

Plant Nursery 4.0

An architectural comparison for different strategies of persistence
(relational vs NoSQL) in JakartaEE-based Web Applications

Jason Ravagli

Iacopo Erpichini

Under the supervision of

Jacopo Parri

Samuele Sampietro

Leonardo Scommegna

Software Architecture and Methodologies
University of Florence

September 2021





Table of Contents

1. Introduction
2. NoSQL Technologies
3. Jakarta NoSQL
4. Case of Study
5. Solution with SQL Database
6. Solution with NoSQL Database
7. Conclusions



Introduction

Introduction - Project Goals

Objectives

Comparing the design and development methodologies of two Jakarta EE web applications

1. Application with an SQL database and JPA
2. Application with a NoSQL database and Jakarta NoSQL

Case of Study - Plant Nursery 4.0

Using IoT and Industry 4.0 principles to innovate a plant nursery company





Introduction - Jakarta EE

Jakarta EE is a set of API specifications extending JSE with features suitable for web applications

Main APIs

- **JAX-RS**: provides specifications for the development of RESTful applications
- **CDI**: manage the dependency injection in a Java application
- **JPA**: a set of specifications for the Object Relational Mapping (ORM) with relational databases



Introduction - JPA for NoSQL?

- JPA has simplified the development process of applications interfacing with relational databases
- The increasing popularity of NoSQL databases has required an ORM-like tool for them
 - No standard specifications have been developed
 - Some independent solutions were provided (e.g. **Hibernate OGM**) but they did not perform well
- Recently, the Eclipse Foundation started the **Jakarta NoSQL** project
 - Java API standard for the integration of NoSQL technologies into a Jakarta EE application



Introduction - Questions to Answer

1. Is using Jakarta NoSQL more complex than JPA?
 - Does it depend on the type of NoSQL DBMS chosen?
2. Is it possible to switch from a JPA-based solution to a Jakarta NoSQL one?
 - Which components have to be redesigned and rewritten?
 - What about transactions and data consistency?
3. How much does it cost to change the NoSQL database type in a Jakarta NoSQL application?
 - What about moving from a database to another of the same type (e.g. from an open source to a commercial solution)?



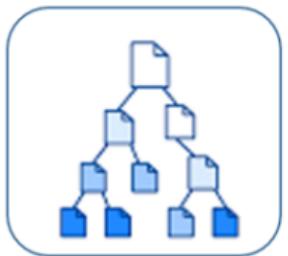
NoSQL Technologies

NoSQL - Not only SQL

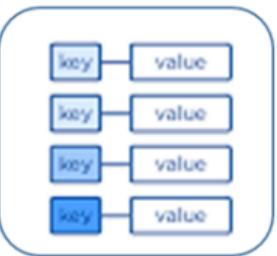
- They gained popularity in late 2000s with the rise of **Big Data** applications
- Many different types exist, but they share some **common features**:
 - Leave the schema normalization in favour of data replication
 - Flexible schemas
 - Distributed and scaling-out architectures
- ACID properties of relational databases are relaxed
 - No concept of transaction
 - Consistency of data is not automatically ensured

NoSQL - Types

There exists **4 main types** of NoSQL databases:



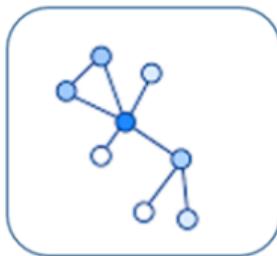
Document
Store



Key-Value
Store



Wide-Column
Store



Graph
Store



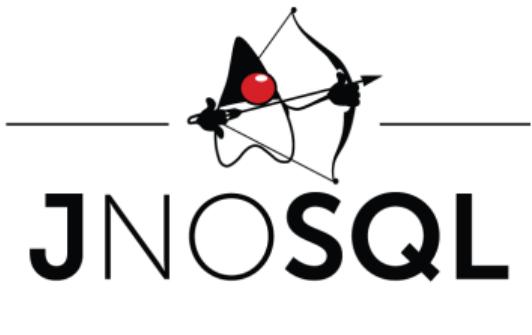
Jakarta NoSQL



Jakarta NoSQL

From the official documentation:

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



Jakarta NoSQL - Premises



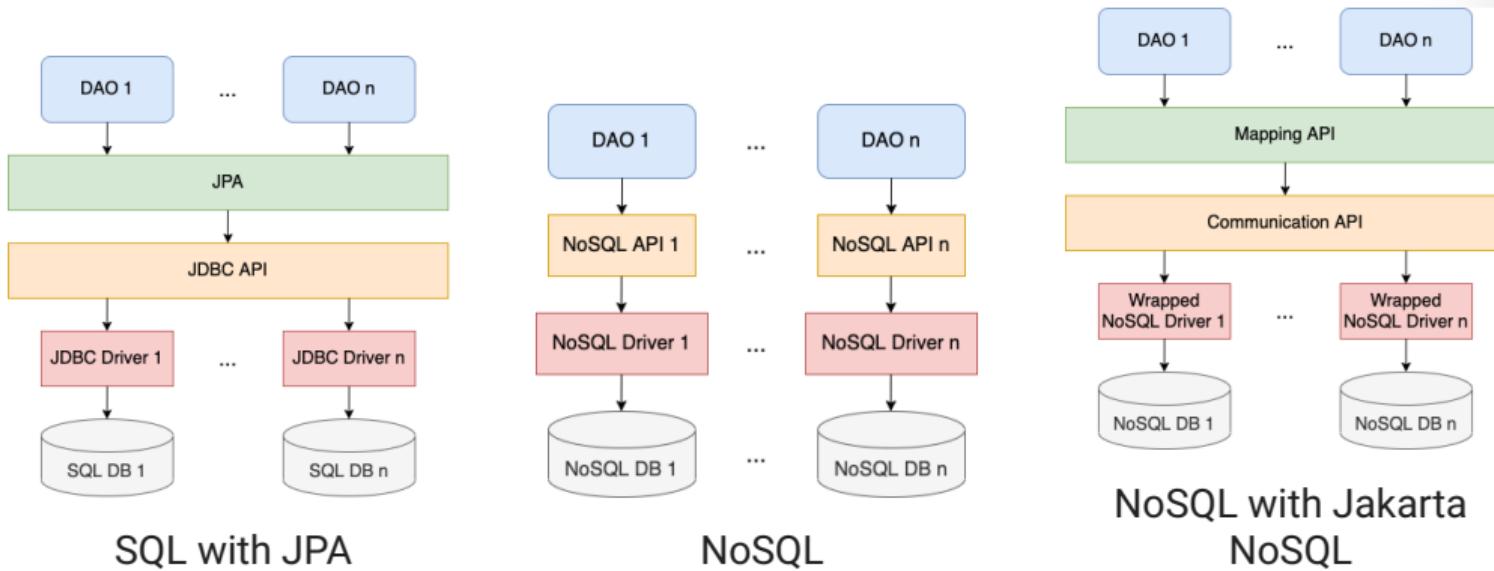
- **JPA and JDBC** allow a Jakarta EE application to access a relational DB
 - The application code is not aware of the specific database used
 - Changing database (e.g. from MySQL to PostgreSQL) is a matter of changing the JDBC driver
- Until recently, **no similar APIs existed for NoSQL databases**
 - The application code was bounded to the database-specific APIs
 - Changing the NoSQL database required lot of effort
- Some solutions have been proposed without much success (e.g. **Hibernate OGM**)
 - They tried to **reuse JPA over NoSQL databases**, with poor performance
 - They ignored the peculiarities of NoSQL (JPA was devised for relational DBMS)



