Plant Nursery 4.0

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

Iacopo Erpichini
iacopo.erpichini@stud.unifi.it

Jason Ravagli
jason.ravagli@stud.unifi.it

Under the supervision of

Jacopo Parri **Samuele Sampietro**
Leonardo Scommegna

Software Architectures and Methodologies
20 September 2021

Università degli Studi di Firenze
Scuola di Ingegneria

Contents

1 Introduction

Java Enterprise Edition (JEE) is a set of API specifications that extends Java Standard Edition introducing additional features suitable for web applications. Among them, three groups of specifications are particularly relevant when talking about modern web applications:

- **JAX-RS**: provides specifications for the development of RESTful applications
- **CDI**: an API specification to manage the dependency injection in a Java application
- **JPA**: a set of specifications for the Object Relational Mapping (ORM), that is the management of relational data in Java and the mapping between Java classes and entities in relational databases

Originally created and maintained by Oracle, JEE was donated to the Eclipse Foundation in 2017. The Eclipse Foundation changed its name to **Jakarta EE**, and their first release (Jakarta EE 8) was fully compatible with JEE8, the last version of JEE. At the time of writing, Jakarta 9.1 is the latest version available.

Since Jakarta EE has been widely studied during the course of Software Architectures and Methodologies while Jakarta NoSQL is still in a prototypal stage, the report will focus on the features offered by Jakarta NoSQL and their comparison with JPA, assuming Jakarta EE is known.

In the last decade, NoSQL databases became more and more popular, giving rise to the need to have an ORM tool also for non-relational databases to speed up the development of applications that make use of them. Although some solutions, like Hibernate OGM, were provided, it did not exist an API widely accepted by the Java community nor a set of specifications like JPA. However, the Eclipse Foundation in the last couple of years started the development of **Jakarta NoSQL**, a Java API standard for the integration of NoSQL technologies into a Jakarta EE application. The purpose of this work is to compare the design and development methodologies of two Jakarta EE web applications applied to the same case of study: the first one will use an SQL database and JPA, while the second one will interface with a NoSQL database using Jakarta NoSQL.

The project takes its name from the selected case of study: **Plant Nursery 4.0**, namely the computerization of a plant nursery industry through the application of the Internet of Things (IoT) principles and technologies. The GitHub repository with the complete code can be found at [8].

The report is organized as follows. In Section 2 and 3 we present an overview of the NoSQL database technologies and Jakarta NoSQL. In Section 4 we will see the two main types of architectures for a Jakarta EE application. Section 5 presents the case of study and our context analysis. Sections 6 and 7 shows the relational solution with JPA and the NoSQL solution with Jakarta NoSQL respectively. Finally, in section 8 we take some summary considerations about

the Jakarta NoSQL technology and we outline some further work of interest about this topic.

2 Beyond Relational Databases: NoSQL

Let us start by introducing what NoSQL databases are and why they are replacing relational databases in certain contexts.

Relational databases, also called SQL databases, describe the domain model by using a table for each domain entity and foreign keys for the relations between entities. They were born between the '70s and '80s when the internet did not exist and software applications run on single central mainframes. Hence these types of databases are optimized to run on a single machine and to improve their performance we need to improve the machine performance, or, to say it in other words, to scale up the database machine.

During the late 2000s, the amount of data handled by applications rapidly increased thanks to the emergence of web and mobile applications. These applications needed to adapt to fast-changing user requirements by using flexible underlying software and hardware architectures. It is in this context that NoSQL gained popularity. NoSQL stands for "Not only SQL", and although many different types of NoSQL databases exist, they all share some common features and concepts:

- They leave the schema normalization typical of relational databases in favor of data replication. For example, an entity and its related ones can be stored in a single record of a single table, instead of partitioning them into multiple records linked together. The principles behind the design of a NoSQL schema are that disk space is cheap (so duplicated data is not a problem, a concept unthinkable when relational databases were born) and that tables must be designed thinking of optimizing use case queries.

- They support flexible schemas so that changing the entities attributes or the relations between them is not a big effort for software developers.

- They take advantage of the internet to create distributed databases. The database can be spread across multiple machines so that the system can scale out. If we need to improve the performance of the database we can just add a machine instead of replacing the existing ones with more powerful (and expensive) ones.

These properties imply that many NoSQL databases relax the ACID properties of the relational databases, not providing for example the concept of transaction and consistency of data, leaving this responsibility to the software developers.

NoSQL databases are widely used in big data applications, where the data came in a semi-structured nature and their amount to write and handle is considerable.

As we said, many different types of them exist. Let us take a look at the most common ones:

- **Document Databases**: they store data in JSON format so that each record corresponds to a JSON object. This format is closer to the data format used in applications, requiring less translation and allowing high

flexibility in entities attributes. They work well with catalogs, user profiles, and content management systems, where each record/document is unique and changes over time.

- **Key-Values Stores**: they are the simplest type of NoSQL database, storing records as a key-value pair. They are suitable for user profiles, preferences, and also IoT applications.

- **Wide Column Stores**: they organize data in tables (called column families) with columns like relational databases. However, they are devised to efficiently retrieve data related to the values of a subset of columns. Typically, a row of the table contains all the information required by a query, without the need to retrieve data from multiple tables. They work well with big amount of data and frequent write operations.

- **Graph Databases**: they focus on relationships between entities, modeling the data as a graph with nodes (entities) and edges (relationships). Unlike relational databases where relationships were implied by foreign keys mechanism, graph databases store them directly solving the problem of the overhead associated with join operations. Some use cases are social network applications and fraud detection.

Hardly a single type of database is suitable for an entire complex problem, therefore mixed types of NoSQL databases also exist.

Throughout the document, to simplify the writing we will improperly use the term *table* also for NoSQL technologies to indicate the analog of relational tables. [1][7]

3 Jakarta NoSQL

The official documentation defines concisely what **Jakarta NoSQL** is and its purposes:

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.[6]

After that Java EE was donated to the Eclipse Foundation, a new project called Eclipse JNoSQL started with the goal to provide a high-level and database agnostic framework to use NoSQL databases inside Java applications. That project evolved into Jakarta NoSQL in 2019 and, even though it is still in an embryonic stage, version 1.0 can support the entire process of integration of NoSQL technologies into a Java application. It aims to be integrated inside version 10 of Jakarta EE.

To understand the Jakarta NoSQL architecture and features we need to examine the structure of a Jakarta EE application that uses a relational database. There are the Data Access Objects (DAOs), which are classes that perform the query against the database to satisfy the required use cases. In order to accomplish this, they use JPA annotated entities and JPA query methods, which are database agnostic. JPA in turn interfaces with the JDBC driver, which is a database-specific driver that implements the JDBC specifications and wraps the database drivers. Therefore, moving, for example, from MySQL to PostgreSQL is a matter of changing the JDBC driver only, and no modifications to the code are required (Figure 1).

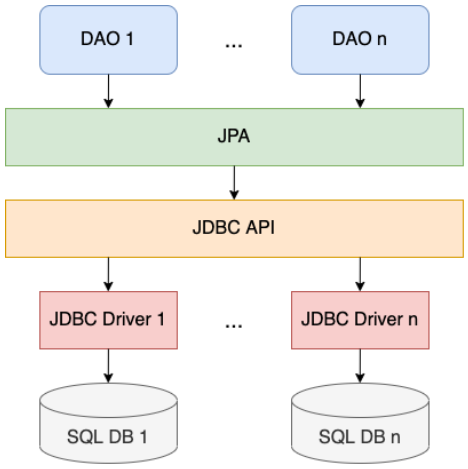

Figure 1: Typical interaction between DAOs and relational DBs through JPA and JDBC in a Jakarta EE application. The application code is totally decoupled from the particular DB used.

Until recently, using a NoSQL database inside a Java application required writing code that used drivers specific to the database chosen, coupling the code with that particular database. Changing the database type resulted in a lot of effort for the developers (Figure 2).

