This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nº 101017331

D4.2 Implementation of Local CPS-IoT RSM

Deliverable No.	D4.2	Due Date	28/02/2022
Description	D4.2 describes the Resource Descriptors, the Resource Gateway, and the measurement collection software components.		
Type	Report	Dissemination Level	PU
Work Package No.	WP4	Work Package Title	CPS-IoT Resource Management System
Version	1.0	Status	Final

Authors

Name and surname	Partner name	e-mail
Luis Carrascal	INETUM	luis.carrascal@inetum.com
Juan Gonzalez	INETUM	juan.gonzalez@inetum.com
Pablo Lombillo	MYS	plombill@mysphera.com
Alejandro Medrano	UPM	amedrano@lst.tfo.upm.es
Eugenio Gaeta	UPM	eugenio.gaeta@lst.tfo.upm.es

History

Date	Version	Change
05/10/2021	0.1	Initial TOC
25/02/2022	0.2	Draft content
21/03/2022	0.3	Consolidation of input from partners
23/03/2022	0.4	Final adjustments
25/03/2022	0.5	Final pre-review
20/05/2022	1.0	Deliverable ready for submission

Key data

Keywords	IoT, resources, robot, gateway, integration, platform, layer, API, messaging, web service, architecture, components, …
Lead Editor	Luis Carrascal (INETUM)
Internal Reviewer(s)	Francesca Manni (PEN) and Marcello Chiurazzi (SSSA)

Abstract

Deliverable D4.2 “Implementation of Local CPS-IoT RSM Features v1” describes the fundamental features of the CPS-IoT Resource Management System, the ODIN platform layer that supports the interconnection of available resources. The Resource Descriptor is the key component that defines and manages the data collection infrastructure. The Resource Gateway manages communication to the ODIN upper layers. The Measurement Collection Software Components are used to register and collect performance indicators.

Statement of originality

Table of contents

List of tables

List of figures

1 Introduction

This deliverable describes the fundamental features of the CPS-IoT Resource Management System, the ODIN Platform layer that supports the interconnection of available resources. The Resource Descriptor is the key component that defines and manages the data collection infrastructure. The Resource Gateway manages communication to the ODIN upper layers. The Measurement Collection Software Components are used to register and collect performance indicators. All these components along with those described in deliverable D4.5 "Implementation of Advanced CPS-IoT RSM Features v1", make up the core ODIN components for resource management and interaction.

1.1 Deliverable context

The following table sets the context of this deliverable:

PROJECT ITEM	RELATIONSHIP	
Objectives	The deliverable is relevant to ODIN's Objective 1, as it describes and defines the software architecture of the ODIN platform to cover medical and technological requirements. The WP4 objectives are: • Specification of the CPS-IoT RMS requirements based on input from WP2 • Specification of KPI and metrics collection framework	
Exploitable results	There is no specific contribution to any exploitable results. Instead, the work presented will be the guide to create the software architecture of solution components.	
Workplan	D4.2 is attributed to WP4 tasks, WP4 - CPS-IoT Resource Management System [Months: 3-42] MYS, CERTH, UPM, INETUM. Task T4.2 CPS-IoT Resource descriptor module (INETUM) [M3-M36], is the main responsible of this deliverable.	
Milestones	D4.2 is a key deliverable for milestones PREPARATION (MS1), PROCUREMENT PROCEDURE SIMULATION (MS2), and IMPLEMENTATION (MS3) phases of the project.	
Deliverables	D3.2	Report on the Data model, ODIN semantic ontology, datasets harmonization plan.
	D4.1	CPS-IoT Resource Management System Specification
	D4.2 to D4.4	Implementation of Local CPS-IoT RSM Features v1 to v3

	D4.5 to D4-7	Implementation of Advanced CPS-IoT RSM Features v1 to v3	4.5Implementation of Advanced CPS-IoT RSM Features v1
	D7.2 – D7.7	KPI Evolution Report (I to IX)	Regarding the collection of KPIs about DevOps activities.
	D7.9	Pilot Studies Evaluation Results and sustainability	Regarding component evaluation results of unit/integration testing.
Risks	The guidelines provided in this deliverable can help in minimizing the following risks identified in the Grant Agreement: • Technologies not available in time • Technical problems during component/module development • Complexity of unification procedure		

1.2 Platform Architecture Overview

The following diagram shows an overview of the ODIN Platform architecture where the components described in this deliverable are marked:

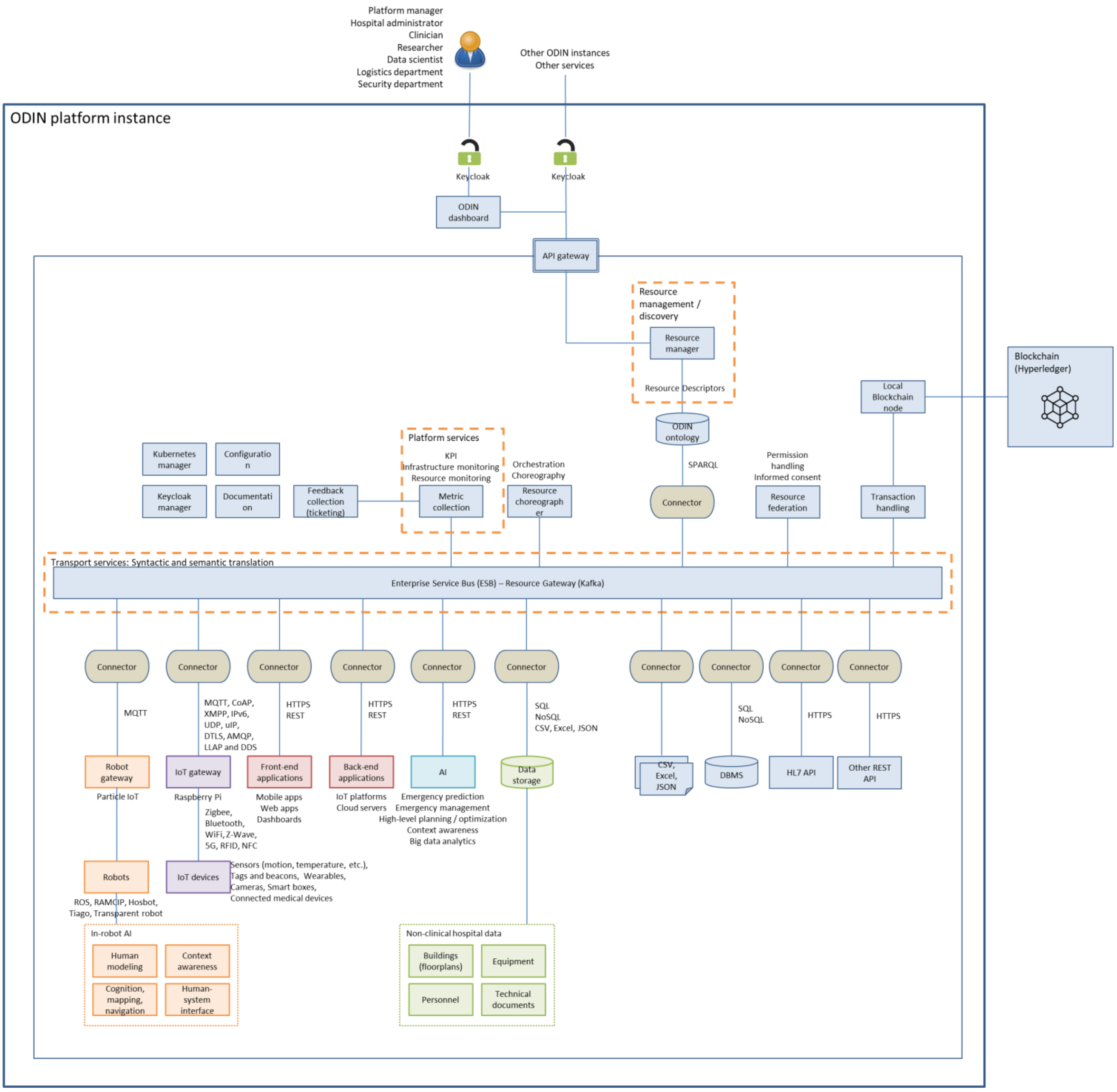


Figure 1 ODIN Platform Architecture

2 Resource Descriptor

2.1 Requirements Review

The different resource types that will be managed by the ODIN Platform are varied and heterogeneous. Therefore, the Resource Descriptor provides the abstraction layer that is necessary for a homogeneous description of data. It will specify the data structures to support multi-domain exchange of information, and provides common interfaces and will be aligned with the semantic models defined in WP3.

The information that the Resource Descriptor manages can be, but not limited to:

- Semantic Resource Description
- Resource Services
- Resource Federation
- Resource Privacy, Security, and Trust
- Resource Metric Reporting
- Resource Health
- Resource UIs
- Resource Administration
- Resource Documentation
- Resource Deployment
- Resource Communication

2.2 Applicable Technology

Three different technologies have been analysed to implement the Resource Descriptor. They will be described in detail in the following sections. In order to decide what technology will be used several possibilities will be explored, like using one of them or even a combination of them. The result of this decision will be described in the following version of this deliverable.

2.2.1 Web of Things

IoT technology today has one main downside, which is that devices do not speak a common language. In fact, there are hundreds of protocols and standards that, in most cases, are not compatible with each other. Therefore, the first objective of WoT is to create a "lingua franca" common to all devices. [1]

[1] https://webofthings.org/2017/04/08/what-is-the-web-of-things/

Web of Things (WoT) is a W3C initiative to overcome interoperability barriers between IoT devices. This initiative proposes that "things" use web standards to communicate with each other and share information. In this way the web would not only consit of digital elements but would also of physical objects in the form of virtual representations.

This common language starts to be built by standardizing the description of devices in a way that is understandable by machines. This allows both humans and computers or other devices to discover IoT things, access their information and interact with them. This description of a particular thing is called the thing descriptor and it has been decided by convention that it should be written in JSON format as it allows for the addition of contexts that make it highly specific while maintaining machine compatibility. It consists of a set of interactions based on a small vocabulary that makes possible both the integration of various devices and the interoperability of various applications.

Thing Descriptor(TD) is the most important block in the WoT architecture as it establishes a standard that describes each object and the way it is used. This information has to be understandable by a machine since communication between objects is prioritized and that is why the JSON/JSON-LD format has been chosen for the description of objects, since it allows adding contexts that make it highly specific while maintaining compatibility with machines. "A TD is instance-specific (i.e., describes an individual Thing, not types of Things) and is the default external, textual (Web) representation of a Thing." [2]

Apart from the Thing descriptor WoT is composed of several interrelated blocks, these are:

- Protocol Binding: It is a continuation of the Thing Description that provides information on how to establish an interface for each network-facing object for different protocols. This allows not only HTTP to be used, but, WoT proposes connectivity from the structural base with all web protocols such as CoAP, MQTT or WebSocket.
- Scripting API: It is an optional block that provides a standard for the creation of object control apps. It consists of a WoT Interface that allows scripts to perform the main operations on a Thing, such as exposing or consuming it, add or read properties, or retrieve its Thing Descriptor. It uses JavaScript, resembling Web browser APIs.
- Security and Privacy Guidelines: Informative document that establishes guidelines for secure implementation and configuration of IoT objects. W3C working group has identified a wide list of authorization schemes that could be added to the Thing Descriptor. Examples of supported schemes include use of API key, OAuth2.0, or Bearer tokens.

