# IoTCrawler Documentation

***Release 0.0.1***

**IoTCrawler**

**Jun 04, 2021**

# CONTENTS

**Note:** Please make sure you are looking at the documentation that matches the version of the software you are using. See the version label at the top of the navigation panel on the left. You can change it using selector at the bottom of that navigation panel.


IoTCrawler focuses on integration and interoperability across different platforms for discovery and integration of data and services from legacy and new systems and mechanisms for crawling, indexing, and searching in distributed IoT systems.

CHAPTER

ONE

# IOTCRAWLER

The IoTCrawler is a project, funded by EU under the H2020 programme, whose main objective is to become a scalable, flexible and secure search engine for IoT information. Its intention is not to become a new IoT platform competing with existing ones, but being a higher frame of reference for all of them, creating an IoT ecosystem, quite like any web.based search engine is for websites and webpages. It provides scalable and efficient methods for discovery, crawling, indexing and ranking of IoT resources in large-scale cross-platform, cross-disciplinary systems and scenarios. IoTCrawler improves on other approaches by providing and considering as main driving pillars the enablers for secure and privacy-aware access of IoT resources while also, providing monitoring and analysis of QoS and QoI that is evaluated during the ranking of suitable resources and supports fault recovery and service continuity.

CHAPTER

TWO

# KEY CONCEPTS

## 2.1 Overview

- brief overview of what is IoTCrawler

### 2.1.1 What is Crawling?

- describe in detail

### 2.1.2 What is Indexing?

- describe in detail

### 2.1.3 What is Ranking?

- describe in detail

### 2.1.4 Why is it useful?

- describe in detail

### 2.1.5 What is IoTCrawler

- how dit it started?
- what is it used for?
- what does it offer?

## 2.2 IoTCrawler Model

- list out key design features of IoTCrawler e.g.
  - *MetaData Repository* — 2 lines description
  - *Indexing* — 2 lines description
  - *Ranking* — 2 lines description
  - *Orchestrator* — 2 lines description
  - *Search Enabler* — 2 lines description
  - *Authorization Enabler* — 2 lines description
  - *Semantic Enrichment* — 2 lines description
  - *Monitoring* — 2 lines description

### 2.2.1 MetaData Repository

The core component of the architecture of IoTCrawler is what we call MetaData Repository, which can be seen as a Context Broker with the features of of distributing information following both query-response and publication-subscription patters among the other components of the IoTCrawler architecture. This Context Broker must be capable of representing semantic, linked data and property graphs. These features has been included in NGSI-LD, a new standard which has been conducted under the ETSI ISG CIM initiative. For this reason, we have selected this technology for the instantiation of our MetaData Repository.

Scorpio is an NGSI-LD compliant context broker developed by NEC Laboratories Europe and NEC Technologies India. This project is part of [FIWARE](https://www.fiware.org/). For more information check the FIWARE Catalogue entry for [Core Context](https://github.com/Fiware/catalogue/tree/master/core). We have selected this broker for the instantiation of our MetaData Repository because of their features related to the representation of distribution and federation scenarios.

### 2.2.2 Indexing

The Indexing component provides a means for clients to search for IoT entities efficiently. It focuses on IoT streams and sensors, where queries can be based on sensor type and absolute or relative location. To initiate the process of indexing, a platform manager needs to register a metadata broker (MDR) with the Indexing component. In turn this will trigger the subscription to sensors and streams at the registered MDR. As metadata descriptions are updated at the MDR, the Indexing component will be notified, and will then index the sensors and streams based on location. For scalability, the Indexing component can be configured so that the persistence it relies on (MongoDB) can be sharded. With respect to other components in the framework, the Ranking component relies on Indexing for generating the ranking of search results.

### 2.2.3 Ranking

The Ranking component facilitates ranking mechanism for IoT resources. Ranking and resource selection rely on the registry built (and constantly updated) by crawling and indexing methods. The purpose of Ranking is to aid users and applications to not only find a set of resources relevant to their needs, but also to select the best or most appropriate one(s) from that set. There are multiple criteria for ranking IoT resources such as data type, proximity, latency, availability. The Ranking component supports application-dependent,multi-criteria ranking.

### 2.2.4 Orchestrator

The orchestrator component is responsible for interactions with client IoT applications. It allows applications to subscribe to streams without having a public endpoint as well as tracks subscription requests. In case of stream failure, orchestrator is able to notify application and provide a list of alternative streams for subscription.

### 2.2.5 Search Enabler

The GraphQL-based search enabler component is considered as a main search-component of IoTCrawler. It employs a query language (GraphQL) and a query processor, which works on top of NGSI-LD-compliant component (the ranking Component or MDR). The component eliminates the lack of expressivity and functional capabilities which prevent NGSI-LD from being the main search interface the large-scale IoT metadata deployments gathered in the IoTCrawler platform. The search enabler fills the gap between low-level sensors and high-level domain semantics about sensors data and deals with the context-dependent entities by maintaining the context in the IoTCrawler platform.

### 2.2.6 Authorization Enabler

The purpose of this enabler is twofold: on the one hand for a secure communication between the IoTCrawler components, and on the other, for controlling the access of the registered users to the resources stored in the IoTCrawler platform. It instantiates the Distributed Capability-Based Access Control (DCapBAC) technology by using both an XACML framework, and a Capability Manager. This latter is responsible for issuing authorizations tokens which must be present in each of the requests aimed at the MetaData Repository.

### 2.2.7 Semantic Enrichment

The Semantic Enrichment (SE) component is responsible for annotating data streams with Quality of Information (QoI). To calculate the QoI the SE subscribes to the MDR for changes in IoTStreams. When receiving notifications for a stream it takes the related metadata of the stream and generates the QoI annotation, which is stored in the MDR afterwards to be accessible by other components of the framework.

### 2.2.8 Monitoring

The Monitoring component is responsible for observing data streams to detect anomalies...

## 2.3 MetaData Repository

### 2.3.1 What is a MetaData Repository

The core component of the architecture of IoTCrawler is what we call MetaData Repository, which can be seen as a Context Broker with the features of distributing information following both query-response and publication-subscription patters among the other components of the IoTCrawler architecture. This Context Broker must be capable of representing semantic, linked data and property graphs. These features have been included in NGSI-LD, a new standard which has been conducted under the ETSI ISG CIM initiative. For this reason, we have selected this technology for the instantiation of our MetaData Repository.

### 2.3.2 Choosing a MetaData Repository

There are available differents implementations of NGSI-LD context brokers like Scorpio, Orion-LD and Djane. In IoTCrawler architecture, Scorpio is the choosen context broker.

Scorpio is an NGSI-LD compliant context broker developed by NEC Laboratories Europe and NEC Technologies India. This project is part of FIWARE. For more information check the FIWARE Catalogue entry for Core Context.

We have selected this context broker for the instantiation of our MetaData Repository because of their features related to the representation of distribution and federation scenarios. Another factor to choosing this broker is the semantic treatment of the information which is more advanced than the rest.

### 2.3.3 API - Scorpio

Resources can be managed through the API (e.g. entities and subscriptions). The most relevant request of MetaData Repository are defined in the next REST API.

For a detail explaination about the API, please look the ETSI spec.

### 2.3.4 How to deploy/test - Scorpio

This component can be deployed following the README.md file.

Once MetaData Repository is running you can test it. You can find postman collection with sample requests in https://github.com/IoTCrawler/iotcrawler-samples/tree/master/authorization-enabler, you only need to define:

- `MDR-IP`:`MDR-Port` : Endpoint of MDR (Scorpio).

## 2.4 Orchestrator

The orchestrator component is targeted to be a key endpoint for IoT application to interact with IoTCrawler platform.

It's main goal of the orchestrator is to provide a subscription mechanism, which allows IoT applications to be deployed in private networks without exposing any REST-endpoints for getting notifications from MDR.

This flow is possible due to AMQP subscriptions implemented using the RabbitMQ server, running together with the orchestrator.

The Orchestrator supports NGSI-LD REST interface: all requests are redirected to MDR with some extra logic in the middle (such as tracking/caching).

This subscription mechanism allows IoT app developer to debug IoT applications using the local instance of NGSI-LD Broker (Scorpio or Jane) and then later just to switch to orchestrator's endpoint.

### 2.4.1 IoTCrawler Client

The orchestrator project contains the core library (Java), which is targeted to be used in IoTCrawler applications. The idea is that standalone IoTCrawler applications would use this library and act as client to an online IoTCrawler deployment.

Due to AMQP subscriptions mechanism the IoTCrawlerClient subscribes to the online AMQP Server and gets notifications, which placed there by the online orchestrator.

The online orchestrator is expected to receive notifications from MDR broker and put them into AMQP server. The IoTCrawlerClient subscribes to a queue associated to an exchange named as a subscription id.

This mechanism allows IoTCrawler apps to run in private networks connected to the Internet (e.g. smart home deployments, mobile apps, etc).

### 2.4.2 Code structure

The orchestrator project consists of a number of modules depending hierarchically (bottom modules depend on top ones):

1. Fiware-models. A library which contains classes implementing the NGSI-LD model used in NGSI-LD client library.
2. Fiware-clients. A library which contains the NGSI-LD client and a number of tests for checking its compatibility with a broker.
3. Core. A library which contains the IoTCrawlerClient interface, models of the core entities (IoTStream, Sensor, Platform, etc.), two different implementations of IoTCrawlerClient. The core is targeted to be a part of client IoTCrawler application and to interact with the Orchestrator in the online IoTCrawler platform.
4. Orchestrator - A standalone standalone component running in the IoTCrawler platform.

### 2.4.3 Build

The orchestrator and it's supplementary libraries be built using the following command:

*sh make.sh install*

During the build process all the modules are will be installed installed into a local maven repository.

### 2.4.4 Deployment

The orchestrator is deployable as a docker container together with a number of assisting services (such as Redis and RabbitMQ).

A docker image for deployment can be build by the following command:

*sh make.sh build-image*

---

**Note:** A Google's Jib maven plugin is used for building image, so there is no traditional Dockerfile. *docker-compose build orchestrator* would not work.

---

A docker-compose file is targeted for help with local running and debugging process.

A helm chart configures the orchestrator and its components to be deployed on Kubernetes cluster.

A gitlab CI configuration configures the building process deployment flow.

### 2.4.5 Configuration

The orchestrator component is configured via a number of environment variables, which can be found in a docker-compose file:

1. **IOTCRAWLER_RABBIT_HOST** - host address of AMQP server
2. **IOTCRAWLER_REDIS_HOST** - host address of Redis server (used for mainain info about already running subscriptions).
3. **NGSILD_BROKER_URL** - URL to NGSI-LD endpoint (MDR).
4. **HTTP_SERVER_PORT** - port, on which orchestrator should listen incoming REST queries (NGSI-LD queries from apps and notifications from a broker).
5. **HTTP_REFERENCE_URL** - URL of orchestrator's endpoint to which notifications would be sent.
6. **VERSION** - for debug purposes during deployment. Automatically taken for the commit ID during image build.

## 2.5 Ranking

The Ranking component facilitates ranking mechanism for IoT resources. Ranking and resource selection rely on the registry built (and constantly updated) by crawling and indexing methods. The purpose of Ranking is to aid users and applications to not only find a set of resources relevant to their needs, but also to select the best or most appropriate one(s) from that set. There are multiple criteria for ranking IoT resources such as data type, proximity, latency, availability. The Ranking component supports application-dependent, multi-criteria ranking.

### 2.5.1 Integration into IoTCrawler Framework

Within the IoTCrawler framework the ranking component can be used by the orchestrator component to facilitate entity discovery. The ranking component relies on a NGSI-LD compliant endpoint as a backend, which is often times the *Indexing* component but could also be any NGSI-LD broker.

### 2.5.2 Ranking score computation

The ranking score is computed by a weighted sum function. Formally, for NGSI-LD properties $p_1 \dots p_n$ and corresponding weights $w_1 \dots w_n$ the ranking score $r$ is computed as following:

$$r = \sum_i p_i w_i$$

For the ranking component the weights and NGSI-LD property names for the values have to be supplied by the query.

#### Example for multi-criteria ranking

We want to rank NGSI-LD entities by the two (abbreviated) properties completeness and artificiality and weigh completeness 4 times more important than artificiality. Both NGSI-LD properties have values between 0 and 1 and we would like to get a ranking score also between 0 and 1. To achieve this the weights for both properties must add up to 1, therefore we fix the weight for completeness as 0.8 and the weight for artificiality as 0.2. If we now have two entities in to rank (with 'switched' values for completeness and artificiality), then the ranking scores are computed as follows:

| entity | completeness value | artificiality value | ranking score value |
|---|---|---|---|
| entity 1 | 0.88 | 0.92 | 0.888 |
| entity 2 | 0.92 | 0.88 | 0.912 |

Due to the higher weight of the completeness property, the ranking score favors entity 2 with the higher completeness value.

### 2.5.3 Setup

The Ranking component is provided as Docker image from Docker Hub (<TODO>: add docker hub URL), it only needs a configuration file.

For the configuration several environment variables (such as the backend NGSI-LD endpoint) have to be set. Either prepare a `config.env` file, similar to sample.env, or set the environment variables as in sample.env on the command line when starting the docker container.

Eventually run the following command to deploy the Ranking component:

```
docker run -p 3003:3003 --env-file=config.env iotcrawler-ranking
```

### 2.5.4 REST API

Please refer to the *Ranking REST API* documentation.

### 2.5.5 Details and code

Please refer to the Ranking Github repository for further details (e.g., how to build your own custom Docker image) and source code of the ranking component.

## 2.6 Indexing

As a core component in the IoTCrawler framework, Indexing provides a means for clients to search for IoT entities efficiently. It focuses on IoT streams and sensors, where queries can be based on sensor type and absolute or relative location. To initiate the process of indexing, a platform manager needs to register a metadata broker (MDR) with the Indexing component. In turn this will trigger the subscription to sensors and streams at the registered MDR. As metadata descriptions are updated at the MDR, the Indexing component will be notified, and will then index the sensors and streams based on location. For scalability, the Indexing component can be configured so that the persistence it relies on (MongoDB) can be sharded. With respect to other components in the framework, the Ranking component relies on Indexing for generating the ranking of search results.

- *Indexing*

### 2.6.1 Indexing and the IoT Crawler Architecture

In the IoTCrawler architecture, the Indexing component operates within the processing layer. It interacts primarily with the MDR (Scorpio Broker) and the Ranking component. Indexing mechanisms are used in order to support efficient search methods. While MDRs hold the complete metadata description of a resource, indexes on the other hand, contain only a subset of the resources' descriptions. In the case of the Ranking component, it relies on the efficient search mechanism of the Indexing component to be able to rank entities based on the search criteria.

### 2.6.2 Indexing Mechanism

IoTCrawler supports three types of indexing strategies: spatial, temporal and thematic.

The index consists of three parts: two indices that map Entity IDs of IoTStream and sosa:Sensor to geo-partition key, and a geo-partitioned index containing the data. A partition key is computed according to the location of the data. The partition key is computed by intersecting sensor location GeoJSON objects with predefined region GeoJSON polygons. Entities in the index are stored as a graph. Instead of storing all entities separately, linked entities are stored as a single document. The top-level entity is the IoTStream and all related entities are stored as nested documents in the properties of the parent document. This structure allows to define compound indices, thus accelerates nested queries.

### 2.6.3 Query Interface

The indexing component includes an NGSI-LD query interface and uses the data from the broker to respond to common queries. This component creates a subscription for metadata changes in the broker to keep the indexes up-to-date. Each time the metadata changes, the indexing component receives a notification and updates the index appropriately.

It exposes full NGSI-LD querying interface. `Link` header or *Temporal Entities* are **NOT** supported. If query is too complex (contains RegExp), involes entities (or related entities) or properties which are not part of the index, it will be forwared to NGSI-LD broker.

### 2.6.4 Implementation

Indexing is a Node.js web service written in TypeScript. It uses sharded MongoDB to store its data. It subscribes to the MDR broker for Stream, Sensor, QoI and Location changes and uses these notifications to keep index up-to-date.

### 2.6.5 Deployment

To deploy the Indexing component, several approaches can be taken.

- Build and run from source
- Build and run from Docker container
- Deploy on Kubernetes Cluster

#### Initial Setup

1. Install dependencies

   As a Node.js project, the dependencies need to first be installed in the target machine. In a terminal, the command for this is:

```
bash npm install
```

2. Configuration of Persistence (MongoDB)

   Indexing makes use of the MongoDB Sharding feaure. This allows the distribution of documents among multiple instances of MongoDB servers. The minimum number of instances required to enable sharding is 4; one acting as the query router, another as a configuration server for the sharded cluster, and the other as shards, where each contains a subset of the documents stored.

   Fig. 1: mongo_sharded

   To enable this, a set of scripts need to be run. These scripts require credentials to be set for the shared cluster and its handlers. These can be found under the source folder `mongodb\scripts`. MongoDB setup instructions are located in the `DEPLOYMENT.md` guide in the source repository.

3. Environment Variables

The web service reads environment variables to be able to configure itself relation to details for persistence, security endpoints, and default brokers. To do this, an `.env` file needs to be created and the variables set appropriately. A template `.env` file is provided in the source (`.env.example`), which can be copied and populated in the copied version, ensuring the file extension is `.env`.

4. Add country boundary data to the database

```
bash npm run populateDB
```

#### Run Web Service from Source

```
npm run dev
```

#### Running in Production mode

1. Build the service

   ```
   bash npm run build
   ```

2. Start the service

```
bash npm run start
```

**Alternative** To run the service inside a container follow the instructions in in the `DEPLOYMENT.md` guide in the source repository.

## 2.7 Seach-enabler

The search-enabler component is targeted to REST endpoint for GraphQL queries coming from IoT applications.

For creating, debugging and testing GraphQL queries a GraphiQL GUI environment provided.

See the tutorial for more details.

### 2.7.1 Build

The search-enabler is implemented using as IoTCrawlerClient provided by a core library (see Orchestrator for more details). As a result it builds orchestrator library during the build process.

*sh make.sh install*

During the build process all the modules are will be installed installed into a local maven repository.

### 2.7.2 Deployment

The search-enabler is deployable as a docker container.

A docker image for deployment can be build by the following command:

*sh make.sh build-image*

---

**Note:** A Google's Jib maven plugin is used for building image, so there is no traditional Dockerfile. Use build-command instead.

---

A helm chart configures the search-enabler to be deployed on Kubernetes cluster.

A gitlab CI configuration configures the building process deployment flow.

### 2.7.3 Configuration

1. **IOTCRAWLER_ORCHESTRATOR_URL** - URL of NGSI-LD endpoint (orchestrator or MDR Broker).
2. **VERSION** - for debug purposes during deployment. Automatically taken for the commit ID during image build.

## 2.8 Authorization Enabler

It is composed of multiple components to provide security in the IoTCrawler framework.