Jakarta NoSQL - Architecture

- Jakarta NoSQL was born to fill up this lack of abstraction
- Inspired by JPA but devised for the NoSQL world
- Composed of **two layers**:
 - **Communication API**: lower-level layer, analogous to JDBC
 - **Mapping API**: higher-level layer, analogous to JPA
- Each API has four specialization, one for each database type
- **We focused on the Mapping API**

Jakarta EE Applications Comparison





Mapping API - Annotations

The Mapping API provides annotations similar to the JPA ones:

- **@Entity**: maps the class into a database entity
- **@Column**: specifies which class attributes must be persisted
- **@Id**: specifies which fields belong to the key. It replaces @Column
- **@MappedSuperclass**: models inheritance relationships
- **@Embeddable**: models composition relationships



Mapping API - Annotations

Class BaseEntity

```
@MappedSuperclass  
public class BaseEntity {  
    @Id("id")  
    private Long id;  
}
```

Class Person

```
@Entity  
public class Person extends  
    BaseEntity {  
    @Column("name")  
    private String completeName;  
    @Column  
    private Address address;  
}
```

Class Address

```
@Entity  
public class Address {  
    @Column  
    private String street;  
    @Column  
    private String city;  
}
```

```
{  
    "id":10,  
    "name":"Jason Ravagli",  
    "address":{  
        "city":"Pistoia",  
        "street":"via Lunga 7"  
    }  
}
```



Mapping API - Querying the Database

- The Mapping API has **two equivalent ways** of querying the database:
Template classes and the **Repository interface**
- **We chose the Template classes**
 - Using them to access a NoSQL database is similar to access a relational one through JPA
- **Template classes are 4**, depending on the type of database used:
 - DocumentTemplate
 - ColumnTemplate
 - KeyValueTemplate
 - GraphTemplate

Template Classes



- They provide **convenient methods to store, delete and retrieve data**
 - insert
 - update
 - delete
 - singleResult
 - select
- The methods use **3 components to dialogue with the Communication API** and access the database
 - EntityManager
 - Converter
 - Workflow
- To better understand how Template classes are used let us consider an **example with a MongoDB database** (a document database)



Template Classes - Instantiation - 1

We have to define a bean producer class that connects to the database and create an EntityManager object:

```
@ApplicationScoped
public class DocumentCollectionManagerProducer {

    private static final String COLLECTION = "developers";

    private DocumentConfiguration configuration;
    private DocumentCollectionManagerFactory managerFactory;

    @PostConstruct
    public void init() {
        configuration = new MongoDBDocumentConfiguration();
        Map<String, Object> settings = Collections.singletonMap("mongodb-server-host-1",
            "localhost:27017");
        managerFactory = configuration.get(Settings.of(settings));
    }

    ...
}
```



Template Classes - Instantiation - 2

```
...
@Produces
public DocumentCollectionManager getManager() {
    return managerFactory.get(COLLECTION);
}
}
```

Then we can instantiate a Template object using the CDI injection:

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



Template Classes - Insert and Update

The **insert** and **update** methods allow to create and edit entities on the database:

```
Person person = new Person();
// Set fields

Person personInserted = template.insert(person);

// Update person fields

template.update(person);
```



Template Classes - Retrieve and Delete

- The ***singleResult*** and ***select*** methods allow to retrieve the data
- They take a **DocumentQuery** object

```
DocumentQuery query1 = DocumentQuery.select()
    .from("Person")
    .where("id").eq(3)
    .build();
Optional<Person> person = template.singleResult(query1);

DocumentQuery query2 = DocumentQuery.select()
    .from("Person")
    .where("name").eq("Iacopo Erpichini")
    .build();
Stream<Person> people = template.select(query2);
```

- Delete queries are similar, but they use the ***delete*** method and **DocumentDeleteQuery**



Case of Study

Case of Study - Plant Nursery Industry

- Multiple steps of the productive cycle
- **Large variety of species** are grown, each with **different growth conditions**
- Multiple types of **growth environments**
- **Extensive lands** spread across a wide area



Case of Study - Vannucci Piante

- Owns lands extended for 545 hectares and cultivates over 3000 species of plants
 - Hard to manage without computer technologies
- Uses sensors to measure cultivation parameters



