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

D5.9 Intrafactory interoperability layer I

Date: 2018-02-27

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: D5.9 Intrafactory interoperability layer I 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	Paolo Vergori (ISMB)	2018-01-09	Table of Contents
0.2	Paolo Vergori (ISMB)	2018-01-17	Consolidated ToC
0.2.1	Farshid Tavakolizadeh (FIT)	2018-02-05	Added contributions to LinkSmart chapter
0.2.2	Matteo Pardi (NXW)	2018-02-06	Added BMS
0.2.3	Javier Romero (ATOS)	2018-02-12	Added Brokering and Security
0.2.4	Nadir Raimondo (ISMB)	2018-02-14	Added Notification Engine
0.3	Paolo Vergori (ISMB)	2018-02-16	Consolidated contributed version
0.3.1	Paolo Vergori (ISMB)	2018-02-16	Introduction, summaries and conclusions
0.3.2	Farshid Tavakolizadeh (FIT)	2018-02-22	Enhanced contribution to LinkSmart chapter
0.4	Nadir Raimondo (ISMB)	2018-02-22	Integration and internal review
0.5	Paolo Vergori (ISMB)	2018-02-23	First draft out for review
1.0	Paolo Vergori (ISMB)	2018-02-28	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Tracy Brennan (BSL)	2018-02-26	Minor adjusts made to Sections 1, 6 and 9
Willie Lawton (TNI-UCC)	2018-02-27	Good deliverable with minor changes

Index:

1 Executive Summary

The present document named “D5.9 Intrafactory interoperability layer I v1.0” 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 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. This version 1.0, submitted at M18, highlights the results of the first iteration of project’s 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 intra-factory scenarios. It is therefore involved in all intra-factory use cases which will be deployed in all intra-factory pilots at end users’ premises.

Key topic addressed is the intra-factory interoperability among components, namely the Building Management System, LinkSmart, the broker-based message distribution and the notification service.

A second iteration of this document, named “D5.9 Intrafactory interoperability layer II” will be submitted in M34 and will be updated with the results of task “5.5 – Adaptation Layer for Intrafactory interoperability” through an iterative approach.

2 Abbreviations and acronyms

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
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

In this document it is described 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 intra-factory use cases. It also emerged the requirement from the end-users to have 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 component is 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 point in a security by design environment, creating a data stream that through many transformation and adaptations, will let data 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, to their 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 document is structured in eight sections and after a comprehensive analysis of the COMPOSITION's architecture, in relation to the intra-factory interoperability model, each sub-components follows in its own section. In fact, the document progresses with the LinkSmart overview and its usage in the intra-factory scenarios. It then progresses with the presentation of one of the main intra-factory components, ergo the Building Management System. After that, the brokering and the security issues are explained in the second last descriptive section, whereas the final one is left for the notification engine that the Intrafactory Interoperability Layer encompasses.

Furthermore, a detailed report of required work that will be necessary to complete this activity and tackle future challenges is included in the final section.

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 will provide 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 will also be 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 is scalability, extensibility, interoperability and integrated security.