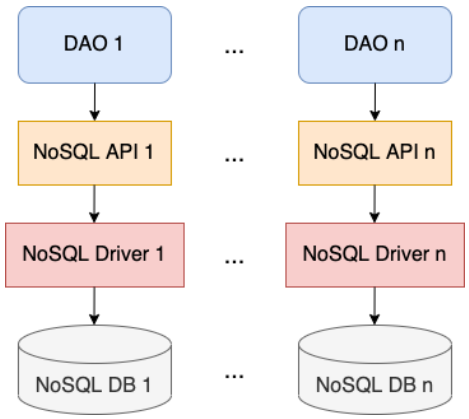

Figure 2: Interaction between DAOs and NoSQL DBs in a Jakarta EE application. No abstractions are available and DAOs directly depends on the database used. For the developer, changing the database means changing the API used and the whole DAO implementation.

Some solutions have been proposed to provide higher-level APIs with certain degrees of abstraction (e.g. Hibernate OGM, Spring Data), but all of them tried to reuse JPA over NoSQL databases. It may not be correct to adapt JPA to NoSQL technologies: JPA was explicitly devised for relational databases, and reusing it would have meant ignoring those particular features that characterize each NoSQL database type and that make them perform well under certain circumstances.

With all of these considerations in mind, the Jakarta NoSQL project was born. It is composed of two APIs that act like isolated layers:

- **Communication API**: the lower-level layer analogous to the JDBC driver in the relational solutions.
  It has four specializations, one for each type of NoSQL database (document, key-value, wide column, graph).

- **Mapping API**: the higher-level layer and the counterpart of JPA.
  Provides annotations to perform the mapping between Java classes and database entities and methods to query the database. Just like the Communication API, it has four different specializations, one for each database type, and a common package.

For the purpose of our project, we will only examine the relevant features of Mapping API, since they are on a higher abstraction level with respect to the

Communication API and can be directly compared to JPA. The complete documentation can be found at [6]. Unfortunately, some specific features are not reported in detail, but it is understandable since it is still a project under development.

3.1 Mapping API - Annotations

Like JPA, Jakarta NoSQL provides annotations to map Java classes into database entities. According to the documentation, only attribute fields can be annotated (with JPA also getter methods can be annotated instead of attributes). Getters and setters are not required since the framework directly accesses the annotated attributes. The mapped class must have a non-private constructor with no parameters.

Jakarta NoSQL developers created annotations with similar names as JPA ones to maintain consistency and facilitate the developers. These annotations are few and there is no distinction between logical e physical ones: physical specifications are available through annotations parameters. Let us see them:

- **@Entity**
  Class-level annotation that indicates Jakarta NoSQL to map that class into the database. It has a single attribute that specifies the name of the corresponding table in the database. The default name is the name of the class.

- **@Column**
  Field-level annotation. It specifies which fields must be persisted. It has an attribute indicating the name inside the database and the default name is the field name. This annotation is mandatory for all the database types but key-value ones: for them, the @Id annotation must be used for the key field, and all the others will be automatically mapped as a single blob inside the value field. Mapping between Java types and database types depends on those supported by the specific database, but in general, primitive and time types are mapped as expected. When @Column is used with an object belonging to an @Entity class, it will be incorporated as a sub-entity with modalities depending on the database type (e.g. for document databases, it will be mapped into a sub-document of the main document entity). Jakarta NoSQL also supports the mapping of collection fields. The collection must contain primitive types, @Entity-annotated or @Embeddable-annotated types (that we will see shortly). A list of the supported collections can be found in the documentation.

- **@Id**
  Field-level annotation. Similar to @Column, specify which fields compose the entity id. As @Column, an optional attribute allows specifying the mapped field name, otherwise $"\_id"$ will be the default value.

- **@MappedSuperclass**
  Class-level annotation. It allows the mapping of inheritance relationships.

When persisting sub-classes of @MappedSuperclass-annotated classes, all fields of the superclass will be considered as an extension of the sub-class ones: inside the database, the mapped class will have its attributes and the attributes of its superclass.

- **@Embeddable**
  Class-level annotation. It allows the mapping of composition relationships. @Embeddable-annotated classes define database objects with no identity. When their instances are used as @Column fields inside an entity, they will be stored as part of that entity, extending its attribute. The behavior is similar to the @MappedSuperclass one.

Below is shown an example of annotated classes and the mapping results inside a document-oriented database.

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

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

    @Column
    private Address 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"
   }
}
```

An important thing to keep in mind when working with the Jakarta NoSQL entity mapping is that there is no automatic generation of the database schema starting from annotated classes, as it happens with JPA. This is due to the already mentioned diversities of NoSQL databases and the purpose of Jakarta NoSQL to be general and database-independent. A Jakarta NoSQL application reads the annotated classes and connects to a provided database with an existing

schema. Then, through the database-specific driver, wrapped and handled by the Communication API, the framework checks if annotated classes and schema are compatible and it interprets how the mapping will happen for that particular database.

In conclusion, the design of the database remains a central part of a Java NoSQL application and cannot be assimilated by the code design as it happens with relational technologies.

3.2 Mapping API - Querying the Database

There are two ways to perform CRUD operations on the database with Jakarta NoSQL: using Template classes or through the Repository interface.

Template classes offer convenient methods to store, delete and retrieve data using three components to dialogue with the Communication API:

- **EntityManager**: the most important component. It handles the lifecycle of entities.
- **Converter**: it converts the entities to low-level objects handled by the Communication API.
- **Workflow**: it defines the workflow to store or delete an entity. It provides a series of events at which the developer can connect to manage fine-grained controls, such as validations of entities before saving.

Template classes and their components have four specializations depending on the database type, and their names change accordingly.

To better explain Template classes and how they are used let us consider a sample application with a document database. To interact with the database we will use a DocumentTemplate object. The DocumentTemplate object in turn will use a DocumentCollectionManager, DocumentEntityConverter, and a DocumentWorkflow.
Let us consider the Person class from the previous section and suppose we want to save a new Person entity, modify it and then update it on the database. We can do it with the following sample code:

```
DocumentTemplate template = //instance
Person person = new Person();
// Set fields

Person personInserted = template.insert(person);

// Update person fields

template.update(person);
```

The DocumentTemplate instantiation process is a bit tricky, and we will be back to this shortly.

To retrieve data from the database we can use the static methods of the DocumentQuery class and its collaborators. The following code first retrieve

the Person with *id* 3, then it retrieves all Persons with field *completeName* equal to *"Iacopo Erpichini"*:

```
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);
```

Note that we have to specify the name that the field has in the database (*name* in the example), not in the Java class (*completeName* in the example).

Finally, to delete records on the database we can use the static methods of the DocumentDeleteQuery class, equal to the DocumentQuery one:

```
DocumentQuery deleteQuery = DocumentDeleteQuery.delete()
                            .from("Person")
                            .where("name").eq("Jason Ravagli")
                            .build();
template.delete(deleteQuery);
```

Concerning the creation of a DocumentTemplate object, the most convenient and easy way is to use the CDI injection mechanism:

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

To make CDI create and inject the object inside the annotated field, we have to define a producer class for the DocumentCollectionManager beans, like the following:

```
@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));
  }

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

In this class we manually create the connection to the collection *developers* of a MongoDB database, we create the DocumentCollectionManager object and we make it available to CDI. With the DocumentCollectionManager, CDI can automatically create and inject DocumentTemplate objects. This concludes the basic usage of Template classes.

The other way to interact with the database in Jakarta NoSQL is using the **Repository** interface. To do it we just need to define a new interface extending from Repository, specifying the mapped class and the type of the entity key

In the following snippet we define a Repository for the class Person:

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

}
```

The interface in the snippet is already usable and provides the methods *save*, *update*, *findById*, and *deleteById*. When extending the Repository interface, Jakarta NoSQL takes care of automatically implement these basic operations.

To instantiate a defined Repository object we can use the CDI injection. We must also specify through the @Database annotation the database type for which the Repository will be used:

```
@Inject
@Database(DatabaseType.DOCUMENT)
private PersonRepository documentRepository;
```

As we saw for the Template class, to allow the injection we need to define a producer class that connects to the database and creates the EntityManager.

We can extend the basic Repository behavior by adding methods definitions to the defined interface. This way, we can use it to retrieve or delete records filtering by the values of the entity fields. The Mapping API will infer the desired queries from the names of the methods and provides the implementations. For this reason, methods must follow a naming convention, formed by a prefix and a series of concatenated operators names.
The prefix must be **findBy** if we want to retrieve data or **deleteBy** if we want to delete some records.
After the prefix, we can specify the name of the fields that we want to involve in the query filtering concatenated with a series of operators that control the filtering process:

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

- Like
- In
- OrderBy
- OrderBy_/_/_Desc
- OrderBy_/_/_Asc

An example of usage is shown below:

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

    Stream<Person> findByName(String name);

    Stream<Person> findByNameOrderByNameAsc(String name);

    void deleteByName(String name);
}
```

Again, we only need to define the methods, and the Mapping API will take care of the implementation. To know more about advanced features, please refer to the official documentation at [6].

