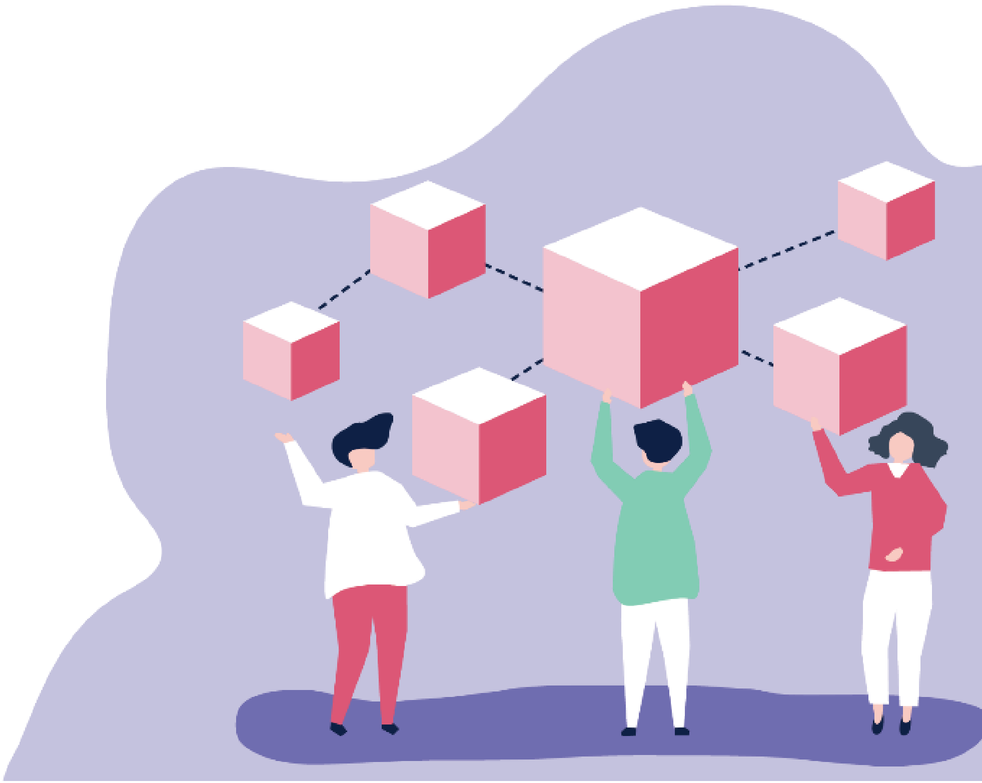
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 857223

GATE KEEPER

# D4.4 Data federation and Integration and Health Sematic Data Lake

| | | | |
|---|---|---|---|
| Deliverable No. | D4.4 | Due Date | 31/12/2020 |
| Description | Report describing the Data Federation & Integration framework allowing the integration and harmonization of data coming from heterogeneous data sources. | | |
| Type | Report | Dissemination Level | PU |
| Work Package No. | WP4 | Work Package Title | GATEKEEPER Things Management Infrastructure & Development |
| Version | 1.0 | Status | Final |



## Authors

| Name and surname | Partner name | e-mail |
|---|---|---|
| Domenico Martino | ENG | domenico.martino@eng.it |
| Paolo Zampognaro | ENG | paolo.zampognaro@eng.it |
| Vincenzo Falanga | ENG | vincenzo.falanga@eng.it |
| Carlo Allocca | SAM | c.allocca@samsung.com |
| Sabino Minervini | HPE | sabino.minervini@hpe.com |
| Franco Mercalli | MME | f.mercalli@multimedengineers.com |
| Robin Kleiner | M+ | robin.kleiner@medisante-group.com |
| Imad Ahmed | M+ | imad.ahmed@medisante-group.com |
| Alessio Antonini | OU | alessio.antonini@open.ac.uk |
| Johan Kling | FUN | johan.kling@funka.com |
| Daniel Rodriguez | S4C | daniel.rodriguez@sense4care.com |

## History

| Date | Version | Change |
|---|---|---|
| 18/06/2020 | 0.1 | Initial Toc |
| 02/09/2020 | 0.2 | Updated ToC |
| 05/10/2020 | 0.3 | Added deadline and edited state of the art |
| 11/11/2020 | 0.4 | Edited section Data Federation & Integration and assigned section to the contributors |
| 27/11/2020 | 0.5 | - Integrated revisions of S4C<br>- Edited sections “Data Integration Process and Requirements, “Data Federation & Integration Design”, “Data federation and Integration V1: deployment environments”, “Source Code”, “Appendix A” and “Introduction” |

| | | |
|---|---|---|
| 13/12/2020 | 0.6 | Edited and completed chapters "Data federation and integration v1: overview, requirements, design", "data federation and integration v1:  implementation details", "data federation and integration v1: deployment environments" and “Conclusion” |
| 07/01/2021 | 0.7 | Addressed internal review and other minor fixes |
| 11/01/2021 | 1.0 | Addressed quality check review and produced the Final Version |

## Key data

| | |
|---|---|
| **Keywords** | Data Federation, FHIR, RDF |
| **Lead Editor** | ENG |
| **Internal Reviewer(s)** | W3C, UPM |

## Abstract

This deliverable reports on the progress of T4.4 aiming to design a framework, named Data Federation & Integration (DFI), conceived (i) to collect data coming from heterogeneous data sources (EHR and IOT), (ii) to harmonize the data against specific semantic models and, finally, (iii) to persist the data in pilot specific cloud nodes. The harmonization step will proceed exploiting the GK-FHIR profile as for guidelines and indications provided by T3.5

The early version of the design of the Data Federation & Integration framework defines specific southbound and northbound APIs to share and retrieve persisted data that will be used by other work packages (e.g., WP5, WP6).

This deliverable is a live document that will be updated in a second version by M27.

## Statement of originality

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

# Table of contents

# List of tables

# List of figures

# Introduction

This document provides the design details of the ***connectivity layer*** bridging IT systems and the GK WoT data space on a pilot-level, thus enabling data access to final applications or to further predictive analytics and data mining core services carried out in WP5. It offers the needed mechanisms to harmonize data coming from **heterogeneous data sources** registered in the platform, including personal clinical data source (EHR/EMR), social care data sources, wearable device data sources, home-based sensor data and activity sensor data, thus producing a Health Semantic Data Lake (HDSL).

According to the development approach, adopted in the whole project, the design of the **Data federation & Integration** framework is defined using an incremental approach. Therefore, the main goal of the design activity reported in this document is to satisfy the requirements scheduled for the first period of the project (M12) while a next version of the document will be released (M24) taking into account feedbacks coming from (i) initial pilot's roll out experiences (ii) issues arising from integration step with the other GK platform components.

In details the document structure is the following:

**Section I** provides a global overview of the standards adopted in the definition of the framework architecture fully described in the next section. In particular, a general picture of the HL7 FHIR standard is reported since it will be the main *semantic model* adopted for the GK data representation. Also, an overview of the RDF framework is provided, since it will be used as for data exposure through the adoption of a RDF based graph database (e.g. RDF4J). RDF is a foundation for processing metadata; it provides interoperability between applications that exchange machine-understandable information on the Web. RDF emphasizes facilities to enable automated processing of Web resources. RDF can be used in a variety of application areas. Finally, the RML language is briefly described since the Data Federation & Integration architecture enables the definition of new "data converter" not only programmatically but also in a *declarative way* by defining *mapping rules* through the RML language.

**Section II** describes the position of Data Federation & Integration into Gatekeeper architecture showing the several components with which it interacts. It offers an easy modality to enable external heterogeneous data sources to send their data by harmonizing such data against common semantic models selected by the project (e.g. HL7 FHIR) and allow to other thing to access such data. It also provides a general overview of the problems with data integration process together with several approaches present in the literature and the perspectives. It lists also all requirements collected during the phc made with the leader of each pilot, such requirements have been the starting point for design the architecture and the interfaces of the Data Federation & Integration. It supports two types of approaches to convert data coming from pilot application into the adopter semantic modes, the first one is a declarative approach where it is used a declarative language that allows to specify the conversion rules that are executed by a specific internal engine, while the second one is a programmatic approach where some JAVA interfaces are provided in order to load a specific converter. Finally, it is described in which way the interaction and integration with Medisantè IoT Connecter and Samsung Health gateway.

**Section III** describes the internal architecture of the Data Federation & Integration. It consists of four main components: gk-integration-engine, gk-fhir-server, gk-rdf4j and keycloak. gk-integration-engine provides the southbound APIs that can be invoked by pilots' applications to send their data by harmonizing such data against common semantic models selected by the project. gk-integration-engine and gk-rdf4j provide the north bounds APIs that allows the external applications to retrieve persisted data in FHIR and RDF format. The keycloak component is used to implement the security level of the application in order to perform integration tests with the other components. These interactions have been exploited for testing purpose only. In production all calls to southbound and northbound APIs are expected to be trusted since the interaction with the Data Federation & Integration is mediated by GTA. For all components consisting of DFI a docker image has been created and added in a docker-compose file so that all the containers can be started with one single command.

**Section IV** provides the migration of DFI to Kubernetes cluster that will be deployed on the HPE (task 4.1) infrastructure. The different possible deployment scenarios are described. The cluster consists of several PODs that interaction with Service. The contact point among Kubernetes and external applications are Ingresses.

**Section V** provides the conclusion the document.

**Appendix A** describes the instruction to build a new Java converter.

# 1 Overview on adopted standards

One of the biggest issues to achieve full semantic interoperability in healthcare in which all the systems seamlessly communicate with each other is still pending. In an ideal environment, the patient would have all the clinical information coming from heterogeneous providers integrated and available in a common format. HL7 Fast Healthcare Interoperability Resources (FHIR) is rapidly becoming a major standard for the exchange of electronic healthcare information. FHIR defines a collection of "resources" to represent different information types and specifies how these resources are to be exchanged using XML, JSON and, as of the latest release, RDF. The FHIR specification also uses links to SNOMED CT and other ontologies as an integral component of the representation of clinical data. The combination of a standardized set of FHIR RDF tags with embedded ontology references provide a number of interesting new possibilities for classification and categorization of clinical data, including recognizing prescriptions that contain particular drug categories, procedures that use a specific technique or approach, diagnoses of general disease categories (e.g. cancer, diabetes), etc. Moreover, the vast amount of data being produced everyday requires semantics to provide meanings to the data. The recent advances of Semantic Web technologies provide us with standard data formats (i.e., Resource Description Framework – RDF), vocabularies (e.g., PROV-O for provenance) and tools (e.g., SPARQL for querying data) to add semantics to data and structuring the metadata. This use of semantics would in turn allows advances data analysis and processing to the data and its metadata. Despite the significant number of existing tools, incorporating data from multiple sources and different formats into the Linked Open Data cloud remains complicated. No mapping formalisation exists to define how to map such heterogeneous sources into RDF in an integrated and interoperable fashion. One of the solutions is the RML mapping language, a generic language based on an extension over r2rml, the w3c standard for mapping relational databases into rdf. Broadening r2rml's scope, the language becomes source-agnostic and extensible, while facilitating the definition of mappings of multiple heterogeneous sources. This leads to higher integrity within datasets and richer interlinking among resources.

## 1.1 Fast Health Interoperability Resources (FHIR)

FHIR [1] (Fast Healthcare Interoperability Resource) is a standard born in the health context that allows the information exchange among health system. From the acronym it is possible to understand its main characteristics which are:

- **Fast:** it is easy to learn from the developers since it provides a big set of examples together with several libraries and reference implementations supporting the developer.
- **Healthcare:** it can be applied in the healthcare context.
- **Interoperability:** interoperable because it allows the communication and interaction among heterogeneous systems enabling information exchange.
- **Resource:** this standard is structured in a modular way, where each module is named Resource.

The introduction of the dematerialization involved that health information is managed exclusively electronically; the quantity of data that are created for a patient, for example during a clinical encounter or during the execution of some laboratory tests, should answer to three main requirements:

- **Availability**: data should be always available.
- **Detectability**: it should always be possible to access data.
- **Understanding**: data must always be consistent.

All data produces, in addition having the aforementioned characteristics, should be used to perform analysis for the clinical decision support and the elaboration required by a computer. These needs make it necessary for data to be structured and standardized.

FHIR allows to fully manage the persistence and data transport processes. The goal of this standard is to simplify the health management processes, without losing the integrity of the information.

The main element of FHIR standard is represented by the concept of Resource. Whatever information or data that has to be represented and moved should be modelled to a specific resource plus a set of characteristics such:

- A way to define, represent and rebuild them.
- A set of metadata.
- A section that can be read by a user.

FHIR philosophy is to define a set of base resources that allow, alone or in an aggregate way, to satisfy the most use cases in healthcare. The model proposed by FHIR follows a composition approach where it is possible to build health documents or more in general to build data structure combining appropriately resources provided by the standard. This information can be used individually or connected among them in order to guarantee the construction of a comprehensive health system.

FHIR defines two specific resources to describe all the resources that are defined and use in a specific system:

- **CapabilityStatement** that describes the API and operations that are exposed by the system, used to know which are the functionalities and technical specification to use a specific software implementation.
- **StructureDefinition** that defines a set of useful rules to fix optional elements, cardinalities, terminologies, primate types and extension used in the software implementation.

Conceptually, it is possible to state that the FHIR specification can be divided into three parts:

- **Documentation** which describes the standard and how resources are structured providing to the final user a base knowledge of the standard.
- **Implementation** in which are provided guidelines about how resources should be used in the exchange message that is adopted by REST architecture.
- **List of resources** defined by the standard with the relative descriptions.

### 1.1.1 Reference

FHIR standard consists of a set of resources that can be linked among them enabling advantages to store, organize and transport data the in healthcare system. The base element of the standard that realizes the interconnection among resources is named "Reference"; in FHIR the most part of the elements defined inside include a Reference to another resource.

Thanks to Reference is possible to link several resources providing a network of information. The standard defines two kinds of reference:

- **Internal references**: the links are build including "physically" the referred resource in the source one.
- **External references**: linked to resource not internal to the source one.

The aforementioned references are always defined and represented in a unidirectional way from source resources to target ones since they have URL that can be both relative and absolute. The inverse relationship, from the target resource to the source one, exists only logically and it is not explicitly represented in the resource.

Since resources are processes in an independent way from each other, relationships are not transitive; for example: if a resource "Condition" contains as subject a reference to a specific "Patient" and in the field reason the reference to a "Procedure" resource, it does not exist any automatic rule that links the same "Patient" of "Condition" to the resource "Procedure". In general, the subject of the resource Procedure must be fixed from the same "Procedure" resource in order to guarantee logic coherence among information.

In a Resource, reference is represented by two elements **reference** and **display** where the last one represents a textual description of the element to which they can refer to; summarize reference is the key element of the FHIR standard where resources are linked by means their URL.

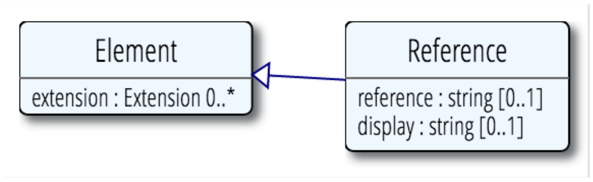


Figure 1 Reference model defined by FHIR

In the described model (showed in Figure 1), the attribute reference contains a URL that

### 1.1.2 Extension

FHIR resources are generic concepts that can be used in several countries, contexts and application not necessarily made for purely health purposes. As this freedom could lead inappropriate results in the health domain, FHIR standard also defines the mechanism of the extensions that can be applied to the resources giving the possibility to define constraints on them (changeability, cardinality and so on).

To avoid that resources defined by the standard grow exponentially without any control, maybe even in an unreasonable way, FHIR establishes a base rule: in a resource can be included only data that are used by a multitude of applications. This does not mean that data must always exist, for example, in some systems, a death date for a patient can be set of course this information is not filled for many patients. Another example is the gathering of information deemed not fundamental such as the colour of the hairs of a person; in the latter case, as this information is not important, FHIR does not include an element in the "Patient" resource to represent this characteristic but it can be acquired by means the creation of an extension for the base resource (in this case "Patient") defined by the standard.

To avoid that the number of the resources grow without any control and in an unreasonable way in order to cover as many scenarios possible, FHIR designed and defined a set of base resources that are part of the specification and can be used in various areas.

### 1.1.3 FHIR Resource

The main element of FHIR standard is the concept of a resource. We must think of the resource as a collector of different types of clinical and administrative information that can be collected and shared; FHIR defines a generic shape for each clinical information, allowing the resources to have a wide field of use.

The proposed data structure is a repository, usually accessible remotely, containing the instances of the several resources which describe information related to the patient such as for example demographic data, health conditions such as diagnosis, clinical procedures and care plans but it can store administrative information, such as healthcare operator that works in the health structure, organization where she works and the physical location where she provides the service.

Some resources are structural components, most of them used to support the exchange of information, about the real capacity of the systems involved in the communication, define the codification and so on.

In FHIR each resource taken individually does not carry a high information content if not aggregated with other resources in order to compose a useful set of information.

In detail the elements that consist of a resource are:

- An URL that identifies a unique way in the space in which is defined and persisted.
- It is always traceable to a basic resource type.
- It contains a set of structured objects.
- It has an identifier that changes as the information it contains changes.
- It has multiple representations.

#### 1.1.3.1 Resource identifier

Each resource has a unique identifier representing the logic entity of the resource assigned by the server which takes care of its persistence. Each resource has always an identifier except when it should be created, in this case, server takes care to assigner the identifier.

This logic id is unique in the resource domain inside the same server and once it is assigned it cannot be modified, obviously, when it is copied to another server it may not keep the same identifier.

The absolute address of the resource is of type http (URL) consisted of:

- Server address to which it is persisted (named as baseUrl).

- Type of the traced resource.
- Id assigned by the server

*<base URL> / <resource type> / <id>*

A complete example is:

*http:// fhir.org/rest/Patient/123*

It is fair to point out that it is possible to retrieve or trace a resource via its URL only when the application has been built according to a REST architecture allowing the access to the server. Id and the name of the resource are case sensitive. An id will have:

- Always be represented in the same way, both within the resource and in its URL.
- Be a maximum of 64 characters.
- Contain any combination of upper- and lower-case letters, numbers, special characters such as "-" and ",".

#### 1.1.3.2 Business identifier

FHIR in the definition of the resource has included, in addition of the logical identifier and URL, also an element named **"identifier"**, that can contain many identifiers. In this way, if a resource were copied from one server and another, only the logical identifier and URL would change while the identifier will not be changed but a new one can be added at most.

This new element just introduced takes the name of **business identifier;** when there are different technologies and standards each representation inherits the business identifier regardless of the context in which the resource is used.

#### 1.1.3.3 Profile

The FHIR profile represents the rules to use a resource that must be respected when the object is instantiated and that must be acquired when the contents are processed.

The base FHIR specification describes a set of base resources, frameworks and APIs that are used in many different contexts in healthcare. However, there is wide variability between jurisdictions and across the healthcare ecosystem around practices, requirements, regulations, education and what actions are feasible and/or beneficial. For this reason, the FHIR specification is a "platform specification" - it creates a common platform or foundation on which a variety of different solutions are implemented. As a consequence, this specification usually requires further adaptation to particular contexts of use. Typically, these adaptations specify:

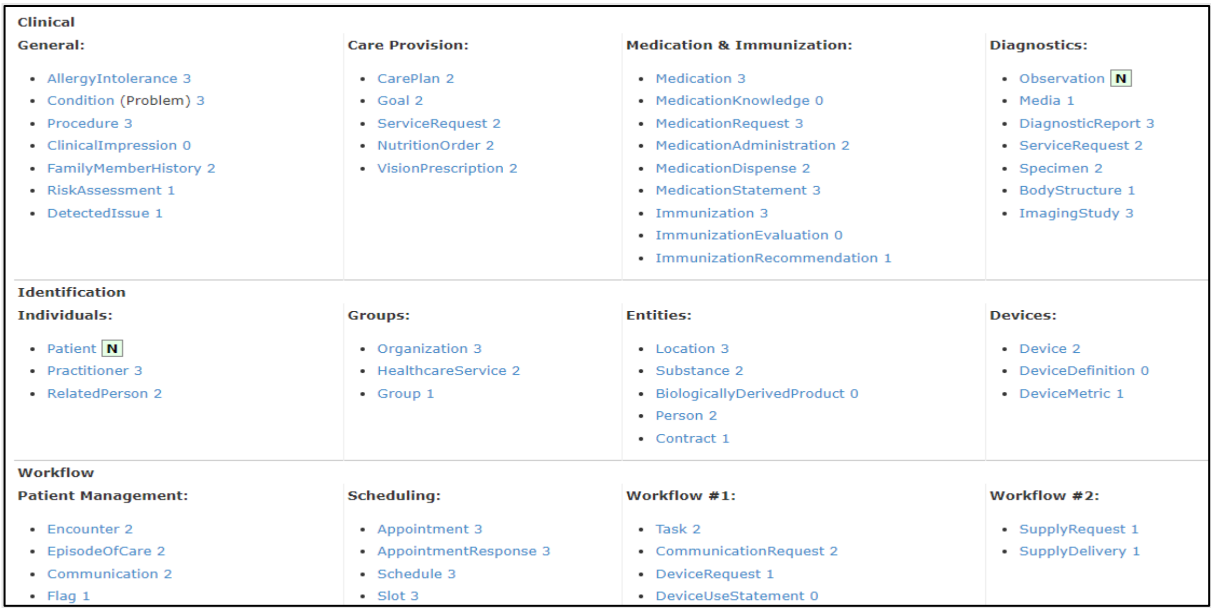
- Rules about which resource elements are or are not used, and what additional elements are added that are not part of the base specification.
- Rules about which API features are used, and how.
- Rules about which terminologies are used in particular elements.
- Descriptions of how the Resource elements and API feature map to local requirements and/or implementations.

#### 1.1.3.4 Resource classification

FHIR specification defines a set of resources and an infrastructure suitable for their management. Resources are divided into six distinct sections:

- **Clinical:** it includes resources used to represent information about the health state of the patient and her history from the medical point of view. Resource are further divided into four subsections: General, Care Provision, Medication & Immunization and Diagnostics.
- **Identification:** it includes actors involved in the care process. These resources are further divided into four subsections: Individuals, Group, Entity and Device.
- **Workflow:** it includes entities involved in the care process of the patient. These resources are further divided into four subsections: Patient Management, Scheduling, Workflow #1 and Workflow #2.
- **Financial:** it groups resources useful to manage specifications, develop and test phase about FHIR solutions. These resources are further divided into four subsections: Terminology, Content, Operations Control and Misc.
- **Infrastructure:** it groups all the resources that provide general functions and resources for the internal workings of the FHIR standard. Resource are further divided into four subsections: Terminology, Document & List, Structure and Exchange.

The following figure shows the whole list of the resources defined by the Standard for version 4.0.1.



**Clinical**

**General:**
- AllergyIntolerance 3
- Condition (Problem) 3
- Procedure 3
- ClinicalImpression 0
- FamilyMemberHistory 2
- RiskAssessment 1
- DetectedIssue 1

**Care Provision:**
- CarePlan 2
- Goal 2
- ServiceRequest 2
- NutritionOrder 2
- VisionPrescription 2

**Medication & Immunization:**
- Medication 3
- MedicationKnowledge 0
- MedicationRequest 3
- MedicationAdministration 2
- MedicationDispense 2
- MedicationStatement 3
- Immunization 3
- ImmunizationEvaluation 0
- ImmunizationRecommendation 1

**Diagnostics:**
- Observation N
- Media 1
- DiagnosticReport 3
- ServiceRequest 2
- Specimen 2
- BodyStructure 1
- ImagingStudy 3

**Identification**

**Individuals:**
- Patient N
- Practitioner 3
- RelatedPerson 2

**Groups:**
- Organization 3
- HealthcareService 2
- Group 1

**Entities:**
- Location 3
- Substance 2
- BiologicallyDerivedProduct 0
- Person 2
- Contract 1

**Devices:**
- Device 2
- DeviceDefinition 0
- DeviceMetric 1

**Workflow**

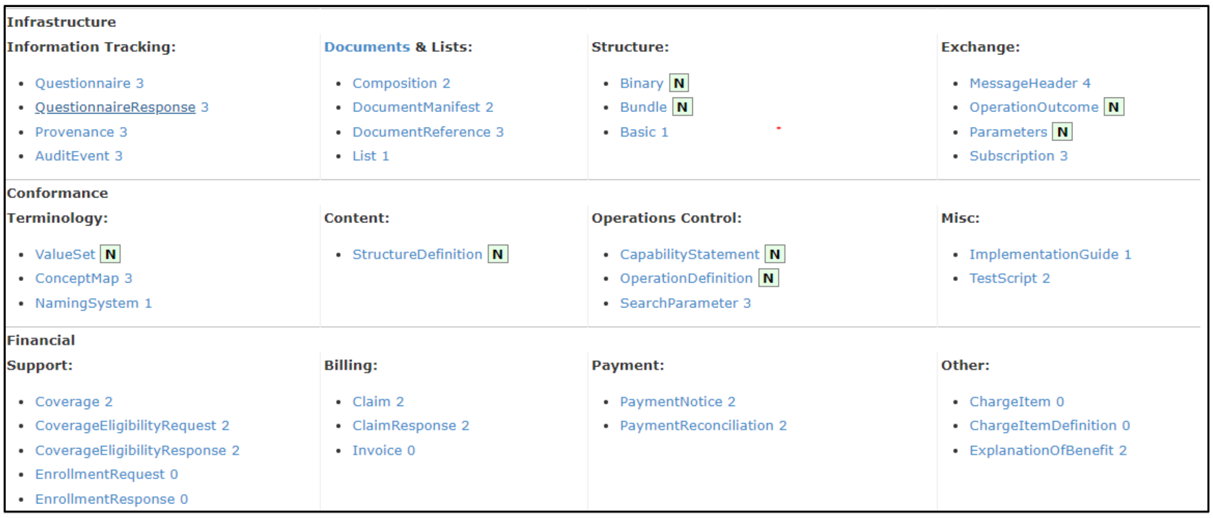
**Patient Management:**
- Encounter 2
- EpisodeOfCare 2
- Communication 2
- Flag 1

**Scheduling:**
- Appointment 3
- AppointmentResponse 3
- Schedule 3
- Slot 3

**Workflow #1:**
- Task 2
- CommunicationRequest 2
- DeviceRequest 1
- DeviceUseStatement 0

**Workflow #2:**
- SupplyRequest 1
- SupplyDelivery 1

Figure 2 List of FHIR Resources v4.0.1 (part 1 of 2)



**Infrastructure**

**Information Tracking:**

- Questionnaire 3
- QuestionnaireResponse 3
- Provenance 3
- AuditEvent 3

**Documents & Lists:**

- Composition 2
- DocumentManifest 2
- DocumentReference 3
- List 1

**Structure:**

- Binary N
- Bundle N
- Basic 1

**Exchange:**

- MessageHeader 4
- OperationOutcome N
- Parameters N
- Subscription 3

**Conformance**

**Terminology:**

- ValueSet N
- ConceptMap 3
- NamingSystem 1

**Content:**

- StructureDefinition N

**Operations Control:**

- CapabilityStatement N
- OperationDefinition N
- SearchParameter 3

**Misc:**

- ImplementationGuide 1
- TestScript 2

**Financial**

**Support:**

- Coverage 2
- CoverageEligibilityRequest 2
- CoverageEligibilityResponse 2
- EnrollmentRequest 0
- EnrollmentResponse 0

**Billing:**

- Claim 2
- ClaimResponse 2
- Invoice 0

**Payment:**

- PaymentNotice 2
- PaymentReconciliation 2

**Other:**

- ChargeItem 0
- ChargeItemDefinition 0
- ExplanationOfBenefit 2

Figure 3 List of FHIR Resources v4.0.1 (part 2 of 2)

### 1.1.3.5 Bundle

In the prevision section, it has been introduced the resource "Bundle" representing the container of the aggregate resources. The main task of this entity is to transport a set of resources following an explicit request to the server that manages them, more in general it is possible to identify different uses:

Carrying a set of resources matching criteria in response to an explicit request from a client.

