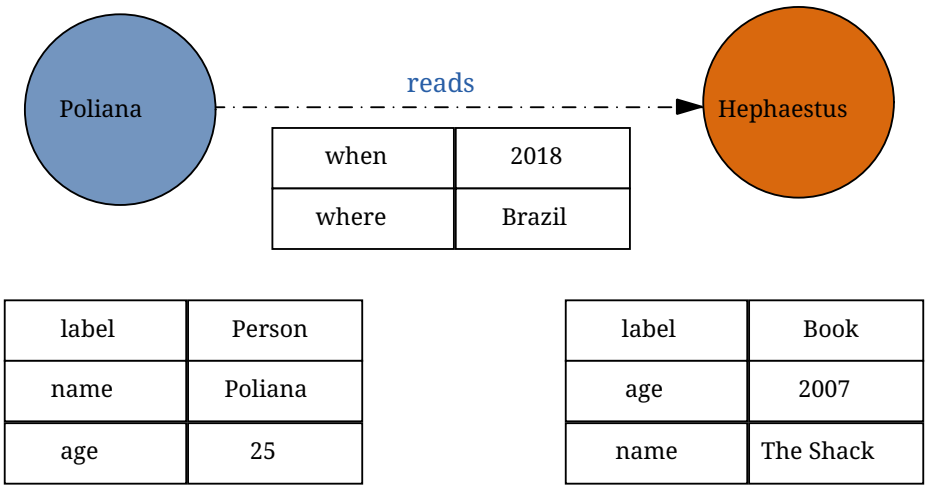


Figure 5. Graph with Vertex, Edge and Properties

The graph direction is an important concept in a graph structure. For example, you can know a person despite this person not knowing you. This is stored in the relationship (edge) direction of the graph.

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

2.5. Multi-Model Database

Some databases have support for more than one kind of model storage. This is the multi-model database.

Examples:

- OrientDB
- Couchbase

2.6. Scalability vs Complexity

Every database type has specific persistence structures to solve particular problems. There is a balance regarding model complexity; more complicated models are less scalable. For example, as shown in Figure 6, a key-value NoSQL database is more scalable simple complexity because all queries and operations are key-based.

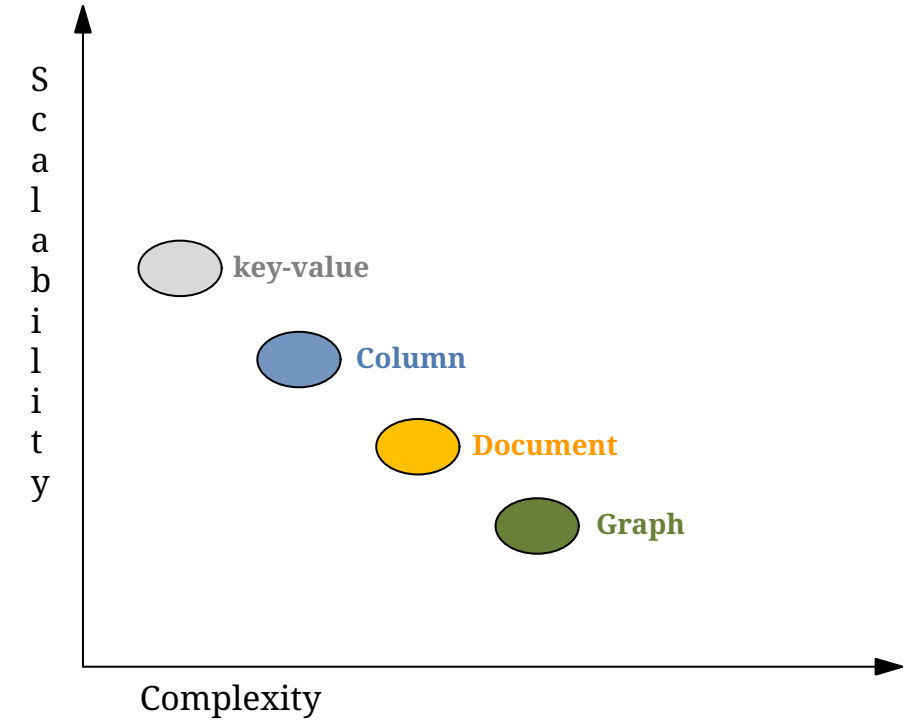


Figure 6. Scalability vs. Complexity

2.7. BASE vs ACID

Figure 7. BASE vs. ACID

Key characteristics of relational persistence technologies are defined as: Atomicity, Consistency, Isolation, Durability (ACID):

- **Atomicity**: Either all transaction operations complete, or none will.
- **Consistency**: The database is in a consistent state when a transaction begins and ends.
- **Isolation**: A transaction will behave as if it is the only operation being performed on the database.
- **Durability**: Upon completion of a transaction, a operation will not be reversed.

In the NoSQL world, the key characteristics are defined as: Basic Availability, Soft-State and Eventual consistency (BASE):

- **Basic Availability**: The database appears to work most of the time.

- **Soft-state**: Data stores don’t have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
- **Eventual consistency**: Data stores exhibit consistency at some point later (e.g., lazily at read time).