4 Monolithic and RESTful Architectures

We are going to briefly introduce the two main types of architectures for a Jakarta EE application: the monolithic and the RESTful architecture.

A monolithic application is built as a single unit. Enterprise Applications are built in three parts: a database (consisting of many tables usually in a relational database management system), a client-side user interface (consisting of HTML pages and/or JavaScript running in a browser), and a server-side application. This server-side application will handle HTTP requests, execute some domain-specific logic, retrieve and update data from the database, and populate the HTML views to be sent to the browser. It is a monolith – a single logical executable. To make any alterations to the system, a developer must build and deploy an updated version of the server-side application.

By contrast, RESTful capabilities are expressed formally with business-oriented APIs. They encapsulate a core business capability, and as such are valuable assets to the business. The implementation of the service, which may involve integrations with systems of record, is completely hidden as the interface is defined purely in business terms. The positioning of services as valuable assets to the business implicitly promotes them as adaptable for use in multiple contexts. The same service can be reused in more than one business process or over different business channels or digital touch-points, depending on need. Dependencies between services and their consumer are minimized by applying the principle of loose coupling. By standardizing on contracts expressed through business-oriented APIs, consumers are not impacted by changes in the implementation of the service. This allows service owners to change the implementation and modify the systems of record or service compositions which may lie behind the interface and replace them without any downstream impact.

So for our comparison between SQL and NoSQL, we decided to create two RestFul type backend applications, this allows us to create the same identical endpoints for the creation of a future frontend. Having the same endpoints allows frontend developers to be able to develop an application that works on both systems in a totally transparent way.

5 Case of Study: Plant Nursery 4.0

5.1 Introduction

A plant nursery is a place where plants are cared for and grown with two main purposes: conservation biology or commercial use. The latter is the most common scenario, with companies that grow and sell plants to retailers, other nurseries or a combination of both.

The plant nursery industry is very heterogeneous, there are multiple steps of the productive cycle (e.g. propagation, growing out, sale of the final product to retailers) and some companies take care of a specific phase only. Furthermore some nurseries focus on specific types of plants, such as floral gardens or fruit trees. However, the largest nurseries grow a large variety of species and embrace all the phases and aspects of their growth.

Nurseries of this type extend on a large area and have to consider different types of environments in which to grow plants: open fields, container fields in the open air, greenhouses or tunnels (Figure 3).

Figure 3: Three types of growing environments: greenhouse, tunnel and open field.

Moreover each type of plant, besides the growing environment, has its own parameters (e.g. temperature, humidity, illumination, soil ph) to grow at his best that sometimes must be scrupulously observed.

Due to the large number of different plants and different parameters to consider for each and also the large area of land owned by the nursery, an automated system that exploit the Industry 4.0 principles can really improve and simplify the productive system.

The work is inspired by the plant industry of Pistoia, excellence that include some of the biggest nurseries in Italy.

5.2 Context Analysis

Vannucci Piante, one of the biggest nurseries in the city of Pistoia, owns lands that extend for 545 hectares and cultivates over 3000 different species of plants [9]. Considering these numbers it seems obvious that industries of these dimensions cannot do without the support of computer technologies.

In the last few years, more and more industries of all types are renewing their production sectors integrating Industry 4.0 and IoT techniques. Big nurseries

like Vannucci Piante use sensors to measure cultivation parameters, like the light intensity or the soil humidity (Figure 4), to monitor their production.

Figure 4: Monitoring the plants cultivation in the Vannucci Piante nursery.

Taking inspiration from this, we devised a management system that can simplify plant production reducing the required personal and increasing the control over the quality of the cultivated plants. Figure 5 shows the deployment diagram with the system components and their interactions.

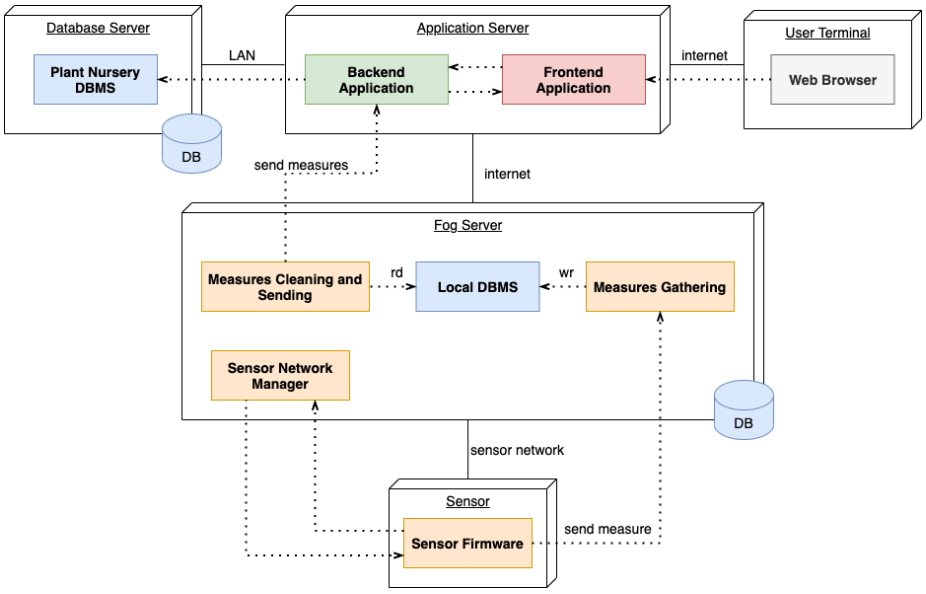

Figure 5: The deployment diagram of our system.

Our hypothetical nursery owns terrains scattered over a wide geographical area. On these terrains, plants are cultivated in 4 different growth environments: open field, tunnel, greenhouse, and containers in the open air. A series of **sensors** are deployed over the growth places to monitor some environmental parameters. We considered 5 parameters: temperature (measured in Celsius

degrees), air humidity (measured in percentage), soil moisture (also measure in percentage), soil pH (measured in pH points), and quantity of light (measured in lux). Since the growing environments can be really distant from each other, we need to manage different sensor networks and gather measures from each of them. The best solution is to install a **fog server** for each remote area (that can comprise multiple growth environments). Each fog server is connected to a sensor network and hosts 4 applications:

- An application that manages the assigned sensor networks, executing services like fault detection, network coordination, and security.
- A local database.
- An application that receives the measures from the sensors and stores them on the local database.
- An application that reads the stored measures from the local database, performs cleaning operations and sends the measures in the proper format to a central backend application.

Fog servers are connected through the internet or a VPN to an **application server** located at the nursery headquarter. The application server runs a backend and a frontend application. The backend application is a RESTful application that allows the nursery employees to monitor and administrate the system. It also gathers the measures from the fog server and provides users real-time reports and summaries of the situation on the field. To do these operations it interacts with a **database server** located in the same building. Interactions happen through a LAN connection.

Finally, the employees connect to the frontend application using a **pc** (if they are in the office) or a **tablet** (if on the field). They can take care of administrative aspects like registering new growth places and species of plants produced by the nursery. They can also do operations like registering new plants, installing new sensors, and assign them to specific positions in a growth place. With plants and sensors on the field, users can monitor the parameters of the cultivation, calculate reports and histories to monitor the quality of life of plants, and intervene promptly in case of detected anomalies (e.g. to open the irrigation, to open/close the windows of a greenhouse, to cover some plants or even to check if a sensor is faulty or misplaced).

With such a system, employees can always be in control of the situation even if not physically on the field, reducing the need for manual checks and interventions and optimizing the efforts.

An extension towards the complete automation would be integrating also actuators inside the growth environments, controlled by the system that takes decisions based on the sensor measurements. However, since the project goal is not to develop a full functioning system for simplicity we did not consider such a scenario.

