Contract No. H2020 – 636078

INFORMATION TECHNOLOGIES FOR SHIFT TO RAIL

D1.9 – Interoperability Framework Integrated Development Environment

Due date of deliverable: 31/10/2017

Actual submission date: 04/01/2018

Leader/Responsible of this Deliverable: INDRA

Reviewed: Y

Document status		
Revision	Date	Description
1	26/04/2016	First issue
2	15/06/2017	Includes Broker with managed Microservices
3	30/06/2017	Includes Common Query Engine
4	18/10/2017	Last Modifications
5	04/01/2018	Final Version after TMC Approval

Project funded from the European Union's Horizon 2020 research and innovation program		
Dissemination Level		
PU	Public	X
CO	Confidential, restricted under conditions set out in Model Grant Agreement	
CI	Classified, information as referred to in Commission Decision 2001/844/EC	

Start date of project: 01/05/2015 Duration: 36 months

REPORT CONTRIBUTORS

Name	Company	Details of Contribution
INDRA's Innovation Team	INDRA	Edition of the document.
UPC	Universitat Politecnica de Catalunya	Updates on additional tooling.
Trenitalia Architecture and Innovation	Trenitalia	Updates on additional tooling.
Maria Laura Trifiletti	RINA C-BE	Quality check

EXECUTIVE SUMMARY

This document describes the Interoperability Framework's "development environment", i.e. a secure, stable and expandable set of tools, frameworks and utilities used for:

- development of the Interoperability Framework technical demonstrator;
- proof-of-concept integration testing with other components of the IT2Rail project;
- execution of the overall IT2Rail overall pilot implementation;
- further development activities in the continuation of the full SHIFT2Rail IP4 program.

The document describes therefore *tooling* available for development, but it is not – in this respect – a description of the Interoperability Framework's *design* or *implementation.* The latter is documented in deliverable D1.8 - Proof-of-Concept Packaged Resolvers Full Features.

The elements of the development environment have been identified through a selection process based on previous results from another Artemis R&D project, SOFIA (Smart Objects For Intelligent Applications), participant partner's available tooling and qualitative and quantitative experimental tests performed on open source frameworks, in order to identify them according to the following criteria:

1. Support the design concepts and assumptions documented in the D7.10-Development Readiness Review Pack (FREL);
2. Limit development effort to the specific *research and innovation* contents of the IT2Rail project, selecting standard tooling for standard tasks such as managing connectivity, persistence, web service development and conventional data manipulation;
3. Consistently with the overall objectives of the IT2Rail Project, eliminate any dependency on proprietary or specialty technology, in order to allow for any alternate choice of implementation of the same specifications, choose tools available under open source licensing policies;
4. Select tooling widely known and used by participating partners in order to reduce to a minimum, or eliminate altogether, training and support requirements;
5. Conform to general Horizon 2020 / SHIFT2Rail regulations, the Grant Agreement and the Consortium Agreement.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AREL	**A**dditional **REL**ease
CloudMdsQL	**Cloud m**ulti-**d**ata **s**tore **Q**uery Engine
CQE	**C**loudMDS **Q**uery **E**ngine
CREL	**C**ore **REL**ease
FREL	**F**inal **REL**ease
IF	**I**nteroperability **F**ramework
IT	**I**nformation **T**echnology
XML	**E**xtensible **M**arkup **L**anguage
SCXML	**S**tate **C**hart **XML**
JSON	**J**ava **S**cript **O**bject **N**otation
JDBC	**J**ava **D**atabase **C**onnectivity
SQL	**S**tructured **Q**uery **L**anguage
API	**A**pplication **P**rogramming **I**nterface
UML	**U**nified **M**odelling **L**anguage
RDBMS	**R**elational **D**atabase **M**anagement **S**ystem
ODBC	**O**pen **D**atabase **C**onnectivity
IoT	**I**nternet **o**f **T**hing
SSAP	**S**mart **S**pace **A**ccess **P**rotocol
SDK	**S**oftware **D**evelopment **K**it
RTDB	**R**eal **T**ime **D**ata**b**ase
HDB	**H**istoric **D**ata**b**ase
CEP	**C**omplex **E**ngine **P**rocessor
POJO	**P**lain **O**ld **J**ava **O**bject
OWL	**O**ntology **W**eb **L**anguage
RDF	**R**esource **D**escription **F**ramework
SaaS	**S**oftware **a**s **a** **S**ervice
Paas	**P**latform **a**s **a** **S**ervice
SIB	**S**emantic **I**nformation **B**roker
KP	**K**nowledge **P**rocessor
SOA	**S**ervice **O**riented **A**rchitecture
SOFIA	**S**mart **O**bjects **F**or **I**ntelligent **A**pplications
TC	**T**ravel **C**ompanion
TS	**T**ravel **S**hopping
TSP	**T**ravel **S**ervice **P**rovider
WP	**W**ork **P**ackage

1. INTRODUCTION

1.1 OBJECTIVES

According to the Technical Annex of IT2Rail, the objective of the Interoperability Framework is to provide IT2Rail functional applications with a 'web of transportation data' abstraction of the distributed resources they need to operate. The abstraction is constructed by using semantic web technology open standards with the following objectives:

1) build a set of components that effectively insulate the applications from handling the "mechanics" of interoperability: semantic web service registry, semantic discovery engine and packaged resolvers;
2) provide a shared machine-readable, explicit and formal description of the transportation domain's knowledge and its computational constraints: ontology specification and registry;
3) mask underlying disparities of data formats and protocols representing the common domain facts and events by providing for their unambiguous interpretation through the application of the shared semantics constraints: semantic query and aggregation engine. By eliminating the need for further, centrally governed convergence on a single prescriptive data format or protocol, the Interoperability Framework allows for:
    - transportation service providers to join the 'web of transportation data' autonomously, with minimal centralised coordination, contributing to an ecosystem of advanced networked travel applications and services whose evolution is driven by market forces;
    - opening up of the domain to participation by an extended range of technology and component suppliers as well as Travel Service providers;
    - acceleration in the adoption of existing technical specifications for interoperability, e.g. TAP-TSI, through the automation of data mappings and semantic interpretation across different data formats;
    - natural expansion of the IT2Rail Project scope into the full SHIFT2Rail IP4 objectives.

1.2 INPUTS

1.2.1 Inputs from deliverables

As inputs from other deliverables:

- D1.1 - IT2Rail Domain Ontology Specification and Repository. The deliverable consists of:
    - Module D1.1.1: specification document of the harmonised, integrated IT2Rail domain ontology;
    - Module D1.1.2: platform independent specification document of IT2Rail domain ontology repository;
    - Module D1.1.3: a software implementation of the ontology repository specification that provides the WP1 Proof-of-Concept packaged resolvers and other IT2Rail

work package functional applications with a shared, machine readable, formal description of the knowledge documented in the IT2Rail domain ontology specification, used to automate the interpretation of data exchanges between functional applications irrespective of syntactic formats or protocols.

- D1.2 – Semantic Web Services Registry. The deliverable consists of:
  - Module D2.1.2: the platform independent specification document of the Semantic Web Services Repository;
  - Module D2.1.1: a software implementation of the semantic web services repository specification that provides semantically annotated web service descriptors of functionality requested at run-time by WP1 Proof-of-Concept packaged resolvers and by other IT2Rail functional applications.
- D1.3 – Semantic Discovery Engine. The deliverable consists of:
  - Module D3.1.1: the platform independent specification document of the Semantic Discovery Engine;
  - Module D3.1.2: a software implementation of the Semantic Discovery Engine specification that provides WP1 Proof-of-Concept packaged resolvers and other IT2Rail functional applications with resource discovery and run-time binding irrespective of syntactic formats or protocols.
- D1.4 – Semantic Query and Aggregation Engine. The deliverable consists of:
  - Module D3.4.1: the platform independent specification document of the Semantic Query and Aggregation Engine;
  - Module D3.4.2: a software implementation of the Semantic Query and Aggregation Engine specification that implements distributed query capabilities on the web of transportation data.
- D1.5 – Interoperability Framework Integrated Development Environment. The deliverable consists of:
  - Module D3.5.1: the platform independent specification document of the Interoperability Framework Integrated Development Environment;
  - Module D3.5.2: a secure, stable and expandable development environment built to selected standard technology resources and enhanced by a set of ad hoc utilities to be based for development, Proof-of-Concept integration testing, support for the IT2Rail pilot activities, and further development activities within the full SHIFT2Rail IP4 programme.

Although the due date of all of them is M36 (22 months later from the current one), we can consider these deliverables as artefacts that will improve the Interoperability Framework that represents the current D1.9.

1.2.2 Inputs from partner's contributions

The development of the Interoperability Framework doesn't start from scratch. We have as software inputs:

- SOFIA2, partner provider "INDRA": Coming from the former SOFIA Artemis project (Smart Objects For Intelligent Applications,), SOFIA2 is an enhanced version successfully tested and deployed in real scenarios. This version has been rebuilt based

on the same semantic model smart M3 but being adapted to work properly and with robustness under real conditions. Indra will provide its SOFIA2 cloud lab for the free research and use with no commercial purposes;

- Common Query Engine, partner provider "UNIVERSITAT POLITECNICA DE CATALUNYA": provides a software infrastructure to allow efficient and easy to program communication among multitude of data management systems. It allows the access to different data sources and models, both SQL and no SQL data stores. UPC will provide the Common Query Engine as open source.

1.3 Main Results

The Interoperability Framework integrated development environment represents a secure, stable and expandable development environment built on selected standard technology resources and enhanced by a set of ad hoc tools and utilities to be used for development, Proof-of-Concept integration testing, support for WP7 IT2Rail Pilot activities, and for further development activities within the full SHIFT2Rail IP4 program.

Resources include:

1. Existing ontologies, web services and data sets;
2. Frameworks, libraries, plugins, triple stores, graph databases, documentation;
3. Integrated Development Environment, including editors and test suites.

Tools and Utilities include:

1. Legacy data extractors and converters;
2. Utilities to semantically annotate and link legacy data sets and web service descriptors;
3. Test utilities and emulators needed to support integration activities coordinated within the IT2Rail WP7 package.

1.4 Links With Other Deliverables

This development environment is used for platform-specific implementations described in deliverables D1.1 through D1.4 and D1.6 through D1.8. It is the support for WP7 IT2Rai Pilot activities.

2. REFERENCED DOCUMENTS

This section lists the document reference number, title, revision, and date of all documents referenced in the specifications document.

Reference Number	Title	Revision	Date
[1]	ITR-GEN-C-DAP-006-02_-_Grant_Agreement_-_IT2Rail.pdf	06	13/04/2015

Table 1: Referenced Documents

3. FUNCTIONALITY

3.1 DEFINITION OF THE INTEROPERABILITY FRAMEWORK

The Interoperability Framework guarantees technical interoperability of all multimodal services by insulating consumer applications from the task of locating, harmonising and understanding an open-ended world of data, events, and service resources, which are consequently made available “as a service”.

In compliance with the principle of openness, the Interoperability Framework is agnostic with respect to any application requiring its services, thus allowing multiple and concurrent implementations of the multimodal services and travel companions to access the full range of available data.

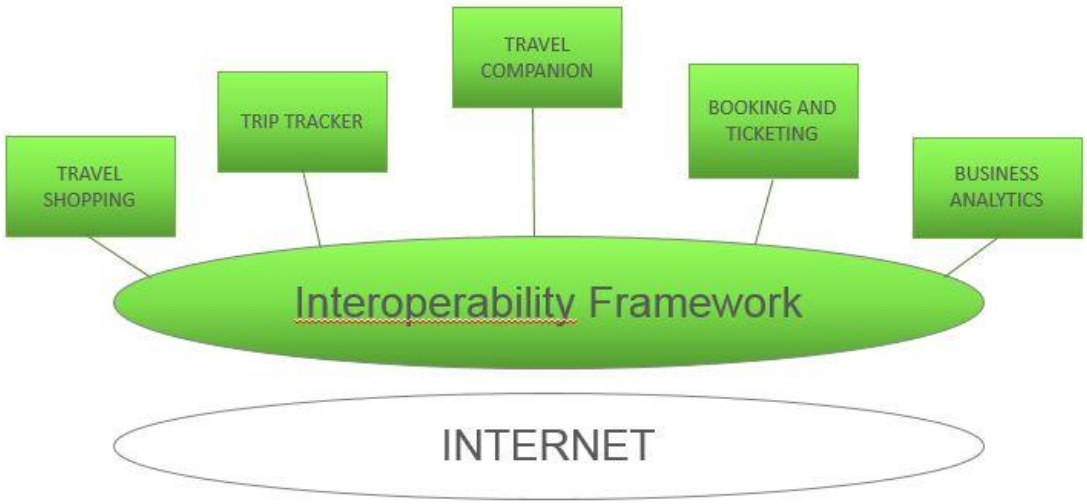

Figure 1: Interoperability Framework Schema

The Interoperability Framework realises its definition by:

- providing travel applications with a uniform abstraction data and services distributed over the world wide web as a “web of transportation data” in the form of *linked* data and service descriptors annotated with machine-readable logical statements which describe their semantics;
- providing applications with technical means to operate on such “web of transportation data”, e.g. publishing, querying, etc. where the semantic annotations are used to automate the process of discovering and matching data sets and service descriptors.

3.2 SEMANTIC INTEROPERABILITY

Semantics refers to the *interpretation* of facts and events encoded in data in some format (or 'syntax'). It describes the meaning of syntactical expressions according to an axiomatic definition of concepts and relationships, which are in themselves independent of the format chosen to represent them, in such a way that any specific representation, or even different representations of the data can be recognised to refer to instances of the same concepts and relationships being represented and processed accordingly.

The process of annotation of data consists in associating the data with their meaning, or semantics, so that the process of interpreting them can be automated, i.e. performed by machines. An example of such "interpretation" that can be automated and acted upon is the mapping of one data format into another: machines can recognise that two different data items in different formats are different representations of the same instance of a concept, e.g. a particular train, and consequently map one into another.

The Interoperability Framework provides such an axiomatic description of the "knowledge" of the domain in the form of first order predicate logic statements expressed in a standard machine-readable language, the problem domain's ontology, and the tools needed to annotate data items in any format with these statements.

The ontology constitutes therefore "common language" in which knowledge about the domain can be expressed *independently* from the data formats and protocols used in exchanges. The Interoperability Framework services use the semantic annotations, i.e. terms from this "common language" to automate the process of discovering, inferring, mapping and processing distributed heterogeneous data in order to relieve applications from the tasks of interoperating. Thus, the meaning of exchanged information is no longer embedded in the applications themselves and can be distributed *in conjunction* with the data using services, thus allowing machines to process the data in *any format* according to their intended *common* meaning. This mechanism, whereby machines interoperate *across* heterogeneous data formats using a common interpretation is called *semantic interoperability.*

3.3 SUPPORT FOR SERVICES

As it is stablished in the Technical Annex of IT2Rail [1], since IT2Rail functional applications rely on its capabilities for interoperability, it is important that the Interoperability Framework's development strategy be provided with 'built-in' characteristics accommodated to minimise possible adverse impacts on the successful completion of the entire IT2Rail Project. It must furthermore allow for the optimal use of the specialised technical knowledge of contributing partners, and for a smooth transition into the full SHIFT2Rail IP4 innovation program development. In order to meet those requirements, an iterative-incremental development strategy will be executed, in which the interoperability framework will be constructed in repeated requirements-to-test build cycles (iteration) of stable, executable artefacts that meet specific requirements synchronised with the development of other work packages, adding features to the foundation provided by each previous iteration (incremental) until completion.

The incremental functionality of the Interoperability Framework, even beyond IT2Rai project, must be considered as input for the development strategy. IT2Rail defines three main releases: CREL (core features), AREL (additional features and resolvers) and FREL (full features and resolvers). The functionality of the IF must be enough to support all the core requirements (CREL), so that the next releases (AREL and FREL) can be increased over it. This doesn't mean that in each new release some improvements into the IF can be included to ensure the performance, robustness and

security. These new technical requirements will be identified during the proof-of-concept testing and implemented when necessary.

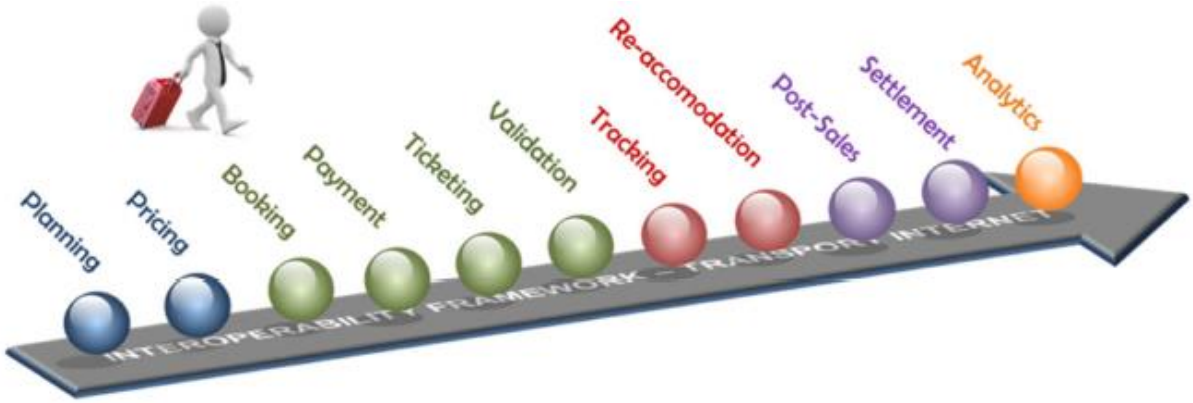

Figure 2: Travel Services Supported by the Interoperability Framework

The framework will allow unprecedented services interoperability whilst limiting impacts on existing systems, without prerequisites for further centralised standardisation. Transport industry incumbents and newcomers will discover wide opportunities to provide new services, products and new competitive business models.

3.4 TECHNICAL REQUIREMENTS

3.4.1 Interoperability

Interoperability represents one of the main technical requirements of the Interoperability Framework. It must support both new and legacy systems. The framework will implement *semantic* interoperability across multiple data formats, allowing unprecedented levels of interoperation of services, while limiting impacts on existing systems, and without prerequisites for further centralised standardisation of data formats and protocols. The Transport industry incumbents and newcomers will discover wide opportunities to provide new services, products and new competitive business models.

3.4.2 Standardisation

Although the ecosystem built around the IF will be based on heterogeneous systems, we need some standards to allow the integration of all of them in order to reduce the complexity of the integration. This standardisation must be agnostic with respect to the technical aspects underlying each system or service, but it must allow the communication and understanding of all the systems to make possible the interaction for the exchange of information. That is, although we must respect that each system has got its own working language, all systems and services connected must be able to "translate" its information terms into a single "interaction language" in order to allow the understanding between each other. Using *semantic* interoperability, the common "interaction language" will consist in the domain's knowledge formalisation expressed in the standard ontology web language (OWL).

3.4.3 Scalability

Scalability is a natural consequence of the interconnected systems and services' expected growth. The interoperability framework must support the scale-up without it meaning a resource saturation that may imply a system malfunction. To do this, the design to be considered must allow for the resources to grow dynamically, according to the own system's growth, and even on demand to support activity peaks. PaaS cloud solutions are outlined as a solution for resource scalability.

Another challenge showed by scalability is the system management complexity, because the services must be coherently orchestrated in every moment. The Semantic Discovery Engine (D1.3) is constituted as the module in charge of talking with the interoperability framework so that the connection of a new system or service, or the disconnection of an existing system, is logically correct.

3.4.4 Decentralisation

The interoperability framework must be able to operate in a distributed environment, which will prevent isolated errors to generate the risk of a full-system inoperability. In a decentralised distributed environment any new or legacy system or service can continue operating normally without migrations to new environments while at the same time it collaborates with the rest of systems and services, thanks to the Interoperability Framework. In this sense, each of the connected systems and services will be decentralised and the IF will become the orchestrator for all of them.

3.4.5 Robustness

The whole system orchestration relies ultimately on the IF, thus it is required for the IF to be resilient. The IF's fault tolerance must be close to zero, requiring for back-up systems to be available to continue with the operation in case of eventual system failure.

3.4.6 Security

System connectivity through the IF must be secure. Weak points allowing for information eavesdropping and capture must be avoided. Should security failures exist, the IF must allow for their early detection or, ultimately, for their discontinuation. Logging, encryption and periodical permission refreshment are introduced as possible measures to guarantee security. Those measures must be applied without them affecting the system's normal operation.

3.5 Semantic Interoperability Approach

3.5.1 Objectives

The semantic paradigm allows for the fulfilment of heterogeneous system interoperability's technological requirements through a use approach of a compliant, simple and standard language. It leverages existing transportation ontologies. It is also open and expandable as data providers can join on voluntary basis by sharing data annotated with terms of the ontology’s vocabulary.

Semantic interoperability addresses the Interoperability Framework’s requirements as follows:

- **Interoperability:** Through the creation of "connectors" that will allow for information exchange using the common shared domain’s ontology. The connectors produce/consume information via publication of services;

- **Standardisation:** Using a common, shared ontology that define a standard message's formal interpretation of the information to be exchanged;
- **Scalability:** Through the aggregation of systems and services that can connect to the interoperability framework and speak the defined ontological language, and the ability to deploy multiple consistent and concurrent instances of the interoperability framework to match load requirements;
- **Decentralisation:** Through the use of SaaS, PaaS solutions. The information will travel in the cloud through different distributed systems;
- **Robustness:** A failure in a satellite system will not affect the other systems exchanging any kind of information with the former. The interoperability framework centralising the system orchestration is identified as a critical point that will be replicated and, in case of failure, replaced with the back-up system;
- **Security:** Exchanged information will be encrypted to travel through the net. The systems' connectivity points will ensure security by using tokens to identify authorised clients.

3.5.2 Common language

A common language defines an open common framework which can facilitate information exchange between all IT2Rail actors in a secure, flexible way. It avoids the problems of:

- replacement of existing Information Systems;
- expensive IT;
- centralised control.

On the other hand, a common language generates two issues to be handled in order to define an Interoperability Framework which allows seamless information exchange between the involved applications:

- Make information univocally understandable with no ambiguity;
- Transfer appropriate information according to application needs.

The challenges associated with working with a common language are met by defining:

- A shared ontology providing a common interpretation of information in any format, including:
  - Annotations of local data representations with the terms of the shared, expandable domain's ontology;
  - Usage of Data driven procedures.
- A service mechanism in order to exchange messages, including:
  - An information flow through interoperable Web services;
  - Service oriented, peer to peer architecture.

3.5.3 Distributed system

Semantics makes easy and implementation based on distributed systems:

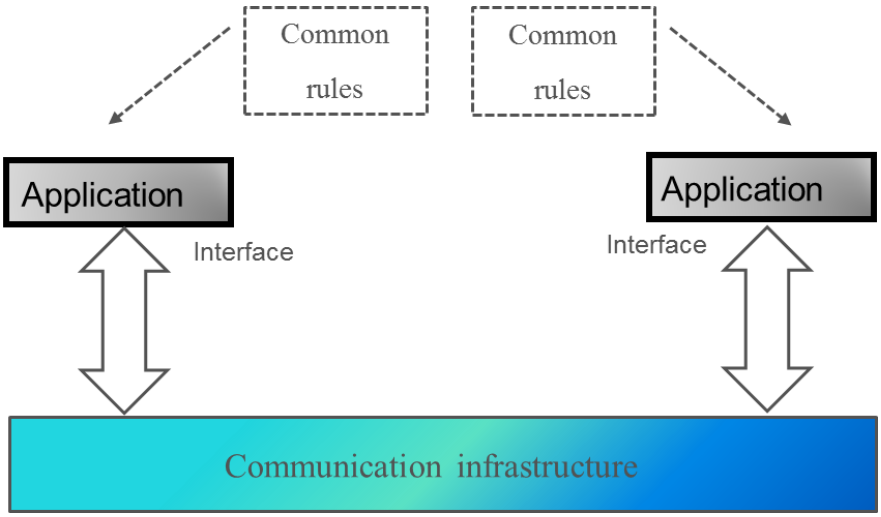

Figure 3: Distribution of Systems

A distributed system consists of several interacting applications which communicate with each other by means of messages. Each application needs to incorporate an interface towards the other applications. The applications are able to understand each other only because they have been designed having in mind a set of common rules, which are embedded in the applications themselves.

3.5.4Interoperable Framework and the semantic technology approach

Stage 1: Semantic Modelling

The semantic model defines the formal interpretation of different information structures that will be handled in IT2Rail's functional domain. The domain's knowledge will be encapsulated in "ontologies" which represent concepts with properties of their own in the real world, and which contain dependent information units. Additionally, within the semantic model, we can also establish relationships between different ontologies. For instance, the concept "Route" can have the relationships with the concepts "Origin" and "Destination". In turn, both the Origin and Destination concepts can have relationships with the concept "Location", which contains properties referring instances of the concepts "Country", "City" and "Station". The structured collection of concepts, relationships and properties in the domain is the domain's ontology which results from the process of semantic modelling or "ontology engineering".

The difference between a traditional Entity-Relationship model, besides the fact that it provides greater flexibility and simplicity and a lesser degree of schema normalisation, lies in that the ontology is a set of first order predicate logic statements that can be used by machine reasoners to perform inference tasks, for example to automatically discover equivalence relationships between two different data representations of the same object.

The shared ontology is thus a completely structured and standardised formalisation expressed in a single language, of information resources, allowing all the interconnected systems to "understand" each other through a common machine interpretation of different data formats.

Stage 2: Publishing information (data provider)

A way to build standard interfaces is to base them on services. Services are interfaces which allow applications to exchange messages. They are constructs transparent to the content or

communication protocols of the message exchange. Applications are therefore decoupled from the communication aspects of the exchange.

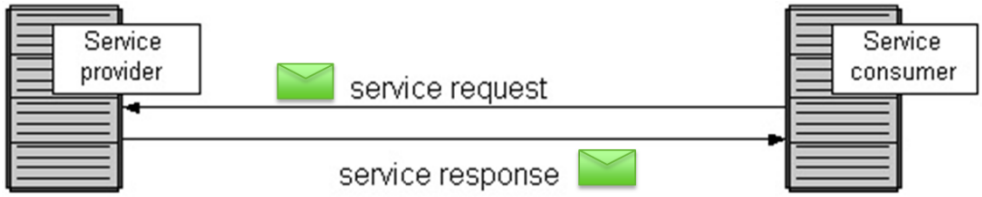

Figure 4: Data Transmission

Publication of information allows each system or service to share this information with its peers. This is done through the IF, which centralises the data supply from different sources.

Stage 3: Subscribing information (data consumer)

Subscription to information allows each system or service to get information that a third party has previously shared. This is done through the IF, which coordinates the data supply from different sources.

Stage 4: Information processing

As data processed through the IF becomes the reference component to process the information. The IF can coordinate data storage, being thus the reference for integration with the different analytic modules. This way, the IF itself becomes one additional data source in the landscape, with derived information that can be used by the other systems.

4. ARCHITECTURE

4.1 CONCEPT MAP

The concept map allows to clarify what is -and what is not- the Interoperability Framework, where it reaches and where it does not, and to know clearly the role it plays in IT2Rail.

The interoperability framework is the layer on which the IT2Rail services lean, and which we can group in Booking & Ticketing, Trip Tracker, Travel Shopping and Business Analytics. These services in turn support the applications offered to the users, and which have been summarised as “Travel companions” in the image.

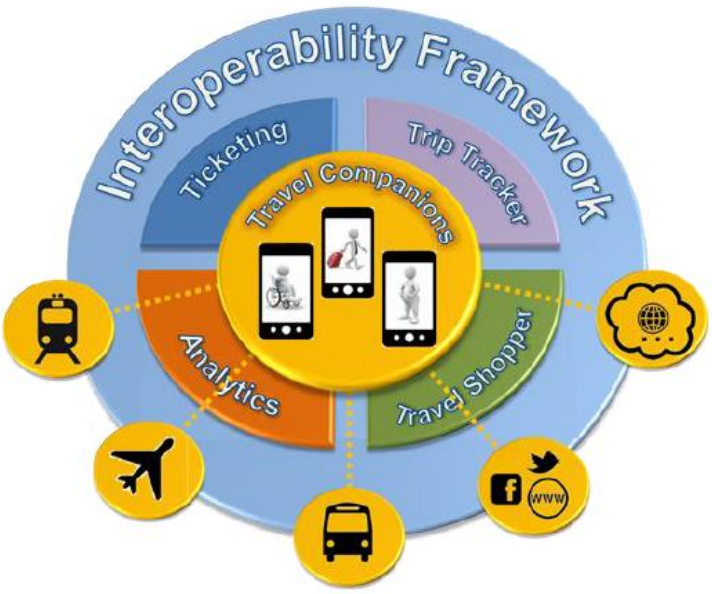

Figure 5: Concept Map

The interoperability framework thus does not implement neither the IT2Rail services nor its applications, but it is the middleware allowing all of them to communicate and understand each other. Thus, a holistic system, where the whole is greater than the sum of its parts, is built based on solutions both new and existing in different functional domains (railway, air, bus, social networks, etc.).

4.2 SERVICE ORIENTED ARCHITECTURE

4.2.1 Connectivity

The connectivity of the different services with the Interoperable Framework is achieved using the paradigm Software as a Service (SaaS). The SaaS implementation is performed using SOA (Service Oriented Architecture). SOA allows for different implementations including SOAP, REST, MQTT, WebSockets, etc.

The interoperability framework must be compatible with the different SOA implementations, but the connected systems require only one of them to be implemented. Therefore:

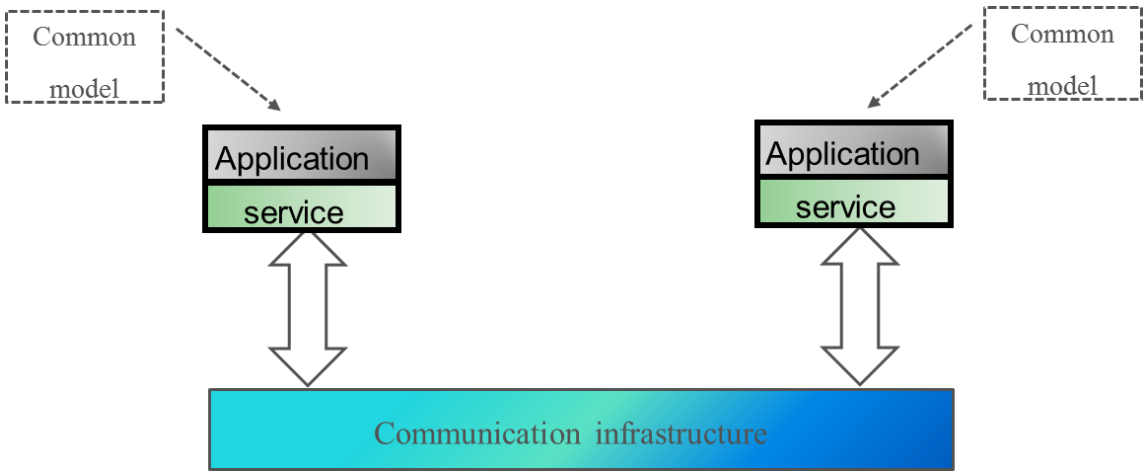

Figure 6: Communication Schema

4.3 SEMANTICS

4.3.1 Ontologies

The Ontology represents the knowledge needed by applications in order to understand each other. The meaning of exchanged information is no more embedded in the applications themselves.

