Ecosystem for COllaborative Manufacturing PrOceSses – Intra- and Interfactory Integration and AutomaTION
(Grant Agreement No 723145)

D5.10 Intrafactory Interoperability Layer II

Date: 2019-06-25

Version 1.0

Published by the COMPOSITION Consortium

Dissemination Level: Public

Co-funded by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145

Document control page

Document file: D5.10 Intrafactory interoperability layer II v1.0.doc
Document version: 1.0
Document owner: ISMB

Work package: WP5 – Key Enabling Technologies for Intra- and Interfactory Interoperability Data Analysis
Task: T5.5 – Adaptation Layer for Intrafactory Interoperability
Deliverable type: OTHER

Document status: ☒ Approved by the document owner for internal review
☒ Approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Luigi Giugliano (ISMB/LINKS)	2019-05-21	Started from "D5.9 Intrafactory interoperability layer I" updated to the second version of the document
0.3	Luca Tomaselli (NXW)	2019-06-07	Updated chapter 6 of the document to the final version after completing the development work
0.4	Mathias Axling (CNET)	2019-06-09	Updated section 4 to reflect the final architecture
0.5	Nacho González (ATOS)	2019-06-14	Updated section 7 with reference to Security Framework final architecture and deployment information
0.8	Luigi Giugliano (ISMB/LINKS)	2019-06-17	Final updates
1.0	Luigi Giugliano (ISMB/LINKS)	2019-06-20	Changes according to reviewers

Internal review history:

Reviewed by	Date	Summary of comments
Patrick McCallion (BSL)	2019-06-18	Minor corrections and suggestions
Peter Haigh (TNI-UCC)	2019-06-18	Minor edits, suggested some changes to the Executive summary and Introduction. Approved when these are updated

Index:

1 Executive Summary

The present document named “D5.10 Intrafactory interoperability layer II” is a public deliverable of the COMPOSITION project, co-funded by the European Union’s Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 723145. It reports the final results of task “5.5 – Adaptation Layer for Intrafactory interoperability” that foresees its development in work package 5 “Key Enabling Technologies for Intra- and Interfactory Interoperability and Data”.

The document owner is ISMB and this final version is a reiteration of the submitted at M18, namely D5.9 Intrafactory interoperability layer I. This version highlights the final results after a second iteration of task 5.5, regarding the development of Intrafactory Interoperability Layer. This communication layer is one of the key components in the COMPOSITION ecosystem, for granting a reliable communication layer across the intrafactory scenarios. It is therefore involved in all intrafactory use cases which will be deployed in all intrafactory pilots at end users’ premises.

Key topic addressed is the intrafactory interoperability among components. Starting with the architecture of the interoperability layer and how it was created using RAMI4.0 architectural model as a reference point. It then describes how the scalability, extensibility, interoperability and integrated security are obtained, detailing, in 4 sections, how these qualities have been achieved. The Building Management System (BMS) describes how the extensibility and the interoperability is obtained through the use of the Hardware Abstraction Layer and the Storage Handler. The interoperability is obtained by the component called LinkSmart which affects also the scalability thanks to its modularity. Finally, the integrated security is deeply explained in the last section. Even though the main qualities can be partitioned in the 4 main components, the truth is that each component contributes almost equally to each of the aforementioned qualities.

The resulting architecture is focused on providing all the necessary elements to enhance the responsiveness of COMPOSITION IoT devices, services and people by creating a dynamic and flexible shop floor communication environment able to balance usability and security. In particular, security has been a major concern for such system that ensures authentication, authorization and messages integrity exploiting the services provided by the COMPOSITION Security Framework.

Furthermore, the Intrafactory Interoperability Layer acts as a self-consistent link among all the heterogeneous physical sensors systems in the factory and the software modules in the upper layers (data processing, decision support, etc.) reducing the duplication of functions and services across them and ensuring the conformity among interconnected components communications.

It is worth mentioning that, in spite of being considered a component by the COMPOSITION’s architecture, the Intrafactory Interoperability Layer is a conglomerate of heterogeneous sub-components that act together for the same scope within a common intrafactory scenario and form the core of the IIMS.

During the lifetime of the component in light of existing used technologies as: Keycloak, RabbitMQ and LinkSmart® the TRL has been raised to from 6 to 7. It is worth specifying that in the cases of RabbitMQ and LinkSmart® the components have been furthermore developed with the addition of several modules. (Please refer to section 5 and section 7.1.1)

The biggest asset of this component is the ability to provide a complete Industry solution which goes from the hardware level with the Building Management System (section 6) all the way up to the data analytics using LinkSmart® (section 5) ensuring the security of each sub component as defined in section 7.

2 Abbreviations and Acronyms

Table 1: Abbreviation and acronyms table

Acronym	Description
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BDA	Big Data Analytics
BMS	Building Management System
CEP	Complex Event Processing
CRUD	Create, Retrieve, Update and Delete
DFM	Digital Factory Model
DLT	Deep Learning Toolkit
DSS	Decision Support System
EPL	Event Processing Language
GPIO	General-Purpose Input/Output
HAL	Hardware Abstraction Layer
HMI	Human Machine Interfaces
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IIL	Intrafactory Interoperability Layer
IIMS	Integrated Information Management System
IoT	Internet of things
JWS	JSON Web Signature
MES	Manufacturing Execution System
MQTT	Message Queuing Telemetry Transport
PDF	Portable Document Format
RAMI	Reference Architectural Model Industrie
OGC	Open Geospatial Consortium
OIDC	Open ID Connect
PLC	Programmable Logic Controller
REST	REpresentational State Transfer
SAML	Security Assertion Markup Language
SCADA	Supervisory Control And Data Acquisition
SQL	Structured Query Language
TCP	Transmission Control Protocol

TLS	Transport Layer Security
TRL	Technology readiness level
UI	User Interface
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

3 Introduction

3.1 Purpose, Context and Scope of this Deliverable

This document describes the development carried out in COMPOSITION's project task 5.5, named Adaptation Layer for Intrafactory interoperability.

In light of the heterogeneous nature of the sub-components that form the Intrafactory Interoperability Layer, a comprehensive study and overview of project's use cases, in which this task has been involved, have been carefully evaluated. It has been immediately clear that the Intrafactory Interoperability Layer is a necessary component for all intrafactory use cases. It also emerged the end-users required the component deployed at the shop-floor level at their premises, in order to have control over data and enforce security with respect of internal policies and regulations.

The developed components are going to be deployed in the aforementioned use cases, in which all project's end users are going to be involved. In fact, the component will be in charge of connecting data points in a security by design environment, creating a data stream that through many transformations and adaptations. The data will flow from the source in which existing and novel sensors act together for creating near real time readings, modelling shop-floor machinery, through the COMPOSITION's components that will shape these data, enriching them with simulations, forecasting, previsions and much more., Finally the data will reach its final destination that would be the Human Machine Interfaces, with the recipients dispatching capabilities offered by the notification engine.

3.2 Content and Structure of this Deliverable

The reminder of the document is divided into seven sections starting with chapter 4 describing the overall intrafactory communications architecture and interoperability. Subsequent chapters then focus's in more depth on those components that enables the system. Chapter 5 describes LinkSmart® which is the module that enables communication between the BMS and COMPOSITION ecosystem as well as providing service registration and discovery services. Chapter 6 discusses the BMS at Boston Scientific and KLEEMANN and how the hardware is abstracted to enable data to flow from the sensors through to the COMPOSITION system for data analysis. Chapter 7 then discusses the brokering and security aspects of the intrafactory communication to ensure correct authorisation and authentication for data access. Chapter 8 Distributed Intrafactory notification enabling service describes how a centralised service to exchange heterogeneous messages through a common interface is implemented. Chapter 9 discusses how Inter and Intra factory communication is achieved, then finally Chapter 10 concludes the report.

4 Architecture Model for Intrafactory Interoperability

4.1 Overview

The role of the Intrafactory Interoperability layer is defined in the Description of Action (COMPOSITION, 2016). It provides the integration and adaptation in the COMPOSITION IIMS of shop-floor data sources, i.e. sensors, control units (e.g. PLCs) and existing software systems (e.g. Manufacturing Execution System (MES), Supervisory Control And Data Acquisition (SCADA)). The aggregated data is also forwarded to the COMPOSITION Agent Marketplace where it is used to support the agent decision making.

The COMPOSITION Intrafactory Interoperability Layer spans two RAMI4.0 Layers: The Interoperability Layer and the Communication Layer. The Integration Layer performs digitization of assets; the mapping from the physical world to the digital and provides virtualization of shop-floor resources. The main component here is the Building Management System (BMS). The Communication Layer provides standardized data formats, protocols and interfaces from the Integration Layer to the Information Layer, which processes and stores data and events. The Message Broker and connected micro services are responsible for this task. Interface endpoints are managed by the Service Catalog. The strict layering principle of RAMI4.0 is not compromised, as no communication bypasses the communication layer. The basic (not composed) administrative shells for the assets are located in this layer.[1]

The main quality concerns for the Intrafactory Interoperability Layer are scalability, extensibility, interoperability and integrated security.

[1] https://www.plattform-i40.de/I40/Redaktion/EN/Downloads/Publikation/structure-of-the-administration-shell.pdf

4.2 Functional View

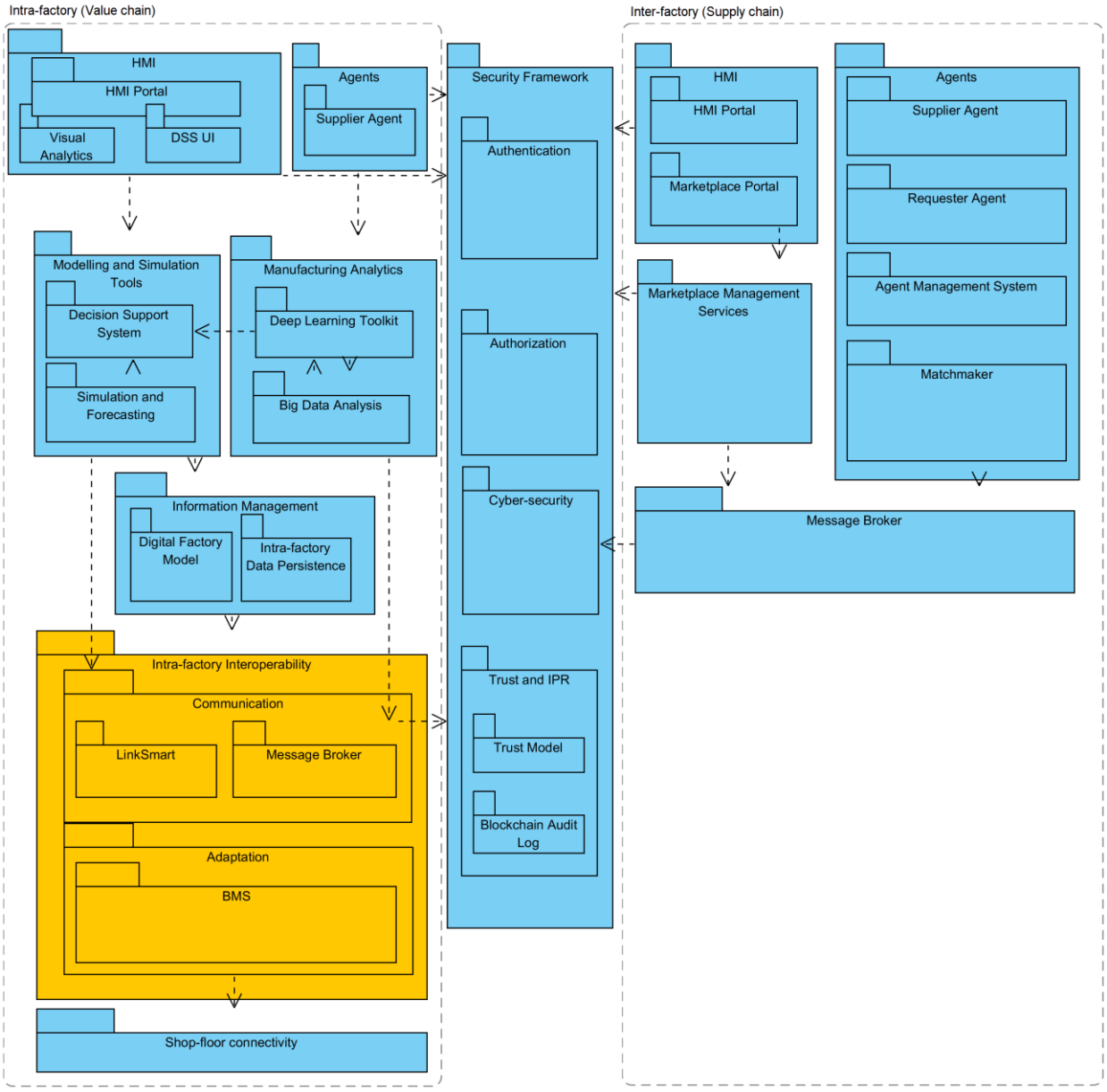

Figure 1: Functional packages of the COMPOSITION system

The functional responsibilities of the Intrafactory Interoperability Layer are (COMPOSITION, 2016):

- Communicate in real-time, near-real time and batch (scheduled) mode with heterogeneous shop floor data sources and existing software systems
- Provide data handling for the COMPOSITION system
  - Message translation
  - Data filtering
- Provide device management for the COMPOSITION system
  - Administrative shells
  - Virtualization of resources
    - Combination of data from the different systems

The BMS fulfils all COMPOSITION requirements for adaptation of shop floor data sources (installed systems ("legacy") and heterogeneous sensors). It provides data filtering, message translation, virtualization of resources and administrative shells. However, should this be desired, e.g. for reasons of previous operational experience or company policy, other complementary components like LinkSmart or IoT Hub could also be installed for this purpose by an organization adopting COMPOSITION.