All the blocks above are implemented within a software runtime named Servient, which can act indifferently as a Server or as a Client. In the first case, the Servient is said to host and expose Things, i.e., it takes the Thing Descriptor as input and creates a dynamic object to serve the requests for accessing the exposed properties, actions, and events. In the second case, the

[2] https://www.w3.org/TR/wot-architecture/

Servient is said to consume Things, i.e., it creates a runtime resource model that allows accessing the properties, actions, and events exposed by the server Thing on a remote device.[3]

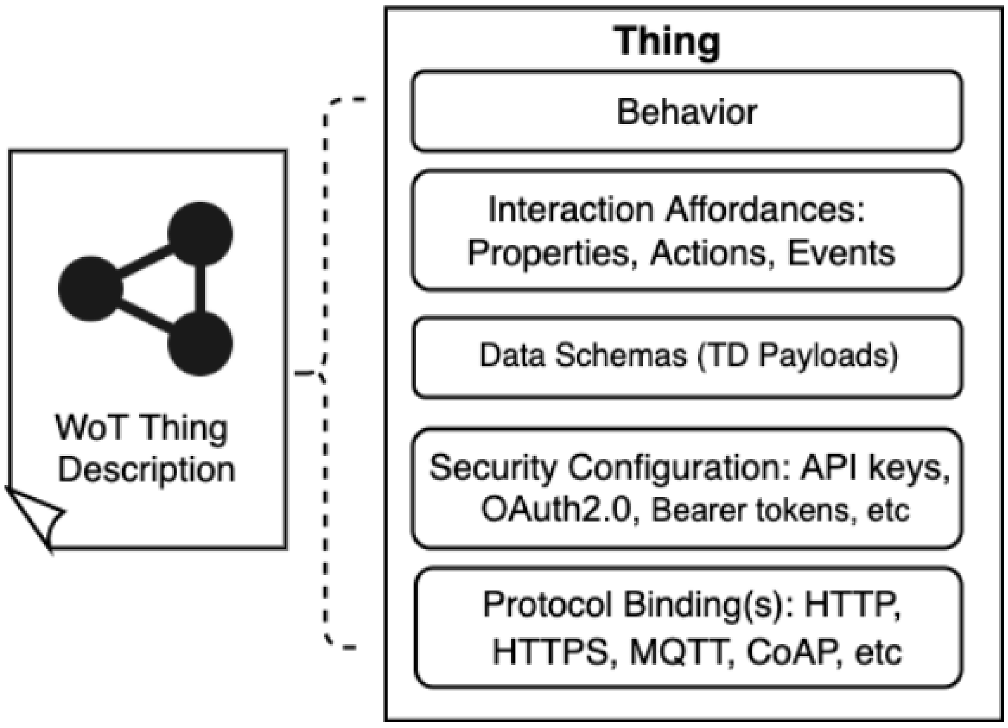

WoT Thing Description
Thing
Behavior
Interaction Affordances: Properties, Actions, Events
Data Schemas (TD Payloads)
Security Configuration: API keys, OAuth2.0, Bearer tokens, etc
Protocol Binding(s): HTTP, HTTPS, MQTT, CoAP, etc

Figure 2 Web of Things Description[4]

The thing descriptor has 4 components:

- Textual metadata of the thing
- Interaction Affordances indicating how the thing is used, how the consumer can interact with the thing
- Schemas with notation for machine-understandability
- Web links that express relationship with other things or pages.

There are three types of Interaction Affordance: Properties, Actions, and Events.

[3] https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-validation-01

[4] OpenAPI Thing Descriptions for the Web of Things – Tzavaras et al.

- Properties expose the state of a thing. This state can be retrieved (read) and optionally updated (write).
- Actions allow invoking a function of the thing to cause a change of state.
- Events imply the asynchronous sending of information from the thing to the consumer. No state is sent but state transitions.

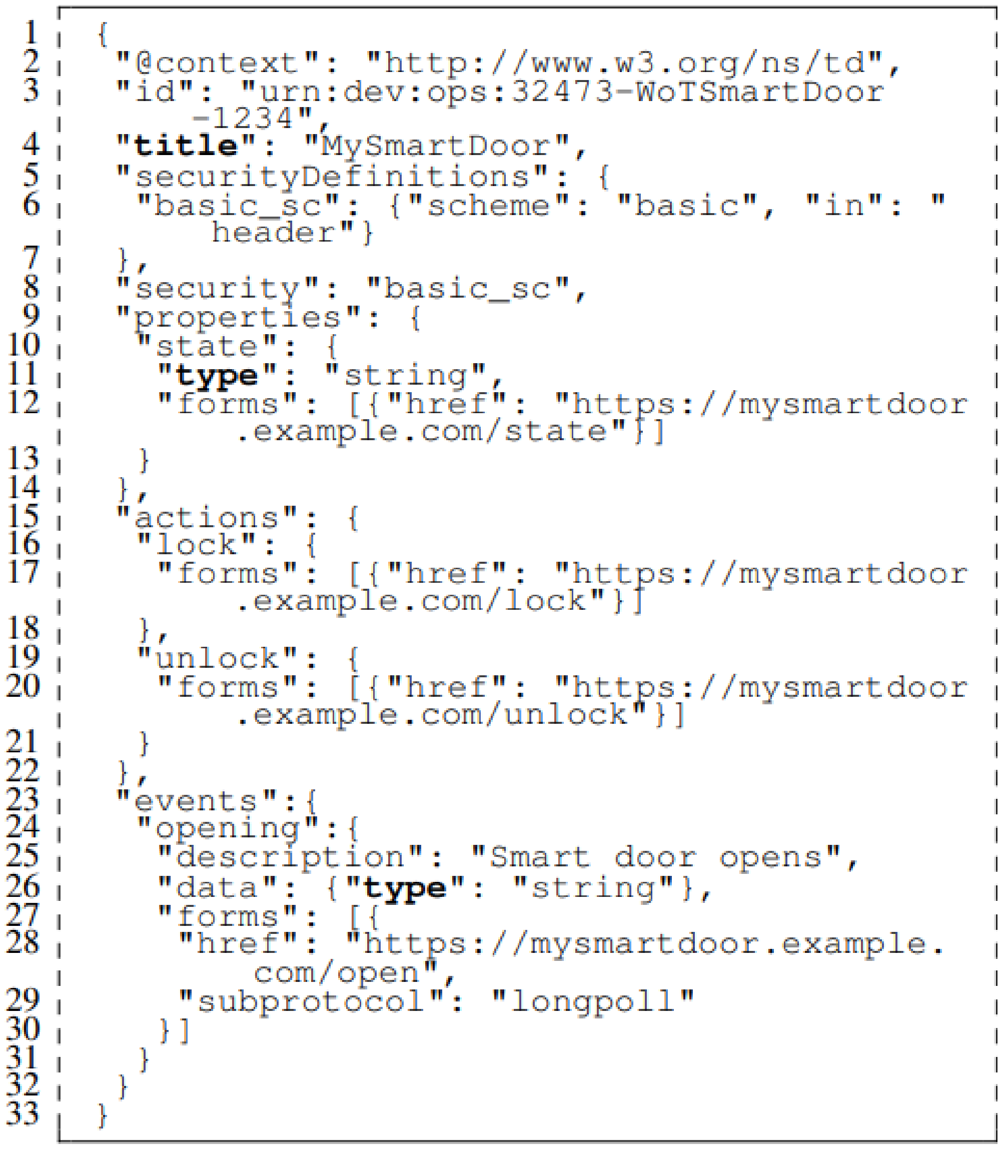
In the ODIN project this Thing Descriptor could be used to grant accessibility to all the resources in each hospital regardless of the communication protocol they implement (e.g., HTTP, Bluetooth, MQTT, ZigBee, WebSocket). This metadata accessibility would allow things metadata to be exposed in the platform so that other Things or clients (i.e., services or users) can interact with them and manage its functionality.[5]

One of the main paradigms of WoT applications is the use of well-established Web architectural principles and protocols to seamlessly interconnect intelligent objects. These include Representational State Transfer (REST), defined by Fielding and Taylor, as the main architectural interaction pattern, and Hypertext Transfer Protocol (HTTP) as the application layer protocol. In WoT RESTful applications, a smart object (typically running an embedded Web server) typically interacts with Web counterparts by exchanging requests and responses over HTTP. Although it is a standard and well-known application layer protocol, it can be too cumbersome and inefficient for implementation on limited, battery-powered devices. One way to implement WoT in these objects is through a HomeHub, built for example with a Raspberry Pi. This would communicate with the rest of the legacy objects and show them to the web providing them with a Thing Description and connectivity using the necessary protocols.[6][7]

[5] Automatic generation of Web of Things servients using Thing Descriptions – Iglesias-Urkia et al.

[6] The Web of things: Challenges and Opportunities – Dave Raggett 2015

[7] Towards a Web of Things-based system for a smart hospital – Mezenner et al. 2020



```
 1  {
 2   "@context": "http://www.w3.org/ns/td",
 3   "id": "urn:dev:ops:32473-WoTSmartDoor
        -1234",
 4   "title": "MySmartDoor",
 5   "securityDefinitions": {
 6    "basic_sc": {"scheme": "basic", "in": "
         header"}
 7   },
 8   "security": "basic_sc",
 9   "properties": {
10    "state": {
11     "type": "string",
12     "forms": [{"href": "https://mysmartdoor
          .example.com/state"}]
13    }
14   },
15   "actions": {
16    "lock": {
17     "forms": [{"href": "https://mysmartdoor
          .example.com/lock"}]
18    },
19    "unlock": {
20     "forms": [{"href": "https://mysmartdoor
          .example.com/unlock"}]
21    }
22   },
23   "events":{
24    "opening":{
25     "description": "Smart door opens",
26     "data": {"type": "string"},
27     "forms": [{
28      "href": "https://mysmartdoor.example.
          com/open",
29      "subprotocol": "longpoll"
30     }]
31    }
32   }
33  }
```

Figure 3 WoT Thing Descriptor Example[8]

Figure 3 shows an example of a Thing Descriptor for a smart door that contains:

a) A context attribute which extends the definition with additional vocabulary terms. (Line 2)

b) The identifier of the device (Line 3)

c) An indicative title (Line 4)

d) The security configuration of the service (Basic Authentication in this example). (Line 5-8)

[8] OpenAPI Thing Descriptions for the Web of Things – Tzavaras et al.

e) Interactions supported by the smart door; the state property, the lock and unlock actions, the door open event (i.e., the state property of the door turning to open).

f) The forms field that describes how each interaction can be performed; it specifies the protocol that should be used (i.e., HTTPS) and the operation endpoint.

The endpoint for getting the last smart door status value is specified in the Properties object, which is in the Forms array. The protocols and endpoints used to perform lock and unlock actions are specified by the Actions object. The protocol, endpoint, and subprotocol for subscribing to smart door open events are specified by the Events object.

