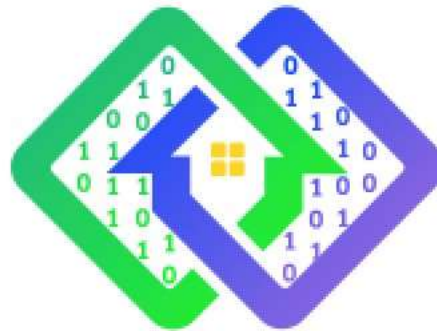


Grant Agreement N° 872592



PLATOON

Digital platform and analytic tools for energy

Deliverable D2.2

Open API Specifications Update

Contractual delivery date:
M27

Actual delivery date:
31 March 2022

Responsible partner:
P17: INDRA-Minsait, Spain

Project Title	PLATOON – Digital platform and analytic tools for energy
Deliverable number	D2.2
Deliverable title	Open API Specifications
Author(s):	INDRA-Minsait, TECN, ENG, CS, UDGA
Responsible Partner:	P17– INDRA-Minsait
Date:	31.03.2023
Nature	R
Distribution level (CO, PU):	PU
Work package number	WP2 – Reference Architecture, Interoperability and Standardization
Work package leader	ENG, Italy

Abstract:	<p>This is the updated version of the original document from M12. No reportable updates or modifications have been notified from participating partners, as the specifications are still being implemented in the pilots and no conclusions have been reached yet.</p> <p>A new section has been annexed in section 8.2, regarding existing open software, such as OGEMA, responsible for communication with proprietary energy generation/storage and monitoring solutions and vendor data formats.</p> <p>This document aims to address integration standards and support integration patterns to enable horizontal interoperability among various heterogeneous systems and business applications.</p> <p>Three sets of existing APIs will be described in this document:</p> <ul style="list-style-type: none"> • NGS-LD API for the internal use between the components within the logical PLATOON architecture and external platforms • Set of APIs for the relation of specific Marketplace components • Specific set of APIs for the Data Analytic Toolbox components
Keyword List:	API, Interoperability, Semantic Networks, NGS-LD, TMForum, Marketplace, OpenAPI

The research leading to these results has received funding from the European Community's Horizon 2020 Work Programme (H2020) under grant agreement no 872592.

This report reflects the views only of the authors and does not represent the opinion of the European Commission, and the European Commission is not responsible or liable for any use that may be made of the information contained therein.

Editor(s):	P17– INDRA-Minsait
Contributor(s):	INDRA-Minsait, TECN, ENG, CS, UDGA
Reviewer(s):	Philippe Calvez (ENGIE) – Platoon Coordinator Erik Maqueda (TECN) – Technical Coordinator Martino Maggio (ENG) – WP leader Vincenzo Savarino (ENG)
Approved by:	Philippe Calvez (ENGIE) – Platoon Coordinator Erik Maqueda (TECN) – Technical Coordinator Martino Maggio (ENG) – WP leader Eduardo Jimenez (IND) – Exploitation Coordinator
Recommended/mandatory readers:	Mandatory: WP2-WP6 leaders and task leaders. Recommended: the rest of the partners

Document Description

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
V1	10/12/2020	First version for revision	INDRA-MINSAIT
V2	14/12/2020	Added section 5 – “Analytics toolbox APIs”	Valentín (TECN)
V3	17/12/2020	Internal review TECN	Erik Maqueda
V4	17/12/2020	Internal review ENG	Martino Maggio
V5	21/12/2020	Internal review ENGIE	Philippe Calvez
V6	11/03/2022	First update version for revision	INDRA-MINSAIT

Table of Contents

Table of Contents

Table of Contents.....	5
List of Figures.....	7
List of Tables.....	8
Terms and abbreviations.....	9
Executive Summary.....	10
1 Introduction.....	12
2 API, interoperability and semantic network general concepts.....	13
2.1 Open API Specifications.....	13
2.2 API design.....	15
2.3 Interoperability.....	16
2.4 Fair Data principles.....	17
2.5 Linked and non-linked data systems.....	18
2.6 RDF.....	19
2.7 JSON-LD.....	20
2.7.1 RDF Serialization/Deserialization.....	22
3 API for interoperability: NGSi-LD API.....	24
3.1 Context information management.....	24
3.2 The NGSi standard.....	25
3.3 The NGSi-LD REST API.....	28
3.3.1 NGSi-LD Information Model.....	29
3.3.1.1 Core Meta Model.....	31
3.3.1.2 Cross-domain ontology.....	32
3.3.1.3 Domain Specific ontology.....	33
3.3.2 A guide to Context.....	34
3.3.2.1 Expansion and Compaction.....	35
3.3.2.2 The Core Context.....	35
3.3.2.3 The default URL.....	36
3.3.2.4 Content-type and Context.....	36
3.3.2.5 Compound context.....	36
3.3.2.6 Value Expansion.....	37
3.3.3 Basic operations.....	38
3.3.3.1 CRUD operations view.....	39
3.3.3.2 Context Information Provision and Consumption.....	40

3.3.3.3	Context Subscriptions	44
3.3.3.4	Context Source Registration	45
3.3.3.5	Context Entity Batch Operations	46
3.3.3.6	Query Pagination	48
3.3.3.7	Temporal evolution	48
4	Marketplace: TMForum APIs	50
4.1	Product Catalog Management API.....	50
4.2	Order Management API	53
4.3	Party Management API	56
4.4	Usage Management API.....	60
4.5	Communication API.....	63
4.6	Customer Management API.....	65
4.7	Customer Bill Management API.....	66
5	Analytics toolbox APIs	69
5.1	Azure Machine Learning example	69
5.2	DEEP Hybrid DataCloud project: DEEPaaS API.....	72
5.3	PLATOON Analytic toolbox OpenAPI definitions	74
6	Reference architecture.....	77
6.1	Context Data Broker.....	78
6.2	IoT Connector	80
6.2.1	IoT Connector APIs.....	81
6.3	Data Connector.....	82
7	Conclusions	83
8	Annex 1	84
8.1	IEC 61850	84
8.1.1	Data models	85
8.1.1.1	Libiec61850 API.....	86
8.1.1.2	Client-server API	87
8.1.1.2.1	Reading and writing data objects.....	87
8.1.1.2.2	Data sets.....	88
8.1.1.2.3	Reports.....	89
8.1.1.2.4	Client authentication.....	90
8.1.1.3	Publisher-subscriber API	90
8.2	OGEMA	92
8.2.1	Framework architecture	92
8.2.2	OGEMA API.....	93
8.2.2.1	Installation and Management of applications	93

8.2.2.2	Resource Management.....	93
8.2.2.3	Resource Listeners	95
8.2.2.4	Logging	96
8.2.2.5	Rest Interface	96
8.2.2.6	Data Models	97
9	References	98

List of Figures

Figure 1: API as a contract.....	13
Figure 2: OpenAPI Specification	14
Figure 3: OpenAPI2.0 vs OpenAPI3.0.....	15
Figure 4: Consumer-First vs API-First approach	15
Figure 5: RDF syntaxes.....	19
Figure 6: Sample JSON document	20
Figure 7: JSON-LD document using full URIS instead of terms	20
Figure 8: Referencing a JSON-LD context	20
Figure 9: External web resources	21
Figure 10: String annotation.....	21
Figure 11: Data types and values	22
Figure 12: Interconnection of Context Information	24
Figure 13: NGSIv2 data model	25
Figure 14: NGSI-LD UML representation.....	26
Figure 15: NGSI-LD Data Model	26
Figure 16: NGSI-LD basic representation	30
Figure 17: Overview of the NGSI-LD Information Model Structure	30
Figure 18: NGSI-LD Core Meta-Model.....	31
Figure 19: NGSI-LD Cross-Domain Information Model	33
Figure 20: Context notion simplified	34
Figure 21: JSON-LD context	37
Figure 22: Example referencing a JSON-LD context	38
Figure 23: Example loading a relative context.....	38
Figure 24: NGSI-LD API operation overview	39
Figure 25: NGSI-LD API Context Information	40
Figure 26: Entity creation example	42
Figure 27: Attribute creation example	42
Figure 28: NGSI-LD Context Subscriptions.....	44
Figure 29: NGSI-LD Context Source Registration	45
Figure 30: NGSI-LD Batch operations	46
Figure 31: Batch create Entity/Attribute example.....	46
Figure 32: Batch Create/Overwrite new entities example	47
Figure 33: NGSI-LD temporal representation operations.....	48

Figure 34: TM Forum API end-to-end	49
Figure 35: TMForum ecosystem APIs	49
Figure 36: Product Catalog Management API Swagger operations I	50
Figure 37: Product Catalog Management API Swagger operations II	51
Figure 38: Product Catalog Management API Swagger operations III	51
Figure 39: Specific catalog creation example	52
Figure 40: Product Order API Swagger ProductOrder operations	53
Figure 41: Product Order API Swagger cancelProductOrder operations	54
Figure 42: Resource order retrieval example	54
Figure 43: Deleting service order example	54
Figure 44: Party Management API Swagger operation	56
Figure 45: Individual resource retrieval example	57
Figure 46: Individual resource creation example	57
Figure 47: Organization resource partial update example	58
Figure 48: Organization entity deletion example	58
Figure 49: Usage Management Api Swagger operations	60
Figure 50: Usage retrieval example	61
Figure 51: Communication API Swagger operations	62
Figure 52: Pre-defined message query example	62
Figure 53: Customer Management API Swagger operations	63
Figure 54: Customer resource creation example	64
Figure 55: Customer Billing Management API Swagger operations	65
Figure 56: Single BillCycle retrieval example	66
Figure 57: DEEP as a Service API endpoint	71
Figure 58: Cross pilot synergies regarding data analytics tools	73
Figure 59: PLATOON reference architecture logical view	75
Figure 60: IoT connector in PLATOON architecture	78
Figure 61: General standard of IEC 61850	82
Figure 62: Protocol stack of of IEC 61850-90-5	83
Figure 63: Data hierarchy	84

List of Tables

Table 1: Context Information Provision Operations	41
Table 2: Context Subscription Operations	44
Table 3: Context Source Registration Operations	45

Terms and abbreviations

API	Application Programming Interface
CRUD	Create, Read, Update and Delete
ETSI	European Telecommunications Standards Institute
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation Linked Data
LD	Logical device
NGSI-LD	Next Generation Service Interfaces Linked Data
OAS	OpenAPI Specification
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	RDF Schema
REST	Representational State Transfer
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
WP	Work Package
XSD	XML Schema Definition
YAML	Yet Another Markup Language

Executive Summary

The present document is the second version of D2.2 that aims to portray any changes or updates that may have taken place since the delivery of the original version (M12). In the 15 months following the first hand-in, partners involved were consulted and there are no updates or modifications to report.

In the first version, there was no mention made regarding the energy domain, OGEMA (Open Gateway Energy Management), so a section has been added on this framework in section 8.2.

This document aims to address integration standards and support integration patterns to enable horizontal interoperability among various heterogeneous systems and business applications.

Three sets of existing APIs will be described in this document:

- NGSi-LD API for the internal use between the components within the logical PLATOON architecture and external platforms
- Set of APIs for the relation of specific Marketplace components
- Specific set of APIs for Data Analytic Toolbox components

Another objective is for readers without previous knowledge of APIs and the NGSi-LD API, to be able to understand and use the NGSi-LD API. In order to meet this objective, this document covers an extensive number of topics and themes that have been structured into four sections as explained below.

The first section takes a look at the general concepts regarding APIs, interoperability and semantic networks. APIs (Application Programming Interface) are described as common, public contract and then the OpenAPI Specification is thoroughly explained as a simple format for describing REST APIs and how this can be achieved with the Swagger tool.

In order to be able to fully understand how APIs work, the two approaches “API First” and “Consumer First” design approaches are explained emphasizing the API development lifecycle.

Then, the concept of interoperability is introduced to highlight its importance in the exchange of information. In this document, only the interoperability through APIs is addressed, diving into the importance of linked and non-linked data systems so that the reader gets the full picture of their differences, followed by an explanation of semantic networks, specifically RDF, and how it is related to JSON-LD, which is explained in detail.

The second section dives into the specification of the NGSi-LD API which is the API designated for interoperability. Context Information Management is first explained, followed by the NGSi (Next Generation Service Interfaces) standard, where the data models are explained.

After the NGSi standard is described in detail, the NGSi-LD REST API is defined. It is a new data exchange protocol that allows the discovery and exchange of information across databases, mobile Apps and IoT platforms. In following sub-sections, the Information Model components are looked at, going into detail on the context and the basic operations supported by the NGSi-LD API, with the Swagger specification and different examples.

In the next section, different APIs will be specified for the PLATOON marketplace to allow the monetization of different kinds of assets. This ecosystem will expose its complete functionality through the implementation of TMForum standard APIs such as the Usage Management API, Customer Billing API, Party Management API, etc.

The following section will define the APIs for the Analytic toolbox services. This will be formed by all the different data analytic tools developed and used in the project by the different partners for the different use cases, divided into two main groups: Energy specific tools and Generic tools. Two different machine learning and data analytics APIs (Azure predictive maintenance OpenAPI and DEEPaaS API) will be explained with their respective examples of use in PLATOON. Lastly, The PLATOON approach for the OpenAPI APIs for the Analytic toolbox services will be defined.

Then, a look at PLATOON'S logical architecture will be taken a look at in order to see the information flow with an emphasis on the components which will make use of the APIs specified in this document.

1 Introduction

In this task, PLATOON will exploit the potential of service-oriented deployment principles to mediate and transform data across a variety of systems, services and APIs. In this way, it will address integration standards and support integration patterns, thus enabling horizontal interoperability among various heterogeneous systems and business applications.

These will be taken into account with existing standardisation activities related to Open API, and data interoperability, for instance NGSI-LD, Context Information Management API, defined by ETSI ISG CIM. To support vertical interoperability with underlying technical and energy assets, PLATOON will be leveraged on existing open software, such as Energy Gateway responsible for communication with proprietary energy generation/storage and monitoring solutions and vendor data formats (e.g. IEC 61850, IEC 61968 etc.).

2 API, interoperability and semantic network general concepts

In this section, the general concepts regarding APIs, interoperability and semantic networks will be briefly explained.

2.1 Open API Specifications ¹

An API (Application Programming Interface) is a common, public contract (figure 1) between services and clients. APIs define a group of rules, specifications, protocols and functions that applications can use to exchange information. It gets services ready for third parties to consume, including a technical description and promoting system integration by clear contracts durable in time. ²

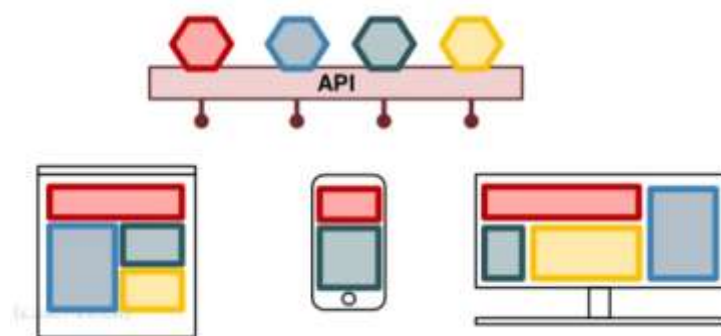


Figure 1: API as a contract

The OpenAPI Specification (OAS) defines a standard, language-agnostic interface to RESTful (Representational state transfer) APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, additional documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. The OpenAPI Specification removes guesswork in calling a service in a similar way as interfaces descriptions have for lower-level programming.

REST is a software architectural style that defines a set of constraints to be used for creating Web services and those that conform to the REST architectural style (RESTful Web Services) providing interoperability between computer systems on the internet. This allows requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations.

Use cases for machine-readable API definition documents include, but are not limited to:

- interactive documentation
- clients and servers
- automation of test cases

OpenAPI documents describe API services and being independent of language, framework and deployment technology, can be represented in either YAML³ or JSON⁴ formats, which may either be produced and served statically or be generated dynamically from an application. The

files describing the RESTful API in accordance with the Swagger specification are represented as JSON objects and conform to the JSON standards.

The Open API specification defines the world standard for RESTful APIs, and it was donated to the Linux foundation under the OpenAPI initiative in 2015, creating a RESTful interface for easily developing and consuming an API by effectively mapping all the resources and operations associated with it.

The Swagger tools were developed by the team behind the original Swagger Specification. Swagger (a project used to describe and document RESTful APIs) offers the most powerful and easiest to use tools to take full advantage of the OpenAPI Specification.

An OpenAPI file allows the description of the whole API including:

- API endpoints (e.g. /users)
- Operations for each endpoint (e.g. GET, POST, DELETE, etc)
- Input Parameters (e.g. /users?active=true)
- Responses and format (200: {"name": "adam"})
- Authentication (Basic Auth, OAuth2, etc)
- API meta info (contact, license, usage conditions, etc)

In summary, OpenAPI Specification offers a simple format for writing REST service contracts (figure 2).



Figure 2: OpenAPI Specification

An OpenAPI definition can then be used by documentation generation tools (such as Swagger as mentioned before) to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, and many other use cases.

The Open API Specification does not require rewriting existing APIs, or binding any software to a service (which may not even be owned by the description creator). However, the service capabilities have to be described within the OpenAPI Specification. Not all services can be described as it is not intended to cover every style of HTTP APIs, but includes support for REST APIs. The Open API Specification does not mandate a specific development process, but facilitates either technique (design-first or code-first, explained in section **Fehler! Verweisquelle konnte nicht gefunden werden.**) by establishing clear interactions with a HTTP API.

Swagger allows the API structure description so that machines can read them, and by reading the API's structure, it can automatically build interactive API documentation. Swagger offers:

- Tags that allow operation definition and control

- Configuration

Figure 3 shows the difference between OpenAPI 2.0 and the changes included in the upgraded version OpenAPI 3.0.

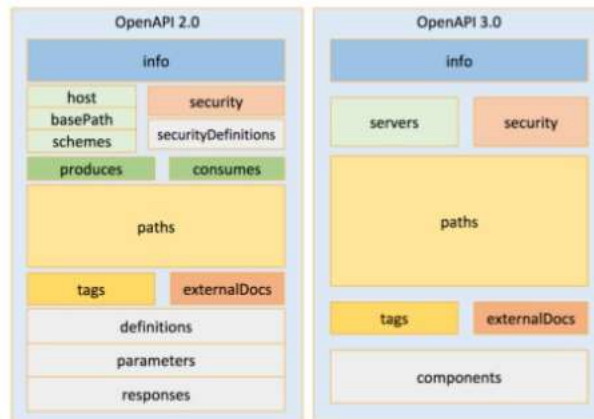


Figure 3: OpenAPI2.0 vs OpenAPI3.0

2.2 API design

There are two approaches that can be taken when designing an API. The more traditional approach consists on creating an application with all the functionalities required for the business logic, creating UI, back-end, etc. and then creating an API able to interact with the applications. This approach is called Consumer-First development.

On the other hand, the API-First development begins with the API design and development. It is an architecture that treats the user as the primary user of the application. Therefore, it requires that the API is complete, responsive and well-documented, requiring a strong collaboration between developers and consumers.^{5 6}



Figure 4: Consumer-First vs API-First approach

Figure 4 shows the differences between the two different API development approaches. The scheme on the left represents the Consumer-First approach, while the one on the right represents the API-First approach. In the latter one, the API interfaces are first defined with mocked data,

so that participants can consume and develop in parallel. This approach has several advantages, such as:

- Development agility
- Time-to-market
- Better client experience.

The API development lifecycle can be categorized in the following three steps (marked on figure 4):

1. Contract

The first step entails API design, where all the stakeholders such as POs, back-end and developers, clients, users, etc. are involved. It is at this point that OpenAPI (OAS3) will be leveraged for the API definition and design.

OpenAPI3 allows definition version control, allowing to track all changes, and finally stakeholders can add to the iterations, so that the API can be accepted with the least possible changes so that development time is minimised.

Contract can be summarized in the following:

- OpenAPI Specification definition
- Version control
- Acceptance

2. Code

- Code generation from the generated YAML/JSON

3. Contract test

These allow the verification of API request and answer schemes and its behaviour.

2.3 Interoperability

Interoperability is associated with the ability of two or more systems or components to exchange data and use information. In this document only the interoperability through APIs will be addressed.

An API can be thought of as the middle man that works between two applications (as explained in section **Fehler! Verweisquelle konnte nicht gefunden werden.**), that accepts requests (if allowed) and returns the data back to the requestor. The API also lets the requestor know about the data that can be requested, exactly how to ask for the data and how it can be received.

There are several levels of interoperability identified by the existing literature: ⁷

- Technical Interoperability (connectivity, network) is usually associated with hardware/software components that enable communication. It presupposes an agreement on how the information is transported across multiple communication networks and the protocols needed.
- Syntactic Interoperability is usually associated with data formats. Messages transferred by communication protocols and their payload need to have a well-defined, agreed syntax and encoding.

- Semantic Interoperability is associated with the meaning of the content that is exchanged. This requires agreement on common concepts and their relationships. It refers to information, is portable and well understood by any subsequent system requesting and reviewing it.
- Organizational Interoperability is the ability of organizations to effectively communicate and transfer meaningful information among a variety of different information systems and infrastructures. Organizational interoperability depends on successful technical, syntactic and semantic interoperability.

For communication across different systems, the semantic level is essential in order to achieve interoperability, and the information exchange must refer to a commonly agreed reference model.

Semantic interoperability is the designed property where various systems can interact with each other and exchange data with unambiguous, shared meaning. It is achieved when there is a generally accepted (consistent) information model/data model (defined in T2.3 “Data models”, the current task focuses solely on API interoperability), by including information regarding the data (metadata) and linking that element to a commonly shared vocabulary. This shared vocabulary and the association to an ontology enable a machine-accessible representation, making the whole input space accessible to intelligent queries and machine reasoning/inferencing that would facilitate analysis. ⁸

Currently, data is often made available through RESTful APIs that form the connecting glue between modern applications. Nearly every application use APIs to connect with corporate data sources, third party data services or other applications. The main purpose is to deliver end users an endpoint to which a generic input about a particular domain will be provided.

PLATOON’s role as an integrator of existing platforms, provides a meeting point for providers and consumers where through a marketplace it will enable advertising and discovery of offerings, give uniform access to distributed repositories and services, and billing and charging functionalities. For this reason, an API will be specified to provide access of semantically described data with descriptions of capabilities. Each component is able to interact with any other component through the provided API, but also with outside components through NGSILD (section **Fehler! Verweisquelle konnte nicht gefunden werden.**).

2.4 Fair Data principles ⁹

Before getting into the different interoperability components, this section will introduce the FAIR data principles, whose aim is to meet standards of findability, accessibility, interoperability and reusability.

There is a need from the industry’s side to make data available and usable in order to be able to generate value, which can be directly mapped to the FAIR data principles. In a business ecosystem that is driven by data, the data must be findable, accessible, interoperable and supported by available legal contracts such as usage policies. These data principles are achieved by different components/specifications that will be used/developed in PLATOON. The following mapping can be established:

- Findable: the IDS broker enables the search of data sources and supports dataset metadata searching.

- Accessible: NGS-LD API and other APIs defined in this task (Marketplace and Data Analytics Toolbox APIs) along with the PLATOON Data models defined in task T2.3 “Data models” support the access to data in a standardized way.
- Interoperable: the NGS-LD API information layer and the information layer from other APIs defined in this task (Marketplace and Data Analytics Toolbox APIs) along with the PLATOON Data models, defined in task T2.3 “Data models”, provide an information model that acts as a foundation for semantic interoperability. By agreeing to this standard, any component within the logical architecture is able to access the context data.
- Reusable: the NGS-LD API and other APIs defined in this task (Marketplace and Data Analytics Toolbox APIs) along with the PLATOON Data models, defined in task T2.3 “Data models”, allow data reusability through the information model used, that identifies data with an exact description. This lets all the interested platforms and apps to use the aforementioned set of data.

PLATOON goes a step beyond and applies the same FAIR principles to the Data Analytics Tools as defined in task T4.1 “PLATOON analytical toolbox design”.

2.5 Linked and non-linked data systems

In single isolated systems, it makes no difference whether a rich, complex linked-data architecture is used or a simpler non-linked data system is created. However, if the data is designed to be shared, then linked data is a requirement in order to avoid silos. External systems cannot know about the relationships unless they are provided with a machine-readable format.
10 11

Without linked data, there is no machine-readable way to connect entities together and every data relationship must be known in advance. In isolated systems, this is not an issue since the system architect will know in advance what-connects-to-what.

On the other hand, with a well-defined data model using linked data, every relationship can be predefined in advance and is discoverable. Using JSON-LD concepts (section **Fehler! Verweisquelle konnte nicht gefunden werden.**), computers can easily understand indirect relationships and can navigate between linked entities. This is why it is possible to create usable models which are ontologically correct and attributes can be assigned directly to entities.

When creating linked data entities, it is important to use common data models, in order to be able to easily combine data from multiple sources and remove ambiguity when comparing data coming from different sources.