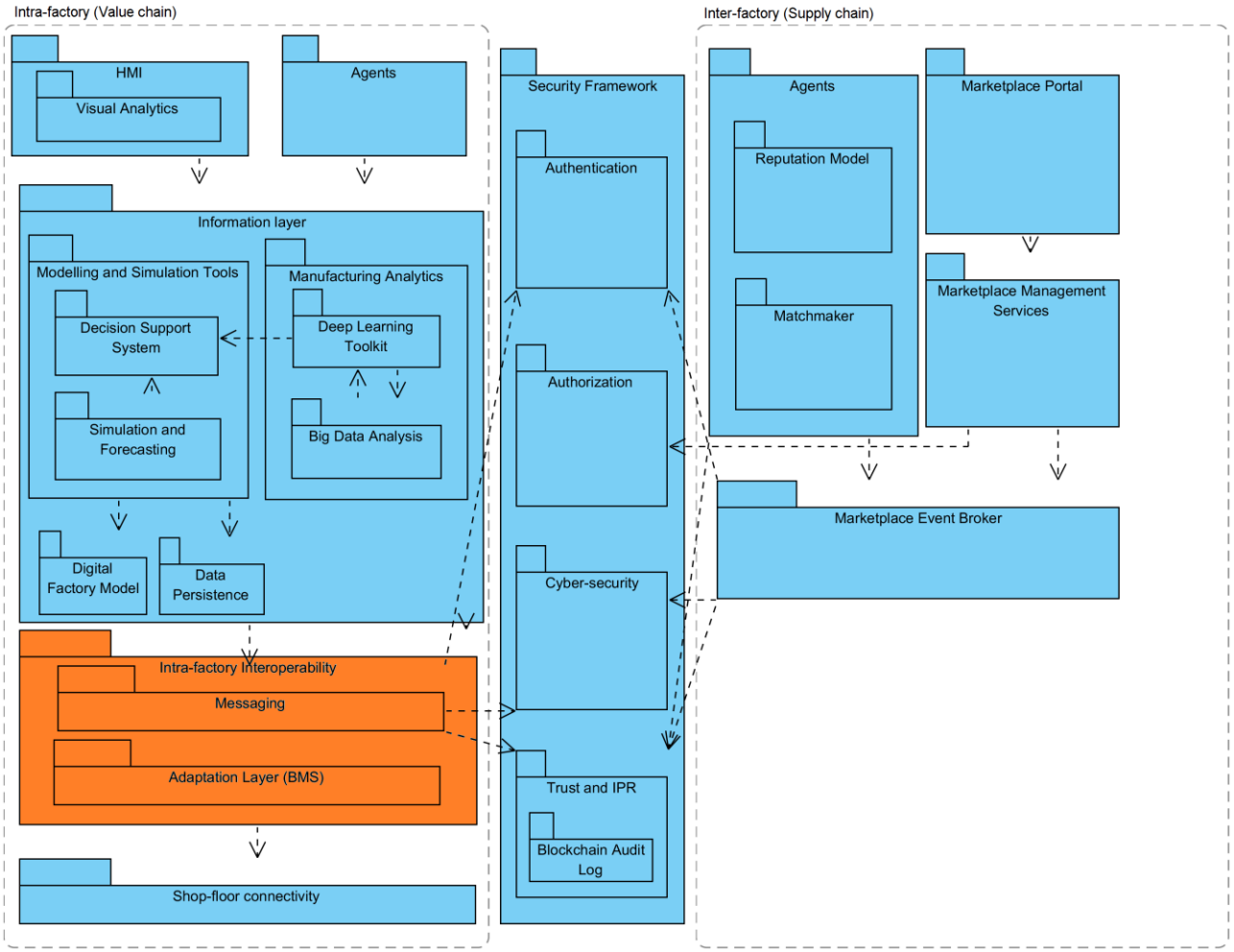

Figure 1: Functional packages of the COMPOSITION system

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

4.2 Functional View

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 intra-factory system, the Message Broker MQTT and REST APIs are used to propagate data. Microservices such as the distributed intra-factory 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.

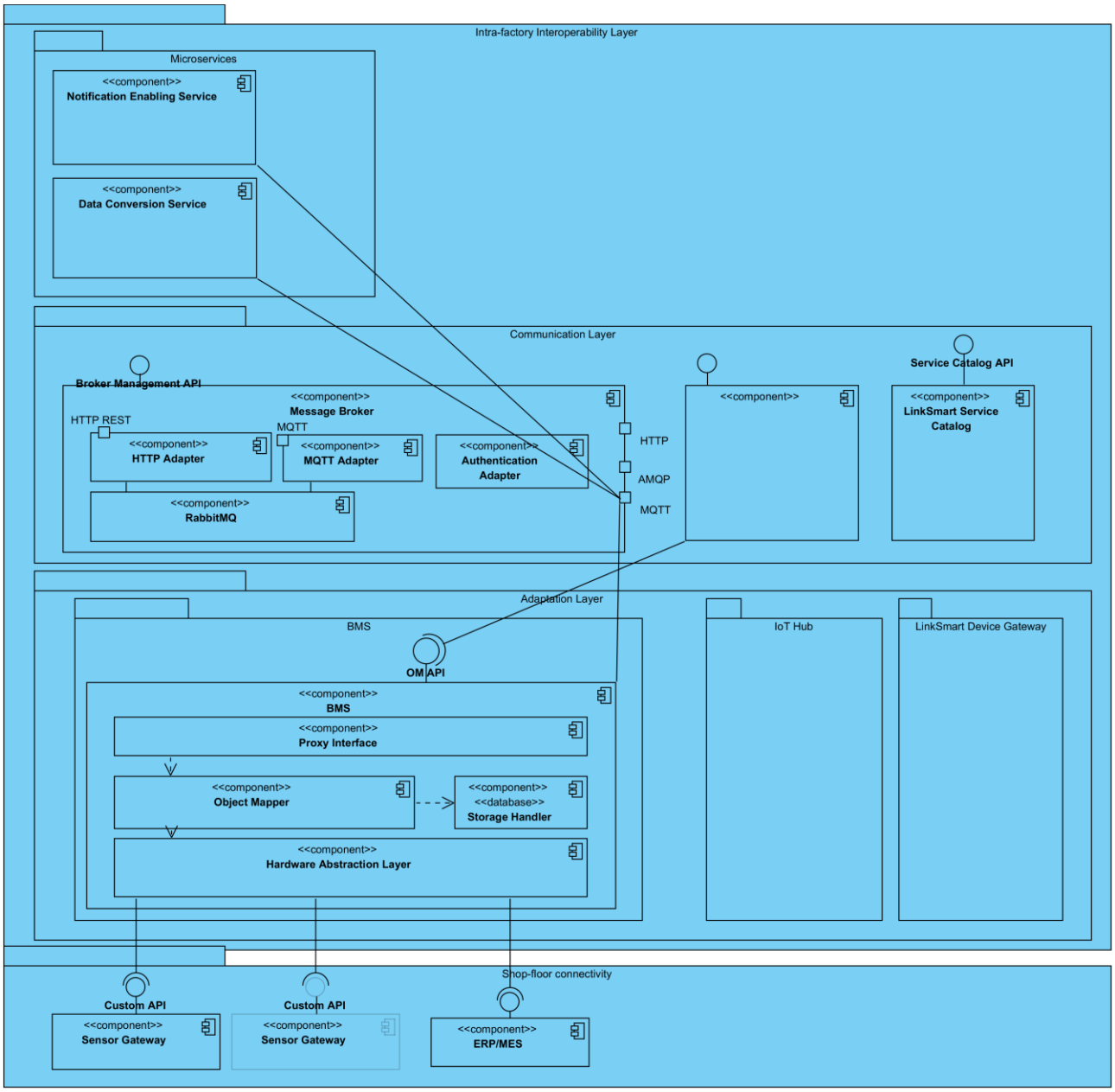

Figure 2: Intrafactory Interoperability Layer and Shop-floor

4.3 Information View

The Digital Factory Model (DFM) (described in D3.2) 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.