5.3 Project Goal

As we mentioned in the introduction, we want to investigate the differences between the design and the implementation process of a Jakarta EE application interacting with a relational database and JPA and one using a NoSQL database with Jakarta NoSQL.

Considering the deployment diagram from the previous section, we will focus on the backend application and the database to which it connects. The choice of a RESTful architecture instead of a monolithic one allows for greater separation between the backend application and the frontend one (that we will not consider).

The Jakarta NoSQL implementation will try to replicate all the functionalities of the JPA one using a single NoSQL database. As we will see, this may appear a bit forced and mixed solutions with a relational and a NoSQL database, or with different types of NoSQL databases could be more appropriate. However, we considered it as part of the didactic purpose of the work: what Jakarta NoSQL offers and its limitations compared to JPA.

6 Solution with an SQL Database and JPA

The first solution we analyze is the one using a relational database together with JPA.

Following the guidelines of the domain-driven design, we outlined a domain model diagram. Keeping in mind the deployment diagram and the intents of the system, we listed a series of use cases that the backend must satisfy. With these use cases, we validated and revised the domain model.

The next sections show the final diagrams and present some implementation details.

6.1 Domain Model

Figure 6 shows our final domain model diagram. We designed it elaborating on the context analysis of section 5.2.

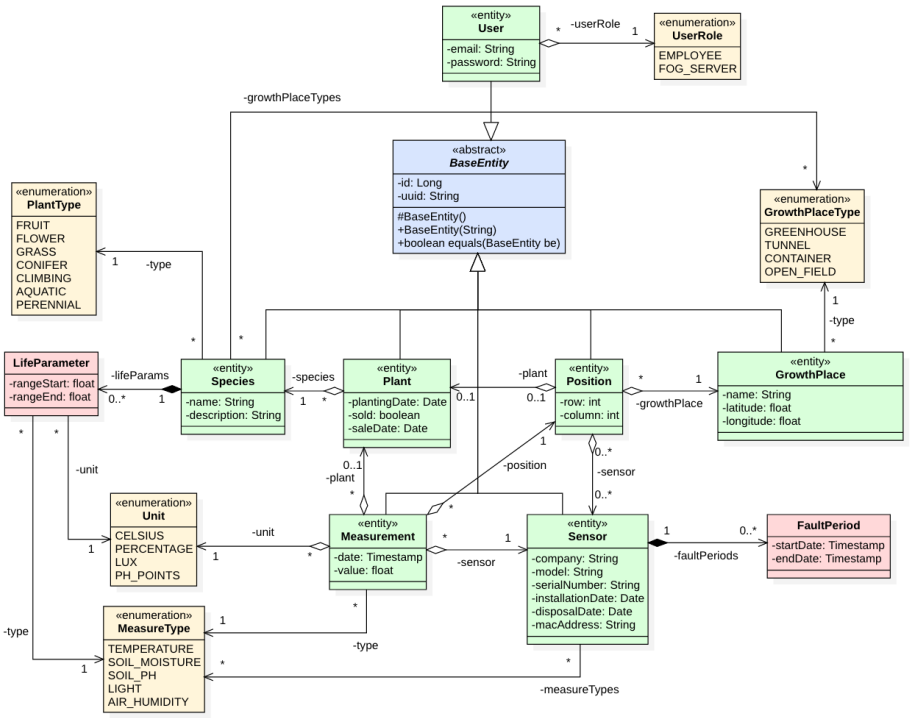

Figure 6: The domain model diagram.

A **Species** identifies a species of plant, with a name and a description. A Species has a PlantType, for which we chose seven possible values (taken from the Vannucci Piante online catalog [9]). A Species also has one or more GrowthPlaceTypes, indicating the most suitable environments for the plant, and can have some LifeParameters. A **LifeParameter** indicates the range

of values for a certain type of measurement at which the plant grows at its best. The considered types of measurement are the ones mentioned in section 5.2: temperature, air humidity, soil moisture, and soil pH. Substantially, the GrowthPlaceTypes and the LifeParameters compose the "recipe" to grow that Species.

Moving on in the diagram, we find the **Plant** entity, which identifies a single plant cultivated and sold by the nursery. A Plant has a planting date, a boolean flag indicating whether is sold, the sale date, and a Species, which it belongs to.

The **GrowthPlace** is the growth environment and it has a unique name, a set of coordinates (to locate it on a map since the nursery can have terrains distant from each other), and a GrowthPlaceType.

Then there is the **Sensor** entity, the sensing device installed inside the growth environments. It has a series of attributes (manufacturer company, model, and serial number) to keep track of its personal parameters. To be rigorous, they should have been modeled as a separate Company entity, but for simplicity, to do not introduce an additional entity, we integrated them inside the Sensor. A Sensor has a list of MeasureTypes that indicate which parameter the Sensor can monitor (of course the possible values are the same as those for the LifeParameters). This entity also has an installation date, a disposal date (to know whether it is still operating), and a unique MAC address. Keeping track of the MAC address is important to allow the fog servers to associate a sensor on a network, for which they can know the MAC address, to a sensor on the database and consequently pair sensors and measurements inside the DB. Finally, we decided to extend the Sensor assigning it some FaultPeriods. A **FaultPeriod** is a range of timestamp values in which the Sensor is considered faulty. An anomaly detection system could mark the Sensor as faulty creating a new FaultPeriod for it when its measures assume strange or unexpected values. All measures acquired during a FaultPeriod are not considered during the data analysis operations.

A growth environment has a series of positions, organized in a 2D grid, in which plants and sensors can be placed: the **Position** entity models this information, relating GrowthPlace, Sensor, and Plant. A Position has a row and column that identifies the position inside the 2D grid. It also has references to the GrowthPlace which it belongs to and to the Sensors and the Plant placed there. There can be zero or one Plant and zero or multiple Sensors in a single Position. This is reasonable since we thought of a position as a physical place in which to cultivate a single plant and for a single position we can monitor multiple types of measures through multiple sensors. Furthermore, the same Sensor can be assigned to multiple Positions since a sensor can cover a wide area (e.g. an environmental thermometer).

The last entity is the **Measurement**. It is the measure acquired by a sensor in a certain position and optionally referring to a plant (if there is a plant placed at that position). The entity has the timestamp of the instant of the acquisition, the value, the type of acquired measure along with its unit of measurement, and the references to the Sensor, the Position, and the Plant.

The **User** entity models an authenticated user that can access the REST

services, and it deals with the security layer of the application. It has a role, that can be either EMPLOYEE or FOG_SERVER, to determine to which services it can access.

All the entities in the schema that are stored in their own tables (i.e. has an identity) inherit from **BaseEntity**. BaseEntity is an abstract class that contains methods and attributes that deal with the identity of an entity (the surrogated key, the UUID assigned at the time of creation, the equals method, etc.).

6.2 Use Cases

The Use Case diagram is shown in Figure 7. The use cases are colored by the entity which they operate on.

We considered three actors interacting with our system.

The **User** is the person who connects to the public website of the nursery to have information about the nursery and the plants it sells. On the website, they can consult the plants catalog along with their information. To facilitate the navigation and the search, they can filter by plant type and name of the species.

The **Employee** is the worker of the nursery that authenticates on a reserved area of the web application. They can administrate the system performing CRUD operations on the main entities of the domain model (Species, GrowthPlace, Plant, Sensor). Reasonably, they operate through web pages showing tables with all the data. To facilitate the search of a particular entity and the group operations, we considered filter operations on the main attributes of each entity. Besides the entity management, an Employee can perform some basic data analysis on the collected measures.

The **Fog Server** is an autonomous actor that connects to the backend application to send the measures gathered from the sensors. To register a new measure, it first needs to identify each sensor through its MAC address.

The domain model from the previous section allows accomplishing all the use cases that we presented.

