Semantic Event Detection in SmartChainDB

dpdodiya@ncsu.edu, rkrajpal@ncsu.edu, ssharm34@ncsu.edu, vvivek@ncsu.edu

December 17, 2019

Contents

1 Introduction

Product manufacturing life cycle starts with Request for Quote (RFQ) process. In the RFQ process, a Contract Manufacturer (CM) is chosen, and the price, delivery, and other terms are negotiated. In a marketplace, the manufacturers listen to the RFQs they are capable of manufacturing. The semantics of such RFQs could be complex and implicit. This could lead to some manufacturers not receiving RFQs even though they possess the capability to manufacture the requested product.

In this project, we attempt to map such complex and vague RFQ events into an appropriate publish/subscribe event detection model. Kafka would be used for publish/subscribe mode. Event publisher would be a marketplace events from SmartChainDB and event consumer would be manufacturers listening to Kafka topics. Terms and relationships of the requests would be captured in an ontology.

2 Motivation and Background

SmartChainDB based marketplace has many advantages including real time notification emitting from SmartChainDB chain. These notifications can be leveraged through Kafka to provide real time notification to manufacturers using stream processing application like Kafka. "Kafka is used for real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies" [5]. We leverage Kafka's topic model to create independent streams for particular capabilities. However, it is a challenge to efficiently map quotes (Request for Quote event generated from SmartChainDB) to appropriate manufacturing capabilities (topics). We can use Kafka in a way so that a manufacturer (consumer) can listen to RFQs corresponding to their capabilities (topics). Since, many manufacturing capabilities can be semantically equivalent, we want to solve for two potential problems because of complex nature of RFQs.

1. A consumer should not receive same RFQ more than once, even if RFQ corresponds to multiple topics and consumer is subscribed to each of them.

2. A consumer should not miss out on an RFQ for a semantically equivalent capability.

To tackle the above problems, we propose a semantic layer processing wherein we make use of manufacturing capabilities ontology. This semantic layer will query a graph database built on top of the ontology to come up with manufacturing capabilities that are equivalent to incoming RFQ. This is equivalent to saying that we find all Kafka topics that are semantically equivalent.
To keep the marketplace fair to all players, it is also necessary that an RFQ event is published to all topics at the same time. This ensures that all players have fair chance of bidding for the RFQ. We ensure that our architecture takes care of this requirement.

3 Related Work

Detecting semantic similarity for events is not a new problem however, its application for SmartChainDB seems to be a novel application. This is a different problem from regular semantic analysis because event handling depends on the RFQ and not just the event name or the source. We explored literature [1] and [2] to finalize our approach.

Another key architectural concerns for the system is handling load for data ingestion, [3] and [4] explain in detail on how to scale Kafka using the partitions.

We make use of manufacturing capability ontology as mentioned in Järvenpää et al. [7]. As expected from the project we will extend this ontology to include new classes and relationships.

4 Approach

Our approach to solve this problem involves stream processor, a real time data stream application and a database to store the ontologies. We can break down the work in 2 parts:

1. Maintaining Ontology
   In order to query an ontology efficiently, we need to load it in a graph database. In our applications, various capabilities in the ontology can be considered as classes and relations between those capabilities represent all possible derivations of the capability under consideration. Modern graph databases serve the purpose of converting an existing RDF file (ontology) into a graph.
   The database would provide an API to search all possible outcomes of any relation for any capability in the graph.

2. Subscribe various consumers to consume incoming Request For Quote (RFQ)
   RFQs can have different capabilities which can be considered as a topic in Kafka for consumers to subscribe to. Any new incoming RFQ generated from blockchain will go to one topic called *"Marketplace"*.
   A stream processing framework is required for this layer. The choice we make has been detailed in a later section. The stream processor makes a query to the database, where we store the graphical format of market capabilities ontology. Hence we will receive all semantically equivalent capabilities that translates directly to Kafka topics. Post this all the RFQ will be written to all the topics received hence all clients will get notified.