The BMS integrates with several heterogeneous systems at shop-floor level through the hardware abstraction layer. The BMS provides an OGC SensorThings Sensing Profile API and a configuration API to the upper layers of COMPOSITION. Towards the intrafactory system, the Message Broker MQTT and REST APIs are used to propagate data. Microservices such as the distributed intrafactory notification enabling service are integrated through the message broker and MQTT. All components in the upper layer use the communication infrastructure provided by the Message Broker and REST endpoints in the Service Catalog.

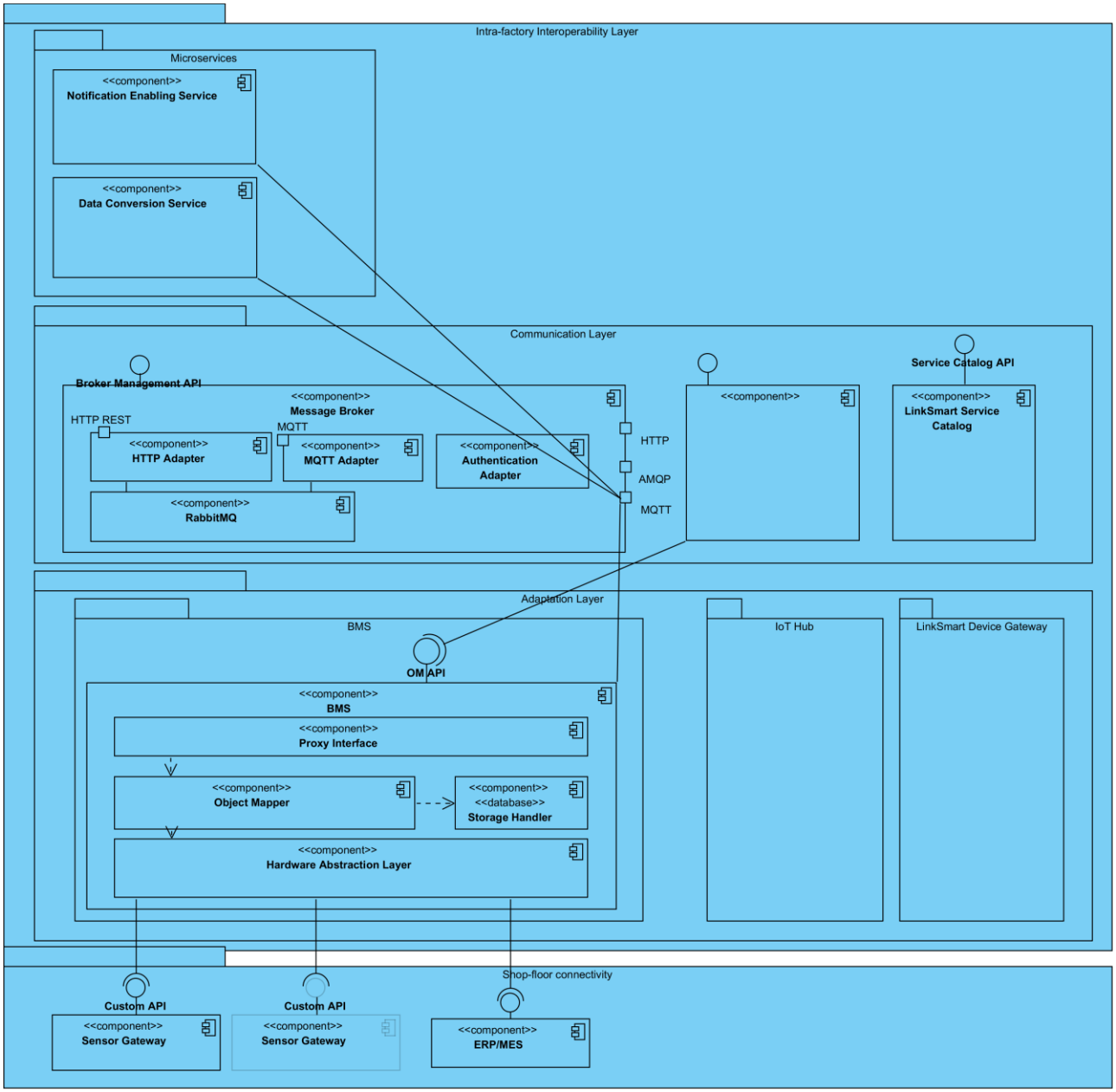

Figure 2: Intrafactory Interoperability Layer and Shop-floor

4.3 Information View

The Digital Factory Model (DFM) (described in D3.3) is the common source for information about the factory equipment and processes for all COMPOSITION components. Static and dynamic data provided from the COMPOSITION system are described in a common format using the DFM schema. The machines, devices and sensors in the factory instance are described in a Deployment Model; this also contains the mapping of these resources to a specific IoT data channel, such as a MQTT topic or REST endpoint. The DFM provides interfaces that other components use for reading and updating the models.

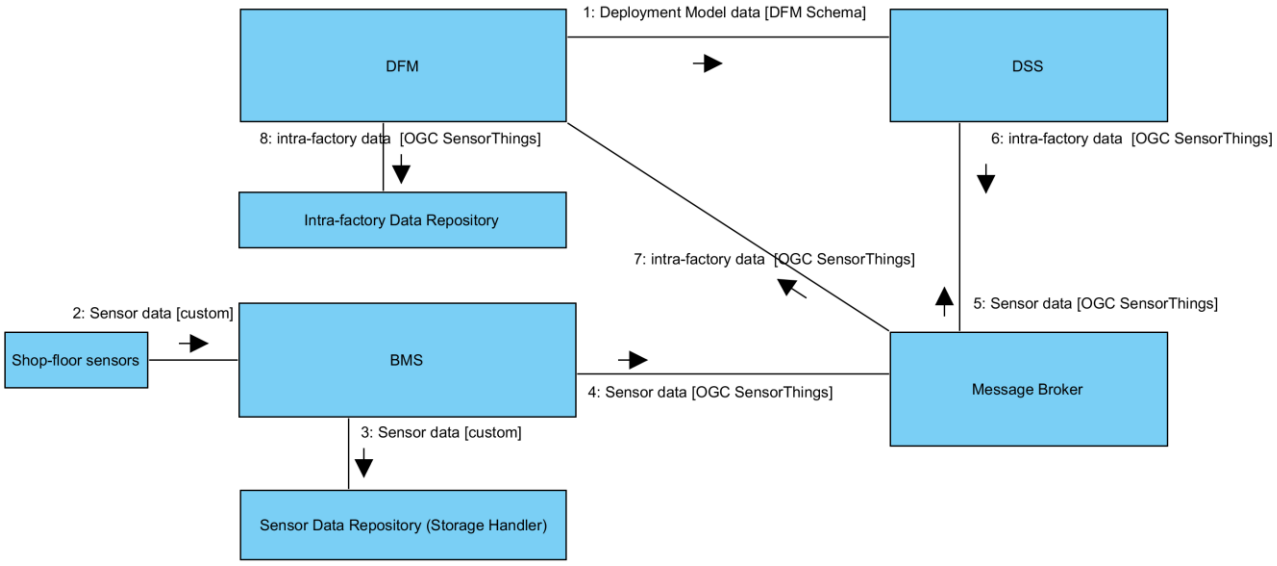

Figure 3: Example data flow

The format chosen for sensor data in COMPOSITION is SensorThings API Sensing Entities[2] JSON encoding. The BMS will deliver data from sensors and other shop-floor sources to the Message broker in this format. The data is published on a MQTT topic structure adapted from the SensorThings Sensing MQTT Extension which allows subscribers to be notified when Observations are added to a Datastream or FeatureOfInterest.

Data consumers may subscribe to these topics to receive the sensor data. Components like the Deep Learning Toolkit (DLT) are configured at deployment to subscribe (mediated via the BDA) to specific data streams.

The Decision Support System (DSS) will dynamically visualize factory processes. The mapping from the factory model processes, assets and equipment to sensors and Datastreams is located in the DFM. It is configured at installation and device provisioning and the information is provided to the other components by a dedicated service in the DFM.

4.3.1 Data Persistence

The BMS provides the Storage Handler (see section 6.2) for persisting shop-floor data. For data generated in the COMPOSITION system, e.g. the results from predictive maintenance deep learning networks or alerts, a storage service provided by the DFM is used.

4.3.2 Operational Management

The onboarding process for sensors by transferring the device definitions and Datastream mappings to assets in the DFM Deployment Model is currently done in the DFM but will, in exploitation of the COMPOSITION system, be the responsibility of a dedicated Commissioning System. This tool is an HMI and a set of services for convenient installation and configuration of various data sources, e.g. PLCs and sensors. Due to low priority in the development backlog and low innovation potential, this has not been realized.

The LinkSmart Service Catalog is used for registration and discovery of COMPOSITION services in both the Intra- and Inter-factory deployments. This information is also used for operational management and system supervision.

[2] http://docs.opengeospatial.org/is/15-078r6/15-078r6.html

4.4 Deployment View

To integrate with the shop floor infrastructure, the typical deployment of BMS will be on a separate node in the factory. However, using adapters in the factory, cloud installation is also possible. Docker deployment is also an option but may not be suitable when the hardware abstraction layer need access to specific drivers.

The message broker and information processing components of the Intrafactory Interoperability Layer is installed as docker containers in a docker host on a node at the factory or in the cloud. The Security Framework is deployed on a separate node.

4.5 Data Model

The Intrafactory Interoperability Layer creates the technical foundations for interconnection between hardware and software systems inside the factory as well as between humans and machines. With this paradigm, multiple and heterogeneous smart objects continuously exchange a great volume and variety of information. A uniform and agreed data model is essential for system cooperation avoiding multiple layers of translations.

The OGC SensorThings API provides an open, geospatial-enabled and unified way to interconnect the Internet of Things (IoT) devices, data, and applications over the Web. At a high level, the OGC SensorThings API provides a standard way to manage and retrieve observations and metadata from heterogeneous IoT sensor systems as well as an efficient machine and human readable JSON representation.

The SensorThings API is designed for the REST on HTTP protocol but it also provides an MQTT extension to enhance the SensorThings services publish and subscribe capabilities. MQTT extension fits perfectly well into the communication architecture described in section 7, providing a shared common data model for COMPOSITION components.

In the following, Figure 4 shows the UML diagram of the entities of the SensorThings API.

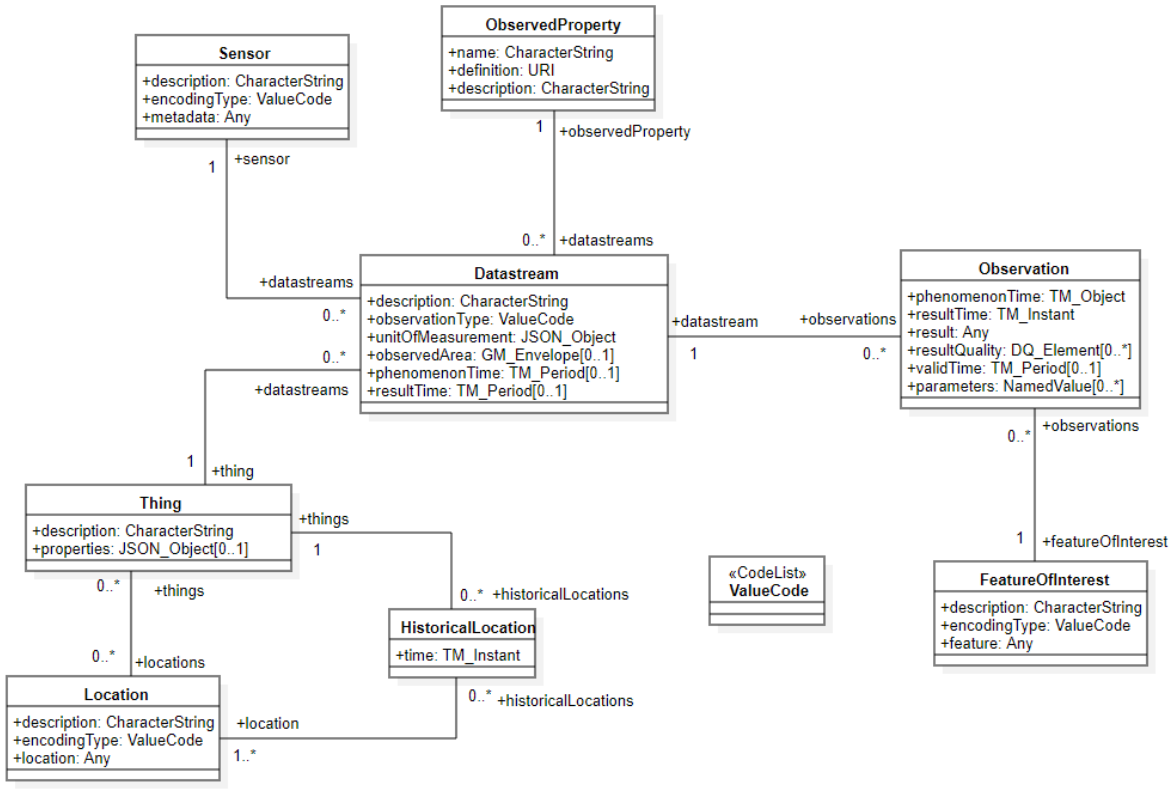

Figure 4: OGC Sensor Things data model

This model has been adopted to exchange information between the COMPOSITION components on the Intrafactory Interoperability Layer. However, a Building Management System (described in section 6) has

been specifically developed for fulfilling COMPOSITION's needs and acts as translation layer for both machineries deployed at shop floor level and low-level sensors that usually operates only with proprietary communication protocols and data formats.