2.8. CAP Theorem

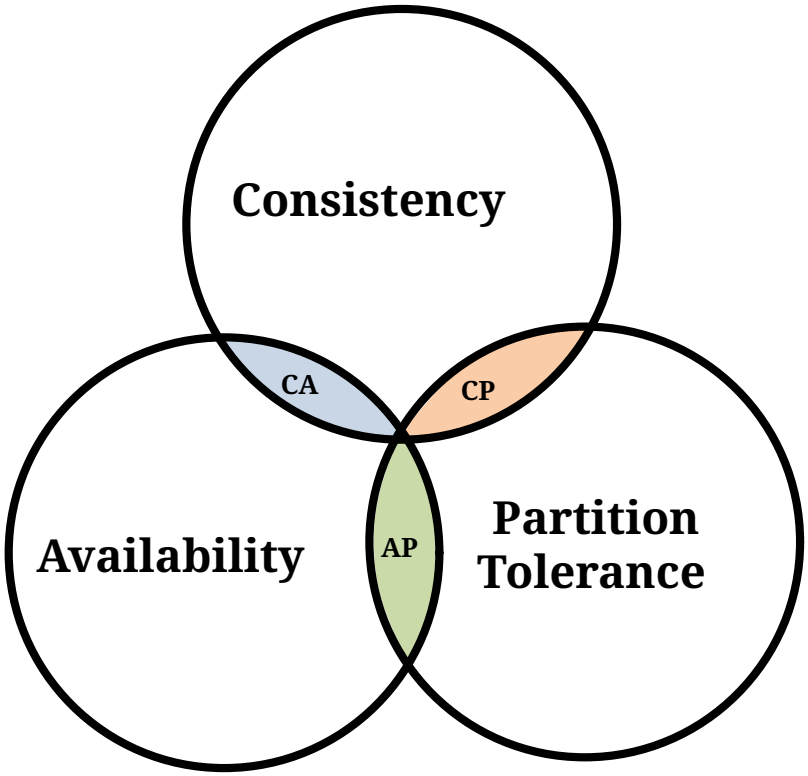


Figure 8. CAP Theorem

The CAP theorem is applied to distributed systems that store state. Eric Brewer, at the 2000 Symposium on Principles of Distributed Computing (PODC), conjectured that in any networked shared-data system, there is a fundamental trade-off between Consistency, Availability, and Partition Tolerance. In 2002, Seth Gilbert and Nancy Lynch of MIT published a formal proof of Brewer’s conjecture. The theorem states that networked shared-data systems can only guarantee/strongly support two of the following three properties at the same time:

- **Consistency**: A guarantee that every node in a distributed cluster returns the same, most recent, successful write. Consistency refers to every client having the same view of the data. There are various types of consistency models. Consistency in CAP (used to prove the theorem) refers to linearizability or sequential consistency - a very strong form of consistency.
- **Availability**: Every non-failing node returns a response for all read and write requests in a reasonable amount of time. The key word here is "every". To be available, every node (on either side of a network partition) must be able to respond in a reasonable amount of time.
- **Partition Tolerance**: The system continues to function and uphold its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

2.9. The Diversity in NoSQL

There are approximately 225 NoSQL databases (at time of writing). These databases usually support one or more types of structures. They also have specific behavior. Particular features make developer's experience more comfortable in different ways, such as Cassandra Query language in Cassandra databases, a search engine in Elasticsearch, live query in OrientDB, N1QL in Couchbase, and so on. Such aspects matter with NoSQL databases.

2.10. Standard in SQL

Java applications that use relational databases have, as a good practice, a layer between business logic and data. This is known as Data Access Object (DAO). There are also APIs, such as JPA and JDBC providing advantages to developers:

- Avoid vendor lock-in. Using a standard (such as JDBC), a database has less impact and is easier to implement - because we just need to change a simple driver.
- There is no need to learn a new API to work with a new database - that's implemented in to the driver.
- There is less code change when changing to a new vendor. There may be some code changes, but not all code that communicates with the database is lost.

Currently, there are no NoSQL standards for Java. This causes a Java developer to:

- Be locked-in to a vendor
- Learn a new API every time an application needs to use a new database. Every database vendor change has a high impact because a rewrite of the communication layer is required. This happens even when changing to a new database that is the same kind of NoSQL database.

There are initiatives to create NoSQL APIs, such as Spring Data, Hibernate ORM, and TopLink. JPA is a popular API in the Java world, which is why all these initiatives try to use it. However, this API is created for SQL and not for NoSQL and, as such, doesn't support all behaviors in NoSQL databases. Many NoSQL databases have no transaction concept, and many NoSQL database also don't support to asynchronous insertion.

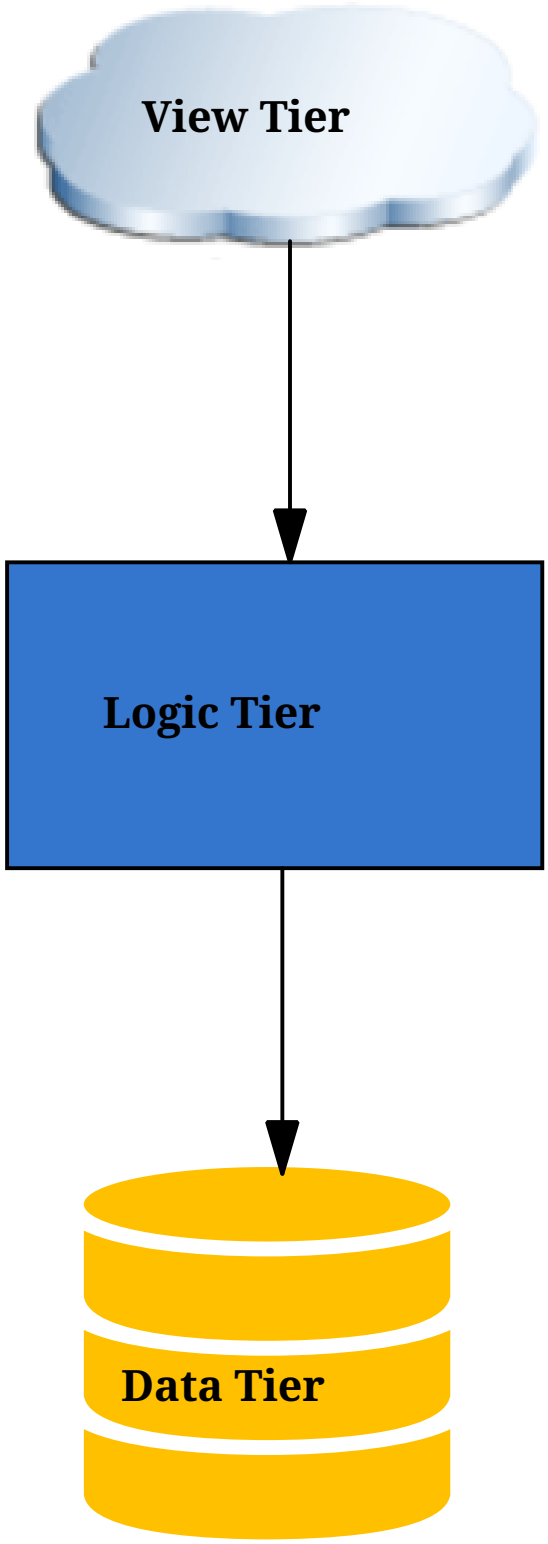
The solution, in this case, is to create a specification that covers the four types of NoSQL database as described earlier in this chapter Each database type has specific structures that must be recognized. This new API should resemble JPA because of its popularity amongst Java developers. It should also be extensible, to support cases when a database has more than one particular behavior.