### 2.8.1 IdM - Keyrock

#### What is the IdM - Keyrock

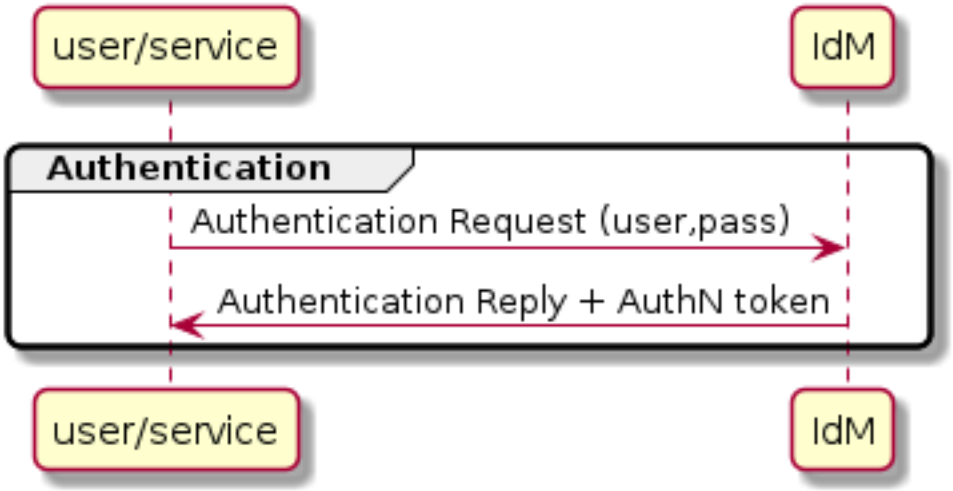
Keyrock is the FIWARE component responsible for Identity Management. Using Keyrock (in conjunction with other security components such as PEP Proxy and Authzforce) enables you to add OAuth2-based authentication and authorization security to your services and applications.

With IdM-Keyrock component the authentication step is covered in IotCrawler environment.

Further information can be found in the IdM-Keyrock documentation.

#### IotCrawler integration/functionality

Inside this environment, IdM-Keyrock provides a REST API for receiving authentication queries. When an authentication request is received, the component recovers user credentials from the JSON body request and returns an IdM token if it's the case. This token will be required by authorisation process (through Capability Manager or Security Facade).



#### API

Resources can be managed through the API (e.g. Users, applications and organizations). One of the main uses of IdM-Keyrock is to allow developers to add identity management (authentication and authorization) to their applications based on FIWARE identity. This is possible thanks to OAuth2 protocol.

The specific IdM-Keyrock requests required in IotCrawler environment are defined in the next REST API.

Further information can be found in the Keyrock Apiary.

#### How to deploy/test

This component can be deployed following the README.md file.

Once IdM-Keyrock is running you can test it. You can find postman collection with two requests needed inside IotCrawler environment in https://github.com/IoTCrawler/iotcrawler-samples/tree/master/authorization-enabler, with the POST one you can obtain a IdM token. You only need to define:

- `IdM-IP:IdM-Port` : Endpoint of IdM-Keyrock.
- Review `name` and `password` of configured IdM user you want to obtain token.

### 2.8.2 XACML - PAP - PDP

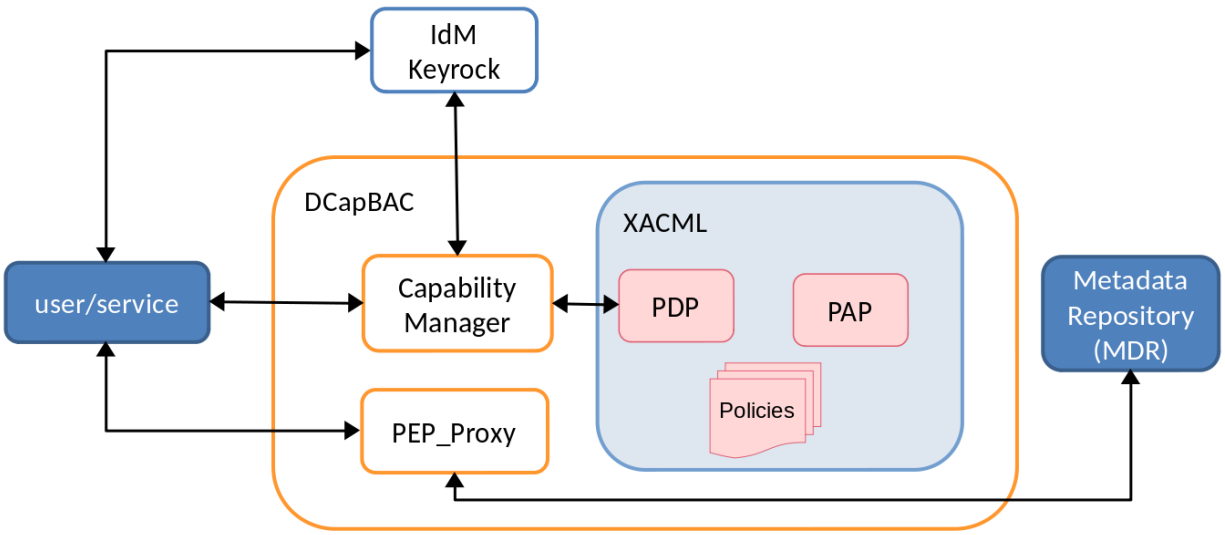
#### What is a XACML - PAP - PDP

This element corresponds to the implementation of the XACML framework. It comprises:

- a Policy Administration Point (PAP) which is responsible for managing the authorisation policies
- a Policy Decision Point (PDP), responsible for issuing possitive/negativer verdicts whenever an authorisation request is received.

The PAP presents a GUI for managing the XACML policies. They must be defined according to a triplet (subject, resource, action).

This project is developed in Java and it's a DCapBAC component as we can show in the next image:



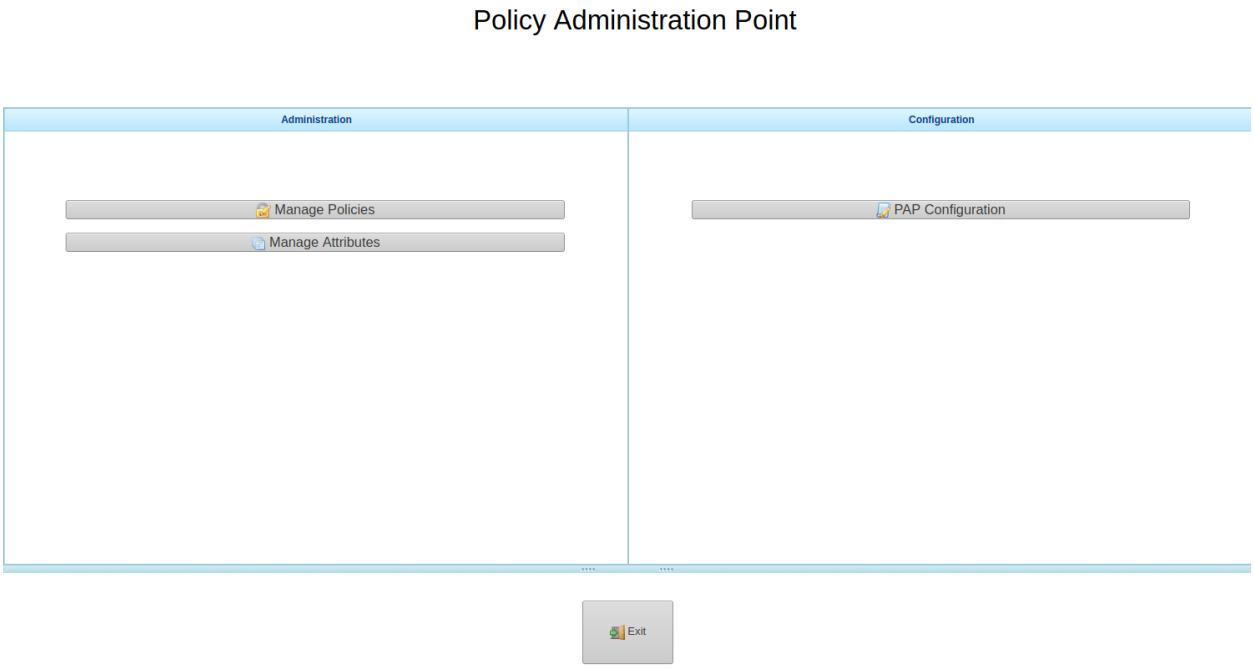

Remembering DCapBAC technology, where access control process is decoupled in two phases:

- 1st operation to receive authorisation. A token is issued
- 2nd operation access to the resource previously validating the token.

XACML-PDP is the component which validates the authorization request inside the first phase.

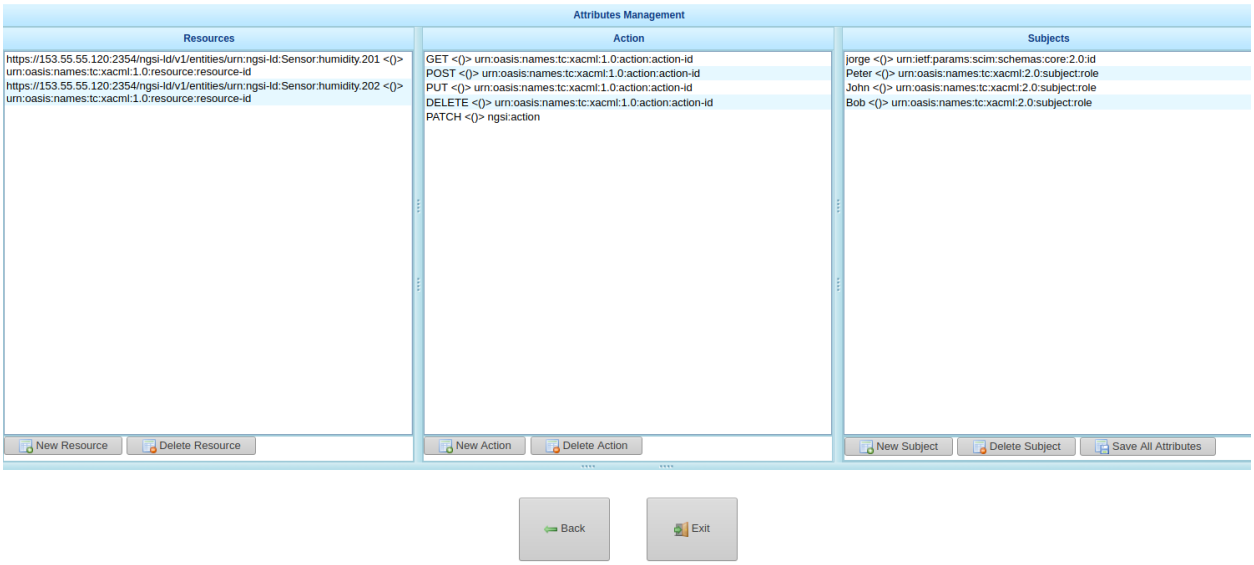
### IotCrawler integration/functionality

XACML-PAP, as mentioned above, it's a GUI for managing XACML policies (configuration), it's not interfering in the obtaining authorisation requests verdict. To define attributes and policies:



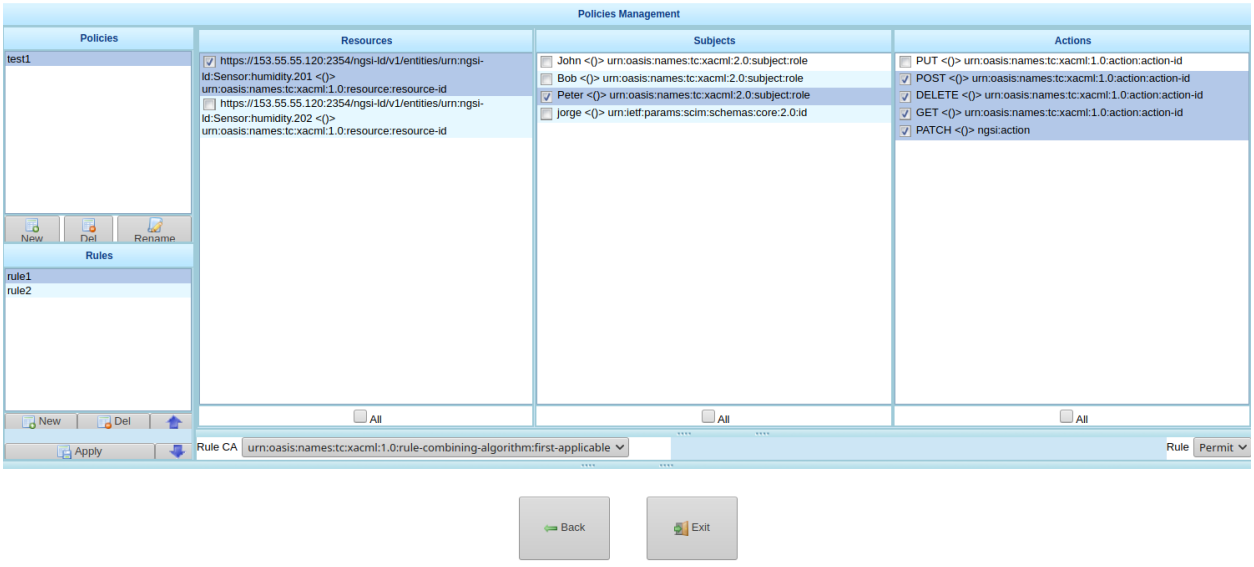

- Firstly, click "Manage Attributes" button to define the resources, actions and subjects. In subject's case you need specify the Usernames you defined in KeyRock – Identity Manager. To save click "Save All Attributes" and "Back".

Policy Administration Point




- Finally, click “Manage Policies” button to define the policies. Here you can see all attributes defined previously. In this page you must define Policies and into them rules. Each rule can link resources, actions and subjects and establish if this combination is “Permit” or “Deny”.

Policy Administration Point



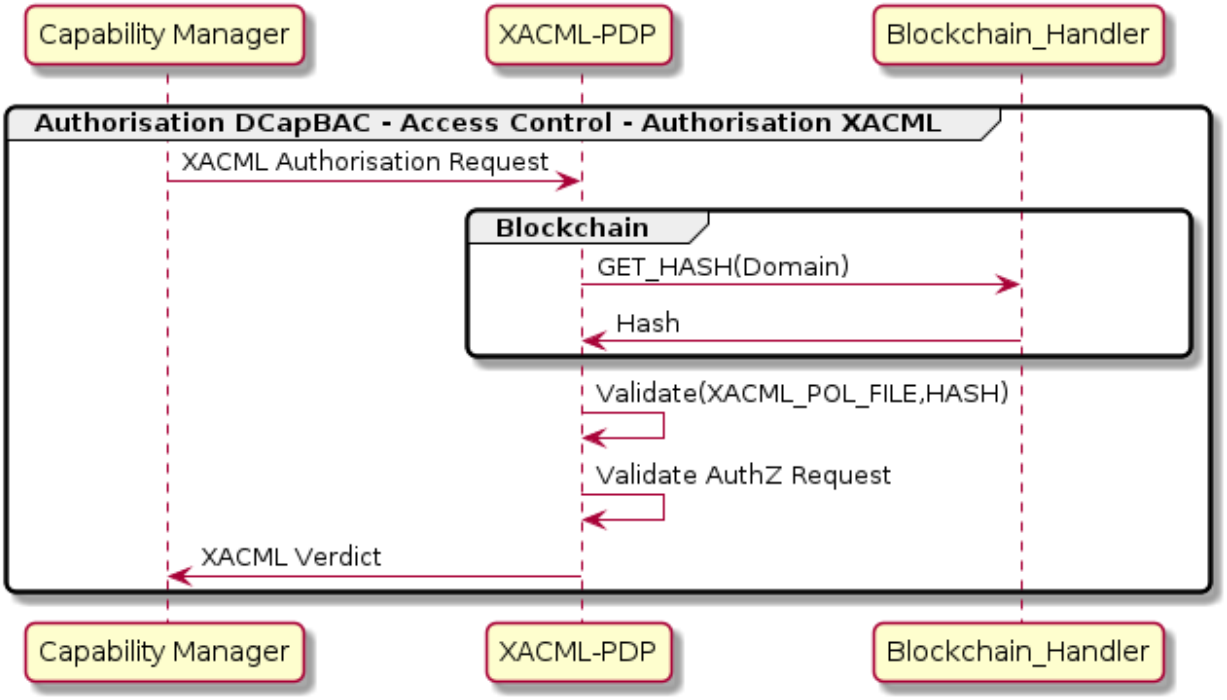

XACML_PDP returns the verdict when an authorisation request is received from Capability Manager, it recovers from the body:

- the subject of the resource’s request.

- the resource: endpoint + path of the resource's request.
- the action: method of the resource's request ("POST", "GET", "PATCH"...)

With this information, XACML-PDP:

- Access to the XACML policies for validating authorisation requests and obtain if the subject can access to a resource and can perform the action over the resource (verdict).
- In IoTCrawler project, PDP also access to Blockchain to obtain the hash of the domain and compare with the hash of XAML policies and if not equal verdict is negative.




### API

The XAML - PDP component supports the next REST API.

### How to deploy/test

This component can be deployed following the README.md file.

To test Policy Administration Point (PAP) and Policy Decision Point (PDP) the next components of IoTCrawler must be configured and launched.

- Blockchain.

Once XACML-PAP-PDP is running you can test it. You can find postman collection with a request sample to obtain a XACML verdict in https://github.com/IoTCrawler/iotcrawler-samples/tree/master/authorization-enabler. You only need to define:

- `XACML-PDP-IP:XACML-PDP-Port` : Endpoint of XACML-PDP. Default port: 8080
- Define `subject`, `resource` and `action` body parameters, where:
  - subject: subject of the resource's request. In DCapBAC scenario, it could correspond with a username (IDM). For example: "Peter"

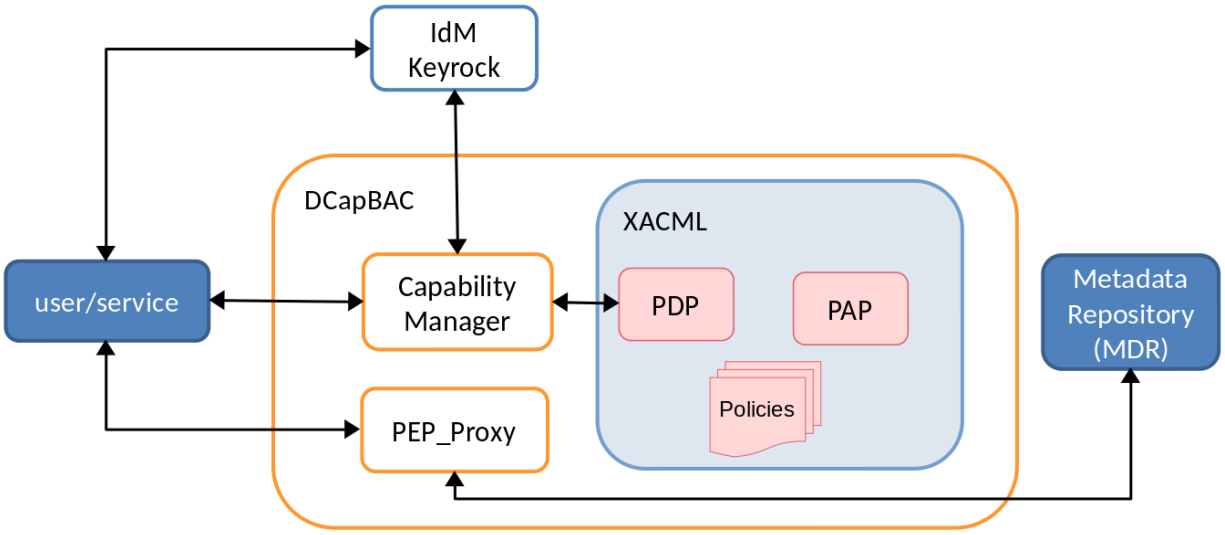
- resource: endpoint + path of the resource's request (protocol+IP+PORT+path). For example: "https://153.55.55.120:2354/ngsi-ld/v1/entities/urn:ngsi-ld:Sensor:humidity.201". In DCapBAC scenario, endpoint corresponds with the PEP-Proxy one.
- action: method of the resource's request ("POST", "GET", "PATCH"...)