More details about the data factory model entities are available in Deliverable 3.2.

5 LinkSmart

5.1 LinkSmart Overview

LinkSmart® is an open source platform for developing IoT applications in various domains, such as smart cities, Industry 4.0, smart grid, and more. The platform provides building blocks as generic and domain-specific services to efficiently implement applications in the Internet of Things. These include basic services such as device abstraction, data storage, live data management, and advanced ones such as stream mining and online machine learning. Following the microservices pattern, LinkSmart services can be arranged together and alongside other services depending on concrete use cases.

In this project, we extend the LinkSmart platform to realize three COMPOSITION modules:

- In Big Data Analysis for propagating real-time data and orchestrating the learning process. This component is addressed in deliverable D5.1 as the LinkSmart Learning Agents.
- As central information point for service registration and discovery within intra- and inter-factory networks. This component is described in Section 5.2.
- Lastly, within the intrafactory interoperability layer (IIL) in order to add networking capabilities to Building Management System (BMS) and connect it to the rest of COMPOSITION ecosystem. Section 5.3 briefly described this component.

5.2 Service Discovery

The COMPOSITION system operates on multiple interconnected networks consisting of numerous web services. The services are standalone components, often unaware of other services configurations and dynamic endpoints. Thus, it is necessary to provide a registry maintaining meta information about all services. While this requirement was not envisioned in the initial architecture designs, later on it was added in order to maintain information such as the public key of each service for verification of published messages by services. Each service will be responsible for submitting the required meta information (incl. endpoints, public key) of itself to the registry such that other services can retrieve them. The COMPOSITION service registry is implemented as part of the LinkSmart platform. This component is called LinkSmart Service Catalog.

Service Catalog describes the services available in the network and exposes a JSON-based MQTT and RESTful HTTP APIs. It contains entries of everything that is meant to be discovered or interacted with by applications and other services. Each entry corresponds to a “service” and not a physical device or a “virtual sensor” worth being considered as such. Examples of COMPOSITION services include BMS, Big Data Analysis, and Decision Support System.

5.2.1 Design

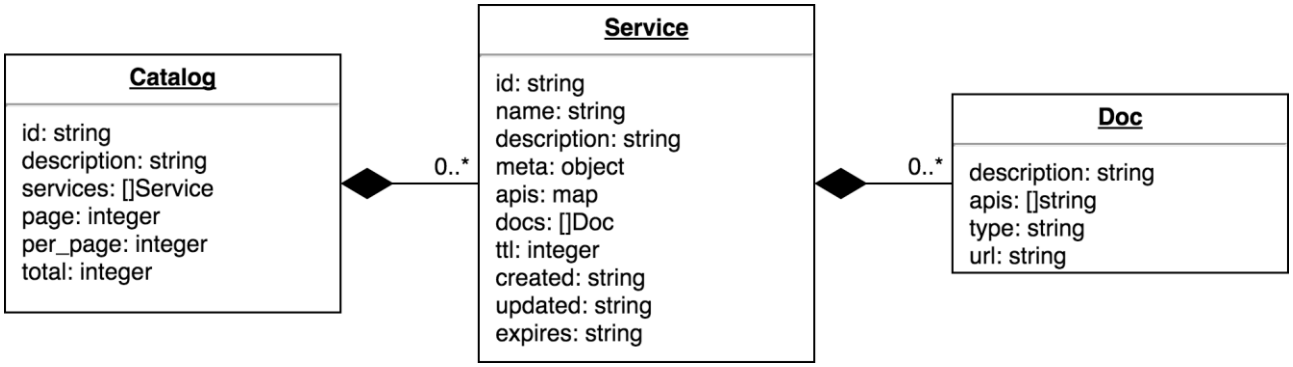

Figure 5: Data model of Service Catalog

Service Catalog API is designed in a way to offer flexibility for service and service meta information discovery. Figure 5 shows the data model of Service Catalog. The attributes of the data model are described below:

Catalog object consists of:

- id: unique id of the catalog
- description: a friendly name or description of the service
- services: an array of Service objects
- page: the current page in catalog
- per_page: number of items in each page
- total: total number of registered services

Service object consists of:

- id: unique id of service
- name: RFC6339 service name (e.g. _bms._tcp)
- description: friendly name or description of a service
- meta: a hash-map for optional meta-information
- apis: a map of API names and URLs
- docs: an array of Doc objects describing service documentations
- ttl: time after which the service should be removed from the catalog, unless if it is updated within the timeframe.
- created: RFC3339 time of service creation
- updated: RFC3339 time in which the service was lastly updated
- expires: RFC3339 time in which the service expires and is removed from the catalog (only if TTL is set)

Doc object consists of:

- description: description of the external document
- apis: an array listing APIs documented in this documentation
- url: URL to the external document
- type: the MIME type of the document (e.g. plain/text for wikis, application/openapi+json;version=2.0 for OpenAPI specs v2.0)

Service Catalog performs CRUD (Create, Read, Update and Delete) operations on service entries. It also caters the list of services on request from applications. RESTful endpoints are provided to retrieve resources from the catalog. The operations are as follows:

Table 2: LinkSmart Service Catalog operations

Method	Path	Description
GET	/	Retrieves API index.
POST	/	Creates new `Service` object with a random UUID
GET	/{id}	Retrieves a `Service` object Updates the existing `Service` or creates a new one (with the provided ID)
PUT	/{id}	Updates the existing `Service` or creates a new one (with the provided ID)
DELETE	/{id}	Deletes the `Service`
GET	/{path}/{operator}/{value}	Service filtering API

This data model and the API are described as OpenAPI specification and can be accessed on the Linksmart project website[3].

OpenID is used to authenticate all requests from the API. The token issuing and validation is done using a supported authentication provider (currently only Keycloak is supported). Applications retrieve appropriate tokens from the authentication server and provide it to Service Catalog upon every request. Service Catalog then validates the token and responds with appropriate HTTP status codes. Alternatively, the applications may use Basic Auth over HTTPS such that Service Catalog internally takes care of token retrieval and validation. In addition, Service Catalog can be configured to perform authorization based on application user id and requested path. Configuration details are described in Section 5.2.4.

5.2.2 Implementation

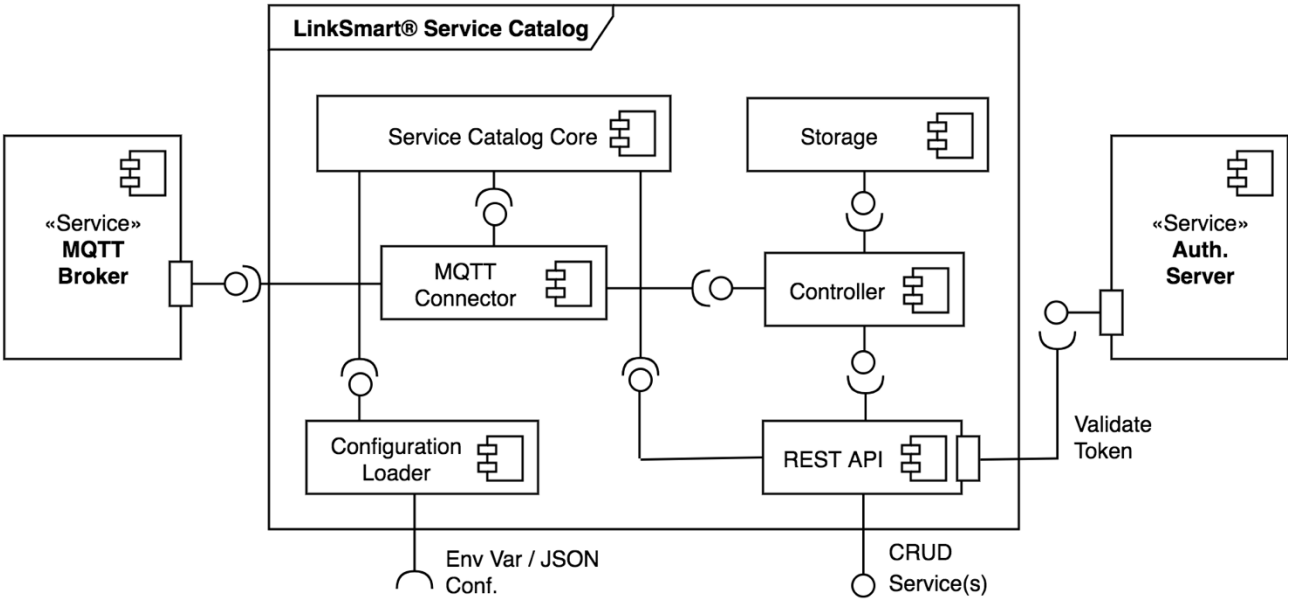

Figure 6: Components of the Service Catalog

Service Catalog implements the registry in the form of a service with several loosely coupled components. Figure 6 illustrates Service Catalog components where:

- Service Catalog Core implements the main function and instantiates other modules.
- Configuration Loader loads the file with the service configuration.
- Storage implements a persistency back-end for the stored information. There is an in-memory, as well as a secondary storage (LevelDB) implementations available.
- Controller abstracts storage calls and provides high level methods to other components.
- REST API implements the REST API of the Service Catalog according to the OpenAPI specification.
- MQTT Connector provides a simple service registration API via MQTT.
- Auth. Server is the Authentication Server (i.e. Keycloak) providing authentication tokens.
- MQTT Broker is one or more MQTT brokers which Service Catalog subscribes to.

Service Catalog is written in Go and provided as open source software under Apache license. The source code is available in a LinkSmart repository[4].

[3] https://docs.linksmart.eu/display/SC/Service+Catalog+API
[4] https://code.linksmart.eu/projects/SC/repos/service-catalog