Chapter 3. The Strategy Behind Jakarta NoSQL

The divide-and-conquer strategy decreases the complexity of systems within modules or structures. These structure levels split responsibility and make maintenance and replaceability more clear. The new Jakarta NoSQL API proposal will serve as a bridge between the logic tier and the data tier. To do this, we need to create two APIs: one to communicate to a database and another one to be a high abstraction to the Java application.

In software, there are structures: tiers, physical structures, and layers. The multi-tier application has three levels:

- **Presentation tier**: Has a primary responsibility to translate results so the user may understand.
- **Logic tier**: Has all business rules, processes, conditions, saved information, etc. This level moves and processes information among other levels.
- **Data tier**: Retrieves and stores information in either a database or a system file.

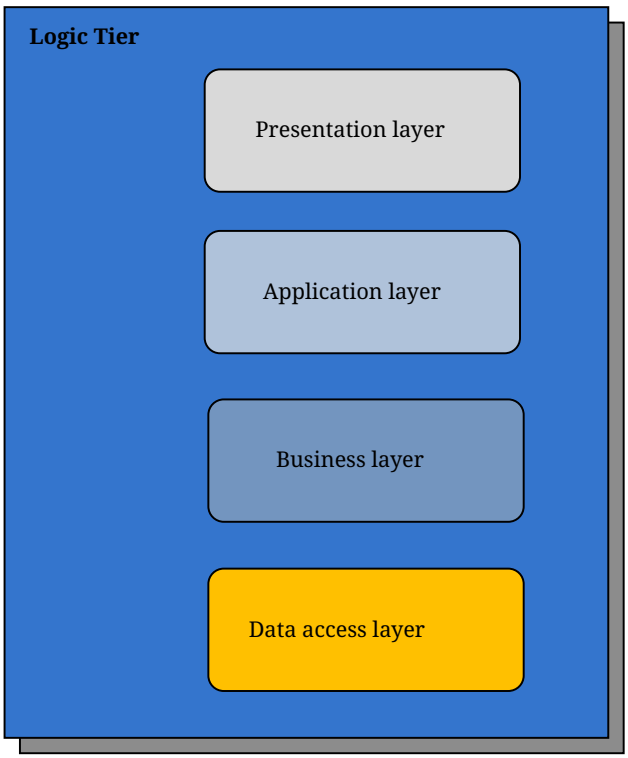


Talking more precisely about the physical layer and the logic to separate responsibilities, there are other layers.

The logic tier, where the application and the business rules reside, has additional layers:

- **Application layer**: The bridge between the view tier and logic tier, e.g., convert an object into either JSON or HTML.

- **Service layer**: May be either a Controller or a Resource.
- **Business Layer**: The part of the program that encodes the real-world business or domain rules that determine how data will be created, stored, and changed.
- **Persistence Layer**: Provides simplified access to data stored in some kind of persistent storage.



A persistence layer has its own layer: A Data Access Object (DAO). This structure connects the business layer and persistence layer. The DAO contains an API that supports databases. Currently, there is a difference between SQL and NoSQL databases:

In a relational database, there are two mechanisms under DAO:

- **Java Database Connectivity (JDBC)**: a deep layer with a database that has communications, underlying transactions, and is basically a driver to a particular database.
- **Java Persistence API (JPA)**: A higher layer that has communication with either JDBC or JPA. This layer has a high mapping to Java containing annotations and an EntityManager. In general, JPA has integrations with other specifications such as Jakarta Contexts and Dependency Injection (CDI) and Jakarta Bean Validation.

A considerable advantage of this strategy is that one change, either through JDBC or JPA, may happen quickly. When you change a database, you need to supersede to the respective database driver, and you're done! The code is ready for the new database.

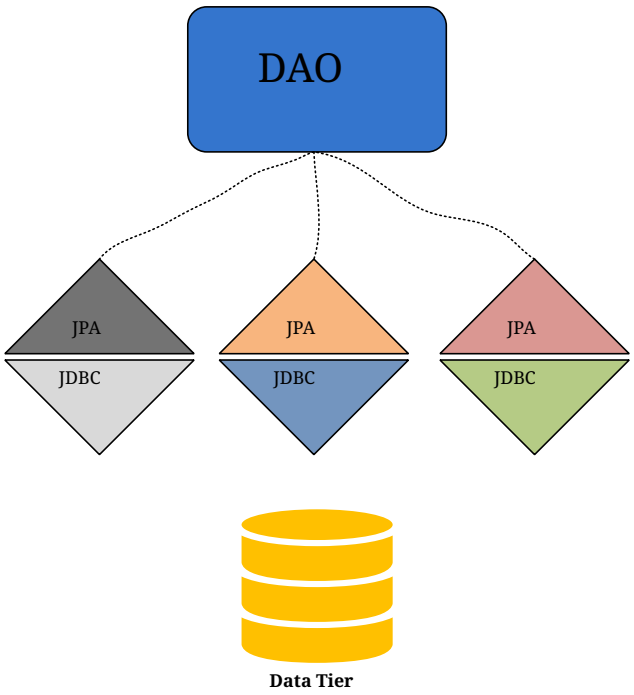


Figure 9. A SQL Java application with JPA layer architecture

In a NoSQL database, there isn't a strategy to save code and there is little impact for change. All APIs are different and don't follow any kind of standard, so a change to a new database may result in a significant amount of work.

- The database vendor needs to be concerned about the high-level mapping to Java world, and the solution provider needs to be concerned about the low level of communication with a particular database.
- The database vendor needs to "copy" these 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 loses code. If a developer uses high-level mapping, they lock-in a Java solution because if this high level doesn't have the support to a particular NoSQL database, the developer needs to change to either a Java solution or directly use a NoSQL API.