The Stream Processor then pushes the incoming RFQ to all the equivalent topics at the same time for the consumers to consume. This whole process can be considered as a set of request transformations viz. extracting set of capabilities from incoming RFQ, fetching similar topics from RDF Store (derivations from ontology) and then pushing into the new set of Kafka topics. The choice of Stream processor should satisfy parallel processing, fault-tolerance and scalability.
The project requirements mention that there is already a setup which prevents a consumer from consuming similar requests again, therefore handling duplicate consumption of RFQ for a particular consumer for the purposes of this project is not in the scope of work.
Below is a diagram which explains the approach discussed above. We can see from this figure that SmartChainDB (Producer 1) produces an RFQ event that is sent to Kafka topic "marketplace". A Spark consumer consumes event from this topic and does semantic equivalence processing. It finds that Topic 1, Topic 2 and Topic 3 are topics where this RFQ should be published based on the manufacturing capabilities in this RFQ. So, Spark processor pushes this RFQ to each of the three topics. The manufacturers subscribe to the topics according to their capabilities. Each of the manufactures get this RFQ, even if the RFQ did not explicitly mentioned each manufacturing capability.

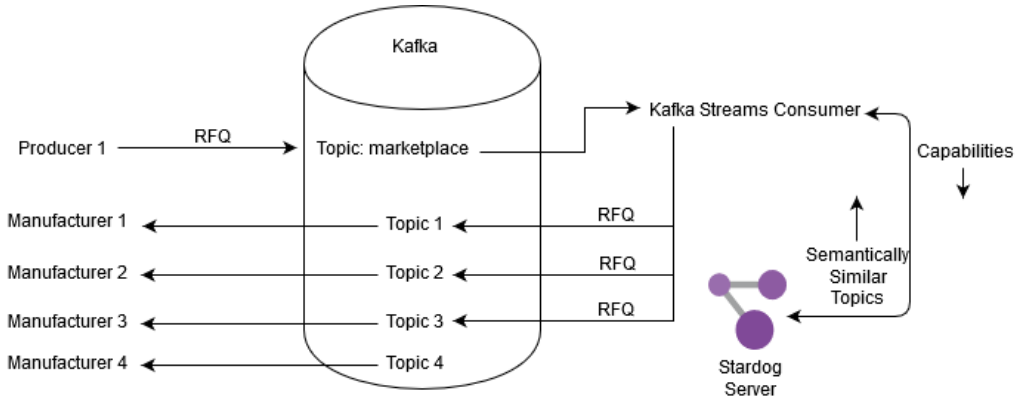

Figure 1: Application Flow Diagram

5 Design Decisions

5.1 Stream Processor

It is obvious that the Request for Quotes (RFQs) needs to be sent via Kafka to various topics but the processing of RFQ leading to publishing of messages can be done in various ways:

5.1.1 Apache Spark

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.
Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

5.1.2 Kafka Streams

Kafka Streams is a library for building streaming applications, specifically applications that transform input Kafka topics into output Kafka topics (or calls to external services, or updates to databases, or whatever). It lets you do this with concise code in a way that is distributed and fault-tolerant. Stream processing is a computer programming paradigm, equivalent to data-flow programming, event stream processing, and reactive programming, that allows some applications to more easily exploit a limited form of parallel processing.

5.1.3 Advantages of Kafka Stream over Spark Streaming

On detailed investigation and suitability to our project we chose Kafka Streaming over Spark Streaming due to following reasons

1. Kafka Streams a fully embedded library with no stream processing cluster—just Kafka and application. Hence we do not need a maintain a separate cluster for processing.

2. Processing model is fully integrated with the core abstractions Kafka provides to reduce the total number of moving pieces in a stream architecture.

3. Per event processing as compared to Spark Streaming that does micro batch processing.

5.2 Ontology Store

5.2.1 Neo4j

Neo4j is a native graph database platform that is built to store, query, analyze and manage highly connected data more efficiently than other databases. The Neo4j database is highly scalable both vertically and horizontally, without introducing data integrity or consistency issues using its Causal Clustering architecture, which now supports mulit-clustering. It is coded in graph-specific syntax to how it computes and executes queries all the way to how it persists connections in storage.Graph use cases are everywhere, and Neo4j covers them all.
Another advantage of Neo4j database is that it the only transactional database that combines performance and trustability in applications that bring data relationships to the fore: native graph storage, native graph processing, graph scalability, high availability, graph clustering, graphs in the cloud, graphs on Spark, built-in ETL, and integration support, plus Cypher, a powerful and expressive language for queries using vastly less code than SQL.