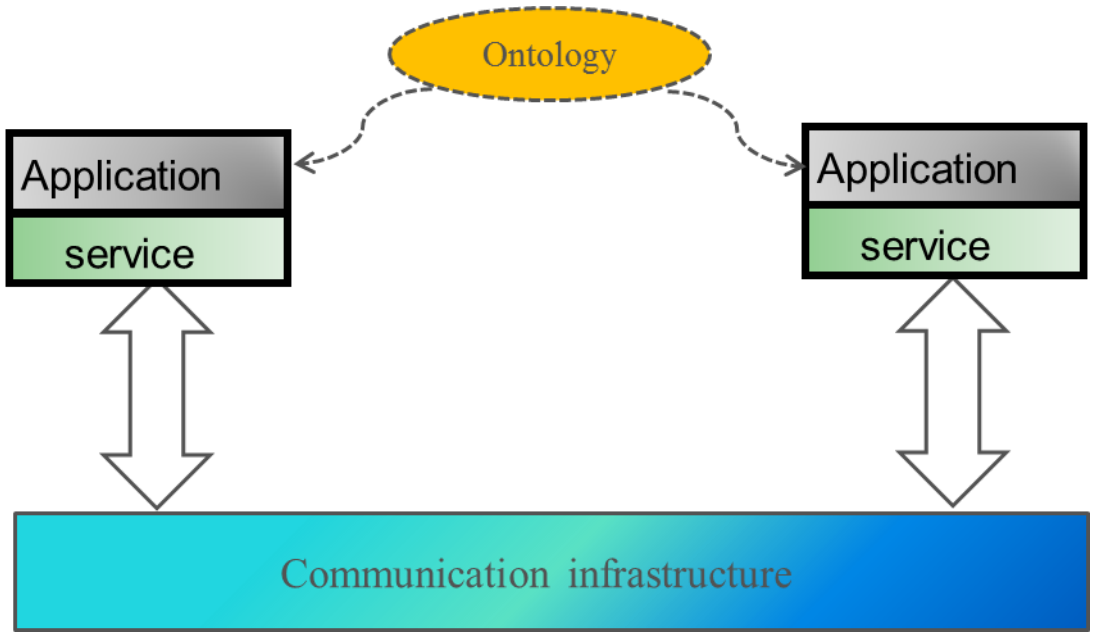

Figure 7: Communication Schema using Ontologies at Application Level

4.3.2 Semantic Information Broker

It is the core of the system. It receives, processes and stores all the information from applications connected to the IF, thus acting as an Interoperability Bus. All the existing concepts in the domain (reflected in the ontologies) and their current states (specific ontology instances) are reflected on it.

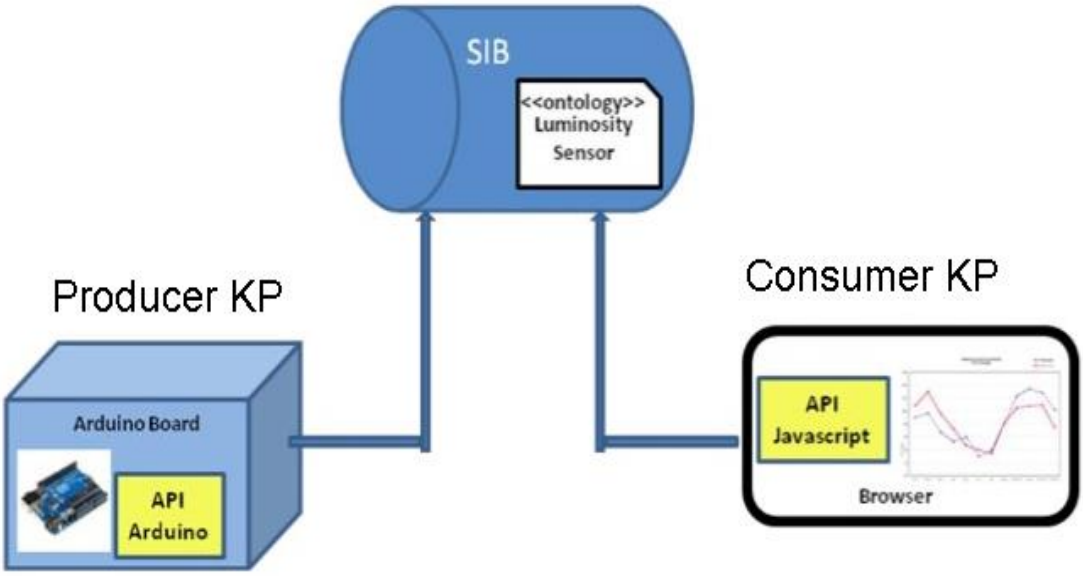

Figure 8: Semantic Information Broker (SIB)

Each of the systems and services connected to the IF must have implemented a connector ("Knowledge Processor") to allow the information exchange. This Knowledge Processor will be implemented under any of the SOA communication protocols supported by the IF, and the information will be modelled following the semantic model specification.

4.3.3 Auxiliary Modules, Tools and Utilities

The Interoperability Framework can be complemented with a number of modules, tools and utilities to optimise and/or complement its base functionality. Those will be open use and easy to integrate in the architecture so that it can be made available without restrictions. Here we find, among others:

- Analytic capabilities that can be implemented using a set of open tools that can be integrated in the IF;
- RDF data management capabilities, allow for the use of the OWL specification to implement full-semantics;
- Capabilities to allow for the implementation of services such as the resolvers.

5. IMPLEMENTATION

5.1.1 Smart M3 semantic model and SOFIA Artemis Project

Description

The smart M3 model aims at providing a Semantic Web information sharing infrastructure between software entities and devices. It combines the ideas of distributed, networked systems and semantic web. The ultimate goal is to enable smart environments and linking of real and virtual worlds.

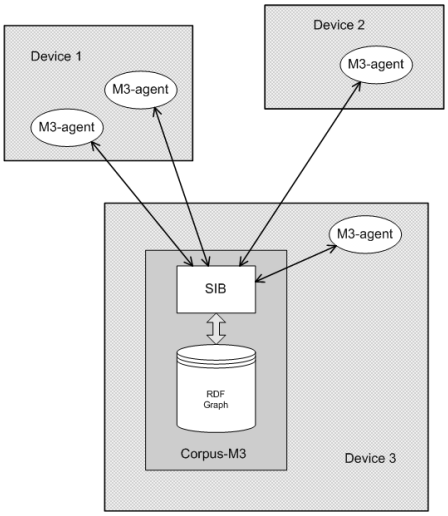

Figure 9: Smart M3 Semantic Model

The Interoperability Framework can be understood as a Smart M3 model, where the devices/agents are the services connected and the Semantic Information Broker is the Interoperable Framework.

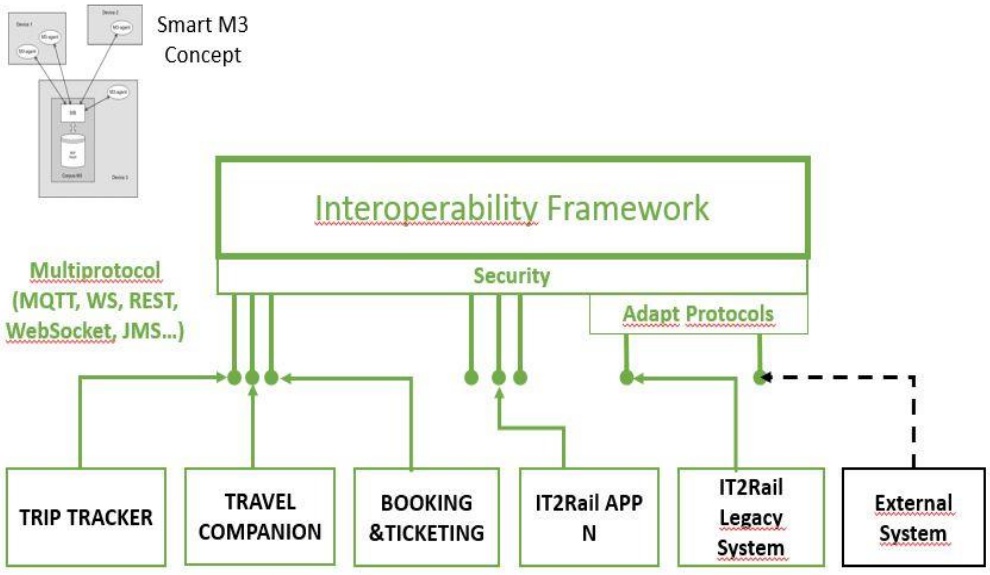

Figure 10: Interoperability Framework as Smart M3 Model

SOFIA Artemis project

SOFIA Artemis project (Smart Objects For Intelligent Applications) (Jan 2009 – Dec 2012). The mission of SOFIA project is to create a semantic interoperability platform which enables and maintains cross-industry interoperability which is a platform for new services. The solution fosters

innovation while maintaining value of existing legacy multi-vendor interoperability platform. SOFIA builds a platform that implements the Smart M3 model.

Problems found

The free and open source Java framework Apache JENA was used for building the Semantic Web Platform and Linked Data applications on SOFIA. Jena supports RDF, OWL and a triple Store. Although the deployment and execution of the open framework JENA was good, when OWL was implemented over it, performance problems appeared in terms of lack of memory and latency for simple tests. OWL-Lite was adopted during the project as a solution. Successful results were achieved in more complex performance tests, but still on lab conditions. Other open frameworks for building semantic apps where analysed but without significantly better results, in particular assuming OWL-full.

5.1.2 SOFIA2 as core solution for the Interoperability Framework

The incoming Internet of Things market demanded a platform like SOFIA. But the performance of SOFIA on real scenarios, managing thousands of sensors and devices and requiring real-time response, was unacceptable.

Indra, as industrial partner of SOFIA2, rebuilt the initial project taking the core concept idea of the semantic approach and the Smart M3 model, but adapting it to be deployed in real conditions solving the problems found.

- Agile technical solutions were adopted. The use of XML/WebServices was obsolete being overcome by other solutions like JSON/RESTful with better results on complex scenarios, with agile response time;
- A light version of semantics was implemented. The standard reference for semantic modelling based on OWL was XML. It was replaced by JSON. OWL-Lite was replaced by the definition and handling of ontologies that could be directly managed by RESTful services. SOFIA was rebuilt from scratch and SOFIA2 was created. SOFIA2 was able to manage the Internet of Things challenges, resigning OWL but respecting Smart M3 model in which it still stands.

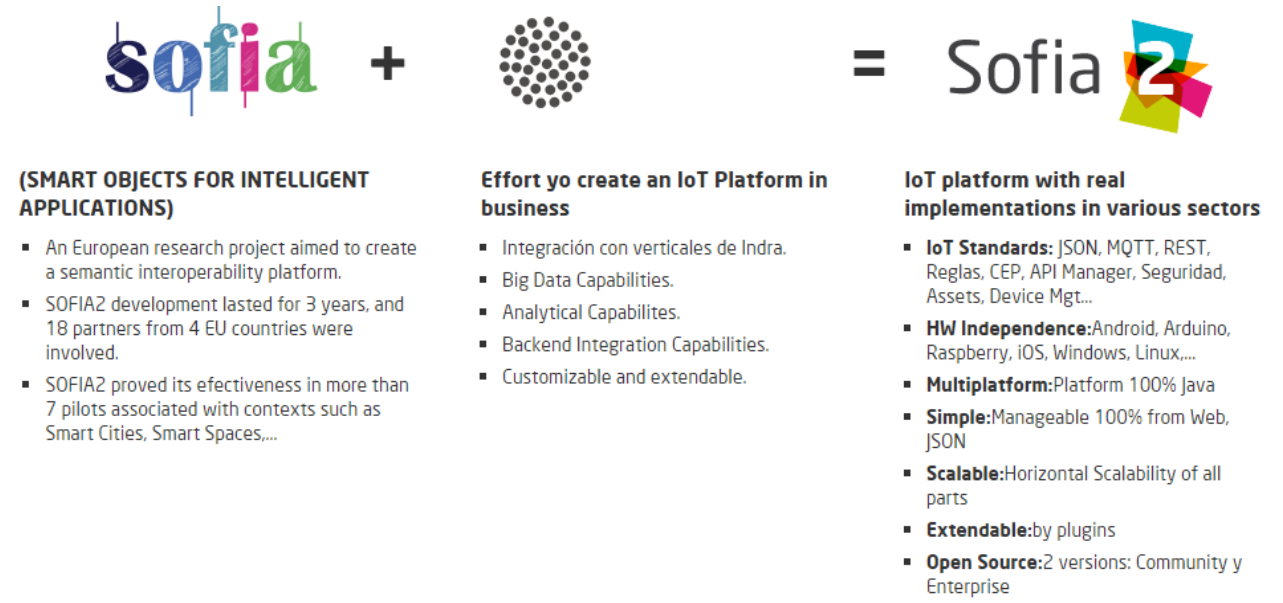

Figure 11: SOFIA2 Rebuilding

The interoperability would be executed semantically through SOFIA2. Besides, SOFIA2 provides integration support for the Analytics and Reporting functionalities of the IT2Rail platform.

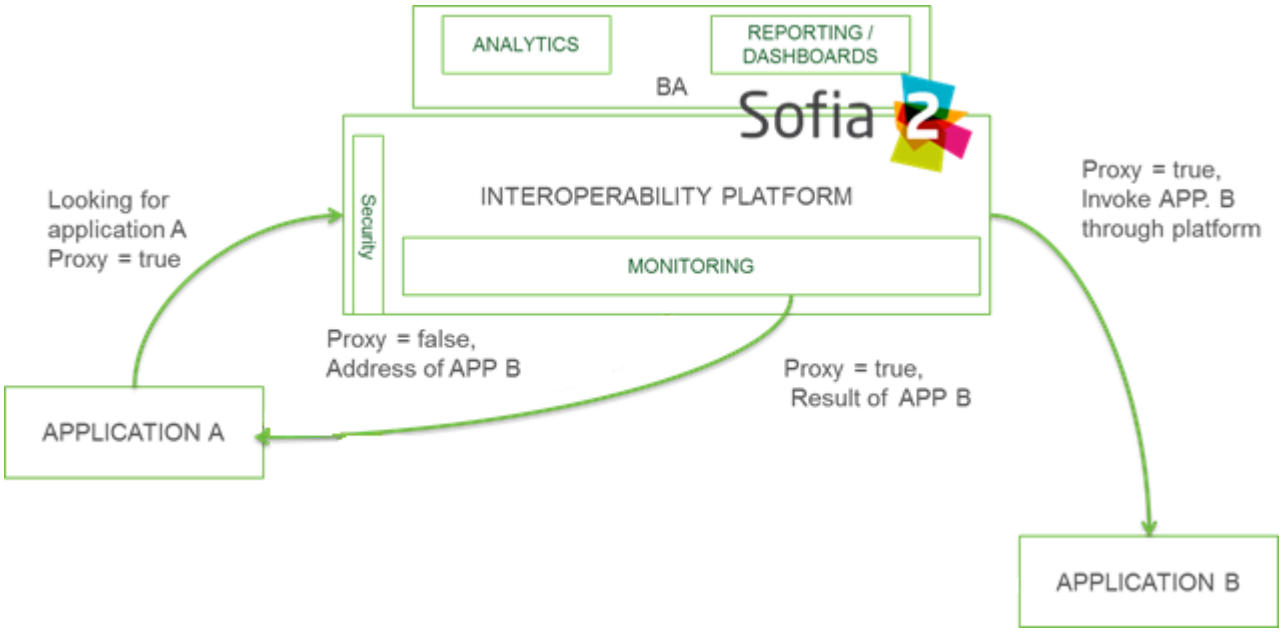

Figure 12: Service Invocation Model

The full description of SOFIA2, functionality, key concepts, use case references, documentation for its use and development can be found in the Table 1Table 2.

In the table we include the most important documents of SOFIA2 for its use as the Interoperability Framework core of IT2Rail. This preliminary documentation can be completed with other available

documents for the development and advance use of SOFIA2. Available at https://sofia2.com/desarrollador_en.html#documentacion.

Title	URL	Description
SOFIA2 IOT PLATFORM: TECHNICAL VIEW	http://sofia2.com/docs/SOFIA2%20-%20Technical%20-%20IoT%20Platform%20(oct%202014).pdf	Technical description at high level including use cases references
SOFIA2 CONCEPTS	http://sofia2.com/docs/(EN)%20SOFIA2-SOFIA2%20Concepts.pdf	Key concepts of SOFIA2 Architecture
SECURITY	http://sofia2.com/docs/(EN)%20SOFIA2-Security.pdf	Security mechanisms applied to SOFIA2
SOFIA2 WEB CONSOLE USE GUIDE	http://sofia2.com/docs/(EN)%20SOFIA2-Web%20Console%20Use%20Guide.pdf	Basic concepts of the Web Console that is integrated in SOFIA2's Platform.
ONTOLOGY DEFINITION IN SOFIA2	http://sofia2.com/docs/(EN)%20SOFIA2-Ontology%20Definition%20in%20Sofia2.pdf	How to define ontologies in a clear and simple manner
FIRST STEPS WITH SOFIA2	http://sofia2.com/docs/(EN)%20SOFIA2-First%20Steps%20with%20SOFIA2.pdf	How to set up communications, sending and receiving data

Table 2: SOFIA2 Reference Documents

5.1.3 Common Query Engine

Description

Providing massive data processing capabilities in the cloud is a major trend in the design of data management solutions deployed on the cloud. The experience of the latest years is that no single data management system is the silver bullet for data processing, where all the data needs can be mapped. Currently, companies are using a variety of data solutions ranging from relational databases to NoSQL data stores, which come in multiple flavours such as graph databases, key-value data stores, array data stores, analytical cloud frameworks, document databases, data stream systems, etc. SOFIA2 provides a software infrastructure to allow efficient and easy to program communication among this multitude of data management systems. In this deliverable, we describe the internal design, the build and the execution process of the query engine that interconnects the data stores.

The query engine is central piece that coordinates the execution of queries in SOFIA2. This module executes queries in the CloudMdsQL language, which defines syntax to mix the operations among the data repositories. The data in the query engine is modelled as tabular data: sets of tuples with a fixed number of attributes. This model is simple enough to allow importation and exportation of data from NoSQL data representations.

Clients connecting to the SOFIA2 infrastructure will have the impression that all systems act as a single database. In order to provide such a feeling from the client perspective, we will provide a JDBC connector for the SOFIA2 infrastructure. JDBC is one of the standard methods to connect to

a database system, and thus clients will connect to the database like a regular database system. However, user queries will be able to take full advantage of the different repositories in the SOFIA2 infrastructure by introducing SQL and NoSQL statements.

Architecture

The Query Engine is able to compute queries by forwarding them to the Data Stores, and also by combining results using the Operator Engine. The architecture of the Query Engine is based on Apache Derby. Therefore, the Operator Engine is able to execute SQL programs that are generated from a CloudMdsQL expression. The Query Engine has also a module, called Table Store, which saves the tabular results of queries into disk, memory or in a remote database by JDBC. The tables contain either the query results or temporary results that will be further processed by the Operator Engine or any of the data stores in the SOFIA2 ecosystem.

The Cloud multi-data store Query Engine (CloudMdsQL) is a query platform integrated with a very scalable transactional processing system that provides full ACID transactions over arbitrary sets of cloud data stores. The Query Mediator is the component that performs communication between the clients that implement the user functionalities, and the set of database engines in SOFIA2. The following picture shows an overview of the Query Engine architecture:

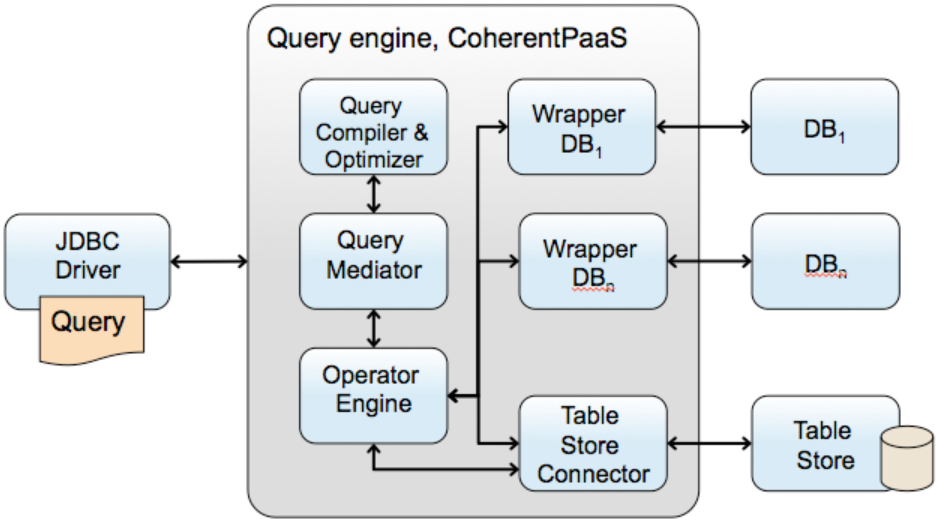

Figure 13: Query Engine

The Query Mediator is built as an extension of the Apache Derby database and is the responsible of creating and preparing CloudMdsQL statements. In order to prepare a CloudMdsQL statement, the Query Mediator needs to initialise a set of structures, which determines a query context. Afterwards, the Query Mediator invokes the compiler and generates the SQL operations to be executed. Finally, the Query Mediator associates the byte code necessary to execute the SQL operations to the CloudMdsQL statement. The Query Mediator is implemented as an extension of the Apache Derby project to reuse the internal Derby Operator Engine.

The Operator Engine is used in order to describe the workflow of the query execution. Derby table functions represent the named table expressions of a CloudMdSQL. These table functions are created during the query context initialisation and are referenced by the generated SQL operations. These references imply to make a table function invocation, which internally creates a statement to

connect to a specific datastore or resolve a nested query. In this section, we cover which kind of statements the CQE supports and their internal design.

Once a CloudMdsQL statement is compiled, the CQE generates the Java bytecode necessary to make the statement executable. The instantiated SQL nodes generate this bytecode during the compilation process. The bytecode generation is an entire functionality that originally Apache Derby performed for SQL expressions. Thus, the Operator Engine consists of the execution process of this generated bytecode.

A Wrapper provides the interface that attaches a data store to the CloudMDS Query Engine (CQE). It handles fragments of the query plan that are intended for execution in a data store and delivers interim results in the appropriate format.

The Wrapper Interface and implementations are based on the Java platform and conceptually similar to the standard JDBC interface, namely, regarding the usage of Driver/Connection/Statement objects to provide nested context for query execution, and a ResultSet to iterate over result data. Namely, the main entry point to the wrapper is the DataStore interface, which provides Connection contexts for the execution of Statements. The query is received as a Java mapping of the JSON data model. Interaction with the Table Store is, in both directions, through the ResultSet batch iterator interface.

Build

The CQE started from Apache Derby, but it was moved to Apache Maven [Maven] to simplify the build process. Currently, the project can be built with the following simple command:

```
mvn package assembly:assembly
```

Table 3: Command for Building Project

This Maven command will produce the binary distribution in *"target/cqe-{version}-bin.zip"* file.

Installation

Having the CQE binary distribution file, the next step is the installation process. If the user unzips the file, the system will create a directory (i.e. CQE_HOME) where the user will find the following directories:

Directory	Description
bin	Contains the binary files that can be executed
lib	Contains the necessary Java library files necessary to start the CQE. It needs to be upgraded with the jars provided by the wrapper implementations the user needs.
config	Config directory to put the wrapper configuration files
samples	Directory with configuration examples
Dockerfile	A new Docker configuration file
docker	A new Docker directory with the docker scripts

Table 4: CQE_HOME Directories

The Docker files, as well as some default wrapper configurations, were added to the distribution for an even easier setup process in the IT2Rail project.

Configuration

Each wrapper configuration must be stored in the *"{CQE_HOME}/config/wrappers/"* directory. A wrapper configuration file, which is a properties file, contains the custom datastore connection properties and what are the implementation classes for the wrapper interfaces. Specifically, the minimum properties set are:

Property	Description
wrapper.class	The wrapper implementation for the Datastore interface.
wrapper.name	The wrapper label to reference this datastore

Table 5: Properties of the Wrapper Configuration

Sample complete configuration files are provided.

Starting the Engine

The command line tool to manually start the CQE is:

```
${CQE_HOME}/bin/startNetworkServer
```

Table 6: CQE Manually Start Command Line

But with the addition of the Docker files it may be more convenient to use docker to start a CQE server:

```
docker build -t cqe/IT2RailCQE_HOME_DIRECTORY

docker run --name cqe -d -p 1527:1527 cqe/IT2Rail
```

Table 7: Docker Files Start Command Line

Statement Preparation

User code requests a PreparedStatement on the JDBC driver. This request is forwarded to the query compiler through the Query Mediator. The query compilation procedure optimises and transforms the user query into a prepared statement that is ready to be executed. To achieve this task, the following steps are needed:

1. Send the query string written in CloudMdSQL to the compiler;
2. Parse the query plan produced by the Compiler in JSON format;
3. Create the equivalent SQL object that represents the query plan inside Apache Derby and compile it to bytecode;
4. Initialise the internal Derby Statement that is linked to the bytecode to be executed by the query;
5. Continue with the normal Statement initialisation that Apache Derby performs.

Statement Execution

User code executes the query. The CQE uses the precompiled query and executes it. As we have commented before, wrapper queries, Python named tables and nested CloudMdSQL queries are executed as Derby Table Functions. Probably, especially if the CQE is resolving a JOIN operation, a table function is called more than once. It is at this point where the CQE takes profit of the table store.

The current implementation always saves the queried data into the table store to avoid the resolution of the same subquery multiple times.

Closing resultsets and statements

By default, the statements are configured to close all intermediate result sets when the main query result set is closed. However, to resolve CloudMdsQL queries, the CQE may have created some unique Derby Table Functions that also need to be removed.

Client applications with JDBC

From the perspective of a client that connects to the CoherentPaaS infrastructure, the query engine will be seen as a database. Therefore, it was decided to implement the interaction between the clients and the query engine as a JDBC connection, which is one of the most popular database connection standards.

The CQE architecture is based on Derby and therefore, as we have commented before, client applications can connect to the CQE using the Derby JDBC driver. This is a public open Java library that can be downloaded from many repositories. In fact, the maven dependency that we are using is

```
<dependency>
<groupId>org.apache.derby</groupId>
<artifactId>derbyclient</artifactId>
<version>10.11.1.1</version>
</dependency>
```

Table 8: Maven Dependency

Query Mediator Architecture

The Query Mediator is built as an extension of the Apache Derby database. Therefore, many Apache Derby components were substituted to remove the assumption that the query language is SQL instead of CloudMdsQL. The following table shows a resume about which Apache Derby classes were overridden (left column), which are the corresponding extensions to build a Query Mediator (center column) and which is their responsibility (right column):

Derby Class	CQE Class	Responsibility
NetworkServerControl	CloudMdSQLServer	Starts/Stops the server
DRDAConnThread	MdSQLDRDAConnThread	Create a DRDAStatement when a new query arrives.

DRDAStatement	CloudMdSQLStatement	Compiles a query creating a EmbedPreparedStatement
EmbedPreparedStatement	CloudMdSQLEmbedStatement	Creates a GenericStmt and asks for its Java bytecode (i.e activation).
GenericStatement	CloudMdSQL2DerbyStatement	Compiles the query string and updates the EmbedPreparedStatement with the generated bytecode (activation)

Table 9: Apache Derby Components Substituted for SQL Query Language

Query Context Overview

Let us see a simple CloudMdSQL query:

```
T1( x int, y string )@db1 = {* yield (1, 'abc') *}
T2( x int, y string ) = ( SELECT T1.x, T1.y FROM T1 )
SELECT T2.x, T2.y FROM T2 WHERE T2.x = 1
```

Table 10: CloudMdSQL Query

Each CloudMdSQL query has its own namespace, because the name of a named table expression (e.g. T1 or T2 in the previous query) is unique inside a query. The QueryContext class represents this namespace and it is very useful for the Query Mediator to store some useful temporary information for query compilation and execution. All the active QueryContext instances are just accessible and created through the CloudMdsQLManager.

Specifically, the more relevant temporal information stored inside a QueryContext is:

- **ResultSets**: Once a named table expression's java.sql.ResultSet object is ready to be read, it is cached in the Query Context. It allows for other named expression to ask for their resolution and read it. Remember that, according to the CloudMdsQL specification, there are several ways to reference an external named table expression:
  - From Python code: writing CloudMdSQL.TNAME in the body of the query;
  - From nested queries. For instance, the T2 named expression, in the previous example, references T1;
  - From native named expressions related to a specific datastore with the REFERENCING clause.

  Every java.sql.ResultSet, produced during the resolution of a named expression, is an implementation that supports multiple reads of the returned rows, through the TableStore;

- **Statements**: Each named query is resolved by the execution of an implementation of Statement interface of the Wrapper's API. During the query compilation process, the Query Mediator instantiates the corresponding statement for each named table and they become accessible to be executed in any point of the query resolution;
- **TableExprs**: Each named table expression is represented as a TableExpr object. There is just one TableExpr per name. The QueryContext stores which is the TableExpr for a

specific name, to resolve the related parameters or the returning columns during the query compilation and execution;

- **TxnCtx**: It is the transactional context related for this query, which is a necessary object for the eu.coherentpaas.cqe.datastore.Statement execution;
- **CloudMdSQLStatement**: It is a necessary object to run Derby SQL statements during the compilation process of a query.

The following UML class diagram represents the data structure of QueryContext with all its components.

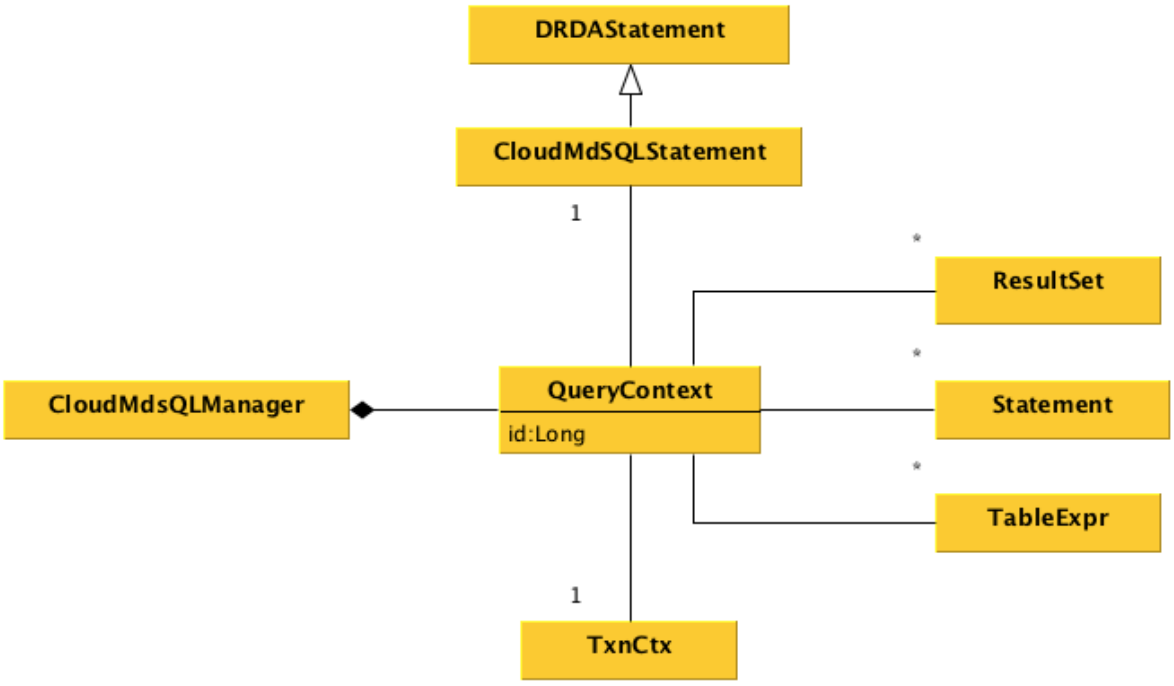

Figure 14: Data Structure of QueryContext

Query Parsing

In order to parse a CloudMdSQL query, the CQE contains a Java class called *CloudMdSQLParser*, which in the same moment the Java class loader initialises the class, it loads the native libraries required by the compiler (libjcql.so for Linux or libjcql.dylib for MacOS), which are inside the CQE classpath. Since this procedure is just executed when the CloudMdSQLParser is referenced for the first time, it is just executed once.

Specifically, the *CloudMdSQLParser* has two methods:

```
parseQueryLanguageExpr (String expr): CloudMdSQLExpr
```

Table 11: CloudMdSQLParser First Method

This method parses the CloudMdsQL string as follows:

1. Invoking the CQL class, which is the Query Compiler JNI interface;
2. Initialising the CQL instance with the all data stores capabilities;
3. Invoking the CQL#createParser, which returns a cql.Parser object;
4. Invoking cql.Parser#parse(String str): String , which returns a JSON string with the query plan;
5. Returning what CloudMdSQLParser# parseQueryPlanExpr computes, with the query plan;

`parseQueryPlanExpr(String expr): CloudMdSQLExpr`

Table 12: CloudMdSQLParser Second Method

This method receives a string that represents query plan (in JSON) and parses it into an equivalent *CloudMdSQLExpr* Java object. To perform it, a Java library called *jackson-databind* is used to apply this mapping between JSON strings and Plain Old Java Objects (POJO) using annotations.