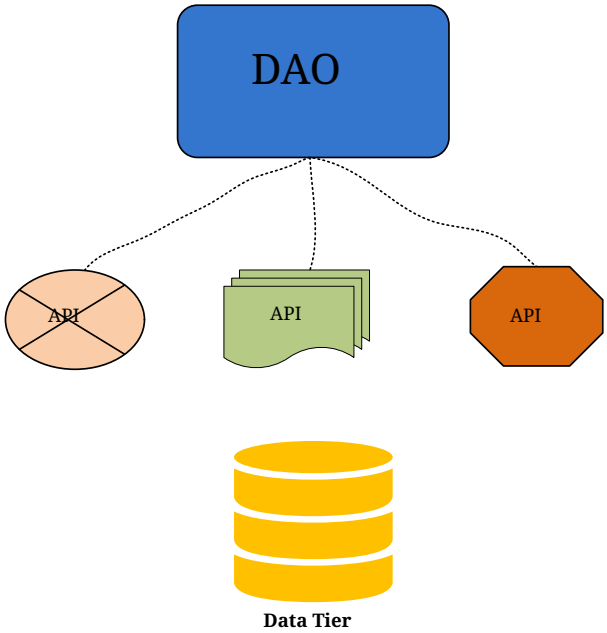


Figure 10. A NoSQL Java application that has lock-in to each NoSQL provider

A wise recommendation might be to use JPA because once the developer already knows this standard SQL API, they can use the same API for a relational database and apply it to a NoSQL database. However, using an API with SQL concepts in NoSQL world is the same as using a knife as a spoon. The result would be a disaster! Furthermore, the NoSQL world has diversity with several data structures and particular behavior to each provider, and both matter in a software solution. Indeed, the merge strategy to use just one API is still a topic of discussion.

A good point about using NoSQL as a consequence polyglot persistence is that data storage is about choice. When a database offers gains, it sacrifices other aspects that would violate the CAP theorem. Hence, an API generic enough to encapsulate all kinds of databases might be useless.

The history between Java and NoSQL has several solutions that may be separated into two categories:

1. NoSQL Drivers
2. Mapper
    - Mapper Agnostic
    - Mapper Specific

The first one is the driver API that has a low communication level such as JDBC to NoSQL. It guarantees full power over the NoSQL database, a semantic closer to a database. However, it requires more code to move it forward to the entity domain and limits portability. Therefore, there is a learning curve.

The Object Mapper allows the developer work in terms of domains, thus it can help a developer follow ethical practices. A mapper may be **specific** which means that a mapper is designed for a particular database. This mapper will support all database features but with the price of vendor lock-in. On the other hand, a mapper may be **agnostic** such that it uses a generic API to encapsulate the database API. This allows a developer to connect several databases. However, it tends to either

not cover numerous features in a database or many other databases.

The rapid adoption of NoSQL combined with the vast assortment of implementations has driven a desire to create a set of standardized APIs. In the Java world, this was initially proposed in an effort by Oracle to define a NoSQL API for Java EE 9. The justification for the definition of a new API, separate from JDBC and JPA, was the following:

- JPA was not designed with NoSQL in mind
- A single set of APIs or annotations isn't adequate for all database types
- JPA over NoSQL implies the inconsistent use of annotations
- The diversity in the NoSQL world matters

Unfortunately, what Oracle proposed for Java EE 9 never came to fruition after Java EE was donated to the Eclipse Foundation.

To bring innovation under the Jakarta EE umbrella, Jakarta NoSQL was born. The goal of this specification is to ease integration between Java applications and NoSQL databases with a standard API to work with different types and vendors of NoSQL databases. To achieve this, the specification has two APIs that work like layers and each layer has a specific goal that can integrate between each and use in isolation:

- **Communication API**: This is layer is analogous to JDBC and SQL. This API has four specializations, one for each database type (column family, document, key-value and graph). The specialties are independent of each other, optional from the point of the database vendor and have their specific TCKs.
- **Mapping API**: This layer is analogous to JPA and CDI. It is based on Jakarta Annotations and preserves integration with other Jakarta EE technologies such as Jakarta Bean Validation and others.

Jakarta NoSQL is the first specification in the Java enterprise. As with any Java specification, it analyzes solutions that already exist, checks the history with both success and failure cases, and then goes in a direction that has a lesser number of trade-offs within an API architecture. The divide-and-conquer method fits well with the layer, communication, mapping, and NoSQL types. Thus, it will provide a straightforward specification with light maintenance. It will define the scope of each API, and will work better in extensibility once the particular features matter to a NoSQL database. CDI create and add new functionality without changing the core code in conjunction with bean validation that listen for those events.

Jakarta EE has a bright future with a significant integration within the Java community and open source. More transparency, after all, is the most meaningful power of Jakarta EE. It's not the technology itself, but the heart of the community, therefore, the success is in the hand of every developer.

Chapter 4. Introduction to the Communication API

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

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

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

The package name might change on the Jakarta EE process.

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

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

- Configuration
- Factory
- Manager
- Entity
- Value

4.1. The API Structure

The communication has four projects:

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

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

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

4.2. Value

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

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

The `Value` interface has the methods:

- `Object get();` Returns the value as type `Object`
- `<T> T get(Class<T> clazz);` Initiates the conversion process to the required type in a safe manner. If the required type doesn't have support, it will throw an exception. However, the API allows creating custom converters.
- `<T> T get(TypeSupplier<T> typeSupplier);` Similar to the previous method, it initiates the conversion process 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();
```

4.2.1. Create Custom Writer and Reader

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

- `ValueWriter`: This interface represents an instance of `Value` to write in a database.

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

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

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

public class Money {

    private final Currency currency;

    private final BigDecimal value;

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

    public Currency getCurrency() {
        return currency;
    }

    public BigDecimal getValue() {
        return value;
    }

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

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

    @Override
    public String toString() {
```

```
        return currency.getCurrencyCode() + ' ' + value;
    }

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

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

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

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

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

```
import jakarta.nosql.ValueWriter;

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 type `T` from an instace of `Object`.

```
import jakarta.nosql.ValueReader;

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

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

Each file will have the qualifier of its respective implementation:

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

```
my.company.MoneyValueReader
```

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

```
my.company.MoneyValueWriter
```

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

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

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

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

4.3. Element Entity

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

as an implementation of `Value`.

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

4.3.1. 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 have a nested document, that is, a sub-document.

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

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

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

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

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

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

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

4.3.2. 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 of a column.

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

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

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

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

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

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

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

4.4. Entity

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

- ColumnEntity
- DocumentEntity
- KeyValueEntity

4.4.1. ColumnFamilyEntity

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

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

4.4.2. DocumentEntity

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

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

String name = entity.getName();

entity.add(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");
```