### 2.8.3 Capability Manager

#### What is a Capability Manager

This component is the contact point for services and users that intend to access the resources stored in our IoTCrawler platform. It provides a REST API for receiving authorisation queries.

Capability Manager is developed in Python and makes use of the functionality developed in Java (.jar file), and it's a DCapBAC component as we can show in the next image:




Remembering DCapBAC technology, where access control process is decoupled in two phases:

- 1st operation to receive authorisation. A token is issued
- 2nd operation access to the resource previously validating the token.

Capability Manager covers the first one.

#### IotCrawler integration/functionality

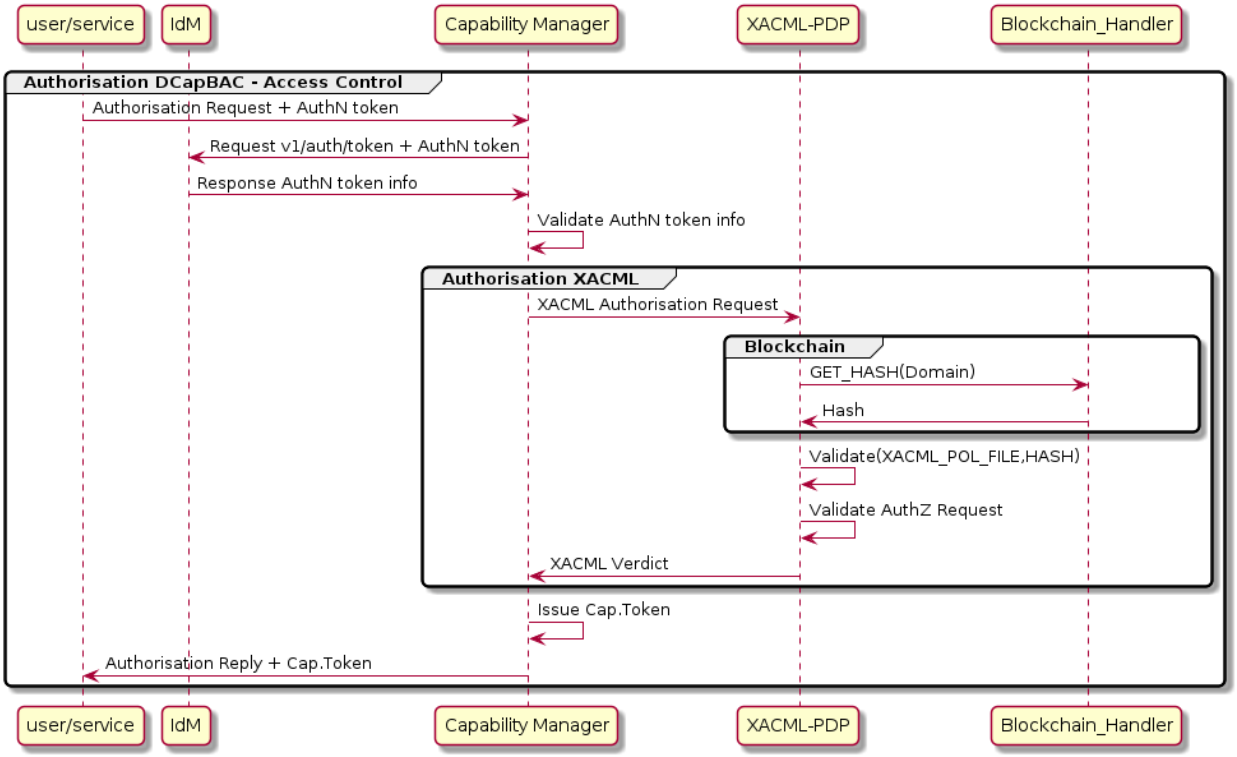
As mentioned above, this component provides a REST API for receiving authorisation queries, which are tailored and forwarded to the XACML PDP for a verdict.

When an authorisation request is received by Capability Manager, it recovers from the JSON body:

- an authentication token which proceeds from authentication phase (access to IdM-Keyrock).
- an endpoint of the resource's request (protocol+IP+PORT). In DCapBAC scenario, it corresponds with PEP-Proxy component.
- the action/method of the resource's request ("POST", "GET", "PATCH"...)
- the path of the resource's request

With this information, Capability Manager:

- Access to authentication component (IdM-Keyrock) to validate authentication token.
- Access to XACML framework for validating authorisation requests (through PDP) and obtain if the subject can access to a resource and can perform the action over the resource (verdict). In IoTCrawler project, PDP also access to Blockchain to obtain the hash of the domain and compare with the hash of XAML policies and if not equal verdict is negative, this additional validation doesn't affects to Capability Manager.
- If a positive verdict is received, finally, the Capability Manager issues an authorisation token called Capability Token which is a signed JSON document which contains all the required information for the authorisation, such as the resource to be accessed, the action to be performed, and also a time interval during the Capability Token is valid. This token will be required to access to the resource (through PEP-Proxy).




### Capability token's considerations:

The format of the capability token is based on JSON. Compared to more traditional formats such as XML, JSON is getting more attention from academia and industry in IoT scenarios, since it is able to provide a simple, lightweight, efficient, and expressive data representation, which is suitable to be used on constrained networks and devices. As shown below, this format follows a similar approach to JSON Web Tokens (JWTs), but including the access rights that are granted to a specific entity. A brief description of each field is provided.

- Identifier (ID). This field is used to unequivocally identify a capability token. A random or pseudorandom technique will be employed by the issuer to ensure this identifier is unique.
- Issued-time (II). It identifies the time at which the token was issued as the number of seconds from 1970-01-01T0:0:0Z.
- Issuer (IS). The entity that issued the token and, therefore, the signer of it.
- Subject (SU). It makes reference to the subject to which the rights from the token are granted. A public key has been used to validate the legitimacy of the subject. Specifically, it is based on ECC, therefore, each half of the field represents a public key coordinate of the subject using Base64.

- Device (DE). It is a URI used to unequivocally identify the device to which the token applies.
- Signature (SI). It carries the digital signature of the token. As a signature in ECDSA is represented by two values, each half of the field represents one of these values using Base64.
- Access Rights (AR). This field represents the set of rights that the issuer has granted to the subject.
  - Action (AC). Its purpose is to identify a specific granted action. Its value could be any CoAP method (GET, POST, PUT, DELETE), although other actions could be also considered.
  - Resource (RE). It represents the resource in the device for which the action is granted.
  - Condition flag (F). It states how the set of conditions in the next field should be combined. A value of 0 means AND, and a value of 1 means OR.
  - Conditions (CO). Set of conditions which have to be fulfilled locally on the device to grant the corresponding action.
    - Condition Type (T). The type of condition to be verified.
    - Condition value (V). It represents the value of the condition.
    - Condition Unit (U). It indicates the unit of measure that the value represents.
- Not Before (NB). The time before which the token must not be accepted. Its value cannot be earlier than the II field and it implies the current time must be after or equal than NB..
- Not After (NA). It represents the time after which the token must not be accepted.

### API

The Capability Manager component supports the next REST API.

### How to deploy/test

This component can be deployed following the README.md file.

To test Capability Manager the next components of IoTCrawler must be configured and launched.

- MetaData Repository (MDR).
- IdM-Keyrock.
- XACML-PDP.

Once Capability Manager is running you can test it. You can find postman collection with two requests needed to obtain a capability token in https://github.com/IoTCrawler/iotcrawler-samples/tree/master/authorization-enabler. You only need to define:

- `IdM-IP:IdM-Port` : Endpoint of IdM-Keyrock.
- Review `name` and `password` of configured IdM user you want to obtain token.
- `CapMan-IP:CapMan-Port` : Endpoint of Capability Manager. Default port: 3030
- `action` : Example: “GET”,
- `PEP-Proxy-IP:PEP-Proxy-Port` : Endpoint of PEP-Proxy. Default port: 1028
- `resource`: path of the resource’s request. Example: “/scorpio/v1/info/”

### 2.8.4 Security Facade

#### What is a Security Facade

This component has been designed as an endpoint for performing both authentication and authorisation operations in a transparent way for the requester. It has been developed in Java.

In a traditional environment, we need to send three requests to have access to a resource:

- 1st operation to authenticate through user credentials (IdM-KeyRock)
- 2nd operation to receive authorisation. A token is issued (Capability Manager - DCapBAC technology component).
- 3rd operation access to the resource previously validating the token (PEP-Proxy - DCapBAC technology component).

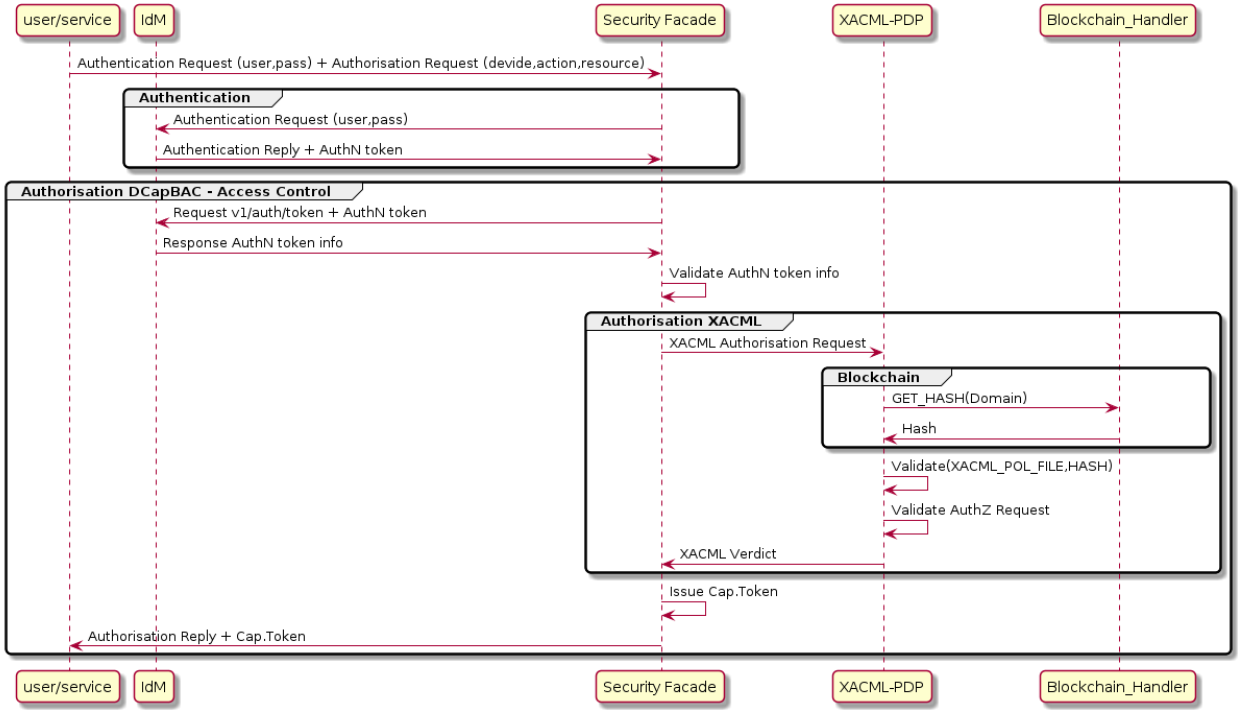
Security Facade offers an API to do possible the fusion of the two first requests in only one.

#### IotCrawler integration/functionality

As mentioned above, this component provides a REST API, when a request is received by Security Facade, it recovers from the JSON body:

- user credentials of IdM-Keyrock (access to IdM-Keyrock).
- an endpoint of the resource's request (protocol+IP+PORT). In DCapBAC scenario, it corresponds with PEP-Proxy component.
- the action/method of the resource's request ("POST", "GET", "PATCH"...)
- the path of the resource's request

With this information, Security Facade interacts with Identity Manager and the Authorization Authorizer (XACML-PDP), and in case the requester has the appropriate permissions for the request, it sends the authorization token (Capacity Token) back to the requester.




### Capability token's considerations:

The format of the capability token is based on JSON. Compared to more traditional formats such as XML, JSON is getting more attention from academia and industry in IoT scenarios, since it is able to provide a simple, lightweight, efficient, and expressive data representation, which is suitable to be used on constrained networks and devices. As shown below, this format follows a similar approach to JSON Web Tokens (JWTs), but including the access rights that are granted to a specific entity. A brief description of each field is provided.

- Identifier (ID). This field is used to unequivocally identify a capability token. A random or pseudorandom technique will be employed by the issuer to ensure this identifier is unique.
- Issued-time (II). It identifies the time at which the token was issued as the number of seconds from 1970-01-01T0:0:0Z.
- Issuer (IS). The entity that issued the token and, therefore, the signer of it.
- Subject (SU). It makes reference to the subject to which the rights from the token are granted. A public key has been used to validate the legitimacy of the subject. Specifically, it is based on ECC, therefore, each half of the field represents a public key coordinate of the subject using Base64.
- Device (DE). It is a URI used to unequivocally identify the device to which the token applies.
- Signature (SI). It carries the digital signature of the token. As a signature in ECDSA is represented by two values, each half of the field represents one of these values using Base64.
- Access Rights (AR). This field represents the set of rights that the issuer has granted to the subject.
  - Action (AC). Its purpose is to identify a specific granted action. Its value could be any CoAP method (GET, POST, PUT, DELETE), although other actions could be also considered.
  - Resource (RE). It represents the resource in the device for which the action is granted.
  - Condition flag (F). It states how the set of conditions in the next field should be combined. A value of 0 means AND, and a value of 1 means OR.

  - Conditions (CO). Set of conditions which have to be fulfilled locally on the device to grant the corresponding action.
    * Condition Type (T). The type of condition to be verified.
    * Condition value (V). It represents the value of the condition.
    * Condition Unit (U). It indicates the unit of measure that the value represents.
- Not Before (NB). The time before which the token must not be accepted. Its value cannot be earlier than the II field and it implies the current time must be after or equal than NB..
- Not After (NA). It represents the time after which the token must not be accepted.

#### API

The Security Facade component supports the next REST API.

#### How to deploy/test

This component can be deployed following the README.md file or for a

To test Capability Manager the next components of IoTCrawler must be configured and launched.

- MetaData Repository (MDR).
- IdM-Keyrock.
- XACML-PDP.

Once Security Facade is running you can test it. You can find postman collection with the request needed to obtain a capability token in https://github.com/IoTCrawler/iotcrawler-samples/tree/master/authorization-enabler. You only need to define:

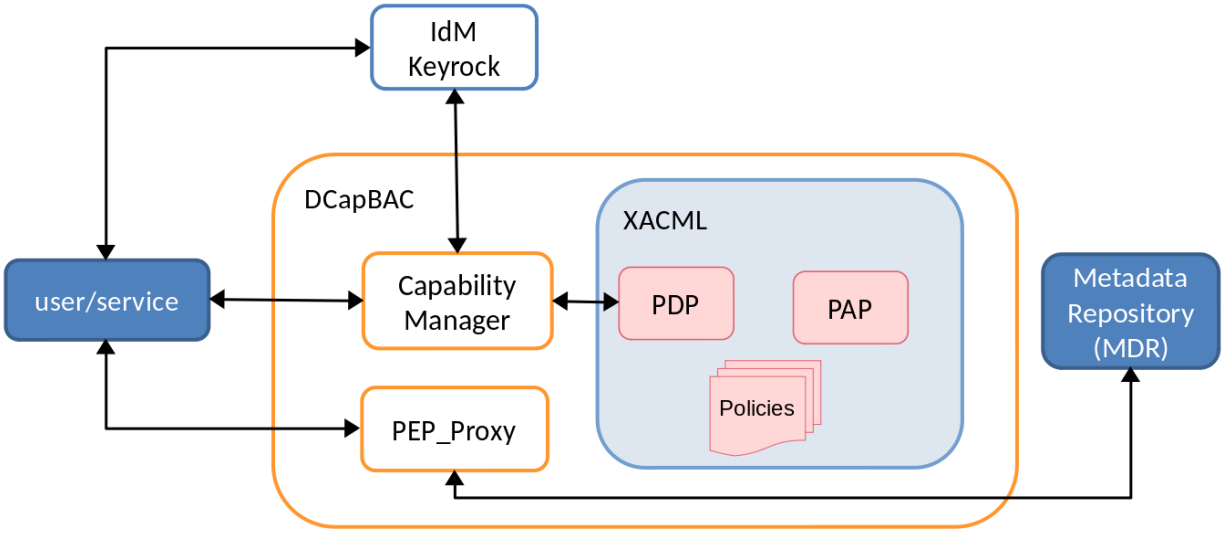
- `Facade-IP:Facade-Port` : Endpoint of Security Facade. Default port: 8443
- Review `name` and `password` of configured IdM user you want to obtain capability token.
- `action` : Example: "GET",
- `PEP-Proxy-IP:PEP-Proxy-Port` : Endpoint of PEP-Proxy. Default port: 1028
- `resource`: path of the resource's request. Example: "/scorpio/v1/info/"

### 2.8.5 PEP-Proxy

#### What is a PEP-Proxy

PEP_Proxy is the component responsible for receiving the queries aimed to access to a resource, they are accompanied by the corresponding `Capability Token` and forwards requests to the corresponding endpoint (for example MetaData Repository) and the responses back to the requester.

PEP-Proxy is developed in Python and makes use of the functionality developed in Java (.jar file), and it's a DCapBAC component as we can show in the next image:




Remembering DCapBAC technology, where access control process is decoupled in two phases:

- 1st operation to receive authorisation. A token is issued
- 2nd operation access to the resource previously validating the token.

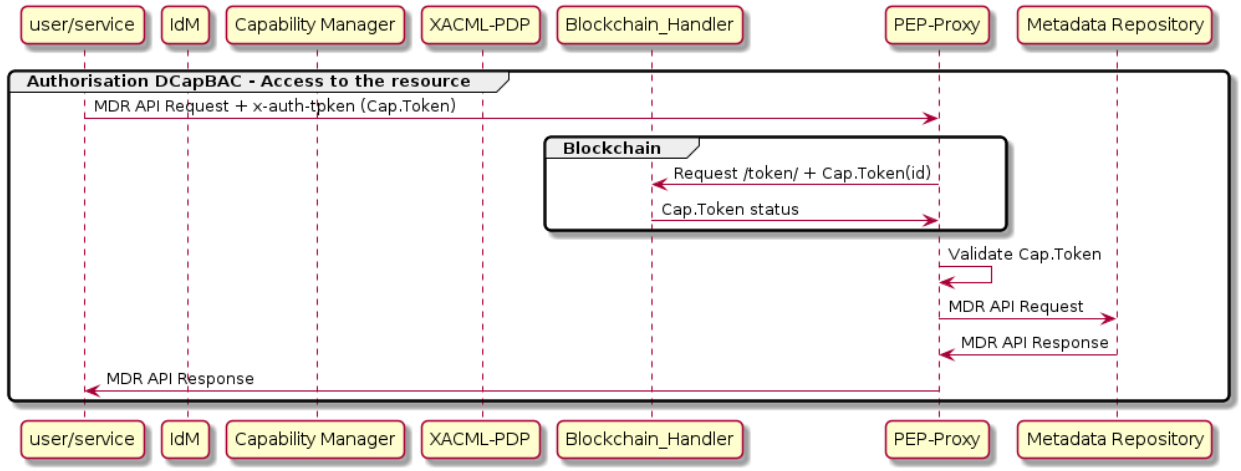
PEP-Proxy covers the second one.

### IotCrawler integration/functionality

As mentioned above, this component receives queries aimed to access to a resource, queries contain a `Capability Token`. The PEP-Proxy validates this token, and in case the evaluation is positive, it forwards requests to the specific endpoint's API.