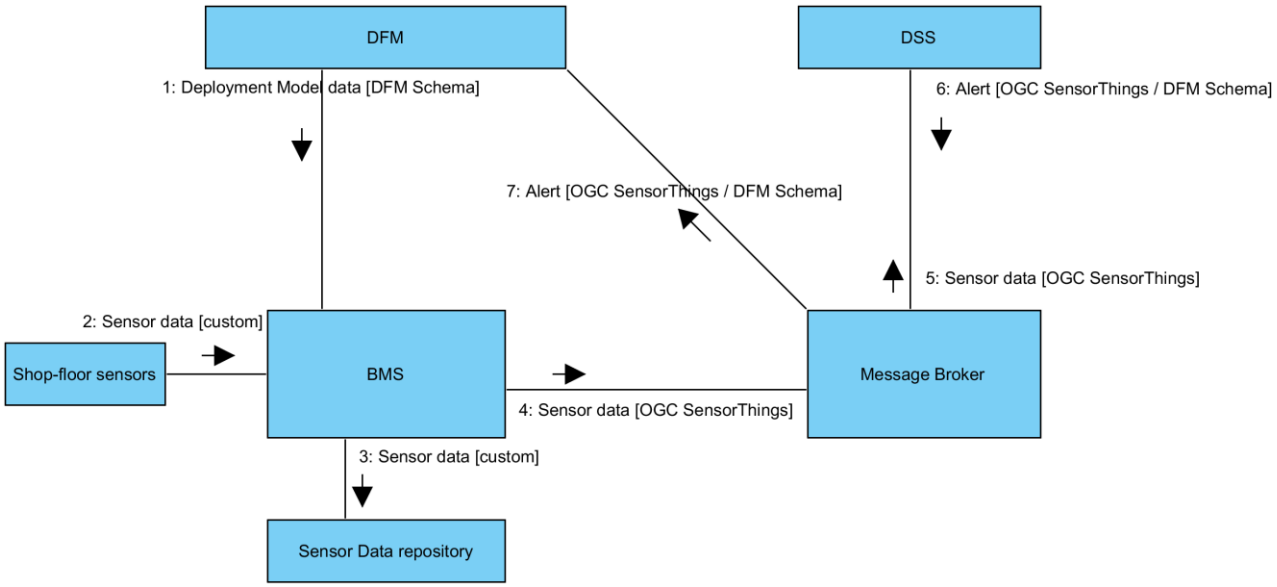

Figure 3: Example data flow

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. Information about the context of the data from the DFM will be added by the BMS Object Mapper. The data will be 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 and will benefit from subscribing to annotated data from a topic where data on an entire process or asset is published. The IoT Agent in the Big Data Analytics (BDA) package may be used to annotate and re-publish data on a MQTT topic structure that includes information from the DFM on e.g. the process involved. Data generated microservices or other system components may also be published on such topics. This mapping from instances in the DFM to FeatureOfInterest in the OCG SensorThings Data Model is not yet developed.

4.3.1 Data Persistence

The BMS provides the Storage Handler for persisting shop-floor data. For data generated in the COMPOSITION system, e.g. the results from predictive maintenance deep learning networks or alerts, the DFM storage is used.

4.3.2 Operational Management

The Commissioning System will be responsible for the configuration management of the COMPOSITION system. This includes the onboarding process for sensors by transferring the device definitions in the DFM Deployment Model to the BMS and other components.

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

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.

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

The message broker and information processing components of the Intrafactory Interoperability Layer will be installed as docker containers in a docker host on a node at the factory or in the cloud. The Security Framework will likely be 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 followings, Figure 4 shows the UML diagram of the entities of the SensorThings API.