4.4.3. KeyValueEntity

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

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

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

String key = entity.getKey();

Value value = entity.getValue();

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

4.5. Manager

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

- **DocumentCollectionManager**
- **ColumnConfiguration**
- **BucketManager**

4.5.1. Document Manager

- **DocumentCollectionManager**: To perform synchronous 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 two hours of TTL

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

// Update operations
manager.update(entity);

manager.update(entities);
```

Search information on a document collection

The Document Communication API supports retrieving information from a `DocumentQuery` instance. This class has six attributes:

- **documents**: The fields to return in a query.
- **limit**: the maximum number of results in a query.
- **skip**: the position of the first result.
- **documentCollection**: the document collection name looks like a table in a document NoSQL database.
- **condition**: the filter in the query.
- **sorts**: the way to order the information, where the first element will have more precedence than the next ones.

By default, there are two ways to create a `DocumentQuery` instance that are available as a static

method in the same class:

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

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

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

- Select all fields from the document collection Person.

Using the select method:

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

Using the builder method:

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

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

Using the select method:

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

Using the builder method:

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

```
                                .build();
```

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

Using the select method:

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

Using the builder method:

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

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

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

Using the select method:

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

Using the builder method:

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

//static imports

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

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

Using the select method:

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

Using the builder method:

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

//static imports

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

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

Using the select method:

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

Using the builder method:

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

//static imports

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

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

Using the select method:

```
DocumentQuery query = DocumentQuery.select("name", "age")
            .from("Person")
```

```
            .where("name").eq("Ada Lovelace")
            .or("age").gt(20)
            .orderBy("name").asc()
            .orderBy("desc").desc()
            .build();

//static imports

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

Using the builder method:

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

//static imports

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

Removing information from Document Collection

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

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

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

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

```
DocumentDeleteQuery query = DocumentQueryBuilder.builder().from("collection")
                                                 .where(DocumentCondition.gt("age", 10
).build();
```

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

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

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

4.5.2. Column Manager

- **ColumnFamilyManager**: To perform synchronous operations.

ColumnFamilyManager

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

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

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

entity.add(diana);
```

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

```
// Insert operations
manager.insert(entity);

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

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

// Update operations
manager.update(entity);

manager.update(entities);
```

The Column Communication API supports retrieving information from a `ColumnQuery` instance. This class has six attributes:

- **columns**: The fields to return in a query.

- **limit**: the maximum number of results in a query.
- **skip**: the position of the first result.
- **columnFamily**: the column family name looks like a table in a column-family NoSQL database.
- **condition**: the filter in the query.
- **sorts**: the way to order the information, where the first element will have more precedence than the next ones.

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

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

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

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

- Select all fields from the column family Person.

Using the select method:

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

Using the builder method:

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

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

Using the select method:

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

Using the builder method:

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

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

Using the select method:

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

Using the builder method:

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

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

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

Using the select method:

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

```
                .where("name").eq("Ada Lovelace")
                .and("age").gt(20)
                .build();
```

Using the builder method:

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

//static imports

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

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

Using the select method:

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

Using the builder method:

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

//static imports
```

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

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

Using the select method:

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

Using the builder method:

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

//static imports

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

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

Using the select method:

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

Using the builder method:

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

//static imports

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

Removing information from Column Family

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

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

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

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

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

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

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

4.5.3. BucketManager

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

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

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

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

bucketManager.put(entity);

bucketManager.put(entities);

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

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

Remove 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"));
```

4.5.4. Querying by Text with the Communication API

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

Queries follow these rules:

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

Key-Value Database Types

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

`get`

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

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

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

`del`

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

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

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

`put`

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

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

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

Column-Family and Document Database Types

The queries have syntax similar to SQL queries. But keep in mind that it has a limitation: joins are

not supported.

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

insert

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

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

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

update

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

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

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

delete

Use the `delete` statement to remove fields or entities

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

select

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

```
select_statement ::=  select ( SELECT_CLAUSE | '*' )
                      from ENTITY_NAME
                      [ where WHERE_CLAUSE ]
```

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

`where`

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

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

Conditions

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

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

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

Operator	Description
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'll go to be used, with an operator, in a field target.

There are six types:

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

`skip`

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

`limit`

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

`order by`

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

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

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`

`PreparedStatement` and `PreparedStatementAsync`

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

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

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

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

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

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

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

preparedStatement.getResult(callback);
```

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

4.6. Factory

The factory class creates the **Managers**.

- **ColumnFamilyManagerFactory**
- **BucketManagerFactory**
- **DocumentCollectionManagerFactory**

4.6.1. Column Family Manager Factory

The factory classes have the duty to create the Column Family manager.

- **ColumnFamilyManagerFactory**

```
ColumnFamilyManagerFactory factory = //instance;
```

```
ColumnFamilyManager manager = factory.get("database");
```

4.6.2. Document Collection Factory

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

- **DocumentCollectionManagerFactory**

```
DocumentCollectionManagerFactory factory = //instance;
DocumentCollectionManager manager = factory.get("database");
```

4.6.3. Bucket Manager Factory

The factory classes have the responsibility to create the `BucketManager`.

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

Beyond the BucketManager, some databases have support for particular structures represented in the Java world such as `List`, `Set`, `Queue` and `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 a `java.lang.UnsupportedOperationException` if the database does not support any of the structures.

4.7. Configuration

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

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

4.7.1. Settings

The `Settings` interface represents the settings used in a configuration. It extends looks like a

`Map<String, Object>`; for this reason, gives a key that can set any value as configuration.

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

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

Each property unit has a tuple where the key is the name and the value is the property configuration. Each NoSQL database has its configuration properties. Also, some standard configurations might be used on NoSQL databases:

- `jakarta.nosql.user`: to define a user in a NoSQL database
- `jakarta.nosql.password`: to define a password in a database
- `jakarta.nosql.host`: the host configuration that might have more than one with a number as a suffix, such as `jakarta.nosql.host-1=localhost` or `jakarta.nosql.host-2=host2`

To read the property information, it will follow the same principal and priority from Eclipse MicroProfile Configuration and Configuration Spec JSR 382. Therefore, it will read from the {@link `System#getProperties()`,`System#getenv()` and `Settings`.