Case of Study - Plant Nursery 4.0

Our Idea

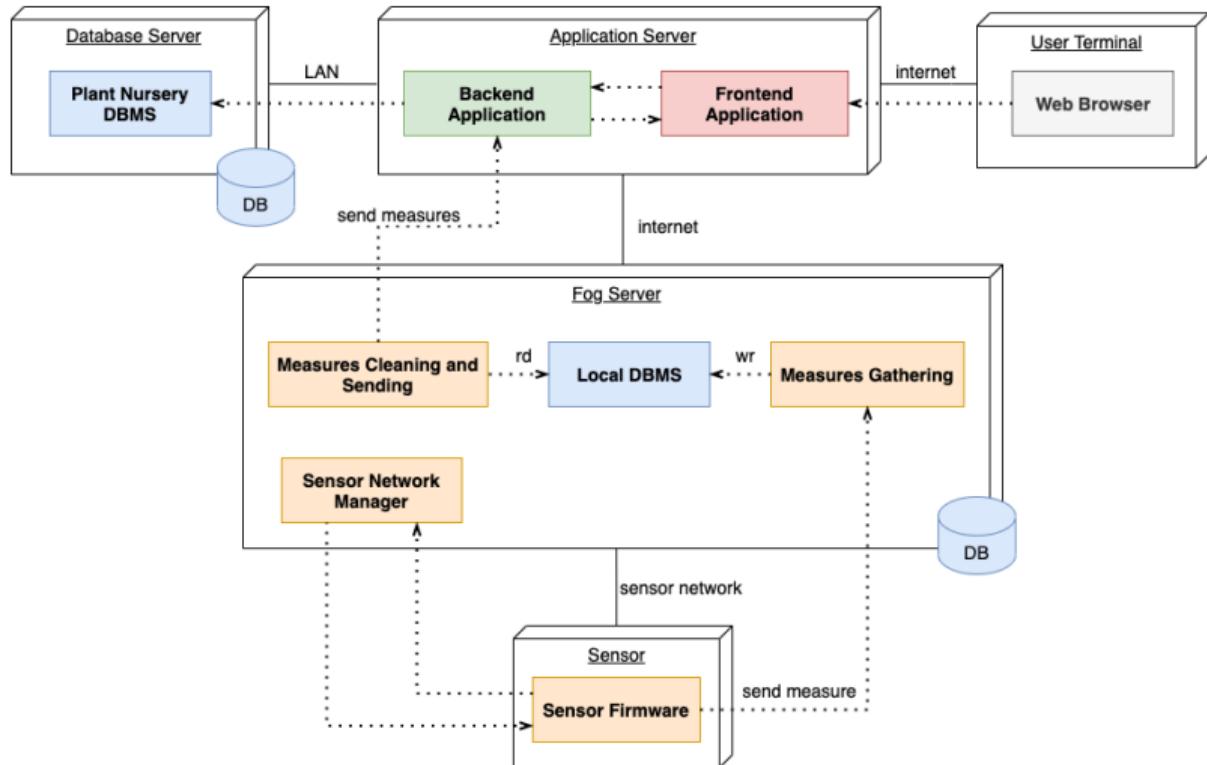
A management system for plant production inspired by industry 4.0

- Reduces the required staff
- Increases the control over the plant quality and life cycle
- Reduces waste

Our Hypothetical Nursery

- Owns lands in a wide geographical area
- Has 4 different growth environments
- Uses **sensors** to monitor environmental parameters

