Plant Nursery 4.0

A comparison between the design and development of a Jakarta EE web application and a Jakarta NoSQL one

Jason Ravagli ***Iacopo Erpichini***

Software Architecture and Metodologies
University of Florence

June 2021

Table of Contents

Introduction

Introduction

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?

- 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 - Plant Nursery 4.0

Project Goal

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

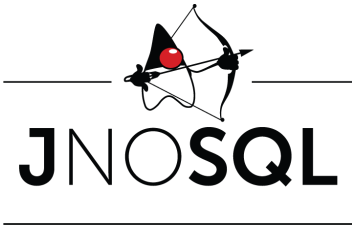
Using IoT and Industry 4.0 principles to innovate a plant nursery company

Jakarta NoSQL

Jakarta NoSQL

From the official documentation:

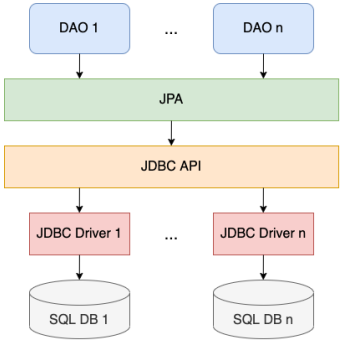
> *Jakarta NoSQL is a Java framework that streamlines the integration of Java applications with NoSQL databases. It defines a set of APIs and provides a standard implementation for most NoSQL databases. This clearly helps to achieve very low application coupling with the underlying NoSQL technologies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



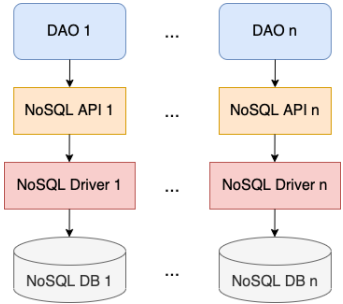
Jakarta NoSQL - Premises

- **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

Jakarta NoSQL - Premises



SQL Application



NoSQL Application

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

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 {
    @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 ways of querying the database: **Template classes** and the **Repository interface**
- Using Template classes to access a NoSQL database is similar to access a relational one through JPA
- They provide convenient methods to store, delete and retrieve data

Mapping API - Insert and Update

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

Person personInserted = template.insert(person);

// Update person fields

template.update(person);
```

- The DocumentTemplate object can be instantiated using the CDI injection
- Must be defined a bean producer that connects to the database and create an EntityManager object

Mapping API - Retrieve and Delete

```
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 **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

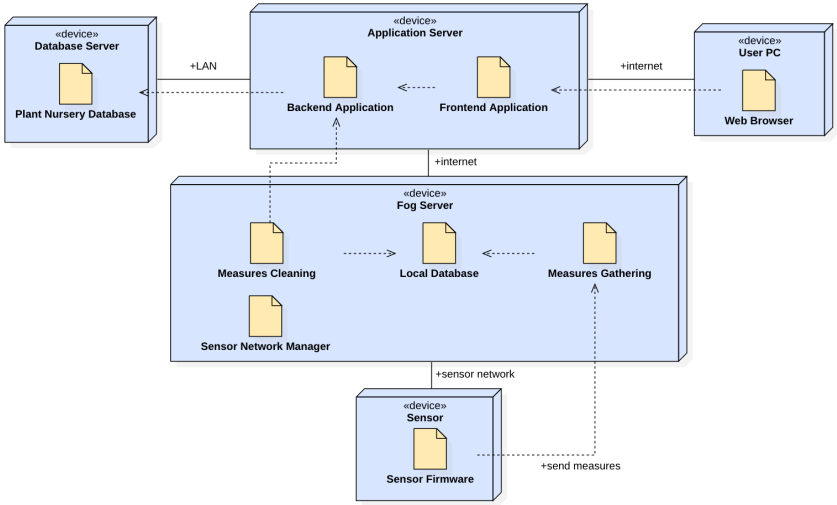
A management system for plant production inspired by industry 4.0