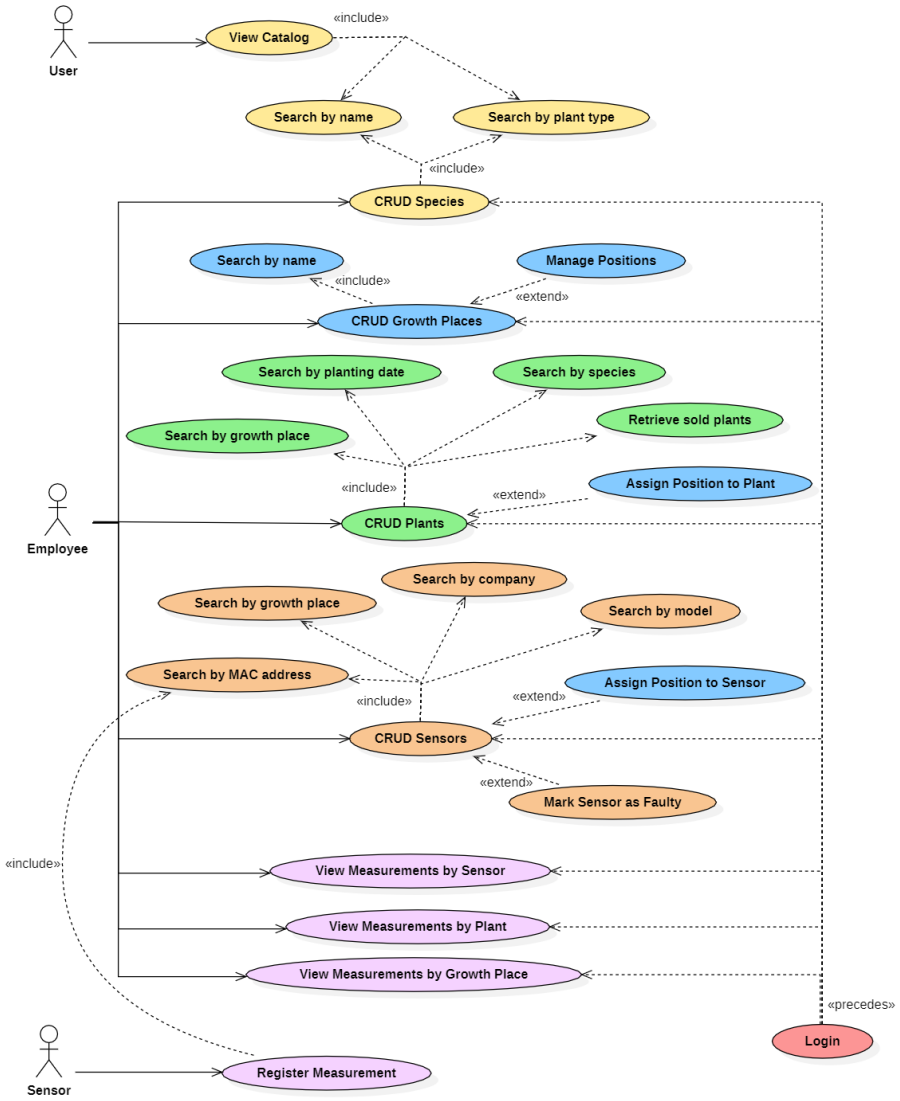

Figure 7: The Use Cases diagram.

6.3 Entity-Relationship Diagram

We created the Java classes from the domain model and we used the JPA annotations to map them to the database entities. The resulting Entity-Relationship diagram is shown in Figure 8.

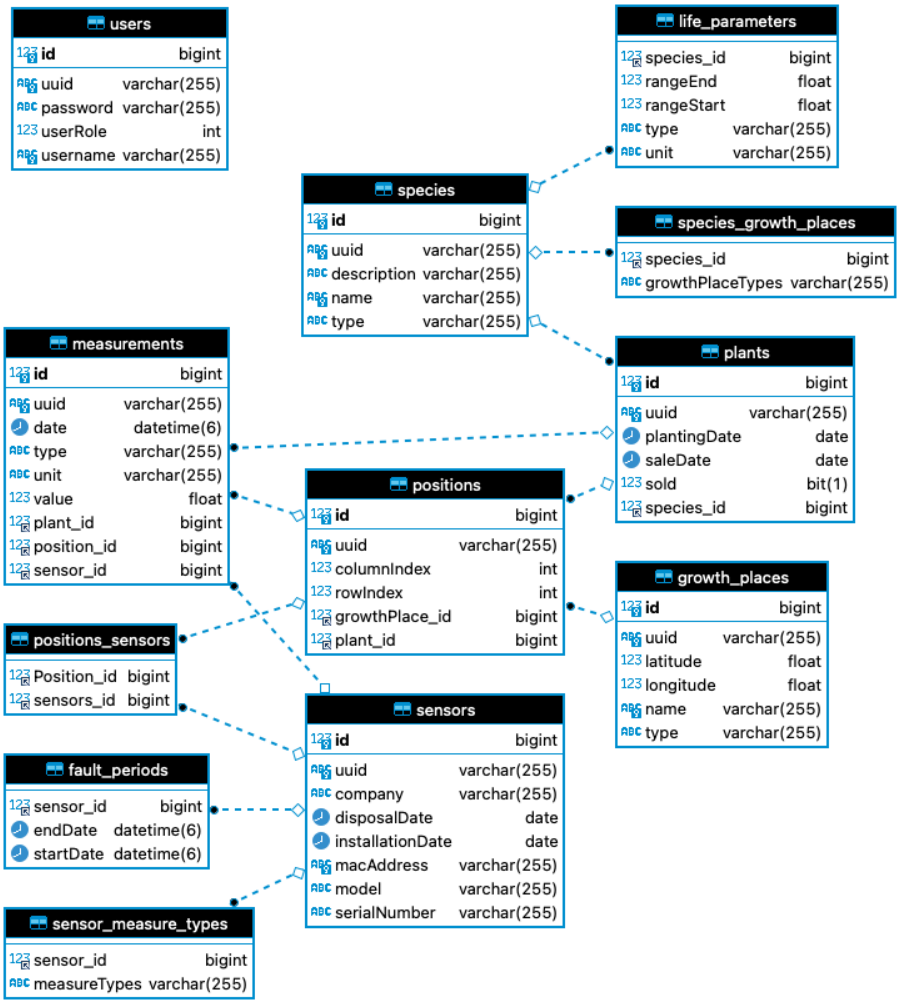

Figure 8: Entity-Relationship diagram generated from the JPA annotations.

Green classes in the domain model diagram were annotated with *@Entity*, while red classes were annotated with *@Embeddable*, becoming *@Embedded* fields inside the classes they belong to. BaseEntity is a *@MappedSuperclass* entity and has no tables associated with it (it is an abstract class that allows you to add identity management functionalities to other classes).

6.4 Software Architecture and Implementation

For the implementation, we used Hibernate as the JPA provider and WildFly 24.0 as the application server. Concerning the database, we used MySQL 8.0 virtualized inside a Docker container. The implementation was pretty straightforward. We organized the application functionalities and responsibilities separating them into business logic, domain model, and object-relational mapping.

Figure 9 shows the implemented packages and their interactions, along with the Jakarta EE technologies used by each of them.

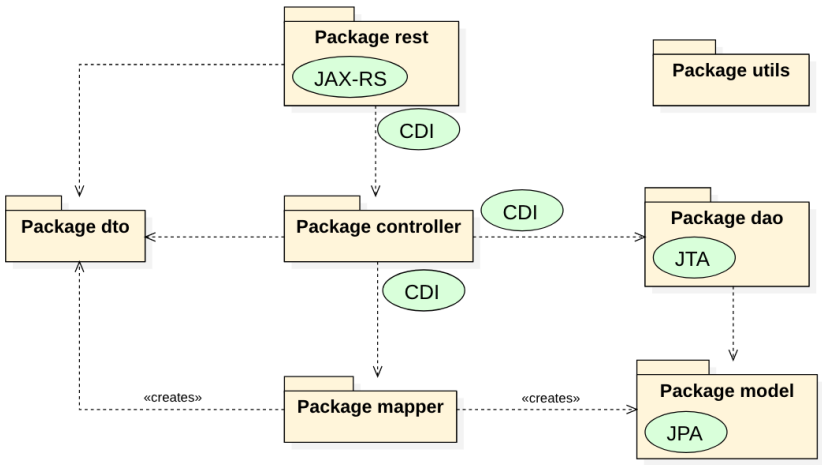

Figure 9: Diagram of the packages of our implementation and their interactions. Green ellipses indicates the Jakarta EE technologies used by packages and dependencies.