5.2.3 Continuous Integration and Delivery

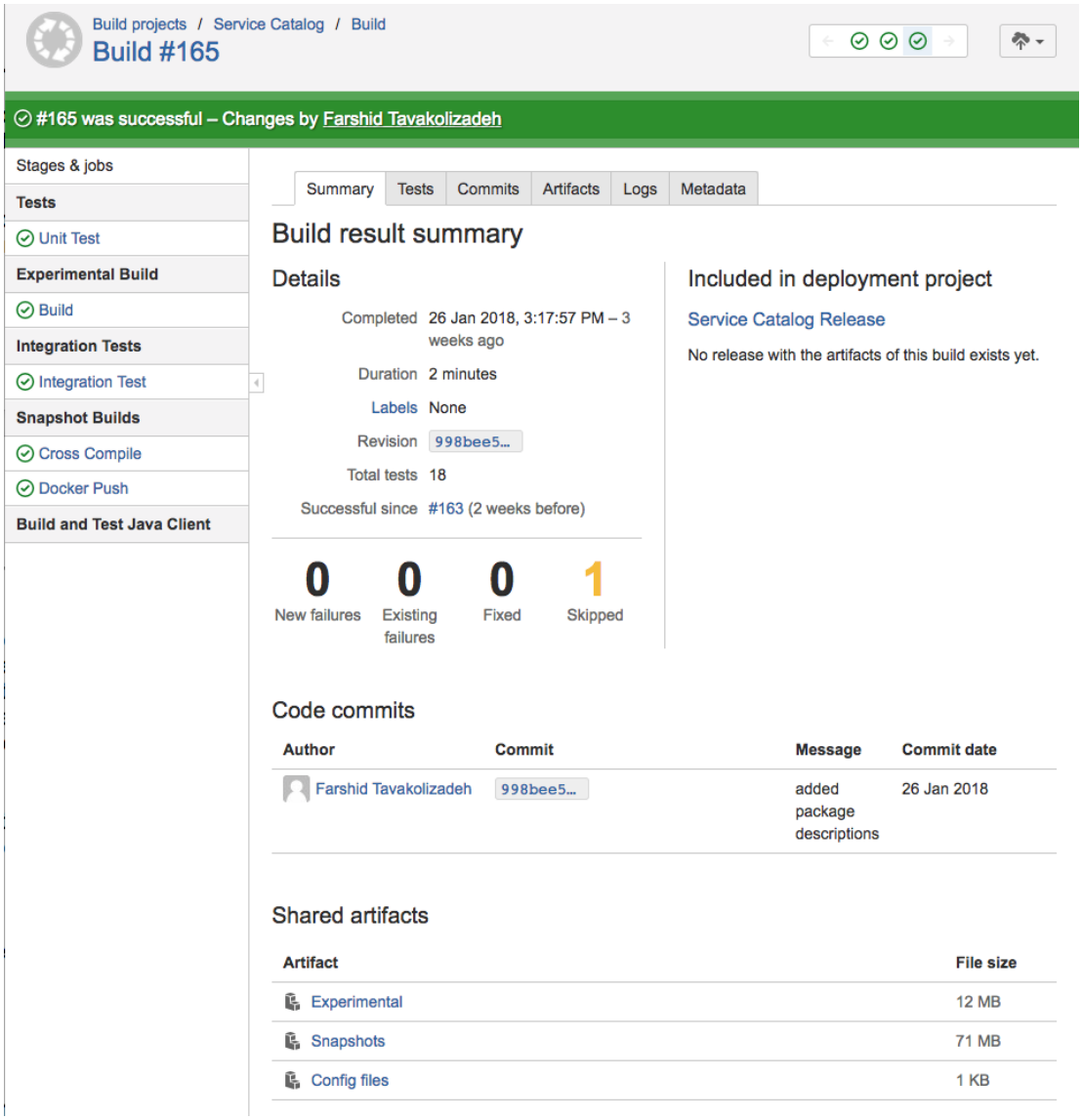

Figure 7: Service Catalog build project

Service Catalog code is built and tested continuously on a publicly accessible continuous integration (CI) server[5]. The CI server also compiles the project into executables for all common platforms. In addition, Service Catalog is delivered as a Docker image that is available to COMPOSITION consortium and public in the LinkSmart Docker Registry[6].

5.2.4 Deployment

In COMPOSITION, the Docker image is pulled from the public LinkSmart registry. The deployment instructions using command line are described below:

Run with default configurations

```
docker run -p 8082:8082 docker.linksmart.eu/sc
```

The index of the REST API should now be accessible at: http://localhost:8082

[5] https://pipelines.linksmart.eu/browse/SC
[6] https://docker.linksmart.eu/repository/sc

Run with custom configuration
For a configuration file located at /path/on/host/service-catalog.json

```
docker run -p 8082:8082 -v /path/on/host:/conf docker.linksmart.eu/sc -conf
/conf/service-catalog.json
```

In COMPOSITION, we deploy Service Catalog and other Docker containers using the graphical user interface of Portainer. Portainer is a lightweight management UI which allows you to easily manage your different Docker environments[7].

Configuration
Service Catalog is configured using a JSON configuration file, path to which is provided to the SC via -conf flag. By default, the service looks for a configuration file at: conf/service-catalog.json

The configuration has the following format:

```
{
 "description": "string",
 "dnssdEnabled": "boolean",
 "storage": {
  "type": "string",
  "dsn": "string"
 },
 "http" : {
  "bindAddr": "string",
  "bindPort": "int"
 },
 "mqtt":{
  "broker": {
    "id": "string",
    "url":"string",
    "regTopics": ["string"],
    "willTopics": ["string"],
    "qos": "int",
    "username": "",
    "password": ""
  }
  "additionalBrokers": [],
  "commonRegTopics": ["string"],
  "commonWillTopics": ["string"]
 },
 "auth": {
    "enabled": "bool",
    "provider": "string",
    "providerURL": "string",
    "serviceID": "string",
    "basicEnabled": "bool",
    "authorization": {}
  }
}
```

Where:

- description is a human-readable description for the SC
- dnssdEnabled is a flag enabling DNS-SD advertisement of the catalog on the network
- storage is the configuration of the storage backend
    - type is the type of the backend (supported backends are memory and leveldb)

[7] https://github.com/portainer/portainer

    - dsn is the Data Source Name for storage backend (ignored for memory, "file:///path/to/ldb" for leveldb)
- http is the configuration of HTTP API
    - bindAddr is the bind address which the server listens on
    - bindPort is the bind port
- mqtt is the configuration of MQTT API
    - broker is the configuration for the main MQTT client
        - id is the service ID of the broker (Optional)
        - url is the URL of the broker
        - regTopics is an array of topic that the client should subscribe to for addition/update of services
        - willTopics is an array of will topic that the client should subscribe to for removal of services (Optional in case TTL is used for registration)
        - qos is the MQTT Quality of Service (QoS) for all reg and will topics
        - username is username for MQTT client
        - password is the password for MQTT client
    - additionalBrokers is an array of additional brokers objects.
    - commonRegTopics is an array of topics that all clients should subscribe to for addition/update of services (Optional)
    - commonWillTopics is an array of will topic that the client should subscribe to for removal of services (Optional in case commonRegTopics not used or TTL is used for registration)
- auth is the Authentication configuration
    - enabled is a boolean flag enabling/disabling the authentication
    - provider is the name of a supported auth provider
    - providerURL is the URL of the auth provider endpoint
    - serviceID is the ID of the service in the authentication provider (used for validating auth tokens provided by the clients)
    - basicEnabled is a boolean flag enabling/disabling the Basic Authentication
    - authorization - optional, see authorization configuration

All attributes can be overridden using environment variables. For example, the bindPort for http can be set via SC_HTTP_BINDPORT variable.

5.2.5 Usage

COMPOSITION services register their meta information to a Service Catalog instance. Overall, there will be two Service Catalog instances: one in the inter-factory network and another one in intrafactory. Services shall use the PUT method to submit their information using predefined unique IDs.

An example for service registration is given below:

```
PUT http://service-catalog-endpoint/service_unique_id
```

Body (including optional fields):

```
{
  "description": "service description",
  "meta": {
    "publicKey": "RSA public key"
  },
  "apis": {
    "REST API": "http://service-local-endpoint"
  },
  "docs": [
    {
      "description": "Open API Specs",
      "type": "application/openapi+json;version=2.0",
      "url": "http://link-to-openapi-specs.json"
    }
  ],
```

```
  "ttl": 120
}
```

In this example, publicKey meta field contains the public key of this service. Other services and applications retrieve this key and use it to verify the messages published by this particular service to the central MQTT broker (see section 7.2.2). The details of message verification are provided in appropriate deliverables and chapters.

5.3 Device Integration

Integrating the shop-floor into the COMPOSITION ecosystem is one of the most important aspects of the project. The integration requires components that realize the following functionalities:

1. Data collection from proprietary and legacy systems.
2. Transformation into OGC SensorThings data model and exposing them using IoT communication protocols.

The initial plan was to realize these functionalities within two separate components; however, we decided to merge them due to several factors. First, data collection and transformation are consecutive so implementing them in a single component is more pragmatic. Second, when dealing with large amounts of data, it is more efficient to perform processing operations in memory and within the same process. Separating the logic into two components will add inter-component communication overhead. Lastly, the merge reduces one level of data model abstraction, reducing the need to design inter-component data models.

Chapter 6 provides a detailed description of this component, offering data collection from shop-floor interfaces, transforming them to OGC SensorThings, and additionally, offering storage capabilities.

6 Building Management System

The Building Management System, provided by a project development stakeholder (NXW), is the translation layer providing shop floor connectivity from sensors to the COMPOSITION system. It's responsible for the data collection from the factory production area. It provides support for the different protocols involved in the chain (e.g. ModBus, KNX, etc.) and then the Hardware Abstraction Layer is in charge of translating all this data into a common format, understandable by the Composition upper layers. In addition to this, the raw data are stored inside the BMS for offline debug purposes.

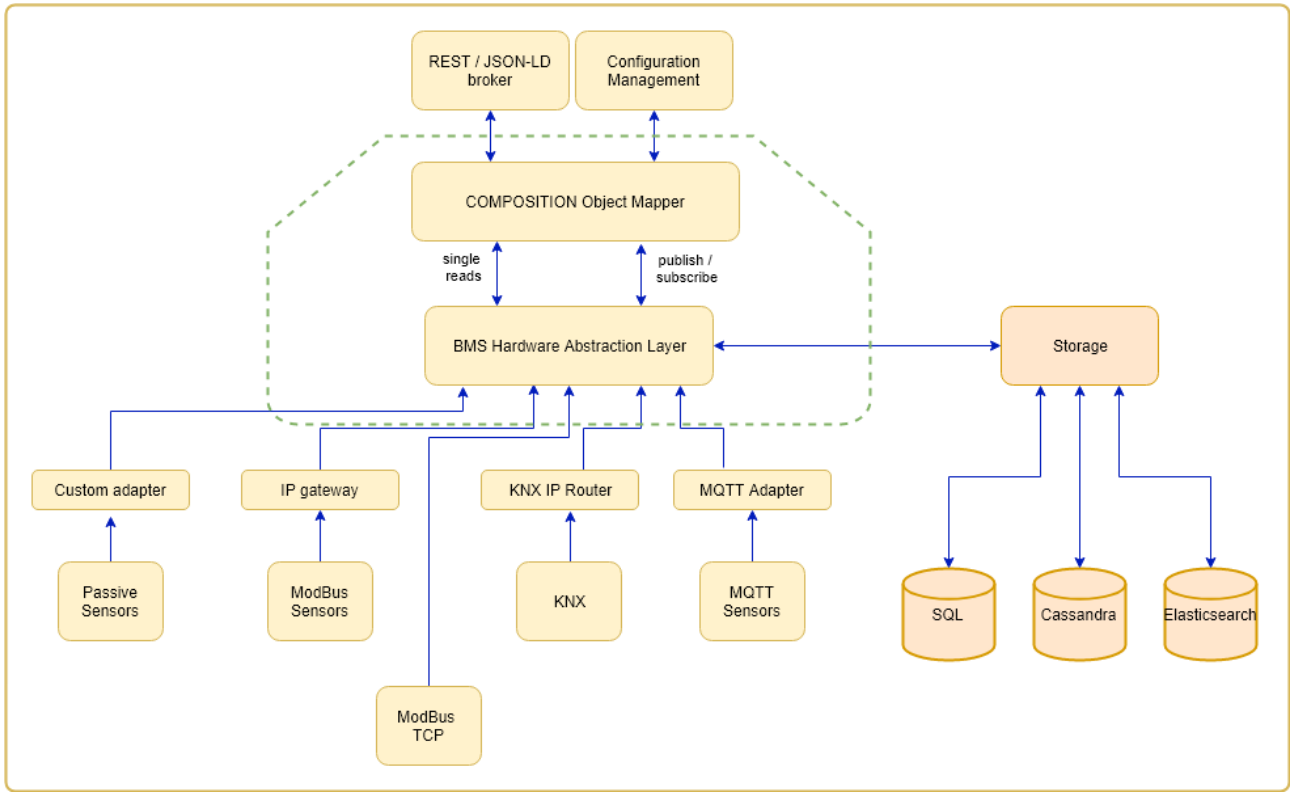

Figure 8: Components on top of the BMS

6.1 Hardware Abstraction Layer

The components depicted in Figure 8 are built on top of the existing BMS software modules provided by NXW, which guarantee low level interoperability with a number of different field buses (this is positioned at the Asset / Integration RAMI layers). Such modules gather data read from the sensors installed in the local environment, potentially interconnected through different field buses (e.g. KNX, Modbus, MQTT), and organize it into a uniform Data Model. This model provides a representation of sensor (and actuator) data which is independent of the physical type of underlying devices (Information/Communication RAMI layers).

The BMS Hardware Abstraction Layer (HAL) is a software module that primarily abstracts the low-level details of various heterogeneous fieldbus technologies and provides a common interface to its users (i.e. other software modules communicating with it). It adapts the fieldbus technologies and provides the necessary logic to manage them accordingly to their respective constraints - e.g. timing constraints. It also implements optimisations - e.g. avoid spamming the KNX bus with too many messages, pack contiguous Modbus reads into a single multi-register read.

It supports KNX, BACnet, Modbus/TCP and, Modbus/RTU as well as, several other proprietary control protocols. It can be interconnected with specific field buses either directly - such as via RS232/485 serial ports or GPIOs - or through the use of IP based gateways - such as KNX IP router and/or interface, Modbus/TCP gateways. It can be extended by developing modules that can be dynamically plugged into its

core. Due to the nature of the pilot use cases, new custom sensors have been developed by COMPOSITION partners (i.e. fill level sensors, vibrometers), therefore a specific plugin has been implemented and integrated in the HAL core component for supporting device communications.