Transport the history of all versions of a resource, in response to a client request.

Carrying a set of specific resources, in explicit response to a client request.

Group a set of independent resources to create a document that can be transmitted or stored.

Make a set of 'create', 'update' or 'delete' on a set of resources within individual operations or transactions.

Store a set of resources.

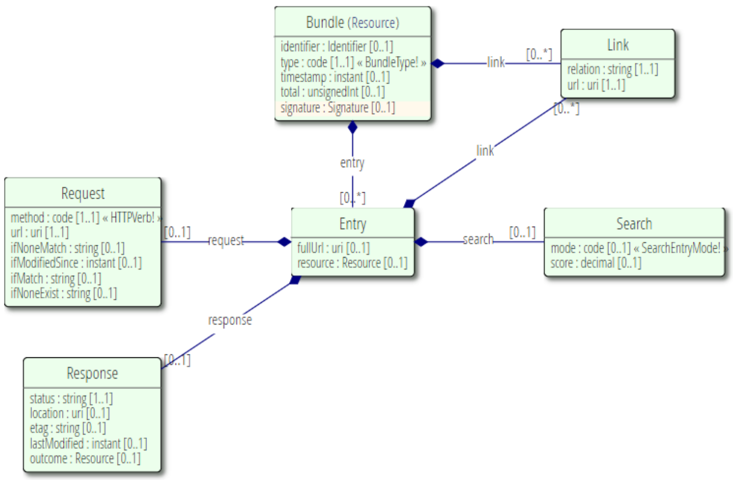

Bundle (Resource)
identifier : Identifier [0..1]
type : code [1..1] « BundleType! »
timestamp : instant [0..1]
total : unsignedInt [0..1]
signature : Signature [0..1]
link
[0..*]
Link
relation : string [1..1]
url : uri [1..1]
[0..*]
link
entry
[0..*]
Request
method : code [1..1] « HTTPVerb! »
url : uri [1..1]
ifNoneMatch : string [0..1]
ifModifiedSince : instant [0..1]
ifMatch : string [0..1]
ifNoneExist : string [0..1]
[0..1]
request
Entry
fullUrl : uri [0..1]
resource : Resource [0..1]
search
[0..1]
Search
mode : code [0..1] « SearchEntryMode! »
score : decimal [0..1]
response
[0..1]
Response
status : string [1..1]
location : uri [0..1]
etag : string [0..1]
lastModified : instant [0..1]
outcome : Resource [0..1]

Figure 4 Bundle UML representation

### 1.1.4 Restful API Operations

FHIR standard, as well as providing modelling of resources useful for data exchange, define a set of software interfaces by means different software systems can communicate and exchange information. FHIR mainly supports four types of paradigms:

- REST interfaces.
- Documents transport.
- Messages exchange.
- Exposure and invocation services.

#### 1.1.4.1 REST Interfaces

REST is the simplest and most used paradigm for exchanging information between software applications. The FHIR standard has defined a set of RESTful APIs able to satisfy a series of operations coming from a software agent (returning to the metaphor it is as if there were an employee who manages the requests regarding the data present in the files). The operations defined by FHIR are:

- **Search**: search for information in the database that meets the criteria defined by the user in the request and returns a copy of the latest version.
- **Read**: returns a copy of the latest version of a specific resource starting from its id.
- **Create:** storage of a new resource with the correct id.
- **Update**: add a new version of an existing resource.
- **Delete**: a resource. The removal is only virtual as the resource is not actually deleted but it is marked as no longer valid and accessible.
- **History**: returns all existing versions of a specific resource accessible with its id. This operation is used more for administrative than clinical purposes.
- **Transaction**: ability to perform multiple operations in atomic or batch mode.
- **Search-operation**: request to the server to operate a specific action or procedure in relation to a specific version of one or more resources. For example: providing the average number of patients, advanced searches, etc.

## 1.2 Resource Description Framework (RDF)

In this section, a brief overview of Resource Description Framework is resumed since it will be used *as the main framework* for data exposure in the Semantic Data lake through the adoption of an (RDF based) graph database (e.g. RDF4J).

### 1.2.1 Introduction

The World Wide Web affords unprecedented access to globally distributed information. Metadata, or structured data about data, improves discovery of and access to such information. The effective use of metadata among applications, however, requires common conventions about semantics, syntax, and structure. Individual resource description communities define the semantics, or meaning, of metadata that addresses their particular needs. Syntax, the systematic arrangement of data elements for machine-processing, facilitates the exchange and use of metadata among multiple applications. Structure can be thought of as a formal constraint on the syntax for the consistent representation of semantics.

The Resource Description Framework (RDF), developed under the auspices of the World Wide Web Consortium (W3C) [2], is an infrastructure that enables the encoding, exchange, and reuse of structured metadata. This infrastructure consists of two main components:

- **RDF model and Syntax**: exposes the structure of the RDF model, and describes a possible syntax.
- **RDF Schema**: exposes the syntax for defining patterns and vocabularies for metadata.

RDF supports specific syntaxes:

- **Turtle**, compact, human-friendly format, and **TriG**.
- **JSON-LD** (JSON based), a JSON-based serialization (for Linked Data).
- **RDFa** (for HTML embedding), not really an RDF syntax, but rather – a compatible format. RDFa is an extension to HTML5 that helps you markup things like People, Places, Events, Recipes and Reviews. Search Engines and Web Services use this markup to generate better search listings and give you better visibility on the Web, so that people can find your website more easily.
- **Notation3** (N3), a non-standard serialization that is very similar to Turtle, but has some additional features, such as the ability to define inference rules.
- **N-Triples**, very simple, easy-to-parse, line-based format that is not as compact as Turtle, and **N-Quads** (line-based exchanges formats).
- **XML**, base syntax for RDF graphs that was the first standard format for serializing RDF).

RDF supports the use of conventions that will facilitate modular interoperability among separate metadata element sets. These conventions include standard mechanisms for representing semantics that is grounded in a simple, yet powerful, data model discussed below. RDF additionally provides a means for publishing both human-readable and machine-processable vocabularies. Vocabularies are the set of properties, or metadata elements, defined by resource description communities. The ability to standardize the declaration of vocabularies is anticipated to encourage the reuse and extension of semantics among disparate information communities.

The goals of RDF are broad, and the potential opportunities are enormous. This introduction to RDF begins by discussing the background context of the RDF initiative and relates it to other metadata activities.

### 1.2.2The RDF Data Model

RDF provides a model for describing resources. Resources have properties (attributes or characteristics). RDF defines a resource as any object that is uniquely identifiable by a Uniform Resource Identifier (URI) [3]]. The properties associated with resources are identified by property-types, and property-types have corresponding values. Property-types express the relationships of values associated with resources. In RDF, values may be atomic in nature (text strings, numbers, etc.) or other resources, which in turn may have their own properties. A collection of these properties that refers to the same resource is called a description. At the core of RDF is a syntax-independent model for representing resources and their corresponding descriptions. The following graphic (Figure 5) illustrates a generic RDF description.

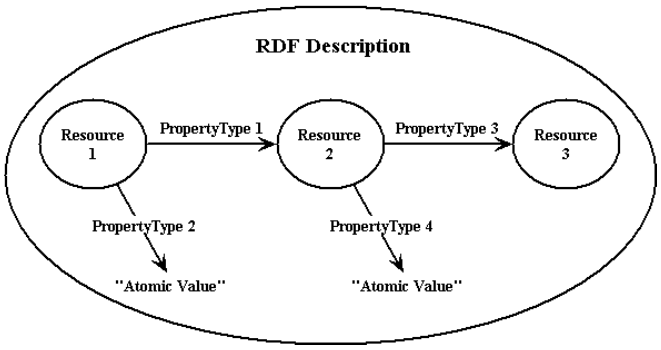


Figure 5 Generic RDF description

The application and use of the RDF data model can be illustrated by concrete examples. Consider the following statements:

"The author of Document 1 is John Smith"

"John Smith is the author of Document 1"

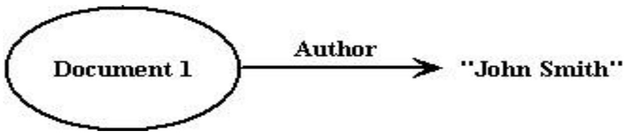


Figure 6 RDF concrete example

To humans, these statements convey the same meaning (that is, John Smith is the author of a particular document). To a machine, however, these are completely different strings. Whereas humans are extremely adept at extracting meaning from differing syntactic constructs, machines remain grossly inept. Using a triadic model of resources, property-types and corresponding values, RDF attempts to provide an unambiguous method of expressing semantics in a machine-readable encoding. RDF provides a mechanism for associating properties with resources. So, before anything about Document 1 can be said, the data model requires the declaration of a resource representing Document 1. Thus, the data model corresponding to the statement "the author of Document 1 is John Smith" has a single resource Document 1, a property-type of author and a corresponding value of John Smith. To distinguish characteristics of the data model, the RDF Model and Syntax specification [SPEC] represents the relationships among resources, property-types, and values in a directed labelled graph. In this case, resources are identified as nodes, property-types are defined as directed label arcs, and string values are quoted. Given this representation, the data model corresponding to the statement is graphically expressed as (Figure 6): If additional descriptive information regarding the author were desired, e.g., the author's email address and affiliation, an elaboration on the previous example would be required. In this case, descriptive information about John Smith is desired. As was discussed in the first example, before descriptive properties can be expressed about the person John Smith, there needs to be a unique identifiable resource representing him. Given the directed label graph notation in the previous example, the data model corresponding to this description is graphically represented as (Figure 7).

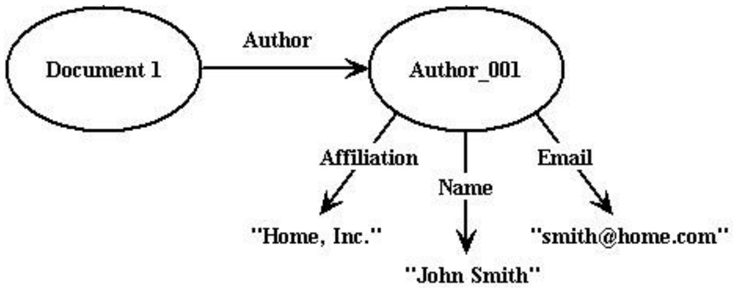


Figure 7 RDF concrete example with addition information

In this case, "John Smith" the string is replaced by a uniquely identified resource denoted by Author_001 with the associated property-types of name, email and affiliation. The use of unique identifiers for resources allows for the unambiguous association of properties. This is an important point, as the person John Smith may be the value of several different property-types. John Smith may be the author of Document 1, but also maybe the value of a particular company describing the set of current employees. The unambiguous identification of resources provides for the reuse of explicit, descriptive information. In the previous example the unique identifiable resource for the author was created, but not for the author's name, email or affiliation. The RDF model allows for the creation of resources at multiple levels. Concerning the representation of personal names, for example, the creation of a resource representing the author's name could have additionally been described using "firstname", "middlename" and "surname" property-types. Clearly, this iterative descriptive process could continue down many levels

### 1.2.3The RDF syntax

RDF defines a simple, yet powerful model for describing resources. A syntax representing this model is required to store instances of this model into machine-readable files and to communicate these instances among applications. XML was the first base syntax for RDF serialization imposing formal structure to support the consistent representation of semantics. For sick of simplicity the examples of this section will rely on the XML syntax.

RDF provides the ability for resource description communities to define the semantics. It is important, however, to disambiguate these semantics among communities. The property-type "author", for example, may have broader or narrower meaning depending on different community needs. As such, it is problematic if multiple communities use the same property-type to mean very different things. To prevent this, RDF uniquely identifies property-types by using the XML namespace mechanism. XML namespaces provide a method for unambiguously identifying the semantics and conventions governing the particular use of property-types by uniquely identifying the governing authority of the vocabulary. For example, the property-type "author" defined by the Dublin Core Initiative as the "person or organization responsible for the creation of the intellectual content of the resource" and is specified by the Dublin Core CREATOR element [4]. An XML namespace is used to unambiguously identify the Schema for the Dublin Core vocabulary by pointing to the definitive Dublin Core resource that defines the corresponding semantics. Additional information on RDF Schemas is discussed later. If the Dublin Core RDF Schema, however, is abbreviated as "DC", the data model representation for this example would be (Figure 8):

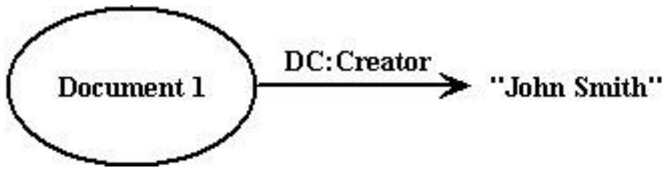


Figure 8 Dublin Core RDF Schema

This more explicit declaration identifies a resource Document 1 with the semantics of property-type Creator unambiguously defined in the context of DC (the Dublin Core vocabulary). The value of this property-type is John Smith. The corresponding syntactic way of expressing this statement using XML namespaces to identify the use of the Dublin Core Schema is:

```
<?xml:namespace ns = "http://www.w3.org/RDF/RDF/" prefix ="RDF" ?>
<?xml:namespace ns = "http://purl.oclc.org/DC/" prefix = "DC" ?>
<RDF:RDF>
  <RDF:Description RDF:HREF = "http://uri-of-Document-1">
    <DC:Creator>John Smith</DC:Creator>
  </RDF:Description>
</RDF:RDF>
```

In this case, both the RDF and Dublin Core schemas are declared and abbreviated as "RDF" and "DC" respectively. The RDF Schema is declared as a boot-strapping mechanism for the declaration of the necessary vocabulary needed for expressing the data model. The Dublin Core Schema is declared in order to utilize the vocabulary defined by this community. The URI associated with the namespace declaration references the corresponding schemas. The element <RDF:RDF> (which can be interpreted as the element RDF in the context of the RDF namespace) is a simple wrapper that marks the boundaries in an XML document where the content is explicitly intended to be mappable into an RDF data model instance [5]. The element <RDF:Description> (the element Description in the context of the RDF namespace) is correspondingly used to denote or instantiate a resource with the corresponding URI http://uri-of-Document-1. And the element <DC:Creator> in the context of the <RDF:Description> represents a property-type DC:Creator and a value of "John Smith". The syntactic representation is designed to reflect the corresponding data model. In the more advanced example, where additional descriptive information regarding the author is required, similar syntactic constructs are used. In this case, while it may still be desirable to use the Dublin Core CREATOR property-type to represent the person responsible for the creation of the intellectual content, additional property-types "name", "email" and "affiliation" are required. For this case, since the semantics for these elements are not defined in Dublin Core, an additional resource description standard may be utilized. It is feasible to assume the creation of an RDF schema with the semantics similar to the vCard [6] specification designed to automate the exchange of personal information typically found on a traditional business card, could be introduced to describe the author of the document. The data model representation for this example with the corresponding business card schema defined as CARD would be (Figure 9):

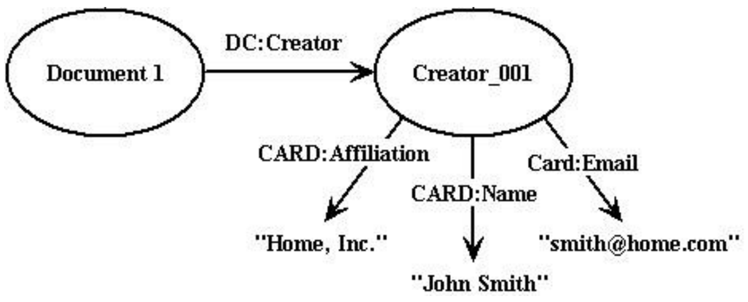


Figure 9 RDF business card schema example

This, in turn, could be syntactically represented as

```
<?xml:namespace ns = "http://www.w3.org/RDF/RDF/" prefix = "RDF" ?>
<?xml:namespace ns = "http://purl.oclc.org/DC/" prefix = "DC" ?>
<?xml:namespace ns = "http://person.org/BusinessCard/" prefix = "CARD" ?>

<RDF:RDF>
  <RDF:Description RDF:HREF = "http://uri-of-Document-1">
    <DC:Creator RDF:HREF = "#Creator_001"/>
  </RDF:Description>

  <RDF:Description ID="Creator_001">
    <CARD:Name>John Smith</CARD:Name>
    <CARD:Email>smith@home.net</CARD:Email>
    <CARD:Affiliation>Home, Inc.</CARD:Affiliation>
  </RDF:Description>
</RDF:RDF>
```

in which the RDF, Dublin Core, and the "Business Card" schemas are declared and abbreviated as "RDF", "DC" and "CARD" respectively. In this case, the value associated with the property-type DC:Creator is now a resource. While the reference to the resource is an internal identifier, an external URI, for example, to a controlled authority of names, could have been used as well. Additionally, in this example, the semantics of the Dublin Core CREATOR element have been refined by the semantics defined by the schema referenced by CARD.

### 1.2.4The RDF Schema

RDF Schemas are used to declare vocabularies, the sets of semantics property-types defined by a particular community. RDF schemas define the valid properties in a given RDF description, as well as any characteristics or restrictions of the property-type values themselves. The XML namespace mechanism serves to identify RDF Schemas. A human and machine-processable description of an RDF schema may be accessed by de-referencing the schema URI. If the schema is machine-processable, it may be possible for an application to learn some of the semantics of the property-types named in the schema. To understand a particular RDF schema is to understand the semantics of each of the properties in that description. RDF schemas are structured based on the RDF data model. Therefore, an application that has no understanding of a particular schema will still be able to parse the description into the property-type and corresponding values and will be able to transport the description intact (e.g., to a cache or to another application). The ability to formalize human-readable and machine-processable vocabularies will encourage the exchange, use, and extension of metadata vocabularies among disparate information communities. RDF schemas are being designed to provide this type of formalization.

## 1.3 RML Mapping Language (RML)

The RDF Mapping language (RML) [7] is a generic mapping language defined to express customized mapping rules from heterogeneous data structures and serializations to the RDF data model. RML is defined as a superset of the w3c standardized mapping language RML, aiming to extend its applicability and broaden its scope.

RML keeps the mapping definitions as in RML but excludes its database-specific references from the core model. The potential broad concepts of RML, which were explained previously, are formally designated in the frame of the RML mapping language and are elaborated upon here. The primary difference is the potential input that is limited to a certain database in the case of RML, while it can be a broad set of (one or more) input sources in the case of RML. RML provides a generic way of defining the mappings that is easily transferable to cover references to other data structures, combined with case-specific extensions, but always remains backwards compatible with RML as relational databases form such a specific case. RML considers that the mappings to RDF of sets of sources that all together describe a certain domain, can be defined in a combined and uniform way, while the mapping definitions may be re-used across different sources that describe the same domain to incrementally form well-integrated datasets.

A RML mapping definition follows the same syntax as RML. The RML vocabulary namespace is http://semweb. mmlab.be/ns/rml# and the preferred prefix is RML. More details about the RML mapping language can be found at http://rml. io. Defining and executing a mapping with RML requires the user to provide a valid and well-formatted input dataset to be mapped and the mapping definition (mapping document) according to which the mapping will be executed to generate the data’s representation using the RDF data model (output dataset). Data cleansing is out of the scope of the language’s definition and, if necessary, should be performed in advance. An extract of two heterogeneous input sources is displayed at Listing 1, an example of a corresponding mapping definition is displayed at Listing 3 and the produced output at Listing 2.

*Logical Source*. A Logical Source (rml:LogicalSource) extends rml’s Logical Table and is used to determine the input source with the data to be mapped. The rml Logical Table definition determines a database’s table, using the Table Name (rr:tableName). In the case of RML, a broader reference to any input source is required. Thus, the Logical Source and source rml:source) are introduced respectively to specify the input.

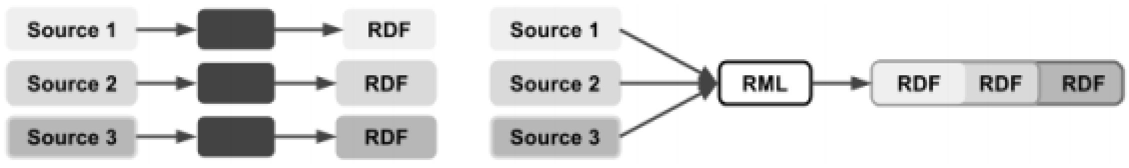


Figure 10 Mapping sources without and with RML

```
{ ...  "Performance" :
        { "Perf_ID": "567",
          "Venue": {  "Name": "STAM",
                      "Venue_ID": "78" },
          "Location": { "long": "3.717222",
                        "lat": "51.043611" } } , ... }
<Events> ...
  <Exhibition id="398">
    <Venue> STAM </Venue>
    <Location>
      <lat>51.043611</lat>
      <long>3.717222</long>
    </Location>
  </Exhibition> ... ...
</Events>
```

Figure 11 performances.json and exhibitions.xml

```
ex:567        ex:venue    ex:78 ;
              ex:location ex:3.717222,51.043611 .
ex:398        ex:venue    ex:78 ;
              ex:location ex:3.717222,51.043611 .
ex:3.717222,51.043611    ex:lat      ex:3.717222
                         ex:long     ex:51.043611.
```

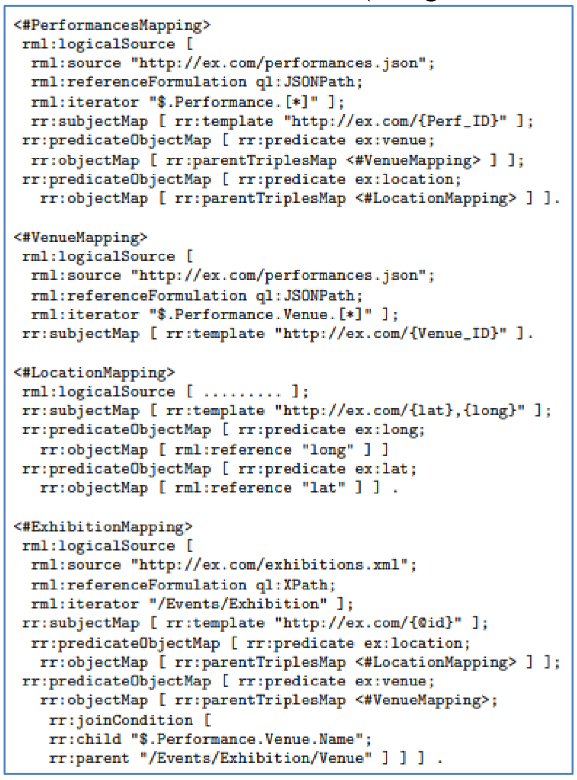
Figure 12 The expected output.

***Reference Formulation***. RML needs to deal with different data serialisations which use different ways to refer to their elements/objects. But, as RML aims to be generic, not a uniform way of referring to the data's elements/objects is defined. R2rml uses columns' names for this purpose. In the same context, RML considers that any reference to the Logical Source should be defined in a form relevant to the input data, e.g. XPath for xml files or jsonpath for json files. To this end, the Reference Formulation (rml:referenceFormulation) declaration is introduced indicating the formulation (for instance, a standard or a query language) used to refer to its data. At the current version of RML, the ql:CSV, ql:XPath and ql:JSONPath Reference Formulations are predefined.

***Iterator***. While in RML it is already known that an arrow iteration occurs, as RML remains generic, the iteration pattern, if any, cannot always be implicitly assumed, but it needs to be determined. Thereafter, the iterator (rml:iterator) is introduced. The iterator determines the iteration pattern over the input source and specifies the extract of the data mapped during each iteration. For example, the "$.[*]" determines the iteration over a json file that occurs over the object's outer level. The iterator is not required in the case of tabular sources as the default per-row iteration is implied or if there is no need to iterate over the input data.

***Logical Reference***. A column-valued term map, according to r□rml, is defined using the property rr:column which determines a column's name. In the case of rml, a more generic property is introduced rml:reference. Its value must be a valid reference to the data of the input dataset. Therefore, the reference's value should be a valid expression according to the Reference Formulation defined at the Logical Source, as well as the string template used in the definition of a template-valued term map and the iterator's value. For instance, the iterator, the subject's template-valued term map and the object's reference-valued term map are all valid jsonpath expressions.