2.2.2 OpenAPI

OpenAPI specification, formerly known as Swagger, is a standardized format for describing Application Programming Interfaces (APIs), resources or services understandable by humans and machines. This description contains information about different aspects of the service such as resources, endpoints, operations, parameters, and authentication and allows anyone referencing the API to understand the service.[9]

With OpenAPI, an API can be described in a uniform way. This is known as an "API definition" and is generated in a machine-readable format. In particular, two languages are used: YAML and JSON and a large set of properties are available for composing service descriptions. Technically, YAML and JSON differ only slightly, so it is possible to automatically convert an existing API definition from one language to another. However, YAML has a clearer structure and is easier for people to read. The differences are that, Basically, JSON does not support comments. On the other hand, YAML requires hyphens before array items and relies heavily on indentation, which can be cumbersome on large files (indentation is entirely optional in JSON).[10]

OpenAPI does not provide a mechanism for detecting or for dealing with ambiguities and to eliminate these ambiguities, OpenAPI properties must be semantically annotated and associated to entities of a semantic model. This mapping can be achieved representing OpenAPI descriptions using ontologies, for example ODIN ontology described in deliverable D3.2. These ontologies can capture all information in a Semantic OpenAPI description. Properties of classes are mapped to classes as well.

[9] https://es.wikipedia.org/wiki/Especificaci%C3%B3n_OpenAPI

[10] https://swagger.io/specification/

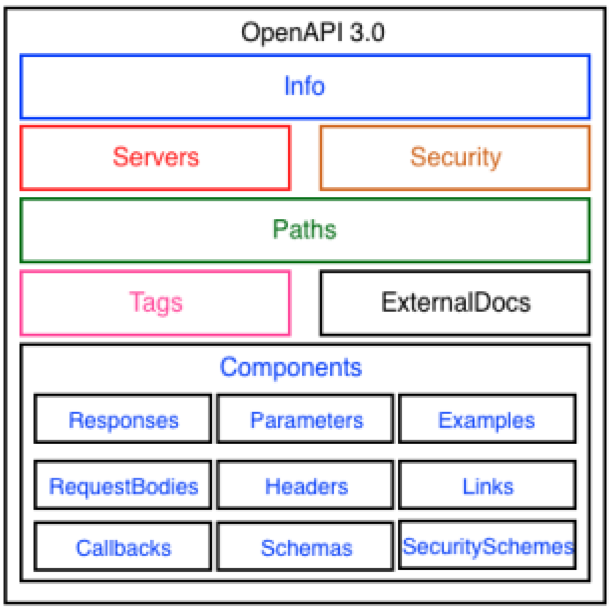
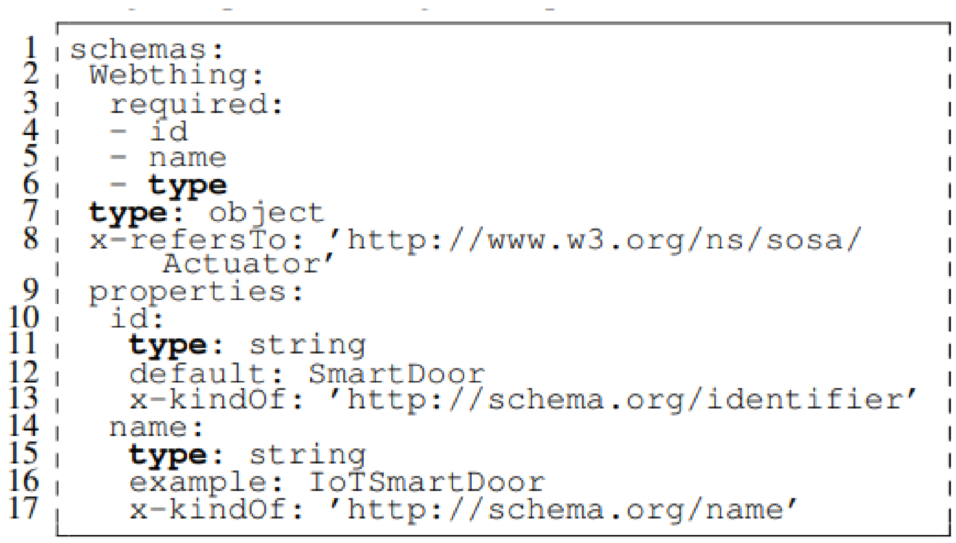


Figure 4 OpenAPI Description[11]

In contrast to WoT Thing Descriptor, OpenAPI Documents can be implemented for service description. As Figure 4 shows, it consists of many parts called objects that specify a list of properties. For example, the Info object provides non-functional information such as the name of the service, service provider, license information and terms of the service. One block, for example, is the Info object, which contains information not relevant to the operation of the service such as the name of the service, its provider, licenses, or conditions. There are also Documentation objects that provide important information, such as the Documentation, Service object, which details where the API servers are located, or Paths object that holds all the available endpoints. finally, the description service contains all possible Tag objects: a Web Thing tag, a Properties tag, an Actions tag and a Subscriptions tag. [12]

[11] OpenAPI Thing Descriptions for the Web of Things – Tzavaras et al.

[12] OpenAPI Thing Descriptions for the Web of Things – Tzavaras et al.



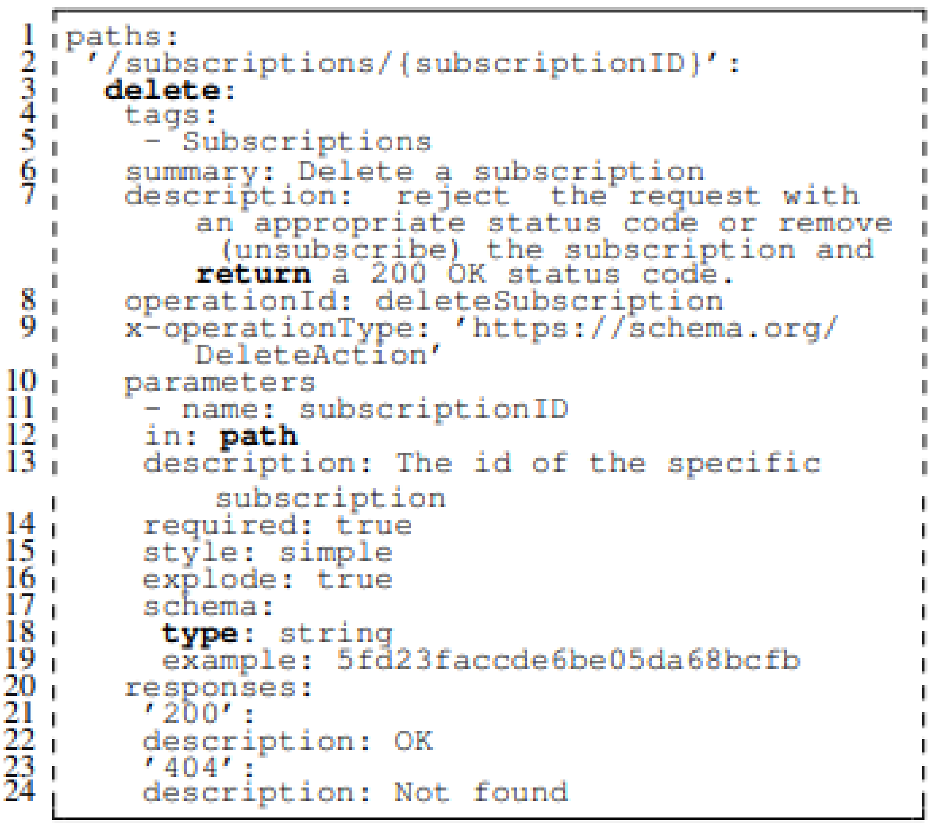
```
schemas:
 Webthing:
  required:
  - id
  - name
  - type
 type: object
 x-refersTo: 'http://www.w3.org/ns/sosa/
     Actuator'
 properties:
  id:
   type: string
   default: SmartDoor
   x-kindOf: 'http://schema.org/identifier'
  name:
   type: string
   example: IoTSmartDoor
   x-kindOf: 'http://schema.org/name'
```

Figure 5 OpenAPI Thing Descriptor Example 1[13]

In addition to properties and actions, the Thing can also support subscriptions. A subscription is the result of subscribing to a particular resource in the Thing (such as a particular property or action) and notifying you of changes in the Thing's state information (such as new temperature values). Subscriptions are stored in the storage structure and can be retrieved via the subscription ID (Thing Descriptor supports subscriptions to unsaved events).

x-refersTo is used to semantically associate the Actuator type of the smart door to the SOSA ontology. The x-kindOf extension property is used to semantically annotate the Thing properties (i.e., id, name) with concepts in www.schema.org vocabulary.

[13] OpenAPI Thing Descriptions for the Web of Things – Tzavaras et al.



```
paths:
 '/subscriptions/{subscriptionID}':
  delete:
   tags:
    - Subscriptions
   summary: Delete a subscription
   description:  reject  the request with
       an appropriate status code or remove
        (unsubscribe) the subscription and
       return a 200 OK status code.
   operationId: deleteSubscription
   x-operationType: 'https://schema.org/
       DeleteAction'
   parameters
    - name: subscriptionID
    in: path
    description: The id of the specific
        subscription
    required: true
    style: simple
    explode: true
    schema:
     type: string
     example: 5fd23faccde6be05da68bcfb
   responses:
    '200':
    description: OK
    '404':
    description: Not found
```

Figure 6 OpenAPI Thing Descriptor Example 2[14]

The example in Figure 6 shows semantic annotations for smart door operations. In this case, the operation is to delete the subscription using the subscription ID. The value of the property Is a description of the operation type given by a URL that points to this semantic description. The action types in the www.schema.org vocabulary provide a detailed hierarchy of action subtypes that can be used in properties. Humans can see the operation description to understand its intended purpose, but the machine needs additional information provided by the x-operationType extension property.

[14] OpenAPI Thing Descriptions for the Web of Things – Tzavaras et al.

2.2.3 Comparison between Web of Things and OpenAPI

Table 1 WoT vs OpenAPI Comparison[15]

WoT	OpenAPI
ADVANTAGES: • WoT Thing Descriptor can be enhanced with a context field for converting the JSON format to JSON-LD [2] • It can handle many protocols such as CoAP, MQTT, WebSocket. [3] • WoT description uses events to represent state transitions (simpler) [4] • WoT is specific for IoT and it applies to any IoT application domain, from consumer electronics to heavy industries	ADVANTAGES: • Enriched with text that can be understood by humans providing both, human and machine-readable descriptions of Web services [1] • OpenAPI defines services in a way that eliminates ambiguities and provides Web Thing service descriptions which are uniquely defined and discoverable [6] • OpenAPI meets the HATEOAS requirement of REST architectural style [7] • It is possible to convert an OpenAPI description to an ontology • OpenAPI is supported by a complete tool pallet (e.g., editors, description validators and client SDK generators)
DISADVANTAGES: • Description is a much shorter document [1] • Ambiguities: The same property may appear with different names [6] • Does not support HATEOAS requirement of REST architectural style [7]	DISADVANTAGES: • Does not support JSON-LD [2] • only supports HTTP(S) and webhooks [3] • Subscription to property changes is more complex [4] • Simpler security scheme than WoT [5]

The main common disadvantage of these two technologies is that it is necessary to handwrite the description of each resource on the ODIN platform. These are many resources across the entire

[15] OpenAPI Thing Descriptions for the Web of Things – Tzavaras et al.

infrastructure and considering healthcare, technologies such as FHIR already have descriptions implemented for many of the resources used in the smart hospitals being proposed.