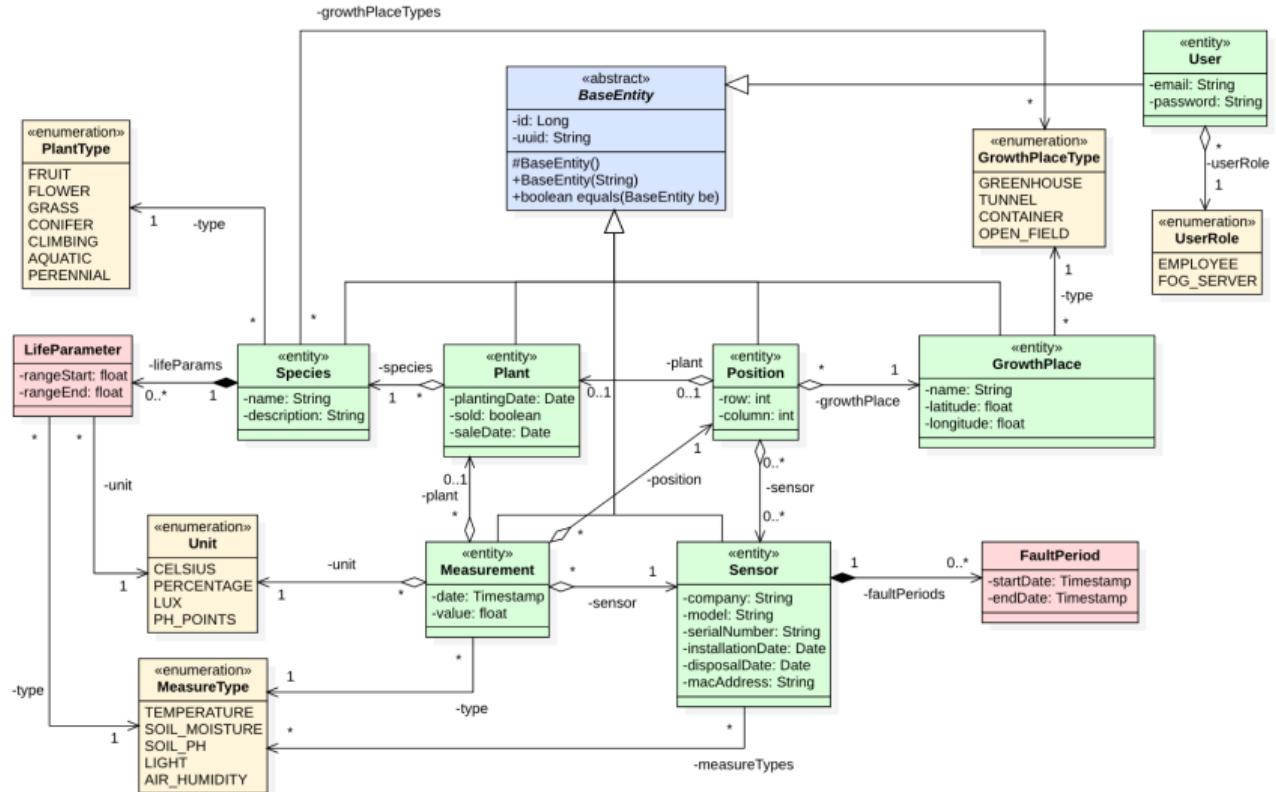
Deployment Diagram





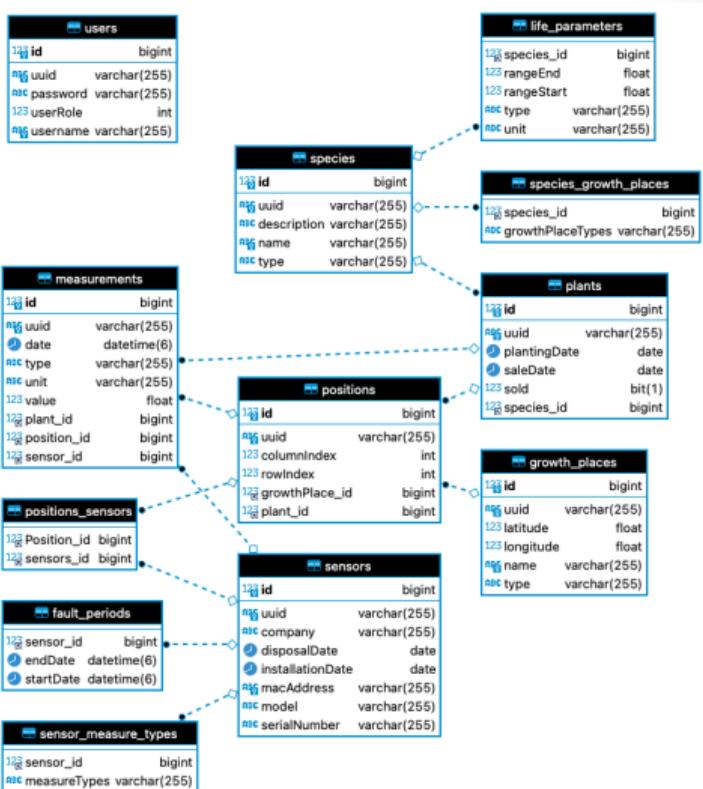
Solution with SQL Database

Domain model

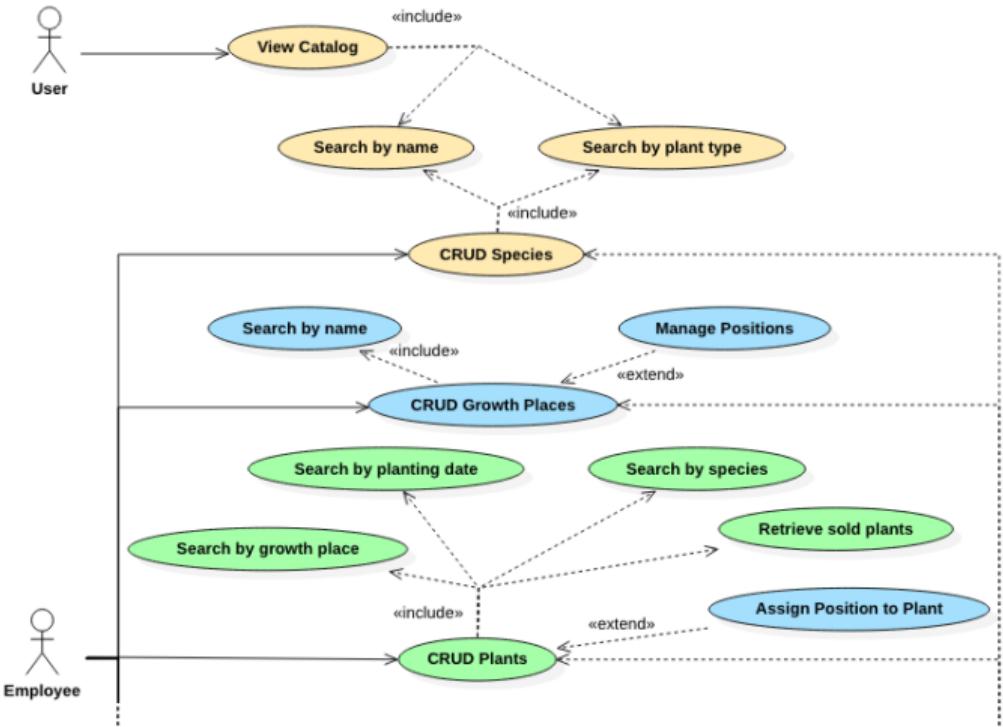


Entity Relationship Diagram

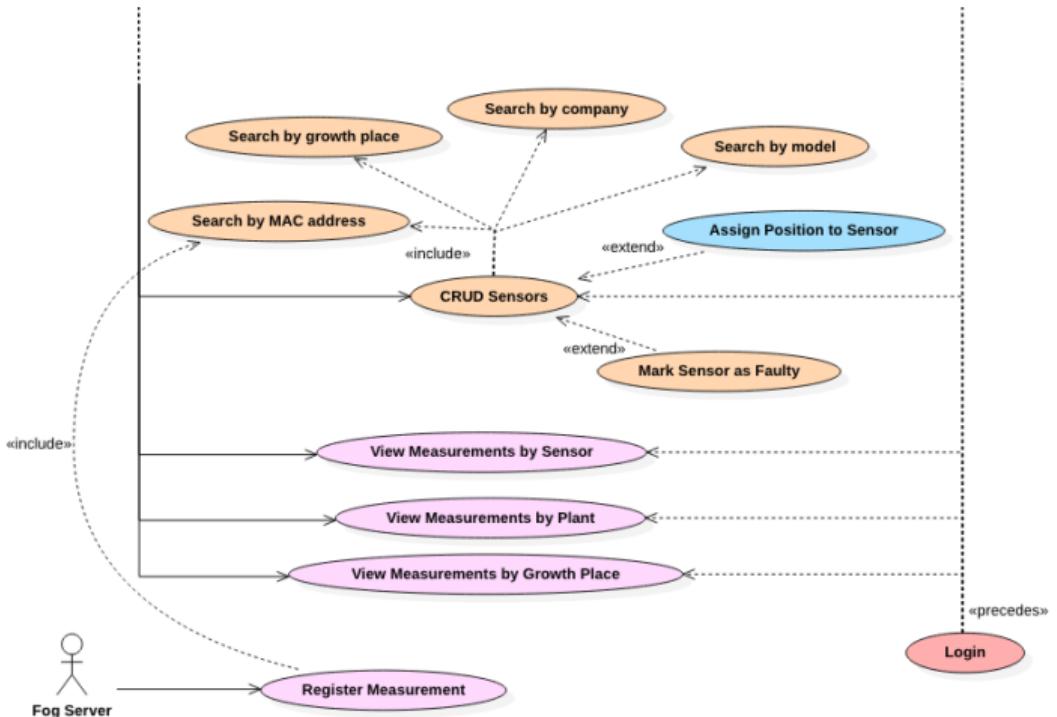
- Green classes are annotated with `@Entity`
- Red classes are annotated with `@Embeddable`, becoming `@Embedded` fields
- `BaseEntity` is a `@MappedSuperclass` entity