A *CloudMdSQLExpr* is a composition of a set of *TableExpr*, which represent the different table expressions that can be referenced with its own *QueryPlan*, and a main plan, which is represented through a wrapper *Operation*. The following picture shows the UML class diagram related to a *CloudMdSQLExpr*. The yellow boxes represent classes that are part of the wrappers API whereas the green boxes are classes that are inside the CQE.

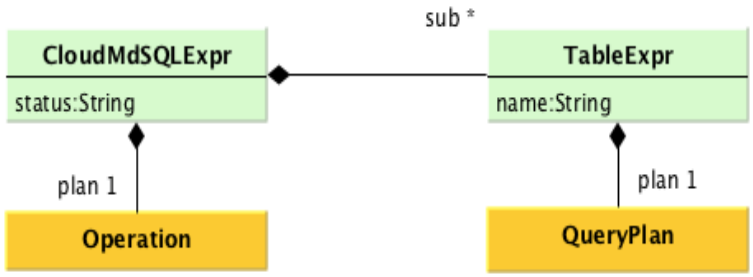

Figure 15: UML Class Diagram *CloudMdSQLExpr*

Query Compilation

The query compilation process consists of preparing an executable statement from a *CloudMdSQLExpr* object. After this executable statement is prepared, the user can execute it multiple times without parsing the query every time.

Since the CQE is built on top of the Apache Derby database, this task can be rewritten to generate an SQL plan that can be executed in Derby. Therefore, a set of transformation rules between a *CloudMdSQLExpr* object and a Derby SQL statement must be established. This section will explain how named table expressions (*TableExpr*) are stored inside Derby to resolve them from a SQL query plan, and how the related *Operation* to a *CloudMdSQLExpr* is transformed to SQL.

Named Table Expression Compilation

Apache Derby has several extensions mechanisms. To perform the query compilation task, the way data is retrieved using *CREATE FUNCTION* statements must be enriched.

The Query Mediator takes profit of the *FUNCTION* statements to resolve named expressions and nested queries.

Named table expressions are defined in the header of a CloudMdsQL query, preceding the SELECT keyword, and are instantiated in the FROM clause and/or from the definitions of other named expressions. Named table expressions are represented inside the CQE as TableExpr objects.

For each named table expressions, the CQE executes a unique *CREATE FUNCTION* statement before compiling the main query plan. To have a unique name per FUNCTION, the CQE uses the *QueryContext#id* followed by the named of the named table expression, which is stored in the *TableExpr#name*.

In Apache Derby documentation, the *CREATE FUNCTION* statement allows you to create Java functions, which you can then use in an expression. For example:

```
CREATE FUNCTION TO_DEGREES ( RADIANS DOUBLE ) RETURNS DOUBLE PARAMETER
STYLE JAVA NO SQL LANGUAGE JAVA EXTERNAL NAME 'java.lang.Math.toDegrees'
```

Table 13: Apache Derby Create Function

The *CREATE FUNCTION* statement has the requirement that the linked Java function has the same number of parameters and with a compatible type.

One important characteristic of named table expressions to take into account is that these can accept dynamic parameters using the *WITHPARAMS* keyword. Let us see an example:

```
T1( x int, y string

  WITHPARAMS a string )@db1 =

(

 SELECT x, y FROM tbl WHERE id = $a

)
```

Table 14: Example of Dynamic Parameters using WITHPARAMS

In order to bind a *CREATE FUNCTION* statement with the same code to resolve a named table expression, a Java code that allows working with a dynamic list of arguments is required. The following method declaration (located in the *WrapperTable* class) is the candidate Java function to bind for all *FUNCTION* statements.

```
public static ResultSet read(

String tableName, Long ctxtId, Object... args) throws Exception {
```

```
// code

}
```

Table 15: Method to Bind all Function Statement

Unfortunately, one limitation of the current Apache Derby database, which has appeared a challenge for the CQE, is that the *CREATE FUNCTION* statement does not accept Java method with a dynamic list of parameters (the *Object… args* parameter), which is a feature introduced in Java 5.

Specifically, when Java compiles a method call that corresponds to a method declaration with dynamic arguments, creates an array with all the values that contain the call. Therefore, for the CQE implementation, the way how Derby compiles *FUNCTION* calls to byte code was modified, simulating the same behaviour. On the other hand, the way how derby validates if the bind Java method has a compatible list of parameter types with the *FUNCTION* parameters required modifications too.

Now, assuming that the CQE admits *CREATE FUNCTION* statements with dynamic parameters, the generated expression that the CQE would generate for the previous named table expression *T1* is the following one:

```
CREATE FUNCTION CTXT234324_T1
(
 name VARCHAR( 50 ), -- `T1'
 ctxt BIGINT, -- '234324'
 a VARCHAR( 50 ) -- param
)
RETURNS TABLE
(
 x  INT,
 y  VARCHAR( 50 )
)
LANGUAGE JAVA
PARAMETER STYLE DERBY_JDBC_RESULT_SET
READS SQL DATA
EXTERNAL NAME 'eu.coherentpaas.cqe.WrapperTable.read'
```

Table 16: CQE Generation Expression T1

Statement Initialisation

According to the CloudMdsQL specification language, there are three types of named table expressions:

Native named table expressions: Expressions of this type are referencing queries to data stores using their native query mechanism. They are executed in the context of a particular datastore connection.

SQL named table expressions: Expressions of this type can reference named tables from the context of the current CloudMdsQL query, but can also reference tables from a corresponding data store. If they are associated with a specific data store, the data store wrapper inside the query engine processes them. Otherwise, they are processed as a sub query inside the query engine applying the same transformation rules as the main query plan.

Python named table expressions: Expressions of this type are not referencing a data source, but are executed in the context of the current CloudMdsQL query. The code of this named table expressions are python.

Each type of named table expression needs to be solved with a different approach. Native named table expressions need to be solved with the wrappers API; SQL named table expressions need to be solved with the wrappers API if they are associated with a specific data store; Python named table expressions need to be solved using a Python interpreter.

To support different approaches from the same Java code (*WrapperTable* class) that is executed when a *FUNCTION* call is produced inside a SQL statement, the CQE adds two implementations for the *Statement* wrappers API interface: *EmbeddedStatement*, which allow to resolve SQL named table expressions not associated to any data store; and *PythonStatementImpl*, which allows to resolve Python named table expressions.

The following UML class diagram shows the type hierarchy of *Statement*, where we can see which classes are part of the Wrapper interfaces; which classes are part of the CQE; and which classes are data store statement implementations.

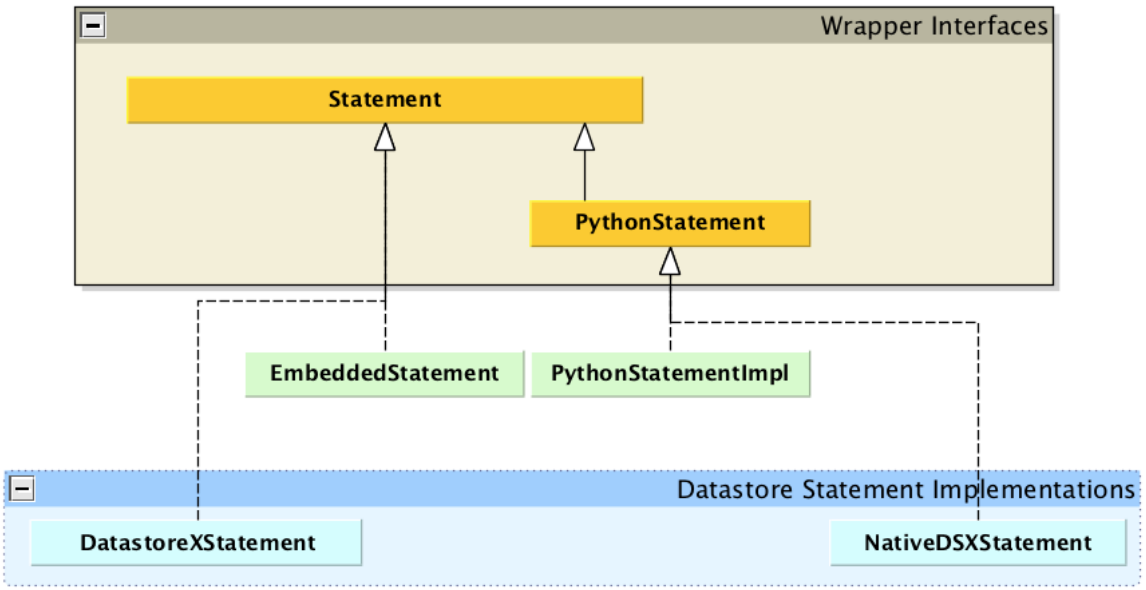

Figure 16: UML Class Diagram of Hierarchy of Statement

Main Query Plan Compilation

Apache Derby is a SQL engine and has its own SQL grammar written in JavaCC. When Derby is built to generate an executable, a set of Java classes to parse SQL expressions are generated automatically. As a result of executing the generated parser by JavaCC, Derby instantiates the equivalent SQL abstract syntax tree (AST) for an SQL expression. Let us see an example starting from the following SQL expression:

SELECT * FROM EMPLOYEES;

Table 17: SQL Expression

Derby generates the following AST (expressed in an UML object diagram):

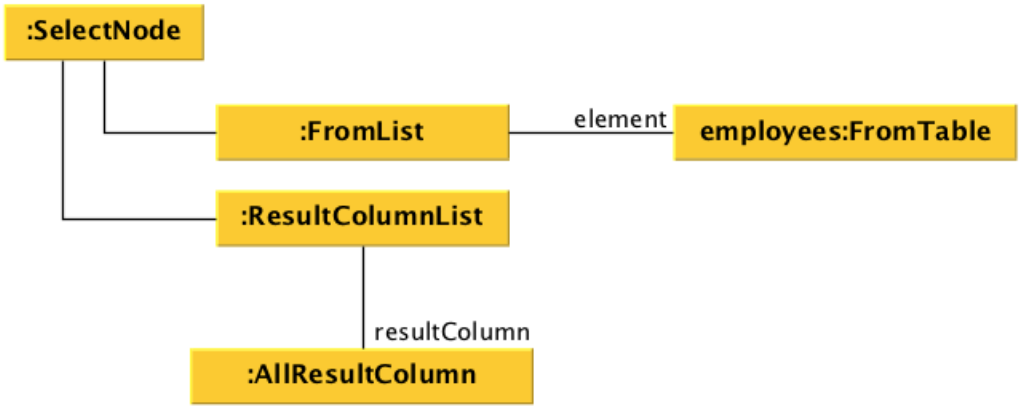

Figure 17: UML Object Diagram of AST

In order to generate the statement byte code associated to an SQL AST, Derby delegates this responsibility to each node type. Afterwards, the statement can be executed.

On the other hand, when a CloudMdsQL is parsed using Jackson, the main query plan has its own abstract syntax tree, whose root node is an *Operation*.

Therefore, the problem of having an executable CloudMdSQL query is equivalent of having the problem of transforming the *Operation* AST into the equivalent SQL AST. To design this language transformation, the most common software engineering pattern to apply is the Visitor pattern.

The main characteristic of the Visitor pattern is to avoid delegating responsibilities to the AST node types. Instead of it, the responsibility is moved to a Visitor class that has a method called visit per each node type. On the other hand, the node types contain an accept method, which delegates to the visitor instance the processing of its inner members/components. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. The following picture is the UML diagram that summarises this design.

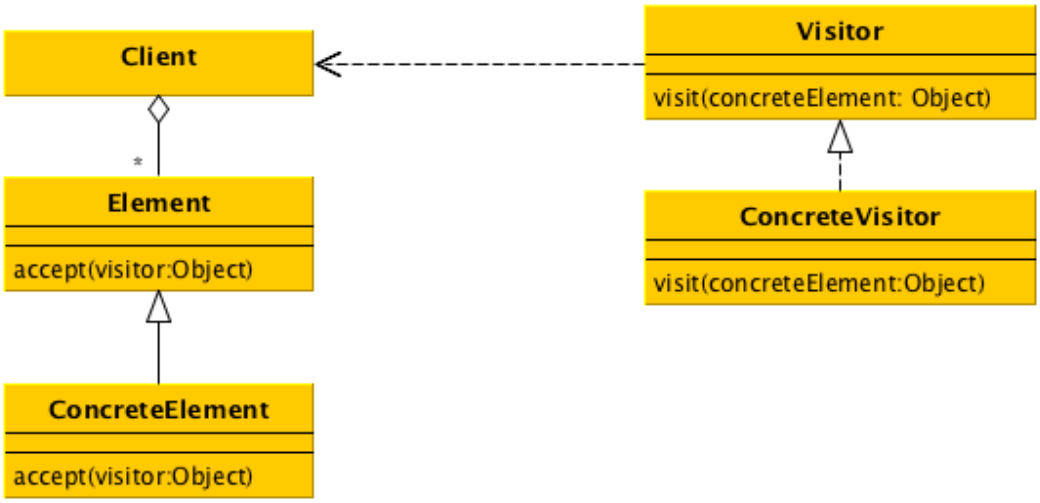

Figure 18: UML Object Diagram of AST

The wrapper API contains the necessary interfaces to implement visitor classes for all the *Operation* and *Expression* subtypes. The visitor class that has the responsibility to transform a CloudMdsQL Operation into an SQL AST is called *OperationToSQLVisitor*. Now, let us see how this visitor works internally.

An SQL query can have nested SQL queries. Therefore, when the *OperationToSQLVisitor* is performing a traversal into the AST, whose root node is CloudMdsQL Operation, it needs to consider different SQL scopes. Specifically, the implemented traversal is a pre-order algorithm. As a first step, if it is necessary, a new scope is created with the current node to visit, and afterwards, the children nodes are processed and the parent node collects their produced SQL nodes and updates the current scope.

The scope of a SQL query is represented by the elements that appear on the top of several Stack data structures. Specifically, the *OperationToSQLVisitor* works with the following stacks:

- Stack<QueryTreeNode>: The root node of an SQL query or subquery. A SelectNode, UnionNode, or SubqueryNode are valid SQL root nodes;
- Stack<ResultColumnList>: The output columns of an SQL node (ex: columns produced by a JoinNode, UnionNode, SelectNode or SubqueryNode node;
- Stack<FromList>: The set of Tables and Join expressions to take into account in the current root SQL node;
- Stack<GroupByList>: The set of columns to take into account as a GROUP BY expression into the current root SQL node;
- Stack<OrderByList>: The set of columns to take into account as an ORDER BY expression into the current root SQL node.

CloudMdsQL node	SQL nodes	Responsibility
Project	SelectNode, SubqueryNode, FromList, GroupByList, OrderByList, ResultColumnList	Initialise in different stacks and push as a QueryTreeNode a SelectNode or a SubqueryNode.
Select		Sets the where and having condition into the current QueryTreeNode.
Join	JoinNode, HalfOuterJoinNode	Creates the JoinNode or a. HalfOuterJoinNode. It implies push in the context a new ResultColumnList to infer the resulting columns of the JoinNode. The created SQL JOIN node is stored inside the FromList of the current context.
Aggregate	ResultColumn, GroupByColumn	Returns in the ResultColumnList of the context, the aggregate columns.
Call	TableName, MethodCallNode, JavaToSQLValueNode, FromVTI,	Creates a call to a table FUNCTION and pushes it to the FromList stack.
ColRef	ResultColumn	Adds a column table into the ResultColumnList of the context.
Const	CharConstantNode, NumericConstantNode UserTypeConstantNode, SQLBlob, BooleanConstantNode	Creates a constant value
Func	BinaryRelationalOperatorNode, BinaryArithmeticOperatorNode, UnaryArithmeticOperatorNode, AndNode, OrNode, NotNode, IsNullNode, AggregateNode	Creates an SQL expression.
TableRef	TableName, MethodCallNode, JavaToSQLValueNode, FromVTI,	Creates a call to a table FUNCTION and pushes it to the FromList stack. It is just executed in CloudMdSQL nested queries (named table expressions).
Sort	OrderByColumn	Adds a column into the OrderByList
Limit		Sets a limit.
Param	ParameterNode	Adds a param into the global parameter list.
Union	UnionNode	Creates a UnionNode. It implies push into the conrtext the UnionNode as QueryTreeNode and new ResultColumnList to infer the resulting columns of the UNION.
InList	InListOperatorNode	Creates a InListOperatorNode with all the values.

CloudMdsQL node	SQL nodes	Responsibility
InQuery	SubqueryNode,SelectNode, FromList	Creates a subquery with the SubqueryNode
Execute	Project/Union	Transforms an execution into a SELECT(0). If there are more than one call, it creates a tree of Union nodes.

Table 18: CloudMdSQL Node and SQL Nodes Responsabilities

Operator Engine Architecture

Once a CloudMdsQL statement is compiled, the CQE generates the Java bytecode necessary to make the statement executable. The instantiated SQL nodes described in the main query plan compilation section generate this bytecode during the compilation process. The bytecode generation is an entire functionality that originally Apache Derby performed for SQL expressions. Thus, the Operator Engine consists of the execution process of this generated bytecode. In this section, we will not cover the details about how Derby has designed the bytecode generation for each SQL node, but we will complete the execution details associated for the previously created SQL FUNCTION tables that represent named table expressions.

Main Query Plan Execution

In order to execute a CloudMdSQL query, the CQE needs to work with different statement types. Indeed, as we have introduced in the Statement initialisation section, the CQE adds to implementations to the Statement interface in order to achieve that all the CREATE FUNCTION statements are linked with the same Java class file called WrapperTable. In this section we will see the implementation details of this class.

All Java classes that are linked in a CREATE FUNCTION table statement must return a JDBC ResultSet. Afterwards, Derby Operator Engine uses this interface to continue with the query execution. However, the ResultSet interface implemented by the datastores' wrapper and other CQE components is not JDBC compliant and, consequently, the CQE needs to implement the Proxy [Proxy] pattern to provide access to it though the JDBC ResultSet interface. The class name that implements the Proxy pattern is called WrapperResultSet. The following UML diagram shows the class structure described.

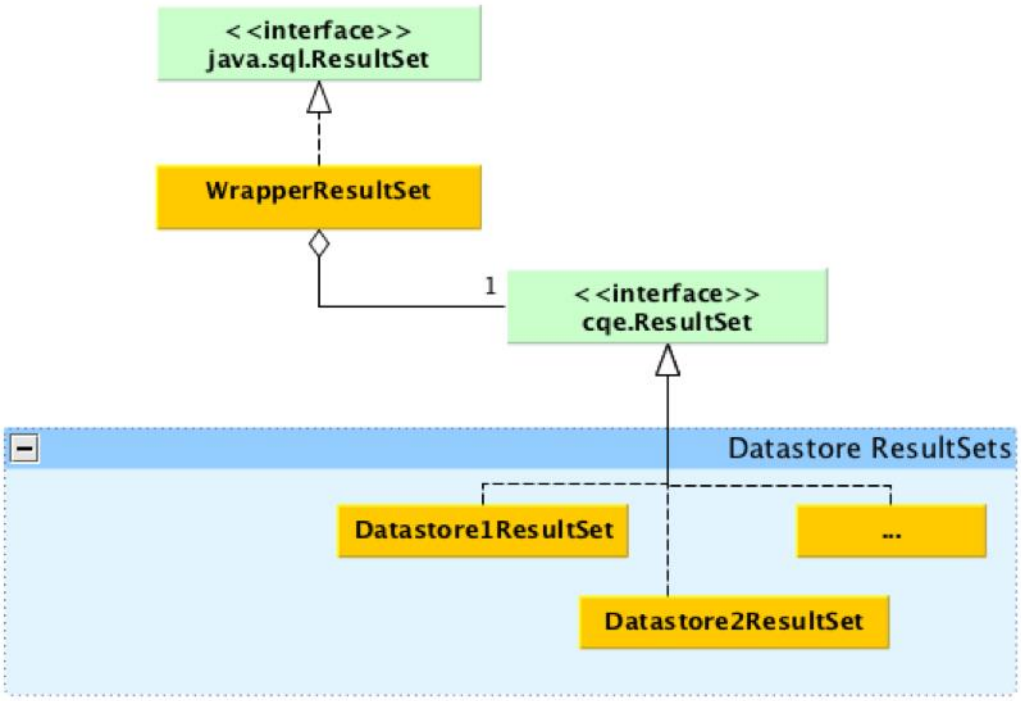

Figure 19: UML Flow Diagram of WrapperResultSet

The main responsibility of the WrapperTable class is to ask for the Wrappers API Statement implementation associated with a specific named table that has been previously prepared in a specific query context. Afterwards, the WrapperTable, needs to:

1. Set all the received parameter values in the statement. For example setInt(parameterName, parameterValue);
2. Add all depending named table expression statements invoking the useNamedTable method on the Statement instance;
3. Call the execute method with the corresponding transactional context;
4. Create a WrapperResultSet instance, which is a JDBC ResultSet implementation that internally contains the ResultSet implementation returned by the Wrappers API Statement interface, which does not implement the JDBC ResultSet interface;
5. Store the WrapperResultSet inside the QueryContext.

Python Statements Execution

Python named table expressions type allows writing Python expressions. Additionally, the CloudMdsQL language specification defines some conventions when the user types a Python query to publish rows and read rows from other named table expressions.

In order to read rows from other named table expressions, there is a special Python object *CloudMdsQL* that holds the context of the current CloudMdsQL query execution. It can be used in the Python code of a named table expression to instantiate other named expressions available in the context of the same query. Let us see the following example:

```
T2( x int, len_y int )@db1 = {*

 . ..

 rs = CloudMdsQL.T1

 while rs.next:

  yield (rs.getString(1))

 . ..

 rs.close()

*}
```

Table 19: Python code of a Named Table Expression to Instantiate other Named Expressions

In order to execute Python code inside the CQE, we have used Jython, a Java library that interprets Python code from the Java Virtual Machine. This library implements the Java Scripting API (JSR 223). Therefore, before running a Python code, the developer can define a scope, which sets for a given name a Java object, which can be referenced from the Python code. The CQE uses this feature through the PythonStatementImpl class to set the special CloudMdSQL object before running the Python code.

The CloudMdsQL object is created by the CQE before instantiating the Python statements. Concretely, the CQE generates a Python class with a method for each named table expression and having the corresponding QueryContext as a member, and then creates a Python object of this class using Jython. The following Python code is an example of a generated method that represents a named table expression.

```
def T1(self):

  return self.ctx.getResultSet("T1");
```

Table 20: Example of Python Code which generated Method that Rapresents a named Table Expression

Afterwards, every time the CQE needs to create a PythonStatementImpl, it invokes a setter with this context object.

Another important convention is the way the user may produce rows from Python code. For this purpose, the CloudMdsQL language uses the reserved keyword **yield**. In order to collect those published rows using the Jython library inside the PythonStatementImpl, the statement object generates a function that contains the Python code written by the user. Moreover, the statement initialises a variable with the output returned by the invocation of that function.

For example:

```
def f4387375834() :

 . ..

 rs = CloudMdsQL.T1

 while rs.next:

  yield (rs.getString(1))

 . ..

 rs.close()

r4387375834 = f4387375834()
```

Table 21: Convention for Producing Rows from Python Code

Nested Queries Execution

Nested queries are named table expressions that need to be executed inside the CQE and are written in CloudMdsQL language. In the statement initialisation section, we have introduced that nested queries run inside of the EmbeddedStatement class, which is an implementation of the wrappers API Statement. Internally, when the CQE prepares an EmbeddedStatement, what happens is the same as with the main query plan. It runs the process of generating an SQL statement from an Operation node using the *OperationToSQLVisitor* class.

Once these are ready to execute, it continues the same workflow as the main CloudMdsQL plan.

Queries Optimisation

The queries optimisation is mainly performed when the execution plan is generated from a CloudMdSQL expression. Some new operations have been added to avoid expensive join operations: scalars and IN clause. Moreover, the compiler applies a very-well known approach called BIND JOIN to create more efficient execution plans for some queries.

Scalars

Scalars allow to execute a named table for a set of values. For example:

```
T0(c string WITHPARAMS x string)@db1 = (...)

T1 (a string, b string)@db2 = (...)

SELECT T0(a), b FROM T1
```

Table 22: Example of Scalars

This query has the following behaviour: For all the rows emitted by the named table T1, the named table T0 is executed for each of the values of the column 'a'. If T1 returns just a few rows, execute T0 for every row returned by T1 becomes more efficient than retrieving all the rows from T0 and after that, apply a JOIN. Data stores usually stores indexes in tables and thus, is more efficient to apply few selections at data store level, than sending all the data and resolve the JOIN at CQE level.

Moreover, the CQE does not know about the data stores' statistics. Consequently, if the user had written T0 JOIN T1, the worst execution plan would have been a nested loop, with the biggest table as the inner one.

Scalars expressions (e.g T0(a)) return a single value for row or tables. However, Apache Derby does not allow to execute functions that return tables (java.sql.ResultSet) as part of the SELECT statements. For example, if we had defined T0 with the same CREATE FUNCTION statement, pattern than T1, the CQE had executed something equivalent to:

```
CREATE FUNCTION

T0(

name VARCHAR( 50 ), -- `T0'

ctxt BIGINT, -- '234324',

X VARCHAR(50))

RETURNS TABLE (C VARCHAR(50))

LANGUAGE JAVA PARAMETER STYLE DERBY_JDBC_RESULT_SET READS SQL DATA
EXTERNAL NAME 'WrapperFunction.read'
```

Table 23: Return of the Scalars Expression

In order to generate executable SQL statements for Apache Derby with scalar expressions, we cannot invoke table functions that whose output is *DERBY_JDBC_RESULT_SET*. For this kind of tables, we need to create as many definitions as returning columns they have and their returning type is the same type than the returning columns. For example, in the case of T0, it should be:

```
CREATE FUNCTION

T0(

name VARCHAR( 50 ), -- `T0'

ctxt BIGINT, -- '234324'

X VARCHAR(50))

RETURNS java.lang.String

LANGUAGE JAVA PARAMETER STYLE JAVA NO SQL EXTERNAL NAME
'WrapperFunction.readString'
```

Table 24: Generate Executable SQL Statements for Apache Derby With Scalar Expressions

Notice that the Java method (*WrapperFunction.readString*) is different because it needs to return a Java object. Moreover, an additional detail is that Apache Derby requires that specified returned type (e.g java.lang.String) must be the same as the one returned by the method signature. Therefore, the WrapperFunction class, has the method:

```
public static String readString(String tableName, Long transactionId,
Object... args) throws Exception;
```

Table 25: Wrapper Function Method

In this case, *T0* has only one returning column. However, if *T0* had been defined as follows:

```
T0(c string , d string WITHPARAMS x string)@db1 = (...)
```

Table 26: T0 Definition

The user should specify the returning column in the select statement. For example, the previous main query should have been rewritten as follows:

```
SELECT T0(a).c, b FROM T1
```

Table 27: Specification of the Returning Column

For this situations, the CREATE FUNCTION statement requires an additional parameter to specify the index that refers to the returning column that reports the execution plan for the evaluation of the T0(a).c expression.

```
CREATE FUNCTION

T0(

name VARCHAR( 50 ), -- `T0'

ctxt BIGINT, -- '234324'

SCALAR_INDEX INTEGER, X VARCHAR(50))

RETURNS java.lang.String

PARAMETER STYLE JAVA NO SQL LANGUAGE JAVA EXTERNAL NAME
'WrapperFunction.readStringAt'
```

Table 28: Specification Index for Returning Column

Notice that the WrapperFunction class has been enriched with a set of methods for the execution of scalar functions. Initially, this class only contained a generic read operation that returns a ResultSet, but now, according the explained modifications to support scalars, this class has read methods for each of the CloudMdsQL types (e.g readString) and for scalar calls for a specific column (e.g readStringAt). The final list of WrapperFunction methods is as follows:

```
public static ResultSet read(String tableName, Long transactionId,
Object... args) throws Exception;

public static String readString(String tableName, Long transactionId,
Object... args) throws Exception;

public static String readStringAt(String tableName, Long
transactionId,Integer position, Object... args) throws Exception;

public static Integer readInt(String tableName, Long transactionId,
Object... args)throws Exception;

public static Integer readIntAt(String tableName, Long transactionId,
Integer position, Object... args) throws Exception;

public static Double readDouble(String tableName, Long transactionId,
Object... args)throws Exception;

public static Double readDoubleAt(String tableName, Long transactionId,
Integer position, Object... args) throws Exception;

public static Float readFloat(String tableName, Long transactionId,
Object... args)throws Exception;

public static Float readFloatAt(String tableName, Long transactionId,
Integer position, Object... args) throws Exception;

public static Date readDate(String tableName, Long transactionId,
Object... args)throws Exception;

public static Date readDateAt(String tableName, Long transactionId,
Integer position, Object... args)throws Exception;

public static byte[] readByteArray(String tableName, Long transactionId,
Object... args)throws Exception;

public static Object[] readArrayAt(String tableName, Long transactionId,
Integer position, Object... args) throws Exception;

public static Object[] readArray(String tableName, Long transactionId,
Object... args)throws Exception;

public static Map readMap(String tableName, Long transactionId, Object...
args)throws Exception;

public static Map readMapAt(String tableName, Long transactionId, Integer
position, Object... args) throws Exception;

public static Long readLong(String tableName, Long transactionId,
Object... args)throws Exception;
```

```
public static Long readLongAt(String tableName, Long transactionId,
Integer position, Object... args) throws Exception;

public static Boolean readBoolean(String tableName, Long transactionId,
Object... args)throws Exception ;

public static Boolean readBooleanAt(String tableName, Long transactionId,
Integer position, Object... args) throws Exception;

public static Serialisable readSerialisable(String tableName, Long
transactionId, Object... args)throws Exception;

public static Serialisable readSerialisableAt(String tableName, Long
transactionId, Integer position, Object... args) throws Exception.
```

Table 29: WrapperFunction class methods for the Execution of Scalar Functions

IN Clauses

The CloudMdSQL clauses support IN clauses, which is an efficient approach to resolve JOIN operations when one of the tables return few values. Let us see the following CloudMdSQL.

```
T0(a string WITHPARAMS x string)@hbase = {* tbl.get($x) *}

T1(a string, b int)@leanxcale= ( SELECT a, b FROM tbl WHERE c IN
T0('abc') )

SELECT * FROM T1 WHERE a IN T0(a)
```

Table 30: CloudMdSQL Clauses support In Clauses

This query has a subquery evaluated by a key-value data store, which returns a single row and another one, which returns a set of values from a relational data store. Finally, the main query plan is evaluating the IN clause without a nested loop, which had implied to sort both tables before joining.

There are two types of IN clauses:

- For a limited list of constant values (e.g IN (1,2,3,8,9))
- For a subquery(e.g IN T0(a))

Each of these types has a specific operation in the execution plan: InList and InQuery. The translation to an SQL AST is performed different for each case:

- InList generates a InListOperatorNode
- InQuery generates a IN SubqueryNode.

Bind JOIN

CloudMdsQL uses bind join as an efficient method for performing semi-joins across heterogeneous data stores that uses subquery rewriting to push the join conditions. For example,

the list of distinct values of the join attribute(s), retrieved from the left-hand side subquery, is passed as a filter to the right-hand side subquery.

Bind join for SQL subqueries. To illustrate the classical bind join method, let us consider the following CloudMdsQL query Q1:

```
T1(id int, x int)@db1 = ( SELECT a.id, a.x FROM a )

T2(id int, y int)@db2 = ( SELECT b.id, b.y FROM b )

SELECT T1.x, T2.y FROM T2 JOIN BIND T1 ON T2.id = T1.id
```

Table 31: Bind Join for SQL Subqueries

To process this query the CQE will use the bind join method where the join condition will be bound to the right-hand side of the join operation. First, the table *T2* is retrieved from the data store *db2* using its query mechanism. Then, the distinct values of *T2.id* are used as a filter condition in the query that retrieves the table A from its data store. Assuming that the distinct values of *T2.id* are *b1* … *bn*, the query to retrieve the right-hand side relation of the bind join uses the following SQL approach (or its equivalent according to the data store's query language), thus retrieving from *T1* only the rows that match the join criteria:

```
SELECT a.id, a.x FROM a WHERE a.id IN (b1, …, bn)
```

Table 32: Processing the Query Table 31

In order to provide a relevant efficiency with the bind join query execution plan, the query compiler internally performs a translation of the originally written query Q1 and generates a plan that actually corresponds to the query below:

```
T1(id int, x int)@db1 = ( SELECT a.id, a.x FROM a WHERE a.id IN T2_id() )

T2(id int, y int)@db2 = ( SELECT b.id, b.y FROM b )

T2_id(id int) = ( SELECT DISTINCT id FROM T2 )

SELECT T1.x, T2.y FROM T2 JOIN T1 ON T2.id = T1.id
```

Table 33: Processing the Query Table 31

In this rewritten query, the intermediate named table *T2_id* is generated to return the distinct values of *T2.id* and an IN clause predicate is pushed down to the *T1* subquery to filter only those rows from *T1* that match the join criteria. Notice that the IN operator in the subquery to the data store *db1* uses a reference to the named table *T2_id*. In order to perform this operation, the final subquery to retrieve the table *T1* is composed at runtime by the *db1* wrapper, which invokes the table storage to get the result set of the intermediate table *T2_id*.