Main Differences

- Compared to OpenAPI, Thing Descriptor is a more abstract description of a Thing that lets the client interact with the device (description is a much shorter document). OpenAPI is detailed and complete: It fully describes the functionality of a device and provides all the information a client needs to use the services it provides.
- OpenAPI resorts to JSON or YAML and WoT to JSON or JSON-LD.
- A Thing Descriptor may also refer to extra IoT protocols (e.g. CoAP, MQTT), while OpenAPI only supports HTTP(S) and Webhooks.
- Thing Descriptors can describe events, while OpenAPI documents can describe subscription operations using Callbacks or Webhooks properties added to a Path object.
- Similar Security scheme but more detailed in WoT Thing Descriptor.
- Both have ambiguities but OpenAPI can eliminate them by mapping to semantic models (ontologies).
- OpenAPI meets the HATEOAS requirement of REST architectural style while WoT does not.

2.2.4 FHIR

The HL7® FHIR® (Fast Healthcare Interoperability Resources) standard defines how healthcare information can be exchanged between different computer systems regardless of how it is stored in those systems. It allows healthcare information, including clinical and administrative data, to be available securely to those who have a need to access it, and to those who have the right to do so for the benefit of a patient receiving care. The standards development organization HL7® (Health Level Seven®) uses a collaborative approach to develop and upgrade FHIR.[16]

Table 2 FHIR vs HL7v2 Comparison

FHIR	HL7
• It is a new standard • It simplifies implementation without sacrificing information integrity • Can be used standalone or integrated • Supports and encourages alignment to HL7's previously defined patterns and best practices	• It has been in existence and in use for 20 years. • It was the first information exchange standard and is one of its most widely adopted • Uses messages composed of re-usable segments to communicate healthcare-related information

[16] https://www.hl7.org/fhir/

• All exchangeable content is defined as a resource. • It is developer friendly	

FHIR resources can be defined by thing descriptions, but this happens at two levels of varying complexity. The first level consists of simply representing things that use FHIR resources with a thing descriptor. The second level aims to deepen the definition of FHIR resources to ingest and use the data produced by these services.

In the first case, to describe a resource, for example, which generates FHIR information, this data production can be represented as an action. On the other hand, for FHIR resource consumption to produce data that can be interpreted by machines, it is necessary to go beyond defining an action in the Thing Descriptor. The FHIR specification defines its resources using a template-based data model, making human interaction necessary for the interpretation of its meaning. To create machine-interpretable representations, one needs to transition from syntactic models based on serialisation formats to formal semantic models that are serialization-agnostic. In other words, FHIR resources need to be described using a formal ontology.[17]

The main advantage of using FHIR in the ODIN project is that it is a formalized and matured definition accepted worldwide for healthcare use cases. This implies that the vast majority of the services and resources used in the proposed smart hospitals are already defined in the FHIR documentation. FHIR devices include durable (reusable) medical equipment, implantable devices, as well as disposable equipment used for diagnostic, treatment, and research for healthcare and public health, as well as devices such as a machine, cellphone, computer, software, application, etc. The Device Definition resource is used to describe the common characteristics and capabilities of a device of a certain type or kind, e.g., a certain model or class of a device such as an x-ray model or personal wearable device model, whereas a Device resource documents an actual instance of a device such as the actual x-ray machine that is installed or the personal wearable device being worn. There are not only devices among the resources described, FHIR has a wide variety of resources in its description, all related to the health sector. Among these we can find locations, personnel, diseases, organizations, medicines... This is very convenient for the ODIN project since most of the resources to be used are already defined.[18]

[17] Gatekeeper, Web of Things (WOT) Reference Architecture – D3.3.2

[18] https://build.fhir.org/devicedefinition.html

FHIR has been working for years, it is limited to the syntactic part delegates perfectly all the semantic terminology and allows the use of a great variety of clinical technology. Besides, all the operations and relationships have to be established, which is not as inconvenient as having to write one by one all the descriptors as it happens with WoT or OpenAPI. Finally, a capability statement must be defined. This is a set of capabilities (behaviours) of a FHIR Server for a particular version of FHIR that may be used as a statement of actual server functionality or a statement of required or desired server implementation.[19] On the other hand, FHIR has many other advantages such as being open-source, free, easy to implement and developer-friendly.

[19] https://www.hl7.org/fhir/capabilitystatement.html

3 Resource Gateway

3.1 Requirements Review

The Resource Gateway oversees connecting all the resources to the platform and to the rest of the services available in it.

Most important requirements are:

- Allow easy connection of new resources.
- Follow an Enterprise Service Bus approach with a messaging system.
- Be the gateway to the rest of the services.
- Allows access to the services in a secure way

3.2 Architecture Review

From the D3.10 ODIN platform v1, the architecture follows a service bus where all the services can communicate, as depicted in Figure 1.

Important services or components of the architecture proposed are:

- The Service Bus, which allows exchanging messages among services in the platform.
- The Gateway, which oversees connecting users and external services to the platform in a secure way.
- The transport services that connect to the resources, such as IoT, robots and other services. Those services read or write information from and to the resources bridging protocols and making them interoperable through the platform.

D4.1 CPS-IoT Resource Management System Specification reviews several technologies to be used for the whole architecture of the CPS-IoT system.

In the following section, a light overview of concrete solutions of those technologies will be carried on discussing the best options.

3.3 Applicable Solutions for Messaging Bus

3.3.1 Kafka

As Kafka website[20] announces, Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications.

[20] https://kafka.apache.org/

Kafka follows the pub/sub architecture and is useful for data integration as they claim and is based on SCALA language.

Most important attributes of Kafka are, scalability, high throughput, distributed, reliability as storages every message sent and high availability. All the mentioned attributes are pointed as must in D2.2 Hospital Requirements Report.

Moreover, the ecosystem it has, makes Kafka very appealing as it supports lots of client libraries to be used in applications to read, write, and process event streams, in addition to its out-of-the box support for other tools, such as databases, message systems, analytic platforms and many more.

Kafka is based on several concepts through which covers its mentioned attributes:

- Server: Kafka servers do have 2 implementation modes and are the base of the infrastructure. The first mode is the Broker or storage, where messages are received and stored in topics. The second mode is supported by Kafka Connect, which connects to the integrated systems to read and write event streams from and to Kafka brokers. The servers can run individually or in cluster across different datacentres or cloud regions. Each Kafka cluster can also connect to other Kafka clusters. In a cluster, if a Kafka server goes down, the rest take its workload to continue working.
- Client: clients make possible to read and write data to applications and services in a distributed way. Lots of client libraries exists and specific libraries to process streams are also available like Kafka Streams and a REST API to connect directly in case no library is available.
- Event, topics, Producer and Consumer: these are the same concepts in pub/sub architecture[21]. To summarize, Producers create messages in the form of Events that are stored in Kafka in Topics or themes that Consumer read.
- Partition: topics are distributed among several Brokers, so the workload is distributed also, providing scalability. If a copy of the topic is created in another datacentre, then we provide fault-tolerance and reliability.

[21] Please refer to D4.1 CPS-IoT Resource Management System Specification. Section 6.5 Event Driven Architecture.

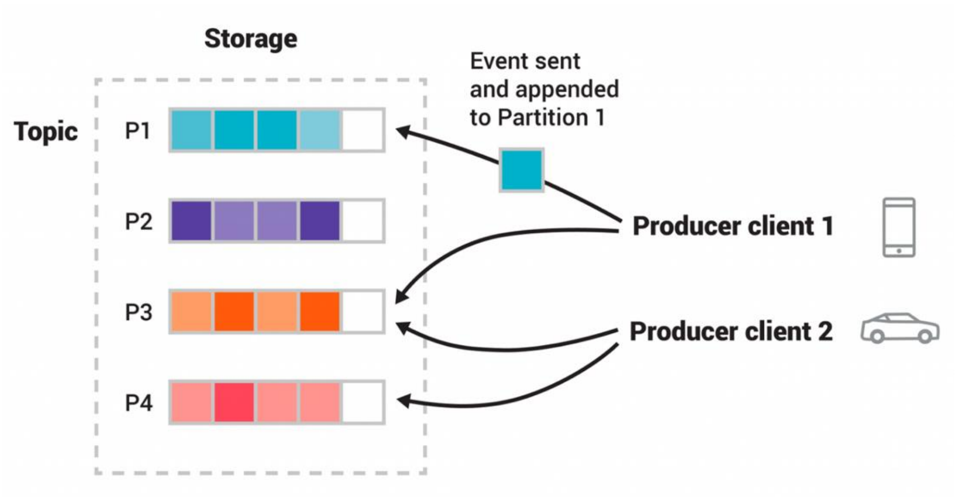


Figure 7 Kafka Architecture with partitions

Security

The default setup [22]lets any producer or consumer, write, or read messages to and from any topic, which is not very appealing in some cases where multitenancy is implemented with one cluster.

Kafka allows several security measures beyond partitions which are not protected. SSL encryption is available at the cost of some performance degradation. SSL (mutual certificate authentication) or SASL authentication (Simple Authorization Service Layer) which supports several kinds of authentication, is also a good option when there is a need of identifying which applications can connect to the system. To complete the security options, to get a tighter control, authorization can be accomplished using Authorisation Control Lists, so using a rule-based list, it can be defined exactly what producers/consumers can do.

Metrics

Kafka uses Yammer Metrics for server and cluster operation but uses its own Kafka Metrics when it comes to the clients connecting to Kafka, so they use another format, but both expose them through JMX, which helps caring about the messaging system wellness. There are lots of metrics available[23], mostly related to messages statistics and cluster status. Depending on the component (server, producer, stream, consumer) the metrics change.

Deployment and setup

Regarding deployment, Kafka does not offer its own solution to be used on Kubernetes but can be used using Strimzi[24]. This eases its configuration and deployment as this opensource project maintains Kafka images and tools to use it with Kubernetes.

[22] https://kafka.apache.org/documentation/#security

[23] https://kafka.apache.org/documentation/#monitoring

[24] https://strimzi.io/

Regarding the setup, Kafka can be configured using files or programmatically, which offers a dynamic management in case it is needed to be integrated form a management component for example.

Other options available are automatic creation of topics or manual creation, adding another plus of dynamic behaviour and automation facilities.

3.3.1.1 Usage in ODIN

Kafka is useful for several use cases, but the most important and the ones that can be profitable for ODIN are:

- Messaging like a message broker would do, but with better throughput than RabbitMQ or ActiveMQ, which are traditional message brokers.
- Metric and aggregation and data availability to be processed. This way Kafka can help routing the data to the right tools to monitor systems or applications and process IoT data for example.
- Stream processing is another field were Kafka shines, providing a very powerful tool to perform real time or batch processing. This way, data gathered from IoT devices for example, can be processed to enrich the data to get more complete messages that are meaningful for other services or applications connected to Kafka.

Although is very powerful, Kafka lacks routing capabilities or priority queues as it serves messages in the received order.

Defining and implementing a very rich messaging protocol and a careful topic selection should be done if we decide on using Kafka. This decision will be taken in the next phase of the task.

3.3.2 RabbitMQ

RabbitMQ[25] is a traditional messaging broker based on the AMPQ protocol which supports pub/sub and point to point communication, but it also supports other protocols such as STOMP and using and HTTP bridge can send MQTT messages with some limitation.

RabbitMQ is also open source as Kafka, and beyond pub/sub which is the mechanism most aligned with ODIN's architecture, its main attributes are also aligned with ODIN requirements.

Message queuing with prioritization and routing based on topics or message content provides the roots for data integration requirements.