Use Cases - 1



Use Cases - 2





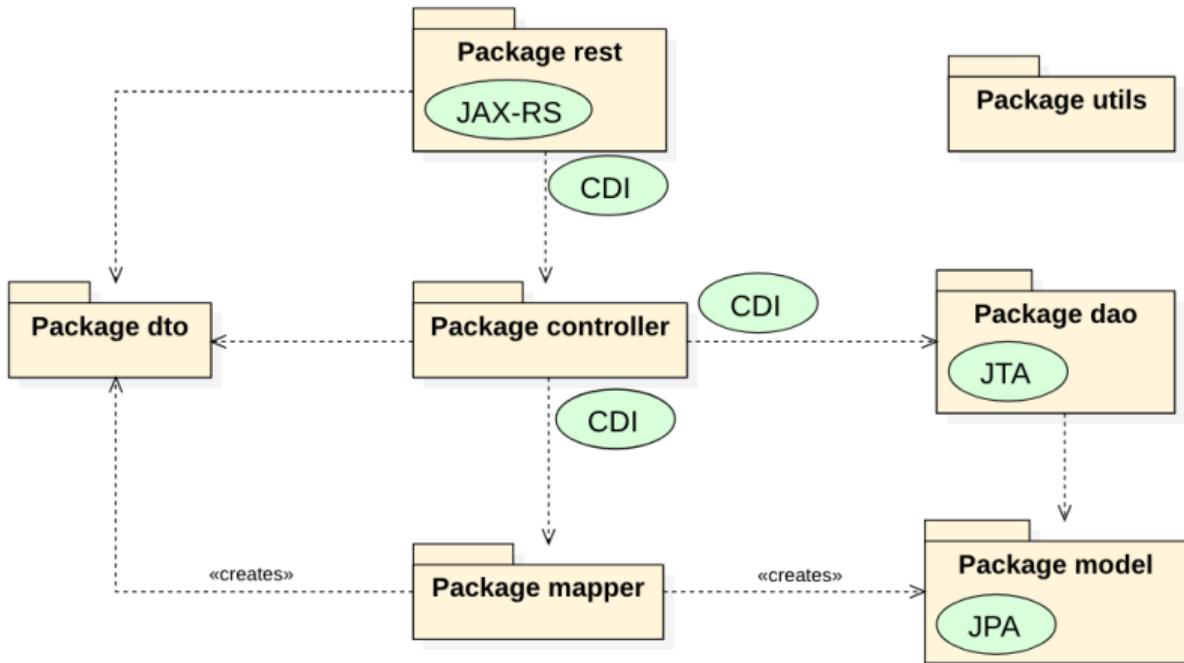
Implementation

The application is **fully functional** and covers all the use cases presented

Development Tools

- **Hibernate** is used as the JPA provider
- **WildFly 24.0** as the application server
- **MySQL 8.0** as the database, virtualized inside a **Docker container**
- **Postman** was used to test the endpoints

Implementation - Package diagram





Solution with NoSQL Database



Choosing a NoSQL Database

- IoT applications must handle huge streams of input data
- Considering the Vannucci Piante example:
 - Tens of thousands of cultivated plants
 - It is reasonable to acquire one measure for each plant every 10 minutes
 - **Tens of thousands of write operations per minute**
- We chose **Cassandra**

Cassandra



- **Wide Column Store:** data are stored in column families
 - 2D layout with rows and columns (similar to SQL tables)
 - Flexible schema: each row can have a different number of columns
 - The number of columns can be very high
 - Rows are identified by a key
- Optimized to handle frequent write operations
- Distributed: it can scale-out easily
- **Keys** are composed of **two parts**:
 - **Partition key**
 - **Clustering key**



Cassandra - Query-Driven Design



- Domain-Driven Design is suitable for relational databases, but it is not the best choice for some NoSQL DBs
- **Query-Driven Design** focuses on the use cases
 - Each column family is designed to satisfy one use case
- The design process is based on the **main characteristics of Cassandra**
 - Write operations are cheap
 - Disk space is cheap
 - Network communications involving multiple nodes are expensive
 - Performance highly depend on the design of the schema