4.7.2. Document Configuration

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

```
DocumentConfiguration configuration = //instance;
DocumentCollectionManagerFactory managerFactory = configuration.get();
```

4.7.3. Column Configuration

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

```
ColumnConfiguration configuration = //instance;

ColumnFamilyManagerFactory managerFactory = configuration.get();
```

4.7.4. Key Value Configuration

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

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

Chapter 5. Introduction to the Mapping API

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

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

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

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

The package name might change on the Jakarta EE process.

5.1. The Mapping structure

The mapping API has five parts:

- The **persistence-core**: The mapping common project.
- 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-graph**: The mapping to Graph NoSQL database.

Each module works separately as a Communication API.

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

5.2. Models Annotation

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

- Annotation Models
- Qualifier annotation

5.2.1. Annotation Models

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

communication entity:

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

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

@Entity

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

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

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

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

```
@Entity
public class Person {

    @Id
    private Long id;

    @Column
    private String name;

    @Column
    private Address address;
}


@Entity
```

```
public class Address {

    @Column
    private String street;

    @Column
    private String city;
}
```

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

@Column

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

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

    @Column("personName")
    private String name;

    @Column
    private List<String> phones;

    // ignored
    private String address;
}
```

@MappedSuperclass

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

This means, that this annotation causes fields annotated with `@Column` in a parent class to be

persisted together with the child class' fields.

```
@Entity
public class Dog extends Animal {

    @Column
    private String name;
}

@MappedSuperclass
public class Animal {

    @Column
    private String race;

    @Column
    private Integer age;
}
```

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

@Id

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

```
@Entity
public class User implements Serializable {

    @Id
    private String userName;

    @Column
    private String name;

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

@Embeddable

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

```
@Entity
```

```
public class Book {

    @Column
    private String title;

    @Column
    private Author author;
}

@Embeddable
public class Author {

    @Column
    private String author;

    @Column
    private Integer age;
}
```

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

@Convert

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

```
@Entity
public class Employee {

    @Column
    private String name;

    @Column
    private Job job;

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

public class MoneyConverter implements AttributeConverter<MonetaryAmount, String> {

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

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

public class MonetaryAmount {
    private final String currency;

    private final BigDecimal value;

    public String toString() {
        // specific implementation
    }

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

Collections

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

The following collections are supported:

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

```
@Entity
public class Person {

    @Id
    private Long id;

    @Column
    private String name;

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

```
    @Column
    private List<Address> addresses;
}

@Embeddable
public class Address {

    @Column
    private String street;

    @Column
    private String city;
}
```

The above classes are mapped to:

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

5.2.2. @Database

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

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

```
@Inject
private DocumentRepository repositoryA;

@Inject
```

```
private DocumentRepository repositoryB;
```

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

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

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

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

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

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

5.3. Template classes

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

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

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.

5.3.1. DocumentTemplate

This template has the responsibility to serve as a bridge between the entity model and communication API to a document collection. It has two classes; `DocumentTemplate` and

`DocumentTemplateAsync` - one for the synchronous and the other for the asynchronous tasks.

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 remove and retrieve information from document collection, there are `DocumentQuery` and `DocumentDeleteQuery` classes.

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

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

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

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

```
@Entity
public class Person {

    @Id("native_id")
```

```
    private long id;

    @Column
    private String name;

    @Column
    private int age;
}
```

```
@Inject
private DocumentQueryMapperBuilder mapperBuilder;

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

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

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

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) Use 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) Use 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);
}
```

5.3.2. DocumentTemplateAsync

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

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

```
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 may be used.

```
Consumer<Stream<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 precede the field with the `@Inject` annotation.

```
@Inject
private DocumentTemplateAsync template;
```

The next step is to produce a `DocumentCollectionManagerAsync`:

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

To work with more than one document template, there are two approaches:

1) Use 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) Use 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);
}
```

5.3.3. ColumnTemplate

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

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

- **ColumnEntityConverter**: 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` classes, respectively. Also, the callback method may be used.

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

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

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

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

```
@Entity
public class Person {

    @Id("native_id")
    private long id;

    @Column
    private String name;

    @Column
    private int age;
}
```

```
@Inject
private ColumnQueryMapperBuilder mapperBuilder;

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

```
}
```

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

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

The next step is to produce a `ColumnFamilyManager`:

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

To work with more than one column template, there are two approaches:

1) Use 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) Use 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);
}
```

5.3.4. Key-Value Template

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

The `KeyValueTemplate` is the template for synchronous tasks. It has 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 = // instance;
User user = new User();
user.setNickname("ada");
user.setAge(10);
user.setName("Ada Lovelace");
List<User> users = Collections.singletonList(user);

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

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

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

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

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

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) Use 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) Use 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);
}
```

5.3.5. Graph template

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

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

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

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

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

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

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

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

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

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

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

The next step: make a `Graph` instance eligible to CDI, applying the producers method:

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

To work with more than one graph template, there are two approaches:

1) Use 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) Use 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);
}
```

5.3.6. Querying by Text with the Mapping API

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

Key-Value Database Types

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

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

Column-Family Database Types

The Column-Family database has a more complex structure; however, a search from the key is still recommended. For example, both Cassandra and HBase have a secondary index, yet, neither have a guarantee about performance, and they usually recommend having a second table whose row key is the "secondary index" and is only being used to find the row key needed for the actual table. Given a `Person` class as an entity, we would like to operate from the field ID, which is the entity from

the Entity.

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

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

Document Database Types

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

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

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

Graph Database Types

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

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

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

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

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

// Keep using gremlin for Graph databases
```

```
PreparedStatement prepare = graphTemplate().prepare("g.V().hasLabel(param)");

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

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

5.4. Repository

In addition to the `Template` class, the Mapping API includes a Repository interface that helps the `Entity` repository to save, update, delete and retrieve information. To use `Repository`, you 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 be used 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 the `@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 databases, 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, Jakarta NoSQL 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);
```

5.4.1. Query by Method

The Repository interface also includes 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
- In
- 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.