Bind join profitability. In the above example, using bind join will be reasonable only in the presence of an index on the column *a.id* in data store *db1*, because such a presence will make the

pushed down IN operator avoid an expensive table scan. In order to be able to determine possible bind join profitability, the query compiler may get from data stores information about the availability of indexes. Such metadata can be contained within the specification of data store capabilities that each data store exposes to the query compiler.

Bind join for native/Python subqueries. The way to do the bind join analogue for native/Python queries is through the use of a *JOINED ON* clause in the named table signature. For example, let us consider a modification of the query Q1, where *T1* is a native query defined as the Python function below.

```
T1(id int, x int JOINED ON id REFERENCING OUTER AS T1_Outer_id)@db1 = {*

 for id in CloudMdsQL.T1_Outer_id:

  yield ( id, db.get_x(id) )

*}
```

Table 34: Bind Join for Native/Python Subqueries

As *T1.id* participates in an equi-join, the values *b1* … *bn* will be provided to the Python code through the iterator *T1_Outer_id* (declared in the signature) that corresponds to the set of join key values from the outer table (the term “outer table” is referred to the other side of the join). Analogously to the processing of bind join for SQL subqueries, the query compiler generates the intermediate named table:

```
T1_Outer_id(id int) = ( SELECT DISTINCT id FROM T2 )
```

Table 35: Bind Join for Native/Python Subqueries

Update Operations

Use cases need to apply update operations on several data stores under the same CloudMdSQL statement. The CQE supports updates through native and transform query plans. It means that update operations can be performed through native data store queries or using SQL-like operations (also referred as named actions) that can be interpreted by each wrapper that support transform query plans. Let us see an example:

```
A0(WITHPARAMS x int, y string)@leanxcale =

( INSERT INTO tbl(a, b, c) VALUES ($x, $y, 'c'))

EXECUTE A0(0, 'hello')
```

Table 36: Update Operations

In order to translate the EXECUTE call into an SQL expression, the CQE translates it to the following pattern:

```
SELECT 0

FROM A0 UNION A1 UNION A2...
```

Table 37: Translation of the Execute call into an SQL Expression

Where A0, A1,...An are the function tables referenced from the EXECUTE sentences. The update statements executed from a function tables (INSERT, DELETE, UPDATE) has a specific operation in the execution plan processed by the CloudMdsQL compiler.

Wrappers Overview

A Wrapper provides the interface that attaches a data store to the CloudMDS Query Engine (CQE). It handles fragments of the execution plan that are intended for execution in a data store and delivers interim results in the appropriate format.

The Wrapper Interface and implementations are based on the Java platform and conceptually similar to the standard JDBC interface, namely, regarding the usage of Driver/Connection/Statement objects to provide nested context for query execution, and a ResultSet to iterate over result data. Namely, the main entry point to the wrapper is the DataStore interface, which provides Connection contexts for the execution of Statements. The query is received as a Java mapping of the JSON data model. Interaction with the Table Store is, in both directions, through the ResultSet batch iterator interface.

Jena JDBC Wrapper

Apache Jena is a free and open source Java framework for building Semantic Web and Linked Data applications. It has a JDBC driver (Jena JDBC) that allows access to the data whichever of the several possible Jena server implementations is used. That makes Apache Jena a good candidate to use from the CQE through a new datastore wrapper that will provide an easy integration of the IT2Rail ontology data.

Jena JDBC is a set of libraries which provide SPARQL over JDBC driver implementations. This is a pure SPARQL over JDBC implementation, there is no attempt to present the underlying RDF data model as a relational model through the driver and only SPARQL queries and updates are supported. It provides type 4 drivers in that they are pure Java based but the drivers are not JDBC compliant since by definition they do not support SQL. Jena JDBC aims to be a pure SPARQL over JDBC driver, it assumes that all commands that come in are either SPARQL queries or updates and processes them as such.

The Jena JDBC Wrapper added to the CQE is based on the python native queries wrapper and uses the Jena JDBC driver to access the Jena data. There are actually three Jena drivers provided currently: In-Memory, TDB and Remote Endpoint.

All of them are supported by our new wrapper just by using the right options in the wrapper config file:

Jena JDBC In Memory Wrapper Configuration

The In-Memory Jena driver uses an in-memory dataset to provide non-persistent storage. It’s useful for quick testing and very simple databases, usually provided directly using an ontology web language (OWL) data file.

An example of the Jena wrapper configuration file to access an in memory Jena datastore:

```
wrapper.class = com.sparsity.wrappers.jenawrapper.JenaDatastore

wrapper.name = jena

jena.driver = mem

jena.database = dataset=/home/root/cqe/samples/JenaDB.owl
```

Table 38: Jena JDBC In Memory Wrapper Configuration

Jena JDBC TDB Wrapper Configuration

TDB is a component of Jena for RDF storage and query. It supports the full range of Jena APIs. TDB can be used as a high performance RDF store on a single machine.

An example of the Jena wrapper configuration file to access a TDB Jena datastore:

```
wrapper.class = com.sparsity.wrappers.jenawrapper.JenaDatastore

wrapper.name = jena

jena.driver = tdb

jena.database = location=/home/root/cqe/tdbdata
```

Table 39: Jena JDBC TDB Wrapper Configuration

But Jena TDB it's meant to be directly accessed from a single JVM at a time otherwise data corruption may occur. So for the IT2Rail project, the Fuseki component which provides a SPARQL server that can use TDB for persistent storage and provides the SPARQL protocols for query, update and REST update over HTTP is higly recommended.

Jena JDBC Remote Endpoint Wrapper Configuration

Apache Jena Fuseki is a SPARQL server. It can run as an operating system service, as a Java web application (WAR file), and as a standalone server. It provides security (using Apache Shiro) and has a user interface for server monitoring and administration. It provides the SPARQL 1.1 protocols for query and update as well as the SPARQL Graph Store protocol.

Fuseki is tightly integrated with TDB to provide a robust, transactional persistent storage layer, and incorporates Jena text query and Jena spatial query. It can be used to provide the protocol engine for other RDF query and storage systems.

An example of the Jena wrapper configuration file to access a remote Jena datastore:

```
wrapper.class = com.sparsity.wrappers.jenawrapper.JenaDatastore

wrapper.name = jena

jena.driver = remote

jena.database =
query=http://localhost:3030/ds/query&update=http://localhost:3030/ds/upda
te
```

Table 40: Jena JDBC Remote Endpoint Wrapper Configuration

MySQL JDBC Wrapper

MySQL is one of the most popular Open Source full-featured relational database management systems (RDBMS) based on SQL and is currently developed, distributed, and supported by Oracle Corporation.

MySQL provides standards-based drivers for JDBC, ODBC, and . Net enabling developers to build database applications in their language of choice. In addition, a native C library allows developers to embed MySQL directly into their applications. This MySQL JDBC driver is used by our CQE MySQL Wrapper to access the MySQL datastore.

An example of the MySQL wrapper configuration file to access a MySQL datastore:

```
wrapper.class = com.sparsity.wrappers.mysqlwrapper.MysqlDataStore

wrapper.name = mysql

mysql.host = hostWhereMySQLRuns

mysql.port = 3306

mysql.options = ?user=TheUsername&password=ThePassword

databaseName = TheDatabaseName
```

Table 41: MySQL JDBC Wrapper

MySQL is a database that can be used with SQL through JDBC, so the MySQL Wrapper is based on the CQE generic JDBC Wrapper described in a following section.

Generic JDBC Wrapper

The Generic JDBC Wrapper handles both NATIVE and TRANSFORM query plans. The first already include native SQL statements ready to be executed. The latter are converted to SQL statements using the *cqe.sqlgen* utility package and then sent for server-side execution in the data store. As shown in the figure, corresponding JDBC entities are kept for each of the Wrapper Interface entities.

A key design principle of the Generic JDBC Wrapper is that it allows queries to be compiled once and reused a number of times. This means that translation from a JSON plan to SQL and generation of a JDBC PreparedStatement, that corresponds to SQL compilation in the data store server happen

at Wrapper Statement prepare. To fully achieve this, parameter values have to be dealt with and not simply embedded in-line in the statement text.

Methods in the default implementation can be overridden to change and add functionality. This functionality can be divided in SQL generation and execution, as detailed in the next sections.

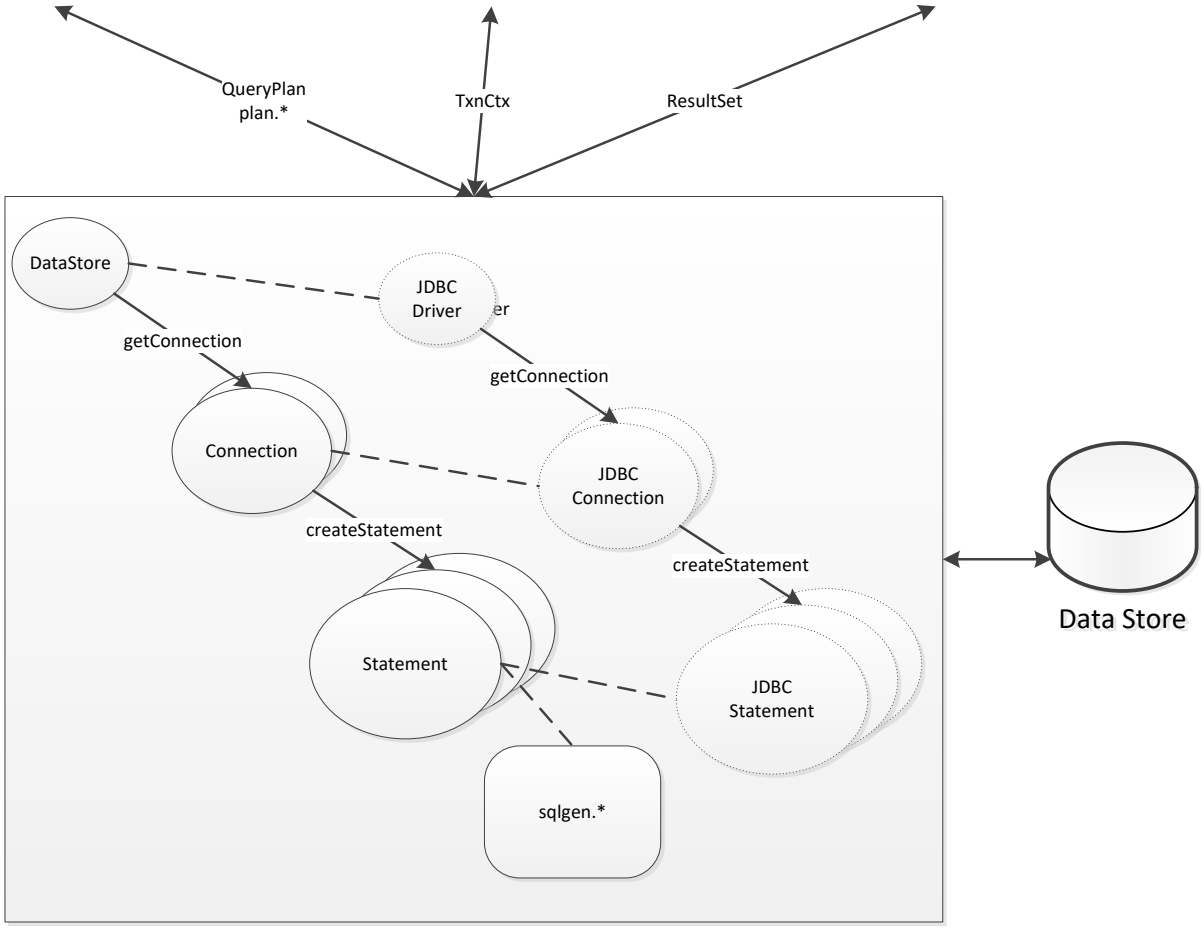

Figure 20: JDBC of the Wrapper Interface Entities

SQL Generator

The CQE package contains a generic converter from transformable query plan fragments to SQL using the Visitor pattern to traverse a tree of operators and expressions. It is thus conceptually similar to the main query plan compilation described before, but with several key differences.

First, the output of the SQL generator is a flat string conforming to SQL instead of an object tree. As an example, this is a challenge as a projection might have different meanings depending on the position in the tree, leading to additional conditions in the WHERE or the HAVING clauses depending on references to aggregate columns. Likewise, expressions used in different operators might end up in the result set, and be referred to by name or have to be included inline in WHERE and ORDER BY clauses.

Second, *ColRef* nodes that in an expression refer to columns in the input result sets to the operator at which the expression is rooted, refer to columns by name instead of position and allow for some

ambiguity. For instance, it is possible to generate SQL from an operator tree that does not contain a projection, thus leading to a star (*) in the projection clause.

Finally, SQL generation has to cope with potentially different implementations for Table Store access from the external data store. Namely, it is conceivable that a data store is able to call back into the Table Store through the Wrapper, or, as an alternative, that the Wrapper sets up a temporary table in the data store with the named table content just before execution. To achieve this the SQL generator provides as output, in addition to the SQL statement itself, a collection of named tables used and the expressions to be used as parameters when calling into the Table Store.

The SQL generator also handles named parameters by inserting a question mark (?) in the SQL statement text and providing the mapping between names and indexes, such that when the CQE provides parameter values just before execution, these can be routed to the appropriate location in the underlying JDBC prepared statement.

Statement Execution

Before execution, the CQE sets a value for each of the statement parameters. The Generic JDBC Wrapper routes the values to the corresponding indexes in the underlying JDBC PreparedStatement. The values are also cached, in case they are referred by expressions that need to be evaluated as parameters to named tables.

Then the CQE provides callbacks for each of the named tables in the form of Parameteric instances. These allow setting parameters and obtaining a result set. At this stage, the Generic JDBC Wrapper just caches them.

The execution of a statement then has three main stages:

1. Set up named tables. This means resolving expressions used as parameters, if any. If only constants or direct references to statement parameters are used, they can be used directly. Otherwise, expressions are executed in the data store to get values. The Table Store is then used to get a result set, which is used by the concrete implementation to make it available to the execution;
2. Invoke execution in the JDBC prepared statement object;
3. Wrap the JDBC result set in a Wrapper result set that converts each datum to the appropriate Java native data type and returns it to the CQE.

Finally, the Generic JDBC Wrapper also catches exceptions and converts them to corresponding exceptions in the Wrapper Interface.

New Wrappers Sample Query

The sample query below will show how to use both new wrappers in a simple CQE query that will query both datastores and join the results.

In this example, we have loaded a Product ontology sample file into an Apache Jena Fuseki server with the following data:

ID	Model #	Division	Line	Location	SKU	Stock
1	ZX-3	Manufacturing support	Paper machine	Sacramento	FB3524	23
2	ZX-3P	Manufacturing support	Paper machine	Sacramento	KD5243	4
3	ZX-3S	Manufacturing support	Paper machine	Sacramento	IL4028	34
4	B-1430	Control Enginnering	Feedback line	Elizabeth	KS4520	23
5	B-1430X	Control Enginnering	Feedback line	Elizabeth	CL5934	14
6	B-1431	Control Enginnering	Active sensor	Seoul	KK3945	0
7	DBB-12	Accessories	Monitor	Hong Kong	ND5520	100
8	SP-1234	Safety	Safety valve	Cleveland	HI4554	4
9	SPX-1234	Safety	Safety valve	Cleveland	OP5333	14

Table 42: New Wrapper Sample Query

While a MySQL server contains information about the Sales managers of each location in the following two tables:

- Managers:

ID	Name	City ID	…
1	John Smith	1	…
2	Jane Doe	4	…
…	…	…	…

- Cities:

ID	Name	Population	…
1	Sacramento	493,025	…
2	Elizabeth	124,960	…
3	Seoul	10,290,000	…
4	Hong Kong	7,374,900	…
5	Cleveland	396,815	…
…	…	…	…

Then, with both datastores properly configured in the CQE wrapper config files, we can write a simple test application to run a query that gets the SKU code and the city of each product from the Jena datastore using SPARQL, gets the sales manager name of each city from the MySQL datastore using SQL and finally joins both results using SQL in the CQE derby engine.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;

class CQESampleQuery {

public static void main( String [] args ) throws Exception {

     // The CQE is based on Derby, so it can be used with the normal
Derby JDBC driver
     String EXTERNAL_DRIVER = "org.apache.derby.jdbc.ClientDriver";
     Class.forName(EXTERNAL_DRIVER);
     Connection c =
DriverManager.getConnection("jdbc:derby://cqehost.sparsity-
technologies.com:1527/db;create=true;");

     // Jena SPARQL query
     String jenaQuery = "PREFIX product:
<http://www.workingontologist.org/Examples/Chapter3/Product.owl#> "
               +"SELECT ?sku ?city "
               +"WHERE { "
               +"?prod product:Product_Manufacture_Location ?city . "
               +"?prod product:Product_SKU ?sku "
               +"} ";

     // T1 is like a "virtual table" resolved with a query to a Jena
database
     // Jena is not a SQL database, so it must be used with the CQE
python native query syntax
     String t1 = "T1( sku string, city string )@jena =”+
          ” {*\njr=jena.compute('"+jenaQuery+"')\nwhile jr.next():\n
yield(jr.getString(\"sku\"), jr.getString(\"city\"))\njr.close()\n*} \n";

     // T2 is a table resolved as a query to a different MySQL database
     String t2 = "T2( salesManager string, city string )@mysql =”+
          ” ( SELECT managers.name, cities.name FROM managers, cities
WHERE managers.city = cities.id ) \n";

     // The CQE query runs T1, T2 and the final SQL query that joins the
results
     String query = t1 + t2 + "SELECT T1.sku, T2.salesManager FROM T1, T2
WHERE T1.city = T2.city";

     PreparedStatement stmt = c.prepareStatement(query);
     stmt.execute();
     ResultSet rs = stmt.getResultSet();

     int count=0;
     System.out.println("SKU\tManager");
     while (rs.next()) {
```

```
            System.out.println(rs.getString(1)+"\t"+rs.getString(2));
            count++;
        }
        System.out.println("Total rows: "+count);