5.2.2 Stardog

Stardog is a knowledge graph platform backed by a performant, scalable graph database. Stardog is scalable, secure, and standards-based.It Encodes the meaning of the data alongside the data itself (persisting metadata, real-world context), Incorporates real-world rules and uses best-in-class inference to make implicit knowledge explicit, Provides a flexible view of data, serving multiple use cases with one model

5.2.3 Advantages of Stardog over Neo4j

During our evaluation, we compared both Stardog and Neo4j for our use cases. After careful consideration of features, we have decided to go ahead with Stardog platform for below listed reasons:

1. Neo4j is general purpose graph database whereas Stardog is developed specifically for ontology related usecases in mind.

2. Stardog has native RDF/OWL support and reasoning capabilities. Neo4j on the other hand requires third party plugin to support the same.

3. Stardog has excellent development tools like Stardog Studio which allows SPARQL and GraphQL queries to run out-of-the-box.

4. Stardog has native support for reasoning based queries which is missing in Neo4J.

5. Stardog has first party tutorials and SDKs with extensive documentation, whereas Neo4j has no official support. Availability of reference material is of critical concern for us.

When we first started the project, we thought to use Neo4j as our RDF store. However, quickly we realized that the native support that Stardog has for RDF/Ontology/Reasoning would be greatly beneficial to our development.

6 Implementation Details

Our Architecture has primarily 5 components as shown in Figure 2, This section of the report details on the implementation of each component.

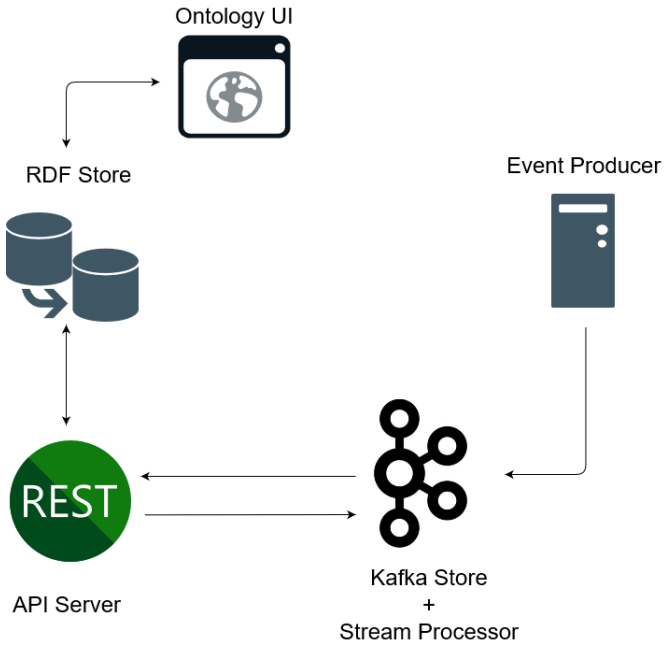

Figure 2: Architecture Diagram

6.1 Event Producer

This is a generic kafka producer that imitates emitting the RFQ's. All the messages land to the "master" topic.

6.2 Stream Processor

Since we have used Kafka Streams, we are doing per event processing semantics. As the "market place" topic recieves the message, it hits the REST Api with the capability.
The REST API returns the matching capabilities, and we then publish it to the individual topics (one each for capability)

Scaling The maximum parallelism at which application may run is bounded by the maximum number of stream tasks, which itself is determined by maximum number of partitions of the input topic(s) the application is reading from. For example, if input topic has 5 partitions, then we can run up to 5 applications instances. These instances will collaboratively process the topic's data.

If we have additional app instances (greater than) partitions of the input topic, the "excess" app instances will launch but remain idle, however, if one of the busy instances goes down, one of the idle instances will resume the former's work.