The HAL component has been enhanced in order to support communications via MQTT, which is the protocol used by sensors deployed in the project use cases. It receives JSON messages through MQTT from the various sensors, extracts the data and organise it into a uniform Data Model (OGC Sensor Things data format).
MQTT is a generic transport protocol and it is reasonable that different type of sensors might publish different JSON messages with different representations of the data. This is the reason why the plugin needs a preliminary configuration before running the system, in order to specify where relevant information (such as sensor Id or data timestamp) is contained. Afterwards the plugin is ready to collect the input from the shop-floor and to publish it to the Composition broker in the OCG Sensor Things data format (as described in Section 4.5).

6.2 Storage Handler

Beside the HAL, the BMS provides a repository for storing the real-time data incoming from the production facilities. The Storage Handler supports different backends such as SQL, Cassandra, ElasticSearch, where data can be pushed directly from the HAL. In Composition, ElasticSearch has been chosen as database engine, because the component is used mainly for debug purposes and ES natively offers RESTful search, which is highly useful in text-based analysis and filtering.

The Storage Handler provides an API whose format is inspired by FIWARE NGSI[8] (Next Generation Service Interface). Its data model is composed by three main elements:

- context entities
- attributes
- metadata

Context entities are the centre of gravity in the FIWARE NGSI information model. An entity represents a thing, i.e. any physical or logical object (e.g., a sensor, a person, a room, an issue in a ticketing system, etc.), each entity has an entity id and an entity type. Context attributes are properties of context entities, for example the axis of vibrometer sensor. Attributes are described by their attribute name, attribute type, attribute value and metadata. The last property, context metadata, is an optional part of the attribute value (an example is the “result time” parameter in OGC Sensor Things). Similarly to attributes, each piece of metadata has a metadata name, a metadata type and a metadata value.

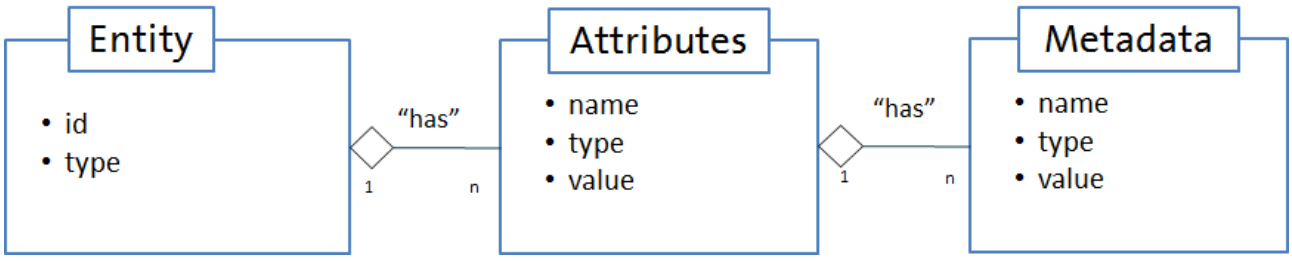

Figure 9: NGSI data model

As previously mentioned, the Storage Handler comes with a RESTful API, which exposes two endpoints:

- `/entities`
- `/entities/{Id}`

The first one returns the complete set of entities, listing their *Ids* and types. The second endpoint presents the details of a specific entity (the one specified in the URL by its *Id*), with all its attributes and metadata.

[8] http://fiware.github.io/specifications/ngsiv2/stable/

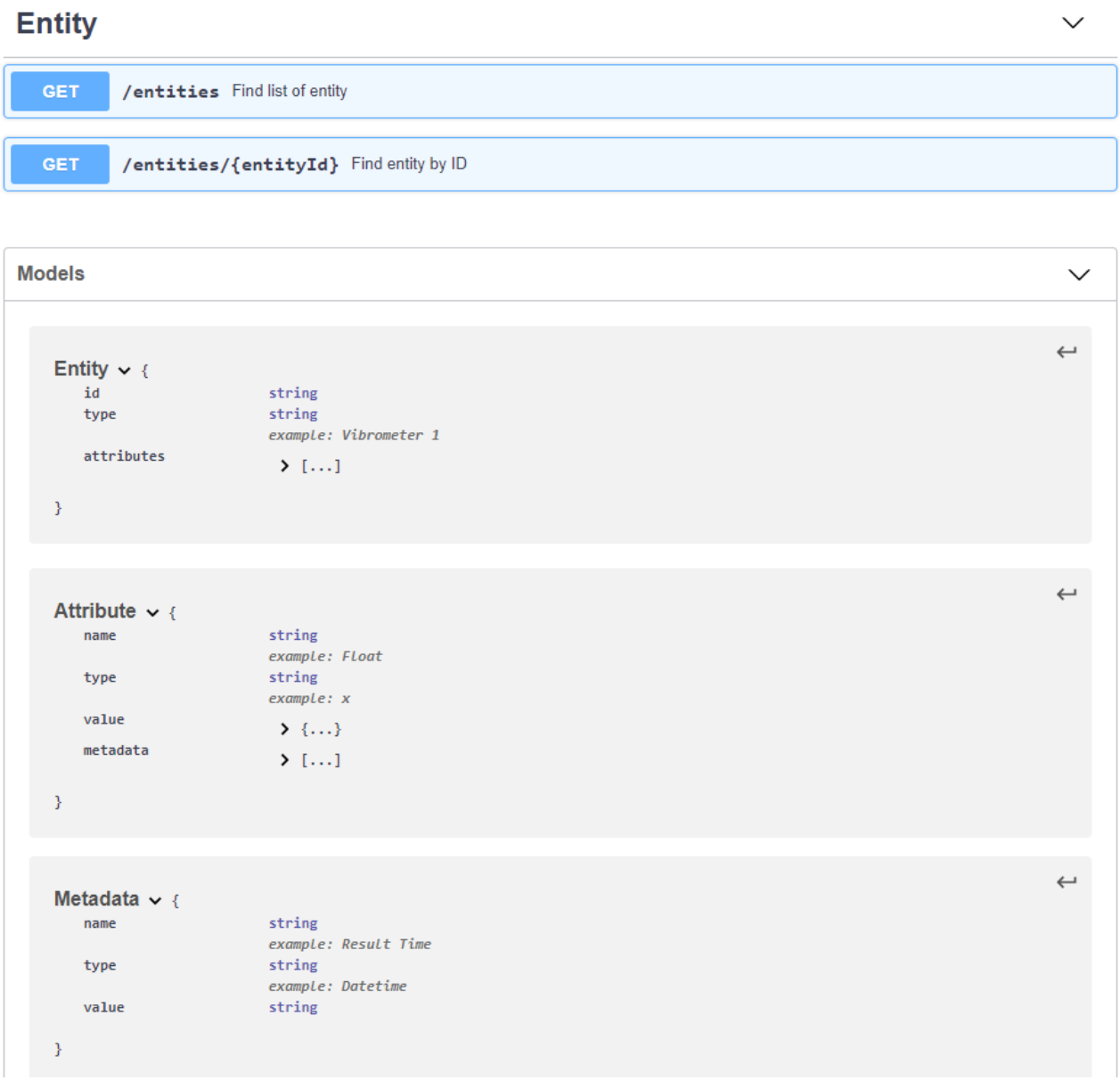

Figure 10: Storage Handler REST API

The Storage Handler also supports a simple query language that enables filtering the data on specific time periods. The filtering request can be specified directly into the URL, as a query parameter ("*q*") and the format of the request is `timestamp {OPERATOR} {DATE}`.

- *{OPERATOR}* can be one of *equal* (==), *greater than* (>), *less than* (<), *greater or equal than* (>=), *less or equal than* (<=), *between* (..)
- *{DATE}* must be expressed in ISO format

For example, to get the data collection related to sensor with *Id=60*, from timestamp *11:00 May 27th, 2019* to now, the URL would be:

`entities/60?q=timestamp>=2019-05-27T11:00:00Z`

Similarly, to get data (of the same sensor) between *11:00* and *14:00* on *May 27th, 2019* the URL is:

`/entities/60?q=timestamp==2019-05-27T11:00:00Z..2019-05-27T14:00:00Z`

It is also possible to limit the results up to a maximum number of datasets by adding the "*limit*" parameter (e.g. `/entities/60?limit=10`).

A specific component called RAAS has been developed for securing all COMPOSITION APIs (see Section 7.1.2). Therefore, Storage Handler REST API accepts connections only from the host where RAAS resides, while all other connection attempts are blocked.

6.3 Deployment

The BMS has been deployed on a separate cloud installation and it collects data from both pilots (BSL and KLE). The two physical on-site installations slightly differ due both to the nature of the data provided (different sensors installed) and to diverse company's privacy policies.

6.3.1 KLEEMANN Pilot

Two different types of sensors have been deployed on KLEEMANN site: one for reading vibrations from production machines and one for monitoring the level of waste in bins and metal containers.

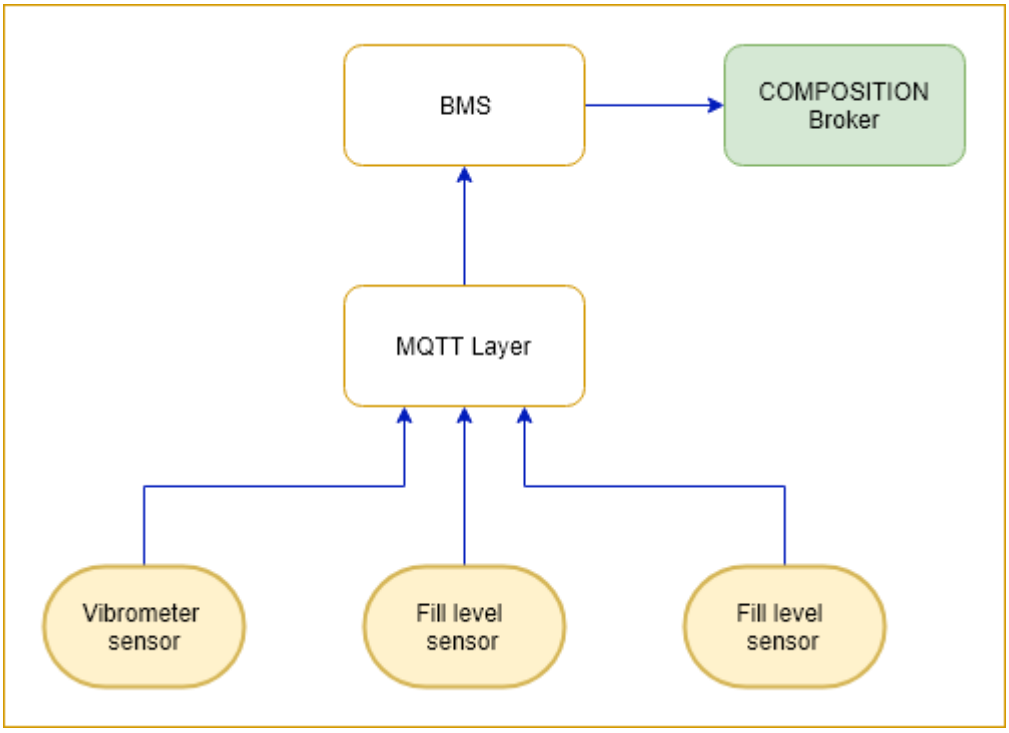

Figure 11: KLEEMANN pilot deployment

In the first use case (UC-KLE-4), the BMS receives data from two vibrometer sensors, located on the machine. Such sensors periodically publish a collection of acceleration measurements that has been read within a time interval. They publish a separate stream for each axis (*x*, *y*, *z*), marking each of them with a different identifier. Then the BMS gets the message, processes the data, transforms it into the OCG Sensor Things format, stores it in the Storage Handler and publishes it to the COMPOSITION broker.