        rs.close();
        stmt.close();
        c.commit();
        c.close();
}

}
```

Table 43: Test Application to Run a Query that gets the SKU Code and the city of each product from the Jena Datastore using SPARQL

5.1.4 Broker with managed Microservices

At the moment of design, a scalable, decentralised, robust, secure and interoperable platform builds the basis for a microservices solution, based (as a technical facilitator) mainly in Sofia2 APIManager Module.

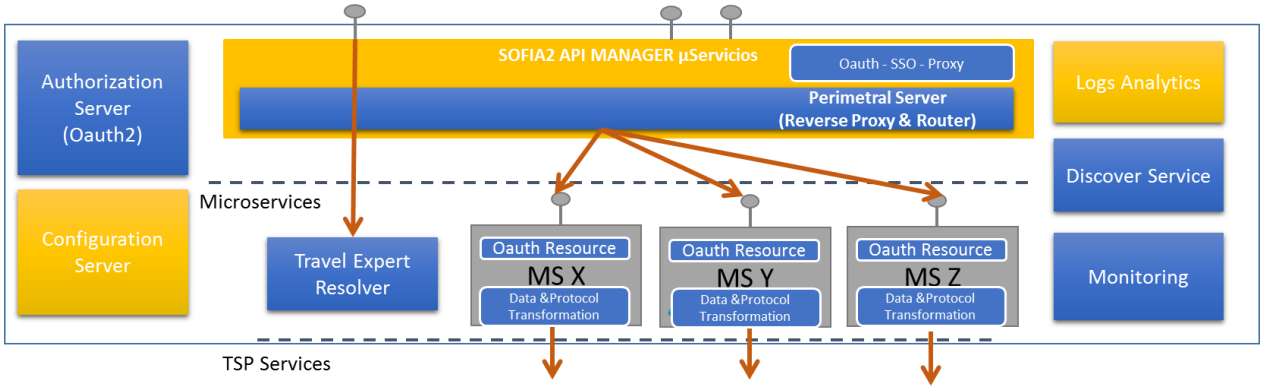

Figure 21: Broker with Managed Microservices

In this way, Sofia2 APIManager module will provide the capability of configuring a broker layer that will:

- Configure pass-through services to whatever TSP Service to be called;
- Capture useful information proactively, to be used to get statistics;
- Throttling and security control of the endpoints.

Integration with new Travel Service Providers will be done through a piece of SW development (bottom side of the diagram), with the following characteristics:

- In charge of data and protocol transformation, facilitating an abstraction layer between the actual TSP services and the functions required by the Semantic Broker.;
- The final goal of this layer will be to avoid new TSP to modify its actual systems, delegating any adaptation to the Broker on it.

The target architecture (so in that sense IT2Rail architecture will be the seed of it) will be based on microservices, with the benefits coming with it such as:

- Having several instances of each TSP integration, according to the actual volumetric expected per each one of them.
- Include typical capabilities of a microservices ecosystem, such as:
  - Centralised logs;
  - Service Discovery;
  - Monitoring;
  - Configuration Server;
  - Authentication and security configuration.

5.1.5 Sofia2 API Manager

One of the core features implemented in the Platform is the existence of an API Manager module. This module allows publishing services through well-known interfaces (such as HTTP REST) to "open" the platform to almost any system, given its standardisation (e.g. Smartphones, browsers, Enterprise systems, etc.). Devices and systems can use these interfaces instead of those published in the Acquisition Module if REST interface is preferred over the other ones published.

The main characteristics and benefits are listed below:

- Based on **standards** (JSON, REST, RESTful);
- **Security integrated** with the rest of the elements of the platform (authentication, authorisation, encryption…);
- **Publication** of data **independently** of the **repository** (real-time or historical);
- Open Data vs Data Monetisation;
- Complete control of the **life cycle of the APIs** (Created, In Development, Published, Deprecated, Deleted), **versioning;**
- **Throttling Control** (management of the number of requests that each user can make per minute);
- Custom Query Methods, API cloning, automatic generation of CRUDs…;
- Transparent integration of **third-party APIs, as external API.**

This last one is the scenario of usage identified for the project so, in this way the API Manager will be the pass-through layer (including all other capabilities such as security control, throttling or monitoring).

As a general benefit of Sofia2 platform, this module also will be configured from a user friendly common administration interface.

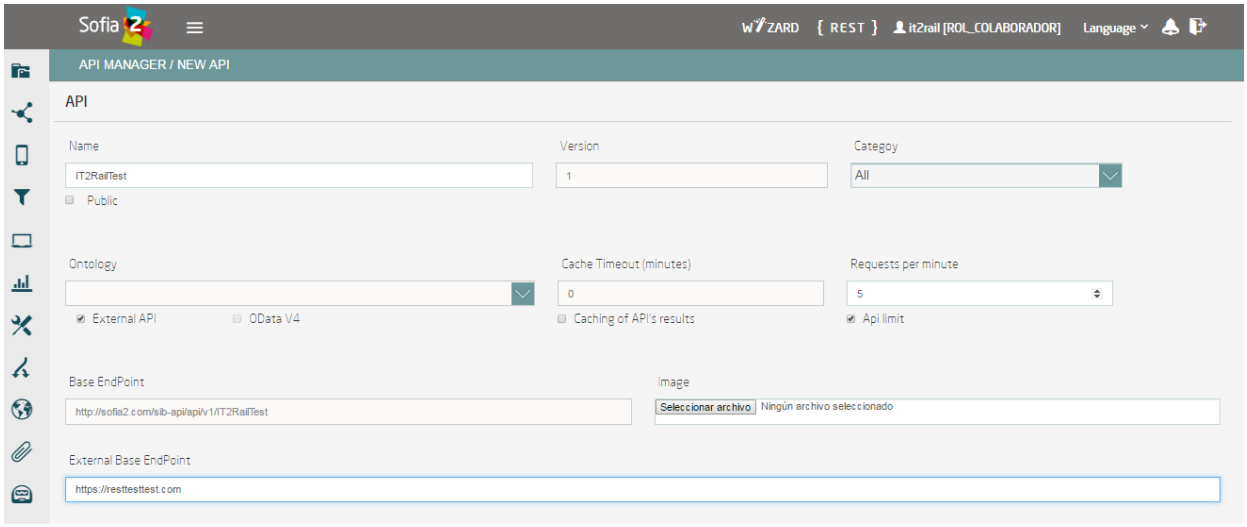

Figure 22: New External API Manager General Conf.

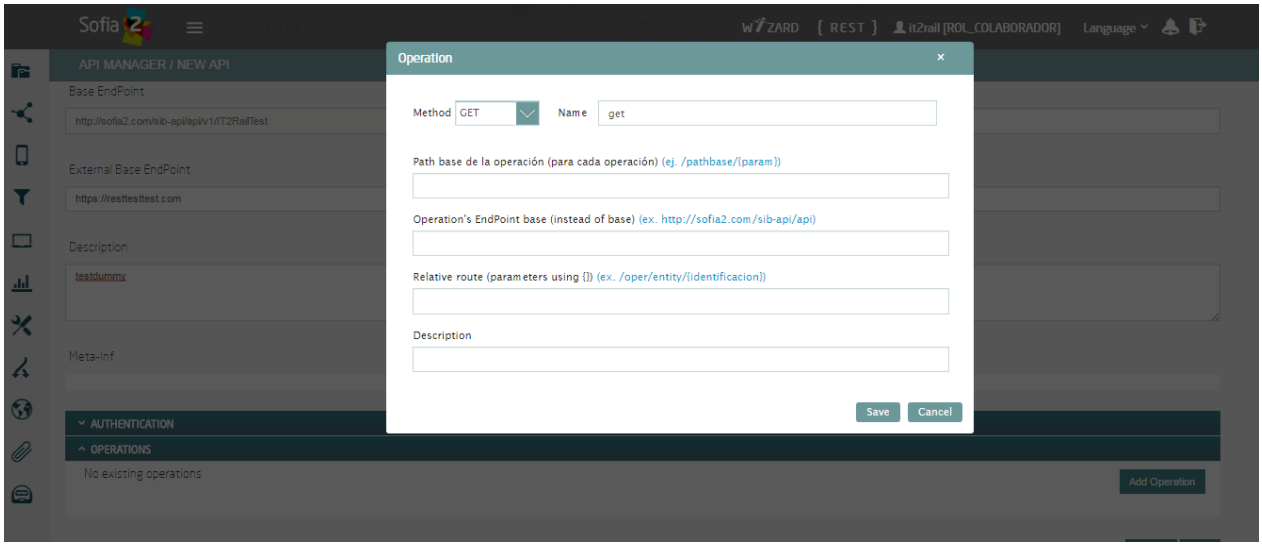

Figure 23: New External API Manager Operations Conf.

6. OPERATION

6.1 SOFIA2 TRAINING PLATFORM

6.1.1 Introduction

SOFIA2 is a middleware architecture allowing for the interoperability of several systems and devices. It allows making real information available for different intelligent applications (Internet of Things) that share semantic concepts to achieve interoperability among them. SOFIA2 is based on open source technologies, multi-platform, multi-language and communication agnostic.

The SOFIA2 platform can be conceptualised through these main four concepts:

- Smart Space: The virtual environment where the applications interoperate with each other, providing complex functionality. Its core is one SIB or a cluster of SIBs;
- SIB (Semantic Information Broker): It receives, processes and stores all the information from applications connected to SOFIA2, and reflects all the existing concepts in their domain (in ontologies) and their current states (in instances of the ontologies);
- KP (Knowledge Processor): Each of the devices implements the KP software functionality to communicate with the Smart Space using ontologies;
- SSAP (Smart Space Access Protocol): The standard messaging protocol to communicate between one SIB and either another SIB or a KP, autonomous from the underlying network.

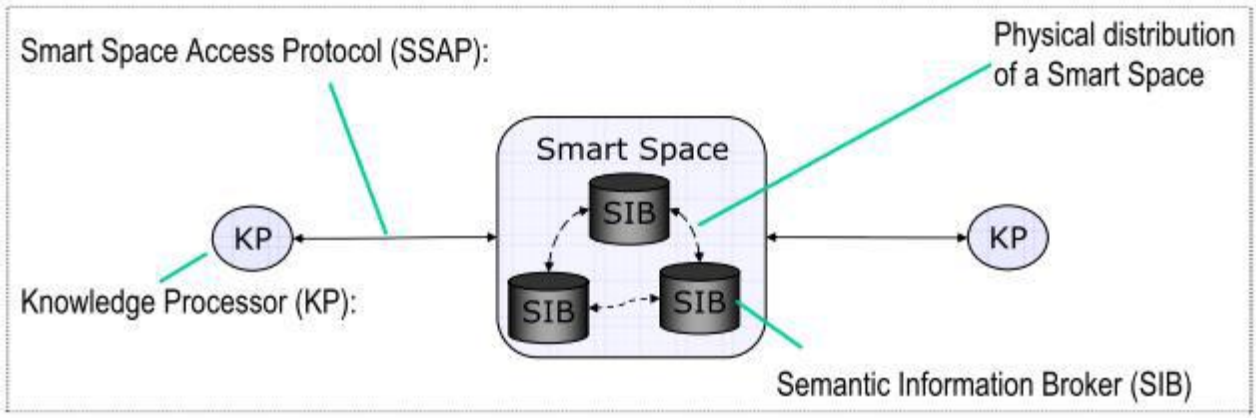

Figure 24: Smart Space Architecture

The ontologies are semantic descriptions of a set of classes. Applications sharing classes (commonly called concepts) from the same ontology can easily exchange information using specific instances of those common classes. SOFIA2 commonly represents ontologies in JSON format following a JSON Schema.

The development is performed in two different environments, based on the distributed ecosystem of SOFIA2: The client-side, related to the Knowledge Processor (KP); and the server-side, related to the Semantic Information Broker (SIB).

Web Console

The Web Console is used to manage the server side of SOFIA2. A user logs in the web console to register her KP on the server, so that the server accepts messages from it or send messages to it. The user can also register her ontologies on the server.

The Web Console's user can program actions to be performed on the server side when a given input takes place, including rules, scripts and queries. The user will upload the script which will be run on the server.

APIs and SDK

Following the IoT paradigm, any device in the physical world needs to send its data (e.g. a thermometer, a light sensor) or receive information (e.g. an alarm, an operation center) from the SOFIA2 server-side.

To do this, the device must communicate with SOFIA2's SIB. This is achieved through a software that first transforms the raw data into ontologies, then sends it to the SIB (or receives the ontologies and then transforms them into useful data). The software functionality to communicate with the SIB is known as Knowledge Processor or KP.

SOFIA2 offers APIs (application programming interfaces), which are specific libraries to generate the KP for the client devices. There are several APIs available for Arduino, iOS, C, Java, JavaScript, Android, Python, . NET and Node.js. The user can download any API and work with it in the development environment for the device's software before implementing the software in the device's runtime environment.

SOFIA2 also provides a SDK specific to the Java API based on Eclipse IDE for Windows, Mac and Linux.

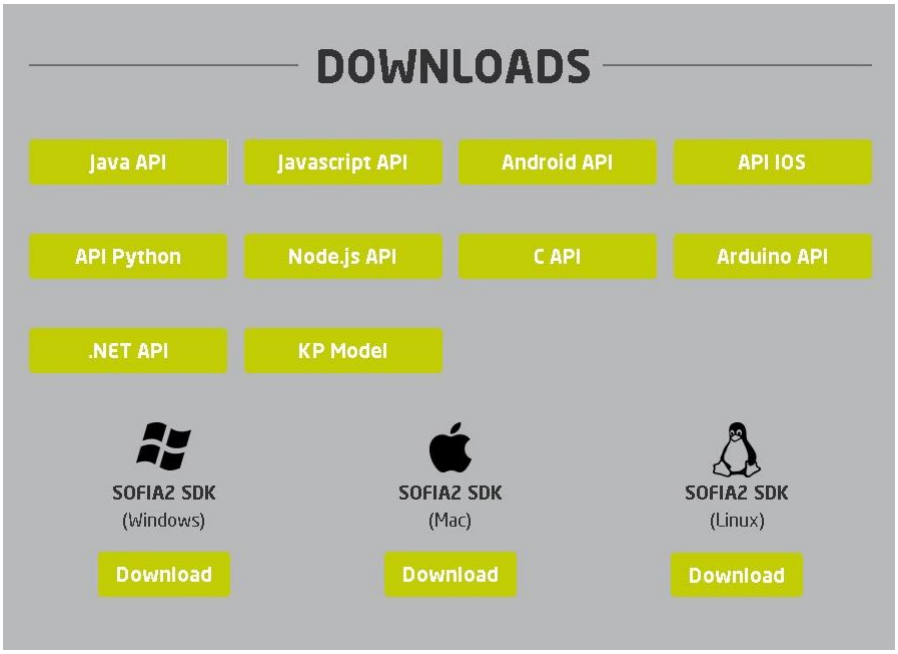

Figure 25: SOFIA2 Website Download Section

Support

SOFIA2 has a public blog, where developers explain the differences of new versions and provide specific training on popular modules of SOFIA2.

SOFIA2 has also added a dedicated support line implementing a new procedure to manage queries and incidences. It is a first-level support team centralising all the requests on the platform. Anyone can contact it simply e-mailing the generic user with contact data, kind of request and description. This new initiative speeds up the query management and helps the platform's constant evolution with the users providing new ideas.

Title	URL	Description
FIRST STEPS WITH SOFIA2	http://sofia2.com/docs/(EN)%20SOFIA2-First%20Steps%20with%20SOFIA2.pdf	How to set up communications, sending and receiving data
HOW TO DEVELOP WITH SOFIA2		Development guide for SOFIA2
SOFIA2 APIS	http://sofia2.com/docs/(EN)%20SOFIA2-SOFIA2%20APIs.pdf	Description of the different APIs provided by the SOFIA Platform to develop KPs
SCRIPT TOOLS DEFINITION		How to build scripts for the Sofia2 ontologies
SOFIA2 CLIENT ARCHITECTURE REFERENCE		Description of the client architecture and development guidelines
APIS SCRIPT	http://sofia2.com/docs/(EN)%20SOFIA2-APIs%20Script.pdf	APIS that can be used from the Scripting Rules engine of the Sofia2 Platform
CEP USE GUIDE	http://sofia2.com/docs/(EN)%20SOFIA2-CEP%20Use%20Guide.pdf	Capabilities of the CEP in the Sofia2 platform, and to know its configuration mechanisms
CEP GUIDE STEP BY STEP	http://sofia2.com/docs/(EN)%20SOFIA2-CEP%20Guide%20Step%20by%20Step.pdf	Mechanisms to set up and use the capabilities of the CEP

Table 44: SOFIA2 Development Documents

6.2 Services Available in SOFIA2 Web Console

SOFIA2 offers an online console where users can log in and access to a great number of services through drop-down menus in an intuitive graphic interface.

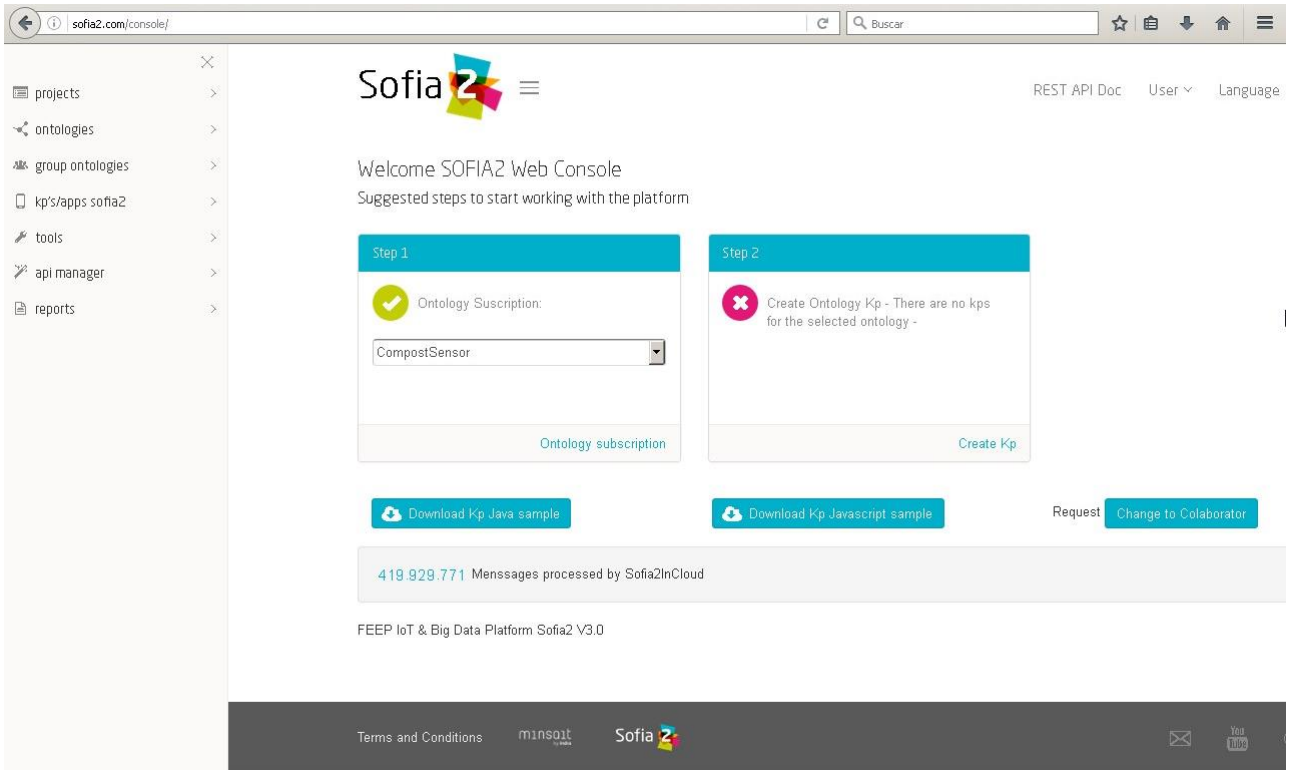

Figure 26: SOFIA2 Web Console with Menus

Some of the services available through the web console include:

- Create a new user with an associated password;
- Manage a user, changing its password or requesting a different user role;
- Change the interface language at any moment;
- Download KP sample source code in different programming languages;
- Monitor all the messages processed by Sofia2InCloud.

6.2.1 Description of roles

Accounts are created with the basic role USER, but they can request an administrator to change their role depending on their needs.

The actions available to a SOFIA2 user depend on that user's role. There are a number of user roles with different privileges:

- USER:
  - The role with less privileges;
  - A User can only access and subscribe to public ontologies and to those ontologies that the ontology owner authorised that User explicitly to see. The user cannot modify those ontologies;

- The user can create and manage KPs associated to the ontologies she is subscribed to;
- A User is the role for someone who simply needs to create and manage a SOFIA2 client in a smart space.

- COLLABORATOR:
  - The Collaborator has all the privileges of a User and more;
  - The Collaborator can create new ontologies, then define whether to make these public (allowing any user to subscribe to it) or open them specifically to some users;
  - A Collaborator is the role for someone who needs to define, schedule and program functionalities in the SOFIA2 server side.
- ANALYTICS:
  - This role has all the privileges of a Collaborator and more;
  - The Analytics role can get analytics on data from heterogeneous sources;
  - The Analytics role has access to predefined links between SOFIA2 and social networks including Facebook and Twitter;
  - An Analytics role is only required if that user is expected to work heavily with that field.
- ADMINISTRATOR:
  - An Administrator can perform all the Analytics' functions;
  - The Administrator also manages templates, roles and users;
  - Administrators can also access other specifically-designed software services;
  - An Administrator is a very specific role, restricted to the SOFIA2's technical staff.

6.2.2 Services available for each role (except Administrator)

This is a detailed table of the services available for each user's role. The Administrator's services are excluded from the list because those are restricted to the SOFIA2's technical staff.

SERVICES	ROLES		
Ontologies	**User**	**Collaborator**	**Analytics**
New Ontology		x	x
New Ontology Field to Field		x	x
Create Ontology from Excel		x	x
Load Status from Excel		x	x
My Ontologies		x	x
Crud Ontologies		x	x
Instance Generator		x	x
My Subscriptions	x	x	x
Ontologies Authorisations		x	x
Group Ontologies	**User**	**Collaborator**	**Analytics**
New Group Ontologies		x	x
My Group Ontologies		x	x
My Group Ontologies Subscriptions	x	x	x

Group Ontologies Authorisations		x	x
KPs/APPs SOFIA2	**User**	**Collaborator**	**Analytics**
My KPs/APPS	x	x	x
My Tokens	x	x	x
My KP/APP Instances	x	x	x
Rules	**User**	**Collaborator**	**Analytics**
New Rule using Wizard		x	x
My Scripts		x	x
My CEP Rules		x	x
My CEP Events		x	x
Assets	**User**	**Collaborator**	**Analytics**
My Assets		x	x
Visualise Assets		x	x
Asset types		x	x
Nodes		x	x
Visualisations	**User**	**Collaborator**	**Analytics**
My Gadgets		x	x
My Dashboards		x	x
Scada	**User**	**Collaborator**	**Analytics**
Start		x	x
My Tags		x	x
My Synoptics		x	x
My Alerts		x	x
Tools	**User**	**Collaborator**	**Analytics**
BDTR & BDH Console	x	x	x
Predefined Queries Management		x	x
Active KPs	x	x	x
Send SSAP Messages	x	x	x
JSON Message validation	x	x	x
Processes Status Visualisation		x	x