Creating linked data using fully qualified names throughout is a hassle, as each attribute would need a URI (explained below). JSON-LD introduces the idea of the @context attribute which can hold pointers to context definitions (explained in section **Fehler! Verweisquelle konnte nicht gefunden werden.**).

URI stands for Uniform Resource Identifier and it provides a standard way for resources to be accessed by other computers over a network or over the World Wide Web. A URI identifies a resource either by location, by name or both. URL (Uniform Resource Locator) is a subset of URI that specifies where an identified resource is available and the mechanism for retrieval, defining how it can be obtained (<http://>, <ftp://>, <smb://>). On the other hand, a URN (Uniform Resource Name) is location independent and allocates a resource permanently but does not

indicate availability.¹² These concepts are explained in more detail in deliverables D2.3 “Data models” and D2.4 “Data integration”.

2.6 RDF

A semantic network or frame network is a knowledge base that represents semantic relations between concepts in a network, used as a form of knowledge representation. Semantic networks are used when one has knowledge that is best understood as a set of concepts that are related to one another. RDF (Resource Description Framework) has been declared as the basic model to capture knowledge of the semantic web.¹³

RDF is a framework for expressing information about resources and is intended for situations in which information on the Web needs to be processed by applications. It provides a common framework for expressing this information so it can be exchanged between applications without loss of meaning, by publishing and interlinking data in the web.

RDF is a standard model for data interchange on the Web, that allows the expression of simple facts in the form of triplets (subject, predicate and object).¹⁴ The subject and the object represent the two resources being related, the predicate represents the nature of their relationship and this relationship is phrased in a directional way (from subject to object) representing the property.¹⁵

It introduces axioms for semantic graphs, making it possible to define hierarchies of classes and properties adding the notion of domain and range for properties. Thus, RDF has features that facilitate data merging even if the underlying schemas differ, and it specifically supports the evolution of schemas over time without requiring all the data consumers to be changed.

RDF extends the linking structure of the Web to use URIs to name the relationship between things as well as the two ends of the link (this is usually referred to as a “triple”). Using this simple model, it allows structured and semi-structured data to be mixed, exposed, and shared across different applications.

There are various concrete syntaxes for RDF, such as Turtle [TURTLE], TriG, [TRIG], and JSON-LD [JSON-LD] as shown in figure 5.^{16 17}

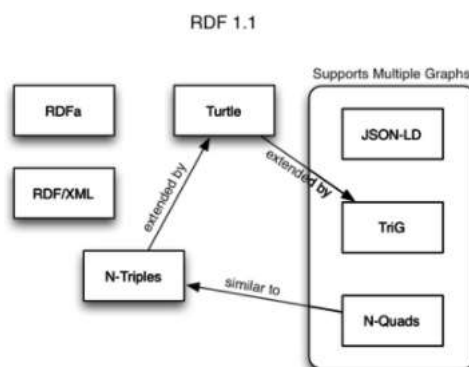


Figure 5: RDF syntaxes

JSON-LD provides a JSON syntax for RDF graphs and datasets and it can be used to transform JSON documents to RDF with minimal changes by offering universal identifiers for JSON objects. This is the mechanism in which a JSON document can refer to an object described in another JSON document elsewhere on the Web.¹⁸

JSON-LD can also be used as RDF in conjunction with other Linked Data technologies like SPARQL, which will be used in other PLATOON components, e.g. in order to realize queries to the Unified Knowledge base. These concepts are explained in more detail in deliverables D2.3 “Data models” and D2.4 “Data integration”.

2.7 JSON-LD

JSON is a lightweight, language-independent data interchange format, easy to parse and generate (figure 6 provides an example of a JSON document). However, it is difficult to integrate JSON from different sources and it has no built-in support for hyperlinks.^{19 20}

```
{
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

Figure 6: Sample JSON document

As mentioned before, Linked Data uses URIs for unambiguous identification and they provide a way to create a network of standards-based machine interpretable data across different documents, allowing an application to start at one piece of Linked Data and follow embedded links to other pieces which may be hosted on different sites.

JSON-LD serializes Linked Data in JSON, primarily intended to be a way to use Linked Data in Web-based programming environments, to build interoperable web services and to store Linked Data in JSON-based storage engines. It is 100% compatible with JSON and on top of all the JSON features, JSON-LD is introduced in this section and further examples will be shown in the next sections. JSON-LD provides:²¹

- a universal identifier mechanism for JSON objects via the use of URIS

```
{
  "http://schema.org/name": "Manu Sporny",
  "http://schema.org/url": { "@id": "http://manu.sporny.org/" }, - The '@id' keyword means 'this value is an identifier that is an IRI'
  "http://schema.org/image": { "@id": "http://manu.sporny.org/images/manu.png" }
}
```

Figure 7: JSON-LD document using full URIS instead of terms

- a way to disambiguate keys shared among different JSON documents and mapping them to URIs via a context²²

```
{
  "@context": "https://json-ld.org/contexts/person.jsonld",
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

Figure 8: Referencing a JSON-LD context

- a mechanism in which a value in a JSON object may refer to a resource on different platforms/webs

```
{
  "@context": "http://www.w3.org/ns/anno.jsonld",
  "id": "http://example.org/anno2",
  "type": "Annotation",
  "body": {
    "id": "http://example.org/analysis1.mp3",
    "format": "audio/mpeg",
    "language": "fr"
  },
  "target": {
    "id": "http://example.gov/patent1.pdf",
    "format": "application/pdf",
    "language": ["en", "ar"],
    "textDirection": "ltr",
    "processingLanguage": "en"
  }
}
```

Figure 9: External web resources

- the ability to annotate strings as shown in figure 10, where the value `http://manu.sporny.org/` is expressed as a JSON string.

```
{
  "@context": {
    ...
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
    ...
  },
  ...
  "homepage": "http://manu.sporny.org/"
  ...
}
```

Figure 10: String annotation

- a way to associate datatypes with values (e.g. dates and times)²³

	Datatype	Value space (informative)
Core types	xsd:string	Character strings (but not all Unicode character strings)
	xsd:boolean	true, false
	xsd:decimal	Arbitrary-precision decimal numbers
	xsd:integer	Arbitrary-size integer numbers
IEEE floating-point numbers	xsd:double	64-bit floating point numbers incl. $\pm\text{Inf}$, ± 0 , NaN
	xsd:float	32-bit floating point numbers incl. $\pm\text{Inf}$, ± 0 , NaN
Time and date	xsd:date	Dates (yyyy-mm-dd) with or without timezone
	xsd:time	Times (hh:mm:ss.sss...) with or without timezone
	xsd:dateTime	Date and time with or without timezone
	xsd:dateTimeStamp	Date and time with required timezone
Recurring and partial dates	xsd:gYear	Gregorian calendar year
	xsd:gMonth	Gregorian calendar month
	xsd:gDay	Gregorian calendar day of the month
	xsd:gYearMonth	Gregorian calendar year and month
	xsd:gMonthDay	Gregorian calendar month and day
	xsd:duration	Duration of time
	xsd:yearMonthDuration	Duration of time (months and years only)
	xsd:dayTimeDuration	Duration of time (days, hours, minutes, seconds only)
Limited-range integer numbers	xsd:byte	-128...+127 (8 bit)
	xsd:short	-32768...+32767 (16 bit)
	xsd:int	-2147483648...+2147483647 (32 bit)
	xsd:long	-9223372036854775808...+9223372036854775807 (64 bit)
	xsd:unsignedByte	0...255 (8 bit)
	xsd:unsignedShort	0...65535 (16 bit)
	xsd:unsignedInt	0...4294967295 (32 bit)
	xsd:unsignedLong	0...18446744073709551615 (64 bit)
	xsd:positiveInteger	Integer numbers >0
	xsd:nonNegativeInteger	Integer numbers ≥ 0
	xsd:negativeInteger	Integer numbers <0
	xsd:nonPositiveInteger	Integer numbers ≤ 0
	Encoded binary data	xsd:hexBinary
xsd:base64Binary		Base64-encoded binary data
Miscellaneous XSD types	xsd:anyURI	Absolute or relative URIs and IRIs
	xsd:language	Language tags per [BCP47]
	xsd:normalizedString	Whitespace-normalized strings
	xsd:token	Tokenized strings
	xsd:NMTOKEN	XML NMTOKENS
	xsd:NCName	XML NCNames

Figure 11: Data types and values

- a facility to express one or more directed graphs, such as a social network in a single document. Directed graphs mean that every property points from a node to another node or value.

Some specific examples for PLATOON can be found in deliverables D2.3 “Data models” and D2.4 “Data integration”.

JSON-LD is designed to be usable directly as JSON, with no RDF knowledge. JSON-LD as a serialization format of RDF, is used by the NGSI-LD API (section **Fehler! Verweisquelle konnte nicht gefunden werden.**), and is the method through which the NGSI-LD graphs can be converted to RDF.

2.7.1 RDF Serialization/Deserialization

The process of serializing RDF as JSON-LD and deserializing JSON-LD to RDF depends on the RDF Serialization-Deserialization Algorithms.^{24 25}

The procedure to deserialize a JSON-LD document to RDF involves the following steps:

1. Expand the JSON-LD documents, removing any context, thus ensuring properties, types and values are given their full representation as URIs and expanded values.
2. Flatten the document, turning it into an array of node objects (represents zero or more properties of a node in the graph serialized by the JSON-LD document).
3. Turn each node object into a series of triples.

Some specific examples for PLATOON can be found in deliverables D2.3 “Data models” and D2.4 “Data integration”.

3 API for interoperability: NGSI-LD API

Several APIs will be specified in this document, the first one being NGSI-LD.

First, the Context information management will be introduced, followed by the definition of the NGSI standard. Here, the difference between NGSIv2 and NGSI-LD will be explained in order to be able to delve into the NGSI-LD data model.

Then, the NGSI-LD REST API, which allows the discovery and exchange of information across databases, platforms, etc. will be explained in depth, going through the Information Model and the different layers within the model structure.

Finally the basic operations supported by the API will be shown with CRUD operations, Swagger specifications and different examples.

3.1 Context information management

PLATOON federated platform can be considered a set of “smart” services and these can be depicted as interconnections of context providing services and context consuming applications that work together to ensure that each application has the information it requires to deliver knowledge and insight, and to exercise control. The context of an application can be regarded as being all the relevant aspects of its operating environment that are required for it to work as intended, sometimes needing a different mix of data (content) from one or more sources.²⁶

A context producer may be a sensor, a database, an open repository, etc. Figure 12 shows how context producers and consumers are connected by a cross-connecting Context Information Management System, where the NGSI-LD (Next Generation Service Interfaces) API is used by data consumers to query for and receive updates on context information.

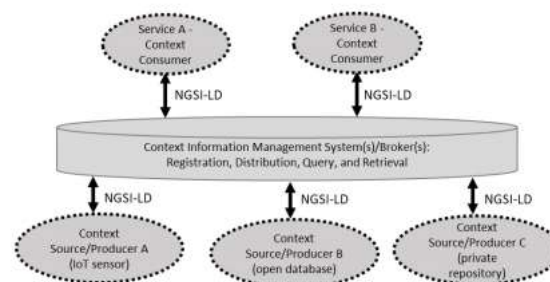


Figure 12: Interconnection of Context Information

Context information is considered to be any relevant information about entities, their properties such as temperature, location or any other such parameter, and their relationships with other entities.

Context information is exchanged among applications, context producers and context brokers.

3.2 The NGSI standard

Next Generation Service Interfaces (NGSI) is a specification, originally defined by Open Mobile Alliance²⁷ (OMA), that stipulates a model enabling the data interchange in systems where a wide range of information, arising from different sources, must be collected and managed. This specification provides the interfaces defining the behavior and functionality of the components to be implemented. NGSI is the information model supporting the working fundamentals for the Context Broker (see section **Fehler! Verweisquelle konnte nicht gefunden werden.**), being the base of the architecture for the communication among components that manage data.²⁸

The overall schema involves the existence of multiple entities, called context elements, which collect information extracted from a given environment that is useful to define the status or features of said environment. For that, two variants of the NGSI data model are currently being used, NGSIv2 (evolution by FIWARE²⁹ of the NGSI specification), implemented by the RESTful NGSIv2 API, and NGSI-LD (ETSI³⁰ standard), that extends the capabilities of the first by including relationships that link data from different entities and can be implemented by the NGSI-LD REST API.³¹

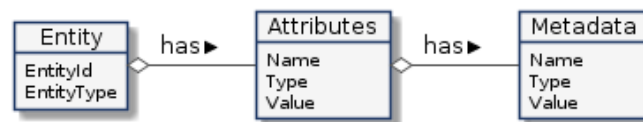


Figure 13: NGSIv2 data model

Figure 13 summarizes the NGSIv2 data model as follows:

1. The core element is the data entity (real object with a changing state). Entities have attributes (such as name and location) and these in turn hold metadata such as accuracy (the accuracy of a reading).
2. Every Entity must have a type (definition of the sort of thing the entity describes).
3. Relationships can be defined in NGSIv2, but only so far as giving the attribute an appropriate attribute name defined by convention.

NGSI-LD is an evolution of the NGSIv2 information model, which has been modified to improve support for linked data (entity relationships), property graphs and semantics (exploiting the capabilities offered by JSON-LD as shown in figure 14).³²

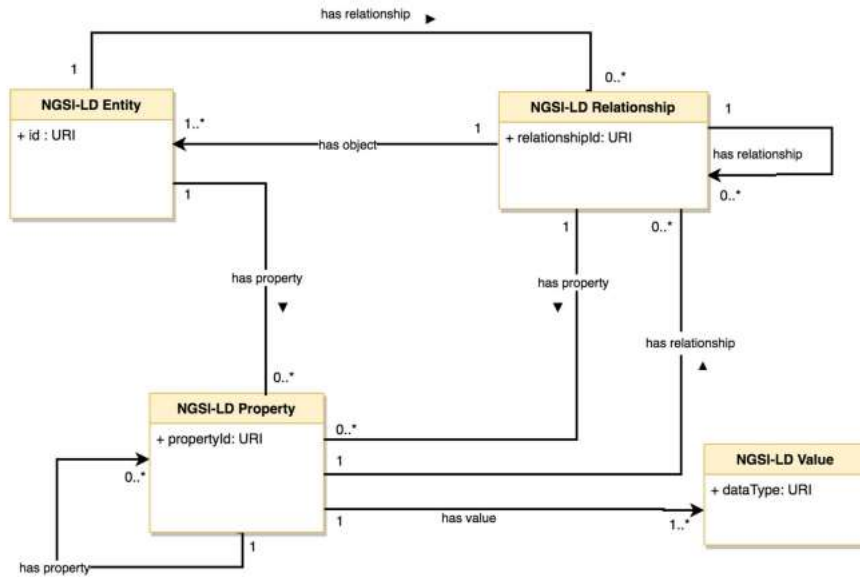


Figure 14: NGSI-LD UML representation

In the NGSI-LD information model, there are Entities, Properties and Relationships. Entities (instances) can be the subject of other Properties or Relationships. In terms of the traditional NGSI data model, Properties can be seen as the combination of an attribute (property) and its value. Relationships allow the establishment of "links" between instances using JSON-LD conventions. In practice, they are similar to NGSI attributes, but with a special value (named object) which happens to be a URI which points to another entity residing in the same system or externally.

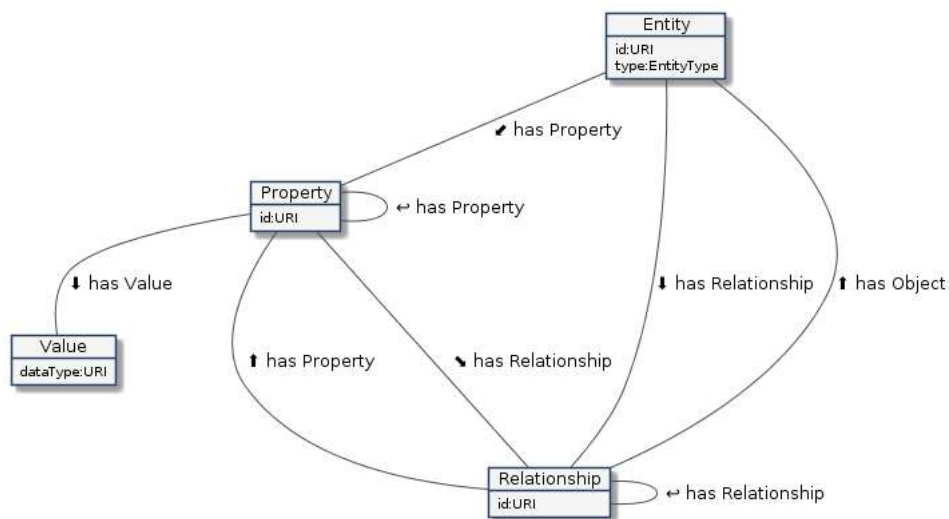


Figure 15: NGSI-LD Data Model

The NGSI-LD data model is more complex than NGSIv2, with more rigid definitions of use which leads to a navigable knowledge graph (figure 15).

1. The Entity (core element) is the informational representation of something that is supposed to exist in the real world, physically or conceptually. For example:

```
"id": "urn:ngsi-ld:TemperatureSensor:001 "
```

It is uniquely identified by a URI and characterized by reference to one or more Entity Types. The Entity Type categorizes the Entity into a class of similar entities or sharing a set of characteristic properties and is also uniquely identified by a URI. It defines the structure of the data held. For example:

```
type": "TemperatureSensor"
```

This URI should correspond to a well-defined PLATOON data model.

If those URIs are expected to participate in external linked data relationships they should be allowed to be referenced.

2. Entities can have properties and relationships (attributes).
3. Property (static or dynamic characteristic of an entity) is a description instance which associates a main characteristic to either an Entity, Relationship or another property, where the name of each property should be a well defined URI corresponding to a common concept found across the web or PLATOON data models. For example `http://schema.org/address` is a common URI for the physical address of an item.

The property will also have a value which will reflect the state of that property (e.g. `batteryLevel`). Value is a JSON value (i.e. a string, a number, true or false, an object, an array) or a JSON-LD typed value (i.e. a string as the lexical form of the value together with a type, defined by a XSD base type) or a JSON-LD structured value (i.e. a set, a list, etc.)

Property may itself have further properties (properties-of-properties) which reflect further information about the property itself.

An example of a property could be as follows:

```
dateObserved": {
  "type": "Property",
  "value": { "@type": "DateTime",
    "@value": "2018-08-07T12:00:00Z"
  }
}
```

4. Relationship (e.g. `controlledAsset`) is the description of a directed link between a subject (NGSI-LD Entity, NGSI-LD Property or another NGSI-LD Relationship) and an object (NGSI-LD Entity) using *hasObject* to define its target object.

An example of a relationship could be:

```
refPointOfInterest": {
  "type": "Relationship",
  "object": "urn:ngsi-ld:PointOfInterest:RZ:MainSquare"
}
```

5. Metadata is replaced by a nested schema of properties for each attribute, such as measurement units, location of the sensor or acquisition date, known as properties of properties or properties of relationships. Said nested properties include:
 - GeoProperties, which involve geospatial data
 - Time Related Properties, which provide acquisition date or similar information
 - “unitCode” refer to Properties for measurements
 - Geometry values for geospatial properties
 - Time Values for time instants or intervals

The shaping of the entities will be carried out according to the NGSI-LD standard. Briefly, entities have to define the data source, the type of data or magnitude provided, its observed value and its measurement units, besides the location of the other related components. All these definitions are encoded in the attributes of the entity. Additional information, such as observation date, geospatial data or other meta-data, are included in nested attributes with the same structure as the primary ones.

Since the context elements are structured in a nested way, they are good targets to be encoded at ontologies with the JSON notation, which is capable of properly distributing the data contained. Furthermore, entities represented as JSON documents can be directly persisted as records in databases like MongoDB.

Furthermore, attributes are also used to represent relationships with other entities within the Context Broker and other components within the PLATOON architecture, by pointing to the URI of the corresponding mates. Thus, relationships are shaped as a list of attributes whose values are the digital locations of the related entities.

3.3 The NGSI-LD REST API

NGSI-LD is a new data exchange protocol that allows the discovery and exchange of information easily with open databases, mobile Apps and IoT platforms.^{33 34}

The NGSI-LD API (developed by ETSI ISG CIM) is a standard API for the management of context information. It is a RESTful API leading to a simplified and accelerated development, while at the same time supporting geo-queries, notification/subscription, federation, Linked Data, etc.³⁵

The API (named NGSI-LD) aims to enable applications to discover, access, update and manage data and context information from many different sources as well as to publish it through interoperable data publication platforms such as Open Data platforms.

The NGSI-LD API provides access to the data through the context broker and the rest of the PLATOON components and allows users to provide, consume and subscribe to context information in multiple scenarios, involving multiple stakeholders. It enables close to real-time access to information coming from different sources, as well as being able to perform updates on context, register context providers, query information on current and historic context information and subscribing to receive notifications of context changes.

3.3.1 NGS-LD Information Model

The NGS-LD Information Model prescribes the structure of context information that shall be supported by an NGS-LD system. It specifies the data representation mechanisms that shall be used by the NGS-LD API itself. In addition, it specifies the structure of the Context Information Management vocabularies to be used in conjunction with the API.

The NGS-LD information model is framed within an ontology and adopts JSON-LD for context information. JSON-LD is a Linked Open Data serialization using the popular JSON (Javascript Object Notation) format that is convenient for back-end and browser-based development, defined by W3C as an official serialization.

For NGS-LD, JSON-LD, which is an extension of JSON devised to support linked data, fits better as the encoding format to collect the relationships and the context describing the entities graph.

As already mentioned before, JSON-LD is a lightweight linked data format, easy for humans to read and write, providing a way for JSON data to interoperate. Linked Data empowers people that publish and use information in the web, creating a network of standards-based, machine readable data across websites. It allows an application to start at one piece of Linked Data and follow embedded links to other pieces of Linked Data that are hosted on different sites across the web.

Although NGS-LD uses ontologies similar to NGS-v2, this format also includes annotations that improve the semantic capabilities of the information encoded at the entities. The most useful annotation is @context (which will be thoroughly explained in section **Fehler! Verweisquelle konnte nicht gefunden werden.**), which stores the context namespace (set of signs/names that are used to identify and refer to objects of various kinds, ensuring that all of a given set of objects have unique names so they can be easily identified). Moreover, annotations have to be used also to stipulate the role of the nested attributes supporting essential data descriptions, @type and @value, when their depth overcomes the first level.

The NGS-LD specification defines an abbreviated representation of Entities, which allows consuming only entity data (the target object of each Relationship or the value of each Property) corresponding to the Properties or Relationships whose subject is the Entity itself. The simplified representation of Entities shall be supported by implementations and can be selected by Context Consumers through specific request parameters.

The entity representation will be described in detail in the next sections, but first a simplified entity representation will shown. It will include:

- A JSON-LD @context
- A JSON-LD object containing the following members:
 - Id, type and @context
 - For each Property a member whose key is the Property Name (a term) and whose value is the Property Value.
 - For each Relationship a term whose key is the Relationship Name (a term) and whose value is the Relationship's Object (represented as a URI).

```

{
  "id": "urn:ngsi-ld:example",
  "type": "observedmagnitude",
  "observationDate": {
    "type": "Property",
    "value": {
      "@type": "DateTime",
      "@value": "2020-08-01T12:00:00Z"
    }
  },
  "magnitude1": {
    "type": "Property",
    "value": value1,
    "unitCode": "unit1",
  },
  "refMeteringPoint": {
    "type": "Relationship",
    "object": "urn:ngsi-ld:meter1"
  },
  "@context": [
    "https://schema.lab.fiware.org/ld/context",
    "https://uri.etsi.org/ngsi-ld/v1/ngsi-ld/core-context.jsonld"
  ]
}

```

Figure 16: NGSI-LD basic representation

Figure 16 shows the basic structure for the representation of an NGSI-LD-formatted entity by JSON-LD notation. This model also uses the entity type and a list of attributes storing the properties, but the id is now the URI locating the element ("urn:ngsi-ld:example"), which can be shortened by using the namespace tagged with the context annotation (@context). Furthermore, other attributes define relationships with other context elements, by pointing to their URIs, being tagged as Relationship type instead of Property.

To face the data requirements of the current project, the NGSI-LD API has been chosen because of its capability to work with linked data, a critical aspect to achieve the goals of interoperability that have been proposed.

This linking structure forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph nodes. This graph view is the easiest possible meta model for RDF and is often used in easy-to-understand visual explanations.