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

Our Hypothetical Nursery

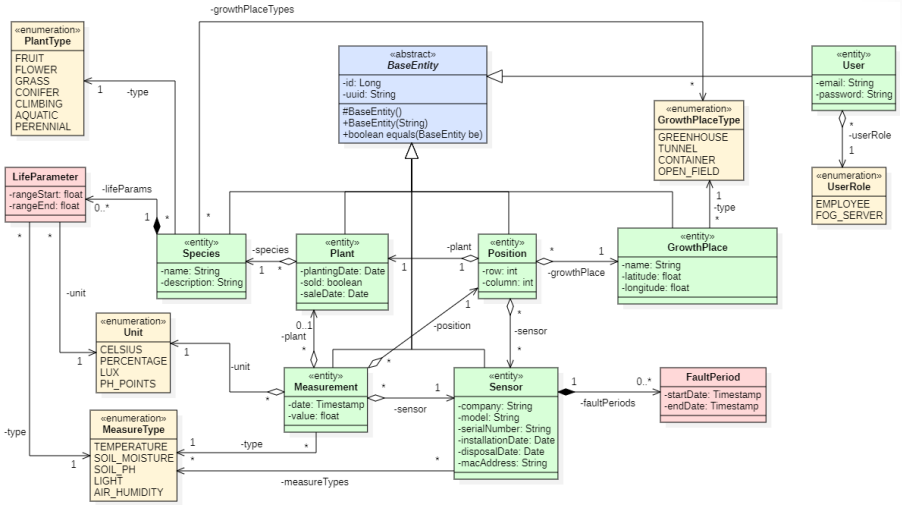
- Owns lands in a wide geographical area
- Has 4 different growth environments
- Uses **sensors** to monitor environmental parameters

Deployment Diagram



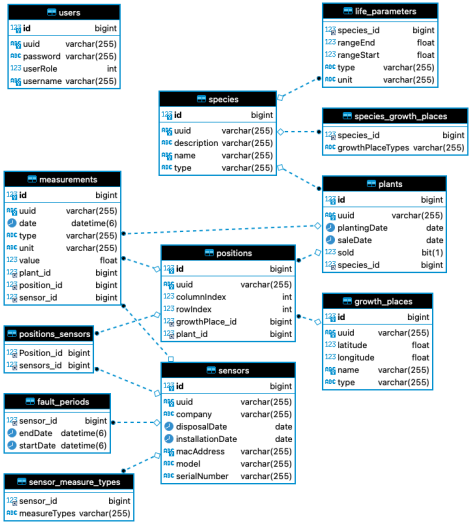
Solution with SQL Database

Domain model

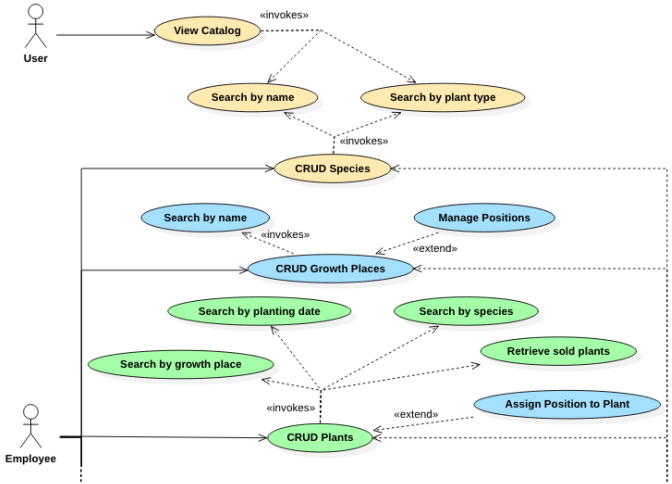


Entity Relationship Diagram

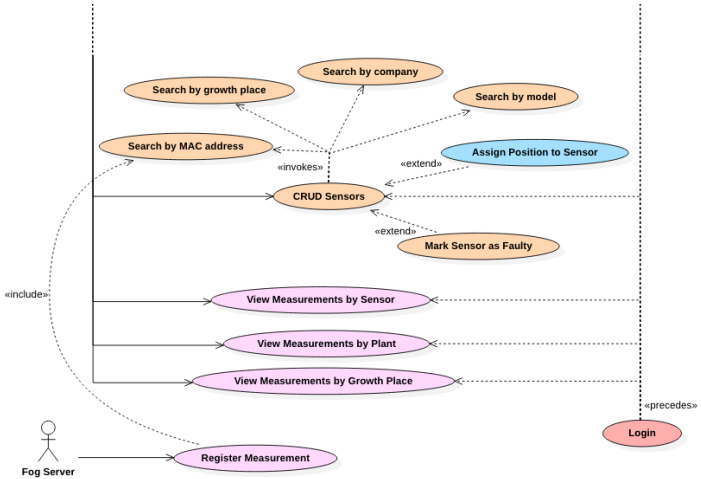
- Green classes are annotated with *@Entity*
- Red classes are annotated with *@Embeddable*, becoming *@Embedded* fields
- BaseEntity is a *@MappedSuperclass* entity



Use Cases - 1


User
View Catalog
«invokes»
Search by name
Search by plant type
«invokes»
CRUD Species
Search by name
Manage Positions
«invokes»
«extend»
CRUD Growth Places
Search by planting date
Search by species
Search by growth place
Retrieve sold plants
«invokes»
Assign Position to Plant
«extend»
Employee
CRUD Plants

Use Cases - 2


Search by company
Search by growth place
Search by model
Search by MAC address
Assign Position to Sensor
«invokes»
«extend»
CRUD Sensors
«extend»
Mark Sensor as Faulty
«include»
View Measurements by Sensor
View Measurements by Plant
View Measurements by Growth Place
«precedes»
Login
Fog Server
Register Measurement