[25] https://www.rabbitmq.com/

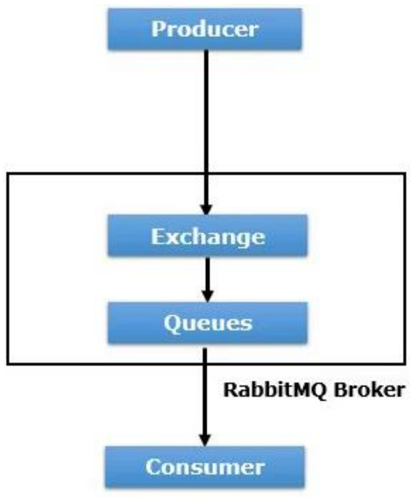


Figure 8 RabbitMQ components

In Figure 8 it can be seen the usual pub/sub components plus the roots of the broker, where the Exchange component is in charge of routing the messages to the correct queue using defined rules.

Security

Security is assured with authorization and authentication mechanisms such as OAUTH, and support basic User-Password, HTTPS and Lightweight Directory Access Protocol (LDAP) or other sources of identity and authorization such as digital Certificates.

Reliability is achieved with the ability to check message delivery status with confirmation acknowledge and data replication using Raft consensus algorithm on cluster mode and Quorum queues, which are better than just data replication, avoiding process faults or data loss.

High availability through cluster and data federation is also available using plugins that make possible to have nodes in a cluster dispersed geographically.

Metrics

Another plus for RabbitMQ it is the Management UI it must control all the functionality. This helps getting a one place to perform all the logistics and monitoring.

Monitoring the bus and getting metrics from the nodes also is a very good feature. MQRabbit can be easily integrated with Prometheus and Graphana to do so. Monitoring the wealth of the bus is key to keep things working. Number of messages sent and remaining in the queues, and other metrics are also important to manage the message bus status. By the way MQRabbit reports most usual infrastructure metrics such as CPU, memory usage, throughput among others.

Deployment and Setup

RabbitMQ offers good integration with Cloud and DevOps tools, making it a good deal when high speed development is a requirement.

Bare metal deployment is useful for small projects, but big ones need tools such as Docker or Kubernetes, to manage clusters and RabbitMQ offers 2 plugins for Kubernetes. First plugin is Cluster Kubernetes Operator that automates provisioning, management, and operations of RabbitMQ clusters running on Kubernetes. The second one is RabbitMQ Messaging Topology plugin, which manages RabbitMQ messaging maps within a RabbitMQ cluster using the RabbitMQ Cluster Kubernetes Operator.

As previously commented, RabbitMQ offers a Management Dashboard so once started the broker or cluster, it can be managed. On the other hand, part of the setup must be handled using configuration files.

3.3.2.1 Usage in ODIN

As Kafka, RabbitMQ is useful to be used as the service bus where all the components and services connect to exchange data. The routing capabilities, queue support and pub/sub make it good to connect high level components and services of ODIN architecture.

Due to its performance limitations, data analytics or high volume of data integration may not be its strong point. It also lacks stream processing but may be integrated with other tools and libraries.

3.3.3 Bus Solution Comparison

In the following table, a light comparison among the solutions is performed to discuss which one is better for implementing the messaging bus.

Attributes (1-none, 10-all)	MQRabbit	Kafka	Reason
Requirements fit	9	8	Kafka is more oriented to streams and has less options than RabbitMQ out of the box[26]
Ease of adoption	8	7	More is better. References[7,27] point that Kafka is harder to adopt
Scalability	7	10	Ping checks and some monitoring task, plus confirmation of messages have a bad impact on RabbitMQ. Kafka has horizontal scalability while RabbitMQ has vertical scalability.
Reliability	8	9	Both support good means to deliver and track the messages
Security	10	10	Both include similar options (Kerberos, OAuth, etc)
Data integration	10	10	Both serve for data integration.

[26] https://www.cloudamqp.com/blog/when-to-use-rabbitmq-or-apache-kafka.html

[27] https://freshcodeit.com/blog-introduction-to-message-brokers-part-1-apache-kafka-vs-rabbitmq

Support	8	8	Documentation is well supported and there are lots of resources for both
Deployment facilities	10	8	Kafka does not offer Operators for Kubernetes, while RabbitMQ does.
Size of project to be used	5	10	More is better. RabbitMQ is for small to medium projects, Kafka is for medium to big projects. Scale is assuming that ODIN is mid to big. Averall recommendations from references[7,28]
License	Mozilla Public License 2.0	Apache License 2.0	Both are open-source. Kafka has Apache License 2.0 (non-copyleft) while RabbitMQ has Mozilla public license 2.0 (weak copyleft)
Number of projects using it	1831	1251	References[29]
Number languages	10	5	
Monitoring	10	10	
Hard dependencies on other projects or solutions	Developed with Erlang	Apache Zookeeper for shared configuration	
Innovation impact	5	10	RabbitMQ is a traditional messaging broker. Kafka is being used in top projects not only for analytics but also for messaging bus by enterprises such as Zalando and other top ones.
Routing capabilities	10	0	It is still under analysis whether routing or storage is needed
Storage	0	10	
Multi-tenant	10	10	
Real time	0	8	RabbitMQ is not suitable for real time. Kafka can work in near real

[28] https://freshcodeit.com/blog-introduction-to-message-brokers-part-1-apache-kafka-vs-rabbitmq

[29] https://stackshare.io/kafka

			time, being end-to-end latency from producer to consumer as low as ~10ms if the hardware and network setup are good enough.
Pub/sub	10	10	
Queuing	10	9	

Other solution that has been considered is MQTT which is not as suitable as the presented ones. MQTT is suited to support communication over an unreliable network and scales bad. Also, ActiveMQ is another option similar to RabbitMQ.

3.3.4 Features Implemented in the First Version

For the first version of the platform the list of features to be implemented are:

- A mechanism to decide the list of the topics that need to be created in the messaging bus related to resources, services, and storage
- A mechanism to implement response to the message sender, when needed in case the bus does not support it.
- The configuration of the bus to work under the situations to be managed

3.4 Applicable Solutions for the Gateway

3.4.1 APIMAN

APIMAN[30] is an opensource project under RedHat umbrella, created to manage API's,

APIMAN separates from the API's code the tasks of controlling API secure access, throttling, quotas, metrics and API developer portals to publish documentation.

In the following image it is described how the API Gateway manages the API requests, acting as a proxy gateway.

[30] https://www.apiman.io/latest/index.html

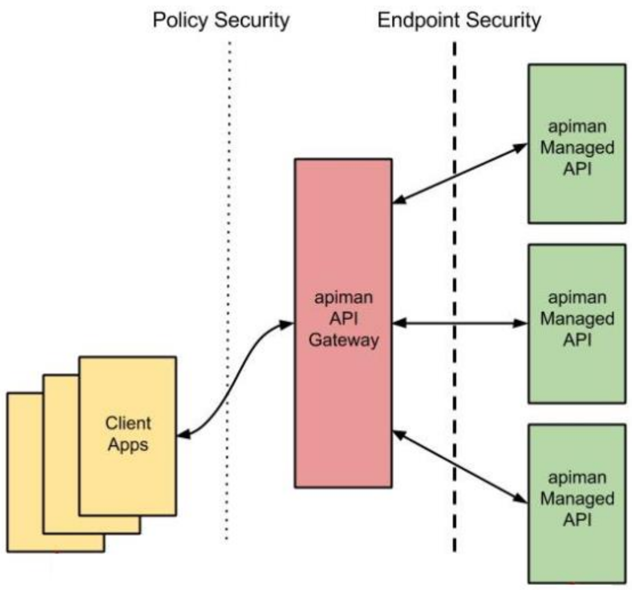


Figure 9 APIMAN Gateway schema

Another strong point is the support of UI and REST API to manage APIMAN, offering powerful option to automate tasks to be integrated for example with a Management module for ODIN platform.

Security

The secure access has several options to control who, what for and when any application or user can access the exposed endpoints. APIMAN is fully integrated with KeyCloack, which has already been selected on WP3 to be the source of identity and authorization. Moreover, APIMAN can specify different rules for the different versions of the API published and control different configurations for each application connecting to them.

APIMAN data model is around Organizations, Plans, APIs, Client Apps and Policies. With these concepts, it manages everything around.

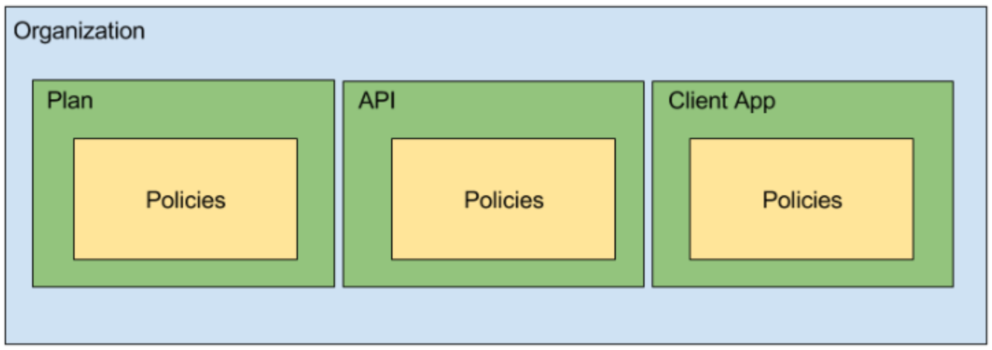


Figure 10 APIMAN data model

As illustrated in Figure 10, an API represents the endpoints exposed to the world. An Organization is the holder or owner of an API that is published. The Client Apps use the API under some restrictions defined in the Policies. The Policies control the rules to be applied to an API. The Plans are sets of Policies to be applied to an API for different Client Apps, so an Organization can have an API with 2 plans and each Plan is used by different Clients because one uses a free reduced access of the API and another Client uses a premium access with higher throttling, quotas and permissions. The model is very powerful.

The way APIMAN handles requests to the API is controlled by API Key, so a Client App must use a valid API Key attached to the request to call the API. Once the PAI Key is verified, the API Gateway (APIMAN) resolves all the polices against the request. When the API resolves the work to be performed in the call, answers to the Gateway which resolves new policies until returns the response to the Client App.

The types of policies to be applied can be grouped into:

- Security policies: type of authentication (Basic, Mutual TLS with certificates), whitelisting, etc
- Limiting policies: rate of requests per second, number of requests during a period of time, etc.
- Modification policies: url rewriting, json transformation, etc.
- Other: cache, logging, etc.

Metrics

Once an API is published, metrics can be obtained in the form of time to resolve requests, time to process the API request to response time, endpoint resolving the requests, status of the requests, throughput, bytes uploaded and downloaded per endpoint, etc.

Which is more important is that metrics can be accessed through REST API or using the UI management board.

Metrics are recorded in Elastic Search, so, it is a dependency during deployment.

Deployment and Setup

APIMAN can be deployed as a single gateway or can be configured to offer several gateways which can be very useful to manage Production and Development environments, easing DevOps approaches.

APIMAN uses JBOSS or Wildfly to run, using JAVA 1.8. Moreover, APIMAN supports Docker so other deployments can be used through Kubernetes for example.

The Console Manager helps creating the Organizations, Plans, Policies and Client applications, making easy the set up. API are also published using the manager.

For example, after publishing and API it can be accessed using the following schema:

```
http://gatewayhost:port/apiman-
gateway/{organizationId}/{apiId}/{version}
```

So, the URI, contains the address, port and the Organization and API ID, plus the version. This allows publishing several versions of an API, coexisting.

3.4.1.1 Usage in ODIN

APIMAN can be used as an API Gateway to publish the platform API to external Application clients or services. It's integration with KeyCloack and set up facilities makes it a very good candidate to implement ODIN Resource Gateway.