SW Management	**User**	**Collaborator**	**Analytics**
My SW Configurations		x	x
Configuration Assignment		x	x
API Manager	**User**	**Collaborator**	**Analytics**
My APIs		x	x
APISs Authorisations		x	x
APIs Subscription	x	x	x
My Subscriptions	x	x	x
My API Key	x	x	x
Social media	**User**	**Collaborator**	**Analytics**
Twitter Users		x	x
Search		x	x
Trending topics		x	x
Scheduled Search		x	x

Facebook pages		x	x
Trending Topics Instagram		x	x
Crawling Web		x	x
Brandwatch Search		x	x
Brandwatch Wizard		x	x
Settings Access		x	x
Reports	**User**	**Collaborator**	**Analytics**
New Template		x	x
New Report	x	x	x
My Reports	x	x	x
Report Authorisations		x	x
Analytics	**User**	**Collaborator**	**Analytics**
My Notebooks			x
Data link			x

Table 45: Services Available for Each Role

The services available for each role are listed in the following table and classified by their category. This document includes a description of the most useful services

The console is constantly being improved, with new services being added and new functionalities implemented as described in the SOFIA2 blog.

6.3 APIs AND SDK FOR SOFIA2 CONNECTED DEVICES

SOFIA2 offers a downloadable software development kit (SDK) with a completely operative version of the SOFIA2 platform, allowing anyone to program in the SOFIA2 environment in a very short time.

6.3.1 Installing the SOFIA2 SDK

The SOFIA2 SDK is available for download at with versions for MS Windows, Mac and Linux, along with several other utilities.

The compressed file includes everything needed to develop SOFIA2 programs (or APPS) without any previous installation in the computer. The user needs only to uncompress the file in a given folder and run Sofia2_SDK-START.bat to mount the virtual disk S: and open a command console to it.

Once this is done, the user can run Sofia2_IDE.bat from S: to open an Eclipse environment to use SOFIA2. By selecting **Window > Preferences > Maven > User Settings**, the user can update the field adding a settings.xml file in its virtual unit like **S:\SOFIA2-SDK\MAVEN\conf\settings.xml**

For training and example purposes, SOFIA2 provides several APIs in the download section. If a user downloads the Java API and simply unzips it in the path where the SDK is installed. then launches the Eclipse IDE, then the user can import the example included in the Java API by selecting **File> Import> General> Existing Projects into Workspace> "s:\SOFIA2_API_JAVA_30102014\Ejemplos\Sofia2_KP_Eclipse\"**

This project has three classes, implemented as JUnit Test and the necessary Tokens to insert in the instance of SOFIA2 on cloud are configured. The classes are:

- **KpGatewayRestFuncional** connect via REST with the instance of SOFIA2 on cloud;
- **KpMqttFuncional** connect via MQTT with the instance of SOFIA2 on cloud;
- **KpWebSocketFuncional** connect via WebSocket with the instance of SOFIA2 on cloud.

To run them, right click over any of them and click on Run As>JUnit Test. The output should be:

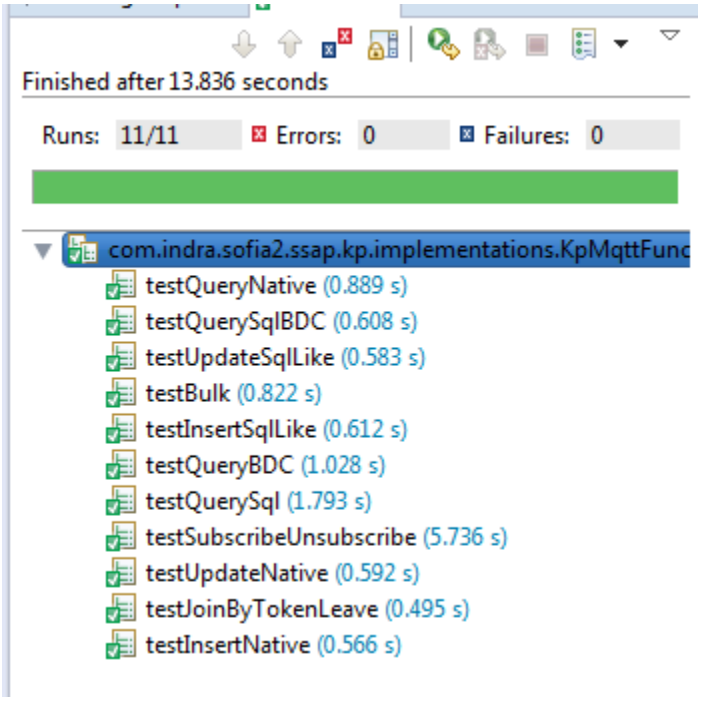

Figure 27: JUnit Test Successful

6.3.2 First steps with the SOFIA2 Console

INDRA provides a free distribution SOFIA2 On Cloud where users can run KP's easily through a web console, especially for training purposes.

To work with the console, the first step is creating an account at:

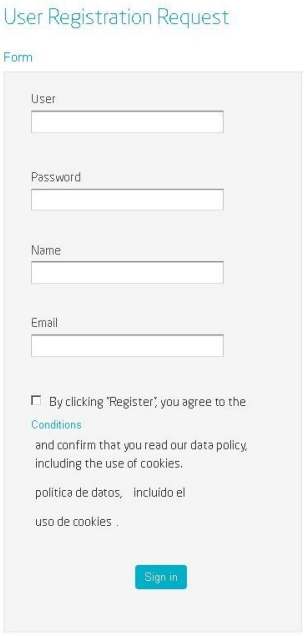

Figure 28: SOFIA2 Console Registration Page

If the user already has an account, then a login screen is available at:

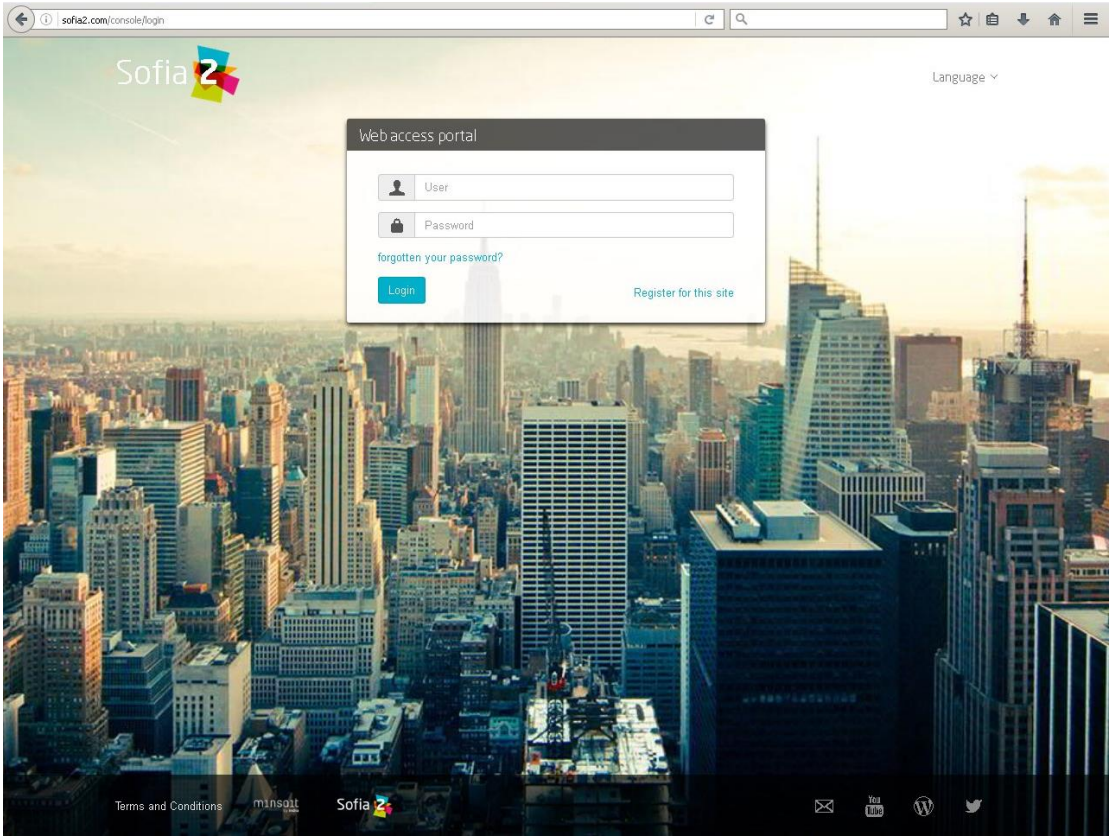

Figure 29: SOFIA2 Console Login Page

Accounts are created with the most restrictive available role, USER (USUARIO). The users can request administrators to upgrade their account to other roles.

Managing Ontologies and KPs

A User can subscribe to ontologies through the console simply by clicking a link to see available, public ontologies. Once an ontology has been created, its owner can allow other users to read, create, modify or delete it. Users can search an ontology, see its details including its JSON schema and subscribe to it using a graphic interface.

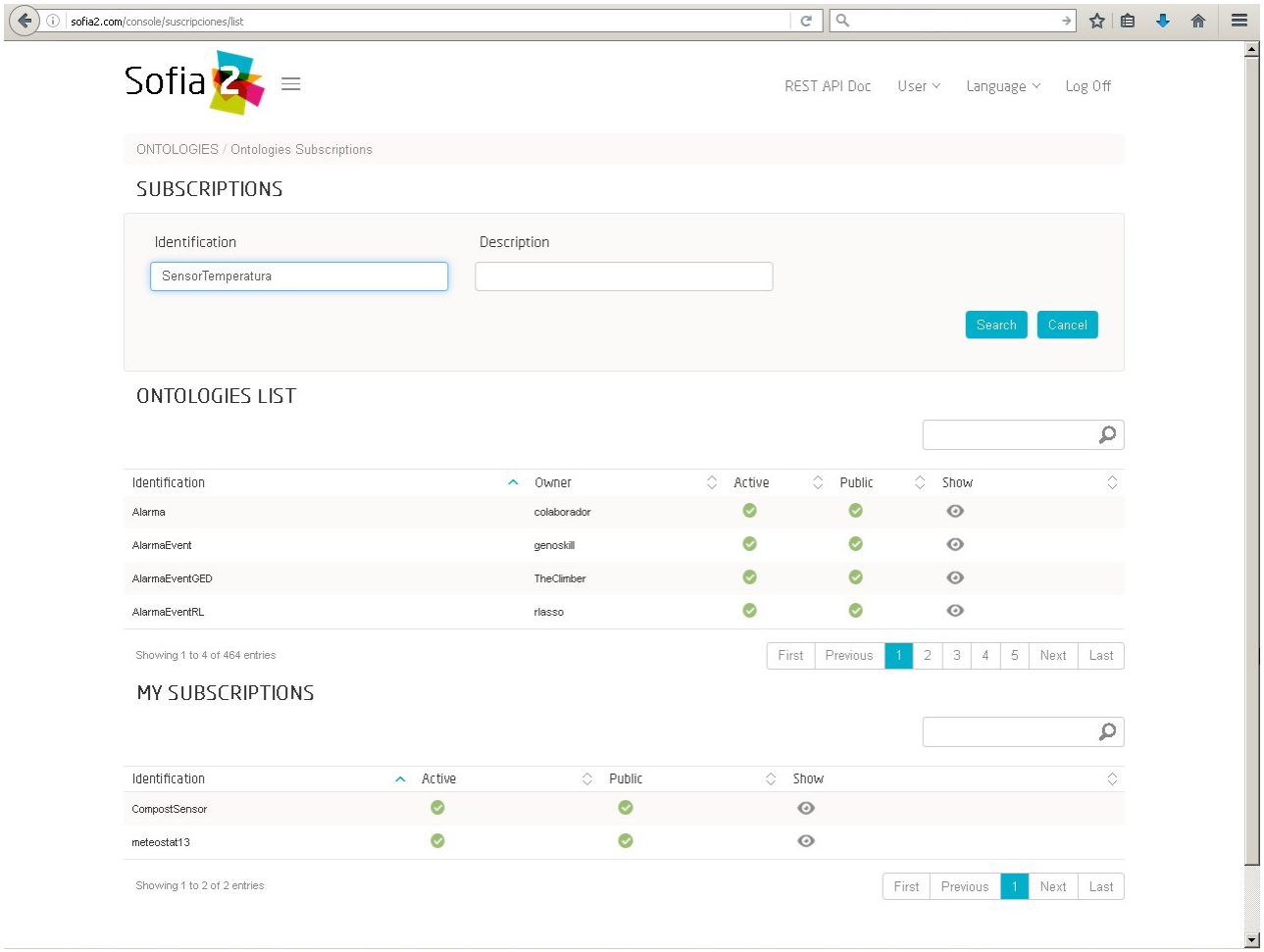

Figure 30: Console's Ontology Management Page

A User who has subscribed to one or more ontologies can create (register) a KP through the Console, simply selecting the ontologies from those she has already subscribed to.

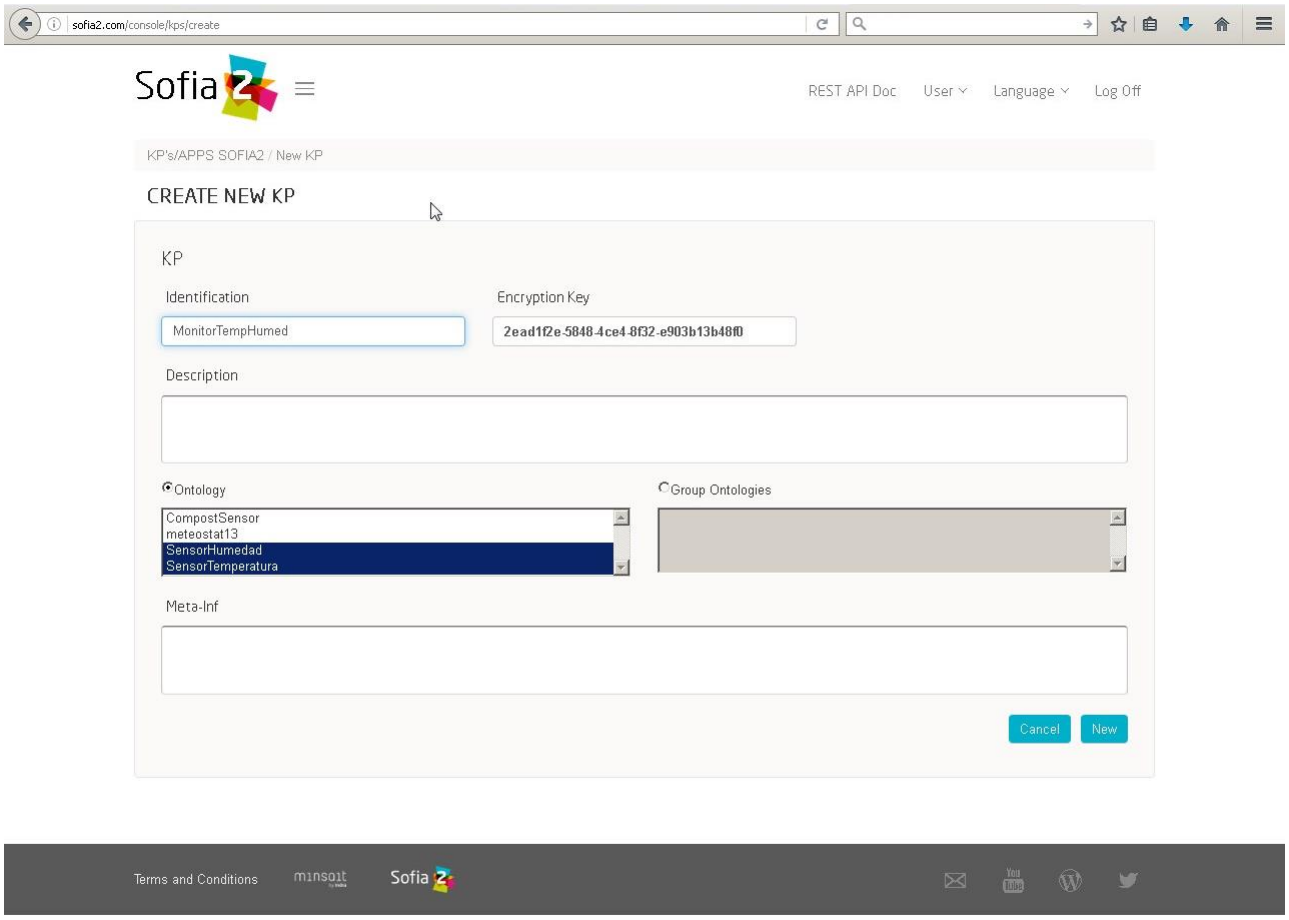

Figure 31: Console's KP Creation Page

This is a created KP:

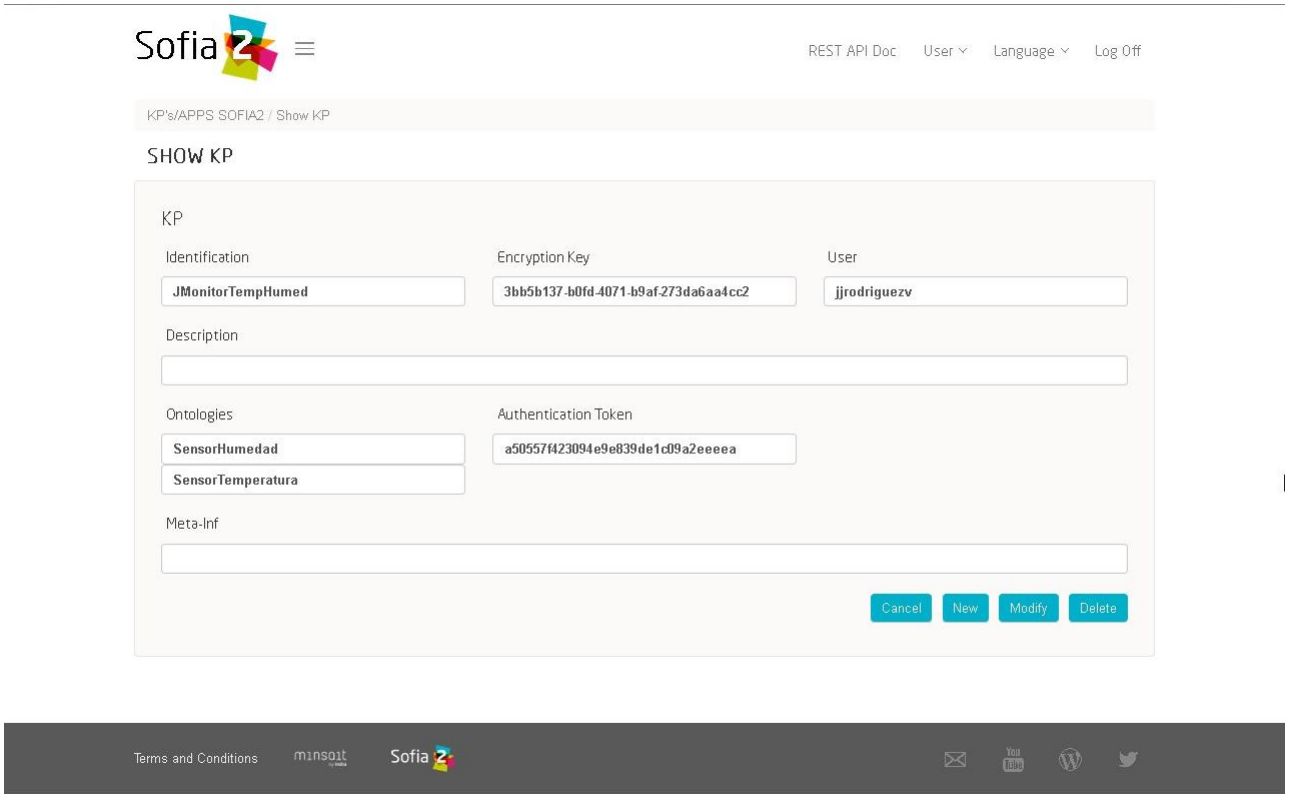

Figure 32: Console's KP Details Page

Once a KP has been created, the user can then generate and then activate authentication tokens for it and, if required, encryption keys (only if the KP is in a device that does not support HTTPS). Notice that the user can activate and de-activate the tokens through the console.

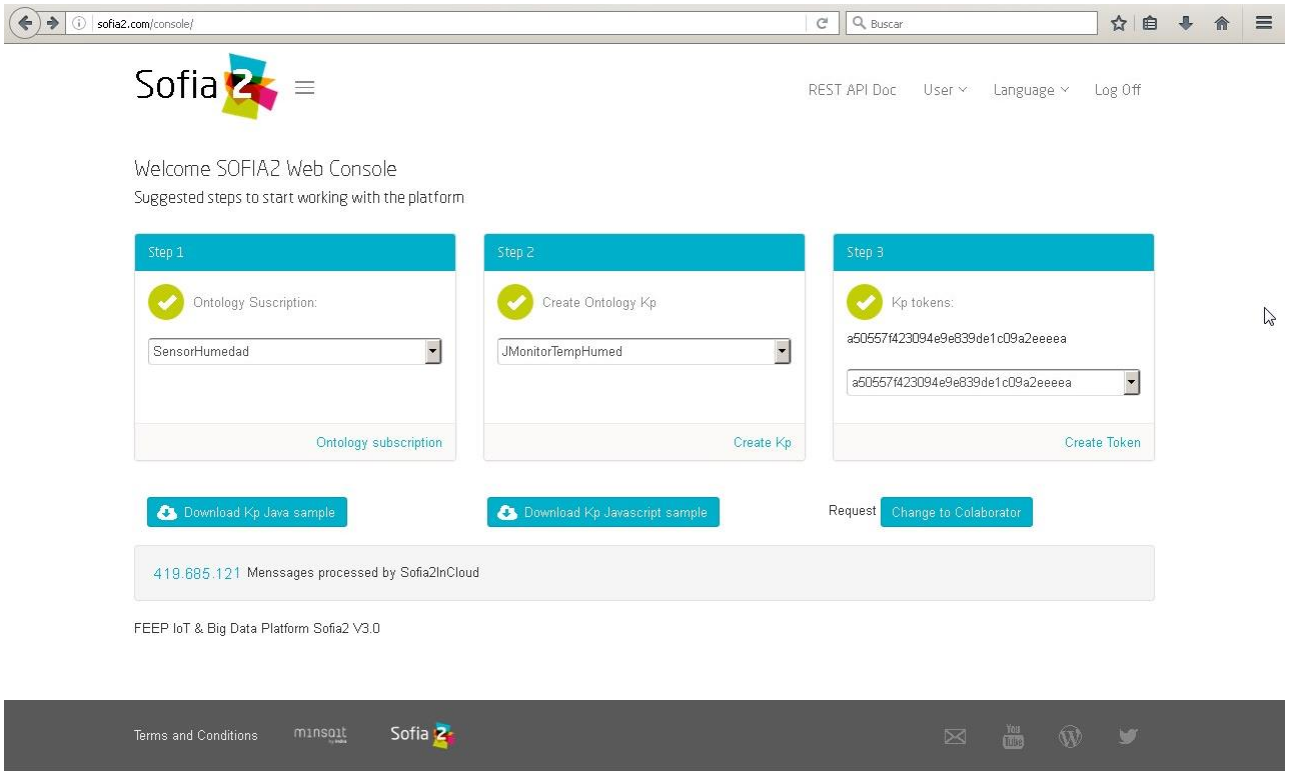

Figure 33: Console Page Showing Token for a KP

Remember that accounts with the User role do not have privileges to create new ontologies, but can work with instances of the ontologies they are subscribed to, for instance to use them in their own KPs.

The creation of ontologies is available from a menu only visible to Collaborators and Administrators. The interface provides the required fields and prevents the SOFIA2 users from inserting wrong values.

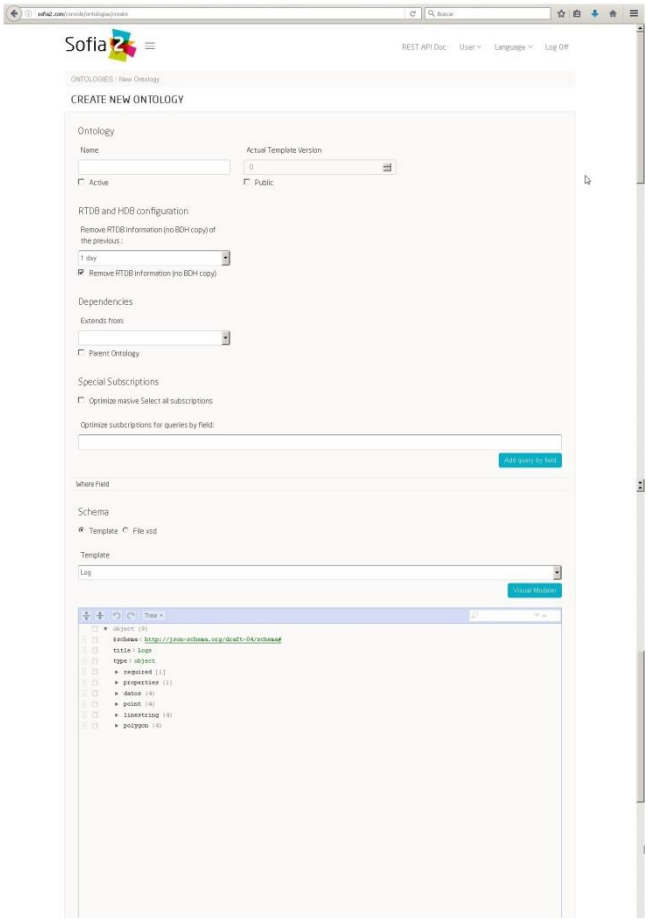

Figure 34: Console's Ontology Creation Page

An ontology can be created but deactivated to prevent its use before a launching date; and it has a number of optional fields that can be left blank such as the processing time of information stored in the RTDB to the HDB. Dependencies between ontologies can also be assigned, including creating father ontologies that cannot be instanced (and are only visible to Administrators and to the ontology's creator) but which can be extended in different daughter ontologies.

Alternatively, a user can create an ontology field by field, or expert it from a properly-formatted Excel or CSV file.

Ontologies can also be edited:

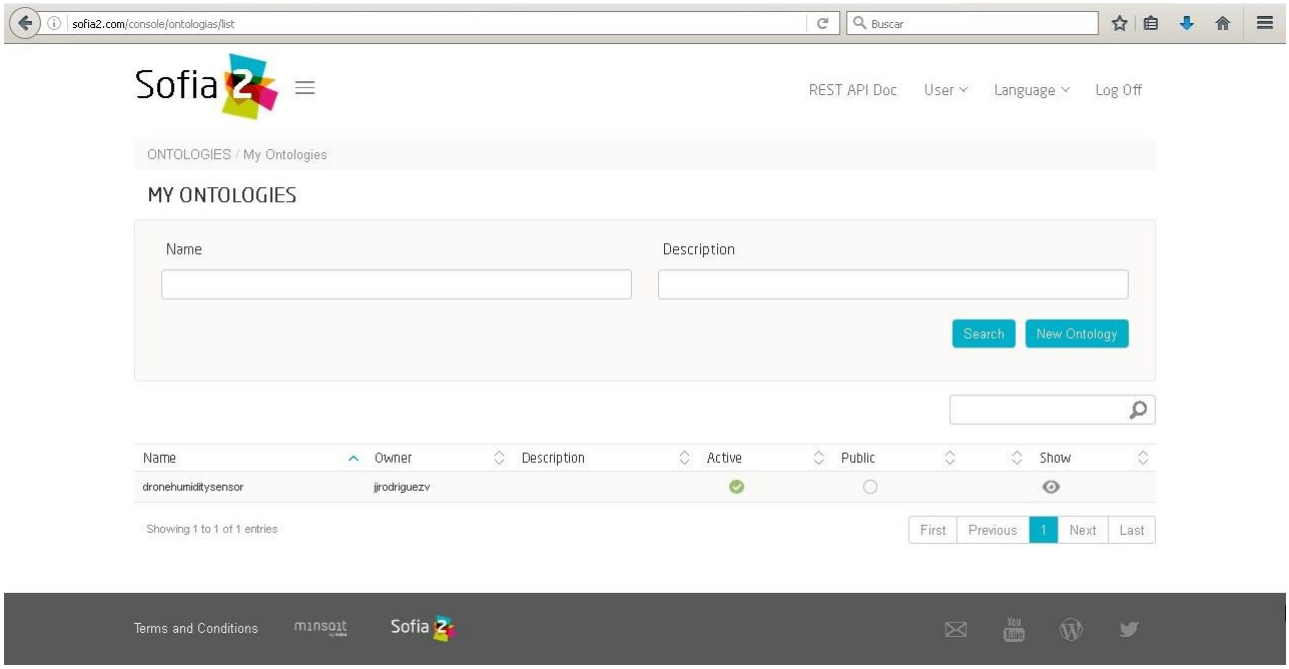

Figure 35: Console's My Ontologies Page

And invalidated (deleted):

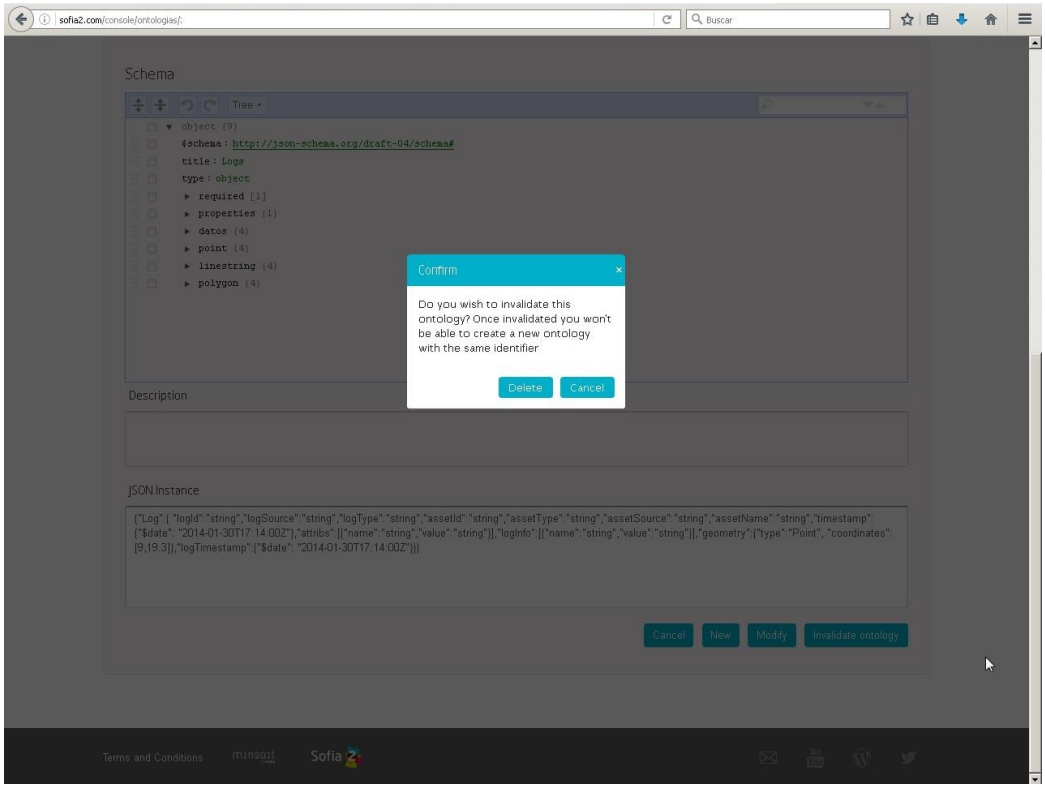

Figure 36: Invalidating an Ontology

The creator of an ontology can authorise specific users to see that ontology and subscribe to it. An Administrator can do this with any ontology, and even revoke a user's permission on an ontology.

Group ontologies operate in a similar way to normal ontologies.

Database queries

The user also has a database query interface to launch queries on the existing ontologies. These can be launched both to the Real Time Data Base (RTDB) and to the Historic Data Base (HDB), and they can be written in SQL or in native language. By double-clicking an ontology name, the FROM clause of the query automatically updates. The user can manually include filters with the SELECT clause. The output will be shown on-screen and can also be downloaded as an XML file, a MS Excel file or a text file with comma-separated values:

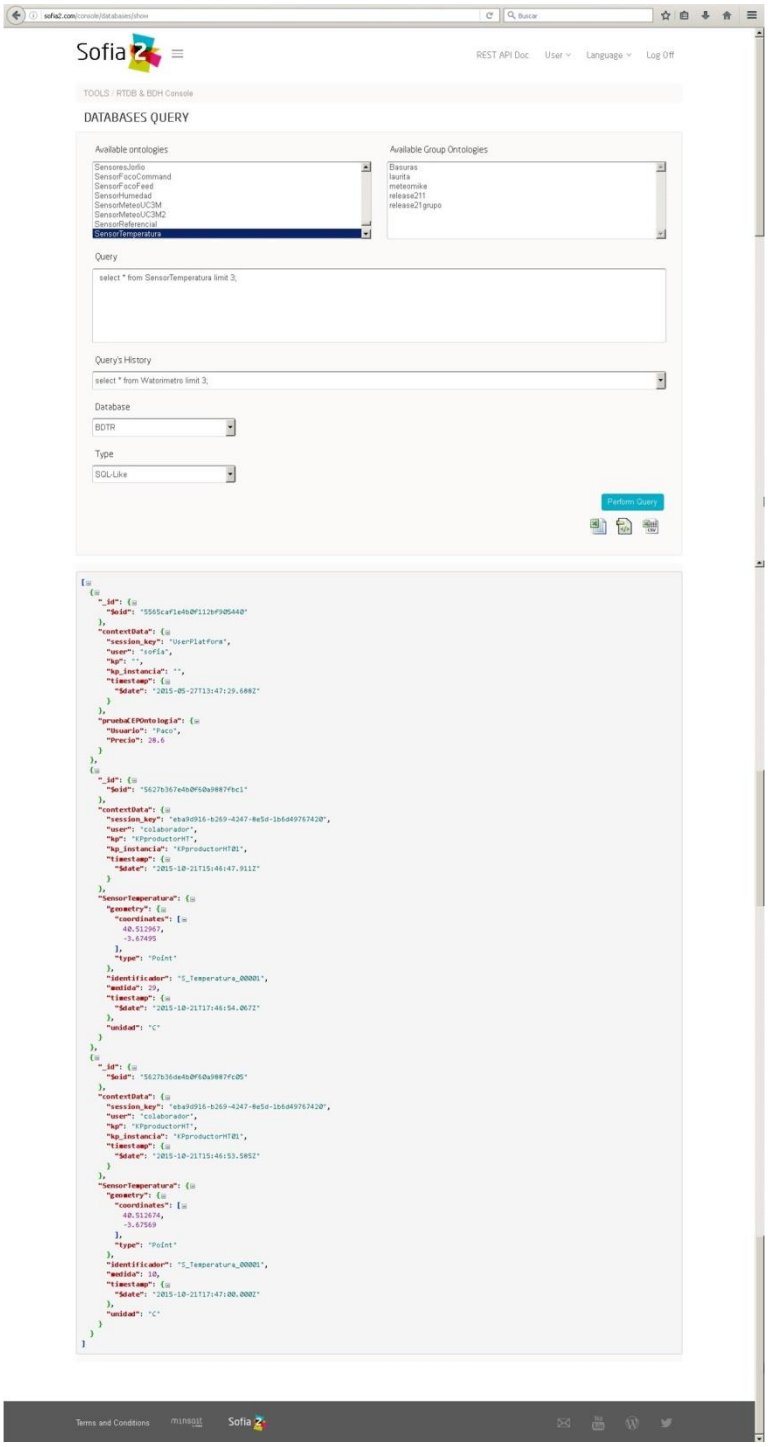

Figure 37: Console's Database Query Example

Message validation

The console offers a functionality to write an instance of an ontology, available from the menu under **Tools > SSAP Message Validation**. Here the user can select any of the ontologies she is subscribed to.

The console will automatically provide the JSON schema for the ontology, specifying all the required fields. This schema is available in tree format, as source code, as text or in a simple view screen similar to the tree.

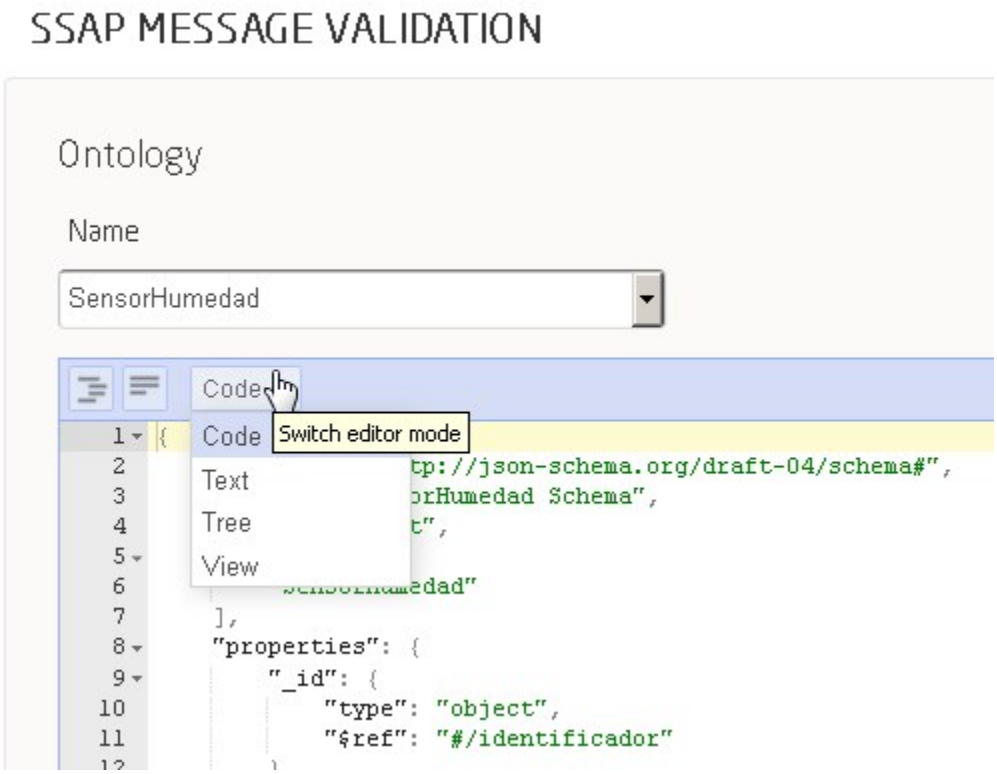

Figure 38: Console's SSAP Message Validation, Detail on Editor Modes

The user can then take the schema as a base to write her own ontology instance in a lower box, intended for that purpose:

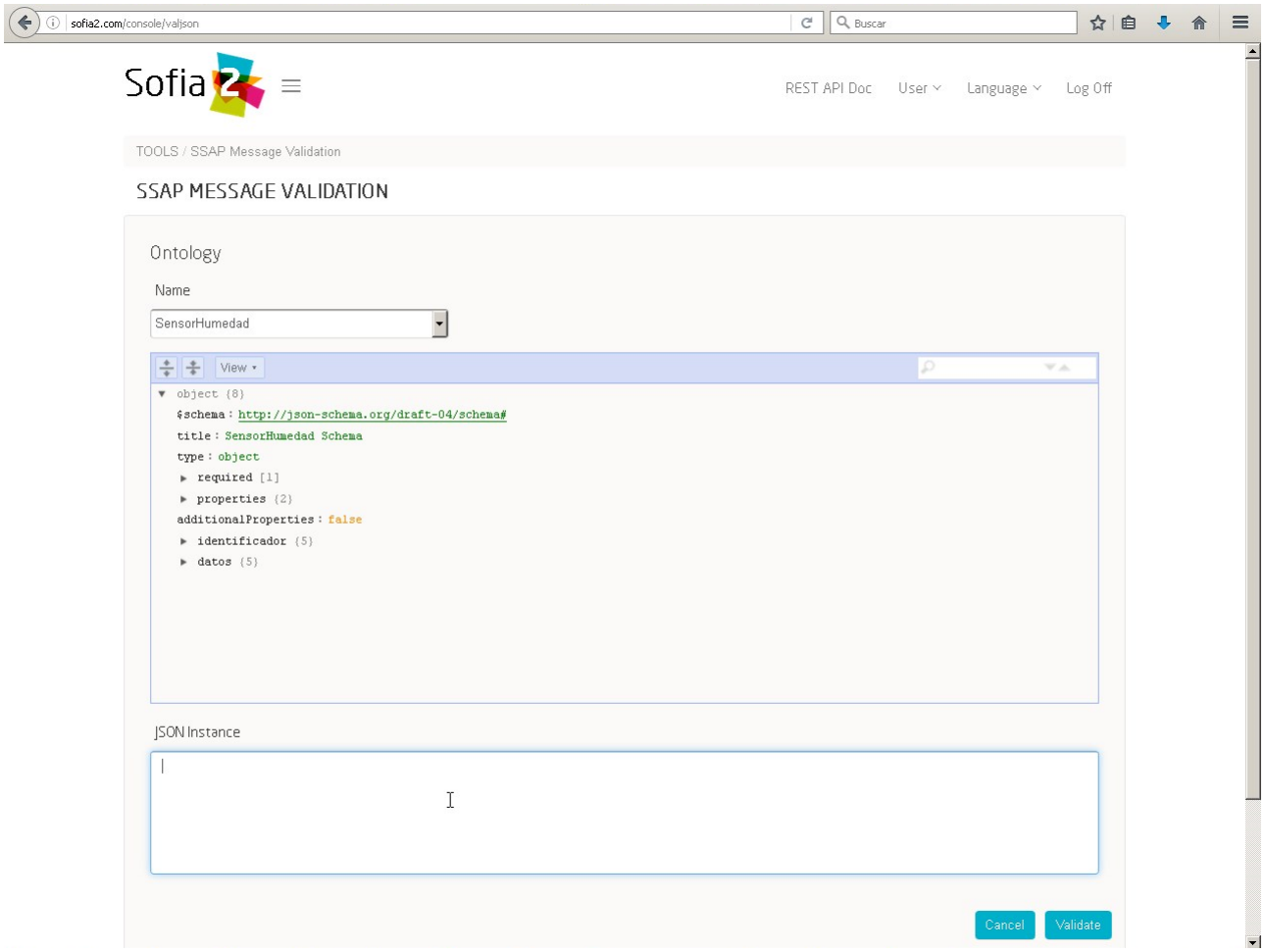

Figure 39: Console's SSAP Message Validation

The user has a Validate button in the lower part. When the example is finished, the user can select that button to confirm whether the ontology instance is valid or not:

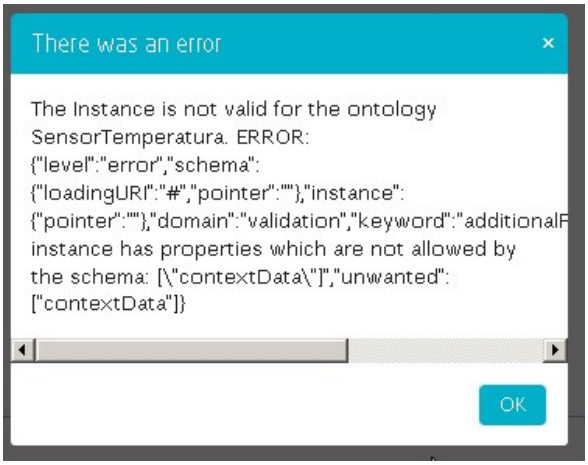

Figure 40: Console's SSAP Message Validation's Output

Rules and scripts

Collaborators can also create rules that react when a combination of ontology instances reach the SIB, automatically generating a response. The CEP is used to do this. To define a rule, the Collaborator has a number of options:

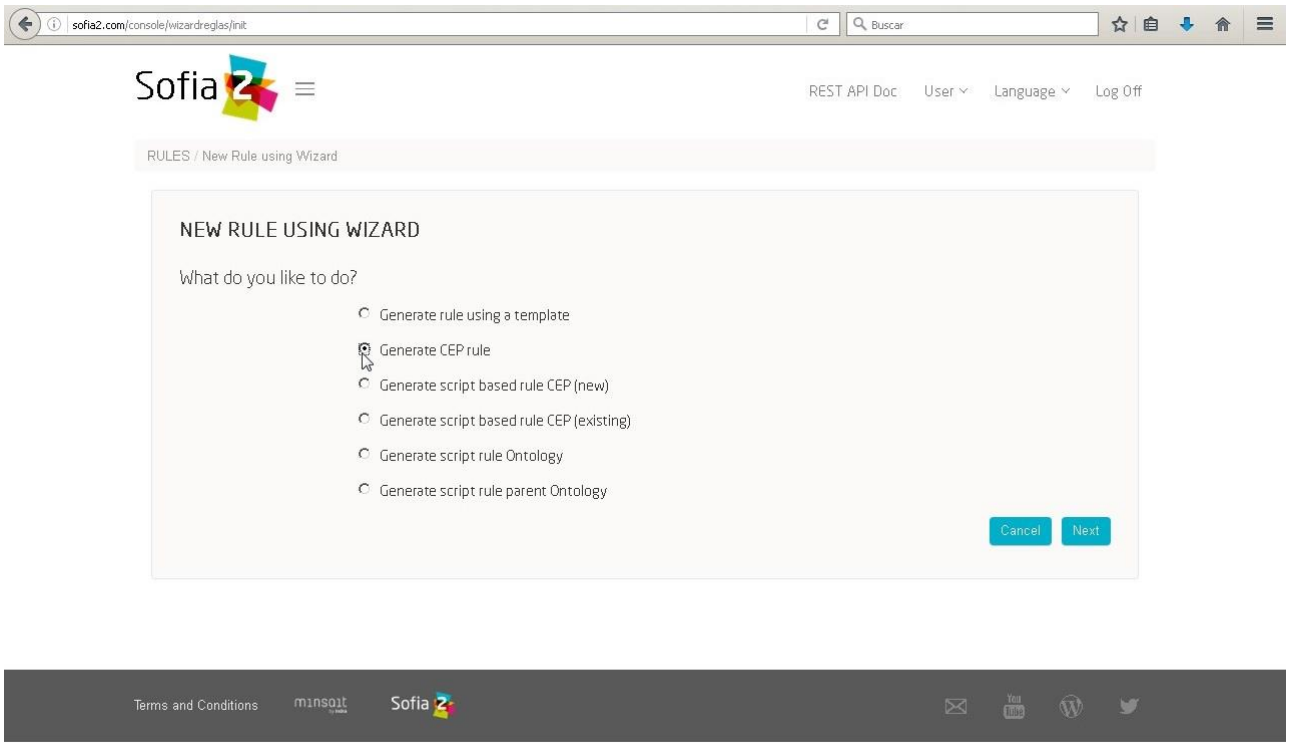

Figure 41: Console's Rule Creation

The user has a graphic interface to define the specifics of the rule:

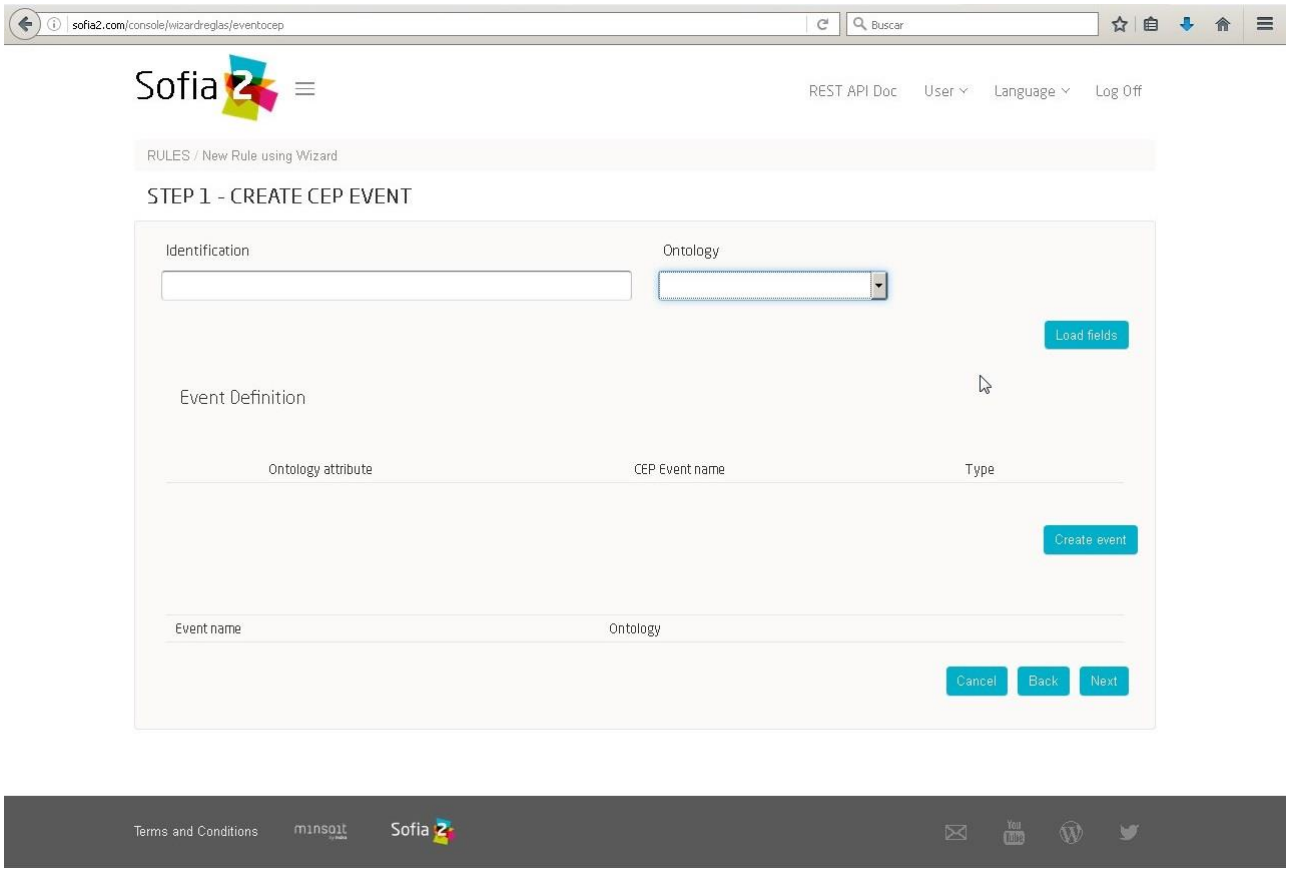

Figure 42: Console's CEP Rule Generation

SOFIA2 supports scripts reacting to events (e.g. ontology instances as inputs) or repeated after a given time. The scripts must be defined in Groovy language, and they can access APIs to perform actions such as sending an e-mail, insert/query/delete an instance of an ontology, invoke a URL or any authorised action that the user programs on the script (i.e. the role User cannot delete an ontology because it does not have that privilege).

Scripts are run on a safe environment to prevent from a script to be affected by an error in a different script. The scripts also have a time-out in their invocation.

The web interface helps in writing the source code for the script:

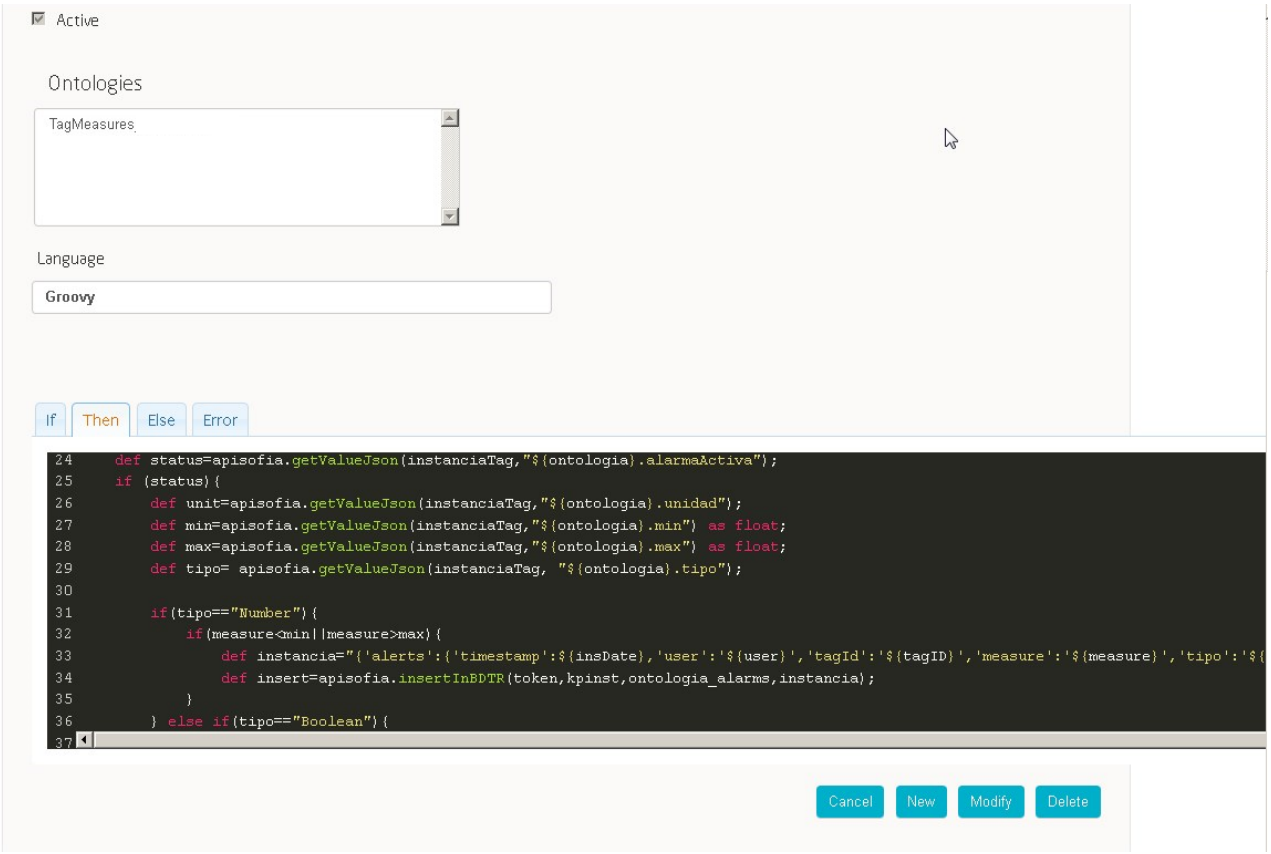

Figure 43: Writing Source Code for a Script

6.3.3 Developing APPS with SOFIA2: JavaScript example

Only after having deployed the SDK and created an account we can consider writing an app. This section will explain this process with a simple example: A software tool that allows us to monitor recent instances of an ontology, as long as know a valid token to do so. This token is available through the web console: Before the process, the user connects to the console and generates a token for any of the ontologies she is subscribed to.

SOFIA2 provides an example JavaScript APP in to be downloaded from https://sofia2.com/desarrollador.html, uncompressed and pasted on **S:** This is an HTML file that allows us to include information on the KP we created online:

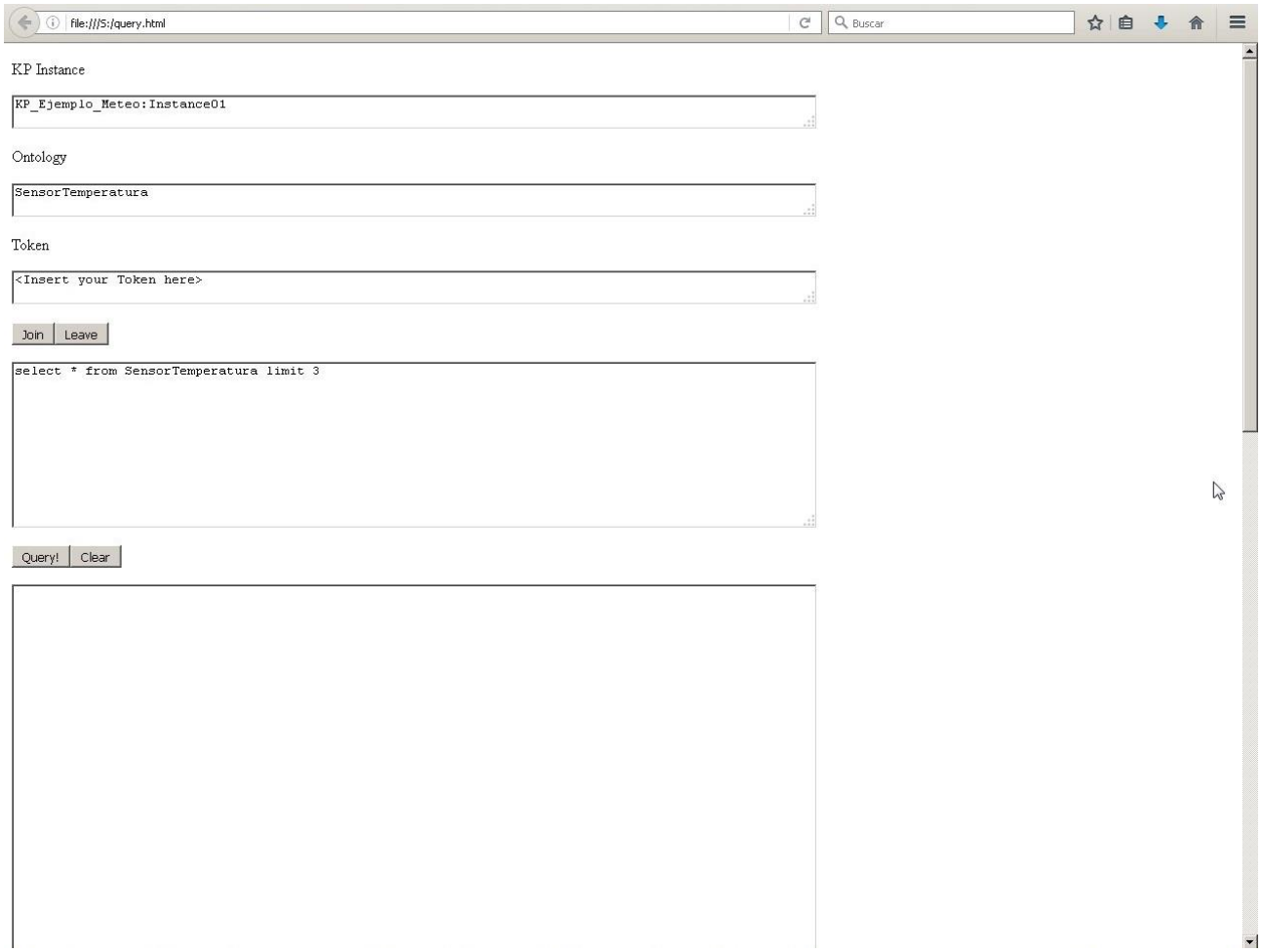

Figure 44: JavaScript APP Screen

If the user inserts her KP, one of the ontologies that the KP is linked to, and a valid token for the KP, then the user can press Join to get a valid session key for a session.

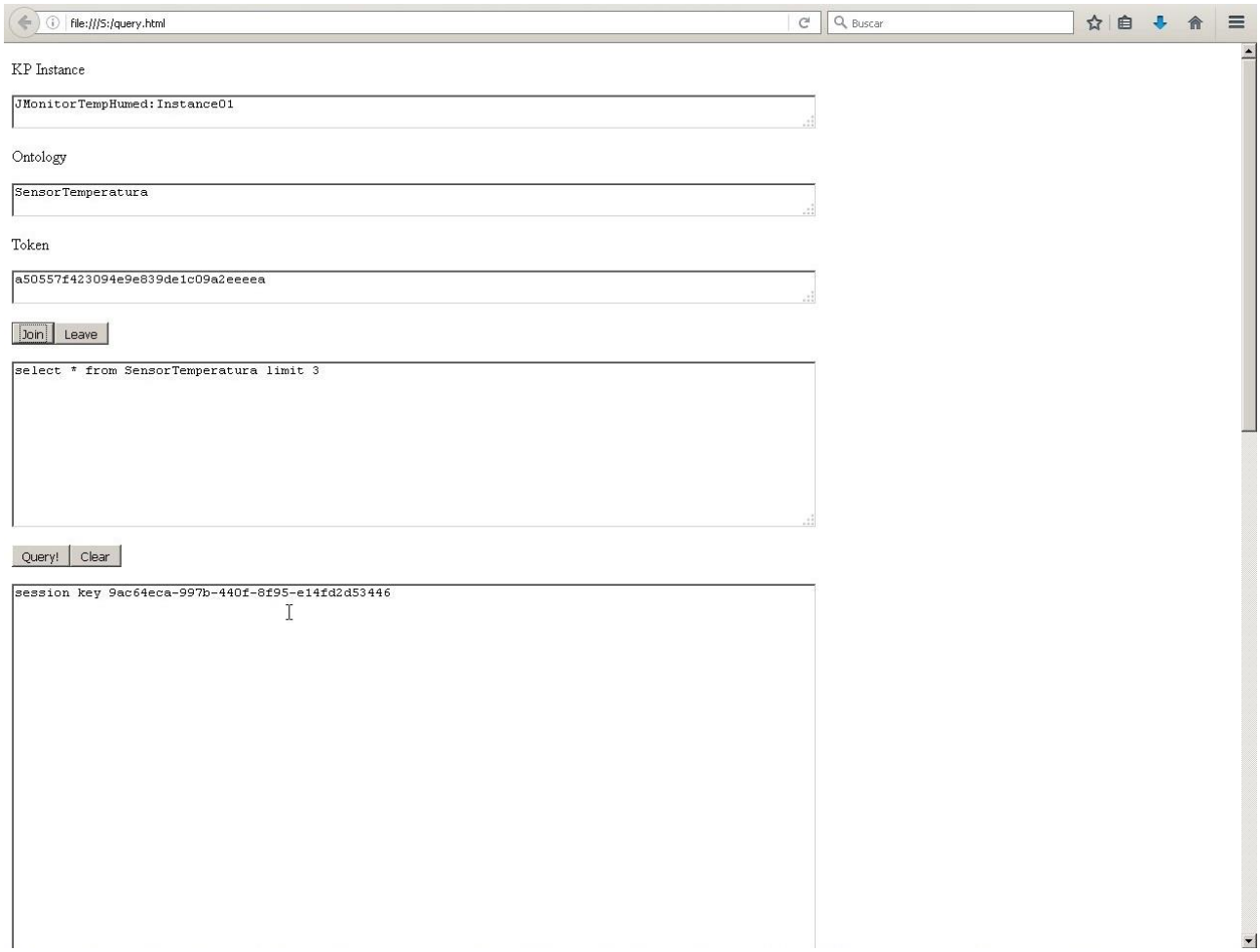

Figure 45: JavaScript APP Generates a Session Key

Then, the user can select the Query button to perform the query and get an answer:

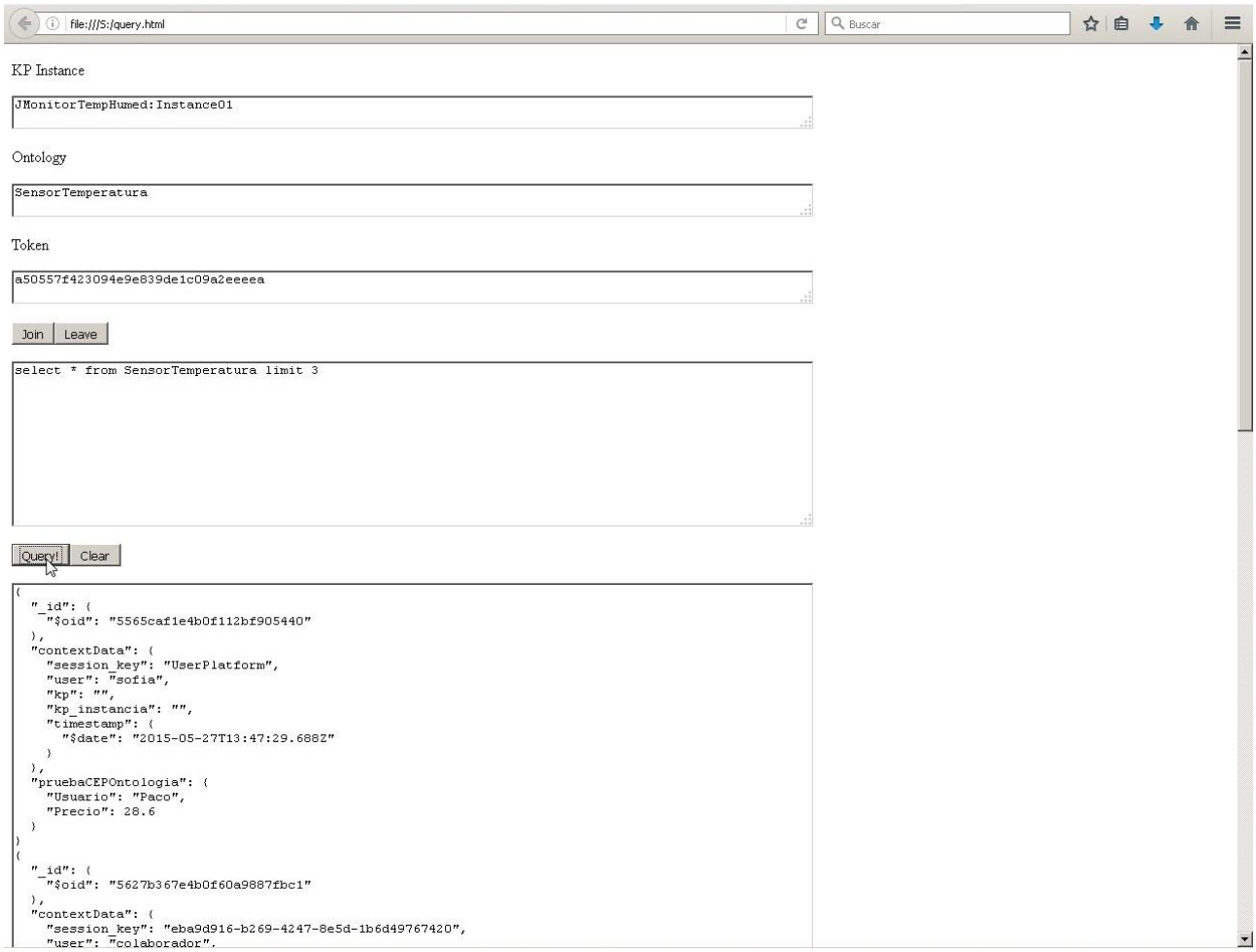

Figure 46: JavaScript APP Provides Answer to a Query

This KP is an HTML file and thus it can be modified using any Windows-based tool for that purpose such as Notepad or Google Chrome.

6.3.4 Developing APPs with SOFIA2 – Java example

We can find more advanced KPs in http://sofia2.comhttps://sofia2.com/desarrollador.html including a Java-based one.

It is available for download at https://sofia2.com/desarrollador.html. Similar to the previous one, it must be uncompressed and placed at S:\QUERY_JAVA, this time as a folder.

As it is a Maven project (pom.xml) including a Java Class (AppQuerySofia.java), the user must open a command prompt at **s:\>query_java** then run **s:\>Sofia2_VariablesEntorno.bat** ("Environment Variables") to set up the environment variables.

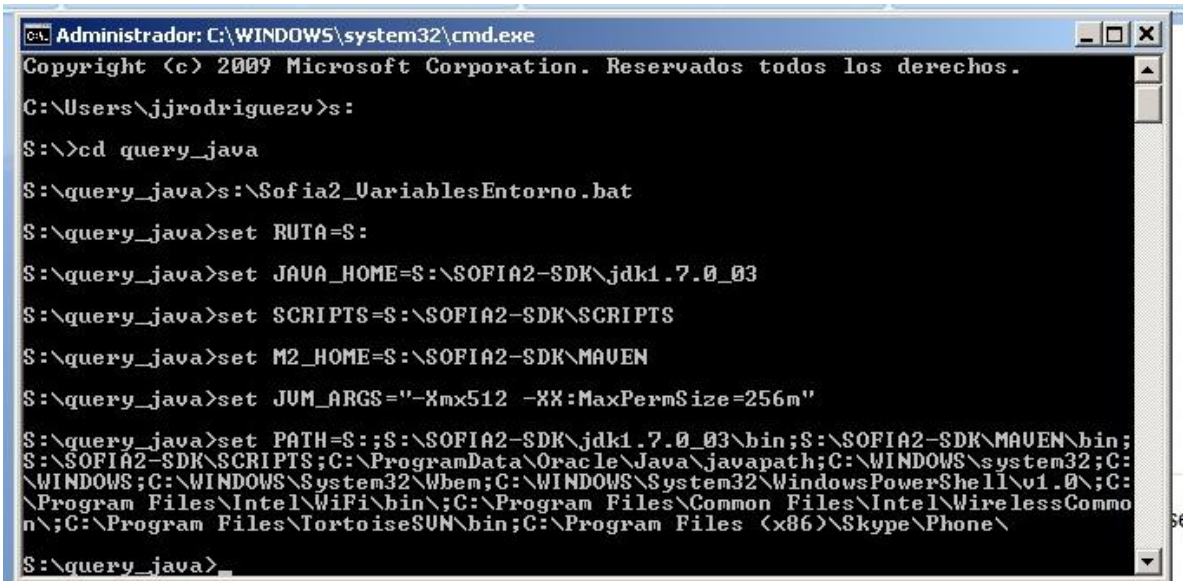

Figure 47: Setting Up Environment Variables

Then type **mvn install** to download all the necessary dependencies for the repository. This will take several minutes and requires the file settings.xml to be configured for Internet connection if there is a proxy.

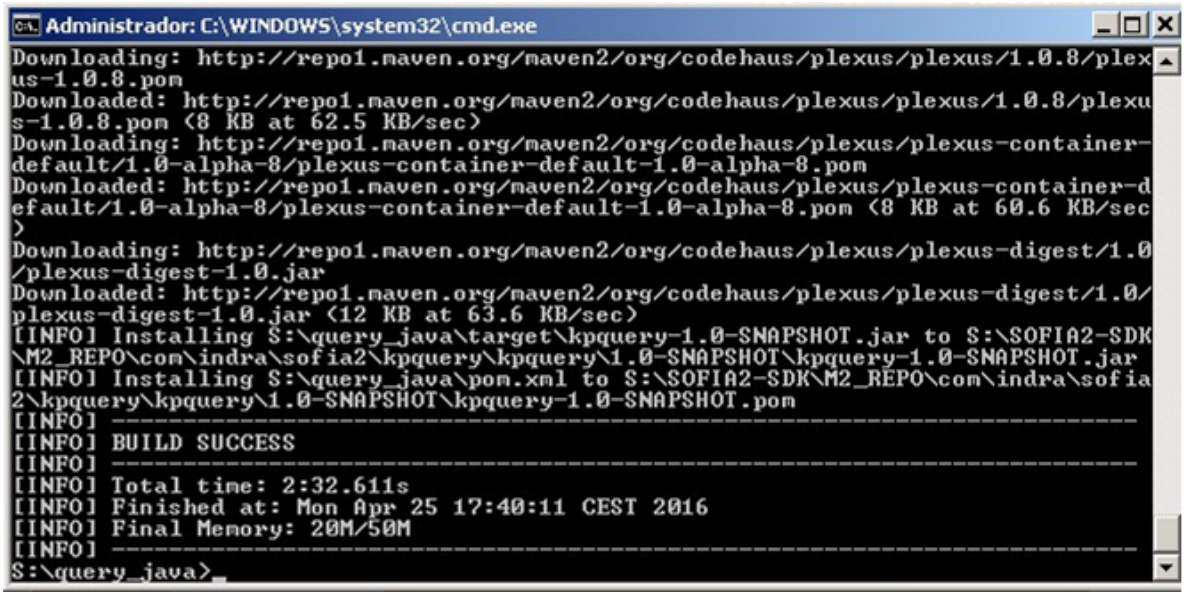

Figure 48: Maven Installation

After this, the Eclipse project can be generated and imported to the SOFIA2 IDE by running **mvn eclipse:eclipse**. This creates the Eclipse files . classpath and . project.

Running **S:\Sofia2_IDE.bat** the user can now load the project with **File>Import>General>Existing Projects into Workspace >"s:\query_java\".** Then the user can run the class AppQuerySofia2 from the Eclipse project and select **Run As > Java Application**. This should open the app in Eclipse's console (This console is only available in Spanish at the current point):

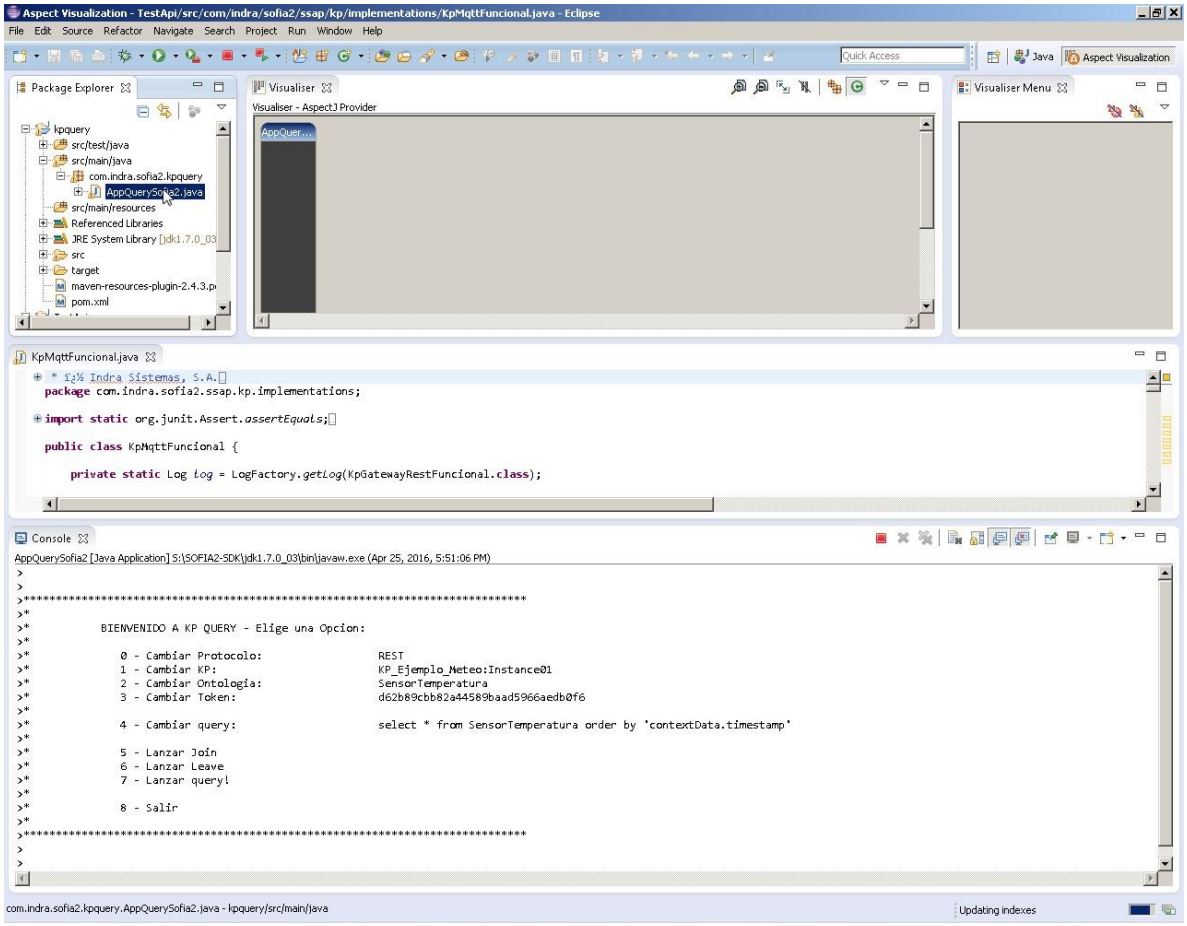

Figure 49: Eclipse-based Console

The user can change at will the KP, ontology, token and query to adapt to their uses:

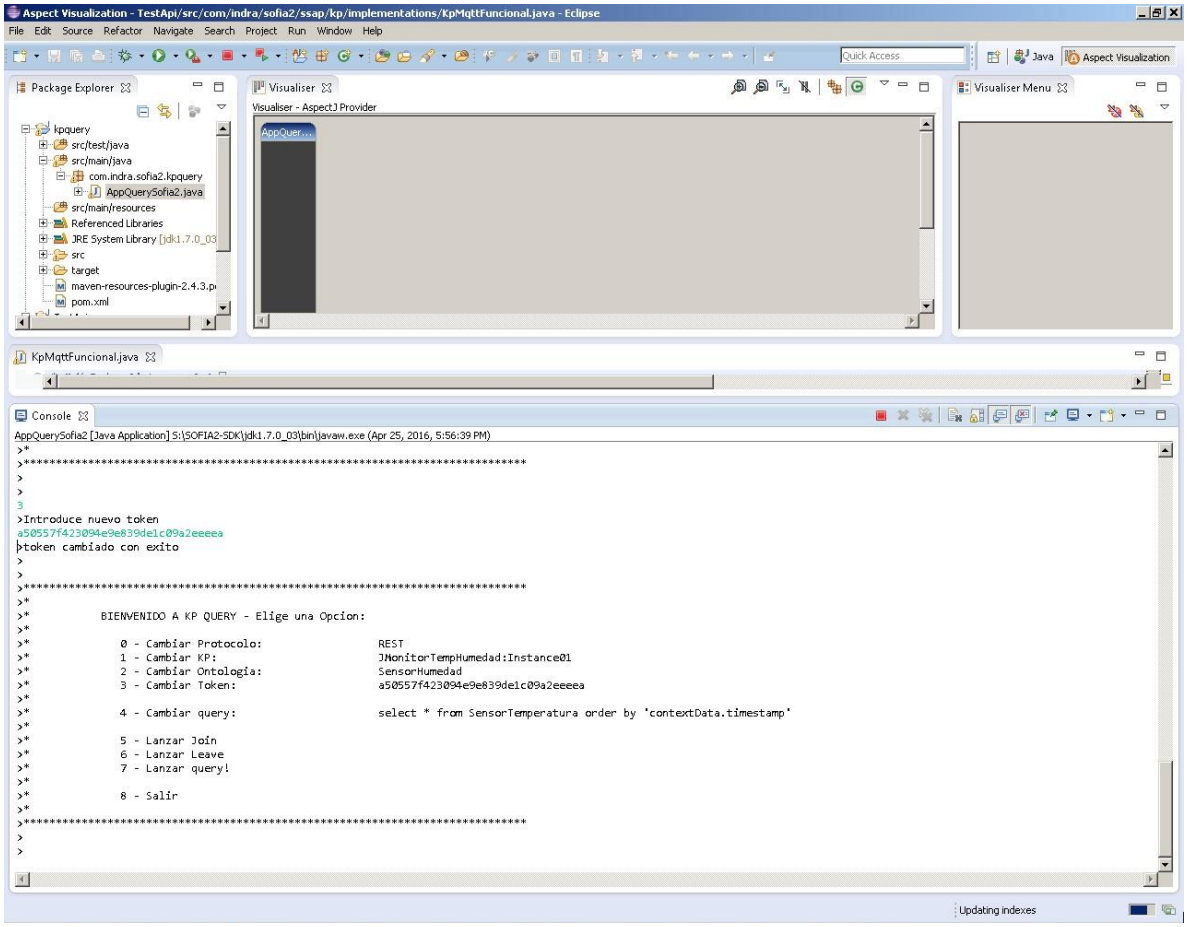

Figure 50: Using Eclipse Console to change Token

To start session, the user must launch Join (warnings may appear):