***Referencing Object Map***. The last aspect of r□rml that is extended in RML is the Referencing Object Map. The join condition's child reference (rr:child) indicates the reference to the data value (using an rml:reference) of the Logical Source that contains the Referencing Object Map. The join condition's child reference (rr:parent) indicates the reference to the data extract (rr:reference) of the Referencing Object Map's Parent Triples Map. The reference is specified using the Reference Formulation defined at the current Logical Source. The join condition's parent reference indicates the reference to the data extract (rml:reference) of the Parent Triples Map. The reference is specified using the Reference Formulation defined at the Parent Triples Map Logical Source definition. Therefore, the child reference and the parent reference of a join condition may be defined using different Reference Formulations, if the Triples Map refers to sources of different format.



```
<#PerformancesMapping>
 rml:logicalSource [
  rml:source "http://ex.com/performances.json";
  rml:referenceFormulation ql:JSONPath;
  rml:iterator "$.Performance.[*]" ];
  rr:subjectMap [ rr:template "http://ex.com/{Perf_ID}" ];
 rr:predicateObjectMap [ rr:predicate ex:venue;
  rr:objectMap [ rr:parentTriplesMap <#VenueMapping> ] ];
 rr:predicateObjectMap [ rr:predicate ex:location;
   rr:objectMap [ rr:parentTriplesMap <#LocationMapping> ] ].

<#VenueMapping>
 rml:logicalSource [
  rml:source "http://ex.com/performances.json";
  rml:referenceFormulation ql:JSONPath;
  rml:iterator "$.Performance.Venue.[*]" ];
 rr:subjectMap [ rr:template "http://ex.com/{Venue_ID}" ].

<#LocationMapping>
 rml:logicalSource [ ......... ];
 rr:subjectMap [ rr:template "http://ex.com/{lat},{long}" ];
 rr:predicateObjectMap [ rr:predicate ex:long;
   rr:objectMap [ rml:reference "long" ] ]
 rr:predicateObjectMap [ rr:predicate ex:lat;
   rr:objectMap [ rml:reference "lat" ] ] .

<#ExhibitionMapping>
 rml:logicalSource [
  rml:source "http://ex.com/exhibitions.xml";
  rml:referenceFormulation ql:XPath;
  rml:iterator "/Events/Exhibition" ];
 rr:subjectMap [ rr:template "http://ex.com/{@id}" ];
  rr:predicateObjectMap [ rr:predicate ex:location;
   rr:objectMap [ rr:parentTriplesMap <#LocationMapping> ] ];
 rr:predicateObjectMap [ rr:predicate ex:venue;
   rr:objectMap [ rr:parentTriplesMap <#VenueMapping>;
    rr:joinCondition [
    rr:child "$.Performance.Venue.Name";
    rr:parent "/Events/Exhibition/Venue" ] ] ] .
```

Figure 13 An RML mapping definition

## 1.4 Conclusions

The previous sections provided a global overview of the standards adopted by the DFI (Data federation Framework) architecture in order to reach an interoperability semantic enabling data exchange by means of uniform data access for further predictive analytics and data mining. In particular, a general picture of the HL7 FHIR standard has been reported since it will be the main semantic model adopted for the GK data representation and data exchange.  Also, an overview of the RDF framework has been provided, since it will be used for data exposure through the adoption of an RDF based graph database (e.g. RDF4J). RDF is a foundation for processing metadata, it provides interoperability between applications that exchange machine-understandable information on the Web. RDF emphasizes facilities to enable automated processing of Web resources. Finally, the RML language has been briefly described since it will be adopted to build "mapping rule", enabling the creation of RDF knowledge graph starting from heterogeneous raw data. The adoption of such a language will offer a declarative approach as alternative to a programmatic approach based on java based converters (see Section 2.3.3 for details).

# 2 Data Federation and Integration V1: overview, requirements, design

## 2.1 Position of the Data Federation & Integration into Gatekeeper architecture

The Data Federation and Integration (DFI) is one of the core components described in deliverable D3.2. Its purpose is twice: (i) to offer an easy modality to enable external heterogeneous data sources to send their data by harmonizing such data against common semantic models selected by the project (e.g. HL7 FHIR) and (ii) to allow the other "Thing" (e.g. the Integrated Dynamic Intervention Services of WP5 or even external applications) to access such data.

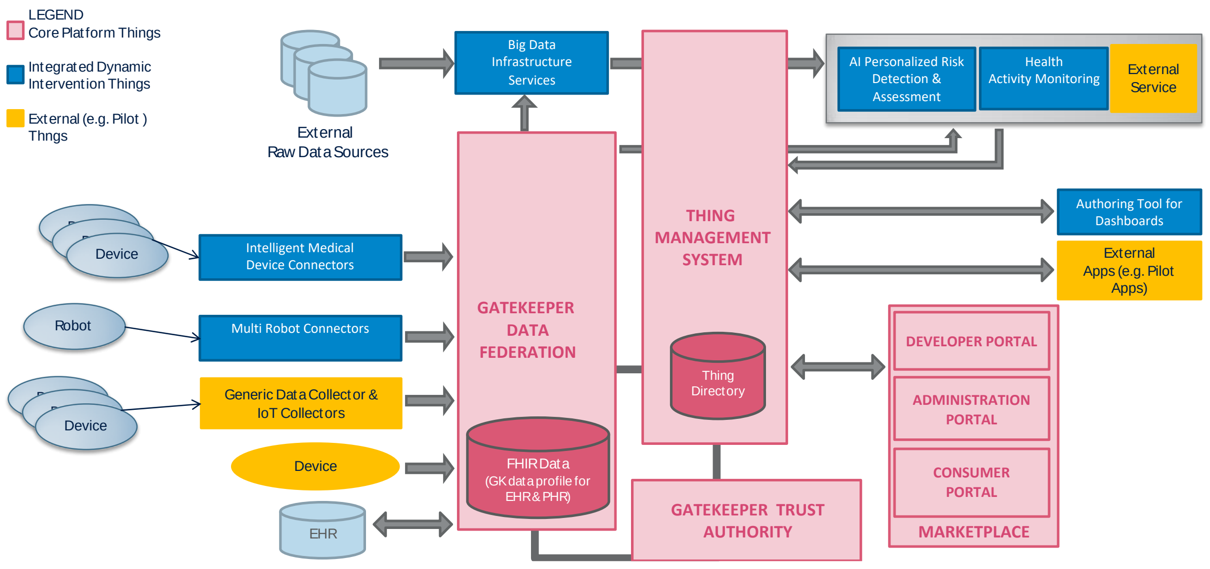

LEGEND
Core Platform Things
Integrated Dynamic Intervention Things
External (e.g. Pilot ) Thngs
External Raw Data Sources
Big Data Infrastructure Services
THING MANAGEMENT SYSTEM
AI Personalized Risk Detection & Assessment
Health Activity Monitoring
External Service
Authoring Tool for Dashboards
External Apps (e.g. Pilot Apps)
Device
Intelligent Medical Device Connectors
Robot
Multi Robot Connectors
Device
Generic Data Collector & IoT Collectors
Device
EHR
GATEKEEPER DATA FEDERATION
FHIR Data (GK data profile for EHR & PHR)
Thing Directory
GATEKEEPER TRUST AUTHORITY
DEVELOPER PORTAL
ADMINISTRATION PORTAL
CONSUMER PORTAL
MARKETPLACE

Figure 14 Gatekeeper architecture

It is worth to mention that the DFI is itself an aggregation of WoT, as it will be clarified in the next sections. Consequently, "Thing(s)" will be exposed and accessible, by the other components, exclusively through the interaction with the Thing Management System (TMS). Such mediated access also guarantees the respect of the authentication and authorization policies since the TMS performs security check (interacting with the Gatekeeper Trust Authority – GTA – component) each time an access to a GK Thing is requested.

## 2.2Data integration process and requirements

### 2.2.1Problems, Approaches and Perspectives

The goal of data integration is to create a single view of data integrating such data coming from heterogeneous data sources, distributed among heterogeneous information sources that can be structured or semi-structured.

Thus, in general, gathering information is challenging, and one of the main reasons is that data sources are designed to support specific applications. Very often their structure is unknown to the large part of users. Moreover, the stored data is often redundant, mixed with information only needed to support enterprise processes, and incomplete with respect to the business domain. Collecting, integrating, reconciling, and efficiently extracting information from heterogeneous and autonomous data sources is regarded as a major challenge.

The integration of multiple data information systems aims to combine the selected systems to form a unique form so that they form a unified new whole and give the user the illusion of interacting with one single information system. Users are provided with a homogeneous logical view of data that is physically distributed over heterogeneous data sources. For this, all data must be represented using the same abstraction principles (unified global data model and unified semantics). This task includes detection and resolution of schema and data conflicts regarding structure and semantics.

Generally, systems are not designed for integration. This means that whenever it is needed integrated access consisting of different systems, sources and their data that not fit among them must be integrated using several adaptation and conciliation functionalities. It is to understand that there is not a single integration problem even if the goal is always to provide a homogeneous and unified view on data from different sources. In details the integration task can depend on:

- the architecture of the information system
- the content and functionalities of the component systems
- the kind of information that is managed by component systems (alphanumeric data, multimedia data; structured, semi-structured, unstructured data)
- requirements concerning the autonomy of component systems intended use of the integrated information system (read-only or write access)
- performance requirements.

Additionally, several kinds of heterogeneity typically have to be considered. These include differences in data management software, data models, schemas, data semantics, middleware, user interfaces and business rules and integration constraints.

In order to select the best approach to be involved in the Data Federation & Integration module, several solutions have been investigated [8] distinguishing integration approaches according to the level of abstraction where integration is performed. Integration can be done manually in this case users directly interact with all relevant information systems and manually integrate selected data. That is, users have to deal with different user interfaces and query languages. Additionally, users need to have details knowledge on location, logical data representation, and data semantics. Another level is a common user interface where the user is supplied with a common user interface that provides a uniform look and feel. Data from relevant information system is still separately presented so that homogenization and integration of data yet have to be done by the users. Integration can be done by applications, in This approach uses integration applications that access various data sources and return integrated results to the user. This solution is practical for a small number of component systems. However, applications become increasingly fat as the number of system interfaces and data formats to homogenize and integrate grows. Integration by middleware where middleware provides reusable functionality that is generally used to solve dedicated aspects of the integration problem, e.g., as done by SQLmiddleware. While applications are relieved from implementing common integration functionality, integration efforts are still needed in applications. Additionally, different middleware tools usually have to be combined to build integrated systems. The integration at the level of uniform data accesses a logical integration of data is accomplished at the data access level. Global applications are provided with a unified global view of physically distributed data, though only virtual data is available on this level. Local information systems keep their autonomy and can support additional data access layers for other applications. However, the global provision of physically integrated data can be time-consuming since data access, homogenization, and integration have to be done at runtime.

The integration using a common data store, physical data integration is performed by transferring data to a new data storage; local sources can either be retired or remain operational. In general, physical data integration provides fast data access. However, if local data sources are retired, applications that access them have to be migrated to the new data storage as well. In case local data sources remain operational, periodical refreshing of the common data storage needs to be considered.

In the Data Federation & integration engine, a kind of Personal Data Integrations Systems (PDIS) has been adopted. PDIS are a special form of manual integration. Here, tailored integrated views are defined (e.g., by a declarative integration language), either by users themselves or by dedicated integration engineers. Each integrated view precisely matches the information needs of a user by encompassing all relevant entities with real-world semantics as intended by the particular user; thereby, the integrated view reflects the user’s personal way to perceive his application domain of interest.

Next section reports all details about the followed process that has been selected based on the requirements analysis came by the several pilots involved in the GateKeeper project.

### 2.2.2Requirements

This section will be briefly described the requirements affecting the GK integration process. They have been collected by analyzing the DOA description from one side and pilot specific requests from the other side. Such requirements represented the baseline for the Data Federation & Integration design activity outlined in the next sections. Requirements from pilots have been collected during dedicated phc and by analysing documents produced in other work packages (e.g. WP3). Before resuming the requirements in Table 8, a brief overview of the analysis pilot per pilot is reported:

- *Puglia*

In the Puglia pilot several external systems are involved as described during dedicated phc and also reported in the pilot specific architecture (deliverable D1.3). More in detail it is expected the involvement of two intermediary collecting services, linked to technologies provided by other GATEKEEPER Partners (namely, Medisantè ELIOT Hub and Samsung Health) as well as – in perspective – the possibility to also integrate market available data collection platforms (e.g. Google Fit, or even specific providers such as Withings, https://www.withings.com/) in order to gather data from a wider set of **IoT** sensors, either provided to patients by healthcare providers or directly acquired by the patients themselves on the consumer market, with the ultimate goal of consistently presenting such data to clinicians (mainly GPs, in the course of the Pilot experiment, but also specialists or hospital clinicians, in perspective), for them to obtain a richer but uniform view on patients' health status, meeting the monitoring needs of various health profiles of elderly citizens in the Puglia Region.

Moreover, other data are expected to be received externally from the **HIS** (Hospital Information System) of "Casa Sollievo della Sofferenza" Hospital as outlined in Figure 15, in order to conduct research on predictive models for diabetes control, that include both features available in the HIS and features coming from consumer devices, such as smartwatches equipped with HR/HRV, physical activity, sleep quality and stress detection sensors.

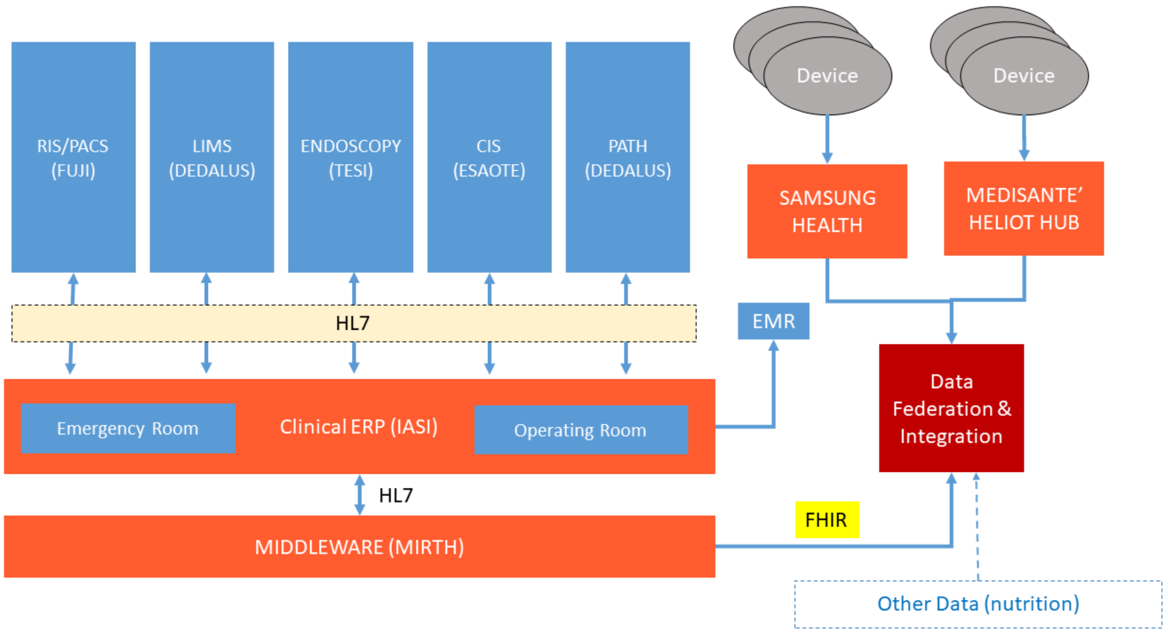


Figure 15 Puglia pilot scenario

The Medisantè ELIOT Hub is a Cloud service able to collect and forward (**PUSH**) data to other systems. It can be configured in order to register the third part API to call for data forwarding. The Samsung Health based client, is an Android based mobile app able to retrieve data from sensors (that need to be paired to the app through Bluetooth) store such data on a local PHR on the smartphone (Samsung Health Store) and synchronize such data with the Samsung Health Server in the Cloud and send (**PUSH**) such data to other systems.

Moving to the CSS's EHR data sources it is worth to point out as several internal (HIS) systems could be involved in principles (e.g. RIS/PACS, UMS, ENDOSCOPY etc.). As showed in the figure an intermediary middleware (Mirth) will be exploited to collect and send (**PUSH**) the data to the Data Federation. The input data format is expected to be already compliant to HL7 FHIR standard, in this pilot, so that the Data Federation will be mainly involved (i) to adapt the structure to the specific GK-FHIR profile (ii) to redirect (**ROUTE**) the data to the pilot specific FHIR server.

By exploring the capability of such middleware, it also resulted, indeed, the possibility for an external system to retrieve (**PULL**) data. At the moment of writing the document it was not yet clear if PULL based retrieval modality will be exploited.

About the output semantic model, the pilot aims at building final applications (e.g. DMCoach for type II diabetes management) relying on state of the art HL7 standard (i.e. **FHIR**). It is expected the input data (both IoT and EHR) to be "harmonized" (i.e. converted) to such semantic model. At the moment of writing this deliverable it was not yet clear if the availability of data also in a graph DB (e.g. RDF4j, Neo4j) is needed, to fully exploit the semantic reasoning capabilities that is something useful for the pilot and the intelligent GK components (i.e. WP5). Finally, about the data location the pilot team has expressed the preference that acquired data, through the data federation, would be held in a **dedicated cloud cluster**.

In the following table, details are resumed regarding the pilot specific requirements arising from the analysis above.

Table 1 Puglia pilot sources

| Source type | Device Type | Gateway | DFI Interaction Modality | Output Semantic Model | Graph DB | Data Location |
|---|---|---|---|---|---|---|
| **IOT** | BP800 (BP, Glucose) | Medisantè ELIOT Hub | PUSH | GK-FHIR Profile | N/A | Dedicated cluster |
| | BC800 (body weight and composition) | | | | | |
| | Biobeat wrist device (HR, BP, SpO2) | | | | | |
| | Samsung smartwatch (HR, physical activity, sleep, stress level) and Activage | Samsung Health and Bixby capsules | PUSH | | | |
| **EHR** | - | Mirth | PUSH (PULL) | GK-FHIR Profile | N/A | Dedicated cluster |

### - *Saxony*

In the Saxony pilot, several external systems (e.g. \) are involved as described during dedicated phone calls and also reported in the pilot specific architecture (deliverable D3.1). More in detail it is expected the involvement of an intermediary collecting service (Samsung Health) in order to gather data from several **IoT** sensors (Figure 16).

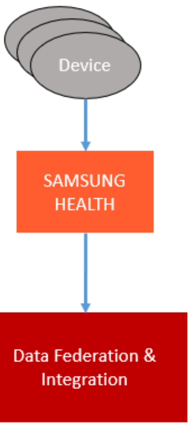


Figure 16 Saxony pilot scenario

Data coming from Samsung devices are collected by Samsung Health and forwarded (**PUSH**) to Data Federation.

In the following table, details are reported about the device name, intermediate gateway involved and main expected interaction modality.

Table 2 Saxony pilot sources

| Source type | Device Type | Gateway | DFI Interaction Modality | Output Semantic Model | Graph DB | Data Location |
|---|---|---|---|---|---|---|
| IOT | Samsung Smartphone | Samsung Health and Bixby capsules | PUSH | GK-FHIR Profile | N/A | Dedicated cluster |
|  | Samsung Tablet |  |  |  |  |  |
|  | Samsung smartwatch (HR, physical activity, sleep, stress level) |  |  |  |  |  |

About the output model, the work package 5 aims at building a final Web-based platform for clinicians relying on the state-of-the-art HL7 standard (i.e. **FHIR**) to communicate, receive notifications and for remote monitoring. It is expected the input data to be "harmonized" (i.e. converted) to such semantic model. At the moment of writing this deliverable it was not yet clear if the availability of data also in a graph DB (e.g. RDF4j, Neo4j), to fully exploit the semantic reasoning capabilities) is something useful for the pilot and the intelligent GK components (i.e. WP5).

Finally, about the data location the pilot team has expressed the preference that acquired data, through the data federation, would be held in a **dedicated cloud cluster**.

- *Aragon*

In the Aragon pilot, several external systems are involved as described during dedicated phc and also reported in the pilot specific architecture (deliverable D3.1). More in detail it is expected the involvement only one intermediary collecting service called **Data Extraction** that is a module that will be implemented inside the Salud Application as outlined in Figure 17.

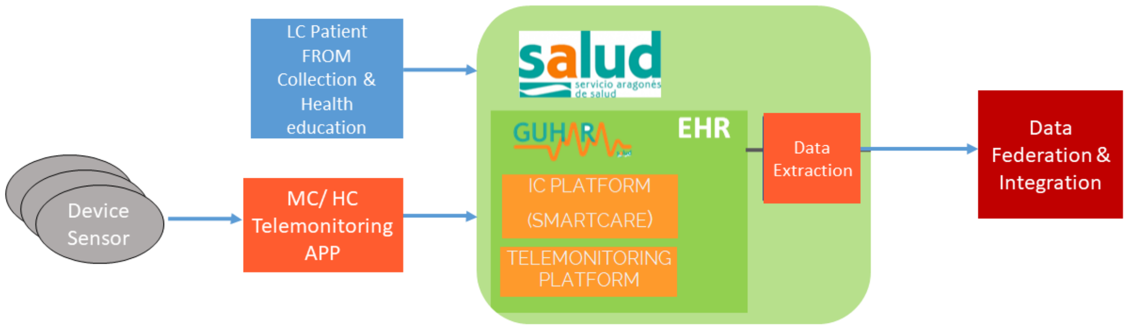

LC Patient FROM Collection & Health education
salud servicio aragonés de salud
GUHARA
EHR
IC PLATFORM (SMARTCARE)
TELEMONITORING PLATFORM
Data Extraction
Data Federation & Integration
Device Sensor
MC/ HC Telemonitoring APP

Figure 17 Aragon pilot scenario

**Salud** is an EHR data source that collects and groups data coming from two components: “LC Patient FROM Collection & Health education” and “MC/HC Telemonitoring APP”. The first is used by patients aiming to manage information about their health education while the second one is a gateway, running on smartphone, that retrieves some data coming from sensors and devices and forwards such data to **Salud** web-app. At the moment of writing the document the list of devices and sensors that will be used is still under the decision.

**Salud** application manages data and information about:

- **Patient / participant** including the basic personal, demographic and recruitment data of the citizen.
- **Social assessment** with basic information regarding social status.
- **Habits** with information on daily routines.
- Clinical Activity **(Hospitalisation)** with information regarding admissions to the Hospital.
- Clinical Activity **(Consultations)** including information related to consultations in primary and specialized care.
- **Prescribed Medication** with information on the drugs that the patient is prescribed.
- **Clinical variables** values, including information on vital signs capture values

- **Symptoms** representing information about the existence and/or the intensity of symptoms
- **Forms and questionnaires** (e.g. PROMS)
- **Comorbidities.** Additional pathologies that belong to episodes active in the patient EHR that are different from the main disease.

As shown in the figure above it is not expected a direct interaction with the GK platform, but the use of a **Data Extractor** engine, deployed inside **Salud** application, that sends data to the Data Federation & Integration. Such engine extracts specific data from the Salut EHR and sends to DFI by mean **PUSH** modality. The input format is expected to be in custom JSON or XML representation, in this pilot, so that the DFI will be mainly involved (i) to adapt the structure to the specific GK-FHIR profile (ii) to redirect (**ROUTE**) the data to the pilot specific FHIR server.

About the output semantic model, the pilot aims at building final applications (e.g. machine learning algorithms) relying on the state of the art HL7 standard (i.e. **FHIR**). It is expected the input data to be "harmonized" (i.e. converted) to such semantic model. At the moment of writing this deliverable, it was not yet clear if the availability of data also in a graph DB (e.g. RDF4j, Neo4j), to fully exploit the semantic reasoning capabilities that is something useful for the pilot and the intelligent GK components (i.e. WP5).

Finally, about the data location the pilot team has expressed the intention of evaluate the opportunity to send out their data from their owner premise in order to feed GK systems. In the case acquired data, through the data federation, would be held in a **dedicated cloud cluster** unless there will be a strong justification to have data in a shared cloud cluster. Anyway, the analysis of the output semantic model and the use of a shared/dedicated cluster is still under evaluation.

In the following table, details are resumed the pilot specific requirements arising from the analysis above.

Table 3 Aragon pilot sources

| Source type | Device Type | Gateway | DFI Interaction Modality | Output Semantic Model | Graph DB | Data Location |
|---|---|---|---|---|---|---|
| **EHR** | Sensor | Samsung Health and Bixby capsules | PUSH | GK-FHIR Profile | N/A | Local premise (details in the section) |
| | Samsung Tablet | | | | | |
| | Samsung smartwatch (HR, physical activity, sleep, stress level) | | | | | |

### - *Greece*

In the Greece pilot, several external systems are involved as described during dedicated phc and also reported in the pilot specific architecture (deliverable D3.1). More in detail it is expected the involvement of two intermediary collecting services (Medisantè and Heliot Samsung Health) in order to gather data from several **IoT** devices.

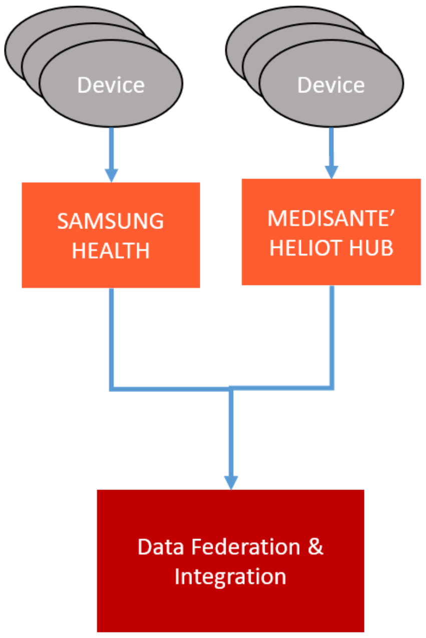


Figure 18 Greece pilot scenario

The Medisantè ELIOT Hub is a Cloud service able to collect and forward (**PUSH**) data to other systems. It can be configured in order to register the third part API to call for data forwarding.The Samsung Health based client, is an android based mobile app able to retrieve the sensors data from the Samsung Health Cloud and send (**PUSH**) such data to other systems.

About the output semantic model, the pilot aims at building final applications relying on state of the art HL7 standard (i.e. **FHIR**). It is expected the input data (both IoT and EHR) to be "harmonized" (i.e. converted) to such semantic model. At the moment of writing this deliverable it was not yet clear if the availability of data also in a graph DB (e.g. RDF4j, Neo4j), to fully exploit the semantic reasoning capabilities) is something useful for the pilot and the intelligent GK components (i.e. WP5). Finally, about the data location the pilot team has expressed the preference that acquired data, through the data federation, would be held in a **dedicated cloud cluster**.

In the following table, details are resumed the pilot specific requirements arising from the analysis above.

Table 4 Greece pilot sources

| Source type | Device Type | Gateway | DFI Interaction Modality | Output Semantic Model | Graph DB | Data Location |
|---|---|---|---|---|---|---|
| IOT | Biobeat wrist bands | Medisantè ELIOT Hub | PUSH | GK-FHIR Profile | N/A | Dedicated cluster |
|  | Biobeat chest patches |  |  |  |  |  |
|  | BP800 (BP, Glucose) |  |  |  |  |  |
|  | BC800 (body weight and composition) |  |  |  |  |  |
|  | Samsung smartwatch (HR, physical activity, sleep, stress level) | Samsung Health and Bixby capsules | PUSH | GK-FHIR Profile | N/A | Dedicated cluster |

- *Basque country*

The Basque country scenario is not yet defined so it was not possible to collect needed information about the devices and the gateway that they want to use. All the information reported in this section is a hypothesis of the possible scenario deduced from the architecture described in the deliverable 3.1. Details of the final scenario will be provided in version two of the deliverable.

Probably it is expected the involvement of two intermediary collecting services (Medisantè and Eliot Samsung Health) in order to gather data from several IoT devices.

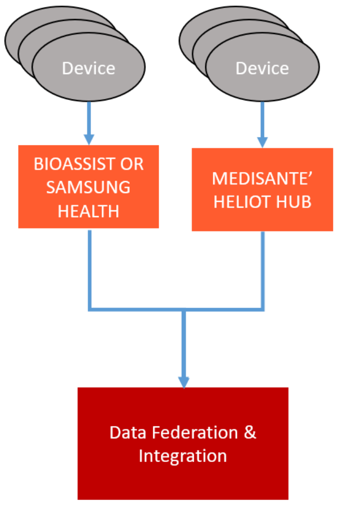


Figure 19 Basque Country pilot scenario

Data coming from Samsung devices are collected by Samsung Health and forwarded (**PUSH**) to Data Federation. The Medisantè Eliot Hub is a Cloud service able to collect and forward (**PUSH**) data to other systems.