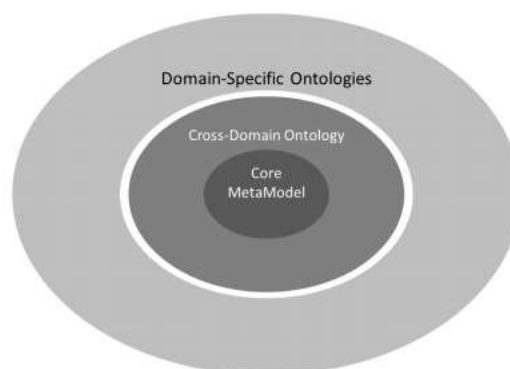


Figure 17: Overview of the NGSI-LD Information Model Structure

Figure 17 shows an overview of the NGSI-LD information model structure. According to the NGSI-LD schema, context information is structured at three levels, each one wrapping the level placed below to increase its degree of specialization:

1. **Core meta-model** is the lower level and describes the common structural features to be fit by the entities created according to this standard. Examples include:
 - a. Entity URI
 - b. Entity Type
 - c. Generic properties
 - d. Relationships
 - e. Nested attributes
 - f. Context namespace
2. **Cross domain model** determines the specific schema for the time (Time Property: observation date) and geographical location (GeoProperty: location).
3. **Domain specific model** implements the selected ontologies, or combinations of them (specific or hybrid ontologies).

3.3.1.1 Core Meta Model

The core meta-model of NGSI-LD (figure 18) amounts to a formal specification of the “property graph” model. It is the lower level and stipulates the common structure for the data entities, defined by their properties and relationships with other entities. It defines the atomic minimal information that can be published (entities, properties and relationships). The possibility to nest properties is also defined at this level.³⁶

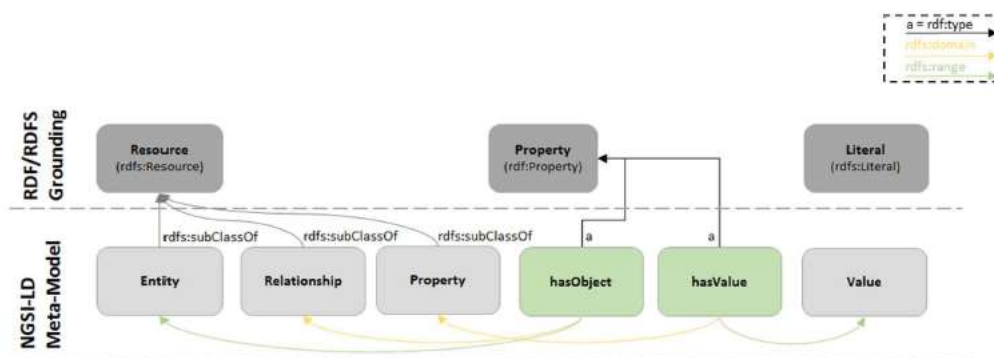


Figure 18: NGSI-LD Core Meta-Model

The Core Meta-Model is shown in figure 18, where it can be seen that:

- Each Entity is identified by a name and can have zero or more attributes
- The Properties define the status of the entity, which can be either static or dynamic
- The Relationships (unidirectional association with a Linked entity), define the topology of the data model, by depicting a network of dependencies among the entities that improve the context definition.

Thus, to model the context from the entities being used the values of the parameters arising from the sensors at the physical layer have to be assigned, as well as their corresponding meta-data, and establish the relationships linking them in their dedicated attributes.³⁷

The Entity	Notes
Has an id	URI/URN. The id must be unique.

Has a type	Fully qualified URI. Short-hand strings for types, mapped to fully qualified URIs through the JSON-LD @context
Has a series of properties	Name, address, category, etc.
Has a series of properties-of-properties	Verified field of address.
Has a series of relationships	Object corresponds to another data entity URI/URN.
Has a series of properties-of-relationships	Holds additional information about relationship.
Has a series of relationship-of-relationships	Hold the URI/URN of another relationship

The following implementations shall support the NGSILD meta-model as follows: ³⁸

- An NGSILD Entity is a subclass of `rdfs:Resource` (RDFS is a semantic extension of RDF and all things described by RDF are resources, and are instances of the class `rdfs:Resource`. This is the class of everything and all other classes are subclasses of this class. `rdfs:Resource` is an instance of `rdfs:Class`)
- An NGSILD Relationship is a subclass of `rdfs:Resource`.
- An NGSILD Property is a subclass of `rdfs:Resource`.
- An NGSILD Value shall be either a `rdfs:Literal` (literal values such as strings and integers) or a node object (in JSON-LD language) to represent complex data structures.
- An NGSILD Property shall have a value, stated through `hasValue`, which is of type `rdfs:Property` (`rdfs:Property` is the class of RDF properties, being an instance of `rdfs:Class`).
- An NGSILD Relationship shall have an object stated through `hasObject` which is of type `rdf:Property`.

RDFS (RDF Schema) provides a data-modelling vocabulary for RDF data. It is a semantic extension of RDF, providing mechanisms for describing groups of related resources and the relationships between them. RDFS is written in RDF and its resources are used to determine characteristics of other resources, such as the domains and ranges of properties.

The NGSILD meta model allows the advantages of RDF and Linked Data standards on top of the expressive Property Graphs (which are de facto industry standards from their use in graph databases), by providing reification (statement about statements). This allows:

- Export/import from/to [Property graph]/[RDF graph]
- Can be easily expressed in JSON-LD
- Model is suitable for SPARQL queries

3.3.1.2 Cross-domain ontology

The NGSILD Cross-Domain Ontology (figure below) is a set of generic, transversal classes which are aimed at avoiding conflicting or redundant definitions of the same classes in each of the domain-specific ontologies.

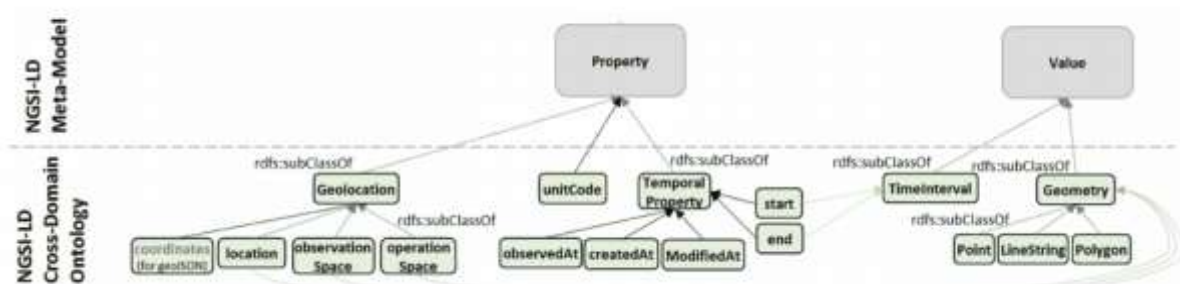


Figure 19: NGS-LD Cross-Domain Information Model

Frequent examples are the geo-properties, such as coordinates where the sensor is located, and the time related properties, such as observation date when the value of the magnitude was measured.

The NGS-LD cross domain ontology introduces the following concepts, with their implementations:

- Geo Properties: Are intended to convey geospatial information and implementations.
- Temporal Properties: They are non-reified Properties (represented only by its Value) that convey temporal information for capturing the time series evolution of other Properties.
- "unitCode" Property: A Property intended to provide the units of measurement of an NGS-LD Value.
- Geometry Values: They are a special type of NGS-LD Value intended to convey geometries corresponding to geospatial properties.
- Time Values: They are a special type of NGS-LD Value intended to convey time instants or intervals representations.

The Cross-Domain core properties that give context to the information include the following:

- location → Geospatial location encoded as GeoJSON.
- observeAt → Observation timestamp.
- createdAt → Creation timestamp (entity, attribute).
- modifiedAt → Update timestamp.
- unitCode → units of measurement.

3.3.1.3 Domain Specific ontology

Below the Core Meta-Model and the Cross-Domain Ontology, the Domain Specific ontologies of vocabularies are devised, in Platoon this task is carried out in T2.3 “Data models”.

At this level, the specific model is defined with the properties that are needed for the particular use case covered by the solution implementing the API. To implement this level, it is necessary the use of domain specific ontologies or, usually, combinations of several ontologies. The goal of the data model is to describe a given part of the real world, to encode the information extracted in this way and to share it with the components responsible for their management and analysis.

An ontology can represent a certain phenomenon, topic, or subject area through the description of classes, properties and instances (also known as individuals). Classes are abstract groups, sets, or collections of individuals and represent ontology concepts. Furthermore, these classes can have a hierarchical relation and can be arranged in taxonomies of superclasses and subclasses. Properties represent features or characteristics of individuals as well as the relationship between them. Finally, instances represent individuals of the classes described in the ontology.

Ontologies can be constructed based on different ontology languages such as the Web Ontology Language (OWL). OWL itself is based on RDF and RDFS, thus the vocabulary used for defining ontologies is a combination of concepts defined in RDF, RDFS and OWL. Certainly, an ontology language provides the expressive capability to encode knowledge about a specific domain and is often complemented with inference rules or validation rules that support the processing of such knowledge.

Once the purpose and the level of detail of the ontology are clear, it is necessary to define the concepts, properties and relationships that suit this purpose. Instead of creating an ontology from scratch, it is a best practice to reuse existing ontologies when possible.

For entities (real world devices, databases, or other information sources) they can be created by extending the Cross Domain Ontologies and the Core Meta Model, with specialized terms drawn from other ontologies.

From the PLATOON use cases, the ontologies to be reused or extended are SAREF, SEAS, Ontowind, CIM, S4BLD, skos, foaf, dcat, vcard, dcterms, brick, xsd, semanco, S4CITY, th, dogont, pep, sch, ifc, bot, bonsai, ssn, sosa, fiemster, S4ENER, time, schema, oema and icc.

Specific PLATOON domain ontologies have been created. Some examples are shown below, for the whole catalog please refer to the deliverable D2.3 “Data models”.

- HVACOntology defines the HVAC vocabulary for PLATOON, importing seas:SystemOntology and saref ontology
 - IRI → <https://w3id.org/platoon/HVACOntology>
- ElectricPowerSystemOntology defines electric power system concepts. Plt:ElectricPowerSystemOntology imports seas:ElectricPowerSystemOntology and seas:StreetLightSystemOntology, extending seas:ElectricPowerSystemOntology
 - IRI → <https://w3id.org/platoon/ElectricPowerSystemOntology>

3.3.2 A guide to Context

Context can be summarized to be nothing but a list of key-values, as shown in figure 20. ³⁹

```
"@context": {  
  "P1": "https://a.b.c/attributes/P1",  
  "P2": "https://a.b.c/attributes/P2",  
  "P3": "https://a.b.c/attributes/P3",  
  ...  
}
```

Figure 20: Context notion simplified

The key (e.g. “P1”) is an alias for the longer name (“<https://a.b.c/attributes/P1>”). This alias provides a way to write really long strings as well as giving the user the freedom to define their own aliases.

3.3.2.1 Expansion and Compaction

An NGSI-LD broker and the other components that will use the API use the context to expand and compact the shortnames that are part of the payload data or that come in as a URI parameter.

Inside the payload data, the context helps expand/compact:

- The Entity Type
- The Property Names (in all levels)
- The Relationship Names (in all levels)

The term Attribute is used to refer to Properties, Relationships, Properties-of-Properties, etc.

So, for example, if the `GET ngsi-ldv1/entities?type=T2` is issued, then T2 will be expanded to the current context and then looked up to find any matching entities. All entities are stored in a fully expanded form, which is the real value of the entity type, attribute name, value, etc. Before returning the resulting payload, all expandable/compactable items are compacted according to the context as follows:

- T2 is expanded (to "<https://uri.etsi.org/ngsi-ld/default-context/T2>", for example)
- All entities with that type are retrieved from storage
- All those entities are compacted (attribute names, entity type ...) => "<https://uri.etsi.org/ngsi-ld/default-context/T2>" goes back to being "T2"
- The response is composed from the compacted entities

3.3.2.2 The Core Context

Components that use the API have a default context built in, the Core Context. The Core Context defines the core of any NGSI-LD broker and its definitions (key-values) override any user-supplied definition. So, when a term is looked up for expansion or compaction in the context, the entire context is searched to the end, and the last hit overrides any previous hit. Therefore, the Core Context is the last context to be searched.

The core context (<https://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-context-v1.3.jsonld>) is fundamental to NGSI-LD and it is added by default to any context sent to a request. @context defines elements such as id and type and terms such as Property and Relationship.

3.3.2.3 The default URL

During lookup in the context, all entity types and attribute names are expanded and if no match is found in the provided context, then a special field “@vocab” of the Core Context is used. This is known as the default URL.

```
"@vocab": "https://uri.etsi.org/ngsi-ld/default-context/"
```

For example, if an attribute has the name “P7” and it is not found anywhere, then it will end up expanded according to the default URL, as follows:

```
"https://uri.etsi.org/ngsi-ld/default-context/P7"
```

3.3.2.4 Content-type and Context

There are two different ways to supply the context in a NGSI-LD request:

- via a HTTP header called `Link`
- as part of the payload

In the context is passed in the `Link` HTTP header, then the `Context-Type` must be **application/json**, whereas if it is passed in the payload data, then the `Context-Type` must be **application/ld+json**.

3.3.2.5 Compound context

Contexts inside the payload data can be expressed in many ways:

- A JSON String (a URL that refers to the context)
- “@context”: “url-to-context”
- A JSON Object that is the context, i.e. a list of key-values

```
"@context": {
  "P1": "https://a.b.c/attributes/P1",
  "P2": "https://a.b.c/attributes/P2",
  ...
}
```

- A JSON Array of string (URLs referring to the context)

```
"@context": [
  "url-to-context1",
  {
    "P1": "https://a.b.c/attributes/P1",
    "P2": "https://a.b.c/attributes/P2",
    ...
  }
]
```

- A JSON Array of a combination of strings and objects

```

"@context": [
  "url-to-context1",
  {
    "P1": "https://a.b.c/attributes/P1",
    "P2": "https://a.b.c/attributes/P2",
    ...
  }
]

```

3.3.2.6 Value Expansion

The value of a key-value in a context can be a little more complex than just a string. This is shown below:

```

"@context": {
  "P1": {
    "@id": "https://a.b.c/attributes/P1",
    "@type": "vocab"
  },
  ...
}

```

If an alias has a complex value in the @context, and that complex value contains a member @type that equals @vocab, then also the value of that attribute (an attribute is a Property or Relationship) is expanded according to the context. However, this only happens if it is found inside that very same context, otherwise the value is untouched.

JSON-LD allows two applications to use shortcut terms to communicate with one another more efficiently without losing accuracy. Context is used to map terms to URIs.

```

{
  "@context": {
    "name": "http://schema.org/name",
    ↑ This means that 'name' is shorthand for 'http://schema.org/name'
    "image": {
      "@id": "http://schema.org/image",
      ↑ This means that 'image' is shorthand for 'http://schema.org/image'
      "@type": "@id"
      ↑ This means that a string value associated with 'image'
      should be interpreted as an identifier that is an IRI
    },
    "homepage": {
      "@id": "http://schema.org/url",
      ↑ This means that 'homepage' is shorthand for 'http://schema.org/url'
      "@type": "@id"
      ↑ This means that a string value associated with 'homepage'
      should be interpreted as an identifier that is an IRI
    }
  }
}

```

Figure 21: JSON-LD context

Figure 21 shows that a context is introduced using an entry with the key @context and may appear within a node or value object.

A context is introduced using the key `@context`. Each Data Model shall have a JSON-LD `@context`, providing an unambiguous definition by mapping terms to URIs. For practicality reasons, it is recommended to have a unique `@context` resource, containing all terms, subject to be used in every Platoon Data Model, the same way as <http://schema.org> does.

Contexts can either be directly embedded into the document or be referenced using a URL. If context from figure 21 can be retrieved, it can be referenced by adding a single line and allow a JSON-LD document to be expressed much more concisely as shown in figure 22.

```
{
  "@context": "https://json-ld.org/contexts/person.jsonld",
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

Figure 22: Example referencing a JSON-LD context

This allows developers to re-use each other's data without having to agree to how their data will interoperate. External JSON-LD context documents may contain extra information located outside the `@context` key, which is ignored when the document is used as an external JSON-LD context document.

A remote context can also be referenced using a relative URL, as shown in figure 23.

```
{
  "@context": "context.jsonld",
  "name": "Manu Sporny",
  "homepage": "http://manu.sporny.org/",
  "image": "http://manu.sporny.org/images/manu.png"
}
```

Figure 23: Example loading a relative context

3.3.3 Basic operations

The NGSI-LD API supports a number of operations, with messages expressed using JSON-LD. It allows context consumers and context producers to interact with context information systems. Not all conceivable operations are supported in the API, but rather a subset which is as simple as possible, yet complex enough to handle the vast majority of interactions.

These API operations allow applications to discover, query and explore the graph-based data by specifying any combination of entities, types, relationships and/or properties as criteria for data queries.

Context consumers can retrieve context information from the Context Broker and components using this API both, in a synchronous and asynchronous way.

Synchronous interactions are carried out by means of queries, which have to be launched directly by the consumer and supplies data describing the current state of the context, or some specific parameters, at the time when it has been required.

Moreover, asynchronous interactions are not answers to queries from the context consumers but occur as a response to events, mainly changes in the state of the context. This service

requires a previous subscription of the context consumer, which will be updated with every change in the values of context parameters by sending notifications from the context Broker when they take place.

Thus, queries enable to get information of the context when the consumer need it, whereas subscriptions ease to be warned about changes reported by the context makers at any time.

The operations listed here are used to set a given data source as context producer, as well as to find context producers which have been already registered to link them to other entities.

3.3.3.1 CRUD operations view

The CRUD operations view will provide a general view of the NGSI-LD operations with the typical operations related to the storage and retrieval of information within a database.

CRUD Operations (Create, Read, Update and Delete) are the four basic functions of persistent storage. For a smart system based on NGSI-LD, CRUD actions allow the developer to manipulate the context data within the system. Every CRUD operation is clearly defined within the NGSI-LD specification, so all NGSI-LD compliant architecture components offer the same interface with the same NGSI-LD operations.

There are four endpoints used for CRUD operations on an individual data entity. These follow the usual rules for hierarchical entities within RESTful applications.

When requesting data or modifying individual entities, the various CRUD operations map naturally to HTTP verbs.

- GET - for reading data
- POST - for creating new entities and attributes
- PATCH - for amending entities and attributes
- DELETE - for deleting entities and attributes

One group of NGSI-LD operations allow Context Producers to create NGSI-LD entities (insert an object with a defined URI into the systems) and allow Context Consumers to retrieve and subscribe to entities.

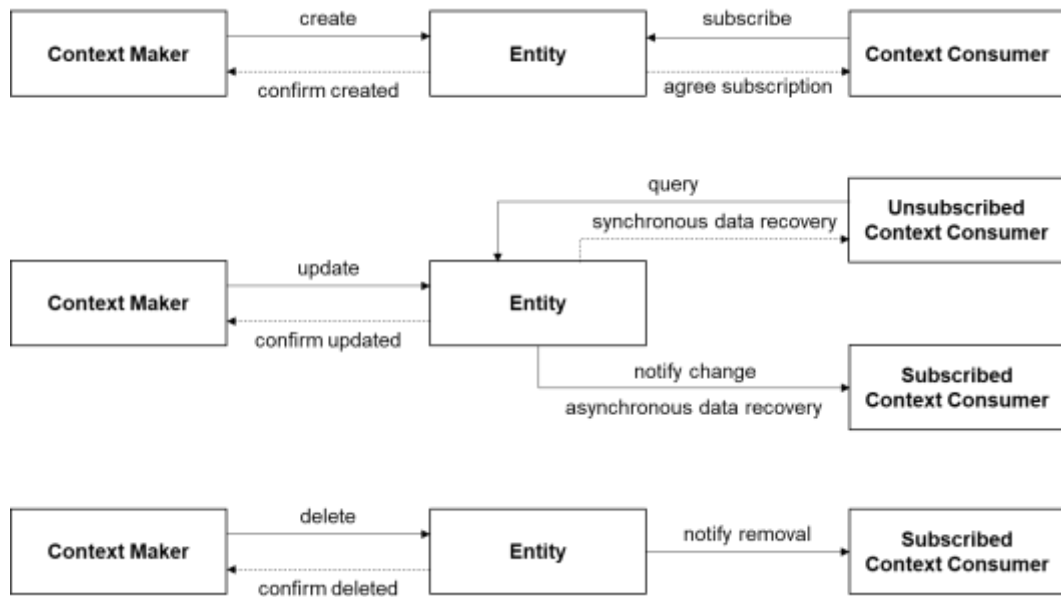


Figure 24: NGSi-LD API operation overview

Figure 24 shows the data transfer-related operations from NGSi-LD REST API. In the upper part of the chart, a context maker creates a new data entity, by means of a context data supply operation, and a context consumer establishes a subscription with a context data subscription operation.

In the middle part, a context consumer is automatically notified about an update event taking place at the entity it is subscribed to, whereas unsubscribed consumers will not recover the updated information until they send the corresponding query since the communication in this case is synchronous. Events leading to the removal of entities are also notified by asynchronous communications to the subscribed consumers.

3.3.3.2 Context Information Provision and Consumption

Context Information Provision and Consumption refer to the operations where context producers can create, update and delete an NGSi-LD entity.

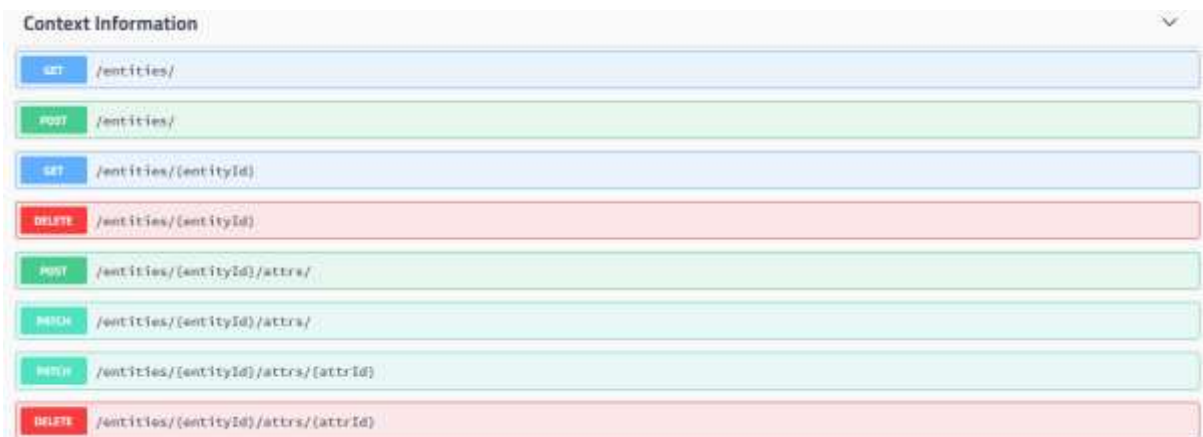


Figure 25: NGSi-LD API Context Information

Figure 25 shows the Swagger representation of the information provision and consumption operations.

Action	Operation
Create Entity	POST /ngsi-ld/v1/entities
Delete Entity	DELETE /ngsi-ld/v1/entities/{entityId}
Append new attributes to Entity	POST /ngsi-ld/v1/entities/{entityId}/attrs/ <i>If attribute already exists it is overwritten</i>
Update Entity Attributes	PATCH /ngsi-ld/v1/entities/{entityId}/attrs/ <i>Error if attribute does not exist</i>
Delete Entity Attribute	DELETE /ngsi-ld/v1/entities/{entityId}/attrs/{attributeName} <i>Error if attribute does not exist</i>
Retrieve Entity by id	GET /ngsi-ld/v1/entities/{entityId}?attrs=<attrList> <i>Returns a Json object representing the concerned Entity.</i> <i>If attrs not present, then all Entity Attributes will be retrieved.</i>
Query Entities	GET /ngsi-ld/v1/entities/? type<typeList> &id=<idList> &idPattern=<RegExp> &q=<Expression> &attrs=<attrList> <i>Returns Entity list as a JSON Array, where pagination is available</i>
Geo-query	GET /ngsi-ld/v1/entities/? &georel <i>Geo-relationship (near, contains, intersects, overlaps, coveredBy, disjoint etc.)</i> &geometry <i>GeoJSON reference geometry (point, polygon, line, box, etc.)</i> &coordinates <i>Array of GeoJSON coordinates encoded as string</i>

Table 1: Context Information Provision Operations

Any newly created entity must have `id` and `type` attributes and a valid `@context` definition. All other attributes are optional and will depend on the system. Though, if additional attributes are present, each should specify both a type and a value.

The response will be 201 – Created or 409 – Conflict (if the entity were already to exist in the context)

The following example adds a new `TemperatureSensor` entity to the context by making a POST request to the `/ngsi-ld/v1/entities` endpoint. ⁴⁰

```
curl --location --request POST 'http://localhost:1026/ngsi-ld/v1/entities/' \
--header 'Content-Type: application/json' \
--header 'Link: <http://context-provider:3000/data-models/ngsi-context.jsonld>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"' \
--data-raw '{
  "id": "urn:ngsi-ld:TemperatureSensor:001",
  "type": "TemperatureSensor",
  "category": {
    "type": "Property",
    "value": "sensors"
  },
  "temperature": {
    "type": "Property",
    "value": 25,
    "unitCode": "CEL"
  }
}
```

Figure 26: Entity creation example

The next example adds a new `batteryLevel` Property and a `controlledAsset` Relationship to the existing `TemperatureSensor` entity with `id=urn:ngsi-ld:TemperatureSensor:001`. By making a POST request to the `/ngsi-ld/v1/entities/{entityId}/attrs/` endpoint.

The payload should consist of a JSON object holding the attribute names and values as shown.

All `type=Property` attributes must have a value associated with them and all `type=Relationship` attributes must have an object which holds the URN of another entity. Well-defined common metadata elements such as `unitCode` can be provided as strings, while all other metadata should be passed as JSON object with its `on type` and `value` attributes.

Subsequent requests using the same `id` will update the attribute value.