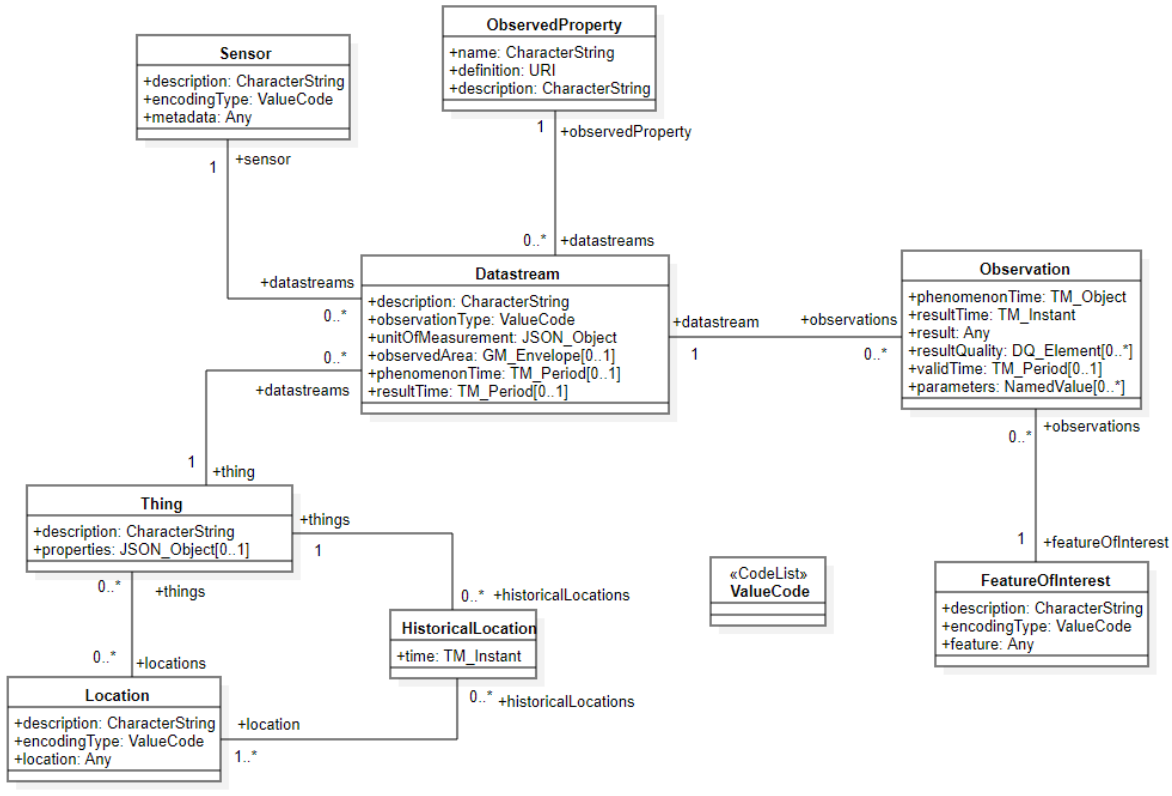

Figure 4: OGC Sensor Things data model

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 fulling 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 on 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 intra-factory 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

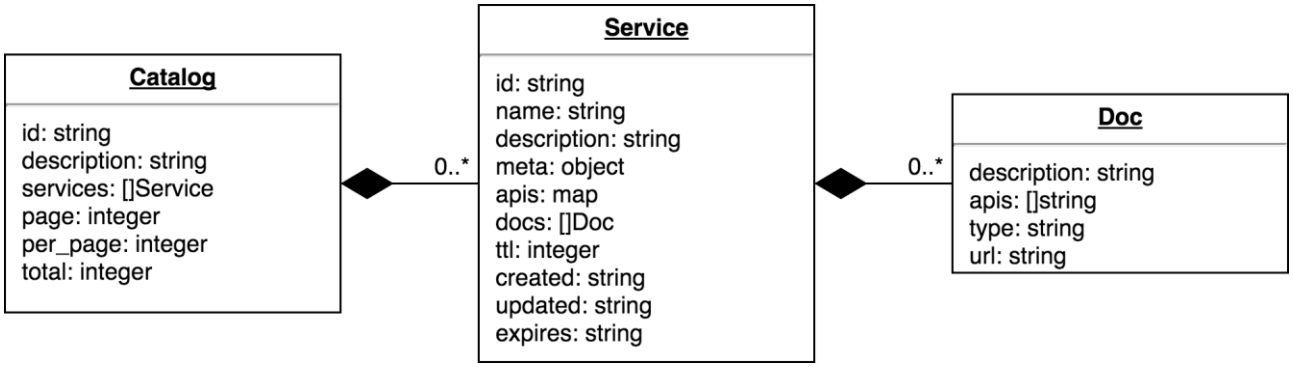

Figure 5. Data model of Service Catalog.

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:

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

Table 1: LinkSmart Service Catalog operations

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

[3] https://docs.linksmart.eu/display/SC/Service+Catalog+API

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

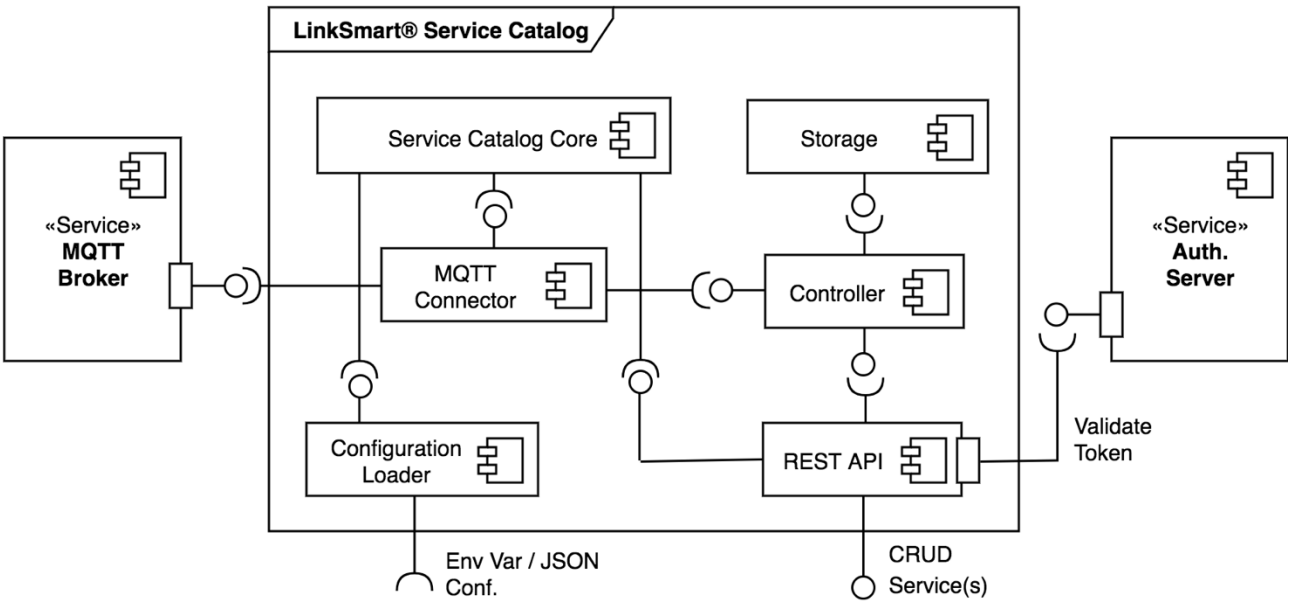

Figure 6. Components of the Service Catalog.

Service Catalog implements the registry in 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].

[4] https://code.linksmart.eu/projects/SC/repos/service-catalog

5.2.3 Continuous Integration and Delivery

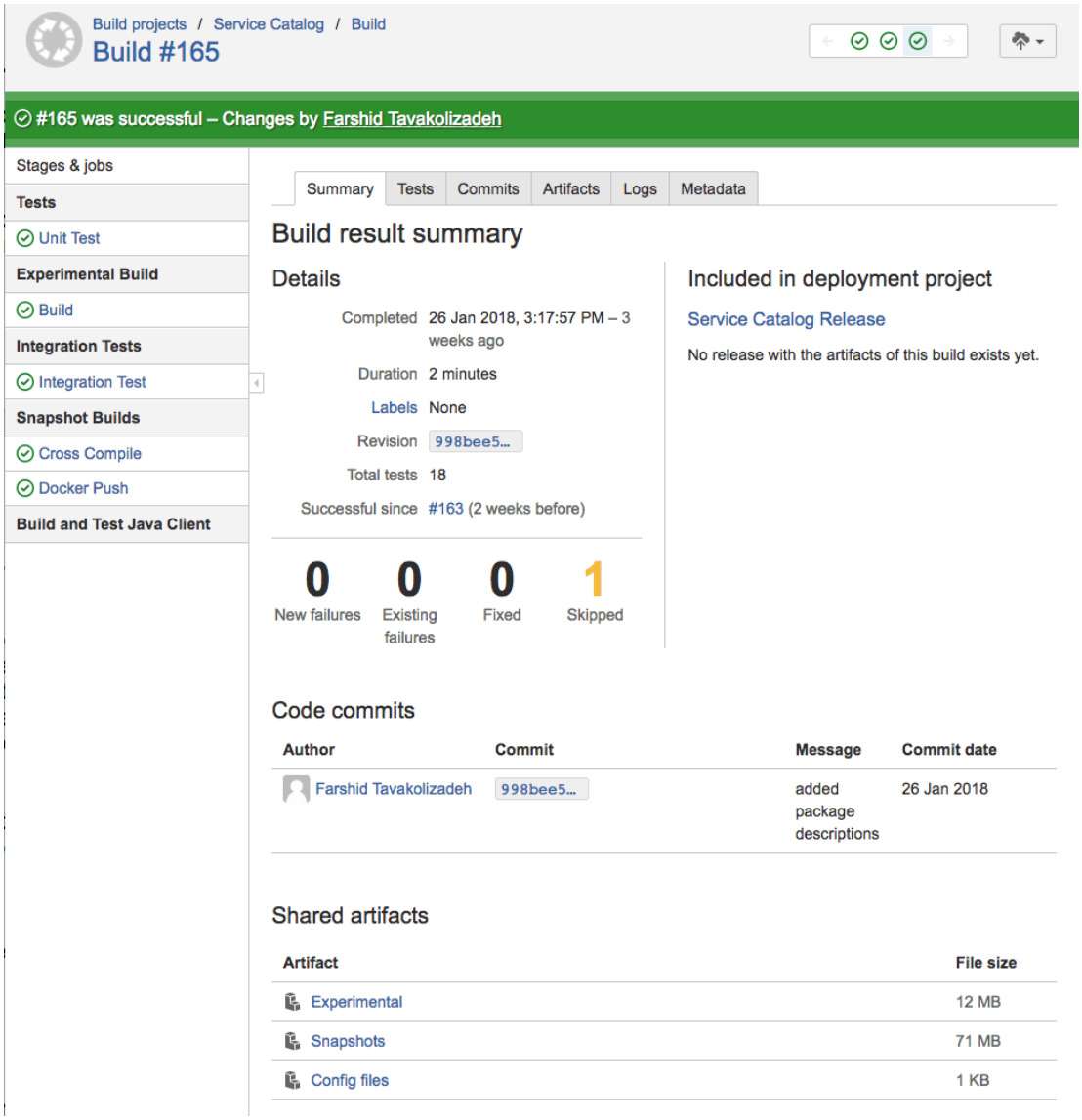

Figure 7. Service Catalog build project.

Service Catalog code is built and tested continuously on a publically 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)
    - dsn is the Data Source Name for storage backend (ignored for memory, "file:///path/to/ldb" for leveldb)

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

- 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 intra-factory. 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.