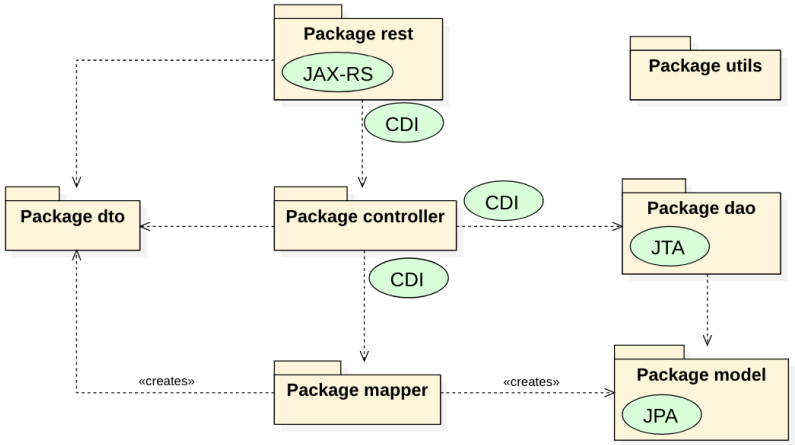
Implementation

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**

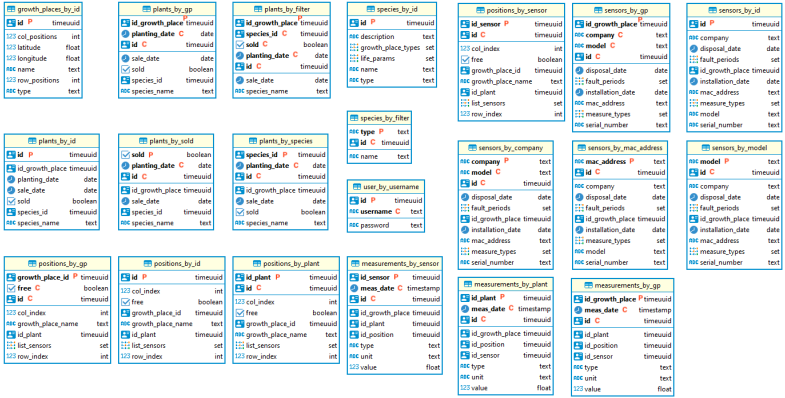
Non-Goals

- Minimize the number of writes
- Minimize the data duplication

Goals

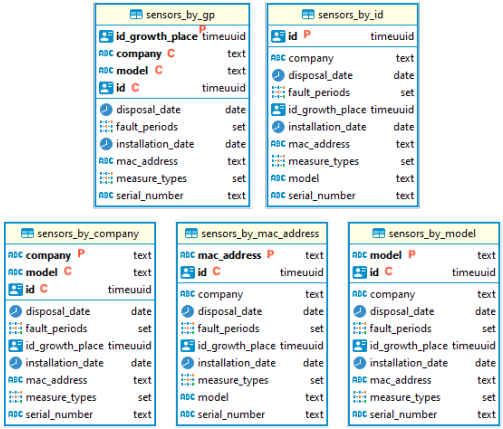
- Spread data evenly around the cluster
- Minimize the number of partition reads

Cassandra - Schema



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

Cassandra - Sensors Tables



P: Partition key - C: Clustering key

Implementation

- We realized a second application with the same functionalities of the relational one
- Organized using the same modular schema for the packages
- Need to determine which classes could be kept and which should be rewritten

Packages Reused

- REST Endpoints
- DTOs

Packages Reimplemented

- Model
- DAOs
- Mapper
- Controller

Implementation - Faced Issues

- Jakarta NoSQL does not autogenerate 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

Implementation - Code Snippets

```
@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
}
```

Implementation - Code Snippets

```
@Dependent
public class SensorMapper {

  public SensorDto toDto(Sensor entity) {
    SensorDto dto = new SensorDto();
    dto.setId(entity.getId());
    dto.setCompany(entity.getCompany());
    // set simple fields

    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;
  }

  // toEntity method
}
```

Conclusions

Conclusions

- The **migration from a SQL to a NoSQL database is not painless** using Jakarta NoSQL
  - **A big part of the project must be redesigned** and rewritten
  - Different technologies and design processes lead to incompatible architectures
- Also **the migration from a NoSQL database to another has problems**
  - NoSQL databases are really different from each other
  - Jakarta NoSQL does not autogenerate the schema
  - Entity classes must be rewritten (as well as Mappers and DAOs)
- Database-specific query operators cannot be used with Mapping API

Conclusions

- Jakarta 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

NoSQL Technologies

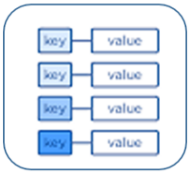
NoSQL - Not only SQL

- 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