The plans can be used to publish different API for example to be used from external services or internal services of the platform with a unique tool, controlling not only version access but monitoring and controlling quotas and throttling.

The management API is something that makes APIMAN very appealing as it leads the implementation of dynamic resource API publishing, something that is desired in the ODIN platform.

Finally, the Developer Portal is a plus, as eases the management of the documentation for each version of the API, making I available to the developers.

3.4.2 Features Implemented in the First Version

The following actions could be the first steps to be covered.

- Define the data model for the API Gateway, naming the organization for each hospital, defining plans and policies.
- Define the API's and services that will be published.
- Create the dynamic API publish controller, so resources can publish their API when registering
- Define the publish strategy for production and development
- Define the versioning strategy

3.5 Applicable Solutions for Transport Services

Transport service manage the task to get the data for IoT, Robots, AI and Data resource, and help introducing the data into ODIN platform.

There exists lots of types of protocols and approaches. One approach is to create an adapter between protocols, second one is to use pre-created adapters and finally using a platform which already has integration to those resources, such as an IoT platform. The first approach is out of the scope due to the huge number of different protocols that potentially could be integrated into ODIN.

3.5.1 Apache Camel Components

Apache Camel[31] is an open-source project which mission is to provide an integration framework in order to connect different systems producing and consuming data.

Apache Camel follows the enterprise Integration Patterns presented in D4.1 CPS-IoT Resource Management System Specification, Section 6.2 Enterprise Integration Patterns, offering a wide range of components that are premade, such as adapters.

[31] https://camel.apache.org/

Camel has several core concepts over which creates its integration framework and architecture.

The CamelContext is the runtime where everything is integrated and serves as the point of access to the rest of the modules.

The Routes are definitions that can be set using DSL language or programmatically, to set the path that a message follows inside Camel. A route has 1 input and one or several outputs.

Processors or message filters are the handlers of messages between other modules in Camel to transform, enrich validate or perform more complex tasks with the messages, implementing all the EIP's.

Finally, the Components are the exposed endpoints to integrate external systems and are the most important for the WP4.

An overview of the concepts and architecture is shown in the following image.

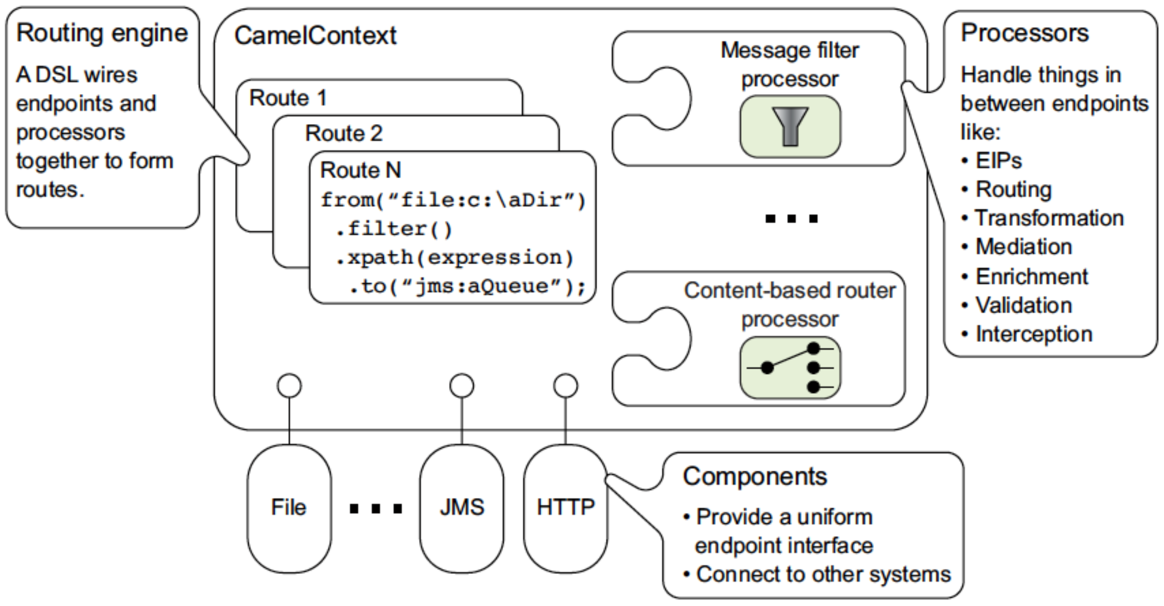


Figure 11 Apache Camel Core concepts and Architecture

Security

Apache Camel offers several types of security levels, where the most important is at route level.

To summarize:

- Route Security – Any interaction can be forced to be authenticated and authorized. To support this, Apache Shiro and Spring Security can be used.
- Payload Security - Messager encryption/decryption services can offer secrecy among endpoints of the routes, so the data remains encrypted while in the pipes. The operations can be on part or full message.
- Endpoint Security – In case any endpoint wants to implement its own security in terms of authentication, authorization, or encryption, it is also possible, beyond the security offered at route or message level.

- Configuration Security – This kind of Security is used to manage sensitive information from configuration files that can be distributed or local.

Metrics

Camel offers JMX support32 by default, so it can be monitored several points out of the box. This supports offers integration with tools such as Prometheus which is one of the most promising tools to be used in ODIN's KPI and monitoring component.

Network and Java Virtual Machine checks are available through pings and remote invocations to get data about the JVM and the applications running on it.

Common CPU, Memory and Disk metrics are basic information included with JMX support.

The key metrics about Camel are also exposed, such as route's status, component list and behaviour, list of processors, thread pools and lots of information to check the status of Camel.

Deployment and Setup

Apache Camel can be deployed in typical environments with web server like Tomcat or Wildfly, using it embedded in a Java application, using containers such Spring Boot or as an OSGI container as Apache Karaf, but all of them start with creating a CamelContext.

Each of the ways to use Camel offer pros and cons. For example, Java applications offers flexibility but start and stop lifecycles is hard and tedious and all the JAR's must be loaded into the app.

Containers and web servers handle better Camel lifecycle and offer monitoring but are more resource consuming.

At the end what is most important is that usually running Camel in production must be started carefully to avoid errors with the routes, so a good setup must be performed.

Beyond those traditional deployments, clustered deployments are also available to support scalability. Clusters can be handled using load balancers, active/passive or active/active routes, and other means but those are very hard to manage as require a lot of configuration.

To solve this problem, Camel offers the possibility to be used with Docker and Kubernetes not only using images but offering Apache Camel K[33], a subproject of Apache Camel, which is a lightweight integration framework that runs natively on Kubernetes and deploys integration code using serverless and microservice architectures in the cloud. Users of Camel K can run code written in Camel DSL using Kubernetes or OpenShift. It is important to mention Camel K components are called Kamelets but are different from Camel Components.

Another important subproject is Camel Kafka Connector, which allows using all Camel Components as extensions of Kafka, expanding new integration to Kafka Connect to the Kafka messaging bus.

[32] Camel in Action 2nd ed. ISBN 1617292931. Claus Ibsen, Jonathan Anstey. Chapter 16.1

[33] https://camel.apache.org/camel-k/1.8.x/

3.5.1.1 Usage in ODIN

Apache Camel has Components[34] for several systems that are of interest, to name a few, the following table is attached. The list contains most important components for WP4 and ODIN.

Those components would cover the Transport Services for several integrations to IoT, Data, AI and Robots, plus transformation message tasks.

Component	Description	Reason
REST	Expose REST services or call external REST services.	To connect to systems with REST interface
XSLT	Transforms XML payload using an XSLT template.	Process messages in XML format
ActiveMQ	Send messages to (or consume from) Apache ActiveMQ.	Integration with messaging bus
AMQP	Messaging with AMQP protocol using Apache QPid Client.	Integration with messaging bus
Async HTTP Client	Call external HTTP services using Async or Websocket	Integration with systems
AWS, Azure, Google	Integration with several AWS, Azure, Google Cloud services	Hybrid cloud platform integration
CoAP	Send and receive messages to/from COAP capable devices.	IoT device integration
MQTT	Send and receive messages to/from MQTT brokers.	IoT & Robot device integration
Debezium	Capture changes from several database types (SQL, NOSQL)	Data integration
Deep Java Library	Infer Deep Learning models from message exchanges data using Deep Java Library (DJL).	AI integration
Docker	Manage Docker containers	Management integration
Elasticsearch	Send requests to ElasticSearch via REST API	Analytics and AI integration

[34] https://camel.apache.org/components/3.15.x/index.html

FHIR	Exchange information in the healthcare domain using the FHIR (Fast Healthcare Interoperability Resources) standard.	Health data integration
Huawei	Use Huawei Cloud	Hybrin Cloud integration
IEC 60870	Client and server SCADA communication	IoT and M2M control
Ignite	Ignite operations management, execution and control	Analytics
IOTA	Blockchain transaction on IOTA	Distributed ledger technologies on IoT
Web3J	Blockchain transaction on Ethereum	DLT
JDBC	Database access using JDBC	Data integration
JPA	Store and retrieve objects from SQL databases	Data integration
JSON	Several JSON components to handle JSON data	Data transformation
Kafka	Send and receive messages to/from Kafka	Messaging bus integration
Kubernetes	Kubernetes management	Management
RabbitMQ	Send and receive messages form RabbitMQ	Messaging bus integration
MongoDB	Mongo data operations	Data integration
Modbus	Modbus bridge	Modbus integration
Nagios	Send passive checks to Nagios using JSendNSCA	Analytics and monitoring
OPC UA	Industrial machine communications	M2M and IoT integration
Paho	MQTT communications using Eclipse Paho	IoT integration
Rest OpenAPI	Configure REST producers based on an OpenAPI specification document delegating to a component implementing the RestProducerFactory interface.	System integration and API publishing
SFTP	SFT integration	Data integration
SLACK	Send and receive messages from SLACK	Data integration, monitoring, support
SQL	Perform SQL queries as a JDBC Stored Procedures using Spring JDB	Data integration

STOMP	Send and receive messages to/from STOMP (Simple Text Oriented Messaging Protocol) compliant message brokers	Data integration

Table 3 Camel Components

Beyond the Components, Camel also supports around 46 data types, related to the systems it supports integration such as JSON, FIHR, HL7 and other ones such as barcodes, gzip.

With those tools, the task to integrate any resource become easier with little customization or development.

In case an integration using Camel K with Kamelets is implemented, the Kamelet catalogue supports connectivity to the most important systems under ODINs' interest, such as Kafka, MQTT, FTP, SQL, NOSQL, FHIR, RabbitMQ and much more.

The advantage of Camel is the availability to connect to IoT, Robotics, Data and AI resources at the end.

3.5.2 EdgeX IoT Platform

EdgeX is an opensource platform, started by Dell and donated to the Linux Foundation with the mission to be a highly open, flexible and scalable software platform to interconnect devices on the IoT edge with enterprise application, with a vendor neutral mindset.

With those premises, EdgeX allows the interconnection of devices to the applications or cloud of a company or entity, acting as a bridge where operations and AI can be performed at the edge.