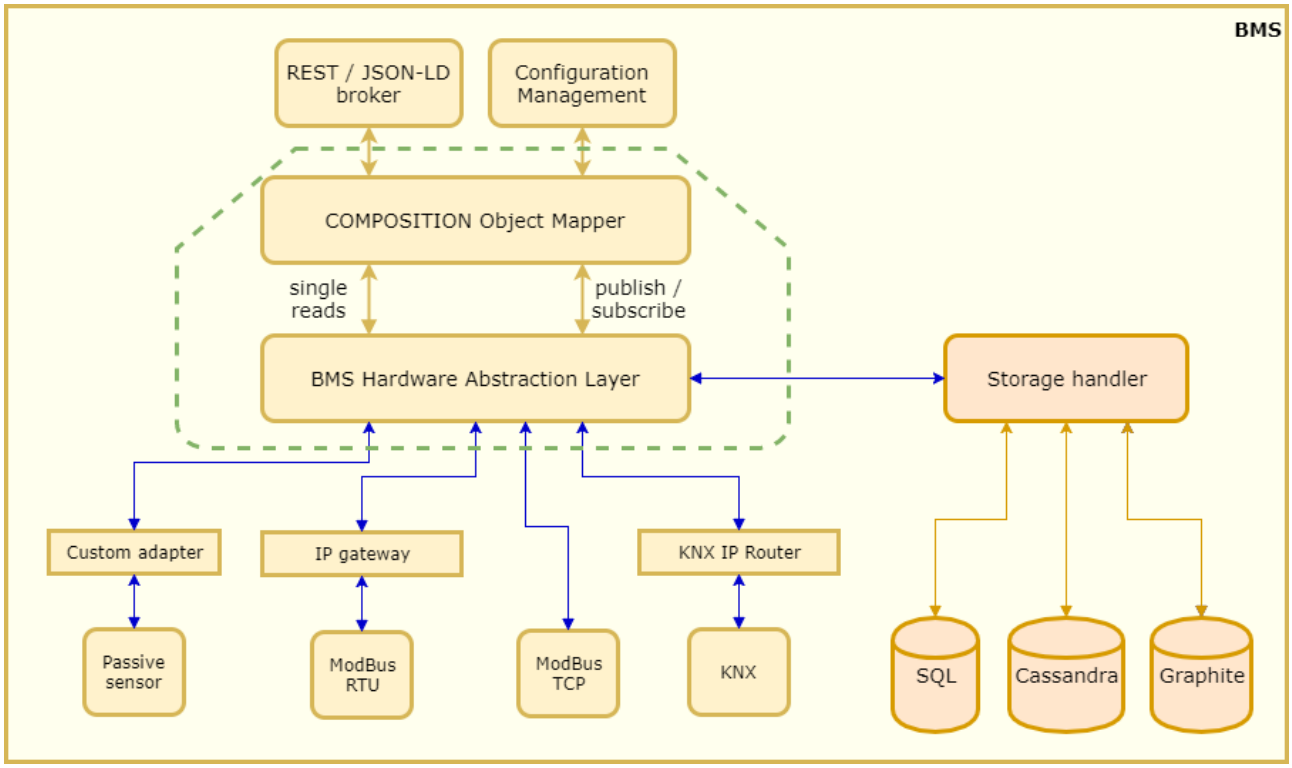

Figure 8: Components on top of the BMS

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, interconnected through different field buses (e.g. KNX, Modbus, BACnet), 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. Regarding to Composition, the HAL component has been enhanced in order to support

communications via MQTT, which is the protocol used by sensors that are going to be deployed in the project use cases (e.g. vibrometer sensor, fill level sensor, etc.).

In general, the HAL exposes a virtualized version of the underlying physical objects to the upper layers, from which information can be read and actuations can be performed. Moreover, in order to be flexible towards the configuration of the integrated devices, the component provides a user interface as well, that is the equivalent of an Administration Shell in the RAMI architecture.

6.2 Object Mapper

Once HAL has received the data from the shop floor, the BMS has been given the access to a set of information, collected into some raw format that is uniform for every sensor (and so for every protocol), but still not ready to be exposed as it is.

The job of the Object Mapper component is to link all this data to all the meta-information of the devices and the sources that have generated it. Therefore, next to the very basic and simple entity that is representing the current sensing, read in a single unit of time by the device, the Object Mapper is in charge of adding the semantics to it. When saying semantics, we refer to all the properties and the related information that is describing what the raw value is representing. This is something that will be presented to the upper layers through the OCG Sensor Things data format as described in Section 4.5.

In this way, the BMS will expose objects which are not only a uniform way to present data, but also a complete description of the different information collected from the shop floor.

6.3 Storage Handler

Automatized production processes produce millions of data in form of events. When it’s possible, in order to extract the most value from the data, these events must be processed in real-time and on demand. Therefore, the data is processed at the moment when it is produced extracting the maximum value, reducing latency, providing reactivity, giving it context, and avoiding the need of archiving unnecessary data. At the same time, some valuable information must also be stored somewhere, in order to be retrieved when necessary, as a historical trace of what has been collected during the process lifetime.