```
>Lanzando join...
log4j:WARN No appenders could be found for logger (org.apache.cxf.common.logging.LogUtils).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
>join correcto, session b7a19bdf-16f5-4393-a540-12880d3bc774
```

Figure 51: Join in Eclipse Console

then perform the query:


```
>Lanzando query...
>insercion de mensaje correcta
sun.net.www.protocol.http.HttpURLConnection$HttpInputStream@15434d8
[ {
  "_id" : {
    "$oid" : "535f45b7d947103a7029bc92"
  },
  "contextData" : {
    "session_key" : "be09247a-782f-4b5b-a0aa-bc55c35a5ea7",
    "user_id" : "1",
    "kp_id" : "60",
    "kp_identificador" : "KPvisualizacionHT01",
    "timestamp" : {
      "$date" : "2014-04-29T08:24:55.695Z"
    }
  },
  "SensorTemperatura" : {
    "geometry" : {
      "coordinates" : [ 40.512967, -3.67495 ],
      "type" : "Point"
    },
    "identificador" : "S_Temperatura_00001",
    "medida" : 17,
    "timestamp" : {
      "$date" : "2014-04-29T08:24:54.005Z"
    },
    "unidad" : "C"
  }
}, {
  "_id" : {
```


Figure 52: Query in Eclipse Console

7. ASSETS MANAGER

The Interoperability Framework plays a central role in IT2Rail as it enables different actors from several companies to exchange data. The Asset Manager is the component devoted to govern the process of maintaining a shared repository of ontologies, data schemas, configuration data and documentation.

The Asset Manager component of the Interoperability Framework is an organised collection and storage of these assets, enhanced by tools to support a workflow process for review, versioning, approval and publishing of the assets.

The demonstration IT2Rail application of the Asset Manager is built on top of the open-source WSO2 Governance Registry[1] with three primary functions:

[1] http://wso2.com/products/governance-registry

1. **Publisher**: a web application through which owners/contributors make informational assets available to the community;
2. **Management Console**: a web application including a workflow process tool that supports the collaborative management of the published assets, i.e. reviews, discussions, versioning, approval, distribution, etc.;
3. **Store**: an organised web repository of digital assets accessible by any participant organisation, human actor or application.

The Asset Manager will be the starting point for the implementation of the Semantic Discovery Engine which will be implemented in the F-REL version of the IT2Rail prototype.

The Interoperability Framework Ontology Repository and Semantic Web Service Registry are particular sections of the store, in which ontology files and semantically annotated web service descriptors are stored for use after having completed the approval/versioning process supported by the manager function, which operates on digital inputs provided by their owners / contributors through the publisher function.

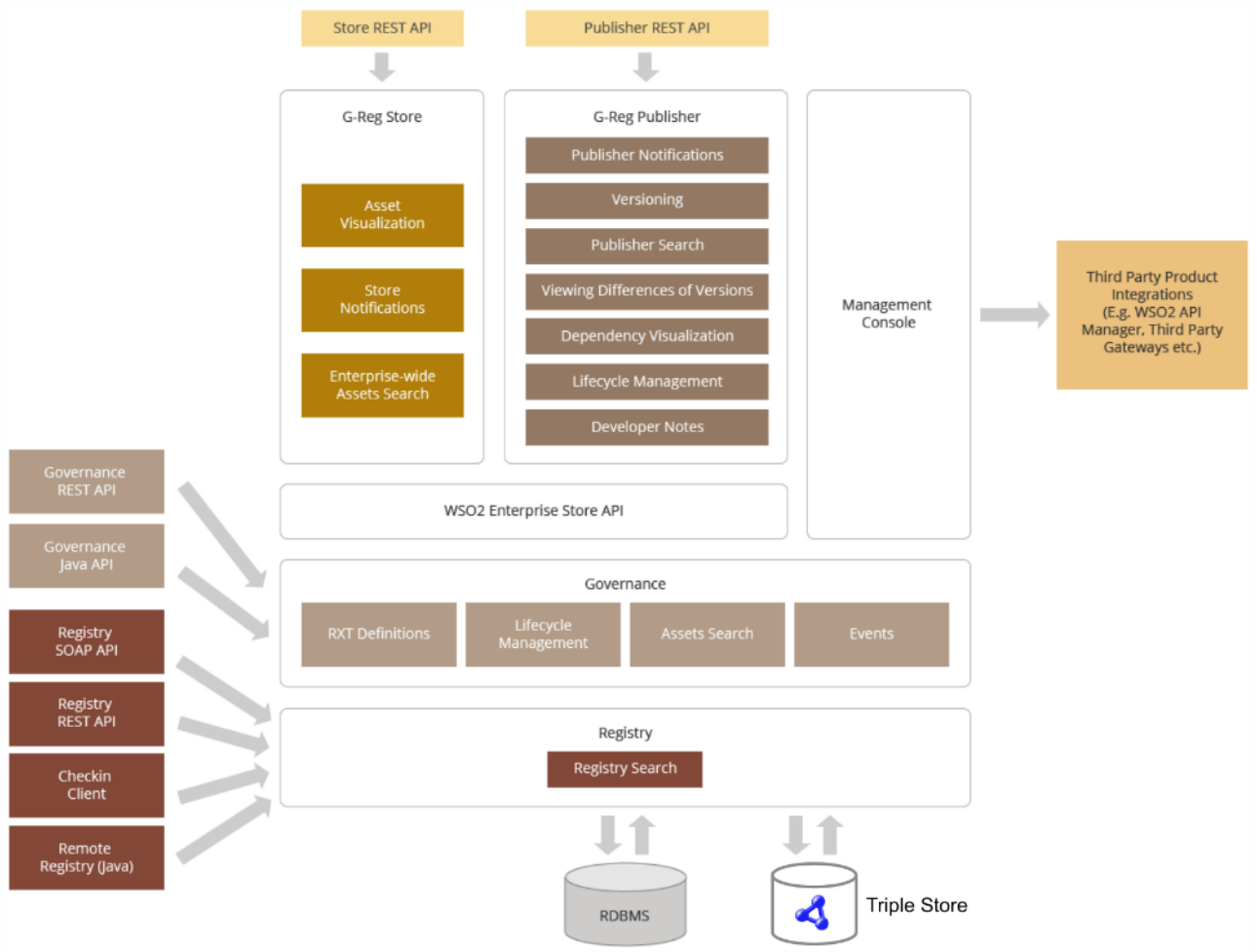

Figure 53: Assets Manager Architecture

7.1 ASSETS TYPES DEFINITION

The Assets Manager allows defining custom asset types by using Configurable Governance Artifacts (RXT) definitions, as documented in[2]. RXT definitions consist of a set of tables containing typed fields. In case of multiple choice fields, the values can be obtained by dynamically calling external Java code, therefore allowing populating fields values using data obtained by running an SQL or SPARQL query. The example reported in Figure 54 contains an example definition of an Ontology asset type, showing the definition of two tables (*Overview* and *Content*) and their fields.