```
curl --location --request POST 'http://localhost:1026/ngsi-ld/v1/entities/urn:ngsi-ld:TemperatureSensor:001/attrs/' \
--header 'Content-Type: application/json' \
--header 'Link: <http://context-provider:3000/data-models/ngsi-context.jsonld>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"' \
--data-raw '{
  "batteryLevel": {
    "type": "Property",
    "value": 0.8,
    "unitCode": "C63"
  },
  "controlledAsset": {
    "type": "Relationship",
    "object": "urn:ngsi-ld:Building:barn002"
  }
}
```

Figure 27: Attribute creation example

The consumption operations allow a Context Consumer to retrieve or query for NGSI-LD entities, being able to filter out Entities by Attribute Value (target value of a property or the target value of a Relationship).

For read operations the `@context` must be supplied in a Link header.

The following example reads the full context from an existing `TemperatureSensor` entity with a known id. The `TemperatureSensor` URN is returned as normalized JSON-LD with additional metadata. By default the `@context` is returned in the payload body.

```
curl --location --request
    GET 'http://localhost:1026/ngsi-ld/v1/entities/urn:ngsi-ld:TemperatureSensor:001?options=sysAttrs' \
--header '
    Link: <http://context-provider:3000/data-models/ngsi-context.jsonld>; rel="http://www.w3.org/ns/json-ld#context";
    type="application/ld+json"'
```

The next example reads the value of a single attribute (temperature) from an existing `TemperatureSensor` entity with a known id. Here it is shown that the sensor `urn:ngsi-ld:TemperatureSensor:001` is reading at 25°C.

```
curl --location --request
    GET 'http://localhost:1026/ngsi-ld/v1/entities/urn:ngsi-ld:TemperatureSensor:001?attrs=temperature' \
--header
    'Link: <http://context-provider:3000/data-models/ngsi-context.jsonld>; rel="http://www.w3.org/ns/json-ld#context";
    type="application/ld+json"'
```

The next example reads the key-value pairs from the context of an existing `TemperatureSensor` entities with a known id. The response contains an unfiltered list of context data from an entity containing all the attributes. Since the `Accept: application/json` was set, the payload body does not contain any context.

```
curl --location --request GET 'http://localhost:1026/ngsi-ld/v1/entities/urn:ngsi-ld:TemperatureSensor:001/?options=keyValues' \
\
--header 'Link: <http://context-provider:3000/data-models/json-context.jsonld>; rel="http://www.w3.org/ns/json-ld#context";
    type="application/ld+json" ' \
--header 'Accept: application/json'
```

3.3.3.3 Context Subscriptions

Context Subscription refer to the operations through which regular or event-driven update notifications of the context of one or more Entities can be created, updated, retrieved and/or queried for.

Subscriptions can be on a particular Entity id/ set of Entities, on an Entity Type/set of EntityTypes and/or a particular set of watched attributes.

Notifications are delivered via HTTP POST requests, as a JSON Array of affected Entities as per the notification details described by the subscription.

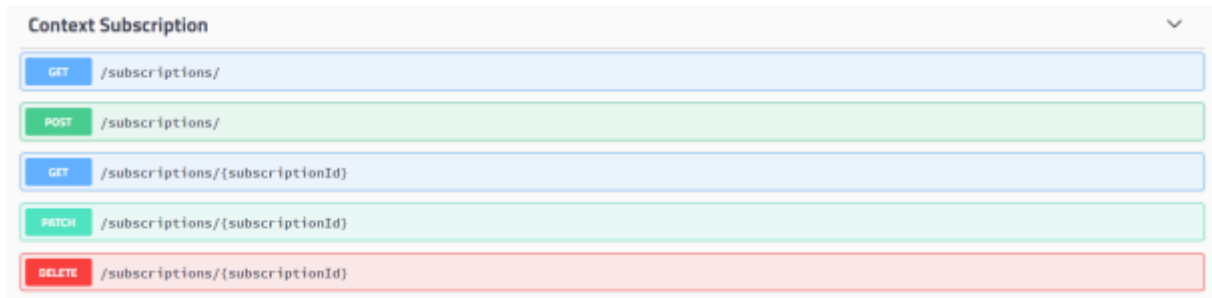


Figure 28: NGSi-LD Context Subscriptions

Figure 28 shows the Swagger for the NGSi-LD context subscription, and the table below shows the corresponding actions to the CRUD operations.

Action	Operation
Create subscription	POST /ngsi-ld/v1/subscriptions
Update subscription	PATCH /ngsi-ld/v1/subscriptions/{subscriptionId}
Remove subscription	DELETE /ngsi-ld/v1/subscriptions/{subscriptionId}
List subscriptions	GET /ngsi-ld/v1/subscriptions/
Retrieve subscriptions by id	GET /ngsi-ld/v1/subscriptions/{subscriptionId}

Table 2: Context Subscription Operations

3.3.3.4 Context Source Registration

Operations to register new context makers as well as to delete existing records of context makers in a synchronous manner. When registered, the entities linked to the context maker have to be stipulated, by direct assignment or by filtering by some criteria. Relationships with the context entities may also be updated with operations belonging to this group.

Figure 29 shows the Context Subscription swagger for the NGSi-LD API.

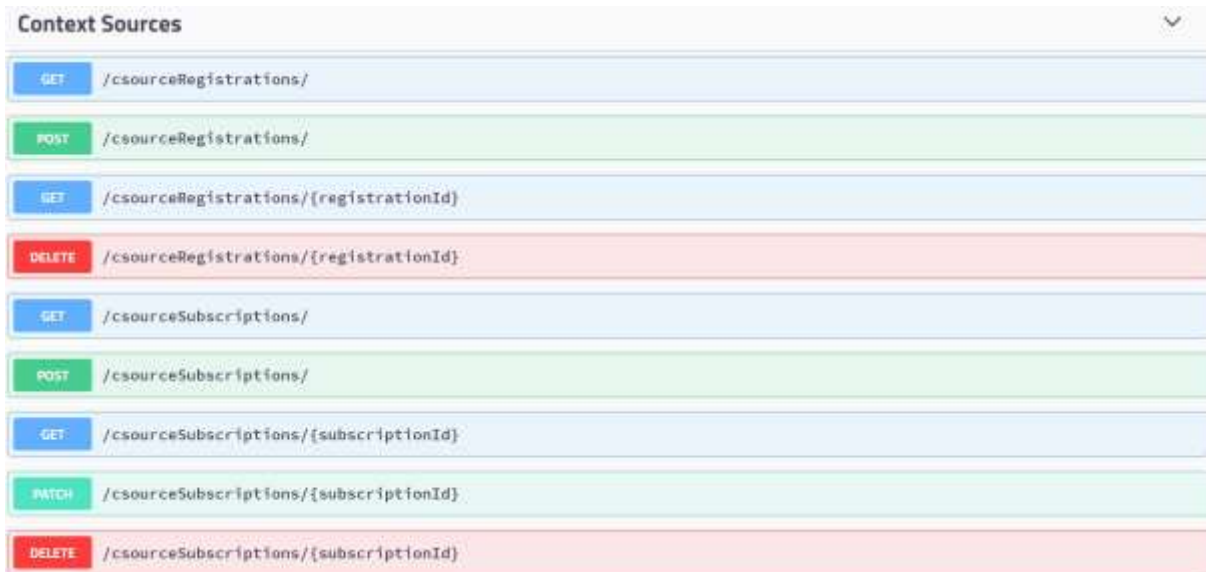


Figure 29: NGSi-LD Context Source Registration

Table 3, shows the corresponding actions to the CRUD operations for Context Source Registration.

Action	Operation
Create Context Source Registration	POST /ngsi-ld/v1/registrations
Update Context Source Registration	PATCH /ngsi-ld/v1/registrations/{registrationId}
Remove Context Source Registration	DELETE /ngsi-ld/v1/registrations/{registrationId}
Query Context Source Registration	GET /ngsi-ld/v1/registrations/?<Same as Query Entities>
Retrieve Context Source Registration	GET /ngsi-ld/v1/registrations/{registrationId}

Table 3: Context Source Registration Operations

3.3.3.5 Context Entity Batch Operations

Batch operations allow users to modify multiple data entities with a single request. All batch operations are mapped to the POST HTTP verb.

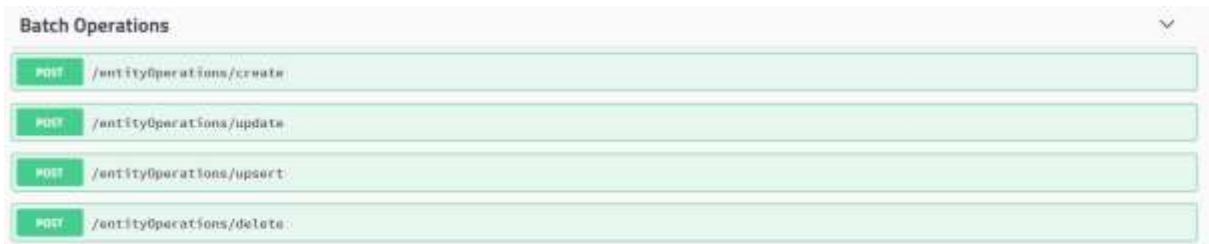


Figure 30: NGSI-LD Batch operations

For example, the batch processing endpoint can be used to add three new `TemperatureSensor` entities to the context, using the `/ngsi-ld/v1/entityOperations/create` endpoint.

```
curl --location --request POST 'http://localhost:1026/ngsi-ld/v1/entityOperations/create' \
--header 'Content-type: application/json' \
--header 'Link: <http://context-provider:3000/data-models/ngsi-context.jsonld>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"' \
--header 'Accept: application/ld+json' \
--data-raw '[
  {
    "id": "urn:ngsi-ld:TemperatureSensor:002",
    "type": "TemperatureSensor",
    "category": {
      "type": "Property",
      "value": "sensor"
    },
    "temperature": {
      "type": "Property",
      "value": 20,
      "unitCode": "CEL"
    }
  },
  {
    "id": "urn:ngsi-ld:TemperatureSensor:003",
    "type": "TemperatureSensor",
    "category": {
      "type": "Property",
      "value": "sensor"
    },
    "temperature": {
      "type": "Property",
      "value": 2,
      "unitCode": "CEL"
    }
  },
  {
    "id": "urn:ngsi-ld:TemperatureSensor:004",
    "type": "TemperatureSensor",
    "category": {
      "type": "Property",
      "value": "sensor"
    },
    "temperature": {
      "type": "Property",
      "value": 100,
      "unitCode": "CEL"
    }
  }
]
```

Figure 31: Batch create Entity/Attribute example

The request will fail if any of the attributes already exist in the context. The response highlights which actions have been successful and the reason for failure (if any has occurred).

The next example adds or amends two `TemperatureSensor` entities in the context.

- If an entity already exists, the request will update that entity's attributes.
- If an entity does not exist, a new entity is created.

A subsequent request containing the same data will also succeed.

```

curl --location --request POST 'http://localhost:1026/ngsi-ld/v1/entityOperations/upsert' \
--header 'Content-Type: application/json' \
--header 'Link: <http://context-provider:3000/data-models/ngsi-context.jsonld>; rel="http://www.w3.org/ns/json-ld#context"; type="application/ld+json"' \
--header 'Accept: application/ld+json' \
--data-raw '[
  {
    "id": "urn:ngsi-ld:TemperatureSensor:002",
    "type": "TemperatureSensor",
    "category": {
      "type": "Property",
      "value": "sensor"
    },
    "temperature": {
      "type": "Property",
      "value": 22,
      "unitCode": "CEL"
    }
  },
  {
    "id": "urn:ngsi-ld:TemperatureSensor:003",
    "type": "TemperatureSensor",
    "category": {
      "type": "Property",
      "value": "sensor"
    },
    "temperature": {
      "type": "Property",
      "value": 22,
      "unitCode": "CEL"
    }
  }
]'

```

Figure 32: Batch Create/Overwrite new entities example

3.3.3.6 Query Pagination

NGSI-LD Query operations can potentially return a result set including a large number of NGSI-LD Elements by dividing and obtaining the information in different blocks/queries, so that pagination of query results shall be supported by compliant implementations. Nonetheless, the NGSI-LD API is agnostic about specific pagination mechanisms and only defines the behaviour that shall be observed by NGSI-LD Systems.

There are four parameters to be taken into account:

1. limit: Number of elements per page (20 by default, maximum 1000)
2. offset: number of elements to jump (from the beginning)
3. count (optional): only returns the total number of elements of the petition
4. orderBy (optional): attributes that can be ordered (creation date by default)

An example would be: `GET /ngsi-ld/v1/entities?limit=20offset=0&orderBy=temp`

3.3.3.7 Temporal evolution

The temporal representation (e.g. its historical evolution), is composed of the sequence of instances during a period of time.

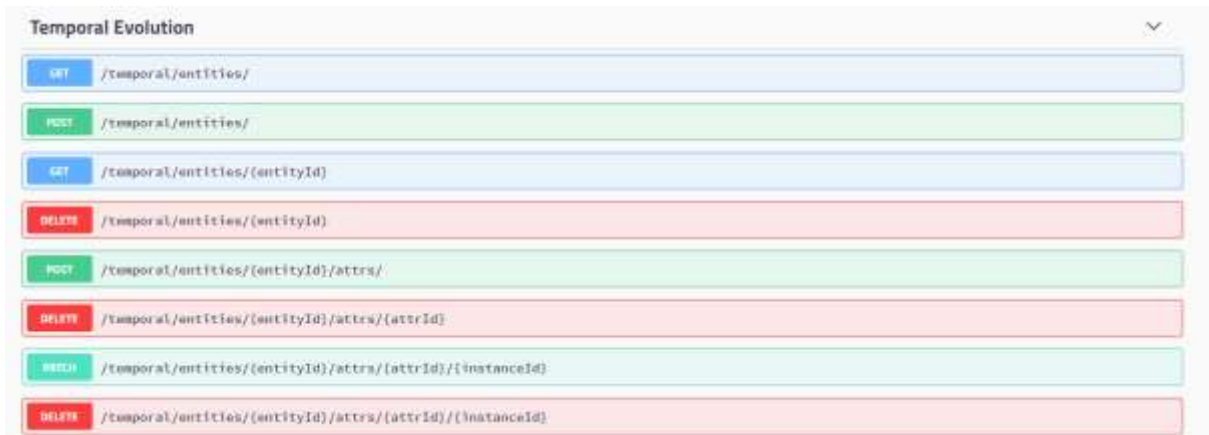


Figure 33: NGSi-LD temporal representation operations

The following example shows a query to retrieve all the operation history of Entities of type Lightbulb that are not outside between the 1st of August at noon the 1st of August at 01 PM.

```
GET /ngsild/temporal/entities/?type=Lightbulb&q=location!
=Outside&attrs=operation,location&timerel=between&time=2018-08-
01:12:00:00Z&endTime=2018-08-01:13:00:00Z&options=temporalValues
Accept: application/ld+json
```

An example of a simplified HTTP response would be:

