
# API ASSET JANUS

Technologies for Big Data Management

**Students**
Niccolò Francioni
Luca Mozzoni
Damiano Pasquini

**Supervisor**
Prof. Massimo Callisto De Donato

**March 2023**

# Project overview

**Asset management prototype tool** based on **JanusGraph** database, able to run **CRUD operations** with focus on **search functionalities** over digital twin properties.




# JanusGraph | Features

**JanusGraph** is a **graph-oriented** database:

- Uses graph structures with **nodes**, **edges** and **properties**
- **Graph Theory** concepts are applicable
- **Query language** is **Gremlin**, which chains together traversal operators to form **path-like expressions**
  - **e.g.** "*from node A traverse to node B and return its outgoing nodes*"
- Storage and index backends are **pluggable**
  - In our case, JanusGraph + Cassandra for storage

# JanusGraph | Storage backend solutions

- JanusGraph is distributed with 3 supporting backends: **Apache Cassandra, Apache HBase and Oracle Berkeley DB**.
- The choice of backend solution falls according to specific implementation needs (e.g. Availability over Consistency and vice versa).
- In our case we choose Cassandra but, since our project is a prototype implementation we didn’t have any strict constraints to be met.

# JanusGraph | Guarantees*

- When used with Cassandra, JanusGraph transactions are **not** ACID
  - simulating them is **costly** in terms of overhead
- Cassandra values **availability** and **partition tolerance** over **consistency** (CAP Theorem)
- Cassandra is an **eventually consistent** storage system
  - Tradeoff to achieve **high availability**
  - Involves some read and write **latencies**

***Guarantees made by JanusGraph depend on the storage backend**


# JanusGraph | Schema and Data Modeling

- Each graph has a schema defined **explicitly** or **implicitly**
  - keys can be defined for **vertex/edge labels** and **property keys**
- A schema can be **evolved** over time
  - it does **not** slow down query answering
  - it does **not** require database downtime
- Schemas allow to set property and connection **constraints**
  - bound properties to specific vertex/edge labels
  - define which vertex labels can be connected by an edge

# JanusGraph | Gremlin Query Language

- Gremlin is used to **retrieve and modify** data in the graph
  - functional language
  - path-oriented language
- Gremlin is the graph traversal language of **Apache TinkerPop**
  - every Gremlin traversal is composed of a sequence of steps
  - a step perform an **atomic operation** on the data stream

# JanusGraph | Transactions

- Almost every interactions with JanusGraph is associated with transactions.
- Transactions are **safe for concurrent use** by multiple threads.
- Transactions are **not necessarily ACID** due to the underlying storage systems like Cassandra and HBase that do not provide such characteristics.

# JanusGraph | Transactions

- Each thread **automatically opens** its own transaction
- Transactions are handled via ***commit()*** and ***rollback()*** methods
- **Thread-independent transactions** are also supported
  - multiple threads work on the same transaction
  - useful for concurrent algorithms or nested transactions
- JanusGraph automatically tries to rerun **failed transactions**
  - the number of attempts and delay are configurable

# JanusGraph | Indexes

- Indexes are used to improve queries performance.
- JanusGraph currently supports:
  - Two types of indexes **graph indexes** and **vertex indexes**.
    - **Composite indexes** and **mixed indexes**.

# JanusGraph | Graph Indexes

- Indexes are used to **speed up** the **random access** to the entire database.
- The goal is to get at the starting point of a query as **efficiently as possible** without having to first search the entire graph.
- Graph indexes should be established for **property keys** or **combination keys** that will be used regularly in the queries.

# JanusGraph | Vertex Indexes

- Vertex indexes are associated with **vertices**.
- Typically used when the number of **incident edges** on a given vertex becomes significantly large such that it can impact on performance.
- While graph indexes are usually created at graph schema creation, vertex indexes are created as the need arises.

# JanusGraph | Composite Indexes

- Composite indexes can be used to speed up queries where an **exact match** with the value for the given property key is sufficient.

```
g.V().has('type','class_room')
```

- For example the query above could take advantage of a composite index as we are only looking for exact matches where the value associated with ***type*** key is the value ***class_room***.

# JanusGraph | Mixed indexes

- For queries that require more than a simple test for equality, the creation of mixed indexes is needed.
- To use them an **indexing backend** is needed. The ones supported by JanusGraph are:
  - **Apache Solr**
  - **Apache Lucene**
  - **Elasticsearch**

# Technical implementation | Objectives

- Implement an asset management tool using **JanusGraph** able to:
  - run **CRUD operations** with focus on **search functionalities** through **Gremlin** queries implemented in **Java**;
  - configure JanusGraph to interact with **Apache Cassandra**;
  - demonstrate functionalities using tools like **Postman**.