When an access resource request is received by PEP-Proxy:

- recovers the `x-auth-token` header (`Capability Token`)-
- If Blockchain integration is configured in PEP-Proxy, that's the IoTCrawler case, PEP-Proxy access to Blockchain to obtain `Capability Token` status to validate if it was revoked.
- validate `Capability Token`.
- If `Capability Token` validation is successful, PEP-Proxy forwards the message and sends responses back to the requester.




### API

The PEP-Proxy component supports multiple REST APIs.

### How to deploy/test

This component can be deployed following the README.md file.

To test PEP-Proxy the next components of IoTCrawler must be configured and launched.

- MetaData Repository (MDR).
- IdM-Keyrock.
- XACML-PDP.
- Capability Manager

Once PEP-Proxy is running you can test it. You can find postman collection with requests needed to obtain access tp a resource in https://github.com/IoTCrawler/iotcrawler-samples/tree/master/authorization-enabler. You only need to define:

- `IdM-IP:IdM-Port` : Endpoint of IdM-Keyrock.
- Review `name` and `password` of configured IdM user you want to obtain token.
- `CapMan-IP:CapMan-Port` : Endpoint of Capability Manager. Default port: 3030
- `action` : Example: “GET”,
- `PEP-Proxy-IP:PEP-Proxy-Port` : Endpoint of PEP-Proxy. Default port: 1028
- `re`: Example: “/scorpio/v1/info/”

## 2.9 Semantic Enrichment

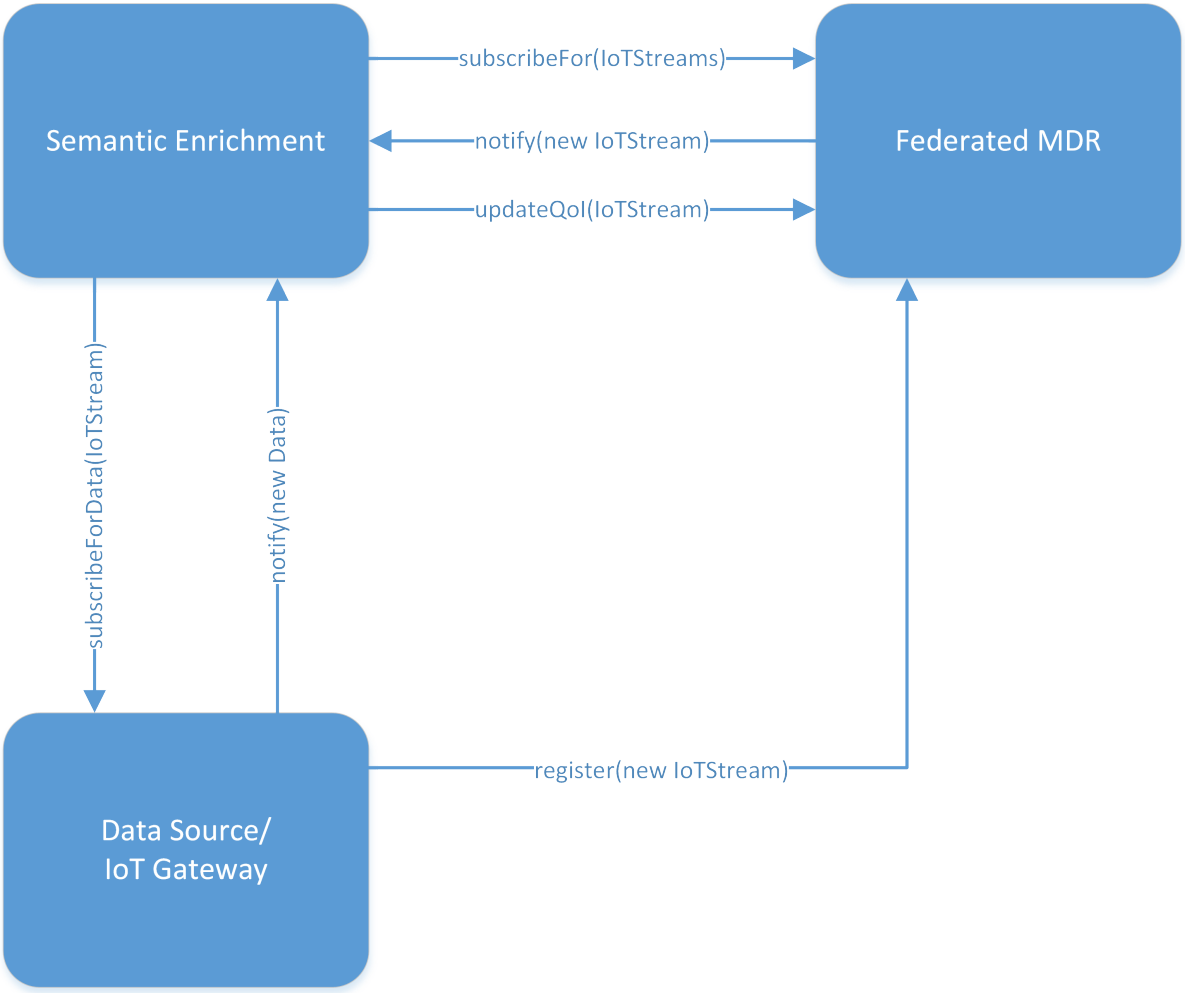
The Semantic Enrichment (SE) is composed of two subcomponents: an analysis component for Quality of Information and a Pattern Extractor to gain additional knowledge.

### 2.9.1 Quality Analyser

The Semantic Enrichment is one of the core components of the IoTCrawler framework. Besides the Pattern Extractor component described in a separate section, it also contains the Quality Analyser, which is responsible for the Quality of Information (QoI) computation. This QoI will further enrich the data provided by a stream so that other components, e.g. the ranking, make use of if.

#### Integration into IoTCrawler Framework

The IoTCrawler framework itself is loosely coupled by a set of single components to be on the one hand as adoptable as possible, by keeping the flexibility to optimise single components without the need to adapt the complete framework, and on the other hand to ensure scalability. The integration mainly focuses on publish/subscribe mechanisms that are used to communicate with the Federated MDR as the central register for all data sources and to access received data from IoT gateways in different domains. As the example data sources used within the IoTCrawler project focus on NGSI-LD, the NGSI-LD publish/subscribe methods are used, but in principle the data access can also be realised by other access methods like MQTT or AMQP protocols. In these cases, the interfaces must be described in the data source's metadata.




The figure above provides a detailed view of the interactions with the "neighbouring" components within the IoTCrawler framework. The interaction is divided into the following steps:

1. In the first step, the Semantic Enrichment sends a subscription message to the Federated MDR to subscribe to all updates in IoTStreams, where an IoTStream is a class type defined in the IoTCrawler information model. This allows the Semantic Enrichment to receive notifications for newly added or updated data streams.
2. In the next step a new data source is registered as an IoTStream at the Federated MDR. The registration contains details on how to access the related data source and additional metadata, e.g. a detailed description of the data sources properties and its characteristics.
3. For each newly registered or updated data source the Semantic Enrichment receives a notification. This allows to adopt the QoI calculation to changes in the metadata or to connect to a new data endpoint description to access data.
4. The Semantic Enrichment component subscribes to the data source based on the information delivered in the notification received from the Federated MDR.
5. In the last step the Semantic Enrichment calculates the QoI for each known data source and stores it to the metadata information within the Federated MDR. This allows other IoTCrawler components, as well as third-party users to access the additional information.

### QoI calculation

In this section we will show how the provided input and output for the quality analysis have to look like. This is strongly related to the IoTCrawler information model. The QoI Analyser relies on provided metadata for the QoI calculation. This is annotated in the description of the sensor that provides a data stream. An example sensor providing several metadata is shown below.

```
{
    "id": "urn:ngsi-ld:Sensor:B4:E6:2D:8A:20:DD:Temperature",
    "type": "sosa:Sensor",
    "location": {
        "type": "GeoProperty",
        "value": {
        "type": "Point",
        "coordinates": [
            52.28267,
            8.023755
        ]
        }
    },
    "sosa:isHostedBy": {
        "type": "Relationship",
        "object": "urn:ngsi-ld:Platform:B4:E6:2D:8A:20:DD"
    },
    "sosa:observes": {
        "type": "Relationship",
        "object": "urn:ngsi-ld:ObservableProperty:B4:E6:2D:8A:20:DD:Temperature"
    },
    "sosa:madeObservation": {
        "type": "Relationship",
        "object": "urn:ngsi-ld:StreamObservation:B4:E6:2D:8A:20:DD:Temperature"
    },
    "qoi:updateinterval": {
        "type": "Property",
        "value": 10,
        "qoi:unit": {
        "type": "Property",
        "value": "seconds"
        }
    },
    "qoi:valuetype": {
        "type": "Property",
        "value": "float"
    },
    "qoi:min": {
        "type": "Property",
        "value": -20
    },
    "qoi:max": {
        "type": "Property",
        "value": 50
    },
    "@context": [
        "http://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
        {
        "sosa": "http://www.w3.org/ns/sosa/",
        "qoi": "https://w3id.org/iot/qoi#"
        }
```

(continues on next page)

(continued from previous page)

```
    ]
}
```

This sensor is related to the following data stream:

```
{
    "id": "urn:ngsi-ld:IotStream:B4:E6:2D:8A:20:DD:Temperature",
    "type": "iot-stream:IotStream",
    "iot-stream:generatedBy": {
        "type": "Relationship",
        "object": "urn:ngsi-ld:Sensor:B4:E6:2D:8A:20:DD:Temperature"
    },
    "qoi:hasQuality": {
        "type": "Relationship",
        "object": "urn:ngsi-ld:QoI:urn:ngsi-ld:IotStream:B4:E6:2D:8A:20:DD:Temperature
↪"
    },
    "@context": [
        "http://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
        {
        "iot-stream": "http://purl.org/iot/ontology/iot-stream#",
        "qoi": "https://w3id.org/iot/qoi#"
        }
    ]
}
```

During quality calculation the relationship "qoi:hasQuality" will be set to a new QoI entity similar to the following one:

```
{
    "id":"urn:ngsi-ld:QoI:urn:ngsi-ld:IotStream:B4:E6:2D:8A:20:DD:Temperature",
    "type":"qoi:Quality",
    "@context":[
        "http://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
        {
            "qoi":"https://w3id.org/iot/qoi#"

    }

    ],
    "qoi:plausibility":{
        "type":"Property",
        "value":"NA",
        "qoi:hasAbsoluteValue":{
            "type":"Property",
            "value":1

    },
        "qoi:hasRatedValue":{
            "type":"Property",
            "value":1.0

    }

    },
    "qoi:completeness":{
```

(continues on next page)

(continued from previous page)


```
        "type":"Property",
        "value":"NA",
        "qoi:hasAbsoluteValue":{
            "type":"Property",
            "value":1

    },
        "qoi:hasRatedValue":{
            "type":"Property",
            "value":0.6000000000000001

    }

    },
    "qoi:age":{
        "type":"Property",
        "value":"NA",
        "qoi:hasAbsoluteValue":{
            "type":"Property",
            "value":10462442.743888

    }

    },
    "qoi:frequency":{
        "type":"Property",
        "value":"NA",
        "qoi:hasAbsoluteValue":{
            "type":"Property",
            "value":11.000385

    },
        "qoi:hasRatedValue":{
            "type":"Property",
            "value":0.6000000000000001

    }

    }
}
```


### Sourcecode

The sourcecode for the component is available at https://github.com/IoTCrawler/SemanticEnrichment

### Installation

The Quality Analyser is provided with a docker file. Within this dockerfile there are several environment variables that have to be provided:

- NGSI_ADDRESS http://mdr.iotcrawler.eu/
- SE_HOST 0.0.0.0
- SE_PORT 8081
- SE_CALLBACK https://semantic-enrichment.iotcrawler.eu/semanticenrichment/callback

The NGSI_ADDRESS is the address of the MDR where information about all available sensors and streams are stored. This address is needed to subscribe for incoming or changing metadata. The SE_HOST is the address/interface where the Quality Analyser component should bind to. SE_PORT defines the port where the component is accessible. Finally SE_CALLBACK is the callback address of the component. This address is used for the subscriptions send to the MDR and will receive all notifications.

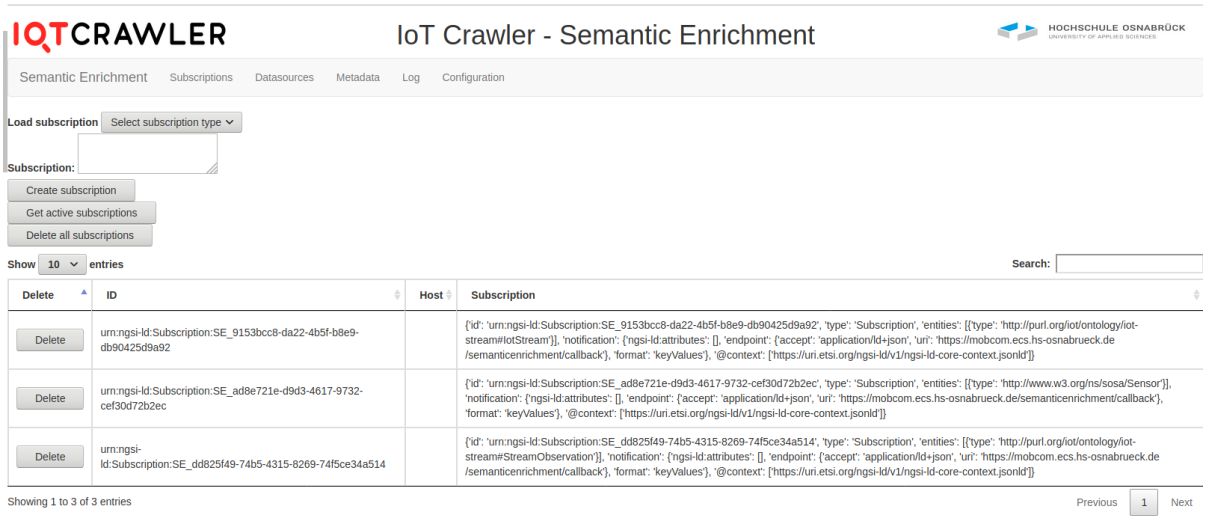
For installation the provided docker-compose script can be used to build and start the component:

- build: docker-compose build
- start: docker-compose up -d

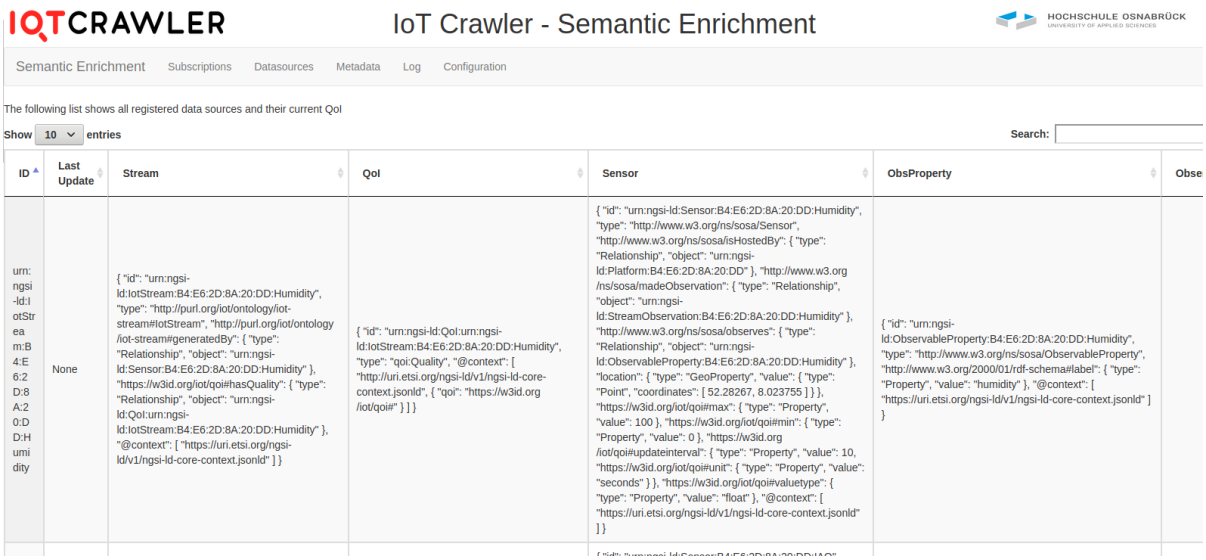
This will start a docker container called “semanticenrichment”.

### UI

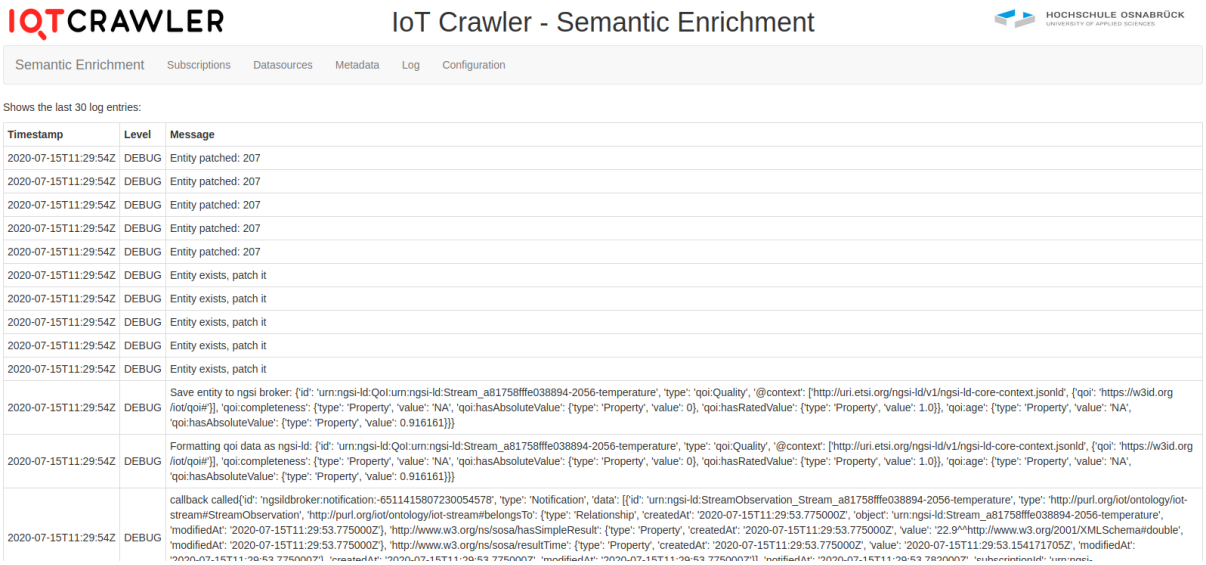
The UI provides some useful information about running subscriptions, known data sources, and the current configuration.