```
200 OK
Content-Type: application/ld+json
[
  {
    "id": "urn:ngsi-ld:Lightbulb:B9211",
    "type": "Lightbulb",
    "location": {
      "type": "Property",
      "values": [ ["livingRoom", ""] ]
    },
    "operation": {
      "type": "Property",
      "values": [
        [4, "2018-08-01T12:03:00Z"],
        [70, "2018-08-01T12:05:00Z"],
        [100, "2018-08-01T12:07:00Z"]
      ]
    },
    "@context": [
      "http://uri.etsi.org/ngsi-ld/v1/ngsi-ld-core-
      context.jsonld",
      "http://example.org/ngsi-ld/lightbulb.jsonld"
    ]
  }
]
```

4 Marketplace: TMForum APIs

The PLATOON marketplace is a component that is still in the design/development phase and it is planned that it will provide functionalities to publish, search and browse for different assets: data and data analytics tools. Asset offerings can be organised into groups/categories in a hierarchical fashion to allow for an easy navigation and discovery of them. Metadata define characteristics and properties of assets (metadata could include also way to exchange / access datasets) They may also be inherited from a higher level in a category hierarchy. The module lets the asset providers be able to define the technical description of the assets they own.

The marketplace APIs provide sellers the means for managing, publishing and generating revenue from the products, apps, data and services across the whole life cycle. Standard APIs (and its reference implementations) provided by the TM Forum API ecosystem will be specified.^{41 42 43}

TMForum ecosystem OpenAPIs (figure 34 and 35) are REST based APIs that enable rapid, repeatable and flexible integration among operations and management systems.



Figure 34: TM Forum API end-to-end

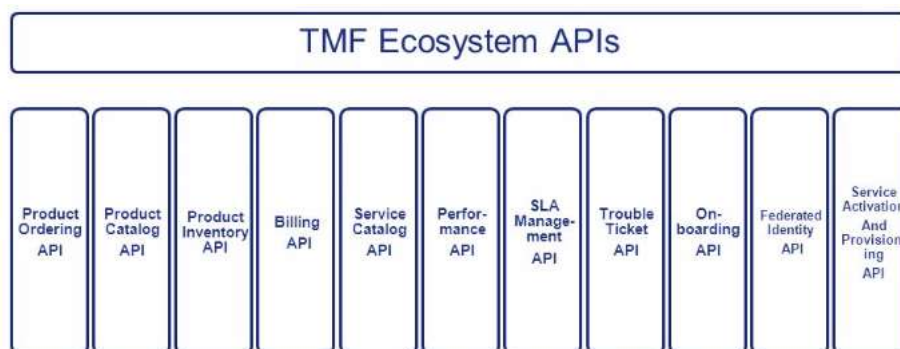


Figure 35: TMForum ecosystem APIs

4.1 Product Catalog Management API

Product Catalog Management API provides a catalog of products, allowing the management of the entire lifecycle of the elements, consultation, etc.

The product catalog management API uses the following fields:

- id - Unique identifier of the catalog
- href - URL pointing to the catalog info
- version - Version of the catalog
- lastUpdate - Date and time of the last update
- category - List of categories of the catalog. For each category the id, href, and name fields are included as described in Category Management section

- name - Name of the catalog
- lifecycleStatus - Current lifecycle status
- relatedParty - List of parties and its roles related to the current catalog. For each party, it is included the id and the href as described in the Party Management section. Additionally, it includes a role field specifying the role of the user in the current catalog

Product Catalog API performs the following operations on the resources (figures 36 - 38 show the Swagger specification):

- Retrieve an entity or a collection of entities depending on filter criteria
- Partial update of an entity (including updating rules)
- Create an entity (including default values and creation rules)
- Delete an entity
- Manage notification of events

The image shows a Swagger API specification for two resources: 'catalog' and 'category'. Each resource has five operations listed in a table-like format. The operations are color-coded by HTTP method: GET (blue), POST (green), PATCH (teal), and DELETE (red). Each entry includes the method, the endpoint path, a brief description, and a right-pointing arrow icon.

catalog			
GET	/catalog	List or find Catalog objects	↑
POST	/catalog	Creates a Catalog	↑
GET	/catalog/{id}	Retrieves a Catalog by ID	↑
PATCH	/catalog/{id}	Updates partially a Catalog	↑
DELETE	/catalog/{id}	Deletes a Catalog	↑
category			
GET	/category	List or find Category objects	↑
POST	/category	Creates a Category	↑
GET	/category/{id}	Retrieves a Category by ID	↑
PATCH	/category/{id}	Updates partially a Category	↑
DELETE	/category/{id}	Deletes a Category	↑

Figure 36: Product Catalog Management API Swagger operations I

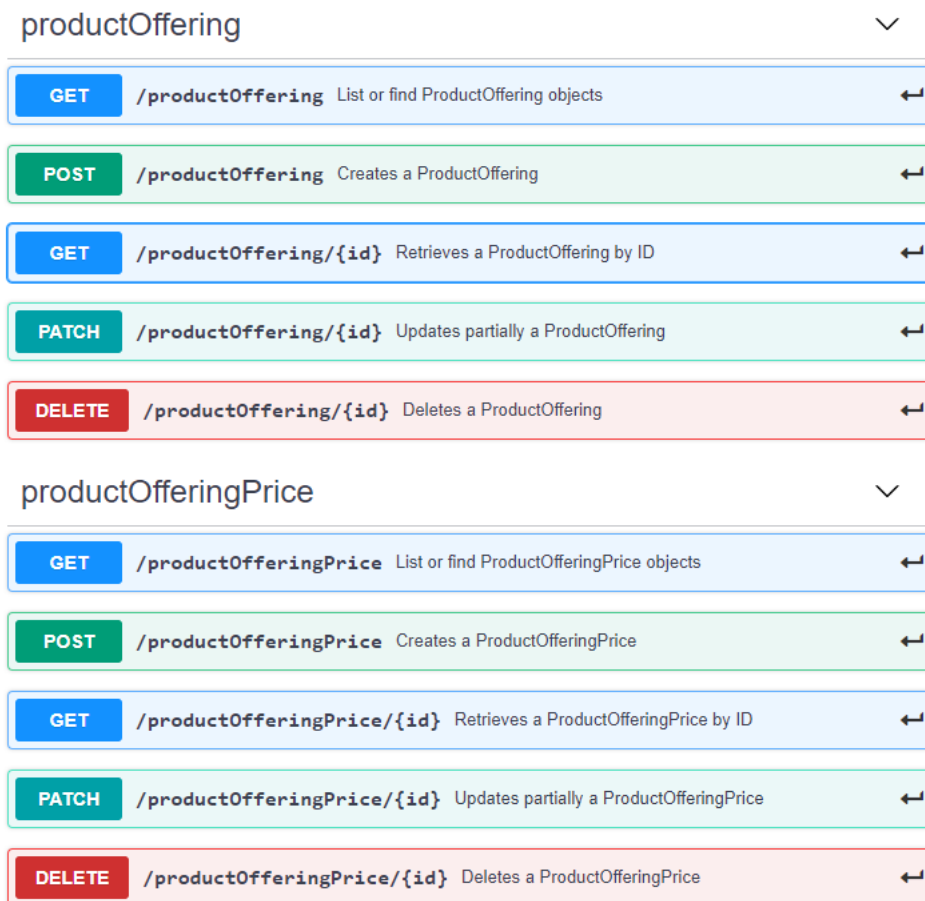


Figure 37: Product Catalog Management API Swagger operations II

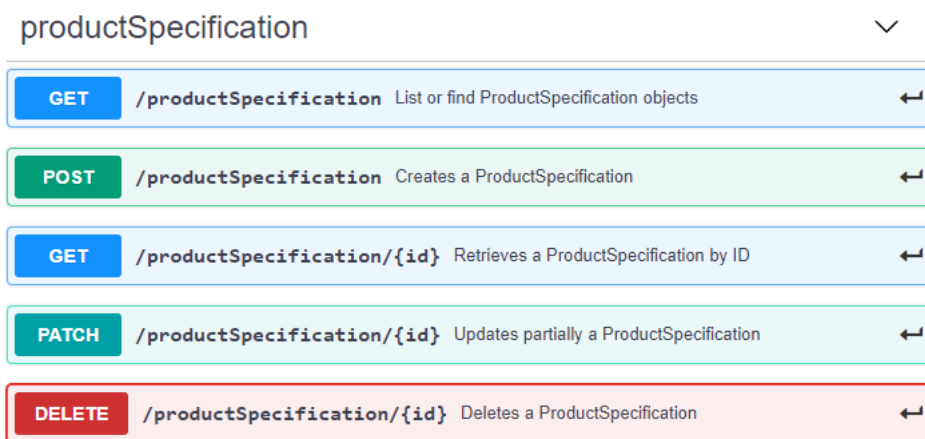


Figure 38: Product Catalog Management API Swagger operations III

The following example shows a request for creating a specific catalog (figure 39).

Request
POST {apiRoot}/catalog Content-Type: application/json <pre>{ "name": "Catalog Wholesale Business", "description": "This catalog describes Product Offerings and technical specifications intended to address the wholesale business segment.", "@type": "Catalog" }</pre>
Response
201 <pre>{ "id": "2355", "href": "https://mycsp.com:8080/tmf-api/productCatalogManagement/v4/Catalog/2355", "name": "Catalog Wholesale Business", "description": "This catalog describes Product Offerings and technical specifications intended to address the wholesale business segment.", "lastUpdate": "2017-08-27T00:00", "lifecycleStatus": "Tentative", "@type": "Catalog" }</pre>

Figure 39: Specific catalog creation example

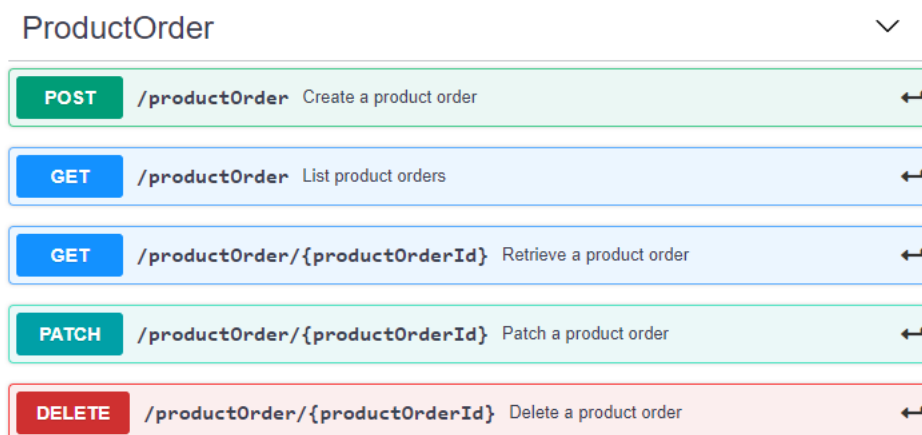
4.2 Order Management API

The Order Management API provides a standardized mechanism for placing a product order with all the necessary order parameters, dealing with the management of Product Orders made by customers. These include a set of order items each specifying an offering to be acquired. When creating an order, customers can select the value of the different configurable characteristics as well as the concrete pricing to be applied.

The fields managed by the API are as follows:

- id - Unique identifier of the order
- href - URL pointing to the product order info
- externalId - Id of the order given by customer, which can be used by them to identify the order in their own systems
- priority - Number between 1 and 4 (1 being the highest) that can be used by the customer to specify the priority of their orders
- description - Description of the product order
- state - Status of the order, relative to the status of the different order items
- orderDate - Date when the order was created
- completionDate - Date when the order was completed
- requestedStartDate - Order start date wished by the requestor
- requestedCompletionDate - Requested delivery date from the requestor perspective
- expectedCompletionDate - Expected delivery date amended by the provider
- notificationContact - Contact attached to the order to send back information regarding the current order

- note - List of extra information about the order. For each note is included the following info:
 - date - Date of the note
 - author - Author of the note
 - text - Text of the note
- relatedParty - Defines parties which are involved in the order and the role they are playing. For each party, it includes the id and the href as described in the Party Management section. Additionally, it includes a role field specifying the role of the user in the current product order
- orderItem - List of order items that have to be treated. For each order item the following information is managed:
 - id - Id of the order item relative to the product order (Only need to be unique within the order)
 - action - Type of the order item. Currently only add is supported (acquisition)
 - state - Status of the order item
 - billingAccount - Billing account selected by the customer to acquire the offering according to the Billing Management API section
 - productOffering - Product offering being acquired. It includes the id and the href of the product offering
 - product - Information provided to create the inventory product. It contains the selected characteristics and the selected pricing. The different fields managed by this object are the same as the described in the Inventory Management API Section



The image shows a Swagger UI snippet for the ProductOrder API. It lists five operations: POST, GET, GET, PATCH, and DELETE, each with its corresponding HTTP method, endpoint, and description. The endpoints are /productOrder, /productOrder, /productOrder/{productOrderId}, /productOrder/{productOrderId}, and /productOrder/{productOrderId} respectively. The descriptions are 'Create a product order', 'List product orders', 'Retrieve a product order', 'Patch a product order', and 'Delete a product order'. Each operation has a small icon on the right side.

Method	Endpoint	Description
POST	/productOrder	Create a product order
GET	/productOrder	List product orders
GET	/productOrder/{productOrderId}	Retrieve a product order
PATCH	/productOrder/{productOrderId}	Patch a product order
DELETE	/productOrder/{productOrderId}	Delete a product order

Figure 40: Product Order API Swagger ProductOrder operations

The Order API performs the following operations on product order (figure 40):

- Retrieval of a product order or a collection of product orders depending on filter criteria
- Partial update of a product order (including updating rules)
- Creation of a product order (including default values and creation rules)
- Deletion of product order (for administration purposes)
- Notification of events on product order.

cancelProductOrder		▼	
GET	/cancelProductOrder	List or find CancelProductOrder objects	←
POST	/cancelProductOrder	Creates a CancelProductOrder	←
GET	/cancelProductOrder/{id}	Retrieves a CancelProductOrder by ID	←

Figure 41: Product Order API Swagger cancelProductOrder operations

The cancelProductOrder resource is used to request a product order cancellation, performing the following operations (figure 41):

- Retrieval of a cancel product order or a collection of cancel product orders
- Creation of a cancel product order
- Notification of events on cancel product order.

Figure 42 shows an example of a request for retrieving ResourceOrder resources. It searches resource orders started on January 1st 2015 and the result items are shrunk to show only the ids (fields=id).

Request
GET /resourceOrderingManagement/resourceOrder?fields=id&orderDate="2015-01-01" Accept: application/json
Response
200 <pre>[{ "id": "986781" }, { "id": "986782" }, { "id": "986783" }]</pre>

Figure 42: Resource order retrieval example

Figure 43 shows an example of a request for deleting a Service order.

Request
DELETE /tmf-api/serviceOrdering/v4/serviceOrder/47
Response
204

Figure 43: Deleting service order example

4.3 Party Management API

Party is an abstract concept that represents an individual (person) or an organization that has any kind of relation with the enterprise. It is created to record an individual/organization before the assignment of any role.

- For the different individuals of the system, the following information is used:
- id - Id of the party. Corresponds with the username of the user in the system
- href - URL pointing to the party info
- gender - Gender of the individual owner of the account
- placeOfBirth - Place where the owner of the account was born
- countryOfBirth - Country where the owner of the account was born
- nationality - Nationality of the owner of the account
- maritalStatus - Marital status (married, divorced, widow, etc)
- birthDate - Date when the owner of the account was born
- title - Preferred title of the user (Mr., Dr., etc)
- givenName - First name of the user owner of the account
- familyName - Family name of the user owner of the account
- contactMedium - List of mediums that can be used to contact the user. Note that this information is public to all the users of the system, so this mediums are used as seller contact. Each medium contains the following fields:
 - type - Type of the contact medium. It could be Email, TelephoneNumber, or PostalAddress
 - preferred - If true, indicates that is the preferred contact medium
 - emailAddress - Full email address in standard format. This field is only used when the type is Email
 - number - Phone number. This field is only used when the type is TelephoneNumber
 - street1 - Describes the street. This field is only used when the type is PostalAddress
 - street2 - Complementary street description. This field is only used when the type is PostalAddress
 - city - City of the medium. This field is only used when the type is PostalAddress

- `postCode` - PostCode of the medium. This field is only used when the type is `PostalAddress`
- `stateOrProvince` - State or province of the medium. This field is only used when the type is `PostalAddress`
- `country` - Country of the medium. This field is only used when the type is `PostalAddress`

An individual represents a single human being while an organization refers to a group of people identified by shared interests/purpose.

The resources provided by the API are the following:

- Organization
- Individual
- Hub

Party Management API performs the following operations (figure 44):

- Retrieve an organization or an individual
- Retrieve a collection of organizations or individuals according given criteria
- Create a new organization or a new individual
- Update an existing organization or an existing individual
- Delete an existing organization or an existing individual
- Notify events on organization or individual

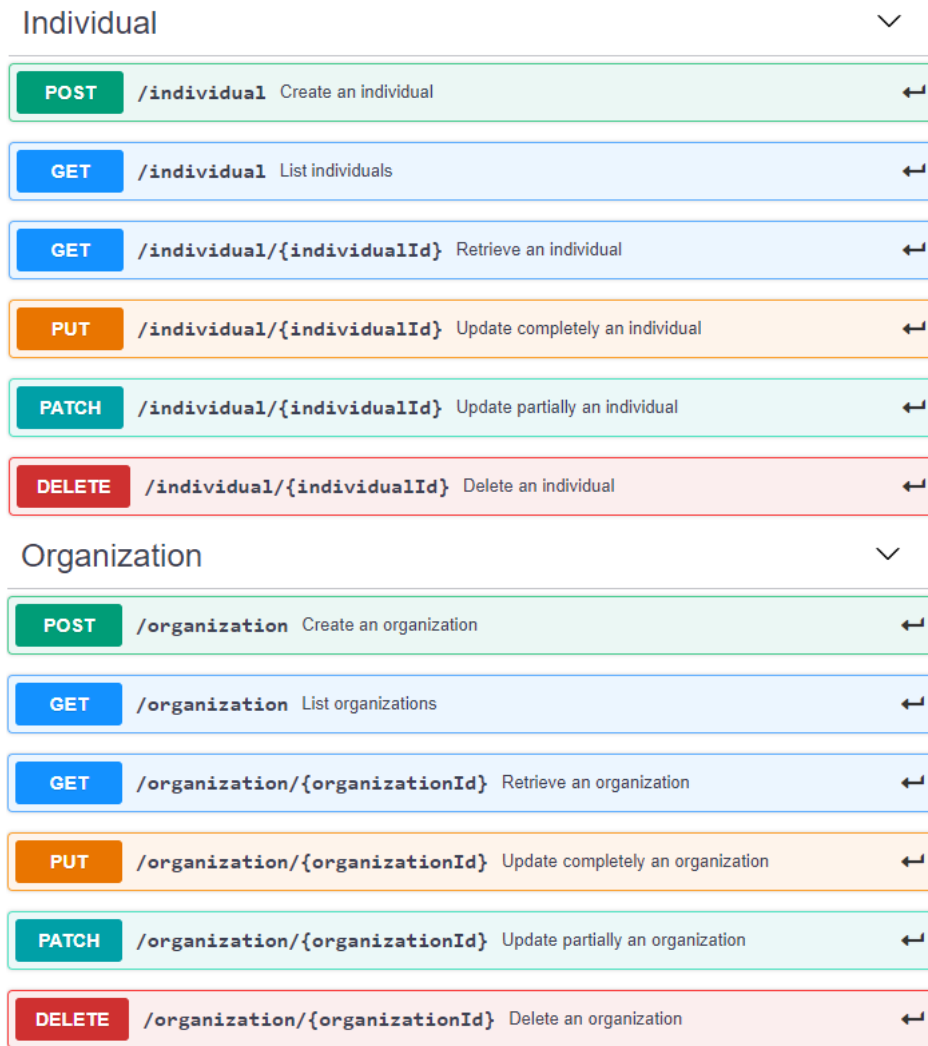


Figure 44: Party Management API Swagger operation

The following example shows a request for retrieving individual resources.

Request
GET /serverRoot/tmf-api/party/v4/individual?fields=id,familyName,givenName&status=validated&creditRating.ratingScore=700 Accept: application/json
Response
200 <pre>[{ "id": "42", "@type": "individual", "givenName": "Jane", "familyName": "Lamborgizzia" }, { "id": "52", "@type": "individual", "givenName": "Paul", "familyName": "Newman" }]</pre>

Figure 45: Individual resource retrieval example

An individual resource can be created as shown in figure 46.

Request
POST /serverRoot/tmf-api/party/v4/individual Content-Type: application/json <pre>{ "givenName": "Jane", "familyName": "Lamborgizzia" }</pre>
Response
201 <pre>{ "id": "42", "href": "https://serverRoot/tmf-api/party/v4/individual/42", "@type": "individual", "givenName": "Jane", "familyName": "Lamborgizzia", "status": "initialized" }</pre>

Figure 46: Individual resource creation example

Figure 47 shows how partial updates can be done on an organization. The attribute is changed using the json-patch.

Request
PATCH /serverRoot/tmf-api/party/v4/organization/42 Content-Type: application/merge-patch+json { "isLegalEntity": true }
Response
200 { Similar JSON as in GET response with isLegalEntity changed }

Figure 47: Organization resource partial update example

An organization entity can be deleted as shown in figure 48.

Request
DELETE {apiRoot}/organization/128
Response
204

Figure 48: Organization entity deletion example

4.4 Usage Management API

The Usage Management API provides a standardized mechanism for usage document management. A usage is an occurrence of employing a Product, Service or Resource for its intended purpose which is of interest to the business and can have charges applied to it.

Usage documents contain the actual usage made of an acquired product, including the information defined in its Usage Specification. This API manages both rated and non-rate usages.

This API manages the following fields:

- id - Id that identifies the Usage
- href - URL pointing to the Usage
- date - Date and time when the usage was created
- type - Type of the Usage document. It refers to the name of a Usage Specification
- description - Description of the Usage Document
- status - Current status of the Usage
- usageSpecification - Reference to the Usage Specification that defines the current usage. It includes its id and its href

- usageCharacteristic - List with the values of the characteristics defined in the Usage Specification
- relatedParty - List of parties that are involved in the Usage. At least this list must include the user the made the usage of the service with the role customer
- ratedProductUsage - In case the customer had already been charged for the usage made in the current document, this field would contain the amount generated by the document, taking into account the pricing model of the product. This field contains the following fields:
 - ratingDate - Date and time when the document was rated
 - usageRatingTag - Fixed to usage
 - isBilled - Specifies if the rated document has been already charged
 - ratingAmountType - Fixed to Total
 - taxIncludedRatingAmount - Total amount generated by the Usage
 - taxExcludedRatingAmount - Amount without taxes generated by the Usage
 - taxRate - Tax rate of the rated amount
 - currencyCode - Currency of the rated amount
 - productRef - href of the product in the inventory that generated the rate

For a usage document to be processed and understood by the system, some fields are required defined as characteristics of Usage Specification. These are:

- orderid - Id of the order where the product was acquired
- productid - Id in the inventory of the product containing the details of the acquisition
- correlationNumber - Sequence number of the usage documents, used to ensure that no usage has been lost
- unit - Unit being monitored while accounting the service (e.g second, call, megabyte, etc)
- value - Usage made of the service of the given unit

The resources managed by Usage Management API are as follows:

- usage
- usageSpecification

The operations supported by the Usage Management API are specified using the Swagger tool and shown in figure 49.

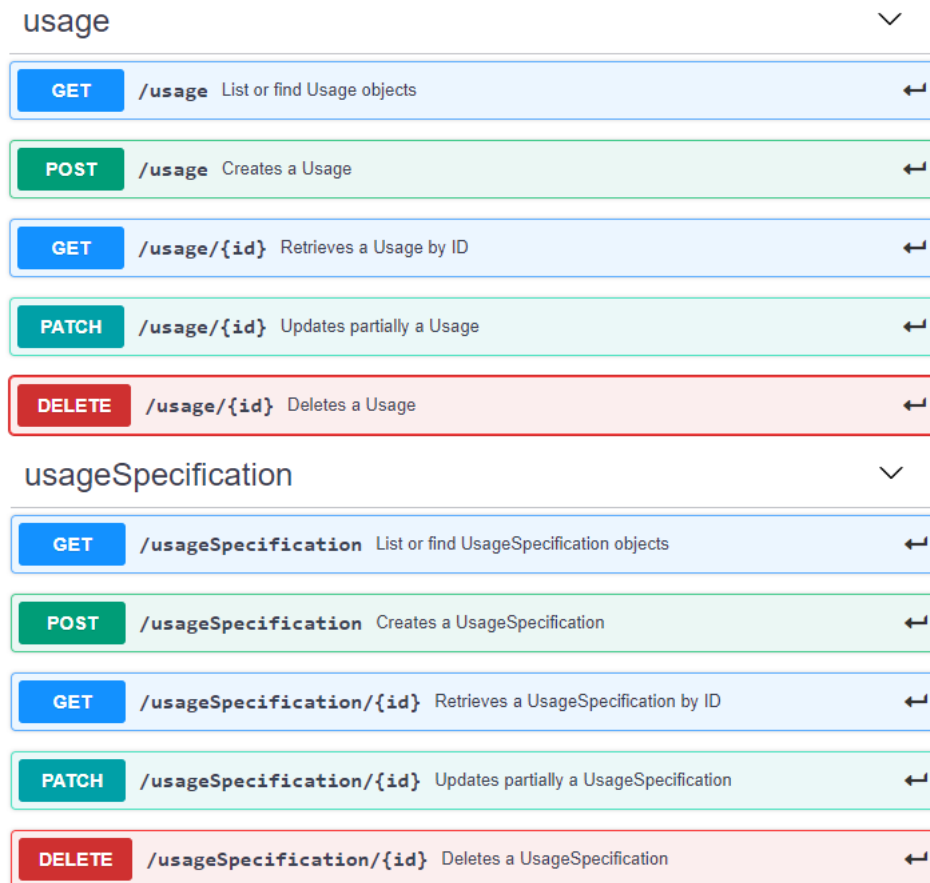


Figure 49: Usage Management Api Swagger operations

The following example shows how Usage details for the usageType of ‘Voice’ can be retrieved.

Request
GET /tmf-api/usageManagement/v4/usage?fields=id,href,status,usageType,usageDate&usageType=Voice Accept: application/json
Response
200 [{ "id": "93c2-683ea281566c", "href": "https://api.csp.com/tmf-api/usageManagement/v4/Usage/93c2-683ea281566c", "status": "received", "usageType": "Voice", "usageDate": "2020-11-20T08:13:16.000Z" }, { "id": "bbba-0c6e63171b78", "href": "https://api.csp.com/tmf-api/usageManagement/v4/Usage/bbba-0c6e63171b78", "status": "rated", "usageType": "Voice", "usageDate": "2020-09-01T08:30:01.000Z" }]

Figure 50: Usage retrieval example

4.5 Communication API

The Communication API provides a capability to create and send communications, notifications and instructions to Parties, Individuals, Organizations or Users.

It provides a standardized mechanism for Communication management such as creation, update, retrieval, deletion and notification of the system communication events. The following data resources are managed:

- Communication message: notification approach in the format of a message which can be dispatched to certain users by the system with the content which can be felt and understood by the recipient. The user can be either a final customer of a customer service agent and it can be done through different interaction channels such as email, short message or mobile app notification.

Communication API performs the following operation on the resource of “Communication Message”. There are two types of operations provided in this API. One is the management of the request message body. Another is for sending the communication message to the customer.

Operations for Communication Message body management

- Retrieval of an existing Communication Message depending on filter criteria
- Creation of a new Communication Message
- Partial update of an existing Communication Message
- Deletion of an existing Communication Message
- Notification of events:

- Creation of Communication Message
- Updating Communication Message
- Deletion of Communication Message
- Operations for sending Communication Message.
- Send a message, including:
 - Send a new message with the whole communication message body (POST operation)
 - Send a message with the predefined communication message body (POST operation)

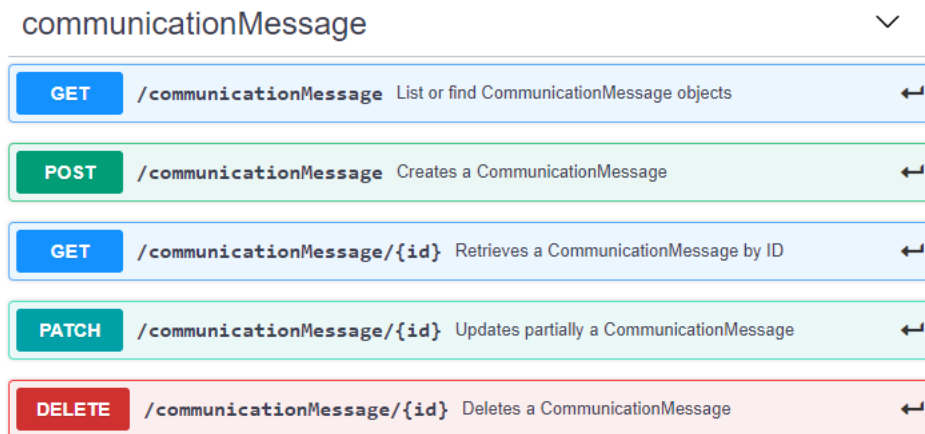


Figure 51: Communication API Swagger operations

The Communication Message is used to express the message itself. Once the communication message exists, this API can be used to send it from the sender to the receiver through `POST /communicationMessage/send`

To send a pre-defined Communication message from the sender to the receiver the following command is used: `POST /communicationMessage/{ID}/send`

Figure 52 shows an example of a query to retrieve a pre-defined message.

REQUEST
GET /communicationMessage/11006 Content-type: application/json Accept: application/json
RESPONSE
200 Content-Type: application/json <i>Following a whole representation of the Communication Message resource with all its attributes.</i> <i>Refer to Communication Message Resource.</i>

Figure 52: Pre-defined message query example

4.6 Customer Management API

The Customer Management API deals with customer information. It is used for saving customer private information that cannot be included within the party resources.

A customer can be a person, an organization that buys products and services or receives free offers or services from another service provider. The Customer Management API allows management of identification and financial information.

This API manages the following fields:

- `id` - Id that identifies the customer object
- `href` - URL pointing to the customer info
- `name` - Username of the owner of the customer object. Note that this field maps the `id` field of the individual object
- `relatedParty` - Party which owns the Customer object
- `contactMedium` - List of contact mediums that define a shipping contact. This list has the same format as the contact medium described in the Party Management API, and must include an email, a telephone, and an address. In this case, this address is private and only visible by sellers when they need it
- `customerAccount` - Reference of the customer account attached to this customer object

This API assumes that the information regarding customer accounts and payment means is obtained by accessing the Account Management API, while the information regarding related parties is obtained by accessing the Party Management API.

customer	
GET	<code>/customer</code> List or find Customer objects
POST	<code>/customer</code> Creates a Customer
GET	<code>/customer/{id}</code> Retrieves a Customer by ID
PATCH	<code>/customer/{id}</code> Updates partially a Customer
DELETE	<code>/customer/{id}</code> Deletes a Customer

Figure 53: Customer Management API Swagger operations

The following example shows a request for creating a Customer resource.

Request
<pre>POST /tmf-api/customerManagement/v4/customer Content-Type: application/json { "name": "Moon Football Club", "relatedParty": [{ "@REFERREDType": "Organization", "id": "500", "name": "Moon Football Club " }] }</pre>
Response
<pre>201 { "@type": "Customer", "href": "https://host:port/tmf-api/customerManagement/v4/customer/1140", "id": "1140", "name": "Moon Football Club", "relatedParty": [{ "@REFERREDType": "Organization", "href": "https://host:port/tmf-api/partyManagement/v4/organization/500", "id": "500", "name": "Moon Football Club " }] }</pre>

Figure 54: Customer resource creation example

4.7 Customer Bill Management API

The Customer Bill Management API allows to find and retrieve one or several customer bills (invoices) produced for a customer. A customer bill is an electronic or paper document produced at the end of the billing process. The customer bill gathers and displays different items (applied customer billing rates generated during the rating and billing processes) to be charged to a customer. It represents a total amount due for all the products during the billing period and all significant information such as dates, bill reference, etc.

This API also provides operations to find and retrieve the details of applied customer billing rates presented on a customer bill.

Finally, this API allows to request in real-time a customer bill creation and to manage this request.

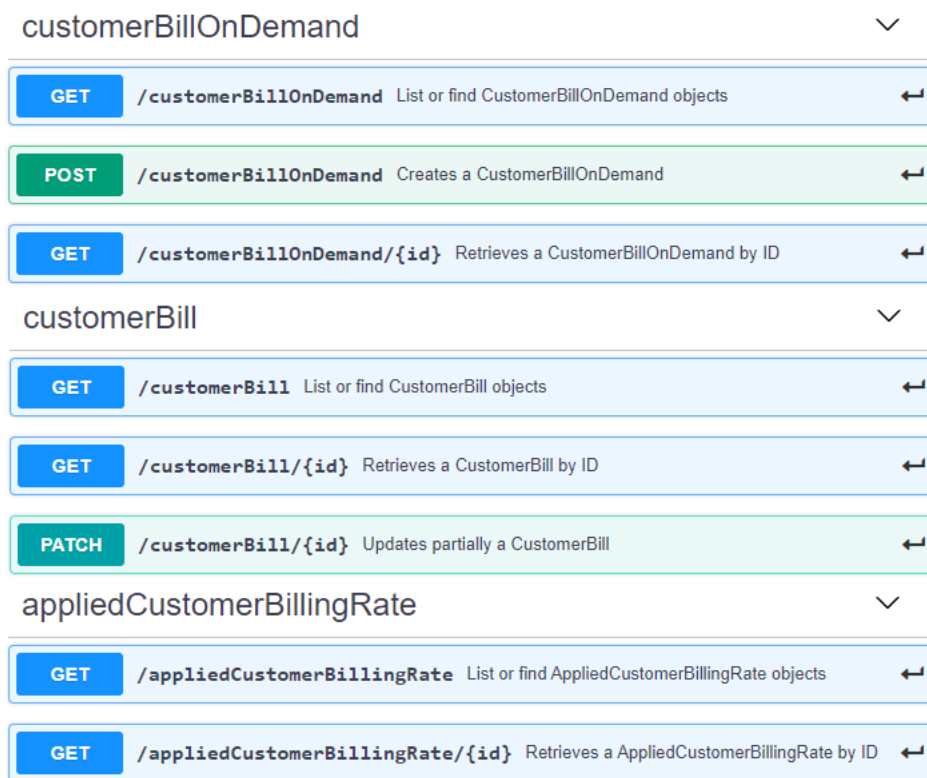
The resources for this API include:

- Customer bill resource
- The billing account receives all charges (recurring, one time and usage) and periodically a customer bill /invoice is produced, concerning different related parties. A payment method could be assigned to the customer bill to build the call of payment, with a tax item created for each tax rate. The customer bill is linked to one or more documents that can be downloaded via a provided URL.
- Applied customer billing rate resources
- A customer bill displays applied billing rates created before or during the billing process.

- Customer bill on demand resource manages the creation request of a customer bill in real-time

Customer Bill Management API performs the following operations (figure 55):

- Retrieve a customer bill or a collection of customer bills depending on filter criteria.
- Partial update of a customer bill (for administration purposes).
- Retrieve an applied customer billing rate or a collection of applied customer billing rates depending on filter criteria.
- Create a customer bill on demand request
- Retrieve one or a collection of customer bills on demand request(s) depending on filter criteria.
- Manage notification of events on customer bill and customer bill on demand.



The image displays a Swagger API interface for the Customer Billing Management API. It is organized into three sections: customerBillOnDemand, customerBill, and appliedCustomerBillingRate. Each section lists the available HTTP methods, their endpoints, and a brief description of the operation.

Method	Endpoint	Description
GET	/customerBillOnDemand	List or find CustomerBillOnDemand objects
POST	/customerBillOnDemand	Creates a CustomerBillOnDemand
GET	/customerBillOnDemand/{id}	Retrieves a CustomerBillOnDemand by ID
GET	/customerBill	List or find CustomerBill objects
GET	/customerBill/{id}	Retrieves a CustomerBill by ID
PATCH	/customerBill/{id}	Updates partially a CustomerBill
GET	/appliedCustomerBillingRate	List or find AppliedCustomerBillingRate objects
GET	/appliedCustomerBillingRate/{id}	Retrieves a AppliedCustomerBillingRate by ID

Figure 55: Customer Billing Management API Swagger operations

Figure 56 shows an example of a request for retrieving a single BillCycle.

Request
GET /tmf-api/customerBillManagement/v4/customerBillOnDemand/D123?fields=id,href,billingAccount Accept: application/json
Response
200 <pre>{ "id": "D123", "href": "http://server:port/tmf-api/Customer_Bill_Management/v4/customerBillOnDemand/D123", "billingAccount": { "id": "A0815", "href": "http://server:port/tmf-api/Account_Management/v4/billingAccount/A0815", "name": "BA Peter Retep", "@referredType": "BillingAccount", "@type": "BillingAccountRef", "@baseType": "BillingAccountRef", "@schemaLocation": "...some href..." }, "@type": "CustomerBillOnDemand", "@baseType": "CustomerBillOnDemand", "@schemaLocation": "...some href..." }</pre>

Figure 56: Single BillCycle retrieval example

5 Analytics toolbox APIs

The PLATOON Data Analytics toolbox will be formed of all the data analytics tools that will be developed and used in the project by the different partners for the different use cases. These tools will allow the extraction of value from heterogenous data sources. There will be two main groups of data analytics tools:

1. **Energy specific tools** for which have been specifically developed for the different applications or services (benchmarking, predictive maintenance, operation optimisation, etc.) and for the different domains of the energy value chain as per the different use cases defined in deliverable D1.1 “Challenges/ Business case definition” (i.e., RES generation, smart grids and End Use of Energy).
2. **Generic tools** that complement the energy specific tools and that are applicable to different applications and domains (e.g. data pre-processing tools, visualisation tools, graph processing tools, etc.).

The PLATOON Data Analytics Toolbox must meet the following principles: Interoperability, Usability, Efficiency (Flexibility) and Reusability (transferability).

In order to meet the interoperability principle, the toolbox must be integrated with the PLATOON reference architecture and common data models and APIs must be defined. Regardless of the implementation type (as an external microservice or in the customer infrastructure) the communication between data analytics tools in PLATOON and the rest of the components will be provided with a REST API endpoint defined using the OpenAPI 3.0 standard.

The **meta information** will be defined by each Analytic Tool and it will follow the format defined by OpenAPI 3.0 specification.

The **path items (end points) and the operations provided** are the main focus of this section. In order to provide a harmonized way to access the Analytic Tools a common set of endpoints and reusable components must be defined. However, each pilot could extend or adapt them.

A study of the art analysis has been performed and two different machine learning and data analytics APIs have been identified.

- **Azure predictive maintenance OpenAPI:** A very simple API with one endpoint to get the result of the analytic model.
- **DEEPaaS API:** DEEP as a Service (DEEPaaS) is a REST API that is focused on providing access to machine learning models. It has been developed by the DEEP Hybrid DataCloud European Project.⁴⁴

Next sections provide information about both approaches and how they will be used in PLATOON.

The reusable components will be defined by each analytic tool and the data models could be specific or reuse the semantic models defined in PLATOON.

5.1 Azure Machine Learning example

Deploying an Azure Machine Learning model as a web service creates a REST API endpoint. You can send data to this endpoint and receive the prediction returned by the model.

In Azure the general workflow for creating a client that uses a machine learning web service is:

1. Get the connection information.
2. Determine the type of request data used by the model.
3. Create an application that calls the web service.

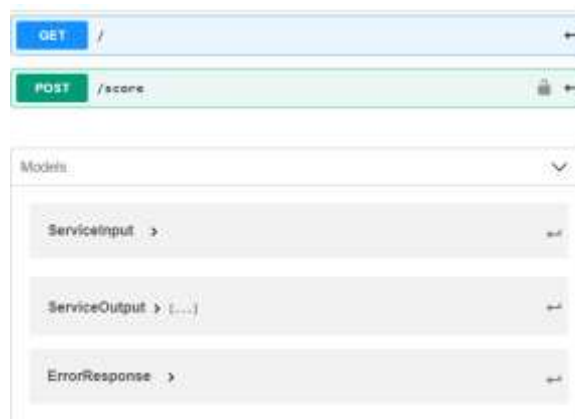
Azure does not define a common OpenAPI for accessing the models but allows each model to define and expose a specific one. However, a very simple example is provided with the minimum functionality.

The following endpoints and operations are defined:

GET / → Simple health check endpoint to ensure the service is up at any given point

POST /score → Run web service's model and get the prediction output

Along with the operations, three data models are defined: ServiceInput, ServiceOutput and ErrorResponse.



Next, the OpenAPI 2.0 definition in Jason format is presented:

```
{
  "swagger": "2.0",
  "info": {
    "title": "myservice",
    "description": "API specification for Azure Machine Learning myservice",
    "version": "1.0"
  },
  "schemes": [
    "https"
  ],
  "consumes": [
    "application/json"
  ],
  "produces": [
    "application/json"
  ],
  "securityDefinitions": {
    "Bearer": {
      "type": "apiKey",
      "name": "Authorization",
      "in": "header",
      "description": "For example: Bearer abc123"
    }
  },
  "paths": {
    "/": {
      "get": {
        "operationId": "ServiceHealthCheck",
```

```

        "description": "Simple health check endpoint to ensure the service is up at
any given point.",
        "responses": {
            "200": {
                "description": "If service is up and running, this response will be
returned with the content 'Healthy'",
                "schema": {
                    "type": "string"
                },
                "examples": {
                    "application/json": "Healthy"
                }
            },
            "default": {
                "description": "The service failed to execute due to an error.",
                "schema": {
                    "$ref": "#/definitions/ErrorResponse"
                }
            }
        }
    },
    "/score": {
        "post": {
            "operationId": "RunMLService",
            "description": "Run web service's model and get the prediction output",
            "security": [
                {
                    "Bearer": []
                }
            ],
            "parameters": [
                {
                    "name": "serviceInputPayload",
                    "in": "body",
                    "description": "The input payload for executing the real-time
machine learning service.",
                    "schema": {
                        "$ref": "#/definitions/ServiceInput"
                    }
                }
            ],
            "responses": {
                "200": {
                    "description": "The service processed the input correctly and
provided a result prediction, if applicable.",
                    "schema": {
                        "$ref": "#/definitions/ServiceOutput"
                    }
                },
                "default": {
                    "description": "The service failed to execute due to an error.",
                    "schema": {
                        "$ref": "#/definitions/ErrorResponse"
                    }
                }
            }
        }
    }
},
"definitions": {
    "ServiceInput": {
        "type": "object",
        "properties": {
            "data": {
                "type": "array",

```

```

        "items": {
            "type": "array",
            "items": {
                "type": "integer",
                "format": "int64"
            }
        }
    },
    "example": {
        "data": [
            [ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 ]
        ]
    },
    "ServiceOutput": {
        "type": "array",
        "items": {
            "type": "number",
            "format": "double"
        },
        "example": [
            3726.995
        ]
    },
    "ErrorResponse": {
        "type": "object",
        "properties": {
            "status_code": {
                "type": "integer",
                "format": "int32"
            },
            "message": {
                "type": "string"
            }
        }
    }
}
}
}

```

5.2 DEEP Hybrid DataCloud project: DEEPaaS API

The key concept proposed in the DEEP Hybrid DataCloud project is the need to support intensive computing techniques that require specialized HPC hardware, like GPUs or low-latency interconnects, to explore very large datasets. A Hybrid Cloud approach enables access to such resources that are not easily reachable by the researchers at the scale needed in the current EU e-infrastructure.

One of the objectives of the project is to define a “DEEP as a Service” solution to offer an easy integration path to the developers of final applications. “DEEP as a Service” includes a set of building blocks that enable the easy development of applications requiring these techniques: deep learning using neural networks, parallel post-processing of very large data, and analysis of massive online data streams

DEEPaaS API is the key component for making the modules accessible to everybody (including non-experts), as it provides a consistent and easy to use way to access the model’s functionality. It enables data scientists to expose their applications through an HTTP endpoint,

delivering a common interface for machine learning, deep learning and artificial intelligence applications.

The DEEPaaS API is available for both inference and training. DEEP platform provides novelties such as asynchronous training and control to launch, monitor, stop and delete the training directly from the web browser in a transparent way through the DEEPaaS API.

The functional improvements are detailed as follows:

1. Multiple asynchronous trainings (i.e. deployment of multiple instances or retraining a model with available data)
2. Two-ways of state monitoring and report: 1) a list of trainings (running or completed) and 2) status of running training (based on UUID)
3. Possibility to cancel running training
4. Various features such as API version information, debug information, passing parameters for prediction, etc.

Through the DEEP as a Service API, endpoint users get access to the DEEPaaS API, via a swagger user interface (GUI) allowing them to interact with their models:

1. check the model metadata and details
2. retrain a certain model with their own data
3. get the list of trainings currently running
4. get the status of a training
5. cancel the training
6. make a prediction with a certain trained model



The image shows a Swagger UI interface for the DEEPaaS API. It is titled 'models' and lists seven endpoints with their respective HTTP methods and descriptions:

Method	Endpoint	Description
GET	<code>/v2/models/</code>	Return loaded models and its information
GET	<code>/v2/models/imgclas/</code>	Return model's metadata
POST	<code>/v2/models/imgclas/train/</code>	Retrain model with available data
GET	<code>/v2/models/imgclas/train/</code>	Get a list of trainings (running or completed)
GET	<code>/v2/models/imgclas/train/{uuid}</code>	Get status of a training
DELETE	<code>/v2/models/imgclas/train/{uuid}</code>	Cancel a running training
POST	<code>/v2/models/imgclas/predict/</code>	Make a prediction given the input data

Figure 57: DEEP as a Service API endpoint

5.3 PLATOON Analytic toolbox OpenAPI definitions

Next, PLATOON approach for the definition of the 3 sections of the OpenAPI APIs for the Analytic toolbox services is defined.

The meta information will be defined by each Analytic Tool and it will follow the format defined by OpenAPI 3.0 specification and it will include: openAPI version, Title, version and description of the API, Authentication (Basic Auth, OAuth2, etc), API meta info (contact, license, usage conditions, etc) Meta data example: info, servers and authentication:

```

openapi: 3.0.0
info:
  version: 1.0.0
  title: PLATOON Predictive Maintenance API
  description: A simple API to illustrate OpenAPI concepts