```
<?xml version="1.0" encoding="UTF-8"?>
<artifactType type="application/vnd.wso2-ontology+xml" shortName="ontology" singularLabel="Ontology" pluralLabel="Ontologies"
    hasNamespace="false" iconSet="27">
    <storagePath>/ontology/@{name}</storagePath>
    <nameAttribute>overview_name</nameAttribute>
    <ui>
        <list>
            <column name="Name">
                <data type="text" value="overview_name"/>
            </column>
            <column name="Version">
                <data type="path" value="overview_version" href="@{storagePath}"/>
            </column>
        </list>
    </ui>
    <content>
        <table name="Overview">
            <field type="text" required="true" >
                <name label="Name">Name</name>
            </field>
            <field type="text">
                <name label="Version">Version</name>
            </field>
            <field type="text">
                <name label="Author">Author</name>
            </field>
            <field type="text">
                <name label="Institution">Institution</name>
            </field>
            <field type="options">
                <name label="Domain">Domain</name>
                <values>
                    <value>Infrastructure</value>
                    <value>Customer services</value>
                </values>
            </field>
            <field type="text">
                <name>Createdtime</name>
            </field>
            <field type="text-area">
                <name label="Description">Description</name>
            </field>
            <field type="date">
                <name label="Expected validity">Expected validity</name>
            </field>
        </table>
        <table name="Content">
            <field type="text">
                <name label="Ontology URL">URL</name>
            </field>
            <field type="text">
                <name label="Upload file">File</name>
            </field>
            <field type="options">
                <name label="Follow imported ontologies">Follow imports</name>
                <values>
                    <value>Yes</value>
                    <value>No</value>
                </values>
            </field>
        </table>
```

Figure 54: Sample Definition of an Asset Type

7.2 ASSETS LIFECYCLE DEFINITION

In IT2Rail the editing process of an asset is split among several actors and companies. Therefore, before deciding to publish an artefact an approval process must take place, featuring a series of "lifecycle stages". For instance, an Ontology may start off as "created", then after quality assurance has confirmed that the Ontology is consistent should be moved to "tested" stage. Upon testing, the

[2] https://docs.wso2.com/pages/viewpage.action?pageId=48284982

Ontology can then move to a "deployed" stage at which point it is released to production. Eventually, the Ontology will be taken down or replaced with another as it moves to a "deprecated" state.

The Assets manager allows defining custom lifecycles, which can be bound to each asset. Such lifecycles are defined as finished state machines expressed using SCXML[3]. Each lifecycle contains a set of states and the allowed transitions, plus additional constraints to be enforced while changing state and actions to be triggered upon successful state transition. The “actions” are implemented with Java code, therefore allowing the execution of arbitrary code. An example lifecycle definition can be seen in Figure 42.

```
<aspect name="IT2RailLifeCycle" class="org.wso2.jaggery.scxml.aspects.JaggeryTravellingPermissionLifeCycle">
    <configuration type="literal">
        <lifecycle>
            <scxml xmlns="http://www.w3.org/2005/07/scxml"
                version="1.0"
                initialstate="Initial">
                <state id='Initial'>
                    <datamodel>
                        <data name="transitionExecution">
                            <execution forEvent="Promote" class="org.wso2.jaggery.scxml.generic.GenericExecutor">
                                <parameter name="PERMISSION:get"
                                    value="http://www.wso2.org/projects/registry/actions/get"/>
                                <parameter name="PERMISSION:add"
                                    value="http://www.wso2.org/projects/registry/actions/add"/>
                                <parameter name="PERMISSION:delete"
                                    value="http://www.wso2.org/projects/registry/actions/delete"/>
                                <parameter name="PERMISSION:authorize" value="authorize"/>

                                <parameter name="STATE_RULE1:Created"
                                    value="Internal/private_{asset_author}:+get,+add,+delete,+authorize"/>
                                <parameter name="STATE_RULE2:Created"
                                    value="Internal/reviewer:-get,-add,-delete,-authorize"/>
                                <parameter name="STATE_RULE3:Created"
                                    value="Internal/everyone:-get,-add,-delete,-authorize"/>
                                </execution>
                            </data>
                        </datamodel>
                        <transition event="Promote" target="Created"/>
                    </state>
                    <state id="Created">
                        <datamodel>
                            <data name="transitionExecution">
                                <execution forEvent="Promote" class="org.wso2.jaggery.scxml.generic.GenericExecutor">
                                    <parameter name="PERMISSION:get"
                                        value="http://www.wso2.org/projects/registry/actions/get"/>
                                    <parameter name="PERMISSION:add"
                                        value="http://www.wso2.org/projects/registry/actions/add"/>
                                    <parameter name="PERMISSION:delete"
                                        value="http://www.wso2.org/projects/registry/actions/delete"/>
                                    <parameter name="PERMISSION:authorize" value="authorize"/>


                                    <parameter name="STATE_RULE1:In-Review"
                                        value="Internal/private_{asset_author}:+get,-add,-delete,-authorize"/>
                                    <parameter name="STATE_RULE2:In-Review"
                                        value="Internal/reviewer:+get,+add,-delete,+authorize"/>
                                    <parameter name="STATE_RULE3:In-Review"
                                        value="Internal/everyone:-get,-add,-delete,-authorize"/>

                                    </execution>
                                </data>
                            </datamodel>
                            <transition event="Promote" target="In-Review"/>
                        </state>
```

Figure 55: Sample Lifecycle Definition

[3] https://www.w3.org/TR/scxml/

7.3 ASSETS PUBLISHER

Once the asset type is defined, the Publisher web application allows authorised users to add new assets or to modify existing ones (see Figure 54). Since the Asset manager is not bound to any specific asset type, the form-based interface for editing information about an asset is auto-generated from the asset type definition. An example form allowing editing an Ontology asset is shown in Figure 56Figure 54.

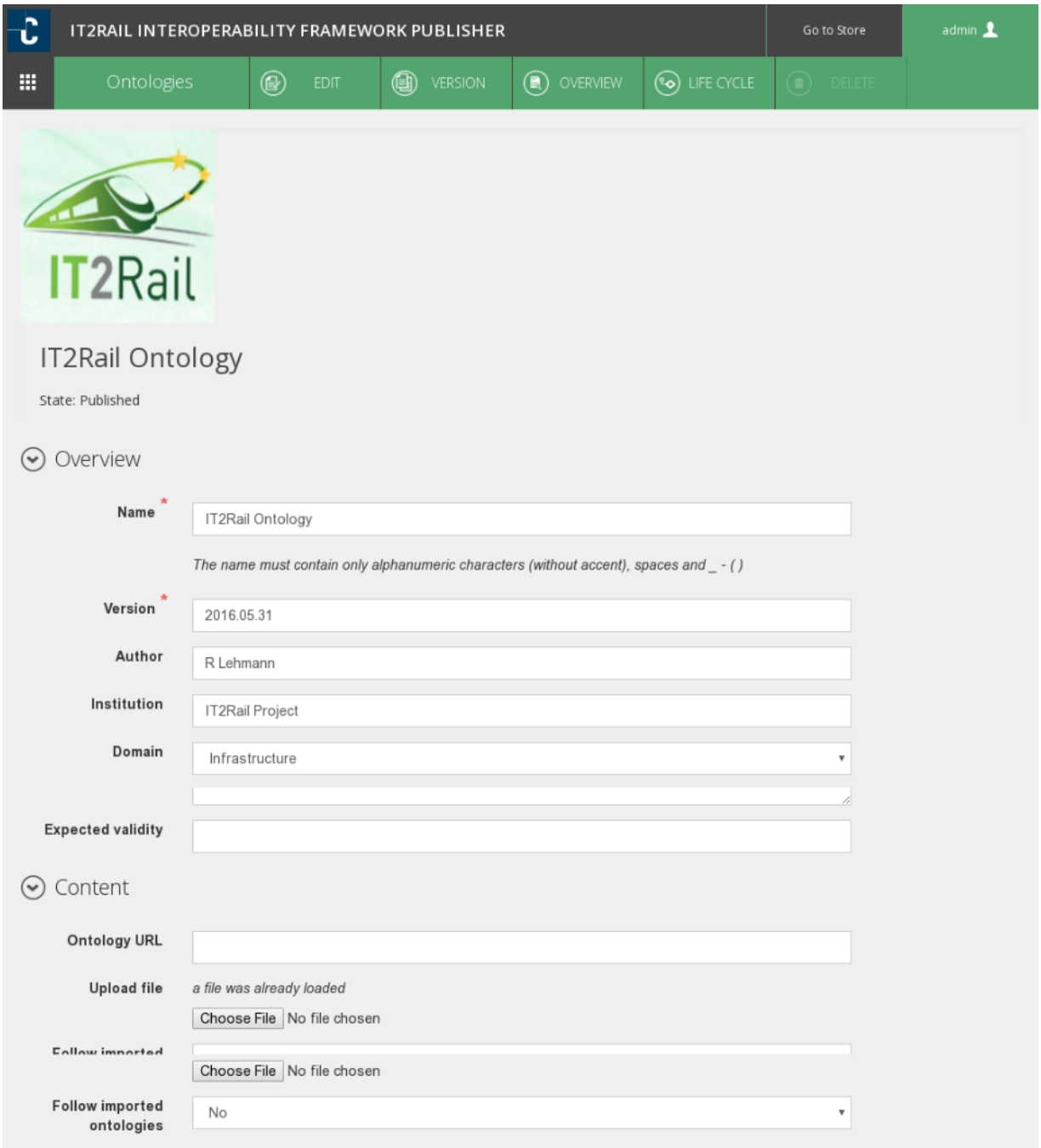

Figure 56: Example form Generated from the Asset Type Definition

Once the lifecycle workflow has been designed and deployed inside the Assets manager, the Publisher application allows the users to modify the status of an asset, as shown in **Error! Reference source not found.**. Data coming from each asset is stored inside the database used by WSO2 as XML documents (like the one reported in Figure 45), which are then converted into RDF and sent to the Triple Store upon successful publishing.

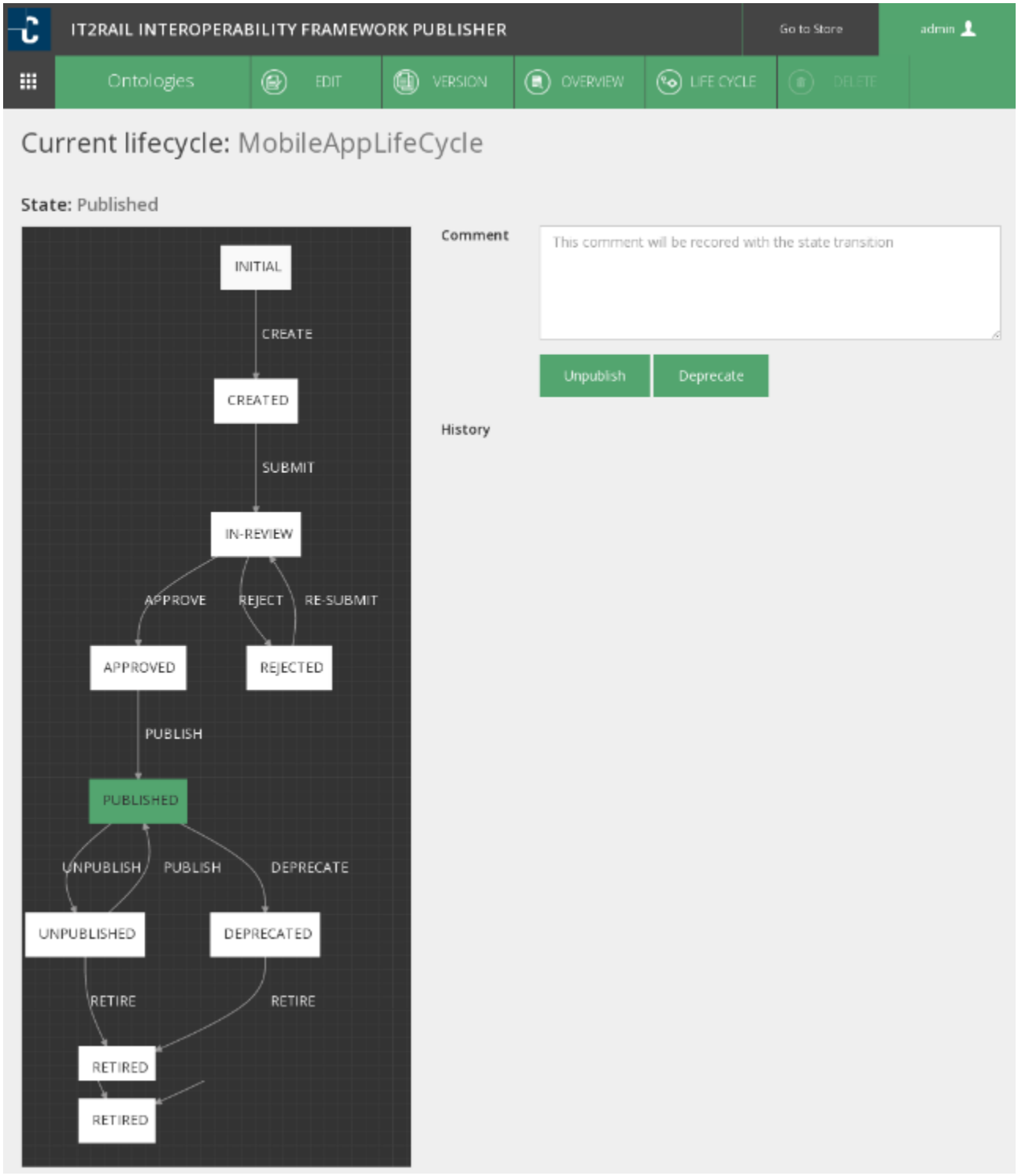

Figure 57: Example of the Asset Lifecycle Editing Interface Inside the Asset Publisher

```
<metadata xmlns="http://www.wso2.org/governance/metadata">
  <overview>
    <institution>It2Rail Project</institution>
    <provider>admin</provider>
    <author>Robert Lehman</author>
    <domain>Infrastructure</domain>
    <name>IT2Rail Ontology for all workpackages</name>
    <description>
      This ontology is as close as possible to the corresponding Capella modells. Although, minor
      changes where already employed: - All concepts and properties are in their singular form. -
      Properties (associations and attributes in UML) had either and “has” or ”is” prepended to their
      name as long as it was not present already - Properties with identical names and same namespace
      have names constructed as follows: “has<Class><Roll>” e.g. “hasItineraryDestination” and
      “hasJournesDestination” - Spacing has been removed, all concepts and properties shall be in camel-
      case Additional knowledge to with respect to a more ontological approch is added, but Capella
      stucture shall be kept as long as it does not break things. Attention! 2016-08-03 To give more
      meaning to reasoning DOMAIN and RANGE are removed from data-/object-properties! In return and to
      keep properties readable two new annotations (i2rumlRange and i2rumlDomain ) are introduced, which
      will reflect exactly domain and range but will have no impact on reasoning.
    </description>
    <version>V2016.08.03</version>
    <createdtime>00000001473917805920</createdtime>
  </overview>
  <images>
    <thumbnail>images_thumbnail</thumbnail>
  </images>
  <content>
    <file>content_file</file>
    <followImports>No</followImports>
    <url>
      http://192.168.150.139:8890/DAV/vowl/data/it2r_onto_2016_08_03.owl
    </url>
  </content>
</metadata>
```

Figure 58: Sample XML Content of an Asset

7.4 ASSETS STORE

The Assets store is the users’ frontend for accessing and searching existing assets according to the defined authorisation policy. The example depicted in Figure 59 shows an “Ontology” asset with its information. The “User reviews” tab allows inserting ratings and comments about assets, therefore enabling the possibility of collecting hints and suggestions to improve the content of the assets.

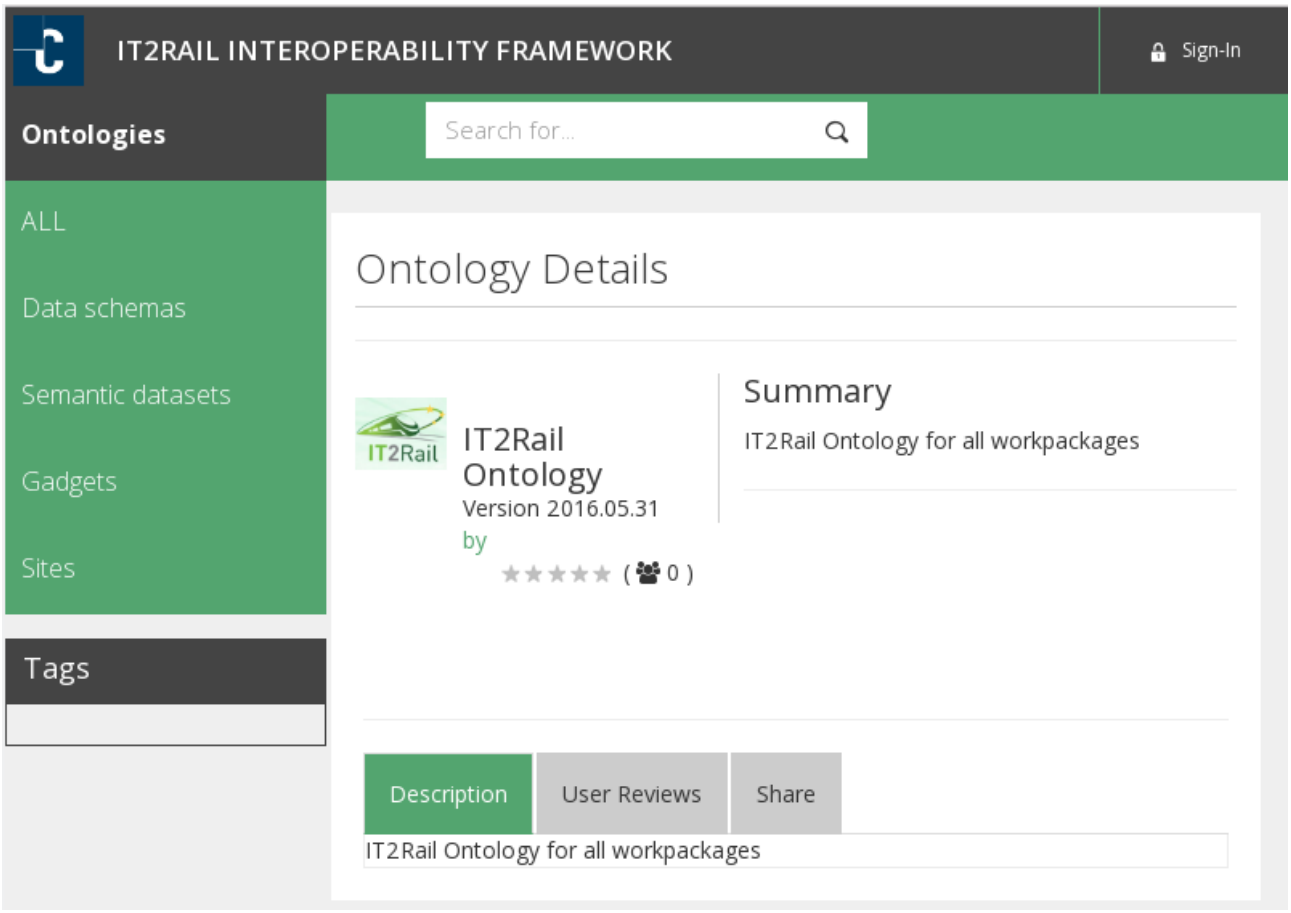

Figure 59: Details of an Asset inside the Store Application

8. PROTÉGÉ ONTOLOGY EDITOR

The IT2Rail ontology is an explicit axiomatic formalisation of the IT2Rail problem domain, expressed in the machine-readable fragment of first-order predicate logic language OWL.

The Ontology OWL file(s) is/are created / maintained using the ontology editor “Web Protégé”, a web based version of the open source Protégé Ontology editor[4].

The editor is equipped with a series of plugins[5] which add features and extend the functionality to support the Ontology engineering process, e.g. validation for consistency of the axiomatic system.

The Ontology file generated and validated with the editor is published to the Asset Manager, where it undergoes the versioning/approval process and is eventually stored in the Asset Manager’s dedicated store, the Ontology Repository, for distribution/access by the community of Travel Operators.

[4] http://protege.stanford.edu/, http://webprotege.stanford.edu/
[5] http://protegewiki.stanford.edu/wiki/Protege_Plugin_Library

9. TRIPLE STORE

A Triple Store, also known as Graph database or store, is a software component designed to store linked data, i.e. machine-readable first-order predicate logic statements about facts of the problem domain expressed as resource description framework[6] "triples" in the form subject-predicate-object, where subject is an instance of a concept of the Ontology, predicate is a binary relationship between instances of the ontology, also described in the ontology, and object is either an instance of the ontology related to the subject by the predicate, or a data value[7].

It allows semantic querying of the stored resources and the generation of additional triples through the process of *inference*, i.e. automated theorem-proving based on the axioms described by the ontology. For example, given the axiom "a Train stops at a Station" and the statements ("triples") "9406 is a Train" and "9406 stops at ABCD", the triple "ABCD is a Station" is an inferred result or "extra" triple generated through inferencing.

Triple stores that allow identifiers of instance of concepts to be differentiable Uniform Resource Locators (URL[8]) institute machine-processable links between triples across the World Wide Web, so that triples stored in any accessible triple store become interlinked and constitute a semantic graph spanning the web. This concept is used in the IT2Rail project to create the "Web of transportation data".

9.1 GRAPHDB

For the IT2Rail project demonstration a free version of the Ontotext GraphDB graph store[9] is used as the main triple store implementing persistence for the Ontology Repository and the Semantic Web Service Registry.

It is also used to store a semantic graph of IT2Rail resources linked to additional triples over the web allowing IT2Rail applications to perform semantic queries over the "Web of transportation data" through the services provided by the Interoperability Framework and the GraphDB-supported SPARQL Graph protocol.[10]

9.2 VIRTUOSO UNIVERSAL SERVER

Virtuoso Universal Server[11] is a rich suite of data and content management applications including a triple store and SPARQL endpoint. In the context of the IT2Rail demonstration scenario is used for multiple purposes:

1. To demonstrate the ability of the Interoperability Framework to operate against any triple store implementation by means of the open standard SPARQL graph protocol. The Interoperability Framework federates triple stores of any implementation providing a Web of transportation data abstraction to applications;

[6] https://www.w3.org/RDF/
[7] A data value, such as the string "abc", is also an instance of a concept, namely of the concept "String". The latter is however a "primitive" concept with a well-defined and understood semantics, not considered part of the problem domain's ontology.
[8] https://www.w3.org/TR/url-1/
[9] http://ontotext.com/products/graphdb/
[10] https://www.w3.org/TR/sparql11-http-rdf-update/
[11] http://virtuoso.openlinksw.com/

2. To leverage the “Sponger” technology available in the Virtuoso triple store to automate the process of linking / importing triples across the Web from multiple sources, including social networks;
3. As temporary Web Server, to be replaced by the full implementation of the Assets Manager store, serving HTML pages, i.e. documentation, in the course of the project’s development.

10. RDF PROGRAMMING FRAMEWORK

In order to reduce to a minimum, the effort and amount of code to be developed for demonstration purposes, particularly of purely utilitarian code with no particular innovation significance, the Interoperability Framework will make use of existing open source frameworks to deal with connecting and operating through the SPARQL graph protocol with triple stores over networks, with standard web services stacks and with standard serialisation of java “plain old java objects” (POJOs).

To this end, the Interoperability Framework demonstration will be based on the following open source stack:

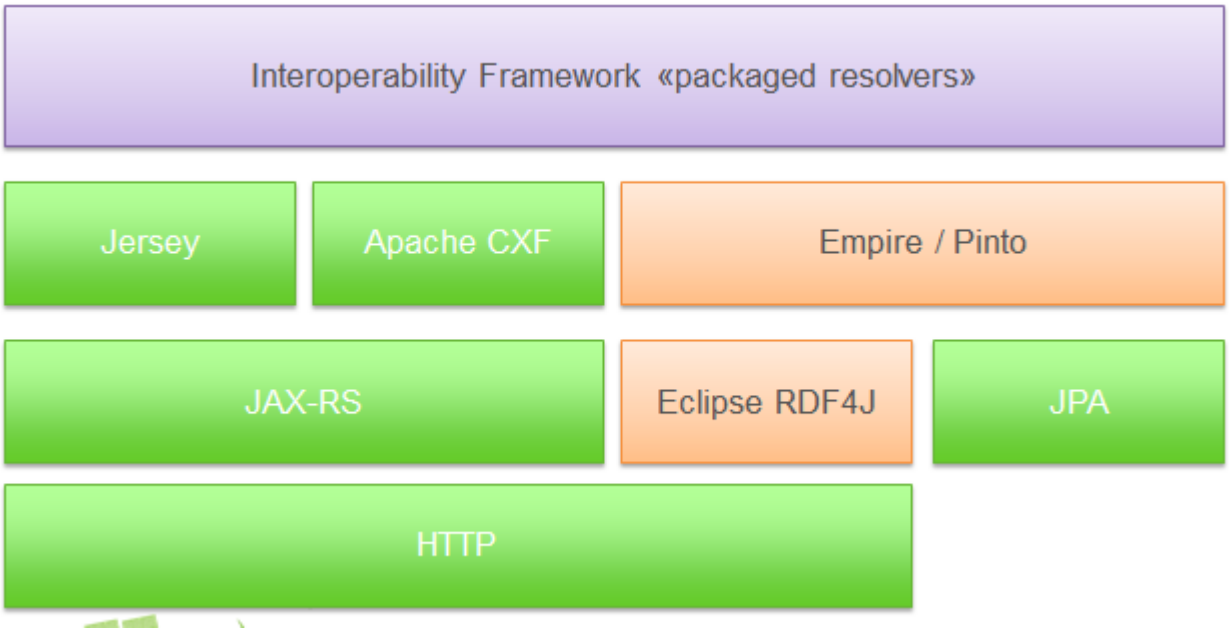

Figure 60: Open Source Stack IF demonstration

HTTP provides utilities to operate across networks using the http protocol, JAX-RS provides a framework for web services over http, and the Jersey ad Apache CXF frameworks provide functionality for Restful and SOAP web service development, respectively, including JSON-POJO and XML-POJO bindings / serialisation. The Java Persistence Architecture (JPA) provides utilities for persistence on a variety of data storage and retrieval mechanisms.

These elements are used in their turn in conjunction with the Eclipse RDF4J framework, used for manipulation of RDF statements and triple stores repositories, which is itself the foundation of the

Empire / Pinto frameworks[12]: the former provides JPA-like functionality for RDF statements while the latter provides mappings for RDF statements into POJOs and vice versa.

Interoperability Framework demonstration services, i.e. "**packaged resolvers**", are developed using the underlying technology stack to implement the IT2Rail specifications which represent the outcome of the research and innovation activity, particularly the operation on semantic, i.e. first-order predicate logic statements to automate interoperability tasks across heterogeneous and distributed systems.

The Empire and Pinto frameworks will be extended with the extra features and functionality required to support the Interoperability Framework objectives, i.e. automatic complex semantic matchmaking across distributed resources.

The fundamental goal of the research for the Interoperability Framework within the IT2Rail project scope is in fact to *direct* the software to create and apply automatically appropriate mappings across message structures through *semantic annotations* using terms from the ontology *without changing or writing extra 'mapping' or 'integration' code*.

This ability is exploited, for example, in the Semantic Integration Broker (SIB) deployed in the SOFIA2 run time environment platform.

10.1 DEPENDENCIES

Additional compile or runtime dependencies for the development of the Interoperability Framework services are specified in the framework's maven pom file:

```
<projectxmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>org.IT2Rail.rdf.framework</groupId>
<artifactId>IT2Rail -rdf-framework</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>WP1 rdf framework for IT2Rail </name>
<description>A POJO/RDF semantic mapping and persistence framework IT2Rail
project</description>
<organisation>
        <name>IT2Rail project Work Package 1 Interoperability Framework</name>
</organisation>
<properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
<build>
        <sourceDirectory>src/main/java</sourceDirectory>
        <resources>
        <resource>
                <directory>src/main/resources</directory>
```

[12] https://github.com/mhgrove/Empire, https://github.com/stardog-union/pinto

```
            <excludes>
                <exclude>**/*.java</exclude>
            </excludes>
        </resource>
        </resources>
            <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.5.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.6</version>
                <configuration>
                    <warSourceDirectory>WebContent</warSourceDirectory>
                    <failOnMissingWebXml>false</failOnMissingWebXml>
                </configuration>
            </plugin>
            </plugins>
        </build>
        <dependencies>
            <dependency>
                <groupId>org.eclipse.rdf4j</groupId>
                <artifactId>rdf4j-runtime</artifactId>
                <version>2.0</version>
                <type>pom</type>
            </dependency>
        <!-- https://mvnrepository.com/artifact/com.google.inject.extensions/guice-multibindings -->
            <dependency>
                <groupId>org.ow2.spec.ee</groupId>
                <artifactId>ow2-jpa-1.0-spec</artifactId>
                <version>1.0.12</version>
            </dependency>
            <dependency>
                <groupId>com.google.inject.extensions</groupId>
                <artifactId>guice-multibindings</artifactId>
                <version>4.0</version>
            </dependency>
            <!-- https://mvnrepository.com/artifact/com.google.inject.extensions/guice-
assistedinject -->
            <dependency>
                <groupId>com.google.inject.extensions</groupId>
                <artifactId>guice-assistedinject</artifactId>
                <version>4.0</version>
            </dependency>
            <!-- https://mvnrepository.com/artifact/org.javassist/javassist -->
            <dependency>
                <groupId>org.javassist</groupId>
                <artifactId>javassist</artifactId>
```

```
            <version>3.17.1-GA</version>
        </dependency>
        <dependency>
            <groupId>org.IT2Rail.cp-rdf4j-utils</groupId>
            <artifactId>IT2Rail.cp-rdf4j-utils</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
        <groupId>com.complexible.common.utils</groupId>
            <artifactId>cp-common-utils</artifactId>
            <version>5.0.1</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.geonames/geonames -->
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>19.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/commons-beanutils/commons-beanutils -->
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
        <version>1.9.2</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/junit/junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.21</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-simple -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.21</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.geonames/geonames -->
        <dependency>
            <groupId>org.geonames</groupId>
            <artifactId>geonames</artifactId>
            <version>1.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Table 46: Runtime Dependencies for the Development of the Interoperability Framework

11. LINKED DATA UTILITIES

Many data resources, such as Railway Station descriptions, are not currently available as linked data or RDF triples, and/or must be obtained through legacy systems and formats such as excel, comma-separated-values files (csv) or other specialty (i.e. EDIFACT) formats. Also, some of these data may require certain forms of 'data cleaning' procedure to eliminate duplications, to resolve synonyms/homonyms, or even to deal with special characters in their descriptions. As part of the project, these resources will be obtained, 'cleaned' and converted into RDF triples and added to the semantic graph as part of the publishing / managing /storing of assets in the Asset Manager.

In order to operate the data 'cleaning' activities and as part of the versioning / approval process the LODRefine tool (Linked Open Data –enabled version of the OpenRefine utility[13]: will be used.

12. INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

Development, assembly and testing of Interoperability Framework components will be supported by an Integrated Development Environment for development on JAVA.

12.1 ECLIPSE

The IDE of choice for the Interoperability Framework is the open source Eclipse environment version 4.4 or above equipped with the following plug-ins/extensions:

- Web Tools Platform;
- Eclipse Data Tools Platform;
- Apache CXF runtime environment;
- Maven connector, maven plug-ins and runtime;
- Github connector;
- Apache Tomcat server runtime environment;
- Java 1.8_051 runtime environment;
- Eclipse Plug-in development environment.

A specific Eclipse plug-in may be developed within the project to extend the Eclipse Java editor in order to provide it with the ability to support the process of semantic annotation of POJOs with terms of the IT2Rail ontology stored in the Ontology repository, e.g. validation that classes and properties in the annotations exist and are consistent with the Ontology.

[13] https://sourceforge.net/projects/lodrefine/

13. CONCLUSION

The Interoperability Framework is the IT2Rail core component, which must ensure the interoperability between new travel services and existing legacy systems.
With it on mind this component is developed with the ability to exchange and use information from a large and heterogeneous network as required from the tendency of the market is going through.

In order to achieve it, the Interoperability Framework was developed taking into account the new edge technologies and the cooperation with heterogeneous companies that worked in synergy was the key for having a new model to unify a very fragmented market.

Interoperability Framework was in this way achieved, becoming a real broker using:

- Sofia 2 Middleware platform for working properly, with robustness, in real (near-real) time, and under real condition;
- Common Query Engine provides a software infrastructure to allow efficient and easy to program communication among multitude of data management systems;
- Software which allowed the correct conversion from the legacy system into the ontology defined in the IT2Rail project.

A lot of work is being done but was just the first step, the new projects that are starting as CONNECTIVE will be another challenge for increasing the quality of the present component and achieve the main scope that the SHIFT2Rail is looking for: Transport multimodality, seamless and sustainability rendering the entire European transportation system a natural extension of citizens work and leisure environments, across all modes, local and long distance, public and private.