In the following table, details are reported about the device name, intermediate gateway involved and main expected interaction modality. The list of devices is not yet defined.

Table 5 Basque Country pilot sources

| Source type | Device Type | Gateway | DFI Interaction Modality | Output Semantic Model | Graph DB | Data Location |
|---|---|---|---|---|---|---|
| **IOT** | Not yet defined | Medisantè ELIOT Hub | Probably PUSH | Information not provided | N/A | Information not provided |
| | Not yet define | Samsung Health | Probably PUSH | | | |

At the moment of writing this deliverable it was not yet clear if the availability of data also in a graph DB (e.g. RDF4j, Neo4j), to fully exploit the semantic reasoning.

Finally, about the data location the pilot team no information has yet expressed about the place where data should be held. Version two of this deliverable will provide missing information.

## - *Cyprus*

In the Cyprus pilot, several external systems are involved as described during dedicated phc and also reported in the pilot specific architecture (deliverable D3.1). More in detail, it is expected the involvement of an intermediary collecting services (Samsung Health) in order to gather data from several IoT devices.

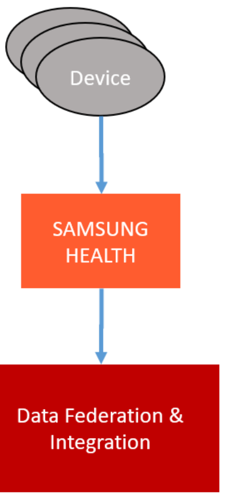


Figure 20 Cyprus pilot scenario

The Samsung Health based client is an android mobile app able to retrieve the sensors data from the Samsung Health Cloud and send (**PUSH**) such data to other systems.

About the output semantic model, the pilot aims at building final applications relying on the state of the art HL7 standard (i.e. **FHIR**). It is expected the input data to be "harmonized" (i.e. converted) to such semantic model. At the moment of writing this deliverable it was not yet clear if the availability of data also in a graph DB (e.g. RDF4j, Neo4j), to fully exploit the semantic reasoning capabilities is something useful for the pilot and the intelligent GK components (i.e. WP5). Finally, about the data location the pilot team has expressed the preference that acquired data, through the DFI, would be held in a **dedicated cloud cluster**.

In the following table, details are resumed the pilot specific requirements arising from the analysis above.

Table 6 Cyprus pilot sources

| Source type | Device Type | Gateway | DFI Interaction Modality | Output Semantic Model | Graph DB | Data Location |
|---|---|---|---|---|---|---|
| **IOT** | Samsung smartwatch (HR, physical activity, sleep, stress level) | Samsung Health and Bixby capsules | PUSH | GK-FHIR Profile | N/A | Dedicated cluster |

## - *Poland*

The scenario of the Poland pilot is not yet defined. The analysis of the draft architecture proposed in deliverable 3.1 does not provide any significant information about the gateway that will be used, their interaction modality (PUSH or PULL) with DFI, the output of sematic model and the preference about the location of acquired data. This information will be provided in the version two of this deliverable.

## - *UK*

In the GK pilot, several external systems are involved as described during dedicated phc and also reported in the pilot specific architecture (deliverable D3.1). More in detail it is expected the involvement of the intermediary collecting services Samsung Health in order to gather data from several **IoT** devices, smartphones and the Robotic platform Human activities.

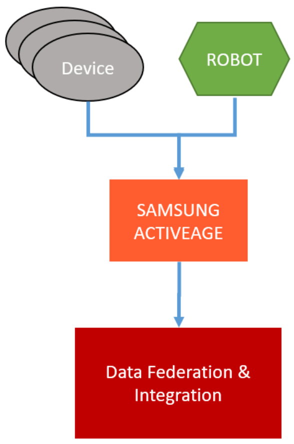


Figure 21 UK pilot scenario

The Samsung Health based client, is an android based mobile app able to retrieve the sensors data from the Samsung Health Cloud and send (**PUSH**) such data to other systems. The Samsung Health Cloud is used also to retrieve data coming from the robot platform, by means a robot event logger module, and forward (PUSH) such data to other systems (in the case of GateKeeper to DataFederation & Integration module).

At the moment of writing of this deliverable the pilot has not any preference about the output semantic model even if they are very interested to adopt HL7 **FHIR** standard, if the information that they manage can be modelled on such standard, so most likely **FHIR** will be the final model that will be used. If this will not be possible, data can be provided in a graph DB (e.g. RDF4j, Neo4j), to fully exploit the semantic reasoning capabilities that is something useful for the pilot and the intelligent GK components (i.e. WP5).

Finally, about the data location, the pilot team has not expressed any preference that acquired data, through the Data federation, would be held or not in a dedicated cloud cluster.

In the following table, details are resumed the pilot specific requirements arising from the analysis above.

Table 7 UK pilot sources

| Source type | Device Type | Gateway | DFI Interaction Modality | Output Semantic Model | Graph DB | Data Location |
|---|---|---|---|---|---|---|
| **IOT** | Samsung smartwatch (HR, physical activity, sleep, stress level) | Samsung Health and Bixby capsules | PUSH | GK-FHIR Profile | N/A | No preference about the use of a Dedicated cluster |
| | Robot platform Human activities Environment data remote control | | | GK-FHIR Profile (if is possible to map the information that Robot provide to FHIR) | Only if with FHIR is not possible to manage shared data | No preference about the use of a Dedicated cluster |

For the sake of simplicity, all the requirements gathered above have been resumed in the table below (Table 8).

Table 8 Pilots requirements

| | Puglia | Saxony | Greece | Aragon | UK | Cyprus | Poland | Basque country |
|---|---|---|---|---|---|---|---|---|
| **Data acquisition modality: PUSH** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | Details expected by the second version | Details expected by the second version |
| **Data acquisition modality: PULL** | | | | | | | | |
| **External system: IoT** | ✓ | ✓ | ✓ | N/A | ✓ | ✓ | | |
| **External system: EHR** | ✓ | N/A | N/A | ✓ | N/A | N/A | | |
| **Output semantic model: FHIR** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| **Output semantic model: SAREF/OTHER** | N/A | N/A | N/A | N/A | N/A | N/A | | |
| **Data availability in a graph DB** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| **Dedicated data repository** | ✓ | ✓ | ✓ | ✓ | No preference | ✓ | | |

## 2.3 Data Federation & Integration design

### 2.3.1 Architecture

Data Federation and Integration aims to integrate data coming from a different and heterogeneous data source in a common selected data model harmonizing their representation in order to create a single view of such data that can be accessed from external applications, i.e. from "Thing" developed in the scope of work package 5.

Figure 22 show a general overview of such component, it is able to accept data coming from a different source (i.e. devices, sensors, electronic health records, and so on) in a different format (json, xml, etc.) and store them in a common repository. It provides some APIs to allow this integration. The selected ontologies are HL7-FHIR v4 and SAREF. Data can be retrieved in FHIR and RDF format.

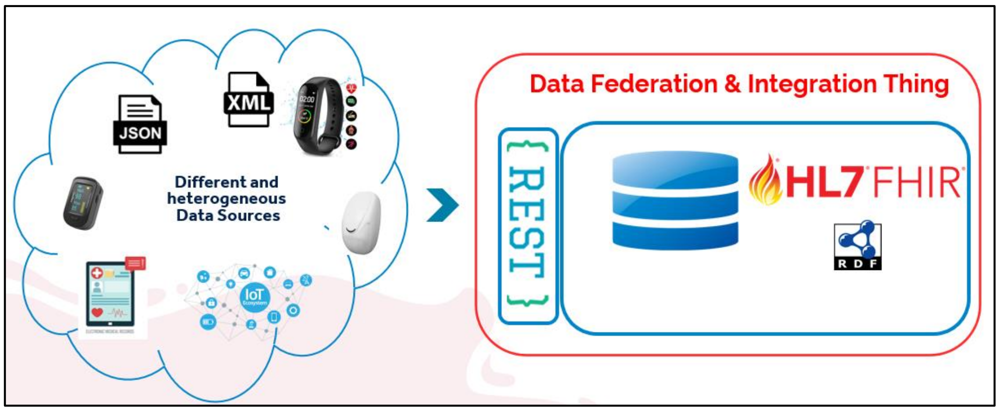

Different and heterogeneous Data Sources
JSON
XML
IoT
Data Federation & Integration Thing
{REST}
HL7 FHIR
RDF

Figure 22 Data Federation & Integration Thing – Overview

DFI offers a utility to harmonize data against the GateKeeper defined FHIR Profile coming from task 3.5. In details:

- It offers REST APIs (southbound) to acquire data from IOT/EHR data source to GK-FHIR Profile compliant data.
- It offers REST API (northbound) to access the converted data for immediate integration in external component or application.

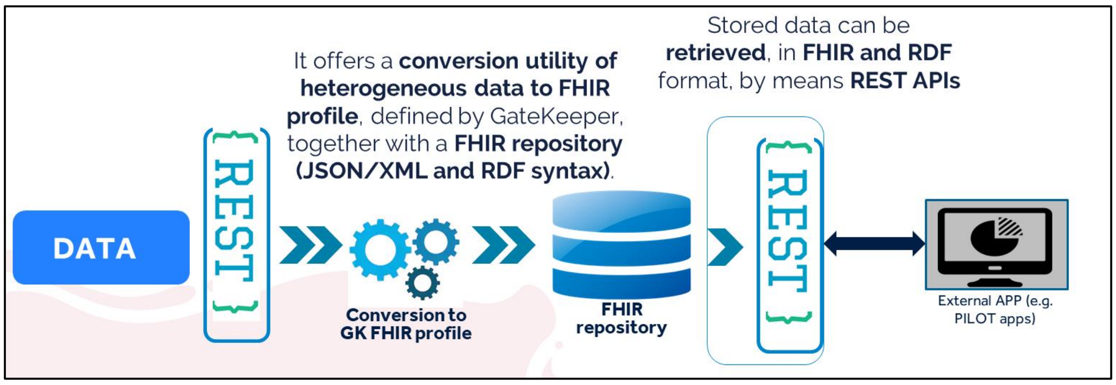

It offers a **conversion utility of heterogeneous data to FHIR profile**, defined by GateKeeper, together with a **FHIR repository (JSON/XML and RDF syntax)**.
Stored data can be **retrieved**, in **FHIR and RDF** format, by means **REST APIs**
DATA
{REST}
Conversion to GK FHIR profile
FHIR repository
{REST}
External APP (e.g. PILOT apps)

Figure 23 Data Federation & Integration pipeline

As shown in Figure 23 DFI provides some REST APIs that are able to accept data coming from different sources. Collected data are converted in FHIR and RDF representation, by a set of conversion routines, and persisted in a common repository. Stored data can be retrieved, in FHIR and RDF format, by means REST APIs

The architecture of Data Federation & Integration consists of three main components gk-Integration-engine, gk-fhir-server and gk-rdf4j. Logically it is a composition of three "Things" each one providing its TD (Thing Descriptor) as shown in Figure 24 Data Federation & Integration Thing. Such TDs describe the three distinct APIs exposed by this component.

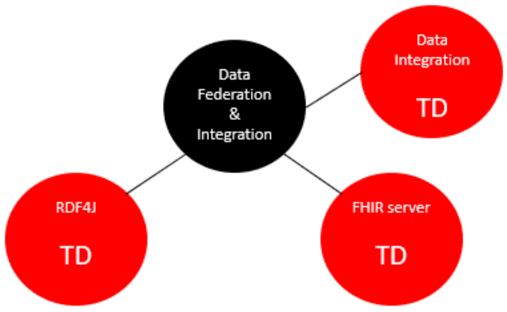


Figure 24 Data Federation & Integration Thing

The internal components are:

- gk-ingration-engine
- gk-fhir-server
- gk-rdf4j
- keycloak

each one with specific features and responsibilities.

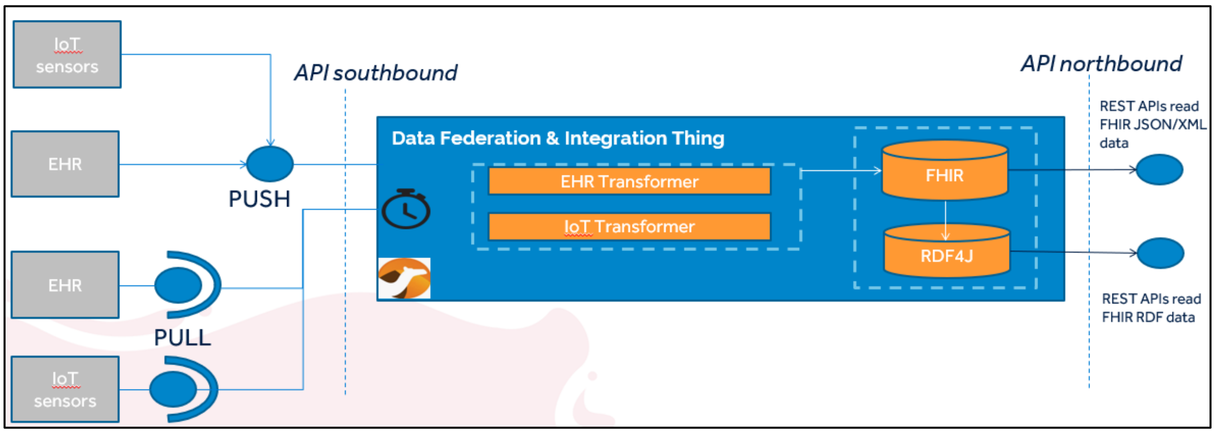


Figure 25 How to use Data Federation & Integration Thing

Gk-integration-engine provides the southbound APIs to receive raw data from external data sources, acquired data are converted to FHIR/RDF representation according to preload conversion rules, finally converted data are sent to gk-fhir-server and rdf4j through of the APIs that they provide.

This component offers two kinds of modalities for the interaction called PUSH and PULL. With the modality PUSH external applications have to invoke the gk-integration-engine APIs in order to send data to DFI instead with PULL modality the DFI invoke APIs offered by external applications in order to share data with DFI. With the latter modality it is clear the external application should provide the APIs that can be called by the DFI.

Gk-fhir-server is a web server compliant to FHIR standard that provides the set of operations to retrieve, store, update and delete FHIR Resources. Data are persisted in a dedicated repository. It offers a set of northbound APIs that can be invoked by external application to retrieve persisted information according to FHIR specification in JSON and XML format.

Gk-rdf4j provides a set of APIs to store, update and retrieve data in RDF format offering a set of utilities to execute SPARKQL queries. It has a dedicated repository where data are stored in RDF representation.

All these modules are described in detail in a dedicate section.

The last component is Keycloak [9]. It is an open-source software product to allow single sign-on with Identity and Access Management aimed at modern applications and services. This component has been installed and configured to simulate the behaviour of GTA (task 4.5) in order to enable the access to southbound APIs only to the authorized applications. An external application that wants to share their data with DFI has to involve a specific keycloak API to receive the access token and pass it to the DFI when its APIs are invoked.

Figure 26 shows the steps followed by Data Federation & Integration to persist data coming from an external application using the PUSH modality. The pilot application that wants to send data to DFI asks the access token to keycloak module passing the client_id, grant_type and client_secret. Keycloak verifies if the passed values are correct and returns an access token. The pilot application makes a request towards the gk-integration-engine passing the received token (by keycloak) and raw data that wants to persist into DFI repository. gk-integration-engine, based on pilot name, selects the right routine to convert custom raw data to FHIR standard according to the GK FHIR profile. Transformed FHIR data are sent to gk-fhir-server invoking the APIs that it provides. gk-fhir-server persists received data in dedicated FHIR repository, convert them to RDF format and sent such data to gk-rdf4j that persists them into its repository. A response message is returned.

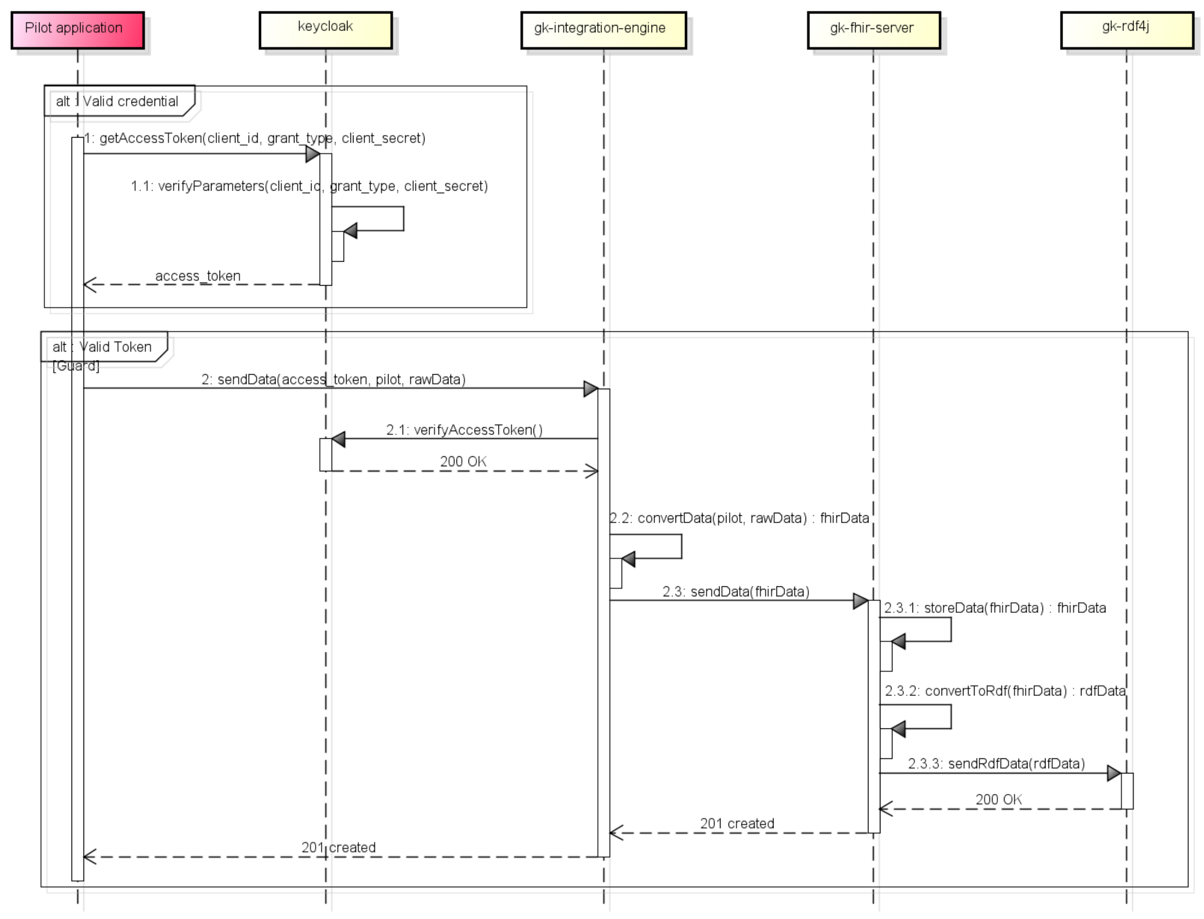

Pilot application
keycloak
gk-integration-engine
gk-fhir-server
gk-rdf4j
alt | Valid credential
1: getAccessToken(client_id, grant_type, client_secret)
1.1: verifyParameters(client_id, grant_type, client_secret)
access_token
alt | Valid Token
[Guard]
2: sendData(access_token, pilot, rawData)
2.1: verifyAccessToken()
200 OK
2.2: convertData(pilot, rawData) : fhirData
2.3: sendData(fhirData)
2.3.1: storeData(fhirData) : fhirData
2.3.2: convertToRdf(fhirData) : rdfData
2.3.3: sendRdfData(rdfData)
200 OK
201 created
201 created

Figure 26 Data Federation & Integration flow

This pipeline works only if the pilot application receives a valid token from keycloak module otherwise the DFI returns the HTTP 401 message (not authorized).

At the moment of writing this deliverable Data Federation & Integration is not fully integrated with TMS and GTA since the HPE infrastructure is not still available due to COVID-19 situation. When it will be available, DFI will be deployed and integrated with TMS and GTA components.

### 2.3.2 Adopted Semantic models

As already described above, Data Federation & Integration aims to integrate and harmonize data coming from heterogeneous data source, registered in GateKeeper platform, including EHR, wearable device data sources, home-based sensor data and sensor activity sensor data, thus, to producing a Health (semantic) interoperability repository enabling the development of advanced services to focus on scenarios and requirements provided by the pilots involved in the project.

Based on the analysis performed during the remote calls scheduled with pilots (also highlighted in Table 8) the main semantic model expected to be adopted is HL7-FHIR. Moreover, to ensure semantic interoperability, a controlled and shared vocabulary must be applied, also based on the use of appropriate terminologies. Such terminology models are built to meet the specific needs of a specific domain, where their nature is structured by vocabularies. Several terminological sources should be available to a community, in order to foster and ensure consistency between the data and information exchanged. Furthermore, the ability to provide coherent representations and the possibility of having access to a wide range of terminologies allows accelerating the interoperability process. Within clinical processes, medical terminology plays a very important role. In fact, it represents a central service for the provision of semantic interoperability between different systems and applications. In particular, appropriate terminology can be used to represent the information contained in clinical databases, data resulting from observations produced by qualified personnel in a specific domain, observations deriving from meetings with patients, as well as health guidelines, expert systems, and medical knowledge. In fact, terminologies provide a means to organize information and serve to define the semantics of the latter, using coherent mechanisms that can be computed by a machine. In addition, they are extensible, meaning that the data described by a particular collection of terms can, in turn, incrementally collect additional terms, which will then be reclassified and re-indexed. Summarizing, therefore, the main purposes for which it is necessary to use standard terminologies concern the ability to provide consistent meaning, the need to promote shared understanding, the ability to facilitate communication with humans, the need to enable comparisons and data integration and the possibility of guaranteeing the portability and sharing of Electronic Health Records (EHR).

#### 2.3.2.1 Process to define GK HL7 FHIR Implementation Guide

In order to build a common semantic and integrated GK repository, the DFI framework has to know which resources of FHIR standard must be used together with the selected vocabularies to represent information coming from the several pilots' applications. To reach this goal inside the scope of the Gatekeeper project has been designed and applied a specific integration and interaction process that has involved tasks 3.4, 3.5 and 4.4.

Task 3.4, as documented by the relative deliverable, has prepared a template to collect data models and vocabularies used by pilots in their applications. Collected and filled templates are used by the task 3.5 to build a set of FHIR logical models continuously harmonized for considering the input progressively received by task 3.4. The output of task 3.5 is the GK-FHIR Profile and more, in general, the **GK HL7 FHIR implementation guide (IG)** that is used by task 4.4 to convert heterogeneous data coming from pilot application to the GK-FHIR data model. An HL7 FHIR implementation guide (IG) is "a set of rules about how FHIR resources are used (or should be used) to solve a particular problem, with associated documentation to support and clarify the usage[1]." A FHIR IG may include very different kinds of artefacts, as FHIR logical models, FHIR API conformance resource; FHIR profiles, and many other FHIR and non-FHIR artefacts. The focus of the Gatekeeper

[1] https://www.hl7.org/fhir/implementationguide.html

FHIR IG (Figure 27) is on the data space, thus logical models, profiles, terminologies, and their relationships are specified for GateKeeper

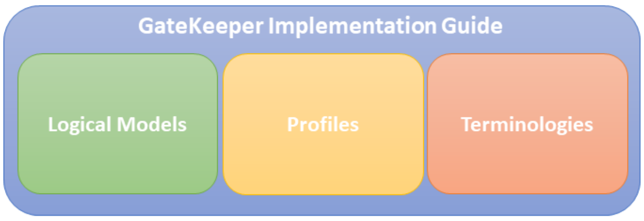


Figure 27 Gatekeeper Implementation Guide (from task 3.5)

Figure 28 shows the interaction process to define the GK Keeper data models, based on FHIR and the relative selected terminologies, that is used to persist and retrieve data from Data Federation & Integration modulo.

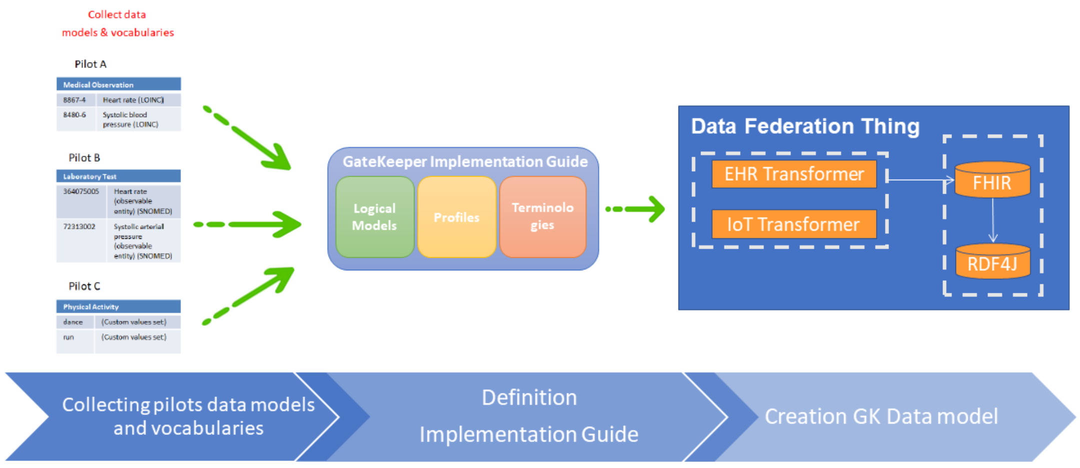


Figure 28 Gatekeeper Data Models definition process

At the moment of writing this deliverable, this is an ongoing activity, so it is not possible to specify a final representation. GK FHIR Implementation guide, more in detail profiles and vocabularies, is a very important input for Data Federation & Integration since it is used to build the conversion rules that are applied by transformers, when external applications invoke the Southbound APIs, to convert pilot data to GK-FHIR profile. Moreover, even if some pilot sends their data already in FHIR format, such data must be adapted and converted to GK FHIR profile in order to be harmonized with data coming from other pilots.

### 2.3.3 Declarative approach

One of the core key components of Data Federation & Integration is gk-integration-engine. This module exposes APIs that are able to acquire data from external heterogenous data sources and "harmonizing" such data to be compliant to GK-FHIR-Profile and the other IOT ontologies selected by the GK project. The heterogeneous data source can be both electronic health record system and IOT devices. Collected data by gk-integration-data must be converted, according specific rules, to GK-FHIR-Profile or some ontology and then sent to the FHIR or RDF repository. In order to perform these conversions, the gk-integration-engine contains a conversion utility that can work in two different approaches:

declarative approach and programmatic approach. This section is focus on the declarative approach.