servers:
  - url: https://platoon.pm.io/v1

# Basic authentication
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
security:
  - BasicAuth: []

```

The **path items (end points) and the operations provided** will follow the DEEPaaS API specification. However, not all the endpoints of the DEEPaaS API have to be implemented by all the tools. The minimum endpoints are:



As an example, a PLATOON endpoint could be: /v2/models/SmartGridLoadEnergyForecaster/predict.

Some potential synergies could be to co-develop a tool amongst similar pilots, validate same tool in different pilots or validate different tools in same pilots. The similar pilots could share the same OpenAPI url.

The result of this analysis is summarised in the following table:

	Pilot						
	1a	2a	2b	3a	3b	3c	4a
Preprocessing tools							
Health_Tracker_Dashboard_Preparation	x						
SCADA_Data_Cleaner	x						
Orchestration tools							
Health_Tracker.	x						
Benchmarking Tools							
Smart Grid Global losses assessment			x				
Prosumer segmentation			x				
Smart Building Energy Consumption Benchmarker					x		
Consumptions and Occupancy Profile Correlator					x		
Forecasting Tools							
Smart Grid Load Energy Forecaster		x					
Wind Energy Production Forecaster		x					
Solar (PV) Energy Production Forecaster						x	x
RES-Effects-Calculator (PV plants)		x					
Electrical Transformer - Top oil temperature model (virtual sensor)			x				
Smart Grid Technical losses model			x				
NTL identification			x				
Building Occupancy prediction				x			
HVAC load Prediction				x	x	x	x
HVAC load Simulation				x		x	
Prediction of potential load interruption				x			
Preheating and precooling time estimation				x		x	
Buildings Lighting Evaluator					x		
Predictive maintenance Tools							
Digital Twin - Wind Turbine - Electric Generator and Power Converter	x						
Anomaly and Failure Detection/Diagnosis, Root Cause Diagnosis	x	x	x		x	x	
Electrical Transformer - RUL Estimation			x				
Electrical Transformer - Oil Analysis			x				
Health Index calculation			x			x	
Health monitoring			x			x	
Maintenance activities impact assesment			x			x	
Vibration Analysis Tool						x	
HVAC components - RUL Estimation						x	
Optimisation tools							
Asset operation and maintenance planning optimisation			x			x	
Smart Grid topology identification			x				
HVAC and RES operation optimisation						x	x
Others							
Supervision of load interruption				?			
Supervision of load shifting				?			

Figure 58: Cross pilot synergies regarding data analytics tools

The **reusable components** that define the common data models will be defined by each analytic tool and they could be specific or reuse the semantic models defined in PLATOON.

Apart from using a common API specification which is the means to send/receive information, in order to guarantee the full interoperability all the developed data analytics tools should take into account the common data models defined in task T2.3 “Data models”.

These common data models define the common language to understand the received data and to send the data to the rest of the components of the architecture according to a standard data model that they can understand.

However, this does not mean that all the components/tools must use internally the common data models defined in task T2.3 “Data models”. In fact, different components/tools internally might

use some internal data models different to the ones defined in T2.3 “Data models”. Furthermore, the common data models defined in task T2.3 will be semantic data models. Nevertheless, not all of the data analytics tools will actually do analytics on semantic data. The proposed solution for dealing with the PLATOON data models in Data Analytics Tools is explained in more detail in task T4.1 “PLATOON analytical toolbox design”.

6 Reference architecture

In task T2.1, the Platoon Ref architecture has been defined (see figure 59). All the details of the specific components can be found in deliverable D2.1.

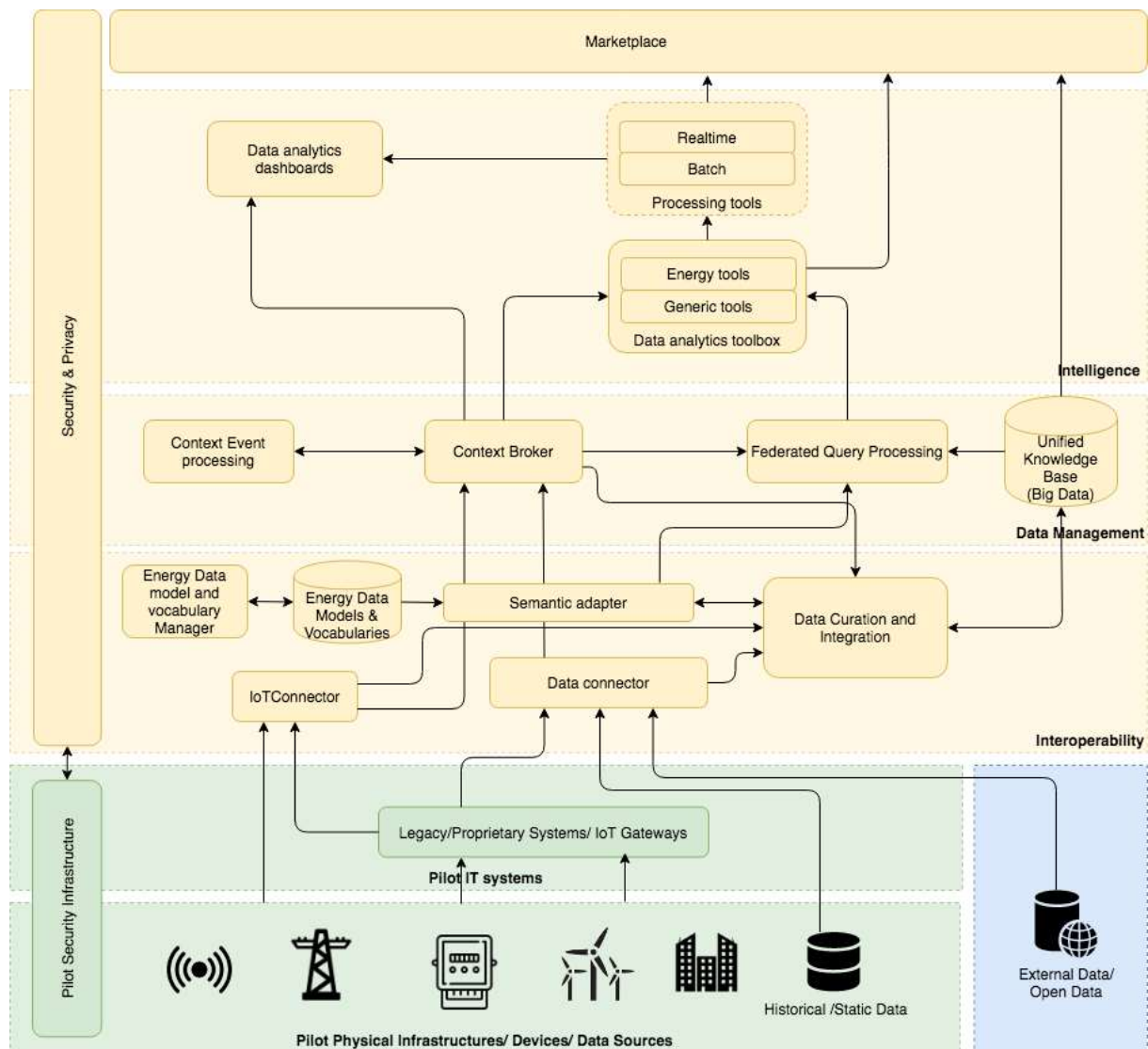


Figure 59: PLATOON reference architecture logical view

The main insight driving the current task is to create solutions capable of controlling, in a secure and efficient manner, the data distribution from the primary sources to the business tools used to take advantage of them. That involves the transition of the data through several components and, thereby, the need for transforming these data at each step to be understood by their receptors.

The concept of interoperability involves the capability of sharing information in systems where several components work together, by using standard notations and converting the data, in execution time, to the formats readable for the consuming components.

This process involves the data acquisition, management and analysis that take place along several layers as follows (please refer to the deliverable D2.1 “Platoon reference architecture”):

1. Raw data is acquired from the pilot security infrastructure: at the pilot physical infrastructures, devices, data sources and from external data/open data sources in several formats.
2. The IoT Connector captures the data coming from physical devices such as sensors, devices or actuators, while the Data Connector those coming from logical software such as complete or legacy systems. The corresponding IoT agent will transform the native IoT protocol to JSON-LD.
3. Data goes through the semantic adapter in order to convert non-semantic data into semantic data and to the context broker, the federated query and the Unified Knowledge Base if no further processing is needed (maps with common data models and can be used to update data).
4. NGS-LD REST API creates the entities from the data packages, according to the data model schema. Entities are encoded with JSON-LD notation.
5. Formatting data as NGS-LD entities involves their step toward the data management layer, entering the Context Broker where they will provide the information from the context makers.
6. Data are addressed to the business intelligence layer, where they are subjected to the analytical processes (see section **Fehler! Verweisquelle konnte nicht gefunden werden.** for Data Analytics Toolbox APIs) making them useful for the target market (see section **Fehler! Verweisquelle konnte nicht gefunden werden.** for Marketplace APIs).

6.1 Context Data Broker

The Context Data Broker is one of the main components from the PLATOON architecture that will use the NGS-LD API.

To understand the meaning of the Context Data Broker (or just Context Broker) and its role within PLATOON, first it is interesting to know the concepts of Data Space and Context. Therefore, the following paragraphs will cover the main insights to explain these key concepts.

Complex processes taking place in the energy sector usually involve high amounts of data traffic among different operating devices, some of them being the source of information and others the ones consuming data. To ease the proper distribution of huge amounts of data being acquired in an efficient, standardized and safe way, modern industry will in future use data spaces. Said data spaces enable companies and organizations to hold a strict control on their data by defining the corresponding methods and connectors for an efficient data traffic.

Within the data space, a connector is the component dealing with the data transfer from both, the data sources at the physical layer to the data space, and from the data space to the context consumers. Besides this, the connector controls the access to the data space and guarantees the security of the data during these transitions.

For said purposes, Data Security, Privacy and Security framework have to be created with the following features:

- Access control, restricted to those users properly identified with their credentials.
- Profiling of the users determining the owning of the data, their management or query and any other useful roles with different permissions.

- Data availability along all the components within the process, which is achieved by creating safe connections among the data owners and the subscribed data consumers.
- Interoperability, since the use of a standardized data model enables the capability of the connectors to link data from different sources within a same data space.
- Wide diversity of data sources, which can arise from different companies working on connected processes, physical sensors, data histories or open datasets.
- Compatibility with other software to extend their applicability and business chances.

WP3 is responsible for creating such Data Security, Privacy and Security framework based on IDS (International Data Space) reference architecture.

Moreover, the term Context refers to a digital description of the physical environment that is built from the extraction of a collection of key reference magnitudes, such as weather parameters, energy consumption in clients, etc. All this information is stored in data entities named as Context Elements, data to be shared in the data space and in PLATOON represented in NGSI-LD format. Magnitudes acquired from sensors in the physical layer are serialized on site by a device firmware and sent through an IoT connector to be written at the corresponding context elements.

Context is also supported by static information from the companies involved, either from their data history or from open repositories belonging to third parties like meteorological services releasing weather parameters or forecasting. The set of devices responsible for the measurement, serialization and transmission of magnitudes, together with the other datasets providing context information, are known as Context Makers, also known as Data Providers/Owners in IDS terminology.

There are also applications known as Context Consumers, also known as Data Consumers/User in IDS terminology, that take advantage of the context data by consuming information from the context elements. Target market utilities also consume this information. In PLATOON the main components holding the role of context/data consumers are the Data Analytics Tool providers and the Market Place is the one-stop shop that puts together Data Providers/Owners and Data Consumers/Users.

The IDS Broker can be described as an information sharing tool aimed to send and acquire data from different organizations, control centers and devices in a safe and structured way. Context Broker is a useful communication channel to carry out projects where great amounts of data arising from different sources have to be managed.

At the technological level, the Context Broker is the component responsible for managing, updating and performing queries to the information stored at the context elements. Data arising from the context makers and stored in context elements are published by the Context Broker, becoming available for the context consumers upon agreement of the privacy policy. Thus, Context Broker works as a server of NGSI-LD formatted entities, which are the only type of data units that can be shared by this.

In PLATOON, the Context Data Broker enables the discovery, gathering and publishing of near real time context information through Context Management APIs. Context Broker, through its interface, makes available the context information regardless data source and using different types of interactions: query and subscription, represents the synchronous and asynchronous interactions with context data source.

Synchronous interactions are performed using a query mechanism to obtain context information; the component allows building queries, using different types of filters, in order to

retrieve information with high levels of precision. The asynchronous interaction is performed by publish-subscribe mechanism: a notification is generated when published data meets the subscription conditions; this feature is really useful to avoid the implementation of a polling process on data sources of interest, allowing to be notified when the context information changes.

6.2 IoT Connector

The IoT Connector is in charge of transmitting the raw data coming from the device to the virtual entity representation at the Data Management Layer; sending commands or actions request to an actuator device; mapping of device and its features to a virtual entity with related attributes and metadata. IoT Connectors should also be able to deal with security aspects (authentication and authorization of the channel) and data sovereignty (IDS) aspects. In addition, they should provide other common services to the device programmer. The following figure indicates the role and position of IoT Connector in PLATOON architecture.

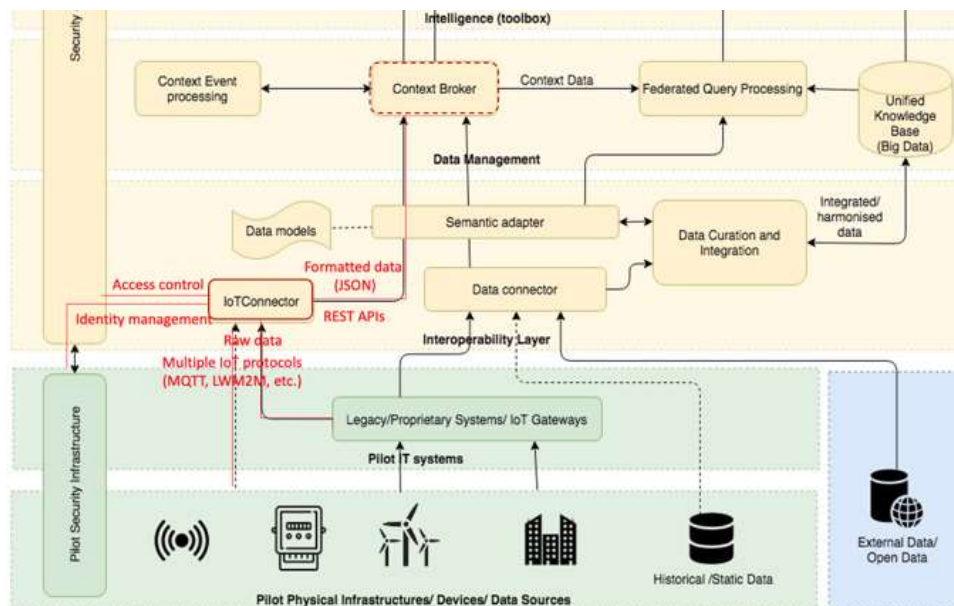


Figure 60: IoT connector in PLATOON architecture

FIWARE IoT Agent⁴⁵ is an example of a reference implementation of IoT Connector. The IoT Agents work as protocol translation gateways, used to fill the gap between traffic sent and received on the South Port (typically lightweight protocols aimed to constrained devices) and traffic sent and received on the North Port which uses the standard NGSI protocol. Multiple IoT Agents are provided such as IoT Agent for JSON for a bridge between HTTP/MQTT messaging (with a JSON payload) and NGSI, IoT Agent for LWM2M – a bridge between the Lightweight M2M protocol and NGSI, IoT Agent for LoRaWAN – a bridge between the LoRaWAN protocol and NGSI, etc.

Each individual IoT Agent offers is driven by a config.js configuration file contains explicit custom settings based on the protocol and payload the IoT Agent is translating. It will also contain some common flags for common functionality provided by the IoT Agent node link (e.g., for connecting to a context broker or for authentication). The IoT Agent node library⁴⁶ offers a standard API for provisioning devices and ensures that each IoT Agent can configure

its device communications using a common vocabulary regardless of the payload, syntax or transport protocol used by the device itself. A series of additional plugins are offered where necessary to allow for expression parsing, attribute aliasing and the processing of timestamp metadata.

Standardized OAuth2-based security is available to enable each IoT Agent to connect to several common Identity Managers (e.g., Keystone and Keyrock) so that communications can be restricted to trusted components.

6.2.1 IoT Connector APIs

The interfaces of IoT Connector are indicated in D2.1 “Platoon reference architecture”, as follows:

Interfacing Component	Interface Description
IoT devices, gateways	These data sources will send data to the IoT Connector.
Security & Privacy	The IoT connector interacts with this module to provide security and sovereignty over the data.
Semantic Adapter	This component is used to convert the data into common data models.
Context Broker	This component is used by the IoT Connector to update data, if data already maps with the common data models

IoT Connector provides the following functions to the interfacing components:

- Offering a standard location to listen to device updates
- Offering a standard location to listen to context data updates
- Holding a list of devices and mapping context data attributes to device syntax
- Security Authorization

IoT Connector works based on REST API principles and the communication of the interfaces is performed by ‘receive data’, ‘receive command’ and ‘update context’ using the following HTTPS requests:

- POST: Creates a resource or list of resources
- PUT: Updates a resource
- GET: Retrieves a resource or list of resources
- DELETE: Delete a resource

In order to send a command to a device, the IoT Agent sends a HTTP POST request to the endpoint supplied by the device. IoT Connector handles the HTTP requests generated from the Context Broker towards an IoT device (Southbound traffic). Southbound traffic consists of ‘commands’ made to actuator devices which alter the state of the real world by their actions. For example, a command ‘ON’ would switch on the lamp in real life by changing the state of a lamp. The body of the POST request holds the command.

A device can report new measures to an IoT Connector using an HTTP GET request. The HTTP requests generated from an IoT device via an IoT Connector towards the Context Broker are known as northbound traffic. Northbound traffic consists of ‘measurements value’ made by IoT devices to relay the state of the real world into the context data of the system. For example, a measurement from a humidity sensor could be relayed back into the context broker to indicate that the moisture level of the entity has changed. A subscription could be made to be informed of such changes and there provoke further actions. The HTTP GET request composes along with the following query parameters:

- *i* (device ID): Device ID (unique for the API Key).
- *k* (API Key): API Key for the service the device is registered on (OAuth2 based security).
- *t* (timestamp): Timestamp of the measure. Will override the automatic IoT Connector timestamp (optional).
- *d* (Data): I.

The *i* and *k* parameters are mandatory. IoT Connector payload

HTTP POST can also be used. This case, *d* (Data) is not necessary - the key-value pairs of the measurement are passed as the body of the request. *i* and *k* query parameters are still mandatory:

- *i* (device ID): Device ID (unique for the API Key).
- *k* (API Key): API Key for the service the device is registered on.
- *t* (timestamp): Timestamp of the measure. Will override the automatic IoT Connector timestamp (optional).

The *i* and *k* parameters are mandatory.

6.3 Data Connector

Data connector is a component which is important and necessary to allow the integration of various field devices (e.g. IED, SMs, PMUs, sensors, etc.). Data connectors might be part of the edge or central PLATOON architecture and in both cases it is responsible to correctly query the legacy and property system or to expose interfaces suitable for receiving information from them in the push/pull manner.

Data Connectors should also be able to deal with security aspects (authentication and authorization of the channel). and data sovereignty (IDS) aspects interacting with the Security & Privacy module of PLATOON’s reference architecture. This component should be able to access to legacy/proprietary data using several approaches (e.g. read from API, read csv or json file, read from SQL or NoSQL databases, etc.), furthermore this component must interact with the Semantic Adapter to convert non-semantic data into semantic data, the Data Curation and Integration module to harmonize and/or integrate the data with additional data and/or metadata or with the Context Broker if no further data processing is needed.

To achieve vertical operability the data connector will address the IEC 61850 standard, whose details are included in the annex (section 8) of this document.

7 Conclusions

The first version of this document was prepared and developed in parallel to T2.1 “Platoon Reference Architecture” and to the design and development of the marketplace component in T3.4 “Marketplace Design and Setup” (M12), so the information contained was not definite. The idea was to complement or change it in the second version planned for M27, depending on the design/development of the components within the PLATOON architecture. Any updates/modifications were to be included in the present version of this deliverable.

The partners who participated in the preparation of the original document were consulted in order to confirm changes or updates achieved. At the present time, there were not many changes reported and the defined schemas are being used and being implemented in the pilots. Therefore in conclusion no reportable updates can be reported in this update deliverable. In the first version, there was no mention made regarding the energy domain, OGEMA (Open Gateway Energy Management), so a section has been added on this framework in section 8.2.

The aim of the deliverable is to define a set of APIs, to enable the interoperability of the different components and modules of the logical architecture of Platoon Platform and also, the effective and easy communication of data with the rest of platforms and systems that we can find in an ecosystem that encompasses the generation, distribution, consumption and value-added services of energy.

We demonstrate that this set of APIs is compatible with the existing different data models used by the different pilot projects, being proprietary or standardized models, and compatible with the different components of the Platoon Platform.

In the deliverable, we introduce the main concepts and the specific terminology behind the design of APIs, and show the importance of interoperability following FAIR principles.

In the first set of APIs defined, by using NGS-LD we found major advantages regarding “context” information exchange:

1. Using NGS-LD APIs, applications can flexibly discover and query relevant information. The data discovery is very agile, and the query capabilities support the most common questions that are used in information systems.
2. NGS-LD APIs precisely communicate the nature of the context information for a given service, such as its period of validity, its geographic constraints, and other semantically important information.
3. To ensure interoperability, the NGS-LD APIs through its information model, defines the meaning of the most commonly needed terms and uses the Platoon domain-specific extensions to model any other type of information.

With the second set of APIs defined, we demonstrate that by using TM Forum standard REST based APIs, we are able to attend the requirements of the Data Marketplace layer components with a flexible integration among operations, management and billing systems.

Finally, the Data Analytics Toolbox dedicated APIs are the key element that will enable the use (and reuse) of the tools from the PLATOON Data Analytics Toolbox by the partners of the project in different pilots.

8 Annex 1

8.1 IEC 61850

IEC 61850 is an international standard that addresses communication protocols for intelligent electronic devices (IED) at electrical substations. The general overview of the standard and its part relations is shown in Figure 61. The IEC 61850 standard was derived from numerous incompatible standards to unify communication among different devices within substation. Standard provides a design guideline for automation system, defining requirements for data transmission, how to describe devices and how to exchange information among device at runtime and at configuration time.

In the PLATOON the following parts are relevant:

- IEC 61850-9-2 Sampled values (SV) over ISO/IEC 8802-3-2
- IEC 61850-7-2 Generic Object Oriented Substation Events (GOOSE)
- IEC 61850-7-2, IEC 61850-8-1 The Manufacturing Message Specification (MMS)

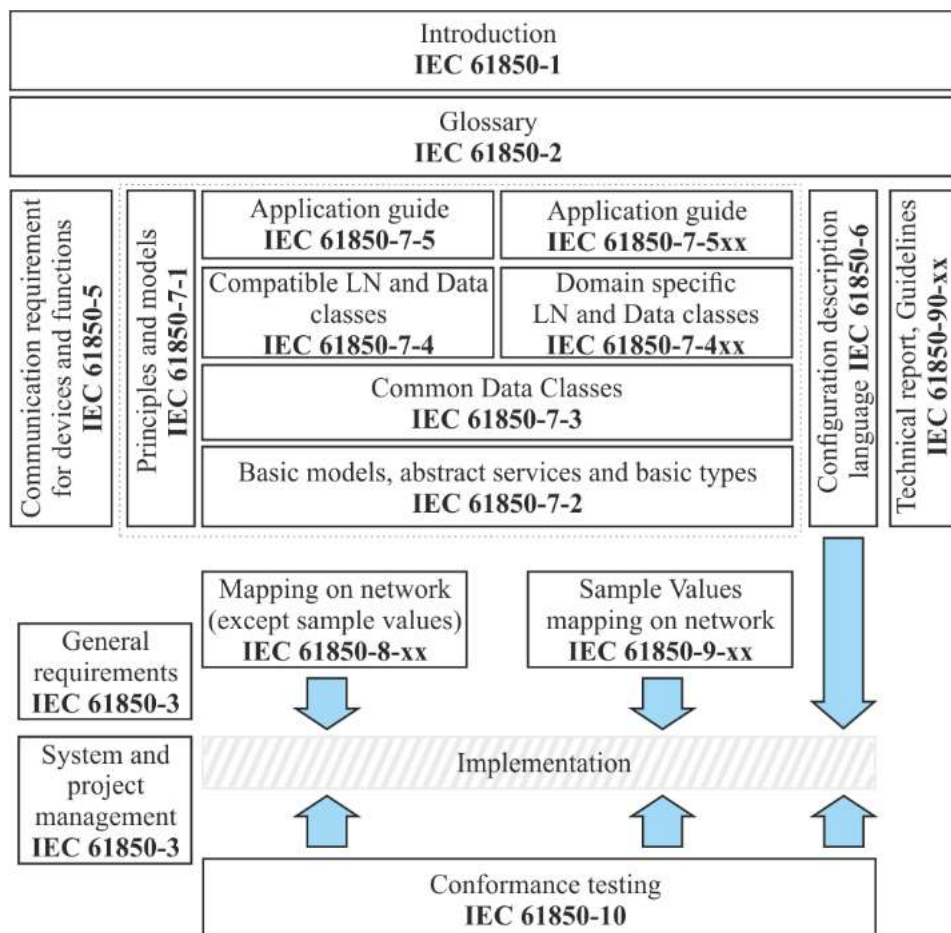


Figure 61: General standard of IEC 61850

IEC 61850 standard provides two ways to exchange information: ⁴⁷

1. Client-server
2. Publisher-subscriber

Client-server variant is used for substation automation system services to control and supervise of substation asset. It supports data read (a value or attribute), write (configuration attributes), control (controlling switching devices with direct operate or select before operate, to validate the control parameters and reserve the resource), reporting (e.g. event driven reporting), logging (local storage of timestamped data/events), get directory information and file transfer. In the client-server model for each information exchange is followed by confirmation message to guarantee the data is received by the IED using TCP/IP addressing scheme.

Publisher-subscriber variant provides two additional ways of transferring messages in the time critical services. First variant is the multicast of time critical messages by means of GOOSE mechanism and second variant by means of SV as shown in figure 16. GOOSE and SV are stream based protocol designed for high speed data transmission across the system with low latency. GOOSE service is mainly used for transmission of information like status changes, blockings, releases or trips between IEDs while the SV is used for streams of data (e.g. current and voltage samples).

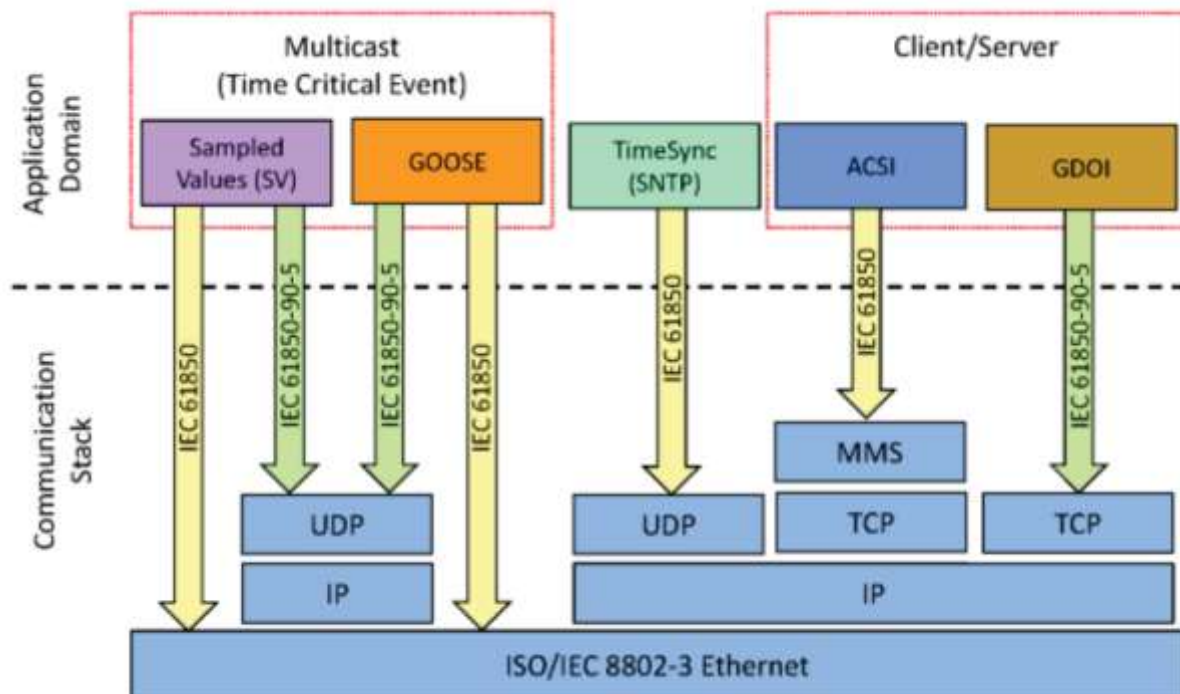


Figure 62: Protocol stack of IEC 61850-90-5

8.1.1 Data models

Data models defined in IEC 61850 standards are application independent and are object oriented data models. All application functions are broken down into the small pieces, which may communicate with each other and may be implemented separately in different IEDs. The basic objects are called logical nodes (LN). The class name of the LN refers to the function the data object belongs to. These data objects may be mandatory, optional or conditional and they contain different attributes, which represent the values or properties of the data objects.

The class names of LNs and names of data objects and their attributes are standardized. The data hierarchy is shown in Figure Fehler! Verweisquelle konnte nicht gefunden werden..

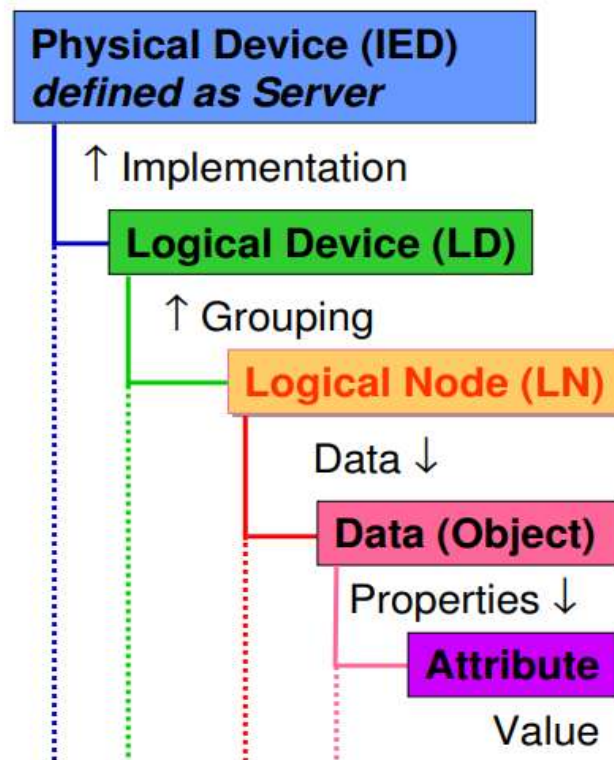


Figure 68: Data hierarchy

Example of the IEC 61850 data model hierarchy to determine the circuit breaker position:

```
A1.LD0.CBXCBR.Pos.stVal
```

Where A1 is a physical device, LD0 is a logical device (LD) that includes many logical nodes and is free to choose the name. CBXCBR1 is a logical node and is the virtual representation of device components contains the information produced and consumed by a group of domain-specific application functions. CB is prefix and is free to choose and is application specific, XCBR Class name is fixed and it stands for circuit breaker, while the suffix 1 is usually the number of instances. The maximum length of LN is 12 characters. A data object (DO) is a representation of the common information in various nodes. All devices need to follow a common naming scheme. Therefore, devices from different manufacturers can access the same device. The Pos stands for the position and stVal status value is CODED ENUM for example: 00 – intermediate-state, 01-off, 10-on, 11 bad-state.

8.1.1.1 Libiec61850 API

At PLATOON we will rely on the opensource libraries such as libiec61580, which is server and client library implementing protocols MMS, GOOSE and SV. The library is written in C applying C99 standard to provide maximum portability among different OS and hardware architectures, for IEC61850.

8.1.1.2 Client-server API

The IEC 61850 client and server API contains functions to support model discovery, reading and writing data attributes, data set handling, configuration and reception of reports, file services and control operations. The libiec61850 API is close related to Abstract Communication Service Interface (ACSI) standard as defined in IEC 61850-7-2. The library consists of IEC 61850 client API and server API (MMS over TCP/IP).

To create a server you first need to create new ledServer object, which is used for managing the MMS protocol stack and data model. The second line of the code initializes the MMS server values with default values, from this point on the MMS server can autonomously handle client connections. In the next line, the code starts the protocol stack and it starts listening to client connection. Each client connection starts in a new thread. After initialization and start used need to provide its own code to process the values from MMS or react on client activities. The last two lines stop the MMS server with closing all clients connections and cleaning up the resources we have defined during the initialization process.

```
IedServer iedServer = IedServer_create(&staticIedModel);
IedServer_setAllModelDefaultValues(iedServer);
IedServer_start(iedServer);

//main code here

IedServer_stop(iedServer);
IedServer_destroy(iedServer);
```

To make a connection to the server, you have to create IedConnection object and afterwards the connection is established by using ledConnection_connect method. Led connects takes as arguments: IedCpmmectopm object, a pointer to error stack, IP address or hostname and TCP port. After finishing the executing the main code, you have to close connection and cleanup resources by calling IedConnection_close and IedConnection_destroy functions.

```
IedClientError error;
IedConnection con = IedConnection_create();
IedConnection_connect(con, &error, "192.168.1.2", 102);

if (error == IED_ERROR_OK) {
// main code

IedConnection_close(con);
}
IedConnection_destroy(con);
```

8.1.1.2.1 Reading and writing data objects

Functions ledConnection_readObject and ledConnection_writeObject can read and writer simple and complex data objects. Both of the functions have similar input parameters: the first argument is the connection object (IeDConnection) as shown in establishing connection examples. The second argument is a pointer to IedClientError variable (client connection

example), next argument is data object reference you want to access (type const char), followed by Functional constraints enumerator. To read all values of an IEC 61850 data object you have to call multiple times readObject function. An example of reading the analog measured values from the server:

```
MmsValue* value = IedConnection_readObject(con, &error, "simpleIOGenericIO/GGIO1.AnIn1.mag.f", MX);
```

Where MX is IEC61850_FC_MX enumerator which present measured analog values, ST would mean reading status values. The MmsValue instance holds the results from the read function call, but the same instance can be used when writing data object to a server as the last input argument in writeObject call:

```
IedConnection_writeObject(con, &error, "simpleIOGenericIO/GGIO1.Nam-Plt.vendor", DC, value);
```

Additionally, the client API also allows using functions that allow read and writer native data types instead dealing with objects. An example of these function:

```
float magF = IedConnection_readFloatValue(con, &error, "simpleIOGenericIO/GGIO1.AnIn1.mag.f", MX);
```

8.1.1.2.2 Data sets

To read data sets (e.g. groups of data attributes or functional constraint data objects), you have first to define a data set and request them with a single read. The client API supports the following data related services:

- Read data set values
- Define a new data set
- Delete an existing data set
- Read the directory (list of variables) of the data set

ClientDataSet is container to store the values and consist of the following functions: functions:

```
Void ClientDataSet_destroy(ClientDataSet self);  
MmsValue* ClientDataSet_getValues(ClientDataSet self);  
char* ClientDataSet_getReference(ClientDataSet self);  
int ClientDataSet_getDataSetSize(ClientDataSet self);
```

To access data set related services the following functions can be used, such as:

- IedConnection_readDataSetValues
- IedConnection_createDataSet
- IedConnection_deleteDataSet
- IedConnection_getDataSetDirectory

An example to read the values from a data set:

```
ClientDataSet dataSet = IedConnection_readDataSetValues(con, &error, "simpleIOGenericIO/LLN0.Events", NULL);  
if (error == IED_ERROR_OK) {
```



```
printf("Read data set %s\n", ClientDataSet_getReference(dataSet));

IedConnection_readDataSetValues(con, &error, "simpleIOGeneri-
cIO/LLN0.Events", dataSet);
}
```

8.1.1.2.3 Reports

Reports are used for event-based transmission, so you do not need to send and read request to the server periodically. Reports are presented as data sets. To handle reports at the client side, the API defines following data types:

- ClientReportControlBlock –data container
- ClientReport – represents a received report
- ClientDataSet – container for the data values of a received report
- ReportCallbackFunction – callback function triggered when a report is received
- ReasonForInclusion – enumeration to indicate the reason for the inclusion of a data set member into the report

The server needs to be configured to receive reports by enabling the Report Control Block (RCB). The IEC 61850 distinguish between buffered and unbuffered reporting. To start reporting, you should first read values of the RCB with IedConnection_getRCBValues function for the unbuffered report:

```
ClientReportControlBlock rcb = IedConnection_getRCBValues(con, &error,
"simpleIOGenericIO/LLN0.RP.EventsRCB01", NULL);
```

And for the buffered report:

```
ClientReportControlBlock rcb = IedConnection_getRCBValues(con, &error,
"simpleIOGenericIO/LLN0.BP.EventsRCB01", NULL);
```

The function reads all values of the RCB from the server and creates an instance of ClientReportControlBlock.

To prepare the client to receive reports, you have to provide a callback function to the client API.