In the first place, as stated above, the BMS provides a set of tools to collect, annotate, filter or aggregate (if needed) the real-time data incoming from the production facilities. This set of tools facilitates the possibility to build applications on top of real-time data. Secondly, through a component called Storage Handler, the BMS provides a repository for valuable information to be kept during the whole machine lifetime. These raw measurements can also be enhanced by providing additional metadata to be attached to them, in case it should become necessary.

7 Brokering and security

7.1 Authentication and Authorization

In order to make use of COMPOSITION Security Framework services for authentication and authorization and thus maintain a centralized point for user management and authorization policies management, RabbitMQ internal authentication and authorization mechanisms have been overridden with the use of the following plugins provided by RabbitMQ:

- 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 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 architecture and interactions between them

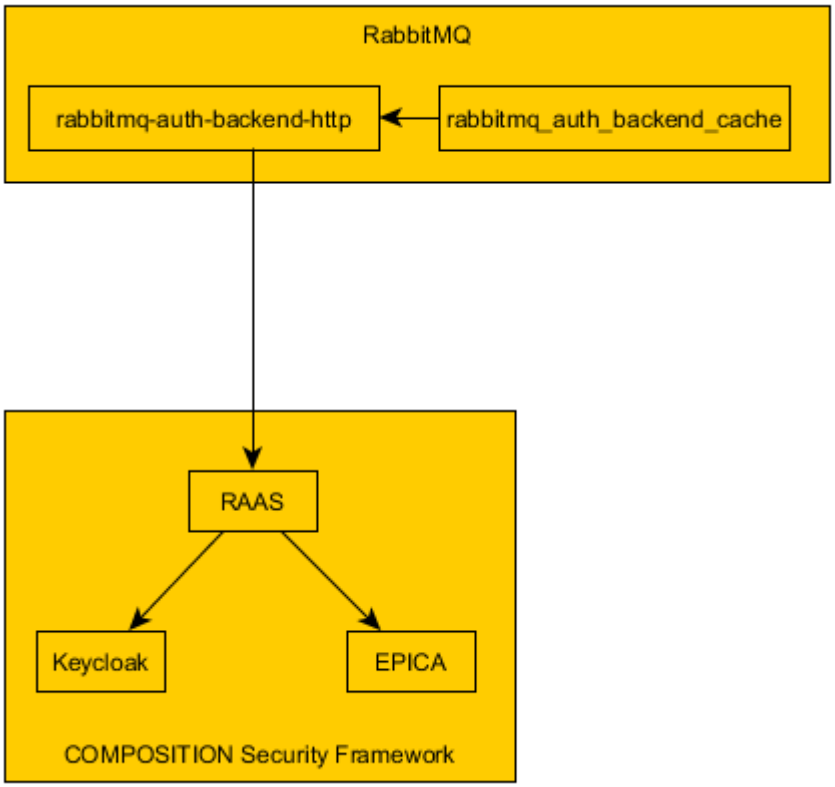

Figure 9: Message Broker - Security Framework architecture overview

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 Intra-Factory 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 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 the responsible to request and manage tokens from Authentication service (Keycloak) and perform authorization request to Authorization service (EPICA) with the obtained tokens. The clients make login in the message broker with username and password.
2. RAAS will be only responsible to verify the 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 make login in 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 of 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[8] (Transport Layer Security) encryption protocol on the communication protocols AMPQ[9] and MQTT[10]. Non-secured communications over these protocols have been disabled as well as all non-secured way 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[11] (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:

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

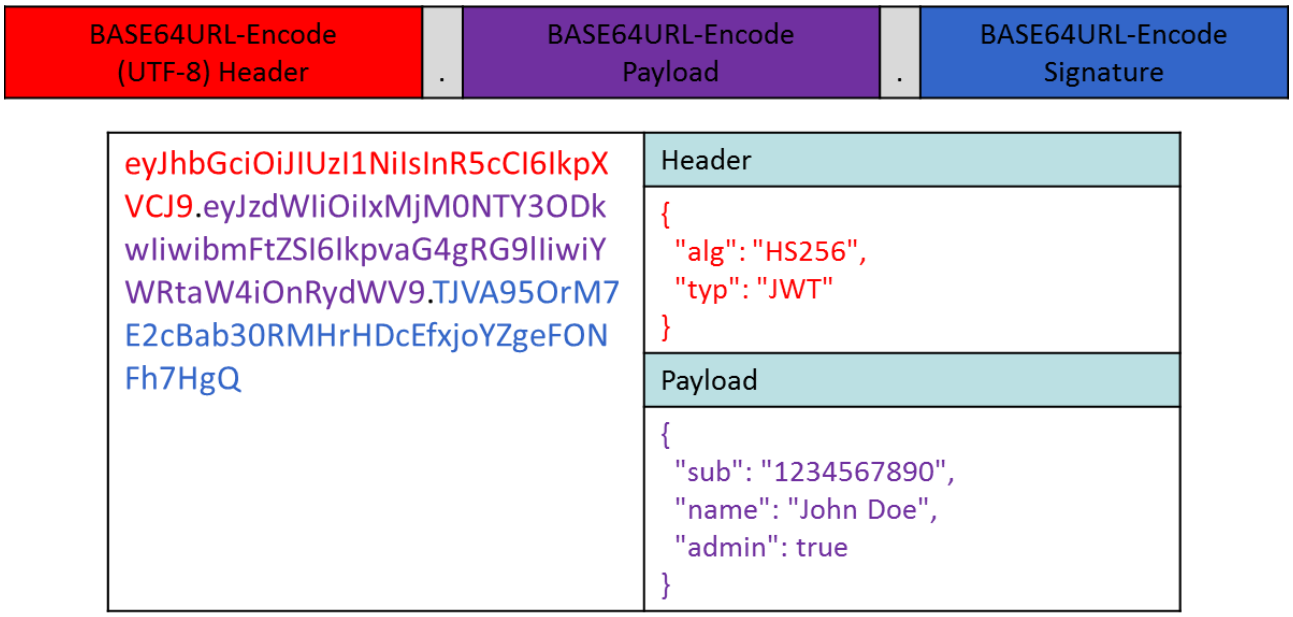

Figure 10: JWS representation

8 Distributed intra-factory 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 this regards, the COMPOSITION architecture already foresee 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 notifications system would not have the necessary flexibility to discriminate properly 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[12] 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:

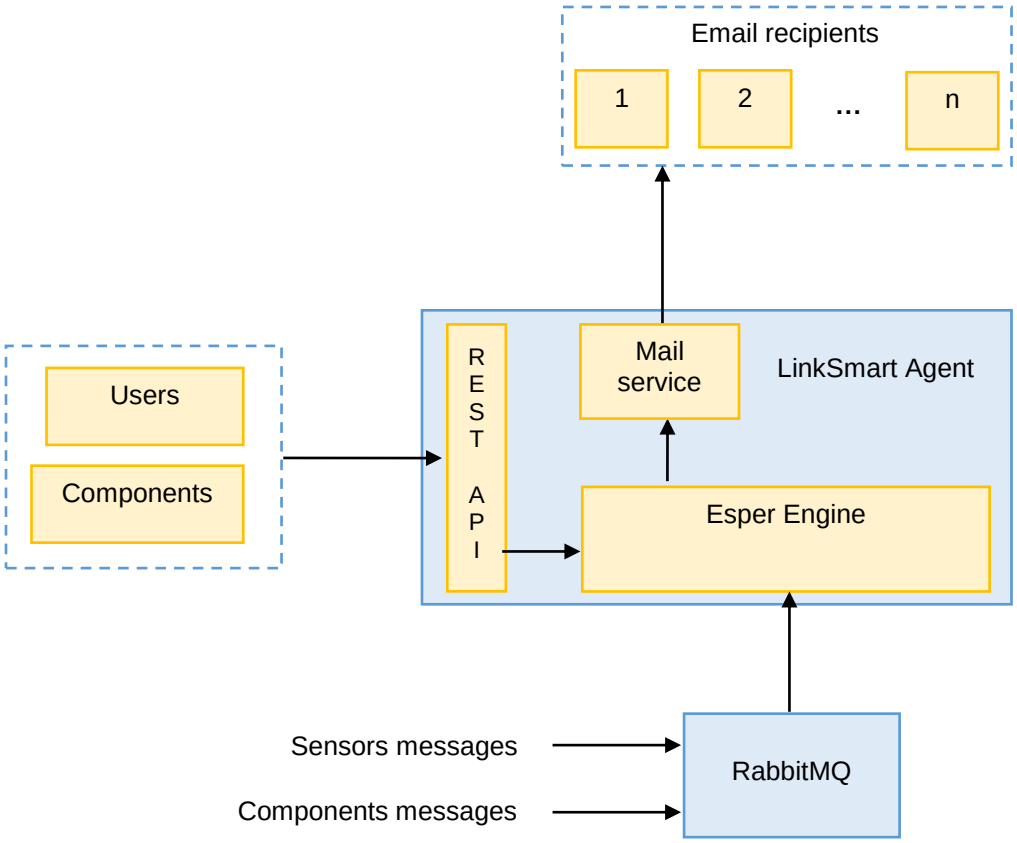

Figure 11: Notification service architecture

This LinkSmart IoT Agent allows email notifications to the specified recipients, based on custom rules defined by users and intra-factory components. These rules, described as continuous query, called statements, are provided to the agent through the API described in section 8.2. The agent behaves by

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

storing provided queries and runs them, automatically and periodically, as live data are 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 in based on the Esper engine[13] 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.

For the first iteration of this component, the query results are reported in plain text in the mail body. Depending on the partners' requirements, a more readable format (e.g. HTML, PDF, etc.) will be evaluated and adopted to facilitate understanding of information in different contexts. Any changes identified by end users during the deployment phase will be reflected in the next iteration of this document.

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[14], 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:

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.

Table 2: Message payload format

The statements are defined through the Event Processing Language[15]. 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.

[13] http://www.espertech.com/esper/
[14] https://docs.linksmart.eu/pages/viewpage.action?pageId=3145798
[15] 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[16]. 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:

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

Table 3: RESTful interface functions

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

9 Conclusions

In this document it has been highlighted how the Intrafactory Interoperability Layer has been developed, deployed and tested in a lab scale environment. 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 intra-factory scenario.

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 ensure 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 intra-factory scenario and form the core of the IIMS.

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.

In the second iteration of this document, named "D5.9 Intrafactory interoperability layer II" and due at M34, it will be reported the final development and adaptation required by mainly two factors. The first one being the obvious changing required by the deployment in the real pilots at end users' premises. The second one, related to the modifications that might be required by the interconnection with not WP5 related components during the integration process with the inter-factory Agent Marketplace.

10 List of Figures and Tables

10.1 Figures

10.2 Tables