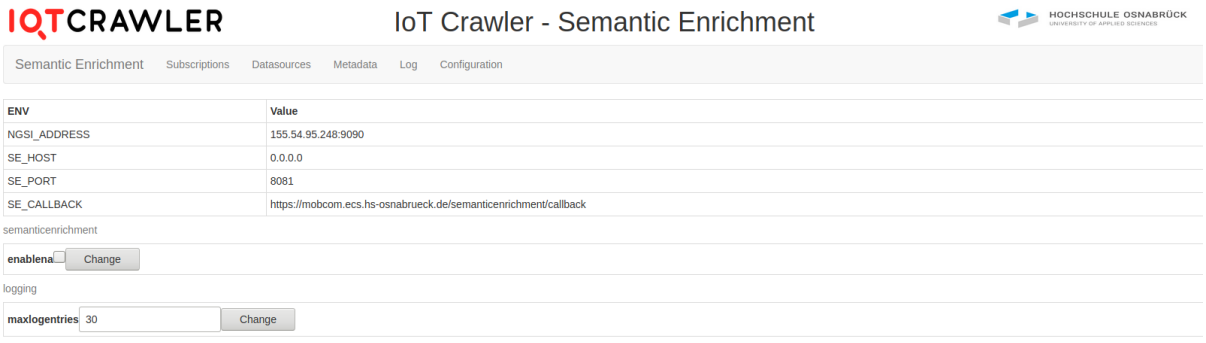

The subscription UI above shows running subscriptions and provides a form to create new subscriptions.




The UI above lists the known data sources in its IoTCrawler information model format.




The log provides information about incoming data, internal exceptions etc.




The configuration UI allows to check the set environment variables and to set the log size provided in the UI.

### 2.9.2 Pattern Extractor

As part of the Semantic Enrichment component, the Pattern Extraction (PE) module enables the generation of higher-level context by analysing the annotated IoT data streams retrieved from IoT data sources that are pushed to the Metadata Repository (MDR). The Pattern Extractor uses the `iot-stream:StreamObservations` to detect higher-level events, which are then published to the broker. A machine learning method is used to analyse a group of `sosa:observesProperties`. This analysis model is registered as a new subscription in the broker. This subscription includes spatial and temporal and the data type specifications for data streams. Once a stream matches the specifications of a registered process, the pattern extractor then caches the observations that are necessary to perform the analysis. When the observations are received for each stream in the group, the pattern extractor constructs a time window to analyse the data. The result of the analysis is a label for a pattern of data. This label, together with the start and end times of the pattern, is then represented as an '"iot-stream:Event"' NGSI-LD entity and published into the broker. The broker then makes these patterns searchable.

#### Known dependencies

- MDR

#### Deployable Container

- Docker

#### Installation

The Pattern Extraction is provided with an `*.env` file for setting environmental variables, a Dockerfile and Docker-Compose file. The first step is to configure the environment variables in the `*.env` file, which relate to:

- Persistence address and credentials
- Authentication enablement, addresses and credentials.
- Broker address and API operation restriction
- TLS validation enablement

A template file `env.example` is provided which should be used and saved as an `.*env` file. Once these variables are set, the local persistence used, in this case MongoDB, needs to be configured and running in order for the pattern extraction to run. A mongoDB script needs to be run on the target Mongo instance, which will create an admin user and also a database with a user. The script must be edited to mirror the persistence address and credentials set in the `env` file.

For deployment, this can be done by either compiling the project source using npm, or by using docker-compose, which will use the Dockerfile to instantiate the pattern extraction module in a container, in addition to creating containers for the required MongoDB instances. The steps to achieve this are by using the commands below in a terminal:

1. Build docker images
   ```
   docker-compose build
   ```
2. Create docker containers
   ```
   docker-compose up --no-start
   ```
3. Start Mongo DB
   ```
   docker-compose start mongo
   ```
4. Login into mongo container and configure authentication (execute command from mongodb/scripts/mongo.js in the mongo shell).

```
docker exec -it pattern-mongo mongo
```

5. Start Pattern Extractor service

```
docker-compose start patternExtractor
```

This will start a docker container called “patternExtractor”.

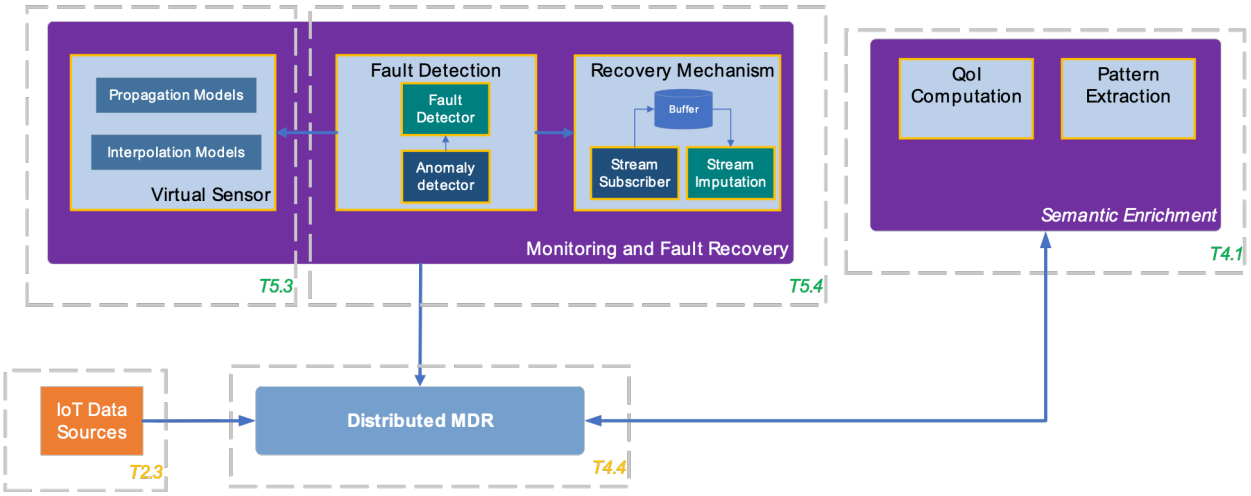
## 2.10 Monitoring

The main task of the Monitoring component in IoTCrawler is to monitor the connected data streams in order to detect faulty/anomalous data samples.

The subcomponent **Fault Detection** is responsible to detect ‘unusual behaviour’ in the data streams and determine if a fault is present. In this case, counter measures are triggered, including recovery mechanisms to provide a quick response by imputing single *StreamObservations* or deploying a Virtual Sensor. To detect faults in a single data stream the **Fault Detecttion** uses a modified Dirichlet Process Gaussian State Machine Model and a ARIMA-based approach. The **Fault Recovery** utilises Markov Chain Monte Carlo (MCMC) methods to generate sensor samples.

In case a faulty data stream is detected, the subcomponent **Virtual Sensor Creator** is able to deploy a Virtual Sensor as a counter measure. It uses machine learning techniques to create sensor samples in relation to correlating, neighbouring sensors.

The following figure shows an overview of the **Monitoring** component and the interactions with other IoTCrawler components. For more details see the respective subsections.




The documentation for the **Monitoring’s** subcomponents can be found here:

### 2.10.1 Fault Detection

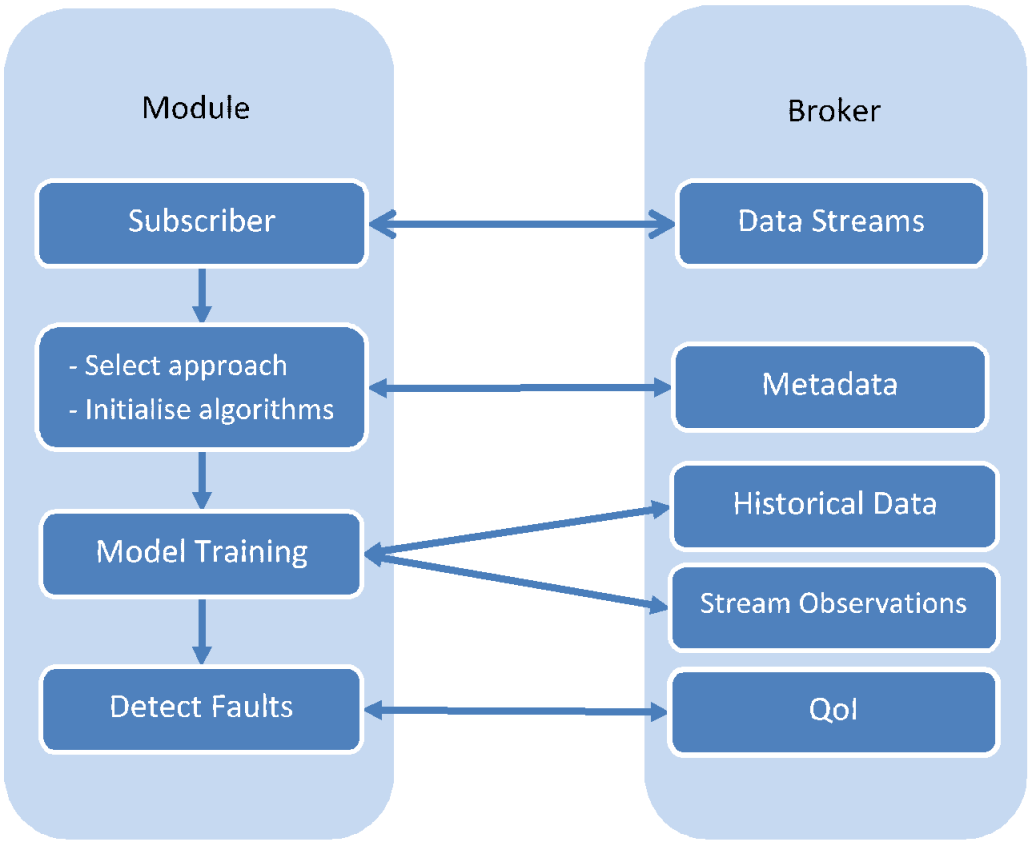
The Fault Detection components monitors every data stream that is available to the IoTCrawler framework. In the first layer, the component categorises faults as definite faults (faults that occur due to hardware issues) or as anomalies, which could occur because of a brief environmental factor, an unexpected behaviour detected through learned patterns and contextual information. These anomalies can be categorised as faults, if they persist for a longer period of time.

As the detection of faults requires different approaches in different scenarios, not every data stream can have the best detector suited to its needs. To cater to the needs of most of the sensor streams, the fault detection component uses two approaches, ARIMA based fault detection with time series analysis and a Dirichlet Process Gaussian State Model, which determines the likelihood for a value to occur based on the previous observations of the sensor.

### Workflow

The Fault Detection component subscribes to any new data stream that becomes available through the MDR. Through the metadata, the Fault Detection will determine which approach should be used. This is differentiated based on how much information is provided in metadata. For example, if seasonality/periodicity of a data stream is defined and the historical data is available for the desired duration, then ARIMA can be applied.

For the actual detection, the component will make use of the trained models and the corresponding QoI values for each data stream, calculated for each stream by the Semantic Enrichment. It will then notify the MDR in case of faults.




### Stop a Fault Detection Instance

The Fault Detection is an autonomous component and does not require human supervision. However, because it can not be used in every scenario, e.g. if it detects a lot of false positives, it can be manually stopped for an individual data stream. Stopping the Fault Detection is not performed autonomously in any case as the ground truth is not known in most scenario and hence, an accurate decision cannot be made. This method allows a manual intervention, e.g. by an administrator of an IoTCrawler instance.

**`GET /api/stopFaultDetection`**

with payload

```
{
  "sensorID" : <The ID of the sensor to stop>
}
```

#### Restart a Fault Detection Instance

A need to restart a Fault Detection instance may arise when the information related to the stream is changed. This change in information may happen due to the change of sensor device, correction of human error or some added insights about the data from the provider. This will require the Fault Detection instance to be restarted to re-evaluate the best available approach, reset the learning parameters and to train on relevant historical data.

**`GET /api/restartFaultDetection`**

with payload

```
{
  "sensorID" : <The ID of the sensor to restart>
}
```

### 2.10.2 Fault Recovery

### 2.10.3 Virtual Sensor Creator

This component is responsible for the creation of Virtual Sensors in the IoTCrawler framework. A Virtual Sensor can be used to replace a broken sensor. To do this, the component searches for other data sources within a given radius. It then calculates if the data by broken sensor correlates with the other data source candidates. Candidates with a high correlation are then used to train various ML models. Via grid search the best suitable model is selected for the Virtual Sensor. The model is then used to predict new virtual *StreamObservations*, which published via the MDR.

#### Usage

The component uses a HTTP interface to deploy and undeploy Virtual Sensors and is intended to be called by the Fault Detection component.

All requests are replied with a JSON response in the following structure:

```
{
    "status": "ok" | "error",
    "success": true | false,
    "description": <error description if not successful>,
    "data": <Optionally, the data in case of success>
}
```

#### Replace a (broken) Sensor with a Virtual Sensor

This method can be used to replace a broken sensor. The request requires the ID of the broken sensor to replace. The optional parameter **maxDistance** can be used to limit the radius the component will search for data sources that may be used as input for the Virtual Sensor (default 20km). With the optional parameter **onlySameObservations** the Virtual Sensor Creator will look only for other data sources measuring the same physical phenomena, identified by the *ObservableProperty's* label. The found data sources, or their data to be precise, is tested if it correlates with the data of the broken sensor, before training the ML model. Not correlating sources are not considered in the training. With **limitSourceSensors**, the number of data sources can be limited further. This may be necessary if data sources have a high number of historic data samples, since it can cause a high strain on computational resources and also require more computational time throughout the Virtual Sensor deployment pipeline, i.e. from the preprocessing of data to the training of the model.

This may be necessary if data sources have a high number of historic data samples, since their temporal alignment to form the training data requires a lot of time and memory. This limit takes effect before the test for correlations, thus the final number of contributing data sources may be even smaller.

If correlating data sources are found, this method returns the NGSI-LD description for the created Virtual Sensor. At this point the training of the ML model starts. After this, the component subscribes to *StreamObservations* of the contributing sensors. The predictions of the Virtual Sensor are done in a fixed update interval (configurable via the **updateInterval** parameter). If not all contributing sensors have published a new *StreamObservation* until then, the last known value is used for the prediction.

**GET /api/replaceBrokenSensor/**

with payload

```
{
  "sensorID" : <The ID of the broken sensor>,
  "maxDistance": <The maximum distance in meter to search for contributing sensors.␣
↪Integer, default 20000.>,
  "onlySameObservations": <Consider only sensors that measure the same observation as␣
↪the broken sensor. Boolean, default true.>,
  "limitSourceSensors": <Limit the number of contributing sensors. Interger, default␣
↪8.>,
  "updateInterval": <Number of seconds, in which the Virtual Sensor shall predict a␣
↪new value>
}
```

The data field in the response will contain the NGSI-LD description for the new Virtual Sensor.

**Please note:** although this API method is named *replace* BrokenSensor, the original sensor is not removed from the IoTCrawler system.

### Deploy a new Virtual Sensor

This method is not fully implemented!

Instead of replacing a broken sensor this component also allows to deploy a Virtual Sensor for locations, where no real sensor is present. For this the following API method can be used. It must include the parameter **location** where the new Virtual Sensor shall be deployed as well as the ID of an *ObservableProperty* (parameter **propertyID**) to indicate what physical phenomena shall be 'measured'. The rest of the optional parameters have the same meaning as for the replaceBrokenSensor method. The Virtual Sensor will use linear regression to predict new *StreamObservations*.

**GET /api/deployVirtualSensor/**

with payload

```
{
  "location" : [<latitude>, <longitude>, <altitude>],
  "propertyID": <The ID of an ObservableProperty the new sensor shall measure>
  "maxDistance": <The maximum distance in meter to search for contributing sensors.␣
↪Integer, default 20000.>,
  "onlySameObservations": <Consider only sensors that measure the same observation as␣
↪the broken sensor. Boolean, default true.>,
  "limitSourceSensors": <Limit the number of contributing sensors. Interger, default␣
↪8.>,
  "updateInterval": <Number of seconds, in which the Virtual Sensor shall predict a␣
↪new value>
}
```

### Stop a Virtual Sensor

This method can be used to stop an previously deployed Virtual Sensor, identified by its ID as returned by the deployment methode. If no Virtual Sensor with the given ID (parameter **sensorID**) exists this method returns an error. Otherwise the Virtual Sensor is stopped and the Virtual Sensor Creator component will delete the related entries in MDR, such as *Sensor*, *Platform* or *StreamObservations*. The un-deployment is triggered by a HTTP request to:

**`GET /api/stopVirtualSensor`**

with payload

```
{
  "sensorID" : <The ID of the Virtual Sensor to stop>
}
```

### List of Virtual Sensors deployed

To get a list of all deployed Virtual Sensors issue the following HTTP request:

**`GET /api/list`**

The response will be a JSON dictionary object with the IDs of the Virtual Sensors as key and the NGSI-LD description of the Virtual Sensor as value. For example:

```
{
    "status": "ok",
    "success": true,
    "description": "",
    "data": {
        "urn:ngsi-ld:Sensor:SolarPowerAarhus:492:currentProduction:VS": {
            "id": "urn:ngsi-ld:Sensor:SolarPowerAarhus:492:currentProduction:VS",
            "type": "http://www.w3.org/ns/sosa/Sensor",
            "http://www.w3.org/ns/sosa/isHostedBy": {
                "type": "Relationship",
                "object": "urn:ngsi-ld:Platform:SolarPowerAarhus:492:VS"
            },
            "http://www.w3.org/ns/sosa/madeObservation": {
                "type": "Relationship",
                "object": "urn:ngsi-
↪ld:StreamObservation:SolarPowerAarhus:492:currentProduction:VS"
            },
            "http://www.w3.org/ns/sosa/observes": {
                "type": "Relationship",
                "object": "urn:ngsi-
↪ld:ObservableProperty:SolarPowerAarhus:currentProduction"
            },
            "location": {
                "type": "GeoProperty",
                "value": {
                    "type": "Point",
                    "coordinates": [
                        56.253,
                        10.149
                    ]
                }
            },
            "https://w3id.org/iot/qoi#max": {
```

(continues on next page)

(continued from previous page)

```
                "type": "Property",
                "value": "NA"
            },
            "https://w3id.org/iot/qoi#min": {
                "type": "Property",
                "value": "NA"
            },
            "https://w3id.org/iot/qoi#updateinterval": {
                "type": "Property",
                "value": "NA",
                "https://w3id.org/iot/qoi#unit": {
                    "type": "Property",
                    "value": "NA"
                }
            },
            "https://w3id.org/iot/qoi#valuetype": {
                "type": "Property",
                "value": "NA"
            },
            "@context": [
                "http://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
                {}
            ]
        }
    }
}
```

### Build a Docker image

Change into the VirtualSensor directory and execute the script **build.sh**. This creates a Docker image tagged **vs_creator:1**.

### Run in Docker container

This repository includes a script to start the previously created image named **run.sh**. For the component to be able to interact with other IoTCrawler components, two environment variables need to be set. The first one is called BROKER_ADDRESS and contains the address of the MDR in the form <IP|DNS name>:<PORT> (e.g. "155.54.95.248:9090"). The second variable is named CALLBACK_ADDRESS and contains the address of the Virtual Sensor Creator component to allow the MDR component to notify this component about new *StreamObservations*. You can test if the component is running by calling http://localhost:8080/api/status .

### Sourcecode

The source code for the example can be found at https://github.com/IoTCrawler/

### Requirements

For a list of required Python modules see requirements.txt in the source repository.

CHAPTER

THREE

# GETTING STARTED

## 3.1 Prerequisites

Install the following tools before you star working with any tutorial of IoTCrawler.

### 3.1.1 Git

Download and install the latest version of git.

### 3.1.2 cURL

Download and install the latest version of cURL tool.

### 3.1.3 Docker and Docker Compose

Download and install the latest version of Docker and Docker Compose.