```
static void reportHandler (void* parameter, ClientReport report)
{
    //report callback function, e.g. custom event code
}

ClientDataSet dataSet = IedConnection_readDataSetValues(con, &error,
dataSetReference, NULL);
IedConnection_installReportHandler(con, "simpleIOGener
icIO/LLN0.RP.EventsRCB", reportHandler, NULL, dataSet);
```

Where dataset is used to store report data, while the function IedConnection_installReportHandler is used for installing a report handler functions and connect it with the dataset.

8.1.1.2.4 Client authentication

Only basic password authentication is supported. Example to activate authentication:

```
AcseAuthenticationParameter auth = calloc(1, sizeof(struct sAcseAu-
thenticationParameter));

auth->mechanism = AUTH_PASSWORD;
auth->value.password.string = "secretpassword";

IsoServer isoServer = IedServer_getIsoServer(iedServer);
IsoServer_setAuthenticationParameter(isoServer, auth);

IedServer_start(iedServer);
```

The first line of the code allocates memory for the AcseAuthenticationParameter data structure, second and third line initialize the structure with the method and the password which is presented as a string. Next line provides access to the IsoServer instance and feeds it with authentication parameters.

8.1.1.3 Publisher-subscriber API

For time-critical situation, the library provides API for the publisher-subscriber scenario, mainly used inside substation to distribute sample values and GOOSE among IED devices. The library supports single or multiple subscribers. For multiple subscriber option, the multicast address is used and clients have to be configured to listen to a specified multicast address.

Below a publisher example is shown.

```
SVPublisher svPublisher = SVPublisher_create(NULL, interface);

if (svPublisher) {
    SVPublisher_ASDU asdu1 = SVPublisher_addASDU(svPublisher, "svpub1",
NULL, 1);

    int float1 = SVPublisher_ASDU_addFLOAT(asdu1);
    int float2 = SVPublisher_ASDU_addFLOAT(asdu1);
    int ts1 = SVPublisher_ASDU_addTimestamp(asdu1);

    SVPublisher_setupComplete(svPublisher);

    float fVal1 = 1234.5678f;
    float fVal2 = 0.12345f;

    while (running) {
        //prepare data

        SVPublisher_publish(svPublisher);
    }

    SVPublisher_destroy(svPublisher);
}
```

In the first line creates a new IEC61850-9-2 SV publisher. The first parameter is optional for setting VLAN options and destination MAC address; the second parameter is the name of the interface over which the SV publisher should send SV packets (e.g. "eth0"). It returns new SV

publisher instance. `SVPublisher_addASDU` creates an Application Service Data Unit (ASDU) and add it to an existing SV publisher. Next three-line reserves the memory for data and maps it to local variables. `SVPublisher_setupComplete` prepare the publisher for publishing. In the main code execution, the data is prepared according to initial definition and `SVPublisher_publish` function publish all registered ASDUs.

On the subscriber side access to the data requires prior knowledge of the data set (see publisher example for data set and definition). The example of subscriber:

```
SVReceiver receiver = SVReceiver_create();

SVSubscriber subscriber = SVSubscriber_create(NULL, 0x4000);
SVSubscriber_setListener(subscriber, svUpdateListener, NULL);
SVReceiver_addSubscriber(receiver, subscriber);
SVReceiver_start(receiver);

if (SVReceiver_isRunning(receiver)) {
    signal(SIGINT, sigint_handler);
    while (running){
        Thread_sleep(1);
    }
    SVReceiver_stop(receiver);
}
SVReceiver_destroy(receiver);
```

In the first line, we define a new SV receiver instance in the next line we create a subscriber listening to SV messages or data stream that is identified by its APPID 4000h, which is the default value if not set differently by publisher. In the third line, we install a callback handler for the subscriber. Following line connect the subscriber to the receiver. `SVReceiver_start` functions start listening to the SV messages in the new thread. The callback function is called once the data is received, so we have to define the callback function as was called in the listener function:

```
svUpdateListener (SVSubscriber subscriber, void* parameter,
SVSubscriber_ASDU asdu)
{
    const char* svID = SVSubscriber_ASDU_getSvId(asdu);
    //handling the data
}
```

The data can be handled similar way as were defined in publisher example with the dual functions. We are again operating with ASDU and extracting data with dedicated functions like `SVSubscriber_ASDU_getFLOAT32()` to get float value from ASDU.

8.2 OGEMA

OGEMA (Open Gateway Energy Management) is an open software platform that supports standardized building automation and energy management. The OGEMA platform can be applied in households, commercial environment and industries.⁴⁸

OGEMA links the customers' loads and generators to building automation and energy management applications. By providing a manufacturer- and hardware-independent platform, OGEMA allows energy flows within end customer premises to be optimized with high degree of modularity.

8.2.1 Framework architecture

The core concept of the framework is to provide a hardware independent environment for energy management applications. The software is designed to be installed on a Gateway computer located between the customer and the Smart grid, acting as a firewall between the public and private communication systems. The Gateway is configured so that only certain interactions are allowed.⁴⁹

Applications installed in OGEMA can obtain access to customer devices, user displays, smart meters, measuring data, as well as data provided by external market participants, like tariff information or grid parameters. The goal is to provide a platform for Smart Building and Smart Home applications supporting the full range of Smart Grid applications at the customer side with a single efficient hardware platform, which features all the necessary communication connections. New devices and functionalities can be connected and added in a “plug&play” manner with minimum user interaction.

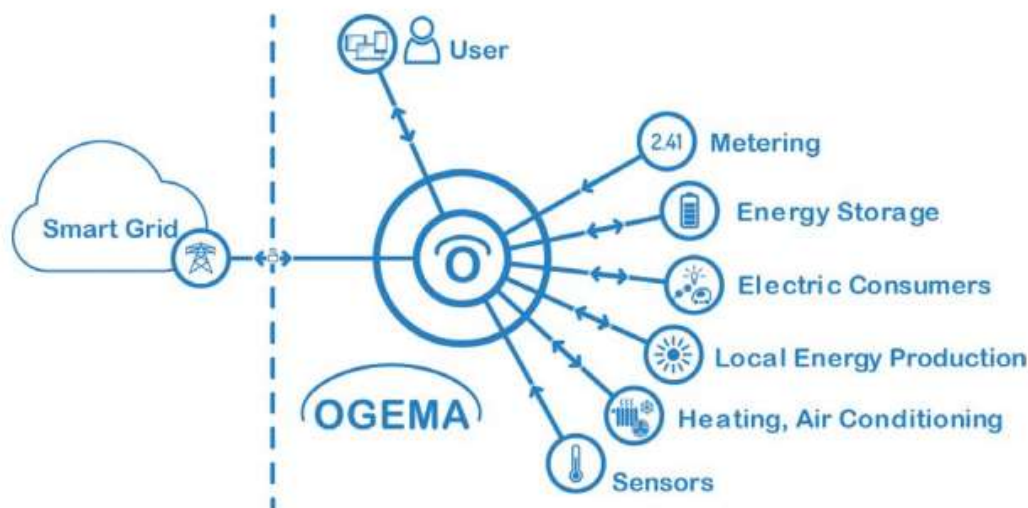


Figure 65: OGEMA framework in a Smart Grid/Smart Building environment

The OGEMA software acts as an operating system that allows applications to access different types of connected hardware and remote service providers without having to care about the actual physical realization of the connection. This is achieved by using drivers to take care of device-specifics. Just as in the case of a normal Operating System, say Linux, the drivers are hardware- or communication-specific pieces of software that transfer OGEMA language into

commands for the devices or communications. The language of OGEMA is the representation of states in a data graph called resource graph and the manipulation of these data.

The framework itself implements basic functionalities such as management of applications and communication drivers, an API to work with the resource graph, user administration and a REST interface.

However, the real value for the customer comes with the installed applications and drivers that perform services related to energy management and home automation and allow unifying devices from different vendors into one system. Depending on the installed applications, different OGEMA systems can be realized with the OGEMA framework.

8.2.2 OGEMA API

The most important further concepts of the OGEMA API are:

- A central run-time environment for applications
- Access to external devices via standardized data models and device services
- Plug & Play registration of new devices, definition of standardized services for typical functions
- Open interfaces for applications, and communication device drivers, and realization of a graphical user interface for user control of the system
- Resource control based on user-specific and application-specific access rights and permissions

8.2.2.1 Installation and Management of applications

OGEMA applications and drivers can be installed via marketplaces or from a file folder. The API functionalities for administrative applications that install new applications (including administrator interaction) are defined in `org.ogema.installationManager`. Before the installation, the required permissions must be declared by the application and must be approved by the administrator.

Declaration of required and optional permissions requested is made by two files: `permissions.perm` and `OGEMAPermissions.perm`. The `permissions.perm` file contains Java permissions (extending `java.security.Permission`) according to the Java specification, and OSGi-permissions according to the OSGi API documentation.

OGEMA applications export the OSGi service application, for which the methods `start()` and `stop()` are defined. The framework automatically detects applications based on this service. To start and stop the applications, the respective methods of the application are called. When calling the start method, the application is being passed a reference to an `ApplicationManager` that serves as the application's entry point to the OGEMA framework. Applications hence do not need to (and should not be permitted to) get OSGi services just for the sake of getting access to OGEMA.

8.2.2.2 Resource Management

OGEMA resources have resource names. For top-level resources, the name can be freely chosen by the application creating the top-level resource. The names of sub-resources are defined by

the resource type of their parent resource – with the exception of decorators, for which again the creating application can choose a name. To avoid naming conflicts between different applications, the method `getUniqueResourceName` has been defined on the `ResourceManager` service, which ensures that two applications requesting the same resource name will not get the same name but a unique one that is similar to the requested name.

The nodes in the resource graph can be addressed via paths, starting at some top-level resource (with a unique name) and then following a sub-resource trail down to the respective resource. This path is not necessarily unique for a given resource: different top-level resources may share a common subresource which has been inserted into either's resource tree via a reference. In principle, any number of paths to a resource can exist. But there is always exactly one path leading to the resource that does not contain references. This path is called the location of the resource. Access permissions to a resource are determined by its location, while listeners on nodes of the resource graph are determined by path.

An application can parse and modify the resource graph via the methods defined in the resource management services `ResourceManagement` and `ResourceAccess`, as well as Java objects of type `Resource`.

Initial access to top-level resources has to be performed via the resource management services, that return a `Resource` object (this is true for direct access as well for indirect access via resource demands, which also have to be registered with the resource management). Given an initial `Resource` object, it is possible to navigate further along the resource graph to which the object is connected. Resource objects are always associated to a path.

On top of the resource access granted to an application in terms of permissions, there are three different access modes to a resource that applications can have, which refer to the values in simple resources. By default, an application is granted shared write access to a resource, which means it can read and write the value. If an application determines that no other application should write the values in a resource, it can demand exclusive write access with some priority. An application that is granted exclusive write access takes the ability to write to the resource from all other applications, leaving them with read access. Higher priorities override lower ones; a demand for shared write access always has a lower priority than any exclusive write demand. Applications can add listeners to resources to be informed about changes in their access states.

Navigation via the resource objects allows navigation into graph nodes that do not exist (yet). Such handles on a non-existing resource are called virtual resources. It is possible to register listeners on virtual resources (e.g. to be informed when someone creates the node) and to call the `create()` method on them. After creation a resource is inactive. In this state it can be navigated to and be manipulated, but it is invisible to resource demands, and therefore to most other applications. To make it visible the resource must be activated, putting it into the active state. OGEMA does not provide default values for simple resources. An application must provide a sensible value before activating a simple resource. If it cannot do so, it should not activate the resource.

In some cases modifications to the resource graph cannot be performed with a single call to the OGEMA services offered, and a so-existing intermediate state would lead to an illegal state of the resource graph (e.g. re-setting an interval which implies re-setting the minimum and the maximum value). To avoid illegal intermediate states of the resource graph applications can combine multiple commands into transactions. Transactions are objects that can be requested by the `ResourceManager` and filled with commands. Upon invocation, all of the commands are

then executed in one “atomic” operation that guarantees that no application will see an intermediate state in which only part of the commands have been executed.

OGEMA allows defining complex resource patterns. Resource patterns consist of a definition of the root node’s type and a set of other resources with a resource type and a defined path relative to the root node. Applications can use such resource patterns to create instances of it in the resource graph – the root node then is a top-level resource. Also, OGEMA allows registering a resource pattern demand that allows an application to be informed and updated about all the matches of the pattern in the resource graph. In the latter case, the root node does not need to be a top-level resource.

8.2.2.3 Resource Listeners

The OGEMA API defines a set of different event listeners related to changes in the resource graph.

1. A resource demand listener is registered on a particular resource type, and invokes a callback whenever a resource of the respective type is activated or deactivated. Additionally, it is invoked for each suitable active resource once directly after registration. The resource demand listener is designed for the task of finding and keeping track of all active resources of a given type.
2. Resource structure listeners are registered on individual resources (implying they are registered on a path, not a location). Once registered, they are informed about structural changes of this resource, such as creation/destruction, activation/deactivation, adding/removing of subresources or the change of the path’s location. A change of a simple resource’s value is not reported to a structure listener – the more specialized resource listener has been defined for this type of event.
3. Access mode listeners can be registered on individual resources to inform the registering application about changes of its access mode to the respective resource. This includes access mode changes caused by the application’s own change requests and changes caused by another application’s requests (e.g. losing write access due to the other application being granted exclusive write access). The change in access mode is not a structure event, since it is specific to each application, while resource structure events are identical for all applications.
4. A resource change listener is registered on an individual resource, and leads to a callback whenever the value or the time series of this resource is written to (even if the write command re-writes the same value or time-series again). The listener can be registered recursively, in which case the listener is invoked whenever the value/schedule of any of the resource’s subresources changes. Due to the reference and the decorator mechanism of the resource graph, recursive registration can lead to unexpected callbacks in a complicated resource graph. A nonrecursive resource change listener registered on a non-simple resource can never be invoked.
5. A resource pattern listener is the equivalent of the resource demand listener registered on a resource pattern instead of a resource type. Instead of keeping the application informed about all active instances of a resource type it keeps the application informed about all active matches of a resource pattern.

8.2.2.4 Logging

Two types of logging are supported by OGEMA, a text logging support for applications and an automatic logging of past resource states. The text logging via the `OgemaLogger` is created as a direct extension of the `slf4j` framework. The logger API suggests applications how to perform their logging, but the actual implementation of the log commands is largely left to the framework. Particularly, an administrator application can get access to the loggers via the `AdminLogger` interface, which allows to configure some of the loggers' behavior.

Some simple resource types can be configured for automatic logging. The log data and their configuration are available from the method `getHistoricalData()` defined in the resources' interface. Data logging can be configured for periodic logging or for logging whenever the resource's value changes. The historic values are stored persistently as a time series. Access to them can be done via the direct access to the time series' entries or, alternatively, with a filtering function (e.g. the mean value over given time intervals).

8.2.2.5 Rest Interface

To facilitate remote access to the resource graph, the OGEMA frameworks provides a REST interface supporting RESTful communication using XML as well as JSON, and other communication requirements according to RFC2616. Each request to the REST interface must be https-encrypted and supply a username identifying the user (or application) whose permissions will be used for the access, and the correct password for the user. Requests send via REST are then performed with that user's access rights.

Each OGEMA resource accessible to applications is also accessible via an URI given by `.../rest/resources/<path>`, where `...` is the base URI of the OGEMA web server, and `<path>` is a valid path to the resource. For schedule resources, up to two additional pseudo-paths `/t1/t2` can be appended to the URL, which must be in the form of time stamps (in ms since epoch). If the first extra path is given, all operations performed on the schedule exclude the entries before `t1`. If the second pseudo-path is also used, schedule entries whose timestamp equals at least `t2` are excluded, too. This way, sub-intervals of schedules can be selected and operated on.

The standard HTTP methods applied to these URIs can be used for interaction with the resources. The GET request corresponds to reading a resource. It returns a textual representation of the resource. For additional configuration of the representation, serialization options can be encoded into the URL in the form `<URL>?<attribute1>=<value1>&<attribute2>=<value2>...`. Possible attributes are `depth` (an integer defining the maximum parsing depth starting from the addressed node), `references` (a true/false boolean defining whether references should be parsed as sub-resources (true) or just as links) and `schedules` (boolean defining if schedules should be included (true) or just linked-to). The default is `?depth=0&references=true&schedules=false`. Resources whose location is encountered a second time during the processing of a request can be included as links, irrespective of the references settings. This terminates the processing of possible loops in the resource graphs and avoids blowing up response messages to a request with a large depth.

The PUT request can be sent to any resource URI and requires an attached textual resource representation. If no resource exists at the URL, a 404 error is returned. PUT applies the attached resource representation to the resource at the selected URL as if the message was

applied to the resource via the `SerializationManager`. In case of schedules resources, the whole schedule content is replaced with the contents of the message. If only a sub-interval of the schedule was selected via pseudo-paths, the schedule entries outside the selected interval are unaffected by the operation. As a result of the request, the equivalent of a GET on the same URL after the request has been processed is returned.

POST with an attached resource is used to attach a new resource to the URL it is sent to. Contrary to a PUT it can also be sent to `.../rest/resources`, in which case it creates a new top-level resource. A 406 Error is returned if a top-level resource with the given name already exists. If the request is sent to a resource location the processing of the event depends on whether a sub-resource with the name as the resource in the message exists. If such a sub-resource already exists, a PUT with the attached message is performed on this sub-resource (which may cause an error if the resource types do not match). Otherwise the resource message is attached to the OGEMA resource the request was sent to as a child node, either as an optional field or as a decorator. Type and name of the new resource are inferred from the respective entries in the message. If the message contains a valid `xs:ResourceLink`, a reference to this resource is created. Sub-resources or decorators are created only if they are explicitly listed as subresource in the XML message. If POST was successful, the result of a GET request on the new resource's URL is returned. Otherwise, an error is returned.

Sending a DELETE message to a resource URL attempts to remove the respective OGEMA resource. It is equivalent to calling the `delete()` method on the resource. A successful delete returns an empty document. An unsuccessful delete returns an HTTP error code.

Access to the log data of resources is available via `.../rest/recordeddata/<path>`, where again `<path>` is the path of the resource to access. These URIs accept only the GET request and return a time series (same as a schedule) of all the recorded data. The `/t0/t1` pseudo-paths are available for the GET request to only get the data points of a sub-interval.

8.2.2.6 Data Models

All resource types are Java interfaces extending `org.ogema.model.Resource`. The basic resource types containing actual values are defined in the package `org.ogema.model.simple`; their respective array resources are defined in package `org.ogema.model.array`. For the representation of physical properties, specialized resource types exist for the most common types of physical properties, which are defined in `org.ogema.model.unit`. On top of defining the nature of the property represented by the resource, they also define the physical unit the property is measured in.

The OGEMA API only defines the most basic resources. Complex resource types and rules how to construct a resource graph using them are defined in the OGEMA data model.

9 References

- 1 <https://swagger.io/specification/#:~:text=Introduction,or%20through%20network%20traffic%20inspect>
ion.
- 2 <https://es2.slideshare.net/pjmolina/building-apis-with-the-openapi-spec>
3
- 4 <https://learn.getgrav.org/16/advanced/yaml#:~:text=YAML%20stands%20for%20%22YAML%20Ain,>
human %2Dreadable%20structured%20data%20format.
- 5 <https://www.json.org/json-en.html>
6 <https://mind.indra.es/pages/viewpage.action?pageId=442704402>
7 <https://mind.indra.es/pages/viewpage.action?pageId=390696854>
8 Semantic IoT Solutions – A Developer Perspective
(<https://www.researchgate.net/publication/336679022>)
9 SmartM2M; Guidelines for using semantic interoperability in industry (ETSI TR 103 535 v1.1.1 2019-
10)
- 11 IDS and the FAIR DATA PRINCIPLES- International Data Spaces Association
12 <https://fiware-tutorials.readthedocs.io/en/latest/relationships-linked-data/>
13 <https://fiware-tutorials.readthedocs.io/en/latest/linked-data/>
14 <https://fusion.cs.uni-jena.de/fusion/blog/2016/11/18/iri-uri-url-urn-and-their-differences/>
15 Knowledge represented using RDF semantic network in the concept of semantic web, Alena luckasoá,
Marked Vajgl, Martin Záček, June 2016
([https://www.researchgate.net/publication/303912673_Knowledge_represented_using_RDF_semantic_](https://www.researchgate.net/publication/303912673_Knowledge_represented_using_RDF_semantic_network_in_the_concept_of_semantic_web)
[network_in_the_concept_of_semantic_web](https://www.researchgate.net/publication/303912673_Knowledge_represented_using_RDF_semantic_network_in_the_concept_of_semantic_web))
16 https://www.thinginthefuture.com/spip.php?article107#main_nav_fermer
17 <https://www.w3.org/TR/rdf11-primer/>
18 <http://www.w3.org/TR/rdf11-new/>
19 https://www.thinginthefuture.com/spip.php?article107#main_nav_fermer
20 <https://www.w3.org/TR/json-ld/#relationship-to-rdf>
21 <https://www.w3.org/TR/json-ld11/>
22 <https://json-ld.org/spec/latest/json-ld/>
23 <https://www.w3.org/TR/json-ld/#interpreting-json-as-json-ld>
24 [https://json-ld.org/spec/latest/json-ld/#example-4-context-for-the-sample-document-in-the-previous-](https://json-ld.org/spec/latest/json-ld/#example-4-context-for-the-sample-document-in-the-previous-section)
section
25 <https://www.w3.org/TR/rdf11-concepts/#dfn-generalized-rdf-dataset>
26 http://travesia.mcu.es/portalnb/jspui/bitstream/10421/7478/1/JSON-LD_1_serialization.pdf
27 <https://json-ld.org/spec/latest/json-ld/#serializing-deserializing-rdf>
28 [https://oascities.org/wp-](https://oascities.org/wp-content/uploads/2018/06/8_Canera_Standards_Context_Information_Management_IoTWeek2018.pdf)
content/uploads/2018/06/8_Canera_Standards_Context_Information_Management_
IoTWeek2018.pdf
29 <https://www.openmobilealliance.org/>
30 https://fiware-datamodels.readthedocs.io/en/latest/ngsi-ld_howto/index.html
31 <https://www.fiware.org/>
32 <https://www.etsi.org/>
33 https://www.etsi.org/deliver/etsi_gs/CIM/001_099/006/01.01.01_60/gs_CIM006v010101p.pdf
34 https://h2020-demeter.eu/wp-content/uploads/2020/10/DEMETER_D21_final.pdf
35 Context Information Management (CIM); NGSI-LD API (ETSI GS CIM 009 v1.2.2 2020-02)
36 [https://www.researchgate.net/publication/330927056_NGSI-](https://www.researchgate.net/publication/330927056_NGSI-LD_API_for_Context_Information_Management)
LD_API_for_Context_Information_Management
37 NGSI-LD API: for Context Information Management (ETSI White Paper No.31)
38 [https://www.itu.int/en/ITU-T/Workshops-and-Seminars/201901/Documents/
Seongmyung_Jeong_Presentation.pdf](https://www.itu.int/en/ITU-T/Workshops-and-Seminars/201901/Documents/Seongmyung_Jeong_Presentation.pdf)
39 <https://es2.slideshare.net/flopezaguiar/data-modeling-with-ngsi-ngsild>
40 <https://www.w3.org/TR/rdf-schema/#bib-RDF11-CONCEPTS>
<https://github.com/FIWARE/context.Orion-LD/blob/develop/doc/manuals-ld/the-context.md>
<https://documenter.getpostman.com/view/513743/TVCb5B6F>

- ⁴¹ <https://projects.tmforum.org/wiki/display/API/Open+API+Table>
- ⁴² <https://fiwaretmfbizecosystem.docs.apiary.io/#reference/asset-management-api/asset-info-collection/update-product-catalog?console=1>
- ⁴³ <https://github.com/FIWARE-TMForum/Business-API-Ecosystem>
- ⁴⁴ <https://deep-hybrid-datacloud.eu/>
- ⁴⁵ <https://fiware-tutorials.readthedocs.io/en/latest/iot-agent/index.html>
- ⁴⁶ <https://iotagent-node-lib.readthedocs.io/en/latest/>
- ⁴⁷ https://www.researchgate.net/publication/316060857_A_Microgrid_Testbed_for_Interdisciplinary_Research_on_Cyber-Secure_Industrial_Control_in_Power_Systems
- ⁴⁸ <https://www.ogema.org/>
- ⁴⁹ https://www.ogema.org/wp-content/uploads/2014/12/OGEMA_2.0_introduction_v2.0.2.pdf