This is an example of a vibrometer sensor message received by the BMS:

```
{
  "hardwareId":"c8f25dc4-5cef-4d3c-af76-8d167b1a7c7c",
  "type": "DeviceMeasurements",
  "request": {
```

```
   "metadata": { "z": "7.24, 7.24, 7.24, 7.28, 7.28, 7.28, 7.28, 7.39, 7.39, 7.39,
7.39, 7.47, 7.47, 7.47, 7.47, 7.51, 7.51, 7.51, 7.24, 7.24, 7.24, 7.24, 7.13, 7.13,
7.13, 7.13, 7.24, 7.24, 7.24, 7.24, 7.43, 7.43, 7.43, 7.43, 7.32, 7.32, 7.32, 7.35,
7.35, 7.35, 7.35, 7.35, 7.35, 7.35, 7.35, 7.35, 7.35, 7.35, 7.39, 7.39, 7.39,
7.24, 7.24, 7.24, 7.24, 7.01, 7.01, 7.01, 7.01, 7.16, 7.16, 7.16, 7.16, 7.39, 7.39,
7.39, 7.39, 7.28, 7.28, 7.28, 7.32, 7.32, 7.32, 7.32, 7.39, 7.39, 7.39, 7.39, 7.43,
7.43, 7.43, 7.43, 7.55, 7.55, 7.55, 7.47, 7.47, 7.47, 7.47, 7.47, 7.47, 7.47, 7.47,
7.66, 7.66, 7.66, 7.66, 7.66, 7.66, 7.66, 7.58, 7.58, 7.58, 7.58, 7.66, 7.66, 7.66,
7.66, 7.74, 7.74, 7.74, 7.74, 7.78, 7.78, 7.78, 7.78, 7.55, 7.55, 7.55, 7.55, 7.55,
7.55, 7.55, 7.39, 7.39, 7.39, 7.39, 7.58, 7.58, 7.58, 7.58, 7.74, 7.74, 7.74, 7.35,
7.35, 7.35, 7.35, 7.20, 7.20, 7.20, 7.20, 7.47, 7.47, 7.47, 7.47, 7.51, 7.51, 7.51,
7.51, 7.24, 7.24, 7.24, 7.39, 7.39, 7.39, 7.39, 7.43, 7.43, 7.43, 7.43, 7.51, 7.51,
7.51, 7.51, 7.62, 7.62, 7.62, 7.47, 7.47, 7.47, 7.47, 7.35, 7.35, 7.35, 7.35, 7.55,
7.55, 7.55, 7.55, 7.62, 7.62, 7.62, 7.62, 7.62, 7.47, 7.47, 7.24, 7.24, 7.24, 7.24,
7.24, 7.24, 7.24, 7.24, 7.35, 7.35, 7.35, 7.35, 7.28, 7.28, 7.28, 7.20, 7.20, 7.20,
7.20, 7.13, 7.13, 7.13, 7.13, 7.24, 7.24, 7.24, 7.24, 7.39, 7.39, 7.39, 7.20, 7.20,
7.20, 7.20, 6.97, 6.97, 6.97, 6.97, 7.20, 7.20, 7.20, 7.20, 7.32, 7.32, 7.32, 7.32,
7.09, 7.09, 7.09, 7.24, 7.24, 7.24, 7.24, 7.20, 7.20, 7.20, 7.20, 7.28, 7.28, 7.28,
7.28, 7.24, 7.24, 7.24, 7.20, 7.20, 7.20, 7.20, 7.16, 7.16, 7.16, 7.16, 7.35, 7.35,
7.35, 7.35, 7.35, 7.35, 7.35, 7.09, 7.09, 7.09, 7.09, 7.20, 7.20, 7.20, 7.20, 7.16,
7.16, 7.16, 7.16, 7.20, 7.20, 7.20, 7.20, 7.24, 7.24, 7.24, 7.28, 7.28, 7.28, 7.28,
7.13, 7.13, 7.13, 7.13, 7.28, 7.28, 7.28, 7.28, 7.32, 7.32, 7.32, 7.13, 7.13, 7.13,
7.13, 7.09, 7.09, 7.09, 7.09, 7.35, 7.35, 7.35, 7.35, 7.32, 7.32, 7.32, 7.32, 7.16,
7.16, 7.16, 7.16, 7.16, 7.16, 7.16, 7.01, 7.01, 7.01, 7.01, 7.05, 7.05, 7.05, 7.05,
7.20, 7.20, 7.20, 7.13, 7.13, 7.13, 7.13, 6.90, 6.90, 6.90, 6.90, 7.09, 7.09, 7.09,
7.09, 7.13, 7.13, 7.13, 7.13, 7.13, 6.90, 6.90, 6.93, 6.93, 6.93, 6.93, 6.93, 6.93,
6.93, 6.93, 7.01, 7.01, 7.01, 7.01, 7.01, 7.01, 7.01, 6.90, 6.90, 6.90, 6.90, 6.78,
6.78, 6.78, 6.78, 6.97, 6.97, 6.97, 6.97, 7.09, 7.09, 7.09, 6.90, 6.90, 6.90, 6.90,
6.86, 6.86, 6.86, 6.86, 6.97, 6.97, 6.97, 6.97, 6.93, 6.93, 6.93, 6.93, 6.90, 6.90,
6.90, 6.93, 6.93, 6.93, 6.93, 6.82, 6.82, 6.82, 6.82, 7.01, 7.01, 7.01, 7.01, 7.05,
7.05, 7.05, 6.74, 6.74, 6.74, 6.74, 6.70, 6.70, 6.70, 6.70, 7.01, 7.01, 7.01, 7.01,
7.05, 7.05, 7.05, 7.05, 6.97, 6.97, 6.97, 6.97, 6.97, 6.97, 6.97, 6.82, 6.82, 6.82,
6.82, 6.93, 6.93, 6.93, 6.93, 7.05, 7.05, 7.05, 6.93, 6.93, 6.93, 6.93, 6.78, 6.78,
6.78, 6.78, 6.93, 6.93, 6.93, 6.93, 7.05, 7.05, 7.05, 7.05, 6.82, 6.82, 6.82, 6.90,
6.90, 6.90, 6.90, 6.97, 6.97, 6.97, 6.97, 6.97, 6.97, 6.97, 6.97, 7.13, 7.13, 7.13,
6.90, 6.90, 6.90, 6.90, 6.70, 6.70, 6.70, 6.70, 6.93, 6.93, 6.93, 6.93, 7.09, 7.09"
   },
   "updateState": true,
   "eventDate":"2018-01-17T14:56:47Z"
  }
}
```


Figure 12: Example of vibrometer stream

In the other two KLE use cases (UC-KLE-1, UC-KLE-3), the BMS gets messages from fill level sensors: in UC-KLE-1 there are three sensors installed on big containers for metal waste, while in UC-KLE-3 there are fourteen sensors installed on internal recycle bins.
The fill level sensors publish:

- the filling level value, expressed in percentage of filling
- the sensor battery level
- an error code (only UC-KLE-3)

The data collection works the same as explained for UC-KLE-4: each sensor publishes its own data stream, specifying its *Id*. Then the BMS processes the message, stores the data in the Storage Handler and publishes it to the COMPOSITION broker.

This is an example of a fill level sensor message received by the BMS:


```
{
  "datastream": {
    "@iot.id": "ds_7-10",
    "sensor": {
      "@iot.id": "7-10"
    }
  },
  "parameters": [
    {
      "battery": 97.0
    },
    {
      "error": 0.0
    }
  ],
  "phenomenonTime": "2019-04-29T12:41:36+0000",
  "result": 68.0,
  "resultTime": "2019-04-29T12:41:34+0000",
  "id": "9a4f35a7-aee3-48a4-a35f-c9ea082332a3"
}
```


Figure 13: Example of fill level sensor datastream (UC-KLE-3)


```
{
  "datastream": {
    "@iot.id": "ds_7-10",
    "sensor": {
      "@iot.id": "7-10"
    }
  },
  "parameters": [
    {
      "battery":97.0
    }
  ],
  "phenomenonTime": "2019-04-29T12:41:36+0000",
  "result": 68.0,
  "resultTime": "2019-04-29T12:41:34+0000",
  "id": "9a4f35a7-aee3-48a4-a35f-c9ea082332a3"
}
```


Figure 14: Example of fill level sensor datastream (UC-KLE-1)

6.3.2 BSL Pilot

BSL pilot provides different data sources to the IIMS: they can be logs from legacy systems or COMPOSITION developed sensor readings. The generic BMS workflow has been modified with respect to its usual behaviour, to match company privacy policies. Since BSL doesn't allow any external system to connect to their network, it was not possible to directly read data neither from legacy systems nor from sensors. All the shop-floor data is serialized into text files and copied to a remote server. A specific software will access afterwards to the remote location and transform such files into an input compatible with the BMS.

The files to be processed in the IIMS are:

- UC-BSL-2: oven log data files, oven event data files, acoustic sensor data files
- UC-BSL-5: equipment monitoring data files

Therefore, such a software component needs to be tailored upon the specific data and file formats that it is going to receive, depending on the source.

The general behaviour is quite simple: the files are copied from BSL local systems to a COMPOSITION server periodically, so that the software can scan the shared directory for new files. All the files are then read line by line, for each line the module parses the information contained and publishes a message on MQTT, pretending to be a sensor.
In this way, all the data reaches the BMS via the same channel, no matter if it is a real sensor or a log data stored in a text file.

An important requirement of the IIMS is that the data, regardless of its source, shall be interrelated. The piece of information included in all the files and that is vital for such processing is the timestamp. Since each value is marked with the timestamp when the reading itself happens, it is easy then for higher layers to correlate events.

The software component that reads the files comes with a dynamic configuration that enables some additional feature than the simple live data flow. It is possible to set parameters in order to replay the old data, simulating specific behaviours or testing specific datasets. For example, the data replay can be sped up or slowed down with respect to the “live” behaviour, or the component can select only partial datasets between specific dates and times, depending on testing needs.

The complete list of features is the following:

- define the period of file folder scan
- define the time period for publishing messages
- publish data between a specific interval (from timestamp / to timestamp)
- continuous replay: the process restarts after sending all the data
- real time simulation: to replicate the same time interval between two data samples
- speed up / slow down the data replay

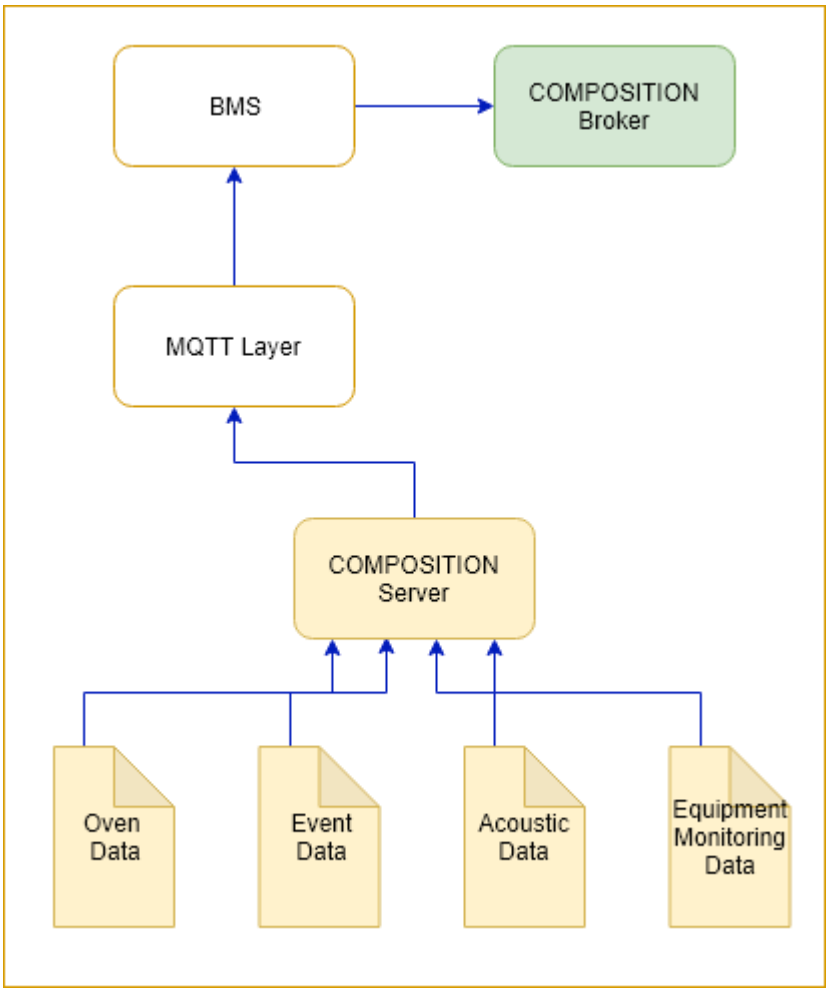

Figure 15: File implementation

As previously mentioned, the files related to UC-BSL-2 are oven data files, event files and acoustic sensor data files. The oven data log files contain present value (real temperature measured by thermocouple, expressed in °C) and output power (output power of solid-state relay, expressed in %) as well as the timestamp associated to such values. Each file includes readings coming from eleven different zones.

The event data files contain notifications on anomalies and occurrences of the ovens. Such alerts are expressed as human readable text, without any formal specification. The number of messages is limited and their formulation is predictable, therefore each event is associated to a numeric code, so that the BMS can publish it as a single reading. In some cases, the current temperature of the oven is specified in such text messages, thus this piece of information is added to the published observation as well.

The acoustic sensor data files are composed as comma-separated values files (.csv). Each file contains 5 values, one for each sensor (there are 5 sensors deployed, in total) and they represent the intensity of the fan noise (expressed in dB).

The files related to UC-BSL-5 are equipment monitoring data sheets, provided by BSL Manufacturing Execution System (MES). Such documents contain information about the actual versus target production for each piece of equipment per hour throughout a build plan and the equipment status. This enables managers to keep track of equipment issues and be informed instantly on changes in equipment status.

An example of this data is shown in Table 3.