The **package rest** contains the REST endpoints at which client and frontend applications can connect through HTTP. We implemented an endpoint for each use case showed in Figure 7. The endpoints use the controllers from the **package controller** to accomplish the requests. The controllers contain the core business logic of the application and use the DAOs (**package dao**) to interact with the database and the mappers (**package mapper**) to convert DTOs (**package dto**) to and from entities (defined in the **package model**). The endpoints use DTOs instead of entities both in input and output since they are more lightweight and adapt better to some use cases.

All endpoints were been manually tested using Postman.

We would like to clarify that our implementation is fully functional and cover all the use cases presented in Section 6.2. The only aspects we left out are the security ones, such as the authentication filter to access REST services and the user role checks. Even if it was not the purpose of the project, we wanted to realize a real application that could be used as a starting point for a real commercial system.

7 Solution with a NoSQL Database and Jakarta NoSQL

One of the first things to do during the design process of the solution with Jakarta NoSQL was to choose the NoSQL database.

Indeed, unlike what happens for a relational solution, the database choice significantly affects the design process of an application with NoSQL technologies. We decided to use Cassandra, a wide column store, because of its good performance when dealing with considerable amounts of data. When designing an IoT system like the Plant Nursery 4.0 one, that uses sensors to acquire data from the environment, the possibly huge streams of data must be considered to deliver a performant application.

Let us consider the Vannucci Piante nursery example again. No official numbers are provided about the cultivated plants, but with over 3000 species treated we can estimate several tens of thousands of plants. Estimating on the downside, it is reasonable to acquire one measure for each plant (considering that a sensor can monitor multiple plants but we can acquire multiple parameters) every 10 minutes. That means a few tens of thousands of write operations per minute done on the database only to collect the measures. Then we have to consider the cost to perform data aggregation and analysis procedures on all those data whether requested by users.

A database like Cassandra, distributed and devised to handle frequent write operations, seemed suited to our problem.

In the next section, we will explain what a column-family store is and we will present the peculiarities of Cassandra. Then in Section 7.2, we will talk about the design process and the resulting database schema. Finally, in Section 7.3 we will analyze some implementation details that use Jakarta NoSQL, focusing on the similarities and differences with JPA and the faced issues.

7.1 Wide Column Databases and Cassandra

Often it is hard to locate a NoSQL database into one single type: they can be a variant of a type or a mix of different types. The same holds for Cassandra: many mistakenly refer to it as a column-oriented database (an type not belonging to any of those listed in Section 2), but it is actually a wide column store. A **wide column store** is defined as follows:

Its architecture uses (a) persistent, sparse matrix, multi-dimensional mapping (row-value, column-value, and timestamp) in a tabular format meant for massive scalability (over and above the petabyte-scale). [10]

It can be seen as a modification of a relational database: data are stored in column families, a 2D schema with rows and columns like relational tables, and rows are identified by a key. However, the schema is flexible, each row can have a different number of columns, and column families can have a very large number of them without performance issues. An important difference between these databases and SQL ones is that entity relationships cannot be represented

efficiently with references between column families, as for the relational tables: indeed joins operations are not contemplated in queries. As we will see with Cassandra when using wide column stores the database schema must be designed to have all the data of interest for a query in a single row.

Since various blogs and posts can be misleading about what **Cassandra** is, we would like to clarify which type of database it is. The definition of Cassandra on its official GitHub repository clearly states that it is not a column-oriented database:

Apache Cassandra is a highly-scalable partitioned row store. Rows are organized into tables with a required primary key. Partitioning means that Cassandra can distribute your data across multiple machines in an application-transparent matter. Cassandra will automatically repartition as machines are added and removed from the cluster. Row store means that like relational databases, Cassandra organizes data by rows and columns. The Cassandra Query Language (CQL) is a close relative of SQL. [2]

Cassandra cannot be a column-oriented store because it does not have its characteristics: column-oriented databases can retrieve efficiently data from a single column (or a small subset of columns) of a table without reading other columns of that data. They are particularly suitable for data aggregation and analysis operations. On the contrary, if we would like to calculate the mean value of the data in a column of a Cassandra column family, we should retrieve all the entire rows of the column family and then select only the column of interest.

Cassandra is designed to be a distributed database. A Cassandra cluster is composed of a set of nodes, organized in a ring topology, where each node runs an instance of Cassandra. The architecture is masterless, that is there is no main node, but each node in the cluster offers the same functionalities as the other ones. Nodes communicate and synchronize talking to each other through a protocol called gossip. These mechanisms are managed under the hood and the database appears to the client as a unified whole. Such an architecture easily scales using commercial on-the-shelf hardware: it is sufficient to add a server to the cluster and Cassandra will handle it for you. Furthermore, a masterless distributed database provides fault tolerance (also through data replication strategies) and load balancing. [3]

We said that wide column stores identify rows through keys as in relational databases. Keys in Cassandra indulge its distributed nature. A key of a column family is composed of a subset of columns and it is divided into two parts:

- **Partition key**: it is mandatory and allows to distribute data evenly across the cluster nodes. Rows with the same value (or combination of values if the partition key is composed of multiple columns) for the partition key are stored in the same node.

- **Clustering key**: the optional second part of the key. Determines the order in which rows are stored in the same node.

The design of the database schema passes through the design of the keys, which is crucial to guarantee high performance on data retrieval and load balancing on the cluster. These topics will be discussed in the next section.

As for the query language, Cassandra uses **CQL** (Cassandra Query Language) which syntax is really similar to SQL. Since it is beyond the scope of the project (we will use the Mapping API of Jakarta NoSQL to query the database), we will not deepen it.

7.2 Query-Driven Design and Database Schema

When designing a Cassandra schema for an application we have to leave the Domain-Driven Design (DDD) and embrace the Query-Driven Design (QDD) instead.

DDD is suitable for relational databases, as we saw in Section 6.1, allowing us to easily design the class diagram and the database schema in a single shot from context analysis and modeling of entity and relationships. NoSQL databases like Cassandra are not devised to explicitly represent relationships on the schema level (not without a performance loss), hence the domain model and the database schema do not match and DDD is not the best choice.

The **Query-Driven Design** focuses on the use cases: database tables are directly modeled based on the data required by each use case. How the modeling process happens depends on the type of database used. With Cassandra, this means that a column family (table) has to be designed to satisfy one (or more, if possible) use cases. A row in the column family must contain all the data required by the considered use case and the column family keys must be designed considering the query parameters of the use case.

The Datastax website provides some clear and useful guidelines for data modeling in Cassandra that follows the QDD principles [4]. These guidelines are based on the main characteristics of Cassandra regarding the performance:

- Write operations are cheap. The database is optimized to have a high write throughput.
- Disk space is cheap. When using this database, the storage cost is irrelevant.
- Network communications involving multiple nodes are expensive.
- The performance highly depends on the database schema.

Keeping these in mind, we have to design our schema avoiding following the so-called non-goals (the objective not to follow). The **non-goals** are the goals that we typically follow when dealing with relational databases but that do not apply to Cassandra:

- Minimize the number of writes: since write operations are a lot cheaper than read operations, it is worthy to improve reads performance at the cost of multiple writes whenever possible.

- Minimize data duplication: as we stated in Section X, a NoSQL schema does not have to be normalized. Furthermore, Cassandra does not provide JOIN operations between column families and we do not have to care about the disk space used. Hence we can exploit data duplication to improve the performance of reads.

The actual **goals** to follow during the design process are two and they relate to the choice of the column families keys:

- Spread data evenly around the cluster: partition keys have to be chosen so that each partition (i.e. each combination of values for the partition keys) has roughly the same amount of rows. Since rows belonging to the same partition are stored on the same node, this helps to spread the data evenly on the nodes, facilitating the load balancing and avoiding the so-called hot points (overloaded nodes).

- Minimize the number of partition reads: when executing a read query, we have to read rows from as few partitions as possible. Involving multiple partitions means querying multiple nodes of the cluster, adding network latency and overhead. Furthermore, even on the same node reading from multiple partitions is more expensive than reading from a single one due to the way rows are stored.

To accomplish the goals and avoid the non-goals, we have to consider the use cases and the query that satisfy them, and design roughly a table for each query. Even if two queries with different search parameters must retrieve the same data, we will create two different tables with the query parameters as the partition keys. We will end up with multiple tables containing the same data, but for everything we have said so far it is ok: they guarantee good performance on data retrieval.