Figure 29 shows the design data model for the Data Source and for the Converter. Data Source is an abstracts entity representing a generic source that can be specialized in two subclasses representing the concrete sources, EHR and IOT. EHR data source represents the electronic heath records containing data, for example about the clinical status of a person, generated by hospital or health care system while IOT data source represents data that are generated by IOT devices such smartwatch, sensors and so on. Such data differs from the one generated by health electronic health records due to the nature of the information that they manage. IOT device are used to perform some measurement, with a certain frequency, on a subject and forwards result to an application or gateway via Bluetooth.

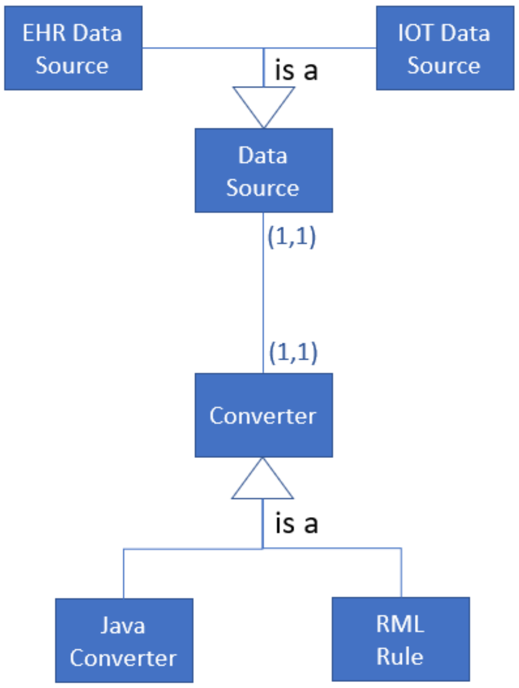


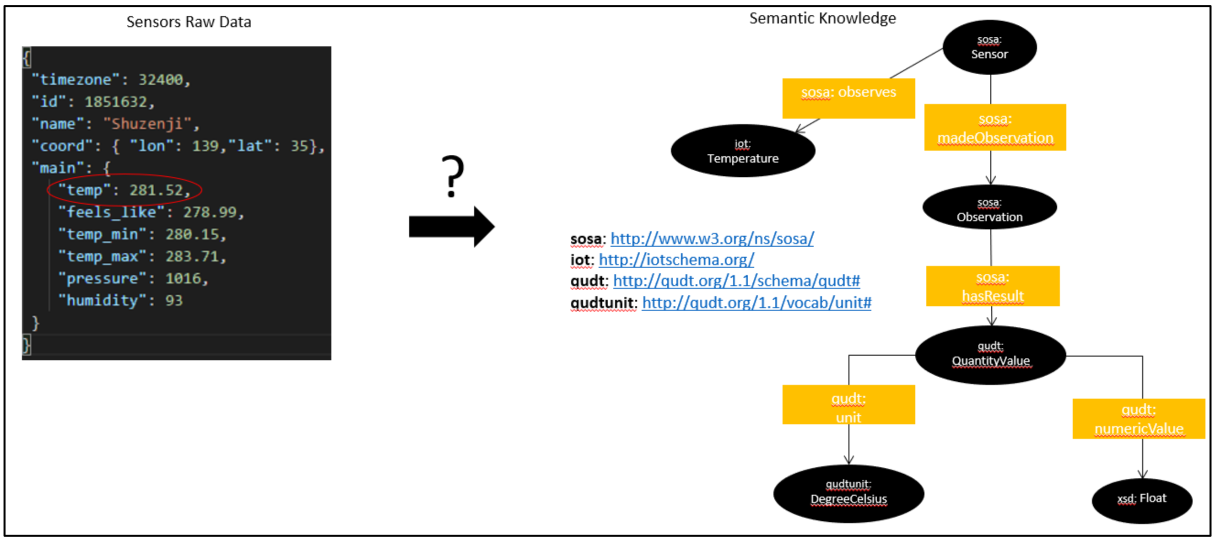
Figure 29 Data Source and Converter model

Each Data source is associated with a specific converter with a relation one to one, this design enforces to have a converter for each instance of data source. Converter, in similar way of data source, is abstract entity that can be specialized to a Java Convert o RML Rule. The RDF Mapping language (RML) [10] is a generic scalable mapping language defined to express rules that map data in heterogeneous structures and serializations to the RDF data model. RML deals with the mapping definitions in a uniform, modular, interoperable and extensible fashion. RML is defined as a superset of the W3C-recommended mapping language, R2RML, that maps data in relational databases to RDF. In RML, the mapping of data to the RDF data model is based on one or more Triples Maps that defines how the triples will be generated. A Triples Map defines rules to generate zero or more RDF triples sharing the same subject. A Triples Map consists of a Logical Source, a Subject Map and zero or more Predicate-Object Maps:

- A Logical Source consists of (i) a reference to input source(s), (ii) the Reference Formulation to specify how to refer to the data and (iii) the iterator that specifies how to iterate over the data. The following reference formulations are predefined but not limited: ql:CSV, ql:CSS3, ql:JSONPath, rr:SQL2008 and ql:XPath.

- The Subject Map consists of the URI pattern that defines how each triple's subject is generated and optionally its type. The references to the input data occurs using valid references according to reference formulation specified at the Logical Source.
- Triples are generated using Predicate Object Maps. A Predicate Object Map consists of Predicate and an Object Map(s). A Predicate Map specifies how the triple's predicate is generated. An Object Map specifies how the triple's object(s) are generated.

The output of RML is a sematic knowledge that can be persisted in sematic repository. An example is provided to show how RML rules can be written to produce a sematic representation raw data.

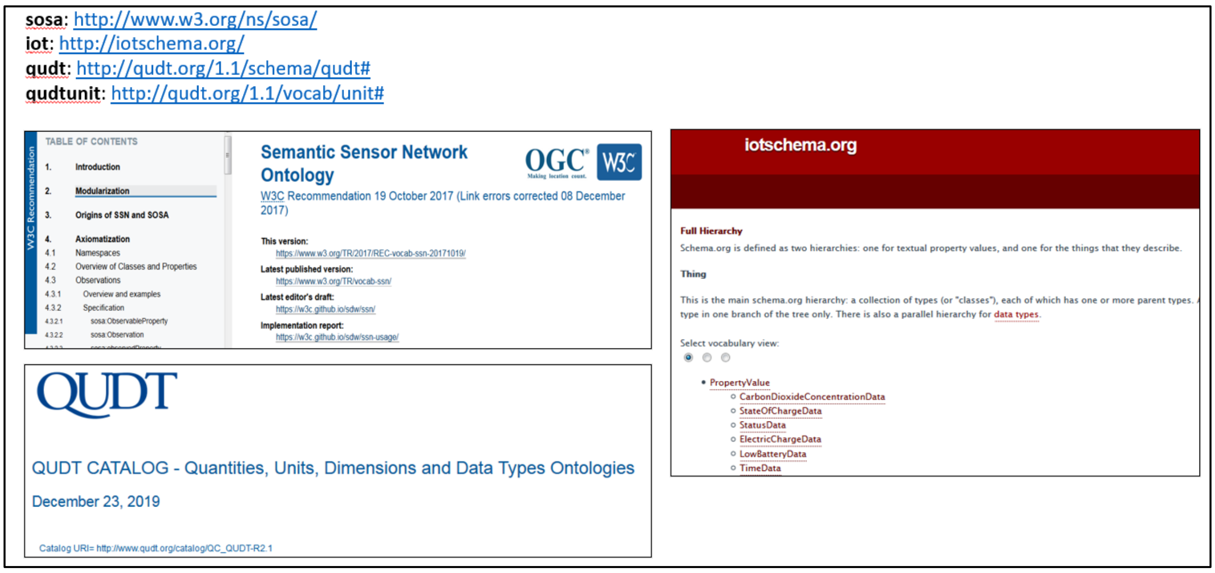


```
{
 "timezone": 32400,
 "id": 1851632,
 "name": "Shuzenji",
 "coord": { "lon": 139,"lat": 35},
 "main": {
    "temp": 281.52,
    "feels_like": 278.99,
    "temp_min": 280.15,
    "temp_max": 283.71,
    "pressure": 1016,
    "humidity": 93
 }
}
```

Figure 30 Sensor raw data to Semantic knowledge

On the left of the Figure 30 there an example of raw representation of weather temperature in JSON forma containing information about time zone, sensorID, name of the city where data are related on (Shuzenjii), coordinates (longitude and latitude), external temperature, minimum and maximum temperature, pressure, and humidity. The goal is to convent the json representation of temperature in a semantic model by mean RML declarative rules language. A preliminary activity to perform this task is to select the ontologies that has to be used for the semantic representation (as already described, an ontology is a formal representation model of the reality and knowledge). It is a data structure that allows the description of the entities (objects, concepts, etc.) and their relationship in a specific knowledge domain. An ontology is the explicit formal description of the concepts of a domain, that is, a model that allows to represent reality (being) in the domain in question, in the form of a set of objects and relations (class of objects).

The ontologies selected to represent the temperature data of the example are sosa[2] and iot[3] for the iot schema, qudt[4] for the representation of a quantity and qutunit[5] for the represent of the unit of measure (Figure 31).



**sosa**: http://www.w3.org/ns/sosa/
**iot**: http://iotschema.org/
**qudt**: http://qudt.org/1.1/schema/qudt#
**qudtunit**: http://qudt.org/1.1/vocab/unit#

W3C Recommendation

TABLE OF CONTENTS

1. Introduction
2. Modularization
3. Origins of SSN and SOSA
4. Axiomatization
4.1 Namespaces
4.2 Overview of Classes and Properties
4.3 Observations
4.3.1 Overview and examples
4.3.2 Specification
4.3.2.1 sosa:ObservableProperty
4.3.2.2 sosa:Observation

Semantic Sensor Network Ontology

W3C Recommendation 19 October 2017 (Link errors corrected 08 December 2017)

**This version:**
https://www.w3.org/TR/2017/REC-vocab-ssn-20171019/
**Latest published version:**
https://www.w3.org/TR/vocab-ssn/
**Latest editor's draft:**
https://w3c.github.io/sdw/ssn/
**Implementation report:**
https://w3c.github.io/sdw/ssn-usage/

QUDT

QUDT CATALOG - Quantities, Units, Dimensions and Data Types Ontologies

December 23, 2019

Catalog URI= http://www.qudt.org/catalog/QC_QUDT-R2.1

iotschema.org

**Full Hierarchy**

Schema.org is defined as two hierarchies: one for textual property values, and one for the things that they describe.

**Thing**

This is the main schema.org hierarchy: a collection of types (or "classes"), each of which has one or more parent types.
type in one branch of the tree only. There is also a parallel hierarchy for data types.

Select vocabulary view:
● ○ ○

- PropertyValue
  - CarbonDioxideConcentrationData
  - StateOfChargeData
  - StatusData
  - ElectricChargeData
  - LowBatteryData
  - TimeData

Figure 31 Sensors ontologies

The right side of the Figure 30 shows the desired target semantic knowledge in a graph representation of the temperature, by means the selected terminologies, represented in the right side of the figure in json format.
Black balls represent concepts of the selected terminologies while yellow rectangles represent relations among the different concepts. Arrows are the navigability directions. In detail the device is a sensor (*sosa: Sensor*) that observes (*sosa:observes*) a temperature (*iot:Temperature*). it makes an observation (*sosa:madeObservation*) and the result is an

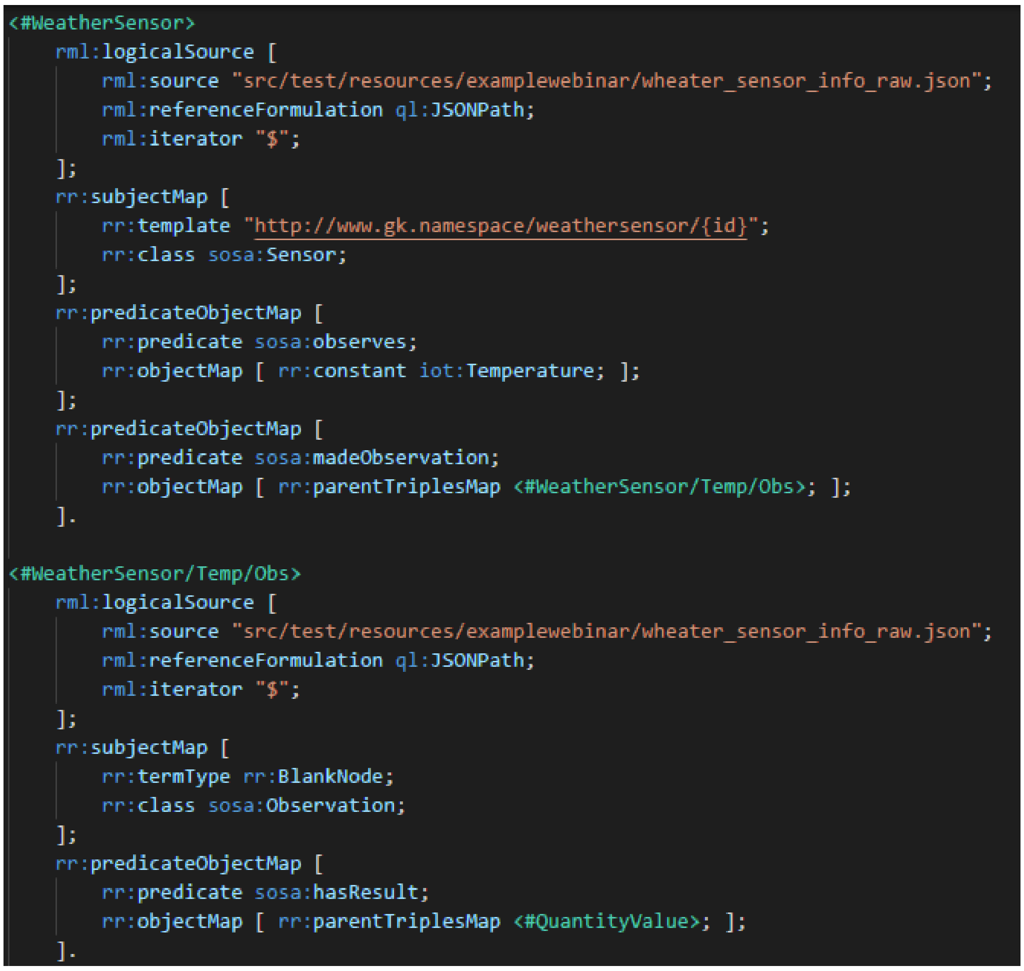
[2] http://www.w3.org/ns/sosa/

[3] http://iotschema.org/

[4] http://qudt.org/1.1/schema/qudt#

[5] http://qudt.org/1.1/vocab/unit#

observation (*sosa:Observation*) that has a result (*sosa:hasResult*) of type quantity (*qudt:QuantityValue*). Quantity consists of a numeric value (*qudt:numericValue*) of type float (xsd:Float) and a unit of measure (*qudt:unit*) of type degree Celsius (*qutunit:DegreeCelsius*).
By means RML, it is possible to write RML rules that analysed the JSON raw data and provides as output the sematic model described above. Figure 32 shows the rules in RML syntax to produce the semantic model representation.



```
<#WeatherSensor>
    rml:logicalSource [
        rml:source "src/test/resources/examplewebinar/wheater_sensor_info_raw.json";
        rml:referenceFormulation ql:JSONPath;
        rml:iterator "$";
    ];
    rr:subjectMap [
        rr:template "http://www.gk.namespace/weathersensor/{id}";
        rr:class sosa:Sensor;
    ];
    rr:predicateObjectMap [
        rr:predicate sosa:observes;
        rr:objectMap [ rr:constant iot:Temperature; ];
    ];
    rr:predicateObjectMap [
        rr:predicate sosa:madeObservation;
        rr:objectMap [ rr:parentTriplesMap <#WeatherSensor/Temp/Obs>; ];
    ].

<#WeatherSensor/Temp/Obs>
    rml:logicalSource [
        rml:source "src/test/resources/examplewebinar/wheater_sensor_info_raw.json";
        rml:referenceFormulation ql:JSONPath;
        rml:iterator "$";
    ];
    rr:subjectMap [
        rr:termType rr:BlankNode;
        rr:class sosa:Observation;
    ];
    rr:predicateObjectMap [
        rr:predicate sosa:hasResult;
        rr:objectMap [ rr:parentTriplesMap <#QuantityValue>; ];
    ].
```

Figure 32 Example of RML rule specification for a sensor raw data

Data Federation & Integration includes a conversion utility that takes in input a raw format (in the case of the example JSON) and the relative RML rules and provides as output the semantic representation, in RDF format, of the source raw data. Rules must be written according the source raw input and the selected terminologies. Output of the application of the RML rules for the JSON representing temperature is showed in the Figure 33.

```
@prefix iot: <http://iotschema.org/> .
@prefix qudt: <http://qudt.org/1.1/schema/qudt#> .
@prefix qudtunit: <http://qudt.org/1.1/vocab/unit#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://www.gk.namespace/weathersensor/1851632> a sosa:Sensor;
  sosa:madeObservation _:0 .

_:0 a sosa:Observation;
  sosa:hasResult _:1 .

_:1 a qudt:QuantityValue;
  qudt:numericValue "281.52"^^xsd:float;
  qudt:unit qudtunit:DegreeCelsius .

<http://www.gk.namespace/weathersensor/1851632> sosa:observes iot:Temperature .
```

Figure 33 Temperature semantic representation in RDF format

As it is possible to see from the figure, it is the same presentation of the graph showed in Figure 30. It is defined a prefix for each selected URL terminology, this means that each terminology can be referred by the prefix without to use the whole URl, this syntax improves the readability of the code. Device with id 1851632 is a sosa Sensor that made a sosa Observation with id *_:0*. Observation with id _:0 is a sosa:Observation that has a quantity value result referred by id *_:1*. QuantityValue consists of a numeric value and a unit. The value of numericValue is 281.52 of type float while the quantity unit is degree Celsius. Last line of code says that the sensor having id 1851632 observes a temperature.
From the view of the model represented by Figure 29 an instance of Converter class is a file containing RML rules that are able to convert data in a semantic representation associated to a specific source that can be an HER data source or IOT data source.

### 2.3.4Programmatic approach

Previous section describes the specialization of Converter class in RML rules while this section describes the converter when it is specialized in a Java routine. As already said to each data source can be associated one Converter that can be a Java Convert or a RML Rule. Figure 34 shows data model of the Java Converter offering the possibility to add a new transformation Java class to a specific data resource. New class will be added by hand in the actual release of the Data Federation & Integration.

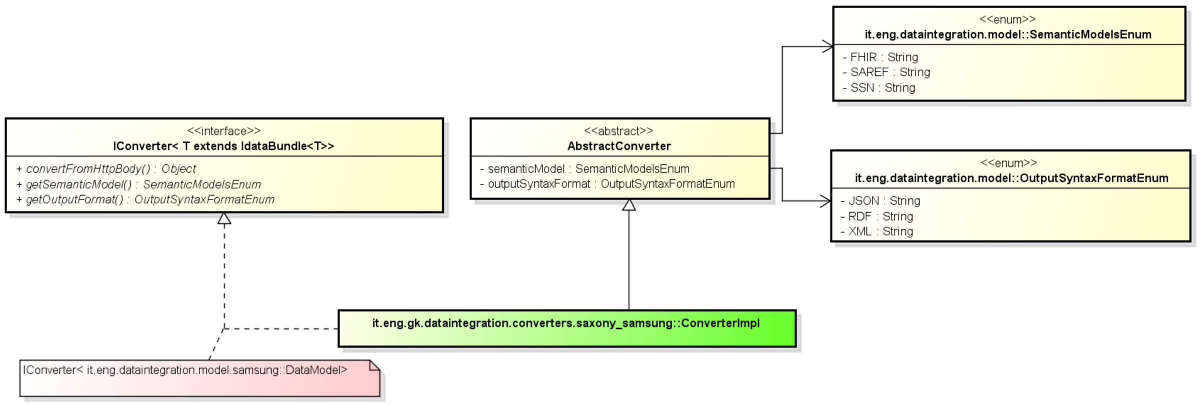

<<enum>>
it.eng.dataintegration.model::SemanticModelsEnum
- FHIR : String
- SAREF : String
- SSN : String
<<interface>>
IConverter< T extends IdataBundle<T>>
+ convertFromHttpBody() : Object
+ getSemanticModel() : SemanticModelsEnum
+ getOutputFormat() : OutputSyntaxFormatEnum
<<abstract>>
AbstractConverter
- semanticModel : SemanticModelsEnum
- outputSyntaxFormat : OutputSyntaxFormatEnum
<<enum>>
it.eng.dataintegration.model::OutputSyntaxFormatEnum
- JSON : String
- RDF : String
- XML : String
it.eng.gk.dataintegration.converters.saxony_samsung::ConverterImpl
IConverter< it.eng.dataintegration.model.samsung::DataModel>

Figure 34 Java Converter Model

Java Converter model consists of an abstract class named *AbstractConverter* that contains two attribute *semanticModel* of type *SematicModelsEnum* and outputSyntaxFormat of type *OutputSyntaxFormatEnum*. Attribute *semanticModel* represents the output format ontology that can be FHIR, SAREF and SSN while attribute *outputSyntaxFormat* represents the format of the output data that can be JSON, RDF or XML. When a new converter for a specific data source, belonging to a specific pilot, the concrete implementation of the class *AbstractConverter* must be provided inside new package containing the name of the pilot followed by the name of the application that is used. Figure 34 shows an instance of implemented converter for data collected by Samsung gateway (class in green color) that is used by Saxony pilot.
Each *ConverterImpl* class must selected both the output data format (JSON, XML, RDF) and adopted ontology (FHIR, SAREF and SSN) and it has to implement methods of the interface *IConverter<T extends IDataBundle<T>>* together with the DataModel representing the Java beans of the incoming data. *IConverter<T extends IDataBundle<T>>* offers three methods:

- convertFromHttpBody() this method contains the logic to unmarshal data from string to Java Object.
- *getSemanticModel()* that returns the selected semantic model.
- *getOutputFormat()* that returns the format of output model.

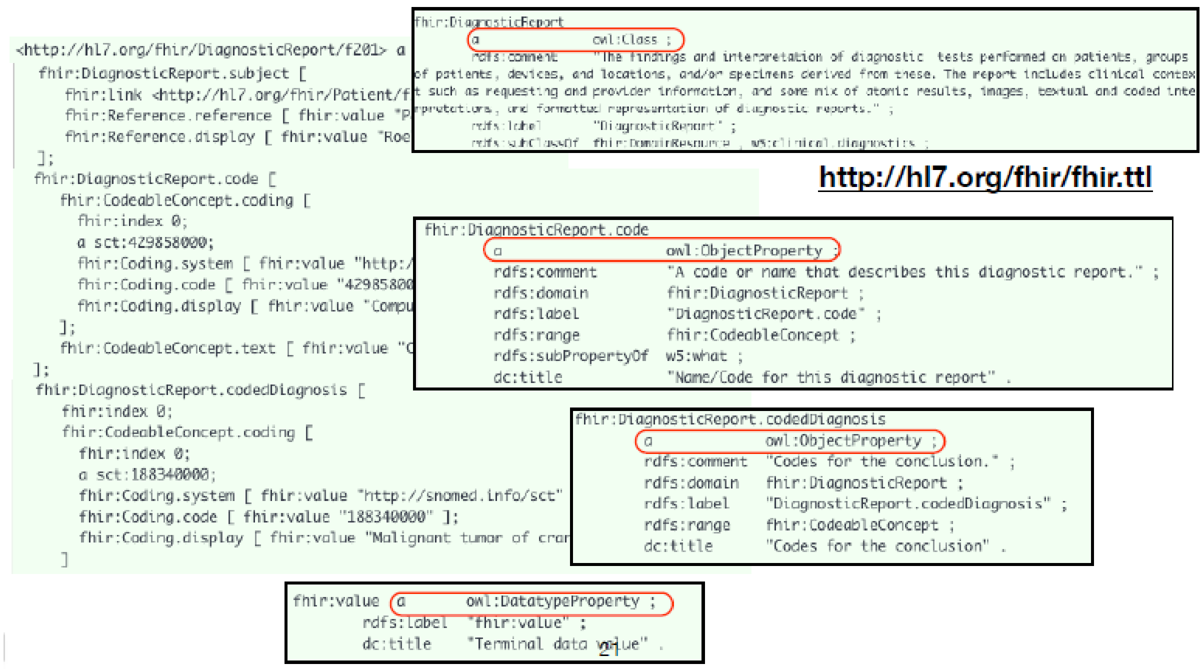
Each *DataModel* must implements the interface *IDataBundle<T>*.
To facilitate the implementation of a new converter, an ECLIPSE sample project has been provided together with a guide containing the instruction to implement a new Java converter; reading such guide and modifying the ECLIPSE sample project it is possible to develop and plug a new Java Converter in easy way.
The complete guide is reported in Appendix A.

### 2.3.5 Persistence and semantic reasoning

It is fair to point out that the main ontology (i.e. semantic model) that will be used by the GK project is HL7 FHIR (see also Table 8). This conclusion is the result of several remote calls made with pilots and the components owners involved in the project; moreover, the major AI frameworks and pilots' applications are interested to manage data in FHIR format and using json/xml syntax. For this reason, DFI is expected to mainly adapt incoming data to the HL7 FHIR semantic model and persist them in (XML/JSON) format in a native FHIR server.
However, it is worth to mention that the DFI design has been conceived to represent *a more flexible solution* able to manage also the harmonization of data against several semantic models (not only HL7 FHIR) relying on the more general RDF syntax and their routing to a graph DB (i.e. the semantic data lake) implemented through a RDF4J (although other graph DB could be adopted). The rationale of this flexibility is due to desire to offer the possibility, to the reasoning software layer or even to the final user applications, to access a graph DB (RDF based) and exploit such a way the whole potential of the semantic reasoning briefly highlighted with an example in the section below.