Special Parameters

In addition to the use of use the query method, the repository has support to a special instance at the parameters in a method:

- `Pagination`: This parameter enables the resource of pagination at a repository.
- `Sort`: It appends sort in the query dynamically if the query method has the `OderBy` keyword. This parameter will add the sort after the sort information from the method.
- `Sorts`: It is a group of a sort, therefore, it appends one or more sort dynamically.

```
   interface PersonRepository extends Repository<Person, Long> {

        List<Person> findAll(Pagination pagination);

        List<Person> findByName(String name, Sort sort);

        List<Person> findByAgeGreaterThan(Integer age, Sorts sorts);
    }
```

This resource allows pagination and a dynamical sort in a smooth way.

```
PersonRepository personRepository = //instance;

Sort sort = Sort.asc("name");

Sorts sorts = Sorts.sorts().asc("name").desc("age");

Pagination pagination = Pagination.page(1).size(10);

List<Person> all = personRepository.findAll(pagination);//findAll by pagination

List<Person> byName = personRepository.findByName("Ada", sort);//find by name order by
name asc
```

```
List<Person> byAgeGreaterThan = personRepository.findByAgeGreaterThan(22, sorts)
;//find age greater than 22 sort name asc then age desc
```

All these special instances must be at the end, thus after the parameters that will be used at a query.

5.4.2. Using the 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 a query method that does 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, Jakarta NoSQL key to key-value and so on.

5.4.3. How to Programmatically Create a Repository Implementation

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.

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;

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

Document repository producer

```
@Inject
private DocumentRepositoryProducer producer;

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

```
@Inject
private ColumnRepositoryProducer producer;

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

```
@Inject
private DocumentRepositoryProducer producer;

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

5.5. Pagination

Pagination is the process of separating the contents into discrete pages. Each page displays a list of entities from the database. The pagination allows retrieving a considerable number of elements

from datastore into small blocks, e.g., returns ten pages with one hundred elements instead of return one thousand in a big shot at the storage engine.

At this project, there an interface that represents the pagination by using the `Pagination` interface.

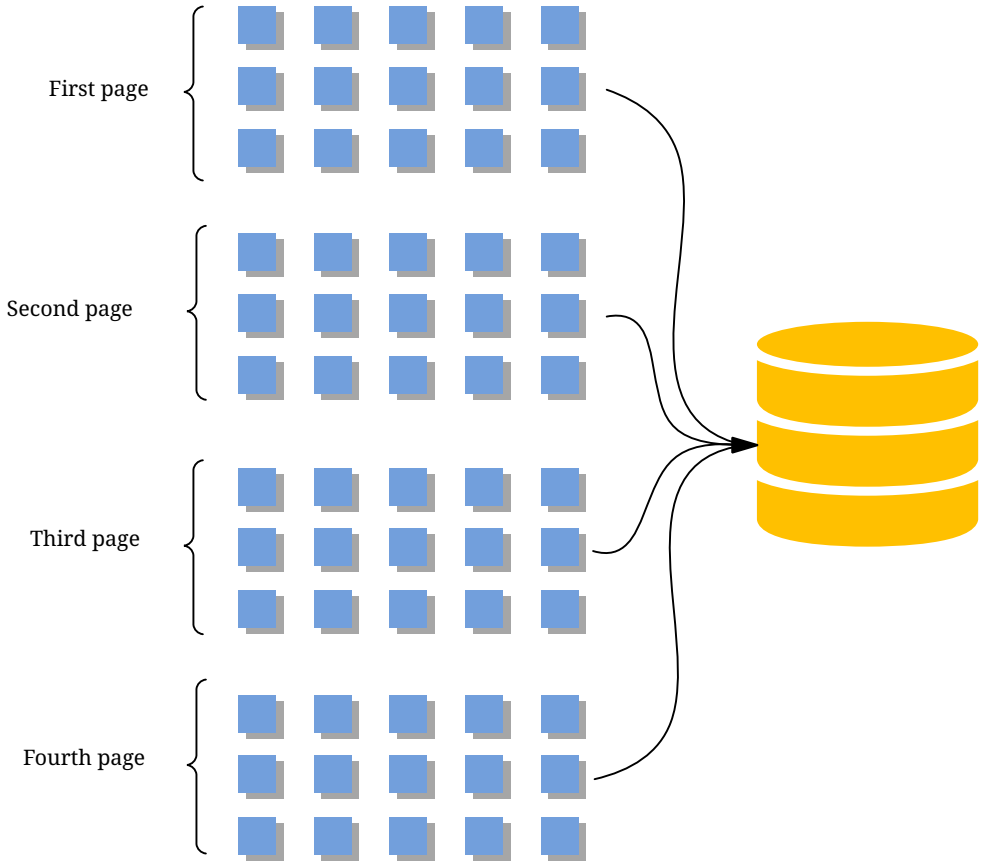


Figure 11. The pagination, instead of query a bunch of elements. The pagination process allows retrieving a small fixed block of entities in a database.

```
Pagination pagination = Pagination.page(1).size(2);

// this creates a pagination instance where it is the first page and each page has the
size of two each one.
long pageNumber = pagination.getPageNumber();

Pagination next = pagination.next();
```

The Column-Family, Document and Graph API has method such as **skip** and **limit** to jump values into a query and to define a maximum size of elements to return in a query respectively.

5.5.1. Column-Family

A `ColumnQueryPagination` is a specialization of `ColumnQuery` that allows the pagination resource at the query. Thus it overwrites the **skip** and **limit** and use the values from a `Pagination` instance.

```
Pagination pagination = //instance;
ColumnQuery query = //instance;

ColumnQueryPagination queryPagination = ColumnQueryPagination.of(query, pagination);

ColumnQueryPagination nextQueryPagination =
 queryPagination.next();
```

Template

Through the `Template` there are two ways to use the pagination resource. The first one is to define the value as `ColumnQuery`. Thus it will return a query as a list; however, it will break the results into pages.

```
ColumnTemplate template = //instance;
Pagination pagination = Pagination.page(1).size(1);
ColumnQuery query = ColumnQueryPagination.of(select().from("person").build(),
pagination);
Stream<Person> people = template.select(query);
```

The second one is representing the page with the `Page` instance. A page is a fixed-length contiguous block of entities from the database, it has the feature to generate the next page.