6.3 RDF Store

We have used Stardog platform as our backend to store ontology related information. Stardog has excellent native support for all common RDF related file formats. We have defined our own sample manufacturing related ontology in Turtle (.ttl) file format (Turtle file is in appendix).

Stardog platform offers Stardog Studio as Integrated Development Environment (IDE). We loaded the ontology into a new database using the ontology file. Our development workflow consisted of using Stardog Studio workspace to run queries and debug.

For production environment, we decided to host Stardog Server to cloud provider so that we can achieve desired scalability based on the workload. Currently the Stardog server is hosted on a dedicated DigitalOcean cloud instance. Stardog platform provides excellent REST API support and Software Development Kits (SDKs) for most programming platforms. The details of which are outlines below.

6.4 REST API

The Stardog RDF store can be accessed using multiple ways. During project development, we could simply use Stardog Studio to interact with server. While in production, we could use it programmatically via a SDK of Java, Python etc.

We went ahead with an implementation that is independent of underlying platform or programming language by exposing REST APIs. The REST API functions as semantic mapping layer between Kafka Stream Consumer. The Kafka Consumer would use the REST API as a layer to interact with the underlying ontology. An example of the API is given below.

```
API Request

GET /topics?term=Welding HTTP/1.1
Host: rtdm-flask-client.herokuapp.com
User-Agent: PostmanRuntime/7.15.2
Accept: */*
Cache-Control: no-cache
Postman-Token: 72e28dd9-f675-4631-90e5-d554519ec7c1,72e8c77b-1645-4c5a-b36d-591c70d836ae
Host: rtdm-flask-client.herokuapp.com
Accept-Encoding: gzip, deflate
Connection: keep-alive
cache-control: no-cache


API Response
[
    "Joining",
    "Welding",
```

```
    "Assembling"
]
```

6.5 Ontology UI

We have developed a web-application using Stardog.js (a library from Stardog providing various ontology related functionalities), ReactJS and D3.js for representing ontology hosted on Stardog server. This User Interface will be helpful in updating the current ontology ie. addition of new entities and current relations between them. This UI will update the ontology directly on the Stardog server without interfering the KafkaStreams consumer which will allow the KafkaStreams consumer to fetch records from latest ontology on the Stardog server.