#### 2.3.5.1 FHIR RDF and DL Reasoner

FHIR is itself an Ontology so that each FHIR resource exploits the concepts (i.e. Class) and relationships (i.e. properties) defined in such ontology. The ontology has been defined and is publicly available (http://hl7.org/fhir/fhir.ttl). In the figure below is possible to see an instance of the DiagnosticReport resource represented in RDF syntax and, in the rounded boxes, are showed the fhir.ttl ontology classes and properties involved.



```
<http://hl7.org/fhir/DiagnosticReport/f201> a
  fhir:DiagnosticReport.subject [
     fhir:link <http://hl7.org/fhir/Patient/f
     fhir:Reference.reference [ fhir:value "P
     fhir:Reference.display [ fhir:value "Roe
  ];
  fhir:DiagnosticReport.code [
     fhir:CodeableConcept.coding [
       fhir:index 0;
       a sct:429858000;
       fhir:Coding.system [ fhir:value "http:/
       fhir:Coding.code [ fhir:value "42985800
       fhir:Coding.display [ fhir:value "Compu
     ];
     fhir:CodeableConcept.text [ fhir:value "C
  ];
  fhir:DiagnosticReport.codedDiagnosis [
     fhir:index 0;
     fhir:CodeableConcept.coding [
       fhir:index 0;
       a sct:188340000;
       fhir:Coding.system [ fhir:value "http://snomed.info/sct"
       fhir:Coding.code [ fhir:value "188340000" ];
       fhir:Coding.display [ fhir:value "Malignant tumor of cran
     ]
```

```
fhir:DiagnosticReport
        a                 owl:Class ;
        rdfs:comment      "The findings and interpretation of diagnostic  tests performed on patients, groups
of patients, devices, and locations, and/or specimens derived from these. The report includes clinical contex
t such as requesting and provider information, and some mix of atomic results, images, textual and coded inte
rpretations, and formatted representation of diagnostic reports." ;
        rdfs:label        "DiagnosticReport" ;
        rdfs:subClassOf   fhir:DomainResource , w5:clinical.diagnostics ;
```

http://hl7.org/fhir/fhir.ttl

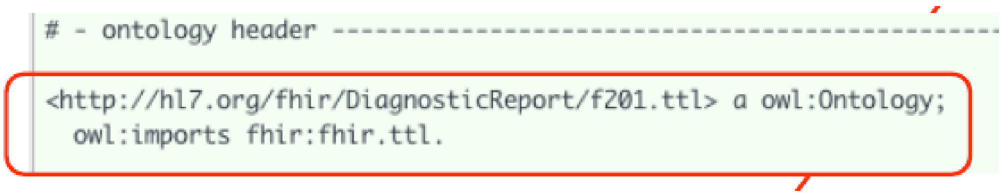
```
fhir:DiagnosticReport.code
        a                     owl:ObjectProperty ;
        rdfs:comment          "A code or name that describes this diagnostic report." ;
        rdfs:domain           fhir:DiagnosticReport ;
        rdfs:label            "DiagnosticReport.code" ;
        rdfs:range            fhir:CodeableConcept ;
        rdfs:subPropertyOf    w5:what ;
        dc:title              "Name/Code for this diagnostic report" .
```

```
fhir:DiagnosticReport.codedDiagnosis
        a                 owl:ObjectProperty ;
        rdfs:comment      "Codes for the conclusion." ;
        rdfs:domain       fhir:DiagnosticReport ;
        rdfs:label        "DiagnosticReport.codedDiagnosis" ;
        rdfs:range        fhir:CodeableConcept ;
        dc:title          "Codes for the conclusion" .
```

```
fhir:value  a          owl:DatatypeProperty ;
         rdfs:label    "fhir:value" ;
         dc:title      "Terminal data value" .
```

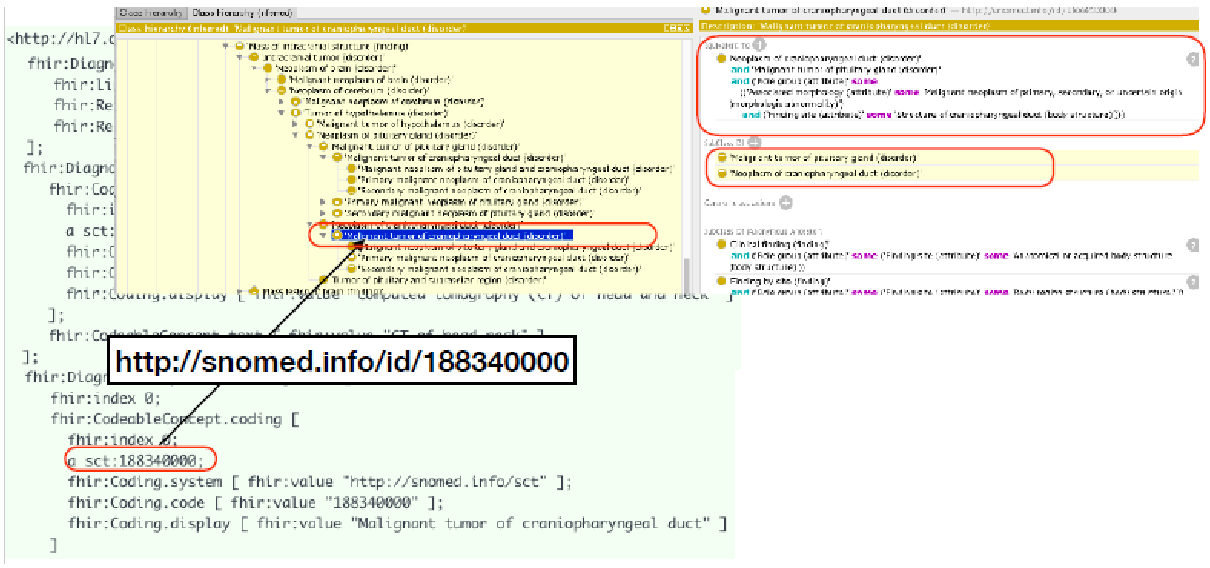
Figure 35 Instance description and FHIR Metadata Vocabulary

It’s important to observe that the resource instance has to declare the import of the FHIR Metadata Vocabulary (i.e. fhir.ttl) so that an OWL framework (as Protegè for instance) correctly interpret the data.



```
# - ontology header ------------------------------------------

<http://hl7.org/fhir/DiagnosticReport/f201.ttl> a owl:Ontology;
  owl:imports fhir:fhir.ttl.
```

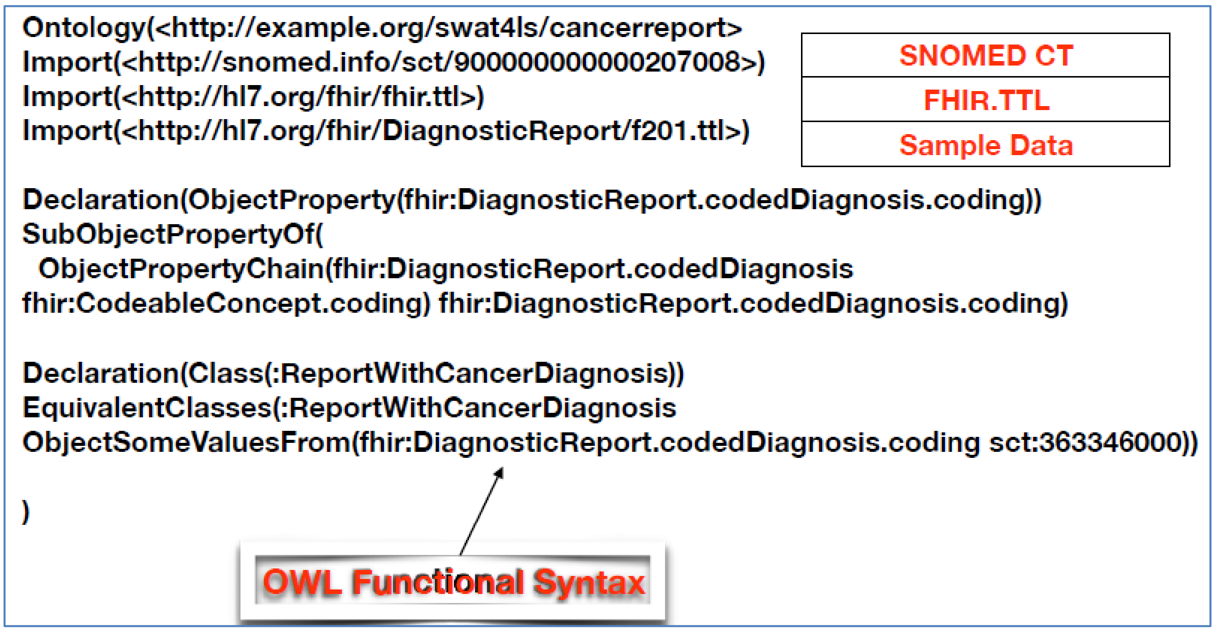
FHIR is a first level Ontology not describing everything but relying on specific coding systems to refer other concepts. So also, the Diagnosticreport instace showed above can refer other concepts (from other Ontologies – i.e. coding systems) to associate further semantics to its resources. For instance, in the figure below the SNOMED CODE: 188340000 is used, as example, to state that the DiagnosticReport is related to a “Malignant tumor of cranyopharingeal duct”. Snomed itself is in turn an Ontology that must be imported in order to allow the OWL tool to correctly “understand” the meaning of that code. In the figure below is showed either the usage of the code and the related position of that code in the SNOMED ontology opened using Protegè tool.



```
<http://hl7.c
  fhir:Diagn
     fhir:Li
     fhir:Re
     fhir:Re
  ];
  fhir:Diagno
     fhir:Coc
       fhir:i
       a sct:
       fhir:C
       fhir:C
       fhir:Coding.display [ fhir:value "Computed tomography (CT) of head and neck" ]
     ];
     fhir:CodeableConcept.text [ fhir:value "CT of head neck" ]
  ];
  fhir:Diagn
     fhir:index 0;
     fhir:CodeableConcept.coding [
       fhir:index 0;
       a sct:188340000;
       fhir:Coding.system [ fhir:value "http://snomed.info/sct" ];
       fhir:Coding.code [ fhir:value "188340000" ];
       fhir:Coding.display [ fhir:value "Malignant tumor of craniopharyngeal duct" ]
     ]
```

Figure 36 FHIR resource instance concept reference

So till now we have a FHIR resource instance (i.e. Diagnosticreport), defined by importing its metadata vocabulary (i.e. fhir.ttl) and we have showed how it can exploit an external concept by referencing an external ontology (i.e. SNOMED ontology). Now we are almost ready to trigger a semantic reasoning example. The last thing to do is to define some “classification rules” exploiting inner semantic reasoning capabilities (e.g. subsumption).


SNOMED CT
FHIR.TTL
Sample Data
OWL Functional Syntax

```
Ontology(<http://example.org/swat4ls/cancerreport>
Import(<http://snomed.info/sct/900000000000207008>)
Import(<http://hl7.org/fhir/fhir.ttl>)
Import(<http://hl7.org/fhir/DiagnosticReport/f201.ttl>)

Declaration(ObjectProperty(fhir:DiagnosticReport.codedDiagnosis.coding))
SubObjectPropertyOf(
 ObjectPropertyChain(fhir:DiagnosticReport.codedDiagnosis
fhir:CodeableConcept.coding) fhir:DiagnosticReport.codedDiagnosis.coding)

Declaration(Class(:ReportWithCancerDiagnosis))
EquivalentClasses(:ReportWithCancerDiagnosis
ObjectSomeValuesFrom(fhir:DiagnosticReport.codedDiagnosis.coding sct:363346000))

)
```

Figure 37 Classification Rule example

This classification rule states that a **ReportWithCanceDiagnosis** is a new ontology class (part of the Ontology cancerreport) that states its **equivalence** to those FHIR **DiagnosticReport** resources having diagnosis coding equal to SNOMED:363346000 (that refers concept of generic “Malignant neoplastic disease”).

It's important to point out that a semantic reasoning tool will be able to fire the matching not only when the code of a DiagnosticReport will be exactly the same but also if it will be a "subclass" (i.e. more specific). This is in what we have in our case since a "Malignant tumor of cranyopharingeal duct" (code: 188340000) – used in the Diagnosticreport - is a specific type of tumor while the code 363346000 – used for the ReportWithDiagnosticReport - refers to a generic "Malignant neoplastic disease". Now it is possible to exploit a reasoning framework as protegè (1) to load the FHIR resource instance (i.e. DiagnosticReport using SNOMED: 188340000) (2) to load the classification rule of Figure 37 included in a separated file (cancereport.owl), select an available reasoner (e.g. FaCT++) and trigger the exectution as showed in the figure below.

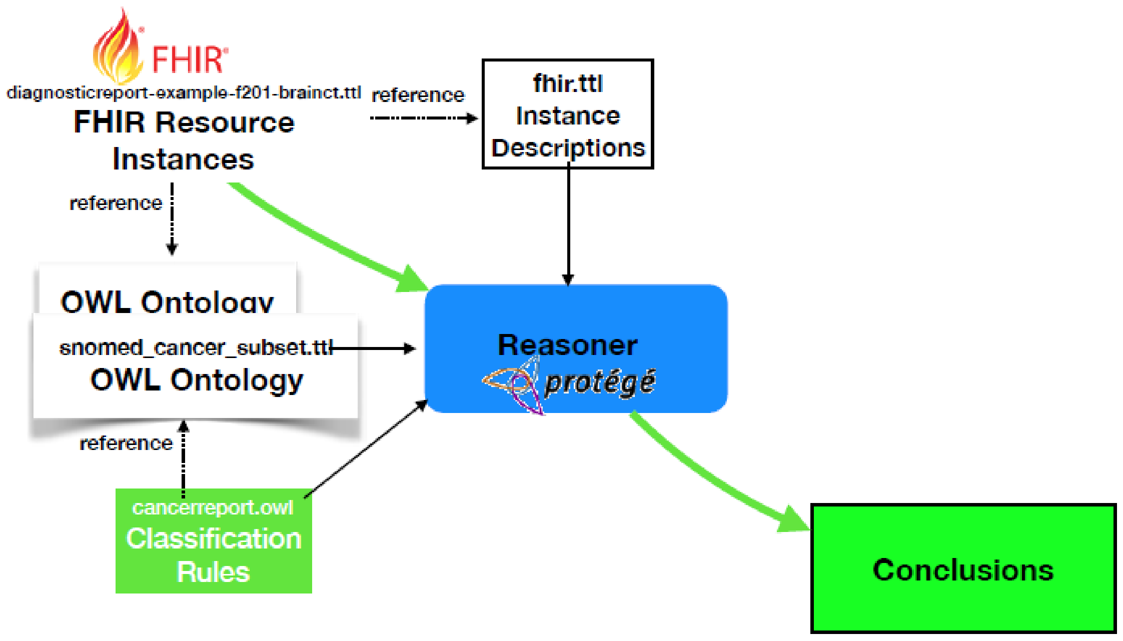
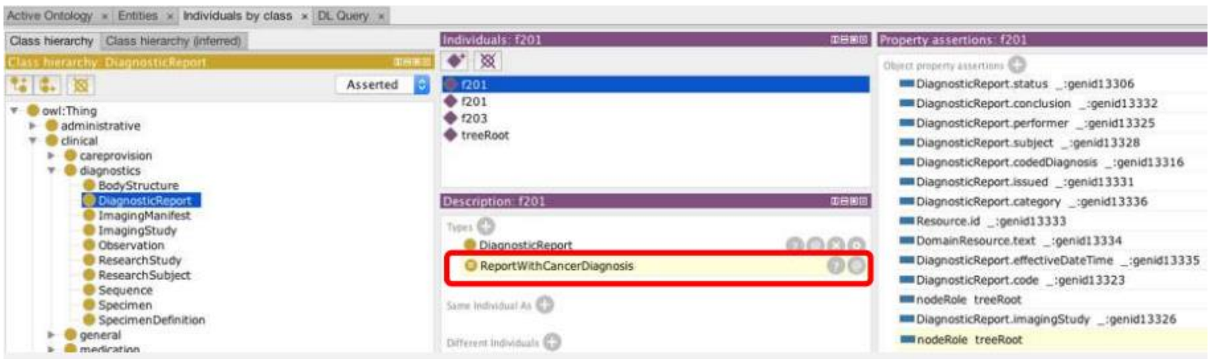


Figure 38 Using FHIR with a DL reasoner

As conclusion protegè will show how our DiagnosticReport has been recognised automatically as also an instance of the class ReportWithcancerDiagnosis



owl:Thing
administrative
clinical
careprovision
diagnostics
BodyStructure
DiagnosticReport
ImagingManifest
ImagingStudy
Observation
ResearchStudy
ResearchSubject
Sequence
Specimen
SpecimenDefinition
general
medication

f201
f201
f203
treeRoot

DiagnosticReport
ReportWithCancerDiagnosis

```
DiagnosticReport.status  _:genid13306
DiagnosticReport.conclusion  _:genid13332
DiagnosticReport.performer  _:genid13325
DiagnosticReport.subject  _:genid13328
DiagnosticReport.codedDiagnosis  _:genid13316
DiagnosticReport.issued  _:genid13331
DiagnosticReport.category  _:genid13336
Resource.id  _:genid13333
DomainResource.text  _:genid13334
DiagnosticReport.effectiveDateTime  _:genid13335
DiagnosticReport.code  _:genid13323
nodeRole  treeRoot
DiagnosticReport.imagingStudy  _:genid13326
nodeRole  treeRoot
```

Figure 39 Semantic reasoning result

#### 2.3.5.2 Persistence routing

Table 9 summarizes the storage options supported by DFI for each combination of converter type (JAVA, RML), selected semantic model, and output syntax.
For instance, if the converter type is JAVA, the selected semantic model is FHIR and the output syntax is the JSON/XML, then data are persisted and available in both FHIR Server and RDF4J. This configuration is supported in GK.
If the converter type is JAVA, the semantic model is FHIR and the output format is RDF, storage is available in RDF4J only.
If the converter type is JAVA, the sematic model is not FHIR and the output syntax is JSON/XML data cannot be stored in any format. This configuration is not support by GK.
If the converter type is JAVA, the semantic model is not FHIR and the output syntax is RDF then data will be held in RDF4J. This configuration is supported by GK.
If the converter type is RML, the semantic model is FHIR and the output syntax is RDF, storage is available in RDF4J. This configuration is supported by GK.
Finally, if the converter type is RML, the semantic model is not FHIR and the output syntax is RDF, storage is available only by RDF4J repository. This configuration is supported by GK.

Table 9 DFI supported converter types, semantic models, output syntax and storage

| Converter Type (JAVA/RML) | Semantic Model | Output Syntax | Storage | GK Support (YES/NO) |
|---|---|---|---|---|
| **JAVA** | FHIR | JSON/XML | FHIR Server and RDF4J | YES |
| **JAVA** | FHIR | RDF | RDF4J | YES |
| **JAVA** | NOT FHIR | JSON/XML | N/A | NO |
| **JAVA** | NOT FHIR | RDF | RDF4J | YES |
| **RML** | FHIR | RDF | RDF4J | YES |
| **RML** | NOT FHIR | RDF | RDF4J | YES |

Thanks to the above table it is possible to understand that data in DFI are always available in a graph DB (i.e. the semantic data lake) when the output syntax is RDF regardless of the type of selected converter type while data will be available also in the FHIR server only when the Converter is a JAVA class, the Sematic model is FHIR and the output syntax is JSON/XML.

## 2.4 Component interaction and integrations

Data Federation & Integration is naturally linked with other GK Thing. In the following sections are documented the interactions already tested till the moment of writing this deliverable. Naturally, other integration will be described in the next version of the document. In order to perform this kind of integration Data Federation & Integration is deployed on ENG server since, due to the COVID-19 situation, HPE had an important delay to provide the whole infrastructure where all components involved in GateKeeper platform should be deployed.

### 2.4.1 Interaction and integration with Medisantè IoT Connector

Eliot Hub is a IoT device platform to enable connectivity of medical devices with any clinical system. It simplifies deployment of telemonitoring at scale on the pilot side providing remote management capabilities. The platform only hosts non-identifiable patient data based on device number (e.g. IEMI) and pull data based on a direct-to-cloud approach into any target system used by the physicians – and patients. The platform relies currently on a limited set of medical devices (CE mark, class I, class II), collecting periodically the most common vital signs (blood pressure, blood glucose level, weight, arrhythmia, …). Over time, the number of vital sign and medical devices will increase to address clinical needs[6].

This section describes how the interactions between DFI and Eliot Hub is performed and tested. Eliot Hub includes a test environment where it is possible to register an application where to send fake data representing information coming from the devices that it supports. This test environment has been used to perform some tests of integration to check if Eliot Hub is able to invoke DFI IOT API to send data representing measures made with the devices that it supports.

On the other side it has been tested if DFI receives correctly data coming from Eliot Hub framework and if such data can be converted and stored in FHIR and RDF repository according the GK-FHIR-profile. The hypothesis is that these devices are used by Puglia pilot.

Since DFI is deployed on the ENG server, it is reached by external applications. All service has been deployed as Docker containers with HTTPS protocol. In order to perform this integration, following tasks has been performed:

1. Register an organization on Eliot Hub cloud application
2. Add a new user for the organization registered at point 1.
3. Register DFI as a new data source in the section “target systems” of Eliot Hub.
4. Added in DFI a new Java Converter for Puglia pilot and Eliot Hub application.
5. From Eliot Hub application run test executions aiming to send test measurements to DFI.
6. Check if data sent by Eliot Hub reach DFI and they are converted and persisted in right way both in FHIR and RDF representation according GK-FHIR profile.

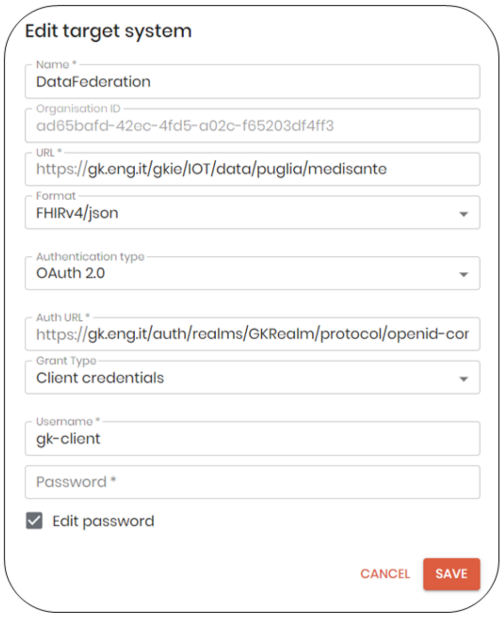
[6] The portfolio of cellular-based devices will continue to increase according to clinical needs, volume, connectivity attributes, data security attributes and internal validation.

Eliot Hub offers a set APIs allowing to register a user, an organization, one or devices and a target system to which sends collected data by the registered devices. The swagger of the application is available on this URL https://api-docs.medisante.net/#/.

The first step is the registration of the organization (e.g. Puglia) and the creation of a new user for such organization. Successful it needs to register the DFI platform as target system. Following there are information required to register the new DFI system:

- Name of the application, “*DataFederation*”.
- URL where send data, ”*https://gk.eng.it/gkie/IOT/data/puglia/medisante*”. Interface enabling IOT medisantè devices, used for Puglia pilot, this API means that data coming from devices used by Puglia pilot registered into Eliot HUB collector
- Authentication type, “*OAuth 2.0*”.
- Auth URL, “*https://gk.eng.it/auth/realms/GKRealm/protocol/openid-connect/token*”. This is the keycloak endpoints to retrieve the Bearer token
- Grant type, “Client credentials”.
- Username.
- Password.

Figure 40 shows the screenshot of the Eliot Hub target system form.

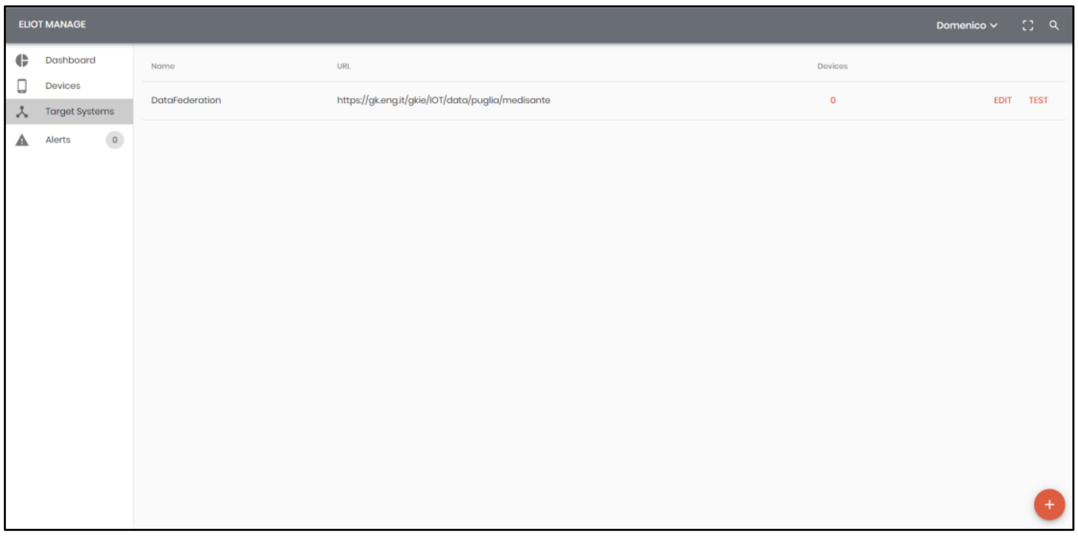

Edit target system

| | |
|---|---|
| Name * | DataFederation |
| Organisation ID | ad65bafd-42ec-4fd5-a02c-f65203df4ff3 |
| URL * | https://gk.eng.it/gkie/IOT/data/puglia/medisante |
| Format | FHIRv4/json |
| Authentication type | OAuth 2.0 |
| Auth URL * | https://gk.eng.it/auth/realms/GKRealm/protocol/openid-cor |
| Grant Type | Client credentials |
| Username * | gk-client |
| Password * | |

✓ Edit password

Figure 40 Eliot Hub Target System

After clicked on save button the new system is registered in the environment test, as shown in Figure 41.



| Name | URL | Devices |
|---|---|---|
| DataFederation | https://gk.eng.it/gkie/IOT/data/puglia/medisante | 0 |

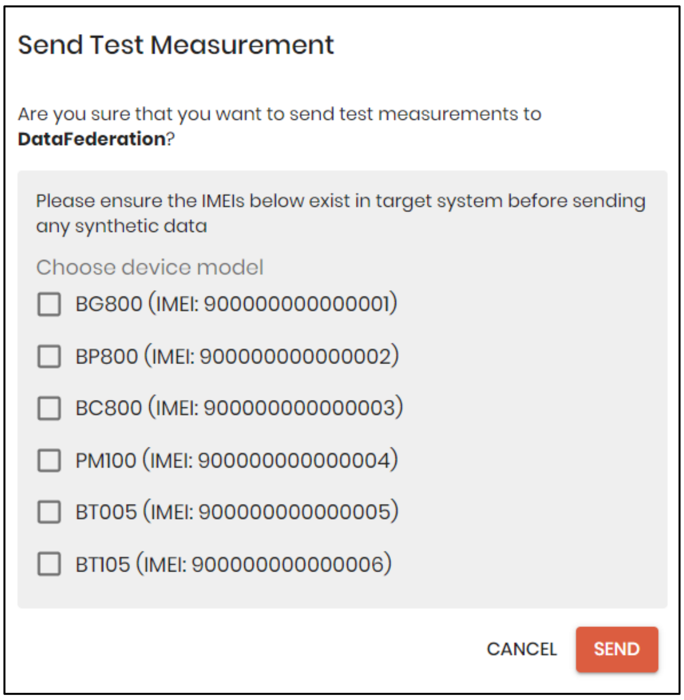
Figure 41 Eliot Manage page

Once the system is registered it is possible to send fake data to DFI. The supported devices[7] from the test environment are:

- BG800, to measure the level of haemoglobin glucometer.
- BP800, to measure the arm blood pressure and blood glucose level.
- BC800, to measure the body weight.
- PM100, the ECG event recorder.
- BT005, to measure the body weight.
- BT105, to measure heart rate together with the blood pressure systolic and diastolic.