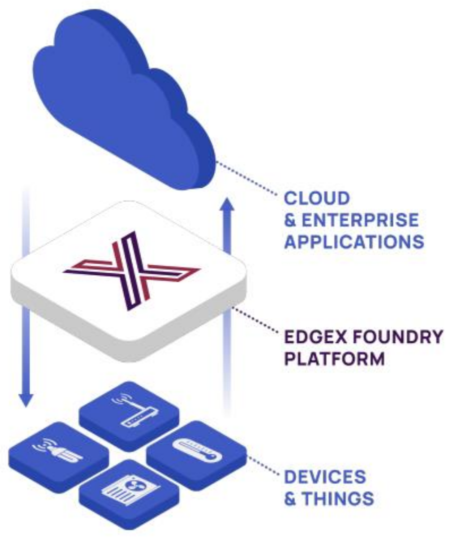


Figure 12 EdgeX Mission

This has several advantages as the data is distilled and processed at the edge, so actions can be taken near real time or real time, with low latency and valuable information can go up to the applications with less effort in terms of data transferred, saving time and costs.

The architecture of EdgeX is based on several layers and concepts:

Device Services: to connect to the devices at the edge. This layer supports proprietary and standard protocols such as REST, OPC-UA, MODBUS, MQTT, SNMP, BACNET, ZigBee, BLE and many more that can be developed. This layer offers the Device Service component that acts as a representation of the device in the platform, so it handles communication, setup and transforming data before transferring to the upper layers. In addition, offers device discovery capabilities. This layers also offers an SDK to integrate any device using C or Go Lang.

Core Services: is the base of the platform to perform data storing and transformations, communicate the upper-level services with the devices and hold the setup and meta data of the devices.

Supporting Services: here starts the layer providing operations and intelligence to the platform. Basic services are logging and alerts or notifications. Beyond those, edge analytics is provided, and task schedulers are available.

Exporting and Application services: this layer provides connectors and data transformation to integrate external systems, such as AWS, enterprise applications, Azure, Google Cloud, and other systems. It can define pipelines of data flows to process data form the Core services. An SDK is available so in case a system has no integration, so data can be retrieved from the messaging bus and filtered and transformed.

Security: This layer offers several services, such as Secret store, API gateway with authentication and authorization, control and manage secure access of users, services and devices, secure communication at core services.

Management: The platform offers management services to configure itself, and an agent to be integrated with externa applications or systems that want to control the platform. Currently this layer is optional as EdgeX supports management through Kubernetes to perform start/stop and other tasks.

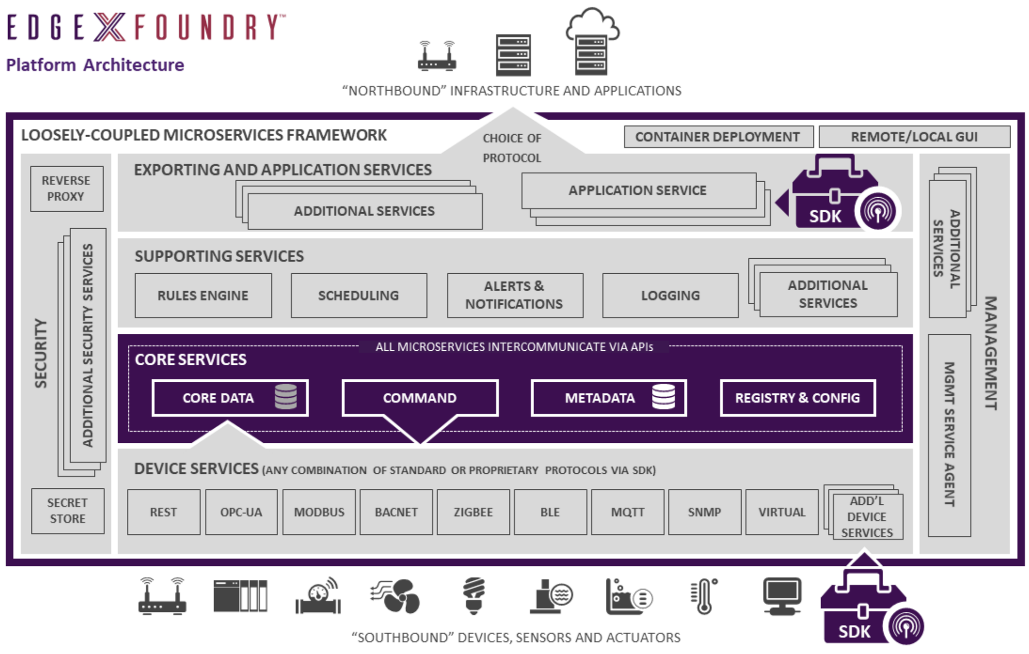


Figure 13 EdgeX Architecture

Security

This is a topic that EdgeX handles very well as not only secures the access to the platform from the applications and user level, but it also manages the access form the device point of view, which means that in case a rogue device is connected, EdgeX can minimize the risks disconnecting it.

The security layer provides lots of services that bring trust to any part of EdgeX. Every communication is secured with HTTPS by default and Json Web Tokens are used as the base to give access to the platform.

Regarding authorization, Authorization Control Lists are available for fine grained management of services and devices.

Metrics

EdgeX does not shine because of its metrics section. From its Management UI it can be checked CPU, RAM, network traffic and other system metrics from a concrete EdgeX instance. This topic must be enhanced to provide better understanding of the cluster. On the other hand, device management offers a nice understanding of the device status.

Deployment and Setup

EdgeX offers Docker[35,36] images for all of its components and Docker Composer files, but it is possible also to deploy it through Kubernetes[37]. This eases the tasks to start with EdgeX.

The setup can be performed using the Management layer agents, but the documentation points that those modules are going deprecated in the following months in favour of the tools provided by Kubernetes. Anyway, the management modules allow handling new devices, configurations, and large range of options.

3.5.3 Transport solution comparison

In the following table, a light comparison among the solutions is performed to discuss which one is better for implementing the messaging bus.

Attributes (1-none, 10-all)	Camel	EdgeX	Reason
Requirements fit	7	8	Camel can be used to connect lots of resources but lacks some functionalities that are present in an IoT platform as EdgeX. It gives more freedom of choice on how to develop the needed components. EdgeX is an IoT oriented platform, but to integrate devices modifications on the devices must be performed. EdgeX also fits very well as IoT integrator and could also work for robots but will not fit for some types of resources.
Ease of adoption	10	7	Camel is very easy to use, while EdgeX is harder to understand and less open.
Scalability	8	10	Camel can be used in clusters with configuration. EdgeX is prepared to work as Docker services that work together using Docker Compose or Kubernetes
Reliability	7	9	Camel depends on the component and the implementation adopted. EdgeX is more atomic in terms of reliability
Security (kerberos, oauth, etc)	8	10	Some Camel components allow to implement security using interceptors when used alone but is harder to keep security at device level. That security is built on top of the security provided by Camel

[35] https://docs.edgexfoundry.org/2.2/getting-started/Ch-GettingStartedDockerUsers/

[36] https://docs.edgexfoundry.org/2.2/getting-started/Ch-GettingStartedUsersNexus/

[37] https://www.lfedge.org/2020/06/25/edgex-foundry-kubernetes-installation/

			context. EdgeX has security very well covered at device level also.
Data integration	10	10	Both are integration frameworks
Support	10	6	Camel has lots of resources. EdgeX has less resources[38]
Deployment facilities	9	10	Both allow quite complete options to deploy and setup them.
Size of project to be used	8	9	Apache Camel may depend its suitableness on the implementation adopted, but it has to take into account that the module for translation is something small to mid-size, as it is going to work on each hospital. EdgeX is expected to work as 1 deployment per location, so it also has a nice fit on the size required.
Cost	10	9	Both opensource, but EdgeX may have more cost in terms of adoption
Number of projects using it	-	-	No concrete data found. Camel is used by big companies, hospitals and other entities to perform integration, while an example of EdgeX users is Dell, Accenture, Intel, Wipo[39]
Number languages	7	2	Camel supports more languages but does not mean they serve to do everything. Main language is Java. EdgeX support C and Go Lang.
Monitoring	10	7	As stated EdgeX has fewer monitoring tools.
Hard dependencies on other projects or solutions			Not known
Innovation impact	6	8	EdgeX is newer and it has a promising future in the IoT domain.
AI capabilities	0	8	EdgeX includes rules and small AI decision tools
Storage	0	0	Both can forward data to where it needs to be stored

[38] https://medium.com/nerd-for-tech/using-edgex-as-an-iot-middleware-6074288ca6dd

[39] https://www.edgexfoundry.org/why_edgex/why-edgex/

multi-tenant	5	4	EdgeX is supposed to have an instance per location. Camel must adopt communication using JMS or other channels to interconnect different CamelContext.
Real time	8	8	Both expose cases to use them in RT.[40],
pub/sub	10	10	Both support pub/sub messaging.
multiprotocol support	5	0	For Camel it depends on the upper layers.

Other solutions that could be used could be Eclipse Kura and OpenRemote, but both are aligned with EdgeX platform approach and need to modify the firmware of devices or at least the gateways.

3.5.4 Features Implemented in the First Version

For the first version of the Transport Protocol Services the decisions to be performed are:

- Select the right implementation strategy (platform vs components)
- Define the initial protocols to be supported
- Define the message format to be used as entry point to the platform
- Define the topics and the topic strategy to be used at resource level
- Define the input/output flows that resources need to fill the use cases

[40] https://dzone.com/articles/real-time-data-batching-with-apache-camel

4 Measurement Collection Software Components

4.1 Requirements Review

Gathering metrics, monitoring components, and configuring alerts is a fundamental piece for setting up and overseeing a service-based system. Having the option to determine what's going on inside a framework, what assets need consideration, and what is causing a stoppage or blackout is necessary. While planning and monitoring can be a challenge, including it from the beginning into the service infrastructure is an important added value that assists the teams with focusing on their work, delegating the obligation of oversight to an automated system. Coming from the requirements we have identified 3 main use cases related to the measurement collection software:

1. Monitoring system performance.
2. Monitoring activities and tracking of services.
3. Monitoring of RUC performance and KPIs.

4.2 Architecture Review

Every architecture for measurement collection can be summarized with the triad:

- Metrics
- Monitoring
- Alerting

Metrics represent the raw measurements of resource usage or behaviour (the “busyness” of a component) that can be observed and collected throughout a system. These might be low-level usage of resources provided by the operating system, or they can be higher-level types of data related to the specific functionality of a service, like requests served per second from a service endpoint. One special type of metric is system logs, which even though they usually are text messages with some metadata (like a time stamp), are fundamental for auditing any problem, and can potentially be post-processed as other sources for system performance metrics.

While metrics represent the data within a system, monitoring is the process of collecting, aggregating, and analysing those values to improve awareness of your components’ characteristics and behaviour. The monitoring system is responsible for storage, aggregation, visualization, and initiating automated responses when the data values meet specific requirements. In general, the difference between metrics and monitoring is equal to the difference between data and information. Data is composed of raw, unprocessed facts, while information is produced by analysing and organizing data to build contexts with added value. Monitoring takes metrics data, aggregates it, and presents it to humans to extract insights from the collection of individual pieces.

Alerting is the component of a monitoring system that performs actions based on changes in metric values. Alerts definitions are composed of metrics-based conditions, and actions to be performed when the values of the metrics don't match the acceptable conditions. While monitoring systems are incredibly useful for active interpretation and investigation, one of the primary benefits of an alerting system is letting administrators disengage from the system. Alerts allow defining situations that make sense of automated and active management of the system while relying on the passive monitoring of the software to watch for changing conditions.