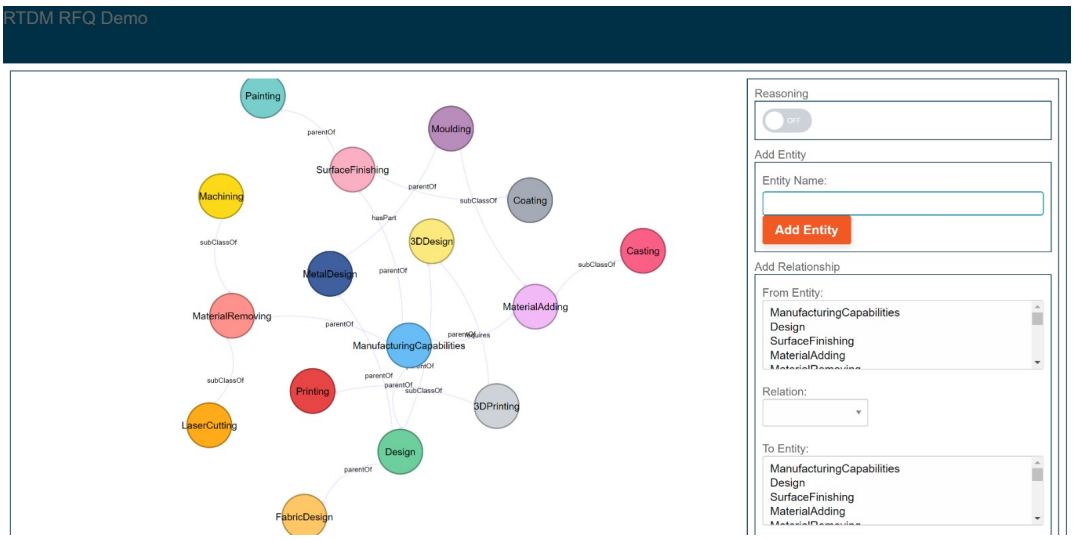

Figure 3: User Interface

7 Limitation

This project involves developing an application to solve a problem to the best of our knowledge. However, there are a few limitations which will be addressed further in this section.
As discussed earlier, we have used KafkaStreams consumer to extract semantic equivalence from incoming RFQ and publishing the RFQ in consideration to various other relevant capabilities. We achieve this by querying a REST API which provides relevant kafka topics as its response to publish to. For all topics received in form of response, we are duplicating the current RFQ and publishing it to those topics sequentially. The reason being KafkaStreams currently do not support sending messages to multiple topics in parallel. Once that feature arrives, we can use that feature to improve our existing implementation.
As far as the Ontology UI is concerned, we did not provide a medium to update existing relationships or add new relationships to our existing ontology.

8 Work Division

We primarily divided the work in two modules and two team members working on each of them.

1. Ravinder and Shantanu: Explore various stream processing frameworks and implement the the structure and code data flow.

2. Darpan and Vivek: Explore various ontology stores and design data storage, query and code Ontology UI

9 Conclusion

The project has reasonably achieved its primary goal of semantic event detection. The architecture has two main components: First is streaming part which is implemented using Kafka platform and another is related to ontology which is implemented using Stardog platform. All pieces of architecture work seamlessly as expected. For example, if a RFQ comes with 3D Printing as capability then it would get sent to consumers listening for 3D Printing, 3D Design and Printing topics by querying underlying Ontology store via REST API. In addition, we have a UI which can modify relations between ontology in real time.

In addition, we have put careful consideration into scalability of our project. As all the components are loosely coupled, they can be scaled out independently as needed to achieve real time processing capabilities.

We'd like to thank Dr. Kemafor Anyanwu Ogan for giving the opportunity to work in a completely new domain. This project has resulted into plenty of learning opportunities for us.

10 Future Work

We did identify certain ways in which our implementation could be refined. These ways are addressed further in this section.
As far as ontologies and their semantic equivalence are concerned, we have implemented them in Terse RDF Triple Language (Turtle) and used rule-based reasonings to derive semantic equivalence of capabilities provided in query. One of the future work is to provide an interface to update those rule-based reasons so as to make those rules more dynamic in nature.
Apart from this, KafkaStreams provided us an architectural advantage for processing streams based on single event utilization. We can refine the implementation to accommodate a fault-tolerant way to query semantic equivalence and publish messages in parallel (as discussed in the previous section).

References

[1] Semantic Complex Event Detection System of Express Delivery Business with Data Support from Multidimensional Space Xin Jing, Jing Zhang, Jun Huai Li

[2] Towards Detecting Semantic Events on Blockchains Abhisha Bhattacharyya, Rahul Iyer, and Kemafor Anyanwu

[3] How to choose the number of topics/partitions in a Kafka cluster? Jun Rao

[4] Should You Put Several Event Types in the Same Kafka Topic? Martin Kleppmann

[5] Kafka documentation (https://kafka.apache.org/)

[6] Neo4J documentation (https://neo4j.com/neo4j-graph-database/)

[7] The development of an ontology for describing the capabilities of manufacturing resources Eeva Järvenpää, Niko Siltala, Otto Hylli, Minna Lanz

11 Source Code

The source code of this project can be found at: https://github.com/darpandodiya/rtdm