Cassandra - Query-Driven Design

The design process must **avoid the non-goals** and **follow the goals**

Non-Goals

- Minimize the number of writes
- Minimize the data duplication

Goals

- Spread data evenly around the cluster
- Minimize the number of partition reads



Cassandra - Schema

<code>growth_places_by_id</code>	<code>plants_by_gp</code>	<code>plants_by_filter</code>	<code>species_by_id</code>	<code>positions_by_sensor</code>	<code>sensors_by_gp</code>	<code>sensors_by_id</code>
<code>id P</code> timeuuid <code>col_positions</code> int <code>latitude</code> float <code>longitude</code> float <code>name</code> text <code>row_positions</code> int <code>type</code> text	<code>id P</code> timeuuid <code>id_growth_place</code> timeuuid <code>planting_date C</code> date <code>latitude</code> float <code>longitude</code> float <code>name</code> text <code>row_index</code> int	<code>id P</code> timeuuid <code>id C</code> timeuuid <code>sold</code> boolean <code>species_id</code> timeuuid <code>species_name</code> text	<code>id P</code> timeuuid <code>description</code> text <code>id C</code> timeuuid <code>free</code> boolean <code>growth_place_id</code> timeuuid <code>growth_place_name</code> text <code>name</code> text <code>type</code> text	<code>id_sensor P</code> timeuuid <code>id C</code> timeuuid <code>col_index</code> int <code>free</code> boolean <code>growth_place_id</code> timeuuid <code>growth_place_name</code> text <code>list_sensors</code> set <code>row_index</code> int	<code>id P</code> timeuuid <code>id C</code> timeuuid <code>disposal_date</code> date <code>fault_periods</code> set <code>id_plant</code> timeuuid <code>installation_date</code> date <code>mac_address</code> text <code>measure_types</code> set <code>serial_number</code> text	<code>id P</code> timeuuid <code>company</code> text <code>model</code> text <code>disposal_date</code> date <code>fault_periods</code> set <code>id C</code> timeuuid <code>id C</code> timeuuid <code>installation_date</code> date <code>mac_address</code> text <code>measure_types</code> set <code>model</code> text <code>serial_number</code> text
<code>plants_by_id</code>	<code>plants_by_sold</code>	<code>plants_by_species</code>	<code>species_by_filter</code>	<code>sensors_by_company</code>	<code>sensors_by_mac_address</code>	<code>sensors_by_model</code>
<code>id P</code> timeuuid <code>id_growth_place</code> timeuuid <code>planting_date</code> date <code>sale_date</code> date <code>sold</code> boolean <code>species_id</code> timeuuid <code>species_name</code> text	<code>id P</code> timeuuid <code>sold P</code> boolean <code>planting_date C</code> date <code>id C</code> timeuuid <code>id_growth_place</code> timeuuid <code>sale_date</code> date <code>species_id</code> timeuuid <code>species_name</code> text	<code>id P</code> timeuuid <code>species_id P</code> timeuuid <code>planting_date C</code> date <code>id C</code> timeuuid <code>id_growth_place</code> timeuuid <code>sale_date</code> date <code>sold</code> boolean <code>species_id</code> timeuuid <code>species_name</code> text	<code>type P</code> text <code>id C</code> timeuuid <code>name</code> text	<code>company P</code> text <code>model C</code> text <code>id C</code> timeuuid <code>disposal_date</code> date <code>fault_periods</code> set <code>id C</code> timeuuid <code>id C</code> timeuuid <code>installation_date</code> date <code>mac_address</code> text <code>measure_types</code> set <code>serial_number</code> text	<code>mac_address P</code> text <code>model P</code> text <code>id C</code> timeuuid <code>company</code> text <code>disposal_date</code> date <code>fault_periods</code> set <code>id C</code> timeuuid <code>id C</code> timeuuid <code>installation_date</code> date <code>mac_address</code> text <code>measure_types</code> set <code>serial_number</code> text	<code>model P</code> text <code>id C</code> timeuuid <code>company</code> text <code>disposal_date</code> date <code>fault_periods</code> set <code>id C</code> timeuuid <code>id C</code> timeuuid <code>installation_date</code> date <code>mac_address</code> text <code>measure_types</code> set <code>serial_number</code> text
<code>positions_by_gp</code>	<code>positions_by_id</code>	<code>positions_by_plant</code>	<code>measurements_by_sensor</code>	<code>measurements_by_plant</code>	<code>measurements_by_gp</code>	
<code>growth_place_id P</code> timeuuid <code>free C</code> boolean <code>id C</code> timeuuid <code>col_index</code> int <code>growth_place_name</code> text <code>id_plant</code> timeuuid <code>list_sensors</code> set <code>row_index</code> int	<code>id P</code> timeuuid <code>col_index</code> int <code>id C</code> timeuuid <code>free</code> boolean <code>growth_place_id</code> timeuuid <code>growth_place_name</code> text <code>id_plant</code> timeuuid <code>list_sensors</code> set <code>row_index</code> int	<code>id P</code> timeuuid <code>id C</code> timeuuid <code>free</code> boolean <code>growth_place_id</code> timeuuid <code>growth_place_name</code> text <code>id C</code> timeuuid <code>id C</code> timeuuid <code>list_sensors</code> set <code>row_index</code> int	<code>sensor P</code> timeuuid <code>meas_date C</code> timestamp <code>id C</code> timeuuid <code>id_plant</code> timeuuid <code>id C</code> timeuuid <code>id C</code> timeuuid <code>unit</code> text <code>value</code> float	<code>sensor P</code> timeuuid <code>meas_date C</code> timestamp <code>id C</code> timeuuid <code>id_plant</code> timeuuid <code>id C</code> timeuuid <code>id C</code> timeuuid <code>unit</code> text <code>value</code> float	<code>growth_place P</code> timeuuid <code>mean_date C</code> timestamp <code>id C</code> timeuuid <code>id C</code> timeuuid <code>id C</code> timeuuid <code>id C</code> timeuuid <code>unit</code> text <code>value</code> float	

So many tables: let us focus on the Sensors ones...