## 3.2 Start MetaData Repository

Download the Docker Compose file, go to the directory and run following command.

```
docker-compose up -f docker-compose-mdr.yml -d
```

This will start the docker containers for kafka, postgres, zookeeper and scorpio broker. Output of the command will be similar to following.

```
Creating network "mdr_default" with the default driver
Creating mdr_zookeeper_1 ... done
Creating mdr_postgres_1  ... done
Creating mdr_kafka_1     ... done
Creating mdr_scorpio_1   ... done
```

You can check the running containers with following command.

```
docker ps -a
```

If everything is running fine then you are ready to add your first entity. Use the following command.

```
curl -X POST 'http://localhost:9090/ngsi-ld/v1/entities/' --header 'Content-Type:␣
↪application/ld+json' -d '{
  "id": "house1:smartrooms:room1",
  "type": "Room",
  "temperature": {
        "value": 21,
        "unitCode": "CEL",
        "type": "Property",
        "providedBy": {
                "type": "Relationship",
                "object": "smartbuilding:house1:sensor001"
         }
   },
  "isPartOf": {
        "type": "Relationship",
        "object": "smartcity:houses:house1"
  },
  "@context": [{"Room": "urn:mytypes:room", "temperature": "myuniqueuri:temperature",
↪"isPartOf": "myuniqueuri:isPartOf"},"https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-
↪context.jsonld"]
}'
```

You can get the entity back with following request.

```
curl http://localhost:9090/ngsi-ld/v1/entities/house1:smartrooms:room1
```

You can remove the containers and clean your environment when you are finished.

```
docker-compose -f docker-compose-mdr.yml down
```

## 3.3 Start Orchestrator

Orchestrator works on top of Metadata Repository (handles subscription requests) or the Ranking Component(handles all types of GET requests). Make sure that at least one of these two component is already running.

Configuration of orchestrator and supplementary services is described in the docker-compose file. Clone the repository to have the file locally:

```
git clone https://github.com/IoTCrawler/Orchestrator
cd orchestrator
cd com.agtinternational.iotcrawler.orchestrator
```

Make sure that you've checked/adjusted environment variables according to the Documentation. To run orchestrator and other services execute the following command:

```
docker-compose up -f docker-compose.yml
```

This will start the docker containers for Orchestrator and RabbitMQ. Output of the command will be similar to following.

```
INFO [com.agtinternational.iotcrawler.orchestrator.Orchestrator] - <Initializing web␣
↪server>
DEBUG [com.agtinternational.iotcrawler.orchestrator.clients.NgsiLD_MdrClient] -
↪<Initializing NgsiLD_MdrClient to http://155.54.95.248:9090/ngsi-ld/. CurURIs=false>
INFO [com.agtinternational.iotcrawler.fiware.clients.NgsiLDClient] - <This product␣
↪includes software developed by NEC Europe Ltd>
```

(continues on next page)

(continued from previous page)

```
INFO [com.agtinternational.iotcrawler.orchestrator.Orchestrator] - <Initialized NGSI-
↪LD Client to http://155.54.95.248:9090>
INFO [com.agtinternational.iotcrawler.orchestrator.Orchestrator] - <Starting␣
↪orchestrator>
INFO [com.agtinternational.iotcrawler.orchestrator.Orchestrator] - <Syncing with␣
↪Redis at redis://redis>
INFO [com.agtinternational.iotcrawler.orchestrator.Orchestrator] - <Initializing web␣
↪server>
DEBUG [com.agtinternational.iotcrawler.core.clients.RabbitClient] - <Trying connect␣
↪to Rabbit at rabbit>
INFO [com.agtinternational.iotcrawler.orchestrator.HttpServer] - <Trying to init web␣
↪server for listening at 0.0.0.0:3001>
INFO [com.agtinternational.iotcrawler.orchestrator.HttpServer] - <Adding endpoint␣
↪http://0.0.0.0:3001/notify>
INFO [com.agtinternational.iotcrawler.orchestrator.HttpServer] - <Adding endpoint␣
↪http://0.0.0.0:3001/ngsi-ld>
INFO [com.agtinternational.iotcrawler.orchestrator.HttpServer] - <Adding endpoint␣
↪http://0.0.0.0:3001/>
INFO [com.agtinternational.iotcrawler.orchestrator.Orchestrator] - <Starting built-in␣
↪web server>
DEBUG [com.agtinternational.iotcrawler.core.clients.RabbitClient] - <Connection to␣
↪Rabbit rabbit established>
```

You can check the running containers with following command:

```
docker ps -a
```

The command should return you the following output:

```
d6843fcca0ae        gitlab.iotcrawler.net:4567/orchestrator/orchestrator/master
↪"java -server -cp /a..."
af9c84616883        rabbitmq:management
↪"docker-entrypoint.s..."
```

If everything is running fine then you should be able to see the version by executing the following command:

```
curl http://localhost:3001/version
```

You can stop the containers and clean your environment when you are finished.

```
docker-compose -f docker-compose.yml down
```

## 3.4 Start Search-Enabler

Search enabler works on top of NGSI-LD compatible component: Orchestrator, Ranking Component or Metadata Repository. Make sure that at least one of these two component is already running.

Configuration of search-enabler is stored in docker-compose file. Clone the repository to have the file locally:

```
git clone https://github.com/IoTCrawler/Search-Enabler
cd search-enabler
```

Please make sure that you've checked/adjusted environment variables according to Documentation. Execute the following command to run search-enabler:

```
docker-compose up -f docker-compose.yml
```

Output of a successfull start expected to be similar to following:

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/app/libs/logback-classic-1.2.3.jar!/org/slf4j/impl/
↪StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/app/libs/slf4j-log4j12-1.7.15.jar!/org/slf4j/impl/
↪StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [ch.qos.logback.classic.util.
↪ContextSelectorStaticBinder]

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.6.RELEASE)

INFO 1 --- [           main] c.a.i.graphqlEnabler.HttpApplication     : Starting␣
↪HttpApplication on e8fdbea680b9 with PID 1 (/app/classes started by root in /)
INFO 1 --- [           main] c.a.i.graphqlEnabler.HttpApplication     : No active␣
↪profile set, falling back to default profiles: default
INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat␣
↪initialized with port(s): 8080 (http)
INFO 1 --- [           main] o.apache.catalina.core.StandardService   : Starting␣
↪service [Tomcat]
INFO 1 --- [           main] org.apache.catalina.core.StandardEngine  : Starting␣
↪Servlet engine: [Apache Tomcat/9.0.21]
INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing␣
↪Spring embedded WebApplicationContext
INFO 1 --- [           main] o.s.web.context.ContextLoader            : Root␣
↪WebApplicationContext: initialization completed in 5160 ms
INFO 1 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing␣
↪ExecutorService 'applicationTaskExecutor'
INFO 1 --- [           main] o.s.b.a.w.s.WelcomePageHandlerMapping    : Adding␣
↪welcome page: class path resource [static/index.html]
INFO 1 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat␣
↪started on port(s): 8080 (http) with context path ''
INFO 1 --- [           main] c.a.i.graphqlEnabler.HttpApplication     : Started␣
↪HttpApplication in 9.806 seconds (JVM running for 11.245)
```

You can check the running containers with following command:

```
docker ps -a
```

The command should return you the following output:

```
gitlab.iotcrawler.net:4567/search-enabler/search-enabler/master   "java -server -cp /
↪a..."   15 hours ago        Up 1 hours         0.0.0.0:8080->8080/tcp  search-
↪enabler_search-enabler_1_4d46b42a3691
```

If everything is running fine then you should be able to see GraphiQL environment by opening the URL `http://localhost:8080` in any web browser.

You can stop the containers and clean your environment when you are finished.

```
docker-compose -f docker-compose.yml down
```

Alright, you have read basic information of the IoTCrawler components and have brief idea about the overall architecture. Now, you want to run an instance of IoTCrawler. Before, we get into the specifics of IoTCrawler components, it is advised to install all the *Prerequisites* installed on your machine.

Once you have installed the prerequisites, you are ready to *Start MetaData Repository*, which is one of the core components of the IoTCrawler.

CHAPTER

FOUR

# TUTORIALS

This section includes tutorials for the usage of the IoTCrawler framework.

## 4.1 Sensor Integration

In this section, we provide and example on how to include sensor data into the IoTCrawler framework. The sourcecode for this example can be accessed at https://github.com/IoTCrawler/iotcrawler-samples/tree/master/sensorintegration.

This example 'translates' CityProbe data available at https://www.opendata.dk/city-of-aarhus/bymiljo-aarhus-cityprobe into the IoTCrawler information model. The information model can be found at ...

### 4.1.1 Example data (CityProbe)

Let's have a look at the data provided by a CityProbe sensor in Aarhus. It can be accessed with: https://admin.opendata.dk/api/3/action/datastore_search?offset=792392&resource_id=7e85ea85-3bde-4dbf-944b-0360c6c47e3b&limit=1. For demonstration we have set the limit to 1.


```
{
  "help": "https://admin.opendata.dk/api/3/action/help_show?name=datastore_search",
  "success": true,
  "result": {
    "include_total": true,
    "resource_id": "7e85ea85-3bde-4dbf-944b-0360c6c47e3b",
    "fields": [
      {
        "type": "int",
        "id": "_id"
      },
      {
        "info": {
          "notes": "St\u00f8j i et JSON-objekt med hhv. gennemsnit, minimum og␣
↪maksimumsv\u00e6rdier\r\n* Enhed: dB SPL",
          "type_override": "",
          "label": "St\u00f8j"
        },
        "type": "json",
        "id": "noise"
      },
      {
        "info": {
```


(continues on next page)

(continued from previous page)


```
          "notes": "Kulilte/carbonmonoxid m\u00e5lt i et 12-bit interval (0 - 4095) p\
↪u00e5 en 3.3V linje. \r\nOpl\u00f8sningen er 0.8 mV per enhed. Modstanden␣
↪formindskes ved tilstedev\u00e6relsen af CO og carbonhydrider.\r\n",
          "type_override": "",
          "label": "Kulilte"
        },
        "type": "int4",
        "id": "CO"
      },
      {
        "info": {
          "notes": "* Enhed: Celcius\r\n* Pr\u00e6cision: \u00b10.5\u00b0C\r\n",
          "type_override": "",
          "label": "Temperatur"
        },
        "type": "float8",
        "id": "temperature"
      },
      {
        "info": {
          "notes": "* Enhed: \u03bcg/m3",
          "type_override": "",
          "label": "Partikelforurening (10 \u00b5m)"
        },
        "type": "int4",
        "id": "PM10"
      },
      {
        "info": {
          "notes": "Batterikapacitet i procent\r\nEnheden oplades ved hj\u00e6lp af␣
↪et solpanel",
          "type_override": "",
          "label": "Batterikapacitet"
        },
        "type": "float8",
        "id": "battery"
      },
      {
        "info": {
          "notes": "Regnintensitet i et JSON-objekt med hhv. gennemsnit, minimum og␣
↪maksimumsv\u00e6rdier i dB SPL.\r\nM\u00e5lemetoden er en mikrofon under en␣
↪polycarbonath\u00e6tte, som m\u00e5ler peak amplitude og frekvensen af dr\u00e5ber,␣
↪som rammer toppen. P\u00e5 Open Data DK er v\u00e6rdierne r\u00e5 og ikke␣
↪analyseret.",
          "type_override": "",
          "label": "Regnintensitet"
        },
        "type": "json",
        "id": "rain"
      },
      {
        "info": {
          "notes": "Enhed: %RH\r\nPr\u00e6cision: \u00b13 %RH\r\n",
          "type_override": "",
          "label": "Luftfugtighed"
        },
        "type": "float8",
```


(continues on next page)

(continued from previous page)


```
        "id": "humidity"
      },
      {
        "info": {
          "notes": "Lysintensitet m\u00e5lt i lux",
          "type_override": "",
          "label": "Dagslys"
        },
        "type": "int4",
        "id": "illuminance"
      },
      {
        "info": {
          "notes": "Lufttryk m\u00e5lt i hPa\r\nPr\u00e6cision: \u00b11.0 hPa",
          "type_override": "",
          "label": "Lufttryk"
        },
        "type": "float8",
        "id": "pressure"
      },
      {
        "info": {
          "notes": "Tidsstempel for m\u00e5lingen i UTC og ISO 8601-format",
          "type_override": "",
          "label": "Tidsstempel"
        },
        "type": "text",
        "id": "published_at"
      },
      {
        "info": {
          "notes": "Enhed: \u03bcg/m3",
          "type_override": "",
          "label": "Partikelforurening (2.5 \u00b5m)"
        },
        "type": "int4",
        "id": "PM2.5"
      },
      {
        "info": {
          "notes": "ID for den p\u00e5g\u00e6ldende enhed, som har foretaget m\
↪u00e5lingen.",
          "type_override": "",
          "label": "Enhedsid"
        },
        "type": "text",
        "id": "deviceid"
      },
      {
        "info": {
          "notes": "Kv\u00e6lstofdioxid m\u00e5lt i et 12-bit interval (0 - 4095) p\
↪u00e5 en 3.3V linje. \r\nOpl\u00f8sningen er 0.8 mV per enhed. Modstanden for\
↪u00f8ges ved tilstedev\u00e6relsen af CO og carbonhydrider.",
          "type_override": "",
          "label": "Kv\u00e6lstofdioxid"
        },
        "type": "int4",
```


(continues on next page)

(continued from previous page)


```
        "id": "NO2"
      },
      {
        "type": "int4",
        "id": "firmware_version"
      },
      {
        "type": "text",
        "id": "device_id"
      }
    ],
    "records_format": "objects",
    "records": [
      {
        "_id": 794720,
        "noise": "{\"max\": \"67.46\", \"average\": \"61.57\", \"min\": \"56.86\"}",
        "CO": 825,
        "temperature": 0,
        "PM10": 0,
        "battery": 89.59,
        "rain": "{\"max\": \"492\", \"average\": \"215.05\", \"min\": \"96\"}",
        "humidity": 0,
        "illuminance": 49595,
        "pressure": 0,
        "published_at": "2020-06-17T10:20:35.255Z",
        "PM2.5": 0,
        "deviceid": "20004c000d50483553343720",
        "NO2": 889,
        "firmware_version": 49,
        "device_id": null
      }
    ],
    "limit": 1,
    "offset": 792392,
    "_links": {
      "start": "/api/3/action/datastore_search?limit=1&resource_id=7e85ea85-3bde-4dbf-
↪944b-0360c6c47e3b",
      "prev": "/api/3/action/datastore_search?offset=792391&limit=1&resource_
↪id=7e85ea85-3bde-4dbf-944b-0360c6c47e3b",
      "next": "/api/3/action/datastore_search?offset=792393&limit=1&resource_
↪id=7e85ea85-3bde-4dbf-944b-0360c6c47e3b"
    },
    "total": 792920
  }
}
```


The provided data contains some meta information within the fields list. The data itself is contained in the records list. In our case there is just one entry because of the limit set before. So let us just have a look at the "data":


```
{
    "_id": 794720,
    "noise": "{\"max\": \"67.46\", \"average\": \"61.57\", \"min\": \"56.86\"}",
    "CO": 825,
    "temperature": 0,
    "PM10": 0,
    "battery": 89.59,
```


(continues on next page)

(continued from previous page)