Each device is assigned a unique identifier, named IMEI, as shown in Figure 42.

[7] https://medisante-group.com/devices


Send Test Measurement

Are you sure that you want to send test measurements to **DataFederation**?

Please ensure the IMEIs below exist in target system before sending any synthetic data

Choose device model

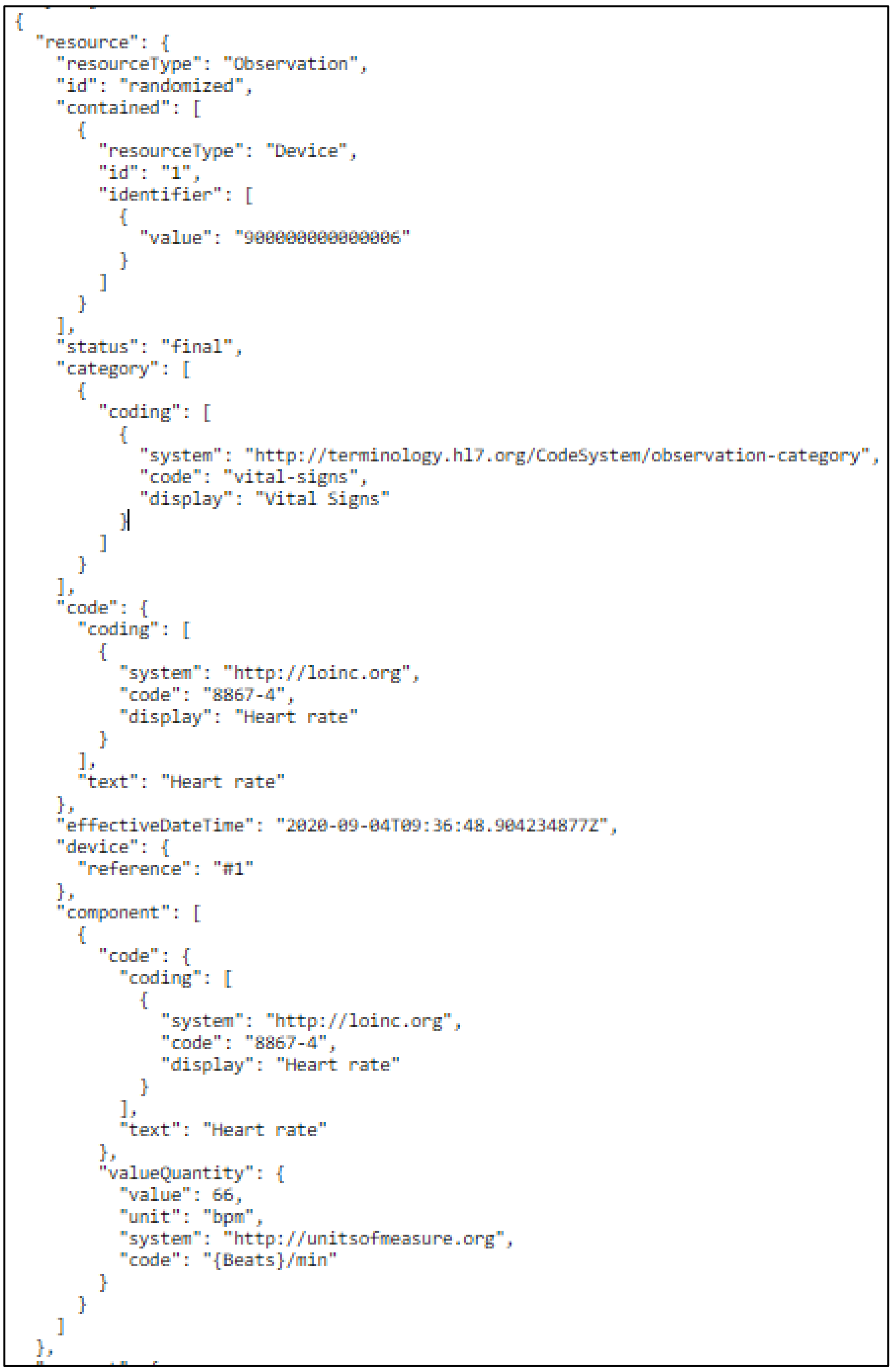
- ☐ BG800 (IMEI: 900000000000001)
- ☐ BP800 (IMEI: 900000000000002)
- ☐ BC800 (IMEI: 900000000000003)
- ☐ PM100 (IMEI: 900000000000004)
- ☐ BT005 (IMEI: 900000000000005)
- ☐ BT105 (IMEI: 900000000000006)

Figure 42 Eliot Hub test devices

Eliot Hub framework produces test data in FHIR v4 JSON representation, such JSON consists of a Bundle resource containing one or more entry, one for each type of performed measure represented as FHIR Observation measure. Each Observation contains the type of measure, the date when it has executed, the value of the measure together with the unit of measure and finally the identifier of the device that has generated the measure represented as contained FHIR Device resource.

When Eliot Hub application sends data to Data Federation & Integration, such data are transformed to GK-FHIR-Profile and stored in the repository in FHIR and RDF format.

Following an example of a piece of information generated by Eliot Hub application for the device BT105.



```
{
  "resource": {
    "resourceType": "Observation",
    "id": "randomized",
    "contained": [
      {
        "resourceType": "Device",
        "id": "1",
        "identifier": [
          {
            "value": "900000000000006"
          }
        ]
      }
    ],
    "status": "final",
    "category": [
      {
        "coding": [
          {
            "system": "http://terminology.hl7.org/CodeSystem/observation-category",
            "code": "vital-signs",
            "display": "Vital Signs"
          }
        ]
      }
    ],
    "code": {
      "coding": [
        {
          "system": "http://loinc.org",
          "code": "8867-4",
          "display": "Heart rate"
        }
      ],
      "text": "Heart rate"
    },
    "effectiveDateTime": "2020-09-04T09:36:48.904234877Z",
    "device": {
      "reference": "#1"
    },
    "component": [
      {
        "code": {
          "coding": [
            {
              "system": "http://loinc.org",
              "code": "8867-4",
              "display": "Heart rate"
            }
          ],
          "text": "Heart rate"
        },
        "valueQuantity": {
          "value": 66,
          "unit": "bpm",
          "system": "http://unitsofmeasure.org",
          "code": "{Beats}/min"
        }
      }
    ]
  },
```

Figure 43 Example of data generated by Medisantè device BT105

Figure 44 shows the steps of the integration between the Eliot Hub application and Data Federation and Integration. In order to perform this integration, it is needed that the DFI is registered to Eliot Hub collect. After the registration whenever a new measure is generated by a Medisantè device, used from a patient of a specific pilot, such measure is shared with Eliot Hub intelligent connector that forwards it (PUSH modality) to DFI as FHIR Bundle in JSON format. DFI loads the implemented transformer for Medisantè application, it transforms data to GK-FHIR- and it invokes the API of gk-fhir-server, belonging to specific pilot, to store arrived data and FHIR and RDF format.

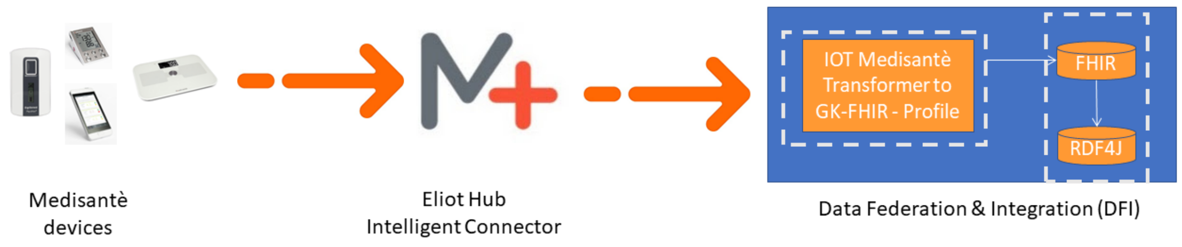

Medisantè devices
Eliot Hub Intelligent Connector
IOT Medisantè Transformer to GK-FHIR - Profile
FHIR
RDF4J
Data Federation & Integration (DFI)

Figure 44 Eliot Hub and DFI integration

### 2.4.2 Interaction and integration with Samsung Health gateway

This section describes how Samsung Health gateway and Data Federation & Integration interact in order to share data collected by the Samsung Health app.

Samsung Health (originally S Health) is a free application developed by Samsung that serves to track various aspects of daily life contributing to wellbeing such as physical activity, diet, and sleep. Launched on 2 July 2012, the application was installed by default only on some smartphones of the brand. It could also be downloaded from the Samsung Galaxy Store.

Since mid-September 2015, the application is available to all Android users. From 2 October 2017, the app is available for iPhones from iOS 9.0. The application is installed by default on some Samsung smartphone models and cannot be removed without root. It is possible to disable this application. The app changed its name from S Health to Samsung Health on 4 April 2017, when it released version 5.7.1.

The dashboard is the main display of the application. This is the main novelty introduced during the redesign of the application in April 2015 in version 4.1.0. The table shows on one page, a general overview of the most recent data saved. In addition, it provides direct access to each feature. Its composition and layout are customizable.

Some features are tracked by testing with phone sensors or phone accessories (Fitbit, Galaxy Active, Galaxy Fit, etc.) and some features are tracked by user input. (food/calories, weight, water amount, etc.).

Even if the Samsung Health App is able to collect a wide range of data, for GateKeeper projects only a subset of data type are collected and share with the platform. In detail the acquired data are:

- Blood Glucose
- Blood Pressure
- Caffeine Intake
- Floors Climbed
- Heart Rate
- Oxygen Saturation
- Sleep

- Sleep Stage
- Step Count
- Exercise
- Water Intake
- Weight
- Height
- Step Daily Trend

Some of these data are tracked by Galaxy Wearable App running on phone accessories (e.g. Samsung Watch) and other ones are tracked by user input. Data tracked by phone accessories are blood glucose, blood pressure, floors climbed, heart rate, oxygen saturation, sleep, sleep stage, step count, exercise, and step. Data tracked user input are caffeine intake, water intake, weight, and height.

Figure 45 shows in which way data acquired by Samsung Health app are send to Data Federation & Integration invoking one of the provided southbound API. In the scope of Gatekeeper project will be developed a background service sdk app running on the smartphone that will provide an API able to collect data tracked by Samsung Health App and forward such data to DFI by means the southbound API that it provides.

Data tracked by Galaxy Wearable App running on the watch are auto synchronized with Samsung Health app by Bluetooth; such data, together with data tracked directly user input inside the Samsung Health app, are sent to the background Gatekeeper Service app (represented by an orange rectangle in the figure below) invoking the specific API.

The GateKeeper service App sends acquired raw health data, with device ID, to Data Federation & Integration framework invoking the southbound IOT API (*https://gk.eng.it/gkie/IOT/data/{pilot}/{sensorID}*) in PUSH modality with an interval of 1 hour. DFI includes a routine that transforms raw data, coming from the Gatekeeper service app, to HL7 FHIR and RDF format (according the define GK-FHIR-Profile) and stores them to the relative repositories. This data can be retrieved from the work package 5 by means the northbound APIs.

Thanks to this workflow, tracked health data by the Samsung Health app, are acquired and stored in Data Federation & Integration FHIR and RDF repositories each hour.

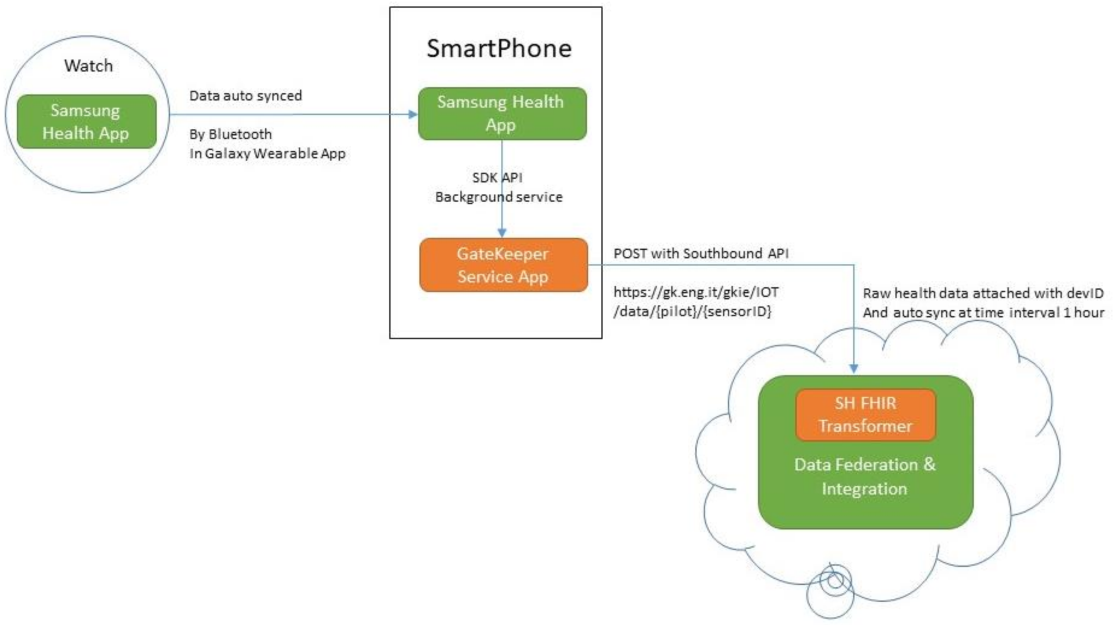


Figure 45 Samsung Health gateway and DFI Integration

# 3 Data federation and integration V1: implementation details

## 3.1 GK-Integration Engine

### 3.1.1 Apache Camel

As described in the section above, the Data Federation takes care “to route” the data (properly converted to the GK FHIR Profile) to the *pilot specific* “data node” (i.e. dedicated FHIR server and RDF4J server) hosted in dedicated cluster - see Section 4 for details. Moreover, it is expected to also route such data toward other external systems (e.g. Big Data infrastructure). These requirements convinced us to adopt the Apache Camel framework that aims to make integrating systems easier relying on message routing features.

Building complex systems from scratch is a costly endeavor, and one that’s almost never successful. An effective and less risky alternative is to assemble a system like a jigsaw puzzle from existing, proven components. We depend daily on a multitude of such integrated systems, making possible everything from phone communications, financial transactions, and health care to travel planning and entertainment.

At the core Camel framework is a routing engine. It allows to define own routing rules, decides from which sources to accept messages, and determine how to process and send those messages to other destinations. More in details Camel is a black box that receives messages from some endpoint and sends it to another one. Within the black box, the messages may be processed or simply redirected.

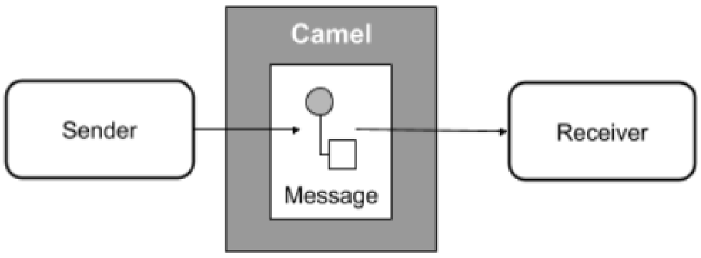


Figure 46 Camel

The rational of this approach is that in practical situations, there may be many senders and many receivers each following its own protocol such as ftp, http and jms. The system may require many complex rules such as message from sender A should be delivered only to B & C. In situations, it could be needed to translate the message to another format that the receiver expects. This translation may be subject to certain conditions based on the message contents. So essentially it is needed to translate between protocols, glue components together, define routing rules, and provide filtering based on message contents. To meet these requirements and design a proper software architecture for many such situations, **Enterprise Integration Patterns** (EIP) were documented by Gregor Hohpe and Bobby Woolf in 2003. Apache Camel provides the implementation of these patterns. Apache Camel is an open-source framework offering a message-oriented middleware that provides rule-based routing and mediation engine. It is possible to define rules such as, for example, *if a "milk" order arrives, redirect it to a milk vendor and if it is an "oil" order redirect it to an oil vendor, and so on.*

Using Camel, it is possible to implement these rules and do the routing in a familiar Java code. It means that a familiar Java IDE can be exploited to define these rules in a type-safe environment. It is not needed to use XML configuration files, which typically tend to be bulky. Camel though supports XML configuration through Spring framework, if preferred. A developer can even use Blueprint XML Configuration files and even a Scala DSL that means you can use your favourite Java, Scala IDE or even a simple XML editor to configure the rules. The input to this engine can be a comma-delimited text file, a POJO (Plain Old Java Object), XML are any of the several other formats supported by Camel. Similarly, the output of the engine can be redirected to a file, to a message queue or even your monitor screen. These are called the endpoints and Camel supports the **Message Endpoint EIP pattern**. The Camel core itself is very small and contains 13 essential components. The rest 80+ components are outside the core. This helps in maintaining a low dependency on where it is deployed and promotes extensions in future. The **Components** module provides an **Endpoint** interface to the external world. The Endpoints are specified by URIs, such as **file:/order** and **jms:orderQueue** that you have seen in the last chapter.

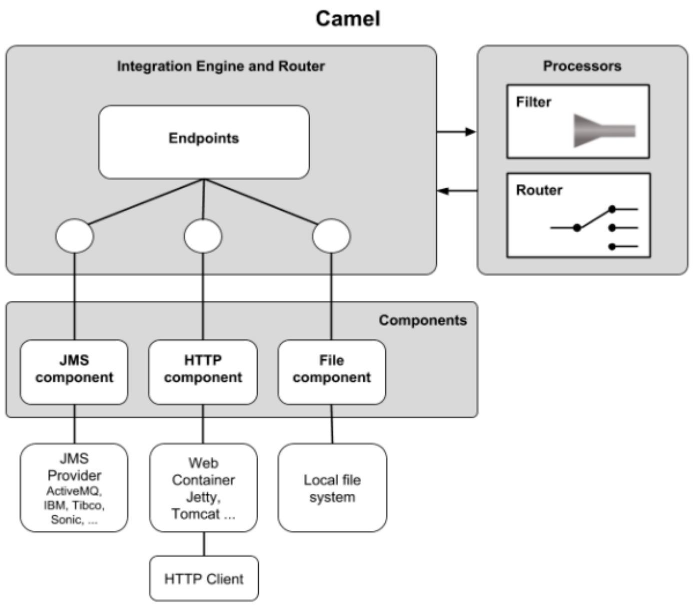


Figure 47 Camel Architecture

The **Processors** module is used for manipulating and mediating messages between Endpoints. The EIPs mentioned earlier are implemented in this module. It currently supports 40+ patterns as documented in the EIP book and other useful processing units. The **Processors** and **Endpoints** are wired together in **Integration Engine and Router** module using DSLs. While wiring these, it is possible to filter messages based on user-defined criteria. As mentioned earlier, several options are available in writing these rules: Java, Scala, Groovy, or even XML for this. Finally, the most important component of Camel, which may be considered as the core - the **CamelContext.**

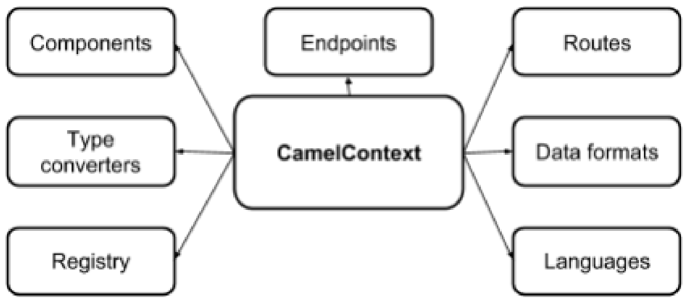


Figure 48 Camel Context

**CamelContext** provides access to all other services in Camel as shown in the Figure 48, here briefly described. The **Registry** module by default is a JNDI registry, which holds the name of the various Javabeans that the application uses. If Camel is used with Spring, this will be the Spring **ApplicationContext**. If Camel is used in OSGI container, this will be **OSGI registry**. The **Type converters** as the name suggests contains the various loaded type converters, which convert the input from one format to another. It is also possible to use the built-in type converters or provide new custom mechanism of conversion. The **Components** module contains the components used by the application. The components are loaded by autodiscovery on the classpath that is specified. In case of the OSGI container, these are loaded whenever a new bundle is activated. The Endpoints and Routes have been already described above. The Data formats module contains the loaded data formats and finally the Languages module represents the loaded languages.

The Apache Camel framework has been used as development framework of the GK integration engine along with the spring framework. In the figure below the real technological stack adopted.

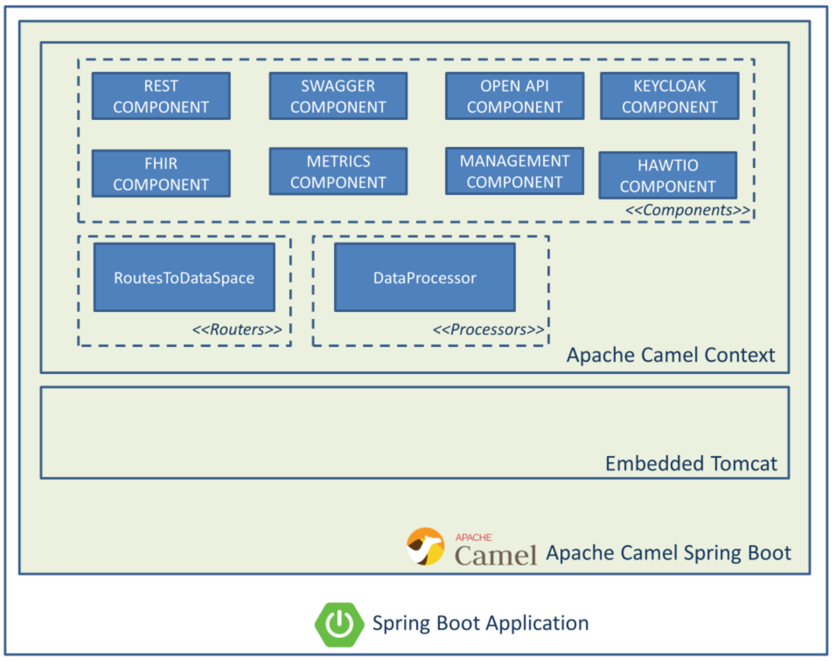
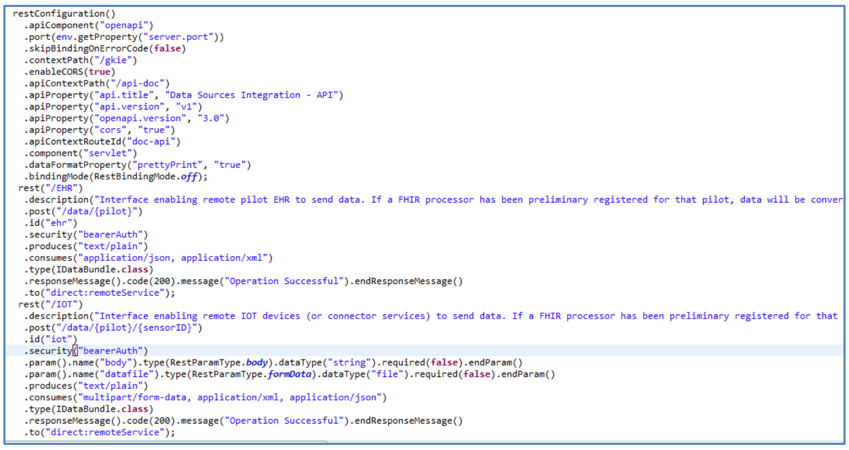


Figure 49 GK Integration Engine: implementation stack

For this purpose, a specific component for exposing REST API has been integrated (see **REST COMPONENT** in the figure). It offers a REST styled DSL which can be used with Java or XML. The intention is to allow end users to define REST services using a REST style with verbs such as get, post, delete etc. To use the Rest DSL in Java then it was sufficient to do as with regular Camel routes by extending the RouteBuilder (see **RoutesToDataSpace** java class in the figure) and define the routes in the configure method.



```
restConfiguration()
  .apiComponent("openapi")
  .port(env.getProperty("server.port"))
  .skipBindingOnErrorCode(false)
  .contextPath("/gkie")
  .enableCORS(true)
  .apiContextPath("/api-doc")
  .apiProperty("api.title", "Data Sources Integration - API")
  .apiProperty("api.version", "v1")
  .apiProperty("openapi.version", "3.0")
  .apiProperty("cors", "true")
  .apiContextRouteId("doc-api")
  .component("servlet")
  .dataFormatProperty("prettyPrint", "true")
  .bindingMode(RestBindingMode.off);
 rest("/EHR")
  .description("Interface enabling remote pilot EHR to send data. If a FHIR processor has been preliminary registered for that pilot, data will be conver
  .post("/data/{pilot}")
  .id("ehr")
  .security("bearerAuth")
  .produces("text/plain")
  .consumes("application/json, application/xml")
  .type(IDataBundle.class)
  .responseMessage().code(200).message("Operation Successful").endResponseMessage()
  .to("direct:remoteService");
 rest("/IOT")
  .description("Interface enabling remote IOT devices (or connector services) to send data. If a FHIR processor has been preliminary registered for that
  .post("/data/{pilot}/{sensorID}")
  .id("iot")
  .security("bearerAuth")
  .param().name("body").type(RestParamType.body).dataType("string").required(false).endParam()
  .param().name("datafile").type(RestParamType.formData).dataType("file").required(false).endParam()
  .produces("text/plain")
  .consumes("multipart/form-data, application/xml, application/json")
  .type(IDataBundle.class)
  .responseMessage().code(200).message("Operation Successful").endResponseMessage()
  .to("direct:remoteService");
```

Figure 50 REST interface using Camel

In order to expose such REST interface trough swagger interface, the appropriate Camel component (see **SWAGGER COMPONENT** in the figure) has been integrated.

One of the core Camel components is **DataProcessor** that implements the **Processor**[8] interface used to implement consumers of message exchanges or to implement a **Message Translator**[9].

The Processor interface requires to implement a process method that accepts an **Exchange** class parameter containing all the information needed for the route. Once a Processor is developed then it can be easily used inside a route by the declaring of the bean in Spring or suing the **DSL** syntax.

Figure 51 shows the operating logic inside the **DataProcessor,** represented with a flow chart, for selecting the converter to be used, the output of the semantic model and the server where to send and store converted data (FHIR server o RDF server). The **DataProcessor** is used by the **RouteBuilder** class that is derived from to create routing rules using the DSL. Instances of **RouteBuilder** are then added to the **CamelContext**.