Table 3: Equipment monitoring sheet example

	Name	Purpose	MES file column
1.	Container	To track individual circuits	Container
2.	Status	What is the status of each individual circuit	Container Status
3.	Material	Type of material	Material
4.	Product	Which product	Material Description
5.	Mix	Mix of what is being built	Model
6.	Batch	Batch number	SAP Batch
7.	Final Qty	This will allow us to determine is sign off is = final confirmed quantity.	Final Confirmed Qty
8.	SLA	SLA (service level agreement) – if we have a service level agreement – how long is it taking to go through ZPK1 – Serialized – Production ZPP1 – Batch ZRD3 – SWR ZRD1 – SWR built as production equivalent	Production Order Type
9.	SWR	SWR (service work requirement) number	SWR #
10.	Work Cell	Current Work cell – where in the line the product is	Tasklist Completion Work Cell
11.	Task	MI (Manufacturing instruction) Step	Task List
12.	Task Description	MI Step	Tasklist Desc
13.	Sign-off	Sign off time	Tasklist Signoff Date and Time
14.	Sign off Qty	Number of units signed off	Sign Off Qty
15.	Static Days	Current static WIP (work in progress) - Based off the last MES sign off – if it hasn't processed through	Static Days
16.	Line	Which line the product is	Tasklist Workstation

7 Brokering and Security

The brokering and security functionalities which are provided by the Security Framework:

- Authentication and authorization
- Message transport
- The implementation and deployment for the intrafactory environment is detailed in deliverable D4.5 Prototype of the backing service providing authentication and authorization.
- rabbitmq-auth-backend-http: This plugin provides the ability to the RabbitMQ server to perform authentication and authorisation by making requests to an external http service.

The next figure (Figure 9) presents a high-level overview of the Message Broker and COMPOSITION Security Framework II following its architectural diagram:

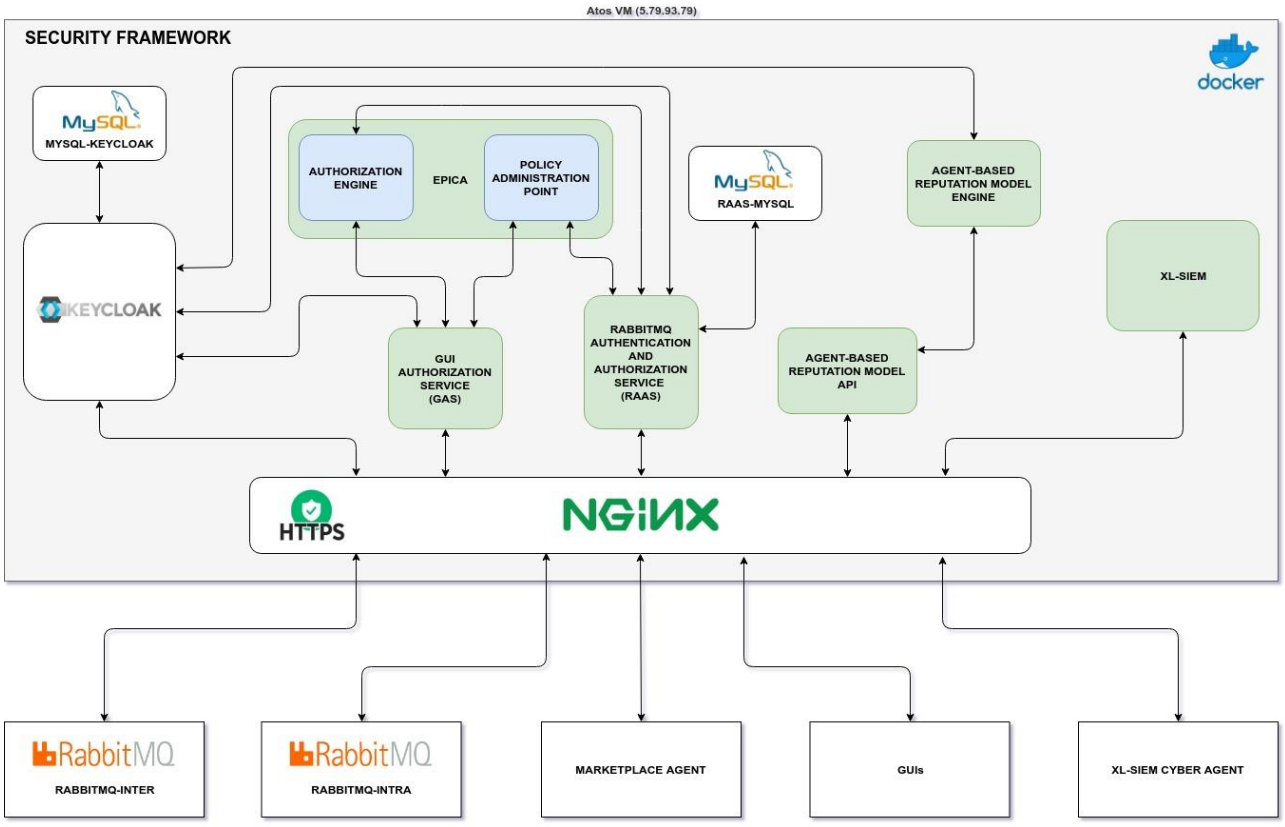

Figure 9: Security Framework architecture diagram

The following sub-sections will focus on the plugins of the message broker to support external authorization and authentication services, a brief description of the COMPOSITION Security Framework services providing authentication and authorization to the message broker and finally a sub-section dedicated to encrypted communication using TLS protocol and the mechanism proposed within COMPOSITION to provide trust on the messages flowing through the message broker.

7.1.1 RabbitMQ Plugins

To be able to make use of COMPOSITION Security Framework authentication and authorization services the following plugins have been deployed, enabled and configured in the broker server.

7.1.1.1 rabbitmq-auth-backend-http

This plugin provides the ability to the broker server to perform authentication and authorisation by making requests to the RAAS HTTP service (see Section 7.1.2). The following configuration snippet shows an

example of how the message broker has been configured to make use of RAAS deployed for Intrafactory scenarios.

```
[
 {rabbit,[{auth_backends, [rabbit_auth_backend_http, rabbit_auth_backend_internal]}]},
 {rabbitmq_auth_backend_http,
  [{http_method,  post},
  {user_path,   "http://172.80.0.5:3000/auth/user"},
  {vhost_path,  "http://172.80.0.5:3000/auth/vhost"},
  {resource_path, "http://172.80.0.5:3000/auth/resource"},
  {topic_path,  "http://172.80.0.5:3000/auth/topic"}]}
].
```

7.1.1.2 rabbitmq_auth_backend_cache

This plugin provides a way to cache authentication and authorization backend results for a configurable amount of time reducing the amount of load on the RAAS http server providing authentication and authorization. The following snippet shows how to configure the message broker to use cache plugin. In this case authentication and authorization backend results are cached for 5 seconds.

```
[{rabbitmq_auth_backend_cache,
      [{cached_backend, rabbit_auth_backend_http}, {cache_ttl, 5000}]
}].
```

7.1.2 Message Broker Authentication/Authorization Service – RAAS

This component is an http service which is being developed as part of the Security Framework whose task is enabling the use of the Authentication (Keycloak) and Authorization (EPICA) services by the Message Broker (RabbitMQ). This service exposes the following end-points:

- /auth/user
- /auth/vhost,
- /auth/resource
- /auth/topic

RAAS will be able to work in two modes:

1. RAAS will be responsible for requesting and management of tokens from the Authentication service (Keycloak) and perform authorization request to Authorization service (EPICA) with the obtained tokens. The clients login to the message broker with username and password.
2. RAAS will be only responsible for the verification and validity of tokens from Authentication service (Keycloak) and perform authorization request to Authorization service (EPICA) with the provided tokens. The clients are responsible to obtain and manage the authentication tokens and provide them to RAAS. The clients login to the message broker with the token from Authentication service, no password involved in this mode.

For more information, related to this component, refer to D4.1 Design of the Security Framework I - Section 4.3 due on M12 and D4.2 Design of the Security Framework II due on M18

7.1.3 Authentication – Keycloak

This Security Framework component is responsible for providing the authentication mechanisms for users, applications, services and devices. It supports the following standard authentication protocols:

- OAuth 2.0: Industry-standard protocol for authorization. Makes heavy use of the JSON Web Token (JWT) set of standards.

- Open ID Connect (OIDC): Authentication protocol based on OAuth 2.0. Unlike OAuth 2.0 OIDC is an authentication and authorization protocol.
- SAML 2.0: Authentication protocol similar to OIDC. It relies on the exchange of XML documents between the authentication server and the application.

For more information, related to this component, refer to D4.1 Design of the Security Framework I - Section 4.1 due on M12 and D4.2 Design of the Security Framework II due on M18

7.1.4 Authorization – EPICA

This component is responsible for providing authorization mechanisms. It´s based on XACML v3.0 which provides an attribute-based access control mechanism and provides the means to define authorization policies used to protect resources. Any request to access a protected resource will first be evaluated against the defined policies and the evaluation result will be enforced depending on the outcome.

For more information, related to this component, refer to D4.1 Design of the Security Framework I - Section 4.2 due on M12 and D4.2 Design of the Security Framework II due on M18

7.2 Message Transport

This section describes the use of TLS (Transport Layer Security) encryption protocol to provide secure communication with the broker and the use of JSON Web Signature standard to sign the messages flowing within COMPOSITION.

7.2.1 Encryption

COMPOSITION Message Broker (RabbitMQ) is configured to make use of TLS[9] (Transport Layer Security) encryption protocol on the communication protocols AMPQ[10] and MQTT[11]. Non-secured communications over these protocols have been disabled as well as all non-secured ways of communication with the broker, like the RabbitMQ management UI.

7.2.2 Signature

All messages flowing within COMPOSITION should be signed using JSON Web Signature[12] (JWS) standard.

JWS represents signed content using JSON data structures and base64-url-encoding, the representation consists of three parts:

- Header: describes the signature method and parameters employed
- Payload: message content to be secured
- Signature: ensures the integrity of both the Header and the Payload

The three parts are base64-url-encoded for transmission, and are typically represented as the concatenation of the encoded strings in that order, with the three strings being separated by period ('.') characters.

The following figure (Figure 10) shows the representation of a JWS:

[9] https://tools.ietf.org/html/rfc5246
[10] https://www.amqp.org/
[11] http://mqtt.org/
[12] https://tools.ietf.org/html/rfc7515

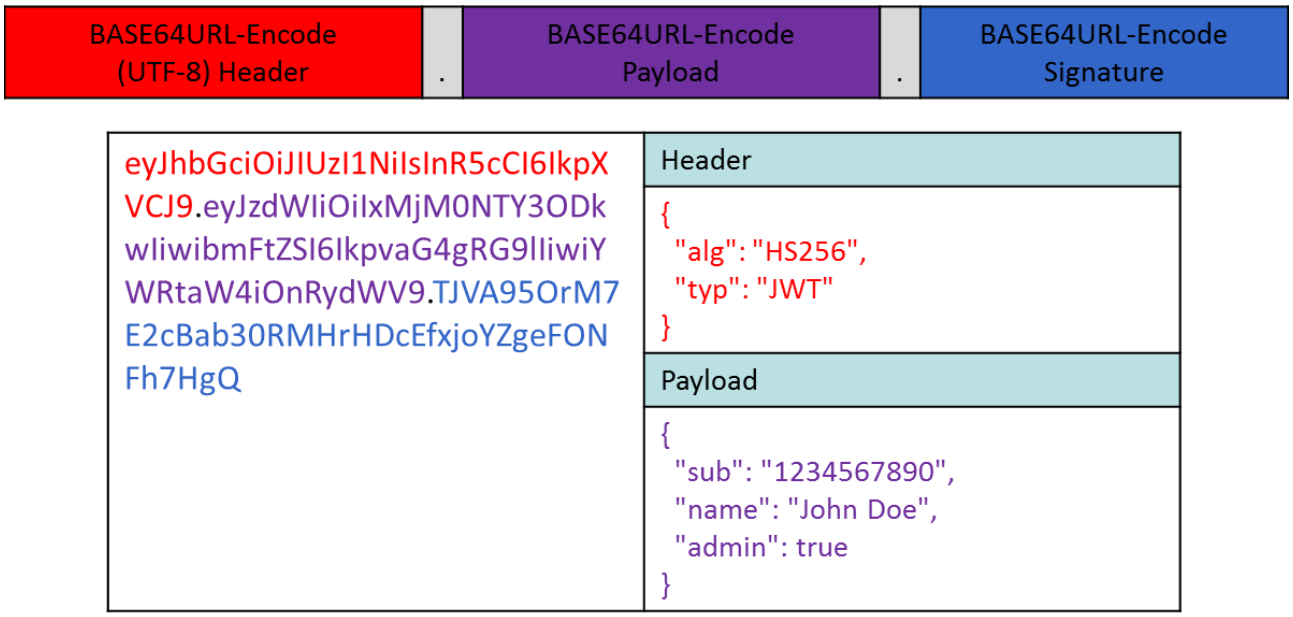

Figure 10: JWS representation

8 Distributed Intrafactory Notification Enabling Service

In the COMPOSITION ecosystem, the Intrafactory Interoperability acts as a centralized layer to exchange heterogeneous messages among sensors and components through common interfaces.

These messages are designed for machine-readable communications through the OCG Sensor Things data format described in section 4.5. When opportunely formatted and delivered to the key personnel, who can handle them best, this information can be a significant support for decision-making and increase the situational awareness.