```
    "rain": "{\"max\": \"492\", \"average\": \"215.05\", \"min\": \"96\"}",
    "humidity": 0,
    "illuminance": 49595,
    "pressure": 0,
    "published_at": "2020-06-17T10:20:35.255Z",
    "PM2.5": 0,
    "deviceid": "20004c000d50483553343720",
    "NO2": 889,
    "firmware_version": 49,
    "device_id": null
}
```


As can be seen the data contains *noise*, *CO*, *temperature*, *PM10*, *battery*, *rain*, *humidity*, *illuminance*, *pressure*, *PM2.5*, and *NO2* as data fields. Besides that, *deviceid* contains the relation to the measuring sensor and *published_at* contains a timestamp of the measurement. The other fields are not used in this example.

### 4.1.2 Translation into IoTCrawler model

In the IoTCrawler information model (cf. ...) we have to split the provided information from the CityProbe dataset. In our example we will have:

- 1 Platform hosting 11 sensors
- 11 Sensors (one for each datafield)
- 11 IoTStreams (one for each Sensor)
- 11 ObservableProperties (one for each Sensor)
- 11 StreamObservations (one for each Sensor)

With the help of the provided script these entities are created. For details please have a look at https://github.com/IoTCrawler/iotcrawler-samples/blob/master/sensorintegration/main.py

### 4.1.3 Connection to MDR/Broker

The translated data is stored into the MDR by using the standardised NGSI-LD API (see NGSI-LD API). In the example script, this is done in a threaded way to avoid blocking. The broker can return several HTTP status codes as feedback, while accessing its interface at:

**`POST /ngsi-ld/v1/entities/`**

**Status Codes**

- 201 Created – entity was successfully created
- 400 Bad Request – bad request, the entity is probably not in ngsi-ld format
- 409 Conflict – the entity already exists
- 500 Internal Server Error – internal server error

In case of a 409 status code we have to PATCH the entity as it is already existing. The interface will change to

**`POST /ngsi-ld/v1/entities/ENTITYID/attrs/`**

where *ENTITYID* has to be replaced by the entity that should be updated. Additionally, the *id* and *type* has to be deleted from the provided entity in NGSI-LD format. Within the script this is done automatically.

### 4.1.4 Sourcecode

The sourcecode for the example can be found at https://github.com/IoTCrawler/iotcrawler-samples/tree/master/sensorintegration

## 4.2 GraphQL Tutorial

**Table of contents**



### 4.2.1 Introduction

GraphQL-based search enabler provides a flexible capabilities for searching metadata about Streams, Sensors and Observable Properties registered in IoTCrawler Metadata Repository (MDR).

In this tutorial we demonstrating a use-case of finding streams delivering a certain type of measurements.

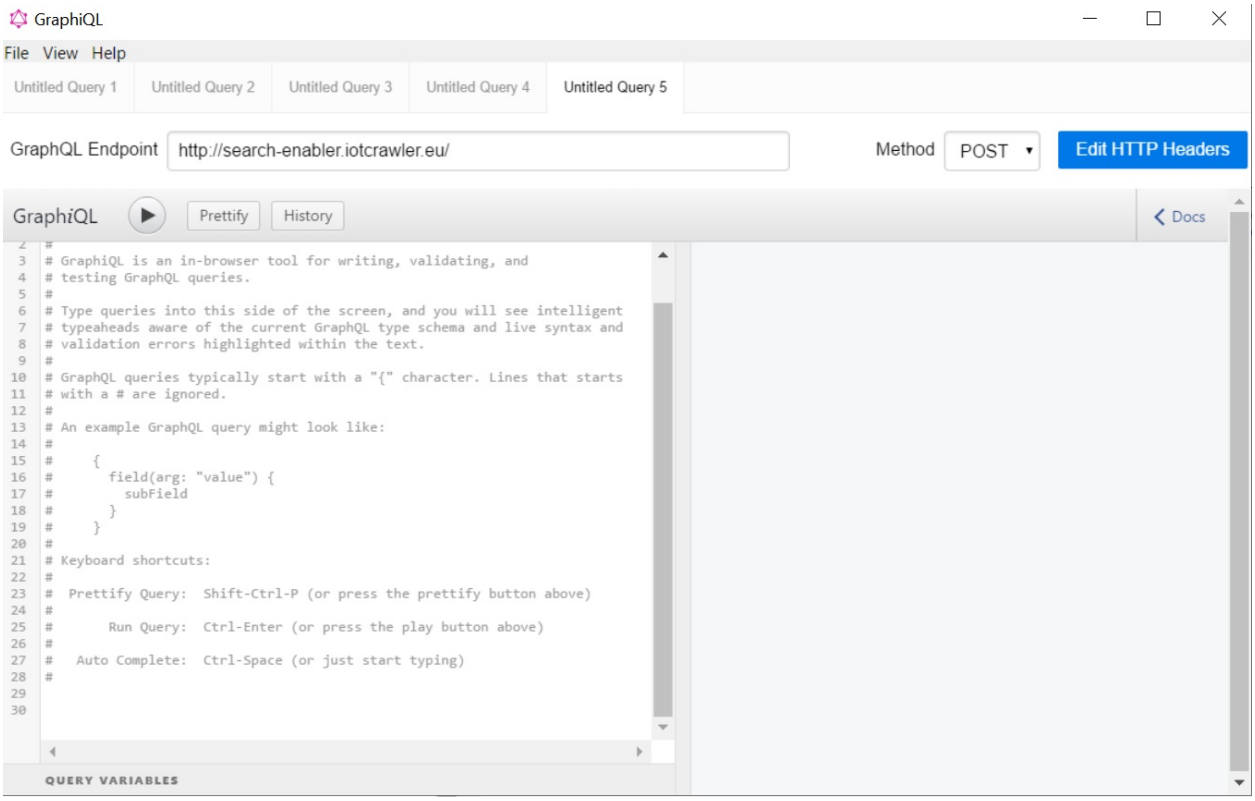
All the demonstrated queries can be executed in the online GraphiQL Environment using the supplied links.

### 4.2.2 GraphiQL

The GraphiQL is a simple IDE for constructing and testing queries against a GraphQL endpoint. GraphiQL might be provided by endpoint provider (like it is provided by IoTCrawler ) or user might run it locally using the desktop client.

GraphiQL provides interactive user assistance functions such as query validation, auto completion (Ctrl+Space), undo/redo, formatting, history and many others.

Then the queries are ready they can be integrated into your application and submitted directly to the GraphQL endpoint without using GraphiQL.




User assistance in GraphiQL is based on a GraphQL schema of a target endpoint. The schema describes the structure of a query language of target endpoint (data types, interfaces, queries, mutations).

To introspect the GraphQL schema of a particular endpoint execute the following query:

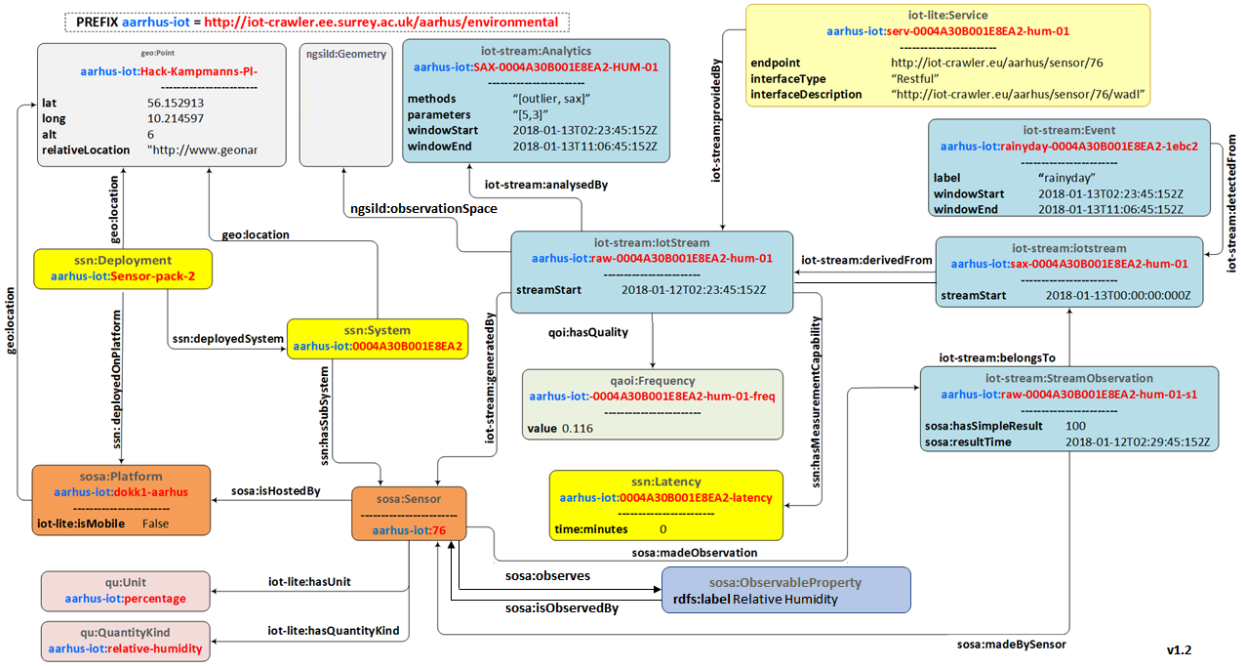
```
{
  __schema {
    types {
      name
      fields {
        name
        type {
          name
        }
      }
    }
  }
}
```

Run online

### 4.2.3 IoTCrawler schema

The search enabler component has an extendable schemas mechanism described in further sections.

The IoTCrawler's Core Schema defines core types (such as IoTStream, Sensor, Platform, Observable Property).




Due to support of type inheritance (feature of IoTCrawler's GraphQL-based Search Enabler), the schema is compliant with SOSA ontology.

For example, according to SOSA a platform (*SOSA:Platform*) hosts systems (*SSN:System*), which includes sensors (*SOSA:Sensor*), actuators (*SOSA:Actuator*) and samples (*SOSA:Sample*).

### 4.2.4 Browsing platforms

For demonstrating inheritance support lets browse over the platforms and objects they host. This can be done by performing the following query:

```
{
  platforms(offset: 0, limit: 5) {
    id
    hosts {
      id
      type
    }
  }
}
```

Run online

The result will look as below.

```
"platforms": [
   {
     "id": "urn:ngsi-ld:Platform:B4:E6:2D:8A:20:DD",
```

(continues on next page)

(continued from previous page)

```
    "hosts": [
      {
        "id": "urn:ngsi-ld:Sensor:B4:E6:2D:8A:20:DD:IAQ",
        "type": "http://www.w3.org/ns/sosa/Sensor"
      },
      {
        "id": "urn:ngsi-ld:Sensor:B4:E6:2D:8A:20:DD:Temperature",
        "type": "http://www.w3.org/ns/sosa/Sensor"
      },
      {
        "id": "urn:ngsi-ld:Sensor:B4:E6:2D:8A:20:DD:Humidity",
        "type": "http://www.w3.org/ns/sosa/Sensor"
      }
    ]
  }
]
```

As you can see, resulting platforms host sensors, while schema declares, that *Platform* hosts *System*. This is possible due to type inheritance feature described above.

### 4.2.5 Browsing observable properties

Let's return back to our use-case, where we want to find streams matching the criteria and get their measurements.

Before searching streams by a certain observable property let's have a quick look at observable properties registered in the IoTCrawler platform. Let can query all the observable properties as paginated list by using the *limit* and *offset* variables. The default limit is 500 (maximal of broker).

```
{
  observableProperties(offset: 0, limit: 5) {
    id
    label
  }
}
```

Run online

Response should look like below, where we see identifiers and labels of observable properties registered in the IoTCrawler metadata repository.

```
{
  "data": {
    "observableProperties": [
      {
        "id": "urn:ngsi-ld:ObservableProperty:B4:E6:2D:8A:20:DD:Temperature",
        "label": "temperature"
      },
      {
        "id": "urn:ngsi-ld:ObservableProperty:B4:E6:2D:8A:20:DD:Humidity",
        "label": "humidity"
      },
      {
        "id": "urn:ngsi-ld:ObservableProperty:B4:E6:2D:8A:20:DD:IAQ",
        "label": "iaq"
      },
      {
```

(continues on next page)

(continued from previous page)

```
        "id": "urn:ngsi-ld:ObservableProperty:SolarPowerAarhus:currentProduction",
        "label": "currentproduction"
      },
      {
        "id": "urn:ngsi-ld:ObservableProperty:AvailableParkingSpaces",
        "label": "available parking spaces"
      }
    ]
  }
  ...
}
```

### 4.2.6 Searching sensors

Imagine that before dealing with the stream, we are interested in details about its sensor and the platform, which provide that data into IoTCrawler. In order to do so perform the following query:

```
{
  sensors(observes: {label: "temperature"}) {
    id
    label
    isHostedBy {
      id
    }
  }
}
```

Run online

Response should look like below:

```
{
  "data": {
    "sensors": [
      {
        "id": "urn:ngsi-ld:Sensor:B4:E6:2D:8A:20:DD:Temperature",
        "label": null,
        "isHostedBy": {
          "id": "urn:ngsi-ld:Platform:B4:E6:2D:8A:20:DD",
          "label": null
        }
      }
    ]
  },
  ..
}
```

### 4.2.7 Searching streams

Finally we need to get streams, which are matching our criteria. For doing this we can filter streams by sensor id (which we've found on a previous step) or by specifying our condition (observes temperature) directly for a filter in the streams query:

```
query streams {
  streams(generatedBy: {observes: {label: "temperature"}}) {
    id
    generatedBy {
      id
      label
      isHostedBy {
        id
        label
      }
    }
  }
}
```

Run online

The results would reflect the details about streams (*id*), their sensors (under *id* and *label* in the *generatedBy* block) and platforms (*id* and *label* in the *isHostedBy* block).

```
{
  "data": {
    "streams": [
      {
        "id": "urn:ngsi-ld:IotStream:B4:E6:2D:8A:20:DD:Temperature",
        "generatedBy": {
          "id": "urn:ngsi-ld:Sensor:B4:E6:2D:8A:20:DD:Temperature",
          "label": null,
          "isHostedBy": {
            "id": "urn:ngsi-ld:Platform:B4:E6:2D:8A:20:DD",
            "label": null
          }
        }
      }
    ]
  }
  ..
}
```

The example demonstrates that it is possible to filter the target object together with all the relevant information (stream, sensor, platform, observable property) in one GraphQL query.

### 4.2.8 Searching stream observations

Stream observations are not considered as metadata and not expected to be stored in IoTCrawler's metadata repository, but expected to be delivered by broker's federation mechanism.

Subscription is the most expected way of receiving stream observations. But there is still an opportunity to request the actual state of a certain stream observation.

Perform the following query to return stream observations of streams, we have been interested in previous examples:

```
{
  streamObservations(belongsTo: {generatedBy: {observes: {label: "temperature"}}}) {
    id
    resultTime
    hasSimpleResult
    belongsTo {
      id
      generatedBy {
        id
        observes {
          id
          label
        }
      }
    }
  }
}
```

Run online

The query is expected to return the following results:

```
{
  "data": {
    "streamObservations": [
      {
        "id": "urn:ngsi-ld:StreamObservation:B4:E6:2D:8A:20:DD:Temperature",
        "resultTime": "2020-07-07T13:18:37Z",
        "hasSimpleResult": 29.09628,
        "belongsTo": {
          "id": "urn:ngsi-ld:IotStream:B4:E6:2D:8A:20:DD:Temperature",
          "generatedBy": {
            "id": "urn:ngsi-ld:Sensor:B4:E6:2D:8A:20:DD:Temperature",
            "observes": {
              "id": "urn:ngsi-ld:ObservableProperty:B4:E6:2D:8A:20:DD:Temperature",
              "label": "temperature"
            }
          }
        }
      }
    ]
  }
}
```

### 4.2.9 Extendable schemas

As was mentioned before, the search enabler supports extendable schemas and allows application owners to register/store custom data models in MDR and expose them via GraphQL queries.

Due to type inheritance it is possible to create custom (e.g. more specific) data types, which will be still be reachable via queries of core IoTCrawler types.

Let's create a couple of specific sensor types: the *Temperature Sensor* and *Indoor Temperature Sensor*. For doing so we need to create a separate GraphQL schema with the following definitions:

```
schema {
    query: Query

}

type Query {
    temperatureSensors(isHostedBy: PlatformInput, observes: ObservablePropertyInput,␣
↪offset: Int = 0, limit: Int = 0): [TemperatureSensor]
    indoorTemperatureSensors(isHostedBy: PlatformInput, observes:␣
↪ObservablePropertyInput, offset: Int = 0, limit: Int = 0): [IndoorTemperatureSensor]
}

type TemperatureSensor @resource(class : "http://purl.org/iot/ontology/extended-iot-
↪stream#TemperatureSensor", subClassOf: ["Sensor"]){

}

type IndoorTemperatureSensor @resource(class : "http://purl.org/iot/ontology/extended-
↪iot-stream#IndoorTemperatureSensor", subClassOf: ["TemperatureSensor"]){

}
```

As you can see, the schema introduces two additional data types (but not extending them with additional fields). Due to two new queries new sensors can be queried in GraphiQL. The schema reuses types (e.g. Sensor, PlatformInput, etc.) from the other schemas (the core schema in this case).

Let's first query indoor temperature sensors as the most specific data type:

```
{
  indoorTemperatureSensors {
    id
    type
    alternativeType
    label
    observes {
      id
      label
    }
    isHostedBy {
      id
      label
      hosts {
        id
        label
      }
    }
  }
}
```

Run online

The result will reflect the temperature sensor we've registered with the following code:

```
{
  "data": {
    "indoorTemperatureSensors": [
      {
        "id": "urn:ngsi-ld:IndoorTemperatureSensor_1",
        "type": "http://www.w3.org/ns/sosa/Sensor",
        "alternativeType": "http://purl.org/iot/ontology/extended-iot-stream
↪#TemperatureSensor",
        "label": null,
        "observes": null,
        "isHostedBy": {
          "id": "urn:ngsi-ld:Platform_homee_1",
          "label": "Platform homee_1",
          "hosts": [
            {
              "id": "urn:ngsi-ld:IndoorTemperatureSensor_1",
              "label": null
            },
            {
              "id": "urn:ngsi-ld:TemperatureSensor_1",
              "label": null
            }
          ]
        }
      }
    ]
  }
}
```

You can see that entity still has type *http://www.w3.org/ns/sosa/Sensor* and the *http://purl.org/iot/ontology/extended-iot-stream#TemperatureSensor* is declared as *alternativeType*. This makes the indoor temperature sensors searchable while "sensors()" queries.

Now let's query temperature sensors, which should include indoor temperature sensors as well:

```
{
  temperatureSensors {
    id
    type
    alternativeType
    label
    observes {
      id
      label
    }
    isHostedBy {
      id
      label
      hosts {
        id
        label
      }
    }
  }
}
```

Run online

The result should return back at least two sensors: one of type *temperature sensor* and one of type *indoor temperature sensor*.


```
{
  "data": {
    "temperatureSensors": [
      {
        "id": "urn:ngsi-ld:TemperatureSensor_1",
        "type": "http://www.w3.org/ns/sosa/Sensor",
        "alternativeType": "http://purl.org/iot/ontology/extended-iot-stream
↪#TemperatureSensor",
        "label": null,
        "observes": null,
        "isHostedBy": {
          "id": "urn:ngsi-ld:Platform_homee_1",
          "label": "Platform homee_1",
          "hosts": [
            {
              "id": "urn:ngsi-ld:IndoorTemperatureSensor_1",
              "label": null
            },
            {
              "id": "urn:ngsi-ld:TemperatureSensor_1",
              "label": null
            }
          ]
        }
      },
      {
        "id": "urn:ngsi-ld:IndoorTemperatureSensor_1",
        "type": "http://www.w3.org/ns/sosa/Sensor",
        "alternativeType": "http://purl.org/iot/ontology/extended-iot-stream
↪#IndoorTemperatureSensor",
        "label": null,
        "observes": null,
        "isHostedBy": {
          "id": "urn:ngsi-ld:Platform_homee_1",
          "label": "Platform homee_1",
          "hosts": [
            {
              "id": "urn:ngsi-ld:IndoorTemperatureSensor_1",
              "label": null
            },
            {
              "id": "urn:ngsi-ld:TemperatureSensor_1",
              "label": null
            }
          ]
        }
      }
    ]
  }
}
```


And again you can see, that both of them have type *sosa:Sensor* and their real types are declared as *alternativeType*.

The alternative type is an optional filed and is interpreted by the search-enabler only.

The actual list of schemas created to Search Enabler can be found in the repository. The list is extendable by new domain-specific schemas provided by application owners.

### 4.2.10 References

Search Enabler Source (Github)

List of schemas

Detailed description of Search Enabler (Project Deliverable)

CHAPTER

FIVE

# ARCHITECTURE REFERENCE

CHAPTER

SIX

# REST API

This section defines all REST APIs supported by all components of IoTCrawler.

## 6.1 MetaData REST API

This section shows the basic entity requests used in the REST API of the MetaData Repository. For a detail explanation, please look the ETSI spec (https://www.etsi.org/deliver/etsi_gs/CIM/001_099/004/01.01.02_60/gs_cim004v010102p.pdf).

**Table of contents**



### 6.1.1 Add entity

**POST /ngsi-ld/v1/entities/**

Add a new entity.

**Request Headers**

- Content-Type – application/ld+json

**Example request**:

Bash

```
$ curl --location --request POST 'http://{{broker-host}}/ngsi-ld/v1/entities/' \
       --header 'Content-Type: application/ld+json' \
       --data-raw '{
         "@context":[
               "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
               {
                   "Vehicle": "http://example.org/vehicle/Vehicle",
                   "brandName": "http://example.org/vehicle/brandName",
```

(continues on next page)

(continued from previous page)

```
                    "speed": "http://example.org/vehicle/speed"
                }
        ],
        "id":"urn:ngsi-ld:Vehicle:TEST1",
        "type":"Vehicle",
        "brandName":{
          "type":"Property",
          "value":"Mercedes"
        },
        "speed":{
          "type":"Property",
          "value":80
        },
        "location": {
              "type": "GeoProperty",
              "value": { "type": "Point", "coordinates": [ -1.1336517, 37.
↪9894006 ] }
        }
      }'
```

**Status Codes**

- 201 Created – Created

### 6.1.2 Obtain entities by type

**GET /ngsi-ld/v1/entities**

Retrieve a list of all the entities.

**Query Parameters**

- **type** (*string*) – entity type code as `http://example.org/vehicle/Vehicle`, `http://www.w3.org/2003/01/geo/wgs84_pos%23Point`, `indexing`, etc.

**Example request**:

Bash

Python

JavaScript

```
$ curl --location --request GET 'http://{{broker-host}}/ngsi-ld/v1/entities/?
↪type=http://example.org/vehicle/Vehicle'
```

```
import requests
url = "http://metadata-repository-scorpiobroker.35.241.228.250.nip.io/ngsi-ld/v1/
↪entities/?type=indexing"
payload = {}
headers= {}
response = requests.request("GET", url, headers=headers, data = payload)
print(response.text.encode('utf8'))
```

```
var request = require('request');
var options = {
  'method': 'GET',
```

(continues on next page)

(continued from previous page)

```
  'url': 'http://metadata-repository-scorpiobroker.35.241.228.250.nip.io/ngsi-ld/
↪v1/entities/?type=indexing',
  'headers': {
  }
};
request(options, function (error, response) {
  if (error) throw new Error(error);
  console.log(response.body);
});
```

**Example response**:

```
[
  {
    "id": "urn:ngsi-ld:Vehicle:TEST1",
    "type": "http://example.org/vehicle/Vehicle",
    "http://example.org/vehicle/brandName": {
      "type": "Property",
      "value": "Mercedes"
    },
    "http://example.org/vehicle/speed": {
      "type": "Property",
      "value": 80
    },
    "location": {
      "type": "GeoProperty",
      "value": {
        "type": "Point",
        "coordinates": [
          -1.1336517,
          37.9894006
        ]
      }
    },
    "@context": [
      "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
    ]
  }
]
```

**Response Headers**

- Content-Type – application/ld+json

**Status Codes**

- 200 OK – no error

### 6.1.3 Obtain entity by id

**GET /ngsi-ld/v1/entities/**(**str:** *get_id*)
Retrieve an entity by identifier.

**Parameters**

- **get_id** (*str*) – get's unique id

**Example request**:

Bash