With these guidelines, we designed the schema shown in Figure 10.

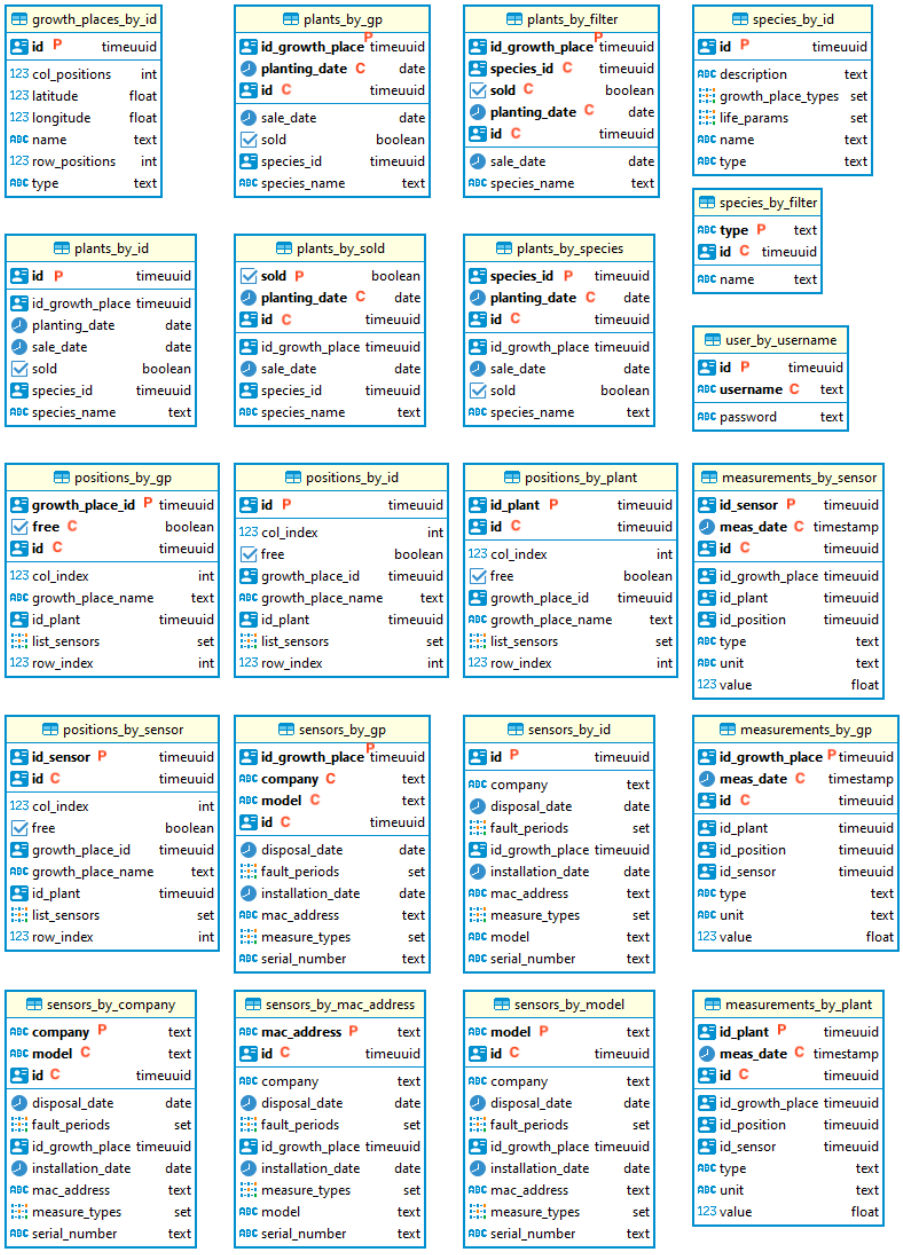

Figure 10: Cassandra schema for the Plant Nursery 4.0 project. Fields marked with P belongs to the partition key of the column family, while those marked with C belongs to the clustering key.

Cassandra does not provide an autogeneration process for surrogated keys such as the "autoincrement" mechanisms of relational databases, hence we used a TimeUUID field as the id of the entities. Cassandra guarantees that TimeUUID can be safely used as the natural key since the risk of conflicts is negligible.

As a final clarification about the design process, let us consider the *sensors* tables. The use case diagram in Figure 7 contemplates 4 different ways to retrieve sensors: by MAC address (that will retrieve a unique sensor), by growth place, by company, and by model. To these, we must add the search by sensor id. The 3 filtering fields (growth place, company, model) can be also combined together to perform multiple filtering.

When retrieving information about a sensor, we want to know its personal information (the attributes in the Sensor entity of the relational data model in Figure 6), its fault periods, and the growth place in which it is placed. Even if the fault periods and the growth place are entities separated from the sensor, we have to store them in the same column family to retrieve all the data in a single query. Hence we modeled 5 column families for the sensor entity, all containing the same info, but having different keys.

If we had followed rigorously the rule "one table for each query", we would have to create too many tables, one for each possible combination of parameters. It is clear that doing this for all the entities would result in a database with a huge number of tables, requiring a lot of code and an unsustainable number of writes: write operations are cheap but not free, and adding an entity would have meant insert a row in each of the column family associated with that.

Looking closer to the 5 column families created for the sensor entity, we will notice that they are sufficient to cover all the use cases with those partition keys. Indeed:

- *sensors_by_model* satisfy the query "retrieve all by model".
- *sensors_by_company* can satisfy the queries "retrieve all by company and model" (using the complete partition key) and "retrieve all by company" (using only the first part of the partition key). The latter will involve reading rows from multiple partitions.
- *sensors_by_gp* can satisfy the queries "retrieve all by growth place", "retrieve all by growth place and company", and "retrieve all by growth place and company and model".
- *sensors_by_mac_address* can satisfy the query "retrieve a sensor by MAC address".
- *sensors_by_id* can satisfy the query "retrieve a sensor by id".

Adding a sensor to the database will cost 5 write operations, but read operations are optimized.

The same reasoning has been applied to all the domain model entities in order to generate the database schema.

To satisfy queries like "retrieve all species with name starting with" we had to create a special type of Cassandra index called SASI index [5], otherwise the database would have not accepted that type of filtering.

7.3 Implementation and Faced Issues

Since we wanted to realize an application with the same functionalities as the one presented in Section 6 but with Cassandra and Jakarta NoSQL, we organized it using the same modular schema shown in Figure 9. Starting from that, we thought to what and which classes we could reuse from the relational implementation.

Endpoints classes could be moved in this new solution as they were since the business logic was implemented inside the controllers and the endpoint only acted as an interface between clients and the backend. The same holds from DTOs: no matter how entities mapped into the database are made, the mapper class will take care of the conversion to and from DTOs. In conclusion, we could use the same REST services, with the same parameters and the same format for the exchanged messages: a client or a frontend application would be unaware of the change from a SQL to a NoSQL implementation. Indeed, we could reuse the same Postman calls to test the application. However, Jakarta NoSQL has played no role in this transition, it is only thanks to the modularity of the software architecture. Unfortunately, no other classes could be brought from the first to this second implementation.

Classes inside the **package model** are totally different in the NoSQL solution and they have to use the Jakarta NoSQL annotations. Their development process is opposite to that of the model classes from the relational implementation. In the latter, we started from the domain model to implement the classes, we annotated them with JPA and the JPA provider (Hibernate) generated the database schema. Here we started from the database schema design (presented in the previous section) and we implemented a class for each database table, annotating them with Jakarta NoSQL considering table properties (keys, column types). Jakarta NoSQL does not provide a mechanism to autogenerate the schema from the annotated classes, it is the classes that need to be annotated based on the schema.

We encountered some other issues implementing the classes of the entities. According to the documentation, Jakarta NoSQL provides an automatic conversion to database text fields for class attributes of type Enum. However, when using collections of Enum errors occurred. Jakarta NoSQL also has the *@Embeddable* annotation. Class attributes of *@Embeddable* types annotated with *@Column* are mapped into a JSON string when using columnar databases (in the sense that they use the column implementation of the Mapping API, see Section 3) like Cassandra. This works as expected when dealing with single attributes, but it throws errors when using collections of *@Embeddable*. To solve these two similar errors we decided to manually map the collections of enum and collections of objects into collections of strings.