The ODIN platform could offer an interesting service for deployers, that is the capability to automatically sharing monitoring and alerts when soliciting support. The platform could include a component to manage support provided by ODIN platform owners (T3.4 within the project), this component would allow system administrators to communicate directly with support providers and attach, annotate, or reference, monitoring information and alerts for quicker resolution of issues. Another option for the platform, like many commercially available systems, is to share anonymous performance metrics with the platform owner in order to improve the system overall.

4.3 Applicable Technology

4.3.1 Prometheus

Prometheus[41] is open-source computer monitoring and alerting software. It records metrics in real time in a time series database (with high capture capacity) based on the content of the entry point exposed using the HTTP protocol. These metrics can then be queried using a simple query language (see PromQL below) and can also be used to generate alerts. The project is written in Go[42] and is available under the Apache 2 license. The source code is available on GitHub and is a project managed by the Cloud Native Computing Foundation[43] along with other projects such as Kubernetes and Envoy.

Prometheus has been developed at SoundCloud since 2012, when the company realized that its monitoring solutions (StatsD and Graphite) weren't right for their needs. Prometheus was therefore designed to address these problems: having a multidimensional database, an easy-to-use tool, a simple and scalable collection mechanism, and a powerful query language, all in one tool. The source code of the project was released under a free license from the very beginning.

In May 2016, Prometheus was the second project incubated within the Cloud Native Computing Foundation after Kubernetes. The tool is in use by many companies, including Digital Ocean, Ericsson, CoreOS, Weaveworks, Red Hat and Google. Version 2 was released in November 2017. In August 2018, the Cloud Native Computing Foundation announced that Prometheus could be used in production.

A typical Prometheus installation includes several building blocks:

- Several agents (exporters) that usually run on the systems to be monitored and will expose the monitoring metrics.
- Prometheus for centralization and archiving of metrics.
- Alertmanager[44] that triggers the issuance of alerts based on rules.
- Grafana[45] for the return of metrics in the form of a dashboard.
- PromQL is the query language used to build dashboards and create alerts.

[41] Prometheus, https://prometheus.io/ , Last Access Jan. 2022

[42] Go Language, https://go.dev, Last access Jan. 2022

[43] Cloud Native Computing Foundation, https://www.cncf.io , Last Access Jan. 2022

[44] Alertmanager, https://prometheus.io/docs/alerting/latest/alertmanager/ , Last Access Jan. 2022

[45] Grafana, https://prometheus.io/docs/visualization/grafana/ , Last Access Jan. 2022

Prometheus data is stored in the form of metrics. Each metric has a name and a series of labels in the form of a key value pair. Each metric can be selected based on these labels. These labels include information about the source of the metric (agent, server address) as well as a variety of application-specific information (HTTP code, request method), endpoints, etc. The ability to specify an arbitrary list of labels and query them in real time explains why the Prometheus data model is called multidimensional.

Prometheus stores data locally on disk. This technique optimizes quick archiving and retrieval. Prometheus can also store metrics on remote servers (especially for long-term archiving).

Prometheus collects data in the form of a time series. Time series are actively retrieved - the Prometheus server queries a list of data sources (the exporters) with a specific polling frequency. These collection points serve as data sources for Prometheus. The server also has mechanisms to automatically detect the resources to monitor.

Prometheus has its own query language PromQL (Prometheus Query Language)[46]. This language allows users to select and aggregate the metrics stored in the database. It is particularly suited to operations with a time series database by providing numerous specific features for manipulating time (time offset, average, maximum, etc.). Prometheus supports four types of metrics:

- Indicator (absolute temperature, amount of disk space consumed)
- Counter (number of requests since the start of a program)
- Histogram (sampling a number of requests in multiple containers to calculate quantiles)
- Summary (relatively similar to the notion of histogram with additional notions).

The configuration of alerts is configured by Prometheus using a condition based on an expression in PromQL format and a time duration that allows you to characterize the time required to trigger an alert. When alerts are triggered, they are passed on to the Alertmanager. The latter is responsible for carrying out a certain number of aggregation, deactivation and time-out operations of these alerts before transmitting them by different means (email, Slack notification or SMS).

Prometheus is not designed to provide dashboard feedback although it does have a workaround for doing so. It is a good idea to use a tool like Grafana even if this solution has the drawback of making the installation of the monitoring system more complex.

Prometheus uses so-called white box surveillance. Applications are encouraged to expose their internal metrics (using an exporter) so that Prometheus can collect them on a regular basis. In case the application (or component) cannot do it directly (database, monitoring server), there are many exporters or agents ready to use to fulfil this role. Some exporters also allow you to manage communication with some monitoring tools (Graphite, StatsD, etc.) to simplify switching to Prometheus during migration.

Prometheus focuses on platform availability and core operations. Metrics are typically archived for a few weeks. For long-term storage, it is recommended to turn to more suitable storage solutions. Prometheus' exporter metrics display format has been standardized with the name OpenMetrics[47] so that it can be reused elsewhere. Some products have adopted this format, such as InfluxDB, Google Cloud Platform, and DataDog.

[46] PromQL, https://prometheus.io/docs/prometheus/latest/querying/basics/ , Last Access Jan. 2022

[47] OpenMetrics, https://openmetrics.io , Last Access Jan. 2022

4.3.1.1 Usage in ODIN

Prometheus in ODIN can be used for "Monitoring system performance" as well as "Monitoring of RUC performance and KPIs", as well as the basis for the optional services for monitoring sharing with technical support, and Anonymous performance sharing.

One of the advantages of using this software is that the technologies proposed for other components, presented in this deliverable and deliverable D4.5, are already aligned with Prometheus.

4.3.1.2 Available Open-Source Projects

The open-source project of Prometheus monitoring system and time series database is available on Github[48].

4.3.2 ELKStack

Elasticsearch[49] is a distributed, open-source search and analysis engine for all types of data, including textual, numeric, geospatial, structured, and unstructured. It is based on Apache Lucene[50] and was first released in 2010 by Elasticsearch N.V. (now known as Elastic). Known for its simple REST APIs, distributed nature, speed, and scalability, Elasticsearch is the core component of Elastic Stack, an open-source toolset for capturing, enriching, archiving, analysing and data visualization. Commonly referred to as the ELK Stack (after Elasticsearch, Logstash and Kibana), Elastic Stack now includes a rich collection of supplements known as Beats for submitting data to Elasticsearch.

In recent years the world of Elasticsearch has expanded considerably. Born as a NoSQL database to support text search through the Apache Lucene index, the project saw the development of other open-source products: LogStash[51], Kibana[52] and Beats[53].

Logstash is a log aggregator that collects data from various input sources (apps, databases, servers, etc.), performs various transformations and data clean up, and finally sends the resulting data to various supported output destinations including Elasticsearch. Kibana, on the other hand, is a visualization tool that works on top of Elasticsearch, giving users the ability to analyse and visualize data. Finally, Beats are software packages (agents) that are installed on hosts to collect different types of data to forward on the stack.

[48] Prometheus GitHub project, https://github.com/prometheus/prometheus , Last Access Feb. 2022

[49] ElasticSearch, https://www.elastic.co/ , Last Access Jan. 2022

[50] Apache Lucene, https://lucene.apache.org , Last Access Jan. 2022

[51] LogStash, https://www.elastic.co/logstash/ , Last Access Jan. 2022

[52] Kibana, https://www.elastic.co/kibana/ , Last Access Jan. 2022

[53] Beats, https://www.elastic.co/beats/ , Last Access Jan. 2022

All of these components are most commonly used together for monitoring, troubleshooting and security of IT environments (although there are many other use cases for ELK Stack such as business intelligence and web analytics). Beats and Logstash take care of data collection and processing, Elasticsearch indexes and stores the data and Kibana provides a user interface to query and view the data.

As previously mentioned, ELK stack represents an open-source solution for the management and analysis of large amounts of data including logs. Monitoring modern applications and the IT infrastructures on which they are deployed requires a log management and analysis solution that will overcome the challenge of monitoring highly distributed, dynamic and noisy environments. ELK Stack allows you to perform these operations by providing users with a powerful platform that collects and processes data from multiple data sources and stores them in a centralized data archive. In addition, it can scale with data growth as well as providing a set of tools for data analysis.

In order to ensure high availability, high reliability and security of applications, ELK Stack rely on the different types of data generated by the applications themselves and by the infrastructure that hosts them. In this way, solutions can be studied to improve the software, the architecture and / or promptly intervene in the event of a Deny Of Service (DOS).

Logs have always existed and so have the different tools available to analyse them. What has changed, however, is the underlying architecture of the environments that generate these logs. The architecture has evolved into microservices, containers (for example Docker, Kubernetes) and orchestration infrastructures distributed on the cloud or in hybrid environments. Furthermore, the volume of data generated by these environments is constantly growing which is a challenge in itself. Centralized log management and analysis solutions such as ELK Stack, allow to have an overview of the information captured and, consequently, ensure that the apps are reliable and performing at all times.

ELK Stack log management and analytics solutions include the following key capabilities:

- Aggregation: the ability to collect and send logs from multiple data sources
- Processing: the ability to transform log messages into meaningful data for easier analysis
- Storage: the ability to store data for extended periods of time to allow for monitoring, trend analysis and security use cases
- Analysis: the ability to dissect data by querying it and creating views and dashboards on it.

The various components of ELK Stack have been designed to interact with each other without too many configurations. However, the way you design the stack can differ greatly depending on your environment and use case. Elasticsearch, of course, is not only used for log analysis. Its architecture can also be used for the analysis of other types of data, such as those deriving from the Internet Of Things (IOT) or from e-commerce transactions.

In fact, thanks to LogStash it is possible to define custom transformers for any type of data. Apache Lucene's text indexing capabilities allow you to create unstructured data search engines. Finally, with the different visualization tools offered by Kibana you can create dashboards for different analysis purposes. The management of accesses, user roles and workspaces ensure the creation of environments oriented to the needs of the various professional figures of a company who must analyse the data.

4.3.2.1 Usage in ODIN

ELKStack in ODIN can be used for “Monitoring activities and tracking of services” as well as “Monitoring of RUC performance and KPIs”, as well as part of the optional services for monitoring sharing with technical support.

4.3.2.2 Available Open-Source Projects

Elastic, however, recently changed its licensing scheme for Elasticsearch and Kibana, moving away from the open-source Apache 2.0 license to the more restrictive Server-Side Public License (SSPL) and Elastic License. The Open-Source Initiative has publicly stated that the SSPL is not an open-source licence. For a true open-source alternative to the Elastic distribution, organizations will need to choose the Open Distro for Elasticsearch[54] instead of one of Elastic’s offerings. The Open Distro for Elasticsearch will be renamed as the community managing the project manages its own fork for the Elasticsearch and Kibana codebases.

4.4 Features Implemented in the First Version

In the first version, we expect to cover the use case of “Monitoring system performance” with one of the before mentioned technologies.

[54] Open Distro for Elastic Search, https://opendistro.github.io/for-elasticsearch/, Last access Feb. 2022

5 Conclusions and Next Steps

For the following period, the three main components described in this deliverable will undergo a deeper study and research so that decisions regarding the technologies to use as well as the final open-source projects to be implemented, are made and the initial tests can be carried out.

These decisions involve not just technological partners but also other stakeholders in the project, not just the pilots, but also business leaders and the open innovation.

The decisions include:

- Technology or technologies to be used for the Resource Descriptor.
- Technology to be used as the Message Bus.
- Technology to be used for Transport Services.
- Technology to be used for Monitoring System Performance.

All this will be reported in deliverable *D4.3 - Implementation of Local CPS-IoT RSM Features v2*.