```
$ curl --location --request GET 'http://{{broker-host}}/ngsi-ld/v1/entities/
↪urn:ngsi-ld:Vehicle:TEST1'
```

**Example response**:

```
{
  "id": "urn:ngsi-ld:Vehicle:TEST1",
  "type": "http://example.org/vehicle/Vehicle",
  "http://example.org/vehicle/brandName": {
    "type": "Property",
    "value": "Mercedes"
  },
  "http://example.org/vehicle/speed": {
    "type": "Property",
    "value": 80
  },
  "location": {
    "type": "GeoProperty",
    "value": {
      "type": "Point",
      "coordinates": [
        -1.1336517,
        37.9894006
      ]
    }
  },
  "@context": [
    "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
  ]
}
```

**Response Headers**

- Content-Type – application/ld+json

**Status Codes**

- 200 OK – no error
- 404 Not Found – not found

### 6.1.4 Update entity

**PATCH /ngsi-ld/v1/entities/**(**str:** *patch*)**/attrs**

Update entity.

**Parameters**

- **patch** (*str*) – patch's unique id

**Request Headers**

- Content-Type – application/ld+json

**Example request**:

Bash

```
$ curl --location --request PATCH 'http://{{broker-host}}/ngsi-ld/v1/entities/
↪urn:ngsi-ld:Vehicle:TEST1/attrs' \
     --header 'Content-Type: application/ld+json' \
     --data-raw '{
         "@context":[
             "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld",
             {
                 "Vehicle": "http://example.org/vehicle/Vehicle",
                 "brandName": "http://example.org/vehicle/brandName",
                 "speed": "http://example.org/vehicle/speed"
             }
         ],
        "brandName":{
             "type":"Property",
             "value":"Seat"
         },
         "speed": {
             "type": "Property",
             "value": 5
         }

     }'
```

**Status Codes**

- 204 No Content – No content, no error
- 404 Not Found – not found

### 6.1.5 Delete entity

**DELETE /ngsi-ld/v1/entities/**(**str:** *delete_id*)

Remove an entity by identifier.

**Parameters**

- **delete_id** (*str*) – delete's unique id

**Example request**:

Bash

```
$ curl --location --request DELETE 'http://{{broker-host}}/ngsi-ld/v1/entities/
↪urn:ngsi-ld:Vehicle:TEST1'
```

**Status Codes**

- 204 No Content – No content, no error
- 404 Not Found – not found

## 6.2 IdM-Keyrock REST API

This section defines specific IdM-Keyrock requests required in IotCrawler environment. Further information can be found in the Keyrock Apiary (https://keyrock.docs.apiary.io).

**Table of contents**



### 6.2.1 Generate/Obtain a new IdM Token

**POST /v1/auth/tokens**

Generate/Obtain a new IdM Token.

**Request Headers**

- Content-Type – application/json

**Request JSON Object**

- **name** (*string*) – IdM-Keyrock user. For example: "admin@test.com"
- **password** (*string*) – IdM-Keyrock user password. For example: "1234"

**Example request**:

Bash

```
$ curl --location --request POST 'https://{{IdM-IP}}:{{IdM-Port}}/v1/auth/tokens'␣
↪\
  --header 'Content-Type: application/json' \
  --data-raw '{
    "name": "admin@test.com",
    "password": "1234"
  }'
```

**Example response**:

```
{
    "token": {
        "methods": [
            "password"
        ],
        "expires_at": "2020-07-21T10:02:19.256Z"
    },
    "idm_authorization_config": {
        "level": "advanced",
        "authzforce": true
```

(continues on next page)

(continued from previous page)

```
    }
}
```

**Response Headers**

- *X-Subject-Token* – IdM-Keyrock token. For example: "cf41496e-d9b2-4f1f-8cba-f4efb3bf9abe"
- Content-Type – application/json

**Status Codes**

- 201 Created – Created
- 401 Unauthorized – Unauthorized

### 6.2.2 Obtain the IdM Token information

**`GET /v1/auth/tokens`**

Obtain the IdM Token information.

**Example request**:

Bash

```
$ curl --location --request GET 'https://{{IdM-IP}}:{{IdM-Port}}/v1/auth/tokens' \
       --header 'X-Auth-token: cf41496e-d9b2-4f1f-8cba-f4efb3bf9abe' \
       --header 'X-Subject-token: cf41496e-d9b2-4f1f-8cba-f4efb3bf9abe'
```

**Example response**: .. sourcecode:: json

{ "access_token": "cf41496e-d9b2-4f1f-8cba-f4efb3bf9abe", "expires": "2020-07-21T10:47:23.000Z", "valid": true, "User": {

"scope": [], "id": "admin", "username": "admin", "email": "admin@test.com", "date_password": "2019-10-22T12:45:58.000Z", "enabled": true, "admin": true

}

}

**Response Headers**

- Content-Type – application/json

**Status Codes**

- 200 OK – OK.
- 401 Unauthorized – Unauthorized

## 6.3 XACML - PDP REST API

This section defines all REST APIs supported by XACML - PDP.

**Table of contents**



### 6.3.1 Obtain Verdict

**POST /XACMLServletPDP/**

Obtain Verdict.

**Request Headers**

- Content-Type – text/plain

**Form Parameters**

- **subject** – subject of the resource's request. In DCapBAC scenario, it could correspond with a username (IDM). For example: "Peter"
- **resource** – endpoint + path of the resource's request (protocol+IP+PORT+path). For example: "https://153.55.55.120:2354/ngsi-ld/v1/entities/urn:ngsi-ld:Sensor:humidity.201". In DCapBAC scenario, endpoint corresponds with the PEP-Proxy one.
- **action** – method of the resource's request ("POST", "GET", "PATCH"...)

**Example request**:

Bash

```
$  curl --location --request POST 'http://{{XACML-PDP-IP}}:{{{XACML-PDP-Port}}/
↪XACMLServletPDP/' \
   --header 'Content-Type: text/plain' \
   --data-raw '<Request xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os">
      <Subject SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-
↪category:access-subject">
         <Attribute AttributeId="urn:oasis:names:tc:xacml:2.0:subject:role"␣
↪DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>Peter</AttributeValue>
         </Attribute>
      </Subject>

      <Resource>
         <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-
↪id" DataType="http://www.w3.org/2001/XMLSchema#string">
            <AttributeValue>https://153.55.55.120:2354/ngsi-ld/v1/entities/
↪urn:ngsi-ld:Sensor:humidity.201</AttributeValue>
         </Attribute>
      </Resource>

      <Action>
         <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"␣
↪DataType="http://www.w3.org/2001/XMLSchema#string">
```

(continues on next page)

(continued from previous page)

```
            <AttributeValue>GET</AttributeValue>
          </Attribute>
      </Action>

      <Environment/>
   </Request>'
```

**Example response**:

```
<Response>
  <Result ResourceID="https://153.55.55.120:2354/ngsi-ld/v1/entities/urn:ngsi-
↪ld:Sensor:humidity.201">
    <Decision>Permit</Decision>
    <Status>
      <StatusCode Value="urn:oasis:names:tc:xacml:1.0:status:ok"/>
    </Status>
    <Obligations>
      <Obligation ObligationId="liveTime" FulfillOn="Permit">
      </Obligation>
    </Obligations>
  </Result>
</Response>
```

**Status Codes**

- 200 OK – no error - Permit, NotApplicable or Deny.

### 6.3.2 Test XACML - PDP is running

**GET /**

Test XACML - PDP is running.

**Example request**:

Bash

```
$ curl --location --request GET 'https://{{XACML-PDP-IP}}:{{{XACML-PDP-Port}}/
↪XACMLServletPDP'
```

**Example response**:

**Status Codes**

- 200 OK – Component is running.

## 6.4 Capability Manager REST API

This section defines all REST APIs supported by Capability Manager.

**Table of contents**

### 6.4.1 Obtain Capability Token

**POST /**

Obtain Capability Token.

**Request Headers**

- Content-Type – application/json

**Request JSON Object**

- **token** (*string*) – subject of the resource's request. In DCapBAC scenario, it could correspond with a token (IDM-KeyRock). For example: "753f103c-d8e5-4f4e-8720-13d5e2f55043"
- **de** (*string*) – endpoint of the resource's request (protocol+IP+PORT). In DCapBAC scenario, it corresponds with PEP-Proxy component. For example: "https://153.55.55.120:2354"
- **ac** (*string*) – method of the resource's request ("POST", "GET", "PATCH"...)
- **re** (*string*) – path of the resource request. For example: "/ngsi-ld/v1/entities/urn:ngsi-ld:Sensor:humidity.201"

**Example request**:

Bash

```
$ curl --location --request POST 'https://{{CapMan-IP}}:{{CapMan-Port}}/' \
  --header 'Content-Type: application/json' \
  --data-raw '{"token": "753f103c-d8e5-4f4e-8720-13d5e2f55043", "de": "https://
↪153.55.55.120:2354", "ac": "GET", "re": "/ngsi-ld/v1/entities/urn:ngsi-
↪ld:Sensor:humidity.201" }'
```

**Example response**:

```
{
    "id": "nlqfnfa6nqrlbh9h7tigg28ga1",
    "ii": 1586166961,
    "is": "capabilitymanager@odins.es",
    "su": "Peter",
    "de": "https://153.55.55.120:2354",
    "si":
↪"MEUCIEEGwsTKGdlEeUxZv7jsh0UdWoFLud3uqpMDvnlT+GD7AiEAmwEu0FHuG+XyRi9BEAMaVPBIqRvOJlSIBkBT3K7LH
↪",
    "ar": [
      {
        "ac": "GET",
        "re": "/ngsi-ld/v1/entities/urn:ngsi-ld:Sensor:humidity.201"
      }
    ],
    "nb": 1586167961,
    "na": 1586177961
}
```

**Status Codes**

- 200 OK – no error
- 401 Unauthorized – Unauthorized
- 500 Internal Server Error – Can't generate capability token

### 6.4.2 Test Capability Manager is running

**GET /**
Test Capability Manager is running.

**Example request**:

Bash

```
$ curl --location --request GET 'https://{{CapMan-IP}}:{{CapMan-Port}}/'
```

**Status Codes**

- 200 OK – Component is running.

## 6.5 Security Facade REST API

This section defines all REST APIs supported by Security Facade.

**Table of contents**



### 6.5.1 Obtain Capability Token

**POST /CapabilityManagerServlet/IdemixTokenIdentity**
Obtain Capability Token.

**Request Headers**

- *idemixIdentity* – IdM-Keyrock user credentials. For example: {"name": "peter@odins.es","password": "iotcrawler"}
- *device* – endpoint of the resource's request (protocol+IP+PORT). In DCapBAC scenario, it corresponds with PEP-Proxy component. For example: "https://153.55.55.120:2354"
- *action* – method of the resource's request ("POST", "GET", "PATCH"...)
- *resource* – path of the resource request. For example: "/ngsi-ld/v1/entities/urn:ngsi-ld:Sensor:humidity.201"

**Example request**:

Bash

```
$ curl --location --request POST 'https://{{Facade-IP}}:{{Facade-Port}}/
↪CapabilityManagerServlet/IdemixTokenIdentity' \
      --header 'action: GET' \
      --header 'resource: /ngsi-ld/v1/entities/urn:ngsi-ld:Sensor:humidity.201' \
      --header 'device: https://153.55.55.120:2354' \
      --header 'idemixIdentity: {"name": "peter@odins.es","password": "iotcrawler
↪"}'
```

**Example response**:

```
{
    "id": "19tp7qp5se7t27hi9lm8uc5iko",
    "ii": 1595400471,
    "is": "capabilitymanager@odins.es",
    "su": "Peter",
    "de": "https://153.55.55.120:2354",
    "si": "I3QObay3SB531ICuwisnXvhhMSjEF77ViKwZkH9ASeMgneIJjuVHx4YAyu3acys/
↪+Jh8pK3Gwh+XC69UMZsm+SnXz+Zh0XJBpo5ZGq3DHZeayimNMW19aVlckTCGxv/
↪YtZydbjsGbJqeKXXWQPv1tzZpHhFWKNfppr13cVON30Irmcm4nbdQp672+IFaBI6WRCrqNAtmQRPW25OqKJ1k+G+Yh4/
↪UU06+kPRTwutBRChgX+Pl8W9vzxxBmknoMQbeJW3dn1DbfhB5zMX2Pa8uMKaufIV/r/
↪H0R6HSZ1d33CCv7SwwVipHq8ktp2G4UEtFYALgRx2pvlTiGEqTiiWVZwu939Q3RG/
↪wqMd8bDX6PQtexf7WMO4sbkTl1Y/
↪65DCGODnWA1Tvwn+NDHqMLXea+cdPFpl9tPFT4FLHbt0U22x81Ks4IdqLInkjXm0gBBLIT3XbKl4UtTBTwI1LZwY7p2U+9
↪oK/M0ELIj/aoApDLWKryqDP2Fx/KWh+e5s43t+T/
↪G5RaTv5HORP8dvTnLkrVgRuDpgbyJrLhPY7wXCeSSKYZk1+LvJKVzS8DW9bDbCFPTJTSyIXMO9VgfY8PLdlUVALOAjJZCQ
↪9+kdFZ+3XyKSNBeIPkx0SaIo=",
    "ar": [
      {
        "ac": "GET",
        "re": "/ngsi-ld/v1/entities/urn:ngsi-ld:Sensor:humidity.201"
      }
    ],
    "nb": 1595401471,
    "na": 1595411471
}
```


**Status Codes**

- 200 OK – OK or "There was an error getting the Token." (Unauthorized)

### 6.5.2 Test Security Facade is running

**GET /CapabilityManagerServlet/**

Test Security Facade is running.

**Example request**:

Bash

```
$ curl --location --request GET 'https://{{Facade-IP}}:{{Facade-Port}}/
↪CapabilityManagerServlet/'
```

**Status Codes**

- 200 OK – Component is running. Served at: /CapabilityManagerServlet

## 6.6 PEP-Proxy REST API

This section defines all REST APIs supported by PEP-Proxy.

**Table of contents**



### 6.6.1 Accesing to resource

The PEP-Proxy component hasn't a specific API, it uses the same API that will forward requests. The unique difference is you must add a new header *x-auth-token* where capability token must be included as a string.

**GET /**

Example of request (GET) to forward to MDR's API.

**Request Headers**

- *x-auth-token* – *Capability Token*

**Example request**:

Bash

```
$ curl --location --request GET 'https://{{PEP-Proxy-IP}}:{{PEP-Proxy-Port}}{
↪{resource}}' \
 --header 'x-auth-token: {"id": "nlqfnfa6nqrlbh9h7tigg28ga1","ii": 1586166961,"is
↪": "capabilitymanager@odins.es","su": "Peter","de": "https://153.55.55.120:2354
↪","si":
↪"MEUCIEEGwsTKGdlEeUxZv7jsh0UdWoFLud3uqpMDvnlT+GD7AiEAmwEu0FHuG+XyRi9BEAMaVPBIqRvOJlSIBkBT3K7LH
↪",
 "ar": [{"ac": "GET","re": "/ngsi-ld/v1/entities/urn:ngsi-ld:Sensor:humidity.201"}
↪],"nb": 1586167961,"na": 1586177961}'
```

### 6.6.2 Test PEP-Proxy is running

**GET /**

Test PEP-Proxy is running.

**Example request**:

Bash

```
$ curl --location --request GET 'https://{{PEP-Proxy-IP}}:{{PEP-Proxy-Port}}/'
```

**Status Codes**

- 200 OK – Component is running.

## 6.7 Ranking REST API

Documentation of the Ranking REST API.

### 6.7.1 Obtain entities by type

**GET /ngsi-ld/v1/entities**
Retrieve a list of all the entities.

**Query Parameters**

- **type** (*string*) – entity type code as `http://example.org/vehicle/Vehicle`, `http://www.w3.org/2003/01/geo/wgs84_pos%23Point`, `indexing`, etc.
- **weights** (*string*) – asdf

**Example request**:

Bash

Python

JavaScript

```
$ curl --location --request GET 'http://{{broker-host}}/ngsi-ld/v1/entities/?
↪type=http://example.org/vehicle/Vehicle'
```

```
import requests
url = "http://metadata-repository-scorpiobroker.35.241.228.250.nip.io/ngsi-ld/v1/
↪entities/?type=indexing"
payload = {}
headers= {}
response = requests.request("GET", url, headers=headers, data = payload)
print(response.text.encode('utf8'))
```

```
var request = require('request');
var options = {
  'method': 'GET',
  'url': 'http://metadata-repository-scorpiobroker.35.241.228.250.nip.io/ngsi-ld/
↪v1/entities/?type=indexing',
  'headers': {
  }
};
request(options, function (error, response) {
  if (error) throw new Error(error);
  console.log(response.body);
});
```

**Example response**:

```
[
  {
    "id": "urn:ngsi-ld:Vehicle:TEST1",
    "type": "http://example.org/vehicle/Vehicle",
    "http://example.org/vehicle/brandName": {
      "type": "Property",
      "value": "Mercedes"
```

(continues on next page)

(continued from previous page)


```
    },
    "http://example.org/vehicle/speed": {
      "type": "Property",
      "value": 80
    },
    "location": {
      "type": "GeoProperty",
      "value": {
        "type": "Point",
        "coordinates": [
          -1.1336517,
          37.9894006
        ]
      }
    },
    "@context": [
      "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context.jsonld"
    ]
  }
]
```


**Response Headers**

- Content-Type – application/ld+json

**Status Codes**

- 200 OK – no error

## 6.8 Ranking REST API

Documentation of the Indexing REST API.

### 6.8.1 Obtain entities by type

**GET /api/health**

Status of api health

**Example request**:

Bash

```
$ curl --location --request GET 'https://{{indexing-host}}/api/health'
```

**Example response**:


```
{
  "status": "UP"
}
```


**Response Headers**

- Content-Type – application/json

**Status Codes**

- 200 OK – no error

CHAPTER

SEVEN

# CONTRIBUTION! WHY NOT

We welcome contribution to the IoTCrawler project with open arms. It is a community driver project and it is the community which governs, develops, builds, tests, monitors and utilizes the IoTCrawler platform. The users and developers of the IoTCrawler framework can contribute in many ways. Let's together build an open source platform of IoT devices and services.

Before, you prepare yourself to report a bug or making pull request, please review the IoTCrawler Code of Conduct. It gives an overview of guidelines to maintain the decorum in the community.

## 7.1 Types of Contribution

### 7.1.1 As a user:

- Feature proposal
- Enhancement proposal
- Bugs reporting
- Testing

### 7.1.2 As a Developer

- Fixing open issues
- Make feature/enhancement proposal and implement them
- Improve documentation

## 7.2 Maintainers

The IoTCrawler project is consist of multiple members. The core maintainers spearhead the all development related activities. They are in-charge of reviewing and merging all pull requests in respective component of IoTCrawler, maintaining documentation, release road-map etc.

## 7.3 GitHub Contributions

All source files of IoTCrawler project are hosted on Github. Developer who wants to contribute in code base, they should follow the GitHub Contribution Guidelines.

CHAPTER

EIGHT

# GLOSSARY

CHAPTER

NINE

# RELEASES

# HTTP ROUTING TABLE

## /



## /CapabilityManagerServlet



## /XACMLServletPDP



## /api



## /ngsi-ld



## /v1