# Technical implementation | Technologies

- JanusGraph 0.6.2
  - as distributed **graph database**;
- Apache Cassandra 3.11.10
  - as **storage backend** for JanusGraph;
- Java 8
  - as language to implement **Gremlin Queries**;




# Technical implementation | Cassandra

- Distributed NOSQL column-oriented database
- Easily **horizontally scalable** when needed
- A node represents a single instance of Cassandra
- Uses **CQL** as query language to store data in tables

# Technical implementation | Spring Boot

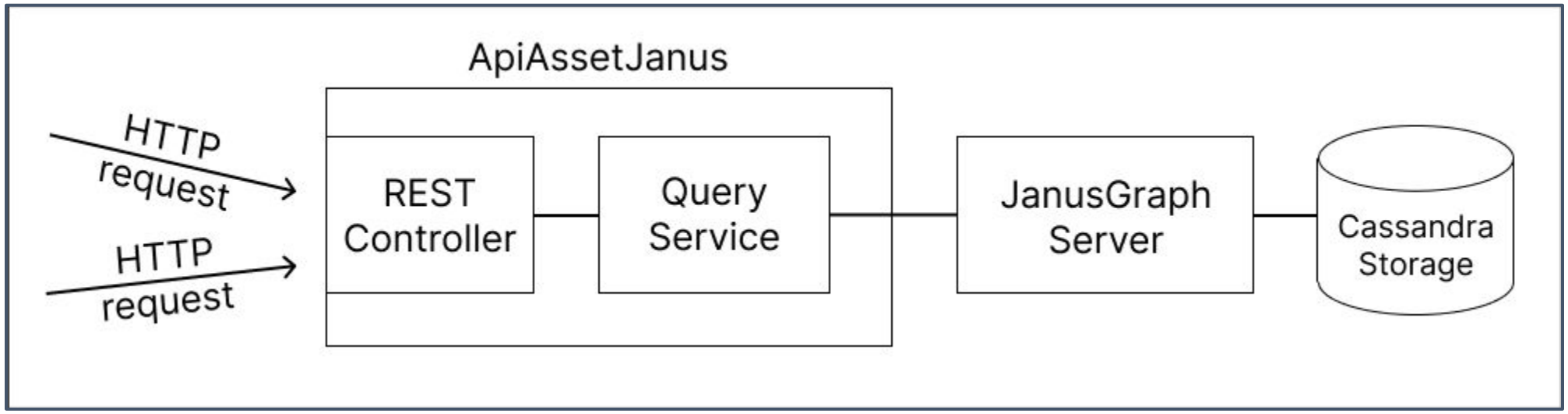
- Spring Boot is a Java framework to develop web applications and microservices.
- Popular framework because it is focused on speed, simplicity and productivity, with a lot of the out of the box solutions.
- It has been used in the project to develop REST-API to query the graph database.

# Technical implementation | Setup

- Installation of **JanusGraph** and **Cassandra**
- Setting up JanusGraph to work with Cassandra
  - step-by-step guide provided in **README**
- Creation of a Maven application with the following **dependencies**
  - **janusgraph-driver**
  - **gremlin-driver**
- Query **implementation** and **testing** with Postman

# Technical implementation | Architecture

- **RESTController**: handles the incoming requests to the **QueryService**
- **QueryService:**
  - implements the code for executing queries
  - speaks directly to the JanusGraph server



# Technical implementation | Queries

**Gremlin-Java** is considered the **reference implementation** of Gremlin

- allows interaction with a **Gremlin Server** (e.g. JanusGraph Server)
- implements Gremlin within the **Java** language (**same syntax**)
- allows **application-level** and **database-level** code to be written in Java

```
public List<Map<Object, Object>> getVerticesByLabel(String label) {
        return this.g.V().hasLabel(label).elementMap().toList();
}
```

# Achieved results

- Our software is capable of interacting with JanusGraph for
  - executing CRUD operations
  - executing complex search operations
- JanusGraph interacts with Cassandra to persist data
- The tool exposes REST API endpoints

https://github.com/TBDM-Project/JanusGraph-API-example

# Possible future improvements

- Implementation of one of the compatible **index backends** to improve search performances (e.g. geo, numeric range and full-text search)
  - Elasticsearch
  - Apache Solr
  - Apache Lucene
- Configure the tool to work in a **distributed environment**
- Integrate the tool with **other services**

# Thank you
# for your attention!