Cassandra - Sensors Tables

sensors_by_gp	sensors_by_id	
P <code>id_growth_place</code> timeuuid RBC <code>company</code> C text RBC <code>model</code> C text P <code>id</code> C timeuuid C <code>disposal_date</code> date C <code>fault_periods</code> set C <code>installation_date</code> date RBC <code>mac_address</code> text C <code>measure_types</code> set RBC <code>serial_number</code> text	P <code>id</code> timeuuid RBC <code>company</code> text C <code>disposal_date</code> date C <code>fault_periods</code> set P <code>id_growth_place</code> timeuuid C <code>installation_date</code> date RBC <code>mac_address</code> text C <code>measure_types</code> set RBC <code>model</code> text RBC <code>serial_number</code> text	
sensors_by_company	sensors_by_mac_address	sensors_by_model
P <code>company</code> C text RBC <code>model</code> C text P <code>id</code> C timeuuid C <code>disposal_date</code> date C <code>fault_periods</code> set P <code>id_growth_place</code> timeuuid C <code>installation_date</code> date RBC <code>mac_address</code> text C <code>measure_types</code> set RBC <code>serial_number</code> text	P <code>mac_address</code> C text P <code>id</code> C timeuuid RBC <code>company</code> text C <code>disposal_date</code> date C <code>fault_periods</code> set P <code>id_growth_place</code> timeuuid C <code>installation_date</code> date RBC <code>mac_address</code> text C <code>measure_types</code> set RBC <code>serial_number</code> text	P <code>model</code> C text P <code>id</code> C timeuuid RBC <code>company</code> text C <code>disposal_date</code> date C <code>fault_periods</code> set P <code>id_growth_place</code> timeuuid C <code>installation_date</code> date RBC <code>mac_address</code> text C <code>measure_types</code> set RBC <code>serial_number</code> text

P: Partition key - **C:** Clustering key



Implementation

- We realized a second application with the same functionalities of the relational one

Packages Reused

- REST Endpoints
- DTOs

Packages Reimplemented

- Model
- DAOs
- Mapper
- Controller



Implementation - Faced Issues

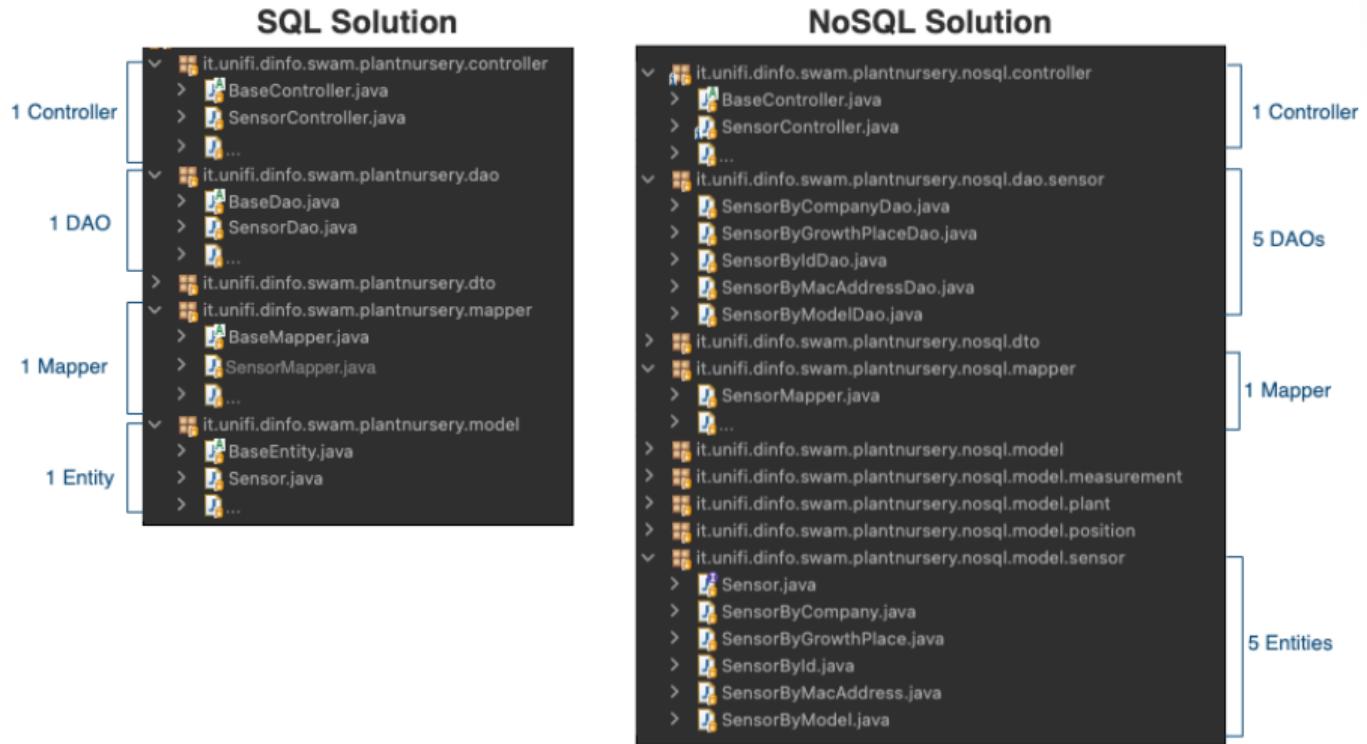
- Jakarta NoSQL does not auto-generate the database schema from the annotated Java classes
- The Java classes must be bounded to an existing database schema through annotations
- We required a class entity for each column family
- We required a Mapper and a DAO for each entity
- **A lot of duplicated code** to write



Implementation - Faced Issues

- Collections of Enum and collections of @Embeddable throw errors
 - We had to use **only collections of strings** in entity classes
 - Mappers had to convert Enum and @Embeddable to/from string
- Mapping API allows textual queries but they must comply the API syntax
 - **Database-specific operators are not allowed** (e.g. ALLOW FILTERING)
 - It increased the data duplication in the database schema
- Next slides present all this issues considering the class related to the **sensor entity**

Sensor - Packages Comparison



Sensor - Entities

SQL

```
@Entity
@Table(name = "sensors")
public class Sensor extends BaseEntity {

    @Column(unique = true)
    private String macAddress;
    private String company;
    private String model;
    private String serialNumber;
    private LocalDate installationDate;
    private LocalDate disposalDate;

    @ElementCollection
    @CollectionTable(name = "sensor_measure_types",
        joinColumns = @JoinColumn(name = "sensor_id"))
    @Enumerated(EnumType.STRING)
    private Set<MeasureType> measureTypes;

    @ElementCollection
    @CollectionTable(name = "fault_periods", joinColumns =
        @JoinColumn(name = "sensor_id"))
    private Set<FaultPeriod> faultPeriods;

    // getters and setters
}
```