12 Appendix

Listing 1: Sample Ontology

```
@prefix : <http://api.stardog.com/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix stardog: <tag:stardog:api:> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

:linkType a owl:ObjectProperty .

:parentOf rdfs:subPropertyOf :linkType ;
   rdfs:subPropertyOf :linkType ;
   owl:inverseOf :subClassOf .

:siblingOf rdfs:subPropertyOf :linkType .

:subClassOf rdfs:subPropertyOf :linkType ;
   rdfs:subPropertyOf :linkType ;
   owl:inverseOf :parentOf .

:isEquivalentTo rdfs:subPropertyOf :linkType .

:hasPart rdfs:subPropertyOf :linkType .

:requires a owl:ObjectProperty ;
   rdfs:subPropertyOf :linkType ;
   owl:inverseOf :isRequiredBy .

:isRequiredBy a owl:ObjectProperty ;
   rdfs:subPropertyOf :linkType ;
   owl:inverseOf :requires .

:checkSiblingOf a <tag:stardog:api:rule:SPARQLRule> ;
  <tag:stardog:api:rule:content> """
   IF {
          ?r a :Capability ;
         {
            ?f :parentOf ?r .
            ?f :parentOf ?o .
            filter ( ?r != ?o )
```

```
        }
        UNION
        {
           ?o :isEquivalentTo ?r .
           filter ( ?r != ?o )
        }
     }
  THEN {
           ?r :siblingOf ?o .
       } """ .

:checkSubClassOf a <tag:stardog:api:rule:SPARQLRule> ;
  <tag:stardog:api:rule:content> """
  IF {
        ?r a :Capability ;
        {
           ?f :parentOf ?r .
        }
     }
  THEN {
           ?r :subClassOf ?f .
       } """ .


:checkEquivalentOf a <tag:stardog:api:rule:SPARQLRule> ;
  <tag:stardog:api:rule:content> """
  IF {
        ?r a :Capability ;
        {
           ?f :requires ?r .
           ?r :subClassOf ?o .
           filter ( ?r != ?f )
        }
     }
  THEN {
           ?r :isEquivalentTo ?f .
       } """ .

<http://stardog.com/ManufacturingCapabilities> a :Capability ;
   :parentOf <http://stardog.com/Design> , <http://stardog.com/SurfaceFinishing>
        , <http://stardog.com/MaterialAdding>, <http://stardog.com/
       MaterialRemoving> ;
  <tag:stardog:api:name> "ManufacturingCapabilities" .

<http://stardog.com/Design> a :Capability ;
   :parentOf <http://stardog.com/3DDesign> , <http://stardog.com/FabricDesign> ,
        <http://stardog.com/MetalDesign> ;
  <tag:stardog:api:name> "Design" .

<http://stardog.com/SurfaceFinishing> a :Capability ;
   :parentOf <http://stardog.com/Coating> , <http://stardog.com/Painting>, <
       http://stardog.com/Varnishing> ;
  <tag:stardog:api:name> "SurfaceFinishing" .

<http://stardog.com/3DDesign> a :Capability ;
  <tag:stardog:api:name> "3DDesign" .

<http://stardog.com/FabricDesign> a :Capability ;
  <tag:stardog:api:name> "FabricDesign" .
```

```
<http://stardog.com/MetalDesign> a :Capability ;
    :hasPart <http://stardog.com/Moulding> ;
    <tag:stardog:api:name> "MetalDesign" .

<http://stardog.com/3DPrinting> a :Capability ;
    :subClassOf <http://stardog.com/Printing> ;
    :requires <http://stardog.com/3DDesign> ;
    <tag:stardog:api:name> "3DPrinting" .

<http://stardog.com/Coating> a :Capability ;
    <tag:stardog:api:name> "Coating" .

<http://stardog.com/Painting> a :Capability ;
    <tag:stardog:api:name> "Painting" .

<http://stardog.com/MaterialAdding> a :Capability ;
    <tag:stardog:api:name> "MaterialAdding" .

<http://stardog.com/Casting> a :Capability ;
    :subClassOf <http://stardog.com/MaterialAdding> ;
    <tag:stardog:api:name> "Casting" .

<http://stardog.com/Moulding> a :Capability ;
    :subClassOf <http://stardog.com/MaterialAdding> ;
    <tag:stardog:api:name> "Moulding" .

<http://stardog.com/MaterialRemoving> a :Capability ;
    <tag:stardog:api:name> "MaterialRemoving" .

<http://stardog.com/LaserCutting> a :Capability ;
    :subClassOf <http://stardog.com/MaterialRemoving> ;
    <tag:stardog:api:name> "LaserCutting" .

<http://stardog.com/Machining> a :Capability ;
    :subClassOf <http://stardog.com/MaterialRemoving> ;
    <tag:stardog:api:name> "Machining" .

<http://stardog.com/Printing> a :Capability ;
    <tag:stardog:api:name> "Printing" .
```