In these regards, the COMPOSITION architecture already foresees components and HMI to provide easy access to both data and functionalities needed by individual users in their particular use cases. Unfortunately, this approach involves an a priori categorization of actions and steps, required for the interactions between the actors and the system. On the other hand, a ubiquitous notification system would not have the necessary flexibility to discriminate properly between the processed events, resulting in sending too many redundant notifications with the expected negative effects on workers daily activities.

The need of a more customizable and condition-based asynchronous distribution mechanism has driven to the development of the Intra Factory Notification Service. This service extends the LinkSmart IoT Agent[13] to provide email notification functionalities. More details about the LinkSmart Agent architecture are available in Deliverable 5.1 “Big data mining and analytics tools I”. The figure below depicts the overall architecture of the notification service itself:

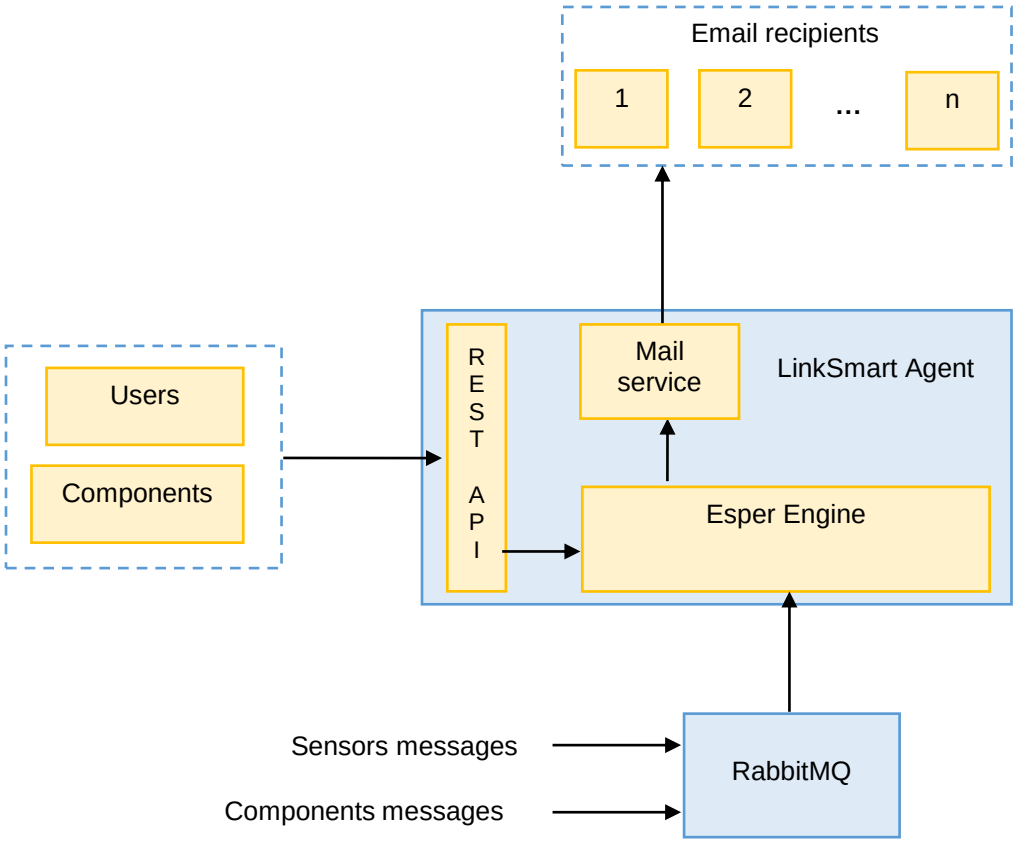

Figure 11: Notification service architecture

This LinkSmart IoT Agent allows email notifications to the specified recipients, based on custom rules defined by users and intrafactory components. These rules, described as continuous query, called

[13] https://docs.linksmart.eu/display/LA/IoT+Agents

statements, are provided to the agent through the API described in section 8.2. The agent behaves by storing provided queries and runs them, automatically and periodically, as live data is collected through the RabbitMQ broker. Section 8.1 describes the statements data format.

After any significant changes on the live data, the agent determines which statement is affected and if a notification has to be triggered. The process is based on the Esper engine[14] a software for Complex Event Processing (CEP) and streaming analytics. It enables rapid development of applications that process large volumes of incoming messages or events, regardless of whether incoming messages are historical or real-time in nature. In particular, it is designed for high-rate events exchange scenarios where is extremely difficult, if not impossible, to store and later query the events using classical database architecture. Esper filters and analyses these events in various ways, and respond to conditions of interest.

The Notification Service receives by the Esper engine the results of each statement that match the specified criteria and ensures their effective delivery to the selected email recipients as soon as the engine processes the events, reducing the redundancies of the exchanged messages.

The flexibility provided by the adoption of dynamic statements allows the system to better fit the punctual needs of clients. Moreover, the service can easily integrate, without any modification, new OGC compliant components connected to the Intrafactory Interoperability Layer, therefore minimizing the effort required to develop specific notifications solutions.

8.1 Statements Data Format

The Notification Service notifies emails to specific recipients based on triggered user provided rules expressed as a single JSON object. This object borrows most of its syntax from the default schema defined for the LinkSmart Agent[15], changing the meaning of some fields to better cope with the new features provided by the service.

The table below summarize the available JSON fields and their descriptions:

Table 4: Message payload format

Field	Description
name	It is the unique name of the rule. Normally, the subject of each email sent contains this information.
statement	The Event Processing Language (EPL) statement.
CEHandler	Defines which handler will manage the result of the event. This parameter value must be: eu.linksmart.services.event.handler.MailEventHandler.
output	Contains the list of email addresses of the output recipients where the query results will be sent.
scope	Contains the SMTP mail server settings.

The statements are defined through the Event Processing Language[16]. It is a declarative language for dealing with high frequency time-based event data derived from SQL-language and offering SELECT, FROM, WHERE, GROUP BY, HAVING and ORDER BY clauses. In COMPOSITION project OGC Datastreams replace tables as the source of data with OGC Observation, received though MQTT, replacing rows as the basic unit of data. Since events are composed of data, the SQL concepts of correlation through joins, filtering and aggregation through grouping can be effectively leveraged.

[14] http://www.espertech.com/esper/
[15] https://docs.linksmart.eu/pages/viewpage.action?pageId=3145798
[16] http://esper.espertech.com/release-5.2.0/esper-reference/html/epl_clauses.html

An example follows:

```
{
  "name":"Sensor value above limit",
  "statement":"select id,result from Observation.win:time_batch(1 sec).std:unique(id)
              as obs where obs.datastream.id=123 and cast(obs.result,double)<50",
  "CEHandler":"eu.linksmart.services.event.handler.MailEventHandler",
  "output":["snapshots@composition.com","admin@composition.com"],
  "scope":["server=smtp.host.com, port=587"]
}
```

In this example, when a sensor Observation published on DataStream with id 123 falls below a certain threshold (50 in the example), an event is triggered and a mail message is sent out to the predefined email recipients.

8.2 LinkSmart Agent Interfaces

The Notification Service infrastructure functionalities are made available through the LinkSmart Agent REST-based interface[17]. Each client can invoke these services by using a combination of resource identifiers and HTTP methods, hence, exchanging messages containing the textual JSON representations of the desired statements. The messages format is described in the previous section.

The RESTful interface provides the four basic functions of CRUD (Create, Retrieve, Update and Delete) summarized in the table below:

Table 5: RESTful interface functions

Operation	HTTP	URL	Description
Create	POST	http(s)://host:port/statement	Add a new statement to the agent collection
Read	GET	http(s)://host:port/statement or http(s)://host:port/statement/id	Retrieves the representation of all the statements in the collection. The details of a single statement can be retrieved by providing its identifier
Update	PUT	http(s)://host:port/statement/id	Replaces the addressed member of the collection
Delete	DELETE	http(s)://host:port/statement/id	Deletes the addressed member of the collection

[17] https://docs.linksmart.eu/pages/viewpage.action?pageId=3145795#IoTData-ProcessingAgentAPI(underrevision)-HTTP-RESTAPI

9 Intra-Inter Factory Communication

In this section it will describe the communication that occurs between a component present in the Intrafactory and one in the Interfactory. The ELDIA-1 use case will be used as an example to analyze such communication in the proceeding section. This use case has been developed using two main components, the Deep Learning Toolkit (DLT) described in the deliverable 5.4 and the Agent-Based Marketplace, described in section 9.1. The DLT is a component that runs within the Intrafactory borders meanwhile the Marketplace is one of the components of the Interfactory environment.
The use case ELDIA-1 can be defined as: The Deep Learning Toolkit component is expected to distribute the latest prediction on the price per ton at which specific commercial partners are likely to accept to buy/sell scrap metal (and other goods) within a fixed timeframe in the future. This information in the form of predictions, are intended to support the agent intelligence in order to improve the decision system that is in charge of accept/emit commercial offers about scrap metal (extended to other goods).

9.1 Agent-Based Marketplace

In this section it will be described the non-WP5 component called “Agent-Based-Marketplace”.
The COMPOSITION Marketplace is a fully distributed multi-agent system designed to support industry 4.0 exchanges between involved stakeholders. It is particularly aimed at supporting automatic supply chain formation and negotiation of goods/data exchanges. The COMPOSITION marketplace exploits a microservice architecture and relies upon a scalable messaging infrastructure. Trust and security are granted in every negotiation step undertaken by automated agents on behalf of involved stakeholders. The Marketplace component includes the following elements: a Marketplace management portal; an easy to deploy Marketplace core based on Docker; a set of "default" agent implementations ready to be adopted by interested stakeholders.

There are two main categories of agents that can be defined a priori, depending on the kind of services provided:

- Marketplace agents:

  the agents providing services that are crucial for the marketplace to operate.

- -Stakeholder agents:

  agents developed and deployed by the marketplace stakeholders to take part in chain formation rounds.

Stakeholder agents can be divided in two different categories, Requester and Supplier. They exploit a common communication protocol to interact with other agents. The protocol defines the logic to successfully complete a negotiation process.

Following is described a simplified version of its architecture:

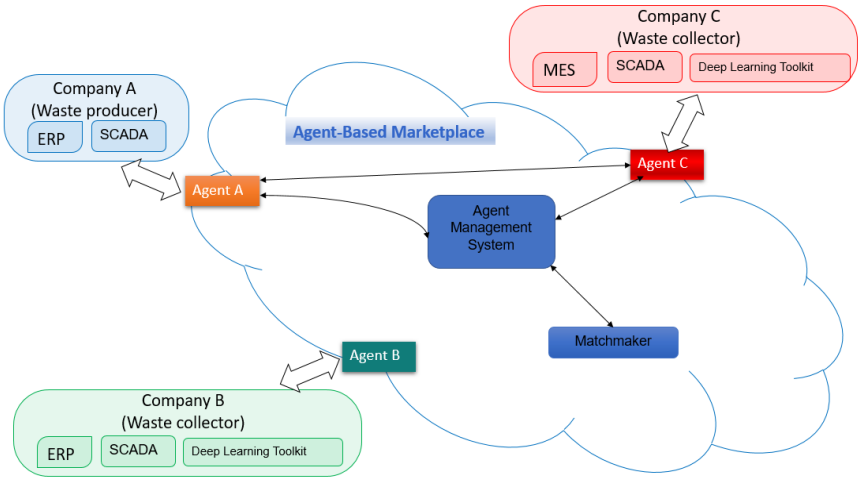

Figure 12: Agent-Based Marketplace Architecture

9.2 Communication Details

In order to have a more secure and contained communication it was necessary to create an ad-hoc channel that will not compromise the modularity and the architecture of the Intrafactory Interoperability Layer. It was then required to create a Peer-to-Peer connection between the Agent, which stands on the Interfactory border, and the DLT which resides on the Intrafactory environment. The aforementioned connection has been obtained using a Virtual Private Network (VPN). Using the previous mentioned technique and static addresses it is possible to create security rules based on the address of the caller and the receiver. The DLT accepts requests that are originated only by the agents and at the same time the agent is able to communicate only with the DLT due to the closed nature of the network. Using the previously described approach it is possible to have a secure communication which does cope with the security by design condition.

The communication is obtained calling REST API using HTTPS with payload formatted as json.

10 Conclusions

In this document it has been highlighted how the Intrafactory Interoperability Layer has been developed, deployed and tested. Each component has been embedded as a Docker container and the interconnection among sub-components tested, in order to provide a reliable communication layer across the entire intrafactory scenario.

A common TRL was previously set to 6, but during the projects' lifecycle, in light of existing used technologies and in correlation to the use case that will require its deployment at the shop-floor level, it has been raised to a more ambitious 7.

During the several months that have been passed from the first iteration of this deliverable D5.9 Intrafactory interoperability layer I, the components have been deeply and continually tested on both of the use case mentioned in the section 6.3.1 and 6.3.2. The components in both of the use cases achieved the expected results regarding latency and lead time. In section 7, it has been described how the "security by design" of the component has been obtained.

For the aforementioned reason it is possible to define that the Intrafactory Interoperability Layer, and the components which is based on, is a robust solution which achieve the expected TRL 7.

11 List of Figures and Tables

11.1 Figures

11.2 Tables