```
ColumnTemplate template = //instance;
Pagination pagination = Pagination.page(1).size(1);
ColumnQueryPagination query = ColumnQueryPagination.of(
        select().from("person").build(), pagination);
Page<Person> firstPage = template.select(query);
Stream<Person> firstPageContents = page.getContent();
Page<Person> secondPage = firstPage.next();
```

Query Mapper

From a mapper query is possible either creates a query that executes using the pagination or creates a `Page` instance.

```
ColumnQueryMapperBuilder mapperBuilder = //instance;

Pagination pagination = Pagination.page(2).size(2);
ColumnQuery query = mapperBuilder.selectFrom(Person.class).build(pagination);
Page<Person> page = mapperBuilder.selectFrom(Person.class).page(template, pagination);
Stream<Person> people = mapperBuilder.selectFrom(Person.class).getResult(template,
pagination);
```

Repository

A Repository interface also allows using the pagination feature at these interfaces. To enable it creates a `Pagination` parameter as the last parameter.

```
 interface PersonRepository extends Repository<Person, Long> {

    List<Person> findAll(Pagination pagination);

    Set<Person> findByName(String name, Pagination pagination);

    Page<Person> findByAge(Integer age, Pagination pagination);
}
```

5.5.2. Document

A `DocumentQueryPagination` is a specialization of `DocumentQuery` that allows the pagination resource at the query. Thus it overwrites the **skip** and **limit** and use the values from a `Pagination` instance.

```
Pagination pagination = //instance;
DocumentQuery query = //instance;

DocumentQueryPagination queryPagination = DocumentQueryPagination.of(query,
pagination);

DocumentQueryPagination nextQueryPagination = queryPagination.next();
```

Template

Through the `Template` there are two ways to use the pagination resource. The first one is to define the value as `DocumentQuery`. Thus it will return a query as a list; however, it will break the results into pages.

```
DocumentTemplate template = //instance;
Pagination pagination = Pagination.page(1).size(1);
DocumentQuery query = DocumentQueryPagination.of(
        select().from("person").build(), pagination);
Stream<Person> people = template.select(query);
```

The second one is representing the page with the `Page` instance. A page is a fixed-length contiguous block of entities from the database, it has the feature to generate the next page.

```
DocumentTemplate template = //instance;
Pagination pagination = Pagination.page(1).size(1);
DocumentQueryPagination query = DocumentQueryPagination.of(
        select().from("person").build(), pagination);
```

```
Page<Person> firstPage = template.select(query);
Stream<Person> firstPageContents = page.getContent();
Page<Person> secondPage = firstPage.next();
```

Query Mapper

From a mapper query is possible either creates a query that executes using the pagination or creates a `Page` instance.

```
DocumentQueryMapperBuilder mapperBuilder = //instance;

Pagination pagination = Pagination.page(2).size(2);
DocumentQuery query = mapperBuilder.selectFrom(Person.class).build(pagination);
Page<Person> page = mapperBuilder.selectFrom(Person.class).page(template, pagination);
Stream<Person> people = mapperBuilder.selectFrom(Person.class)
        .getResult(template, pagination);
```

Repository

A Repository interface also allows using the pagination feature at these interfaces. To enable it creates a `Pagination` parameter as the last parameter.

```
 interface PersonRepository extends Repository<Person, Long> {

    List<Person> findAll(Pagination pagination);

    Set<Person> findByName(String name, Pagination pagination);

    Page<Person> findByAge(Integer age, Pagination pagination);
}
```

5.5.3. Graph

At the Graph database, the `Pagination` implementation works within a `GraphTraversal`. A `GraphTraversal` is a DSL that is oriented towards the semantics of the raw graph.

```
Pagination pagination = Pagination.page(1).size(1);
Page<Person> page = template.getTraversalVertex()
        .orderBy("name")
        .desc()
        .page(pagination);
```

Repository

A Repository interface also allows using the pagination feature at these interfaces. To enable it creates a `Pagination` parameter as the last parameter.

```
 interface PersonRepository extends Repository<Person, Long> {

    List<Person> findAll(Pagination pagination);

    Set<Person> findByName(String name, Pagination pagination);
}
```

Graph repository implementation does not support the `Page` conversion.

5.6. Bean Validation

Mapping API supports Bean Validation. This is implemented as a plugin that listens to **preEntity** events and performs bean validation on the entity.

```
@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 validation failure, a `ConstraintViolationException` will be thrown.

```
Person person = Person.builder()
                .withAge(10) //this is lower than @Min(21) defines
                .withName("Ada")
                .withSalary(BigDecimal.ONE)
                .withPhones(singletonList("123131231"))
                .build();
```

```
repository.save(person); //throws a ConstraintViolationException
```

Chapter 6. References

6.1. Frameworks

Spring Data

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

Hibernate OGM

http://hibernate.org/ogm/

Eclipselink

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

Jdbc-json

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

Simba

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

Apache Tinkerpop

http://tinkerpop.apache.org/

Apache Gora

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

6.2. Databases

ArangoDB

https://www.arangodb.com/

Blazegraph

https://www.blazegraph.com/

Cassandra

http://cassandra.apache.org/

CosmosDB

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

Couchbase

https://www.couchbase.com/

Elastic Search

https://www.elastic.co/

Grakn

https://grakn.ai/

Hazelcast

https://hazelcast.com/

Hbase

https://hbase.apache.org/

Infinispan

http://infinispan.org/

JanusGraph IBM

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

Janusgraph

http://janusgraph.org/

Linkurio

https://linkurio.us/

Keylines

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

MongoDB

https://www.mongodb.com/

Neo4J

https://neo4j.com/

OriendDB

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

RavenDB

https://ravendb.net/

Redis

https://redis.io/

Riak

http://basho.com/

Scylladb

https://www.scylladb.com/

Stardog

https://www.stardog.com/

TitanDB

http://titan.thinkaurelius.com/

Memcached

https://memcached.org/

6.3. Articles

Graph Databases for Beginners: ACID vs. BASE Explained

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

Base: An Acid Alternative

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

Understanding the CAP Theorem

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

Wikipedia CAP theorem

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

List of NoSQL databases

http://nosql-database.org/

Data access object Wiki

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

CAP Theorem and Distributed Database Management Systems

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

Oracle Java EE 9 NoSQL view

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