NoSQL

```
@Entity("sensors_by_gp")
public class SensorByGrowthPlace extends BaseEntity
    implements Sensor {

    @Column("mac_address")
    private String macAddress;
    @Id("company")
    private String company;
    @Id("model")
    private String model;
    @Column("serial_number")
    private String serialNumber;
    @Column("installation_date")
    private LocalDate installationDate;
    @Column("disposal_date")
    private LocalDate disposalDate;
    @Column("measure_types")
    private Set<String> measureTypes;
    @Column("fault_periods")
    private Set<String> faultPeriods;
    @Id("id_growth_place")
    private UUID idGrowthPlace;

    // getters and setters
}
```

Sensor - Mapper - NoSQL



SensorMapper exploits the Sensor interface to map all the sensor entities with a single method:

```
@Dependent
public class SensorMapper {

    public <T extends Sensor> T toEntity(UUID id, SensorDto dto, Class<T> type) throws InstantiationException,
        IllegalAccessException {
        Sensor entity = type.newInstance();
        entity.setId(id);
        entity.setCompany(dto.getCompany());
        // set simple attributes

        Set <String> measureTypes = dto.getMeasureTypes().stream().map(t -> t.toString()).collect(Collectors.toSet());
        entity.setMeasureTypes(measureTypes);

        Set <String> faultPeriods = dto.getFaultPeriods().stream().map(fp -> fp.toString()).collect(Collectors.toSet());
        entity.setFaultPeriods(faultPeriods);

        return type.cast(entity);
    }

    public SensorDto toDto(Sensor entity) {...}
}
```

Sensor - DAOs - SQL

Only one DAO with one method to retrieve Sensor entities:

```
public List<Sensor> getFilteredSensors(GrowthPlace growthPlace, String company, String model, String macAddress,
    Boolean active) {
    TypedQuery<Sensor> query;
    String queryString = "";

    if (growthPlace != null) {
        queryString = "SELECT DISTINCT s FROM Position p JOIN p.sensors s LEFT JOIN FETCH s.faultPeriods "
            + "JOIN FETCH s.measureTypes WHERE p.growthPlace = :growthPlace AND "
            + "(:company is null OR company = :company) AND (:model is null OR model = :model) AND (:macAddress is null OR
                macAddress = :macAddress)";
    } else {
        queryString = "SELECT DISTINCT s FROM Sensor s LEFT JOIN FETCH s.faultPeriods JOIN FETCH s.measureTypes "
            + "WHERE (:company is null OR company = :company) AND "
            + "(:model is null OR model = :model) AND (:macAddress is null OR macAddress = :macAddress)";
    }

    if (active != null) {
        queryString += active ? " AND disposalDate IS NULL" : " AND disposalDate IS NOT NULL";
    }
    // set parameters

    return query.getResultList();
}
```



Sensor - DAOs - NoSQL

5 DAOs with multiple query methods (one for each parameters combination):

```
@Dependent
public class SensorByGrowthPlaceDao extends BaseDao<SensorByGrowthPlace> {
    private static String TABLE_NAME = "sensors_by_gp";

    public void delete(UUID idGrowthPlace, String company, String model, UUID idSensor) {
        ColumnDeleteQuery deleteQuery = ColumnDeleteQuery.delete().from(TABLE_NAME).where("id_growth_place").eq(idGrowthPlace)
            .and("id").eq(idSensor).and("company").eq(company).and("model").eq(model).build();
        columnTemplate.delete(deleteQuery);
    }

    public void update(Sensor oldSensor, SensorByGrowthPlace updatedSensor) {
        delete(oldSensor.getIdGrowthPlace(), oldSensor.getCompany(), oldSensor.getModel(), oldSensor.getId());
        save(updatedSensor);
    }

    public List<SensorByGrowthPlace> getSensorsByGp(UUID idGrowthPlace) {
        ColumnQuery query = ColumnQuery.select().from(TABLE_NAME).where("id_growth_place").eq(idGrowthPlace).build();
        Stream<SensorByGrowthPlace> sensors = columnTemplate.select(query);
        return sensors.collect(Collectors.toList());
    }

    public List<SensorByGrowthPlace> getSensorsByGpAndCompany(UUID idGrowthPlace, String company) {...}

    public List<SensorByGrowthPlace> getSensorsByGpAndCompanyAndModel(UUID idGrowthPlace, String company, String model) {...}
}
```



Conclusions

Conclusions

Question 1

Is using Jakarta NoSQL more complex than JPA?

- Does it depend on the type of NoSQL DBMS chosen?
- It is **simple to learn** how to use the **Mapping API of Jakarta NoSQL**
 - Annotations and querying using Template classes are similar to JPA
- Using the Mapping API is almost **independent of the type of NoSQL database**
 - Although we have to use the right API specialization, all of them provide the same methods and functionalities



Conclusions

Question 2

Is it possible to switch from a JPA-based solution to a Jakarta NoSQL one?

- Which components have to be redesigned and rewritten?
- What about transactions and data consistency?
- **The migration from a SQL to a NoSQL database is not painless** using Jakarta NoSQL
 - Different technologies and design processes lead to incompatible architectures
 - **Model, Mappers, DAOs and Controllers must be redesigned** and rewritten
- Mapping API does not contemplate transactions



Conclusions

Question 3

How much does it cost to change the NoSQL database type in a Jakarta NoSQL application?

- What about moving from a database to another of the same type (e.g. from an open source to a commercial solution)?
- **The migration from a NoSQL database to another presents problems**
 - NoSQL databases are really different from each other
 - Jakarta NoSQL does not autogenerate the schema
 - **Entity classes** (as well as Mappers and DAOs) must be **rewritten to guarantee good performance**
- Changing to another database of the same type does not require any modification



Conclusions

Jakarta NoSQL is a promising project but it is still at an embryonal stage

- Does not provide a real abstraction API for NoSQL Java applications
- Does not offer too many advantages over database-specific drivers

Future Works

- Compare the performance of our Jakarta NoSQL implementation with another that uses Cassandra drivers
 - To test the overhead introduced by the framework
- Plant Nursery 4.0 can be a good starting point for a real world application
 - Could be interesting to see the whole system implemented

Thank you for your attention



UNIVERSITÀ
DEGLI STUDI
FIRENZE