Since all entities have a TimeUUID as id and need the identity methods

like equals, all class entities inherit from a *BaseEntity* class (annotated with *@MappedSuperclass*).

```
@MappedSuperclass
public abstract class BaseEntity {
  @Id("id")
  private UUID id;

  public UUID getId() {
    return id;
  }

  public void setId(UUID id) {
    this.id = id;
  }

  // equals, hashCode and toString methods
}
```

To save code, entity classes modeling the same domain entity inherit from a common interface exponing the setter and getter methods. This allows managing different entities as the same container of data and using a single Mapper for multiple class entities.

To clarify the points presented so far, let us consider a concrete example from our implementation. The **SensorByGrowthPlace** class models the *sensor_by_gp* database table:

```
@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
}
```

As we can see, it inherits from BaseEntity and implements the Sensor interface (that we will not show). Hence it will have an additional id field annotated with *@Id* and must implement all the getters and setters methods of Sensor. The *@Id* fields are four, there is no way to indicate if they form a partition key or a clustering key: this will be inferred from Jakarta NoSQL when connecting to the Cassandra database.

Concerning the two issues presented above, we can observe that measureTypes and faultPeriods, that they should have been a Set of Enum and a Set of FaultPeriod objects respectively, are defined as Set of strings instead.

For completeness, the code snippet below shows the **SensorMapper**.

```
@Dependent
public class SensorMapper {

  public SensorDto toDto(Sensor entity) {
    SensorDto dto = new SensorDto();
    dto.setId(entity.getId());
    dto.setCompany(entity.getCompany());
    dto.setModel(entity.getModel());
    dto.setSerialNumber(entity.getSerialNumber());
    dto.setMacAddress(entity.getMacAddress());
    dto.setDisposalDate(entity.getDisposalDate());
    dto.setInstallationDate(entity.getInstallationDate());
    dto.setIdGrowthPlace(entity.getIdGrowthPlace());

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

    Jsonb jsonb = JsonbBuilder.create();
    Set<FaultPeriodDto> faultPeriods =
        entity.getFaultPeriods().stream()
      .map(s -> jsonb.fromJson(s,
          FaultPeriodDto.class)).collect(Collectors.toSet());
    dto.setFaultPeriods(faultPeriods);

    return dto;
  }

  public <T extends Sensor> List<SensorDto> toDto(List<T> entities) {
    return
        entities.stream().map(this::toDto).collect(Collectors.toList());
  }

  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());
    entity.setModel(dto.getModel());
```

```
    entity.setSerialNumber(dto.getSerialNumber());
    entity.setMacAddress(dto.getMacAddress());
    entity.setInstallationDate(dto.getInstallationDate());
    entity.setDisposalDate(dto.getDisposalDate());
    entity.setIdGrowthPlace(dto.getIdGrowthPlace());

    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);
  }
}
```

Through the Sensor interface, the Mapper can transparently handle all the sensor-related entities and convert them to/from DTOs. We can also observe that enumerations and embedded objects are converted to/from JSON strings.

Following the usual practice, we implemented a **DAO** for each entity class. SQL and JPA gave us some tricks to handle queries with multiple optional parameters using a single SQL statement and thus saving a lot of code, both in DAO classes and Controllers. The Mapping API of Jakarta NoSQL provides a functional API with limited features, hence we had to manually implement a method for each of the possible combinations of parameters for each of the possible queries, ending up with a lot of if-else statements, classes, and methods.

For example, for the DAO managing the **SensorByGrowthPlace** entity we have:

```
@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) {
    ColumnQuery query = ColumnQuery.select()
                        .from(TABLE_NAME)
                        .where("id_growth_place").eq(idGrowthPlace)
                        .and("company").eq(company)
                        .build();
    Stream<SensorByGrowthPlace> sensors = columnTemplate.select(query);

    return sensors.collect(Collectors.toList());
  }

  public List<SensorByGrowthPlace>
      getSensorsByGpAndCompanyAndModel(UUID idGrowthPlace, String
      company, String model) {
    ColumnQuery query = ColumnQuery.select()
                        .from(TABLE_NAME)
                        .where("id_growth_place").eq(idGrowthPlace)
                        .and("company").eq(company)
                        .and("model").eq(model)
                        .build();
    Stream<SensorByGrowthPlace> sensors = columnTemplate.select(query);

    return sensors.collect(Collectors.toList());
  }
}
```

A further problem we encountered is that even though the Mapping API accept also textual queries, these queries must comply with the Mapping API query syntax, not allowing database-specific operators. This is a big limit when using Cassandra since we could not use the ALLOW FILTERING keyword in the queries. Without going into details, a query containing ALLOW FILTERING can perform filtering operations also on fields not belonging to the partition key. It allows also filtering operations on non-ordered partial partition key fields. Being able to use it means simplifying the database schema reducing the data duplication (and the code duplication).

We will not show the code from **Controller** classes, we'll just say that even if the business logic remains the same as the relational solution, the duplication of entity classes and DAOs forced us to rewrite them from scratch. We had to handle the multiple writes on multiple tables for each insert, update and delete operation. Furthermore, in NoSQL, there is no explicit concept of transaction

and consistency as in SQL databases: Controllers had to take care also of these aspects, sometimes without much success. For data retrieval, Controllers must perform all the checks on the input parameters to determine which DAO to use and which method to execute.

8 Conclusions

We have seen that the migration from a SQL database to a NoSQL database is not painless using Jakarta NoSQL. On the contrary, very few elements can be reused and a big part of the project must be redesigned and rewritten. The cause can be found in the different design processes of relational and NoSQL solutions: while the Domain-Driven Design fits well with a traditional application, leading to a standardized design process and straightforward implementation, the Query-Driven Design with a NoSQL requires a certain experience with the chosen database and must consider the peculiarities of it along with the use cases.

Jakarta NoSQL provides a unified set of APIs common to all the NoSQL databases. In principle, the main advantage is that developers do not have to learn database-specific APIs to work with different databases. In practice, also the migration from a NoSQL database to another cannot be done in a transparent way. A document-oriented database is really different from a column-oriented one, as well as a column-oriented database can be different from a wide column store like Cassandra. Jakarta NoSQL does not provide a way to auto-generate the database schema from Java classes: we have to design the database entities and create Java classes based on them. The domain model cannot be reused without a considerable loss of performance, since we have not exploited and considered the new database peculiarities during the design process. A new domain model means new DAOs, Mappers and Controllers, hence we would have to rewrite the application almost entirely, as we did for our second solution.

The Mapping API, the highest level API offered by Jakarta NoSQL, provides useful methods to transparently query many different types of databases but does not offer the opportunity to use database-specific features.

In conclusion, Jakarta NoSQL is an interesting project that could have much to say in the future. However, it is still at an embryonal stage and currently does not provide a real abstraction API for NoSQL Java applications. Indeed, it does not offer too many advantages over using database-specific drivers.

It could be interesting to compare the performance of our Jakarta NoSQL implementation with another that uses directly the Cassandra drivers, to see how much overhead Jakarta NoSQL adds.

Concerning the Plant Nursery 4.0 system, we are confident that it can be a good starting point for a real-world application, and would be very interesting to see the whole system implemented and functioning in a real nursery.

References

[1] Amazon web services - what is nosql? https://aws.amazon.com/nosql/.

[2] Cassandra github repository. https://github.com/apache/cassandra.

[3] Cassandra overview. https://cassandra.apache.org/_/cassandra-basics.html.

[4] Datastax - basic rules of cassandra data modeling. https://www.datastax.com/blog/basic-rules-cassandra-data-modeling.

[5] Datastax - using a sstable attached secondary index (sasi). https://docs.datastax.com/en/dse/5.1/cql/cql/cql_using/useSASIIndex.html.

[6] Jakarta nosql documentation. http://www.jnosql.org/spec/.

[7] Mongodb - what is nosql? https://www.mongodb.com/en-us/nosql-explained.

[8] Plant nursery 4.0 repository. https://github.com/jasonravagli/plant-nursery-4.0.

[9] Vannucci piante. https://https://www.vannuccipiante.it/.

[10] What exactly is a wide column store? https://newbedev.com/what-exactly-is-a-wide-column-store.