[8] https://camel.apache.org/manual/latest/processor.html

[9] https://camel.apache.org/components/latest/eips/message-translator.html

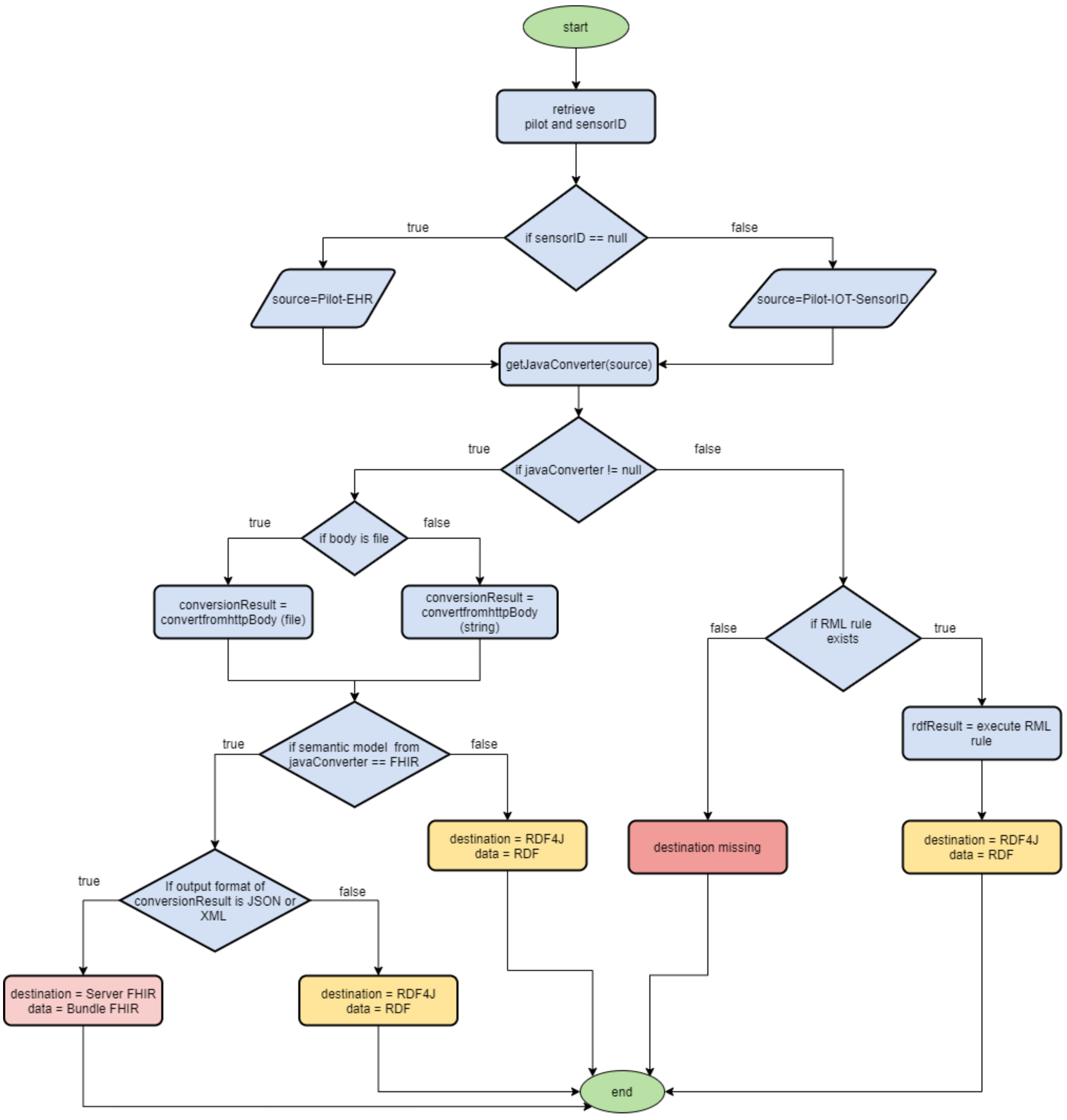

start
retrieve pilot and sensorID
if sensorID == null
true
false
source=Pilot-EHR
source=Pilot-IOT-SensorID
getJavaConverter(source)
if javaConverter != null
true
false
if body is file
true
false
conversionResult = convertfromhttpBody (file)
conversionResult = convertfromhttpBody (string)
if RML rule exists
false
true
rdfResult = execute RML rule
if semantic model from javaConverter == FHIR
true
false
destination = RDF4J data = RDF
destination missing
destination = RDF4J data = RDF
If output format of conversionResult is JSON or XML
true
false
destination = Server FHIR data = Bundle FHIR
destination = RDF4J data = RDF
end

Figure 51 DataProcessor flowchart

When the processor is invoked, it retrieves the name of the pilot and the sensorID passed as path parameter in the REST request; this information is needed to understand which of the two RESP APIs has been invoked, if EHR or IOT. To perform this operation, it is checked if the sensorID is null (or empty), if true, the source consists of the pilot name and the EHR otherwise the source is an IOT with pilot name and the id of the sensor. Information added inside the source are needed to select the relative converter from raw data to FHIR/RDF. Afterwards it is invoked a method, named getJavaConverter that takes in input the source built in previous step that returns the corresponding Java converter. If the returned javaConverter is not null, then there is a java converter associated to that source otherwise it is checked if there exists e RML processor for the selected source.

If a Java converter exists, it is verified if the body of the request is a file or a string (with a JSON/XML representation). If the body is a file, it is invoked a method (convertfromhttpBody) that reads the content from the file otherwise the same method is invoke but it reads the content from a string variable. After it is verified if the output sematic model retrieved by JavaConverter is FHIR, if false the destination of the converted data is set to RDF4J and data are RDF, this mean that the converted data are send to RDF4J server with a RDF format. If the semantic model is FHIR then it is verified if the output format is XML or JSON, if true the destination is set to server FHIR and data is the FHIR Bundle, this means that converted data are in the FHIR Bundle and sent to the FHIR server. If the output format is neither JSON nor XML then the destination is RDF4J server and data are in RDF format. If the semantic model is not FHIR then the destination is RDF4J and data are in RDF format.

Returning to the step where it is checked if the javaconverter is not null (second rhombus), if false this mean that the conversion has been developed using the RML rule languages. For this reason, it is checked if exists a RML processor associated to the built source, if true the RML engine is executed with the associated rules and the destination is set to RDF4J with data in RDF format otherwise if it is not existing the RML processor associated with that source then the destination is missing, and data are not sent to any server.

Summarizing, the goal of the flow chart (described above) is to retrieve the type of the source for the specific pilot (EHR or IOT), the type of converter (JavaConverter or RML), the output format (FHIR or RDF) and the server to send converted data (FHIR Server of RDF4J Server).

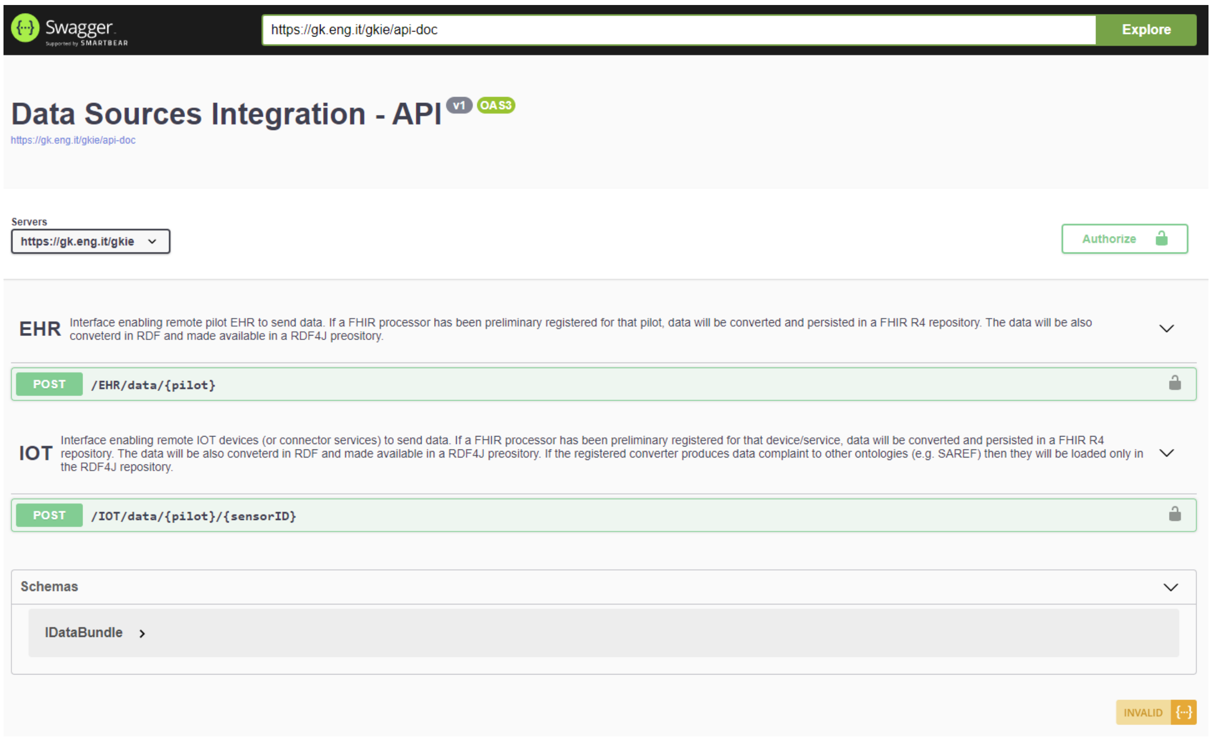
Following there is a description of other two components used in Apache Camel context:

- **Keycloak component**: it is a component that has been introduced to implement the OAuth 2.0 protocol for the rest APIs defined in **REST Component.**
- **FHIR component** [11]: it integrates with the HAPI-FHIR library which is an open-source implementation of the FHIR (Fast Healthcare Interoperability Resources) specification in Java. It uses the URL format *fhir://endpoint-prefix/endpoint?[options]*, *endpoint* prefix can be one of capabilities, create, delete, history, load-page, meta, operation, patch, read, search, transaction, update and validate. It is used in the **RouteBuilder** to invoke FHIR Server to store FHIR data.

### 3.1.2 Interfaces

Data Federation & Integration exposes two southbound APIs to accept data coming from heterogeneous data sources registered in the platform, including personal clinical data source (EHR), social care data sources, wearable data sources, thus producing a HL7 FHIR and semantic repository.

These APIs aiming to accept heterogeneous data, coming from the several pilots' applications registered into the platform, to produce a repository where data can be retrieved from the northbound APIs described in the next section. Figure 52 shows the swagger of the two interfaces while Table 10 and Table 11 provide their descriptions.



Data Sources Integration - API v1 OAS3

https://gk.eng.it/gkie/api-doc

Servers

https://gk.eng.it/gkie

**EHR** Interface enabling remote pilot EHR to send data. If a FHIR processor has been preliminary registered for that pilot, data will be converted and persisted in a FHIR R4 repository. The data will be also conveterd in RDF and made available in a RDF4J preository.

POST `/EHR/data/{pilot}`

**IOT** Interface enabling remote IOT devices (or connector services) to send data. If a FHIR processor has been preliminary registered for that device/service, data will be converted and persisted in a FHIR R4 repository. The data will be also conveterd in RDF and made available in a RDF4J preository. If the registered converter produces data complaint to other ontologies (e.g. SAREF) then they will be loaded only in the RDF4J repository.

POST `/IOT/data/{pilot}/{sensorID}`

Schemas

IDataBundle

Figure 52 Swagger of gk-integration-engine

EHR Interface enabling remote pilot EHR to send data. If a FHIR processor has been preliminary registered for that pilot, data are converted and persisted in a FHIR R4 repository. The data are also converted in RDF and made available in a RDF4J repository.

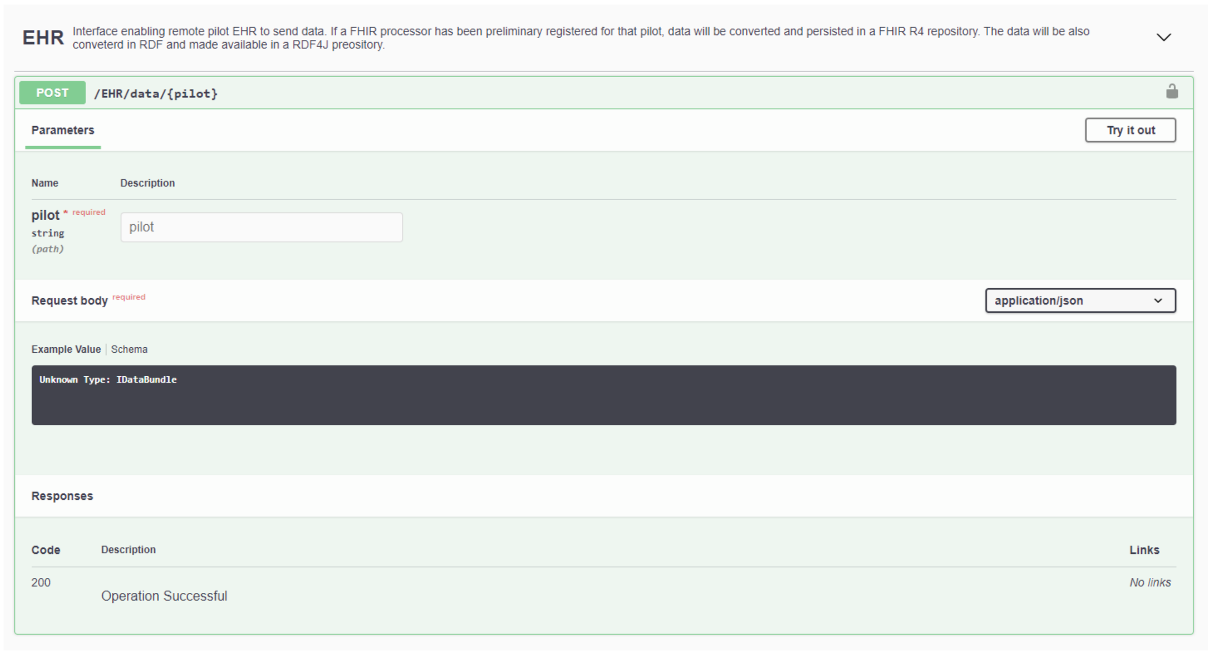
The interface accepts two inputs: the name of the pilot and the JSON/XML representation of data to be stored in GK platform. The Pilot's name is passed in the URI pattern of the request and in order to understand to which pilot data belongs to.

The second input is data in JSON/XML format to be stored. The structure of this data must be the same of the FHIR processor that has been preliminary registered for the specific pilot. gk-integration-engine, based on the name of the pilot, select the corresponding FHIR processor that is applied on the data passed in the body of the request. The output of the interface is an HTTP 201 message if data are successful converted and persisted in FHIR repository or HTTP 500 if an error has occurred, this could happen for example when there is no FHIR processor registered to the pilot declared in the URI pattern of the FHIR server is not available.

Table 10 EHR southbound API

| ID Operazione | 3.1.2.1 |
|---|---|
| **Signature** | void saveEHRdata (String pilot, String data) |
| **URI pattern** | POST https://{host}:{port}/gkie/EHR/data/{pilot} |
| **Input** | • pilot (pilot name)<br>• Json/XML representation of EHR data to be stored in Data |

| | |
|---|---|
| | Federation & Integration |
| **Output** | • HTTP 201 if data are successful persisted<br>• http 500 if an error has occurred |


EHR Interface enabling remote pilot EHR to send data. If a FHIR processor has been preliminary registered for that pilot, data will be converted and persisted in a FHIR R4 repository. The data will be also conveterd in RDF and made available in a RDF4J preository.

POST /EHR/data/{pilot}

Parameters

| Name | Description |
|---|---|
| pilot * required<br>string<br>(path) | pilot |

Request body required — application/json

Example Value | Schema

```
Unknown Type: IDataBundle
```

Responses

| Code | Description | Links |
|---|---|---|
| 200 | Operation Successful | No links |

Figure 53 Swagger EHR southbound API

The IOT Interface enables remote IOT devices (or intelligent connector services) to send data. If a FHIR processor has been preliminary registered for that device/service, data will be converted and persisted in a FHIR R4 repository. The data will be also converted in RDF and made available in a RDF4J repository. If the registered converter produces data complaint to other ontologies (e.g. SAREF) then they will be loaded only in the RDF4J repository.

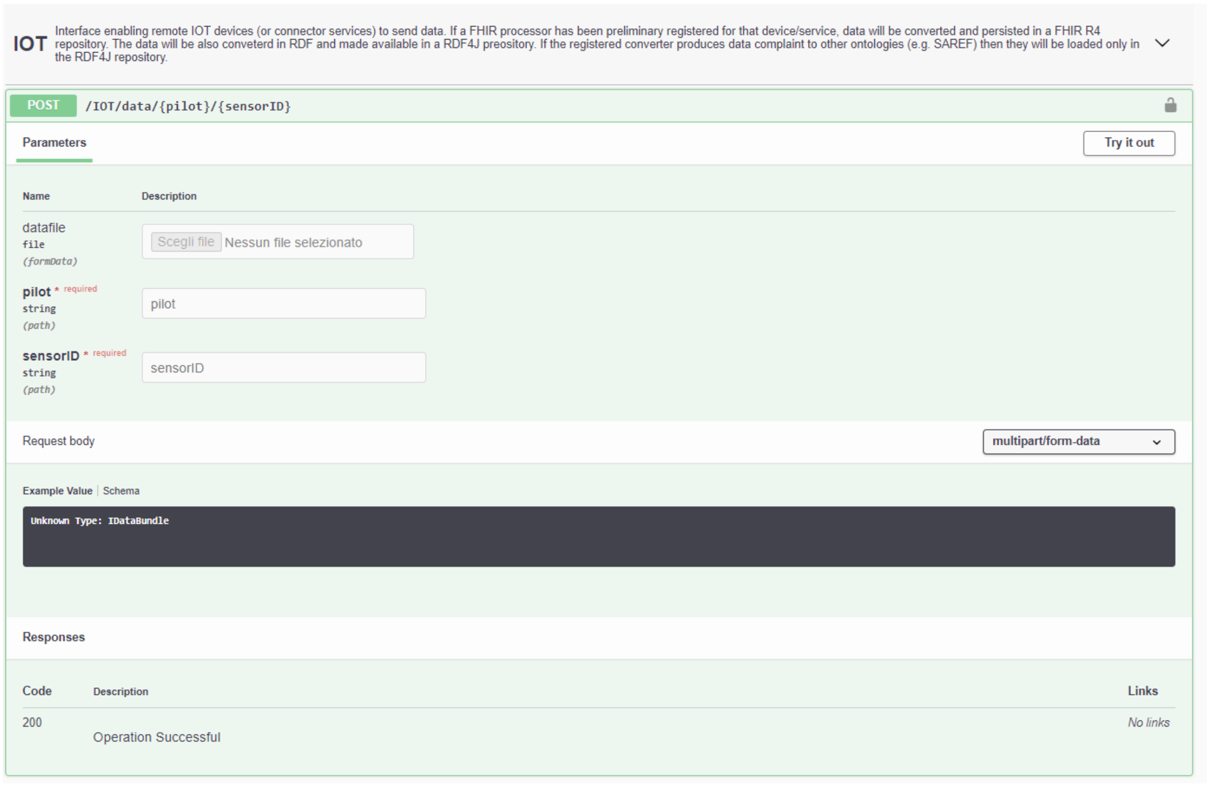
The interface accepts in input three parameters: name of the pilot, sensorID (or the name of the intelligent connector) and the JSON/XML representation of raw data as string or in a file.

The association between pilot and sensorID allows to select the FHIR processor, if preliminary registered, to convert data to FHIR format and RDF4j repository. If it is registered only a RML converter are compliant to other ontology and they will be stored only in RDF repository and not in FHIR.

The output of the interface is an HTTP 201 if data are successful converted and persisted in FHIR repository and HTTP 500 if an error has occurred, this could happen for example when there is no FHIR processor registered to the pilot declared in the URI pattern of the FHIR server is not available.

Table 11 IOT southbound API

| ID Operazione | 3.1.2.1 |
|---|---|
| **Signature** | void saveIOTdata (String pilot, String sensorID) |
| **URI pattern** | POST https://{host}:{port}/gkie/IOT/data/{pilot}/{sensorID} |
| **Input** | • pilot (pilot name)<br>• sensorID (sensorID or name of the IOT collector that is sending data)<br>• Json/XML representation of raw data to be stored in Data Federation & Integration<br>• Or file containing JSON/XML representation of raw data to be store in Data Federation & Integration |
| **Output** | • HTTP 201 if data are successful persisted<br>• http 500 if an error has occurred |


IOT Interface enabling remote IOT devices (or connector services) to send data. If a FHIR processor has been preliminary registered for that device/service, data will be converted and persisted in a FHIR R4 repository. The data will be also conveterd in RDF and made available in a RDF4J preository. If the registered converter produces data complaint to other ontologies (e.g. SAREF) then they will be loaded only in the RDF4J repository.

POST /IOT/data/{pilot}/{sensorID}

Parameters

| Name | Description |
|---|---|
| datafile file (formData) | Scegli file Nessun file selezionato |
| pilot * required string (path) | pilot |
| sensorID * required string (path) | sensorID |

Request body: multipart/form-data

Example Value | Schema

```
Unknown Type: IDataBundle
```

Responses

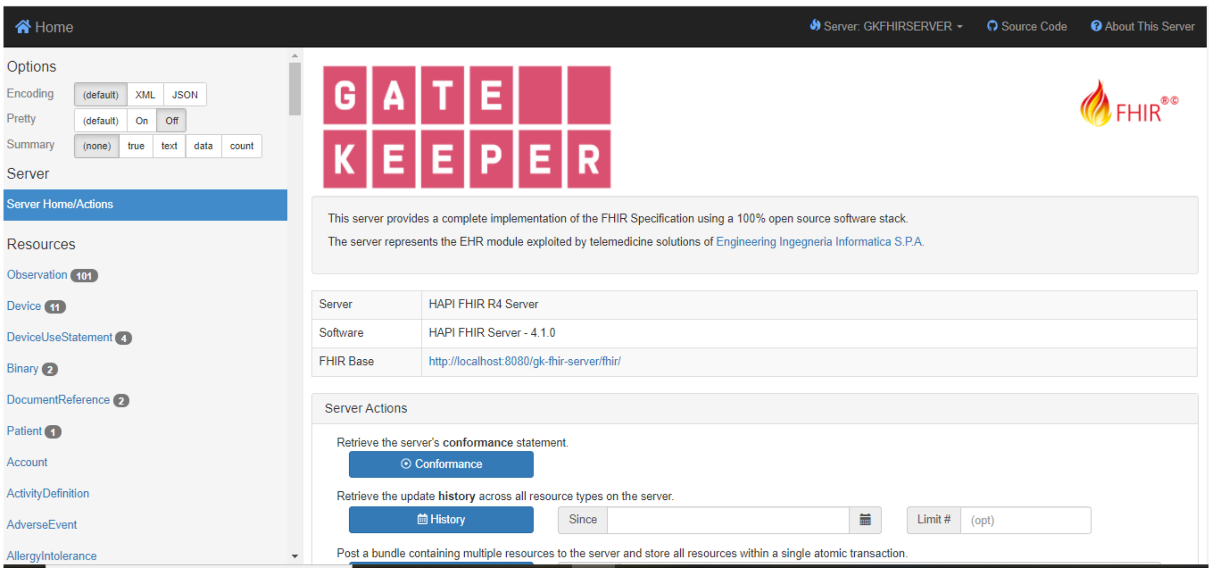
| Code | Description | Links |
|---|---|---|
| 200 | Operation Successful | No links |

Figure 54 Swagger IOT southbound API

## 3.2GK-FHIR Server

The gk-fhir-server is a Sprint Boot project that implements the HL7 FHIR v4 specification according the official standard and customized for Gatekeeper project. It uses the HAPI FHIR [12] that is Java software library facilitating a built-in mechanism for adding FHIR's RESTful Server functionalities to a software application. The HAPI FHIR Java library is open source. The HAPI RESTful (Representation State Transfer) Server is based on a Servlet, so it should be deployed with ease to any compliant containers that can be provided. Simple annotations could be used to set up the server on the large part.

Project gk-fhir-server provides all the REST APIs defined by the standard together with the whole data model based on the concept of Resource. Figure 55 gk-fhir-server GUI shows the GUI of the application. On the left side there is the list of all supported Resources, selecting one of them it is possible to access to the relative Rest APIs.



This server provides a complete implementation of the FHIR Specification using a 100% open source software stack.

The server represents the EHR module exploited by telemedicine solutions of Engineering Ingegneria Informatica S.P.A.

| | |
|---|---|
| Server | HAPI FHIR R4 Server |
| Software | HAPI FHIR Server - 4.1.0 |
| FHIR Base | http://localhost:8080/gk-fhir-server/fhir/ |

Server Actions

Retrieve the server's **conformance** statement.

Retrieve the update **history** across all resource types on the server.

Post a bundle containing multiple resources to the server and store all resources within a single atomic transaction.

Figure 55 gk-fhir-server GUI

The customization of the server, the definition of the capability statement and the profiles of the resources are an ongoing activity which is closely connected with the definition of GK-FHIR-Profile coming from the output of the task 3.5. The selected database to persist data in gk-fhir-server is Maria DB.

Data Federation & Integration provides two kinds of northbound APIs to retrieve data in RDF Format (JSON or XML) and RDF format (JSON or XML). The source code of HAPI FHIR library does not provide any mechanism to retrieve data in RDF format so a dedicated module has been implemented and integration into the tool.

This module is a python script that has two inputs: the JSON of the FHIR resource and the endpoint of the RDF server where to send transformed RDF data. Internally this script converts the Resource from JSON to RDF format and sends converted data to the specified endpoint. This script is based on the fhirtordf [13] library.

The following table shows an example of a generated RDF representation generated by the python script starting from a JSON representation of Bundle resource containing two entries: Condition and Patient where in first column contains the JSON format while the second one contains its equivalent in the FHIR RDF format.

This routine is applied to each resource persisted in gk-fhir-server.

Table 12 Example of conversion from FHIR JSON to RDF format

| JSON format | RDF format |
|---|---|
| { "resourceType":"Bundle", "entry":[ { "fullUrl":"urn:uuid:80c129ba-dde5-42b8-8cb8-c302f9541e5d", "resource":{ "resourceType":"Patient", "identifier":[ { "system":"CAREACROSS", "value":"group/1" } ] } }, { "fullUrl":"urn:uuid:4dc96d50-454b-4f11-b5e8-70078976e20b", "resource":{ "resourceType":"Condition", "extension":[ { "id":"http://hl7.org/fhir/StructureDefinition/is-primary-disease", "url":null, "valueBoolean":false } ], "category":[ { "coding":[ { "system":"http://www.crowdhealth.eu/hhr-t", "code":"diagnosis", | *@prefix fhir: <http://hl7.org/fhir/> . @prefix loinc: <http://loinc.org/rdf#> . @prefix owl: <http://www.w3.org/2002/07/owl#> . @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> . @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> . @prefix sct: <http://snomed.info/id/> . @prefix v2: <http://hl7.org/fhir/v2/> . @prefix v3: <http://hl7.org/fhir/v3/> . @prefix w5: <http://hl7.org/fhir/w5#> . @prefix xml: <http://www.w3.org/XML/1998/namespace> . @prefix xsd: <http://www.w3.org/2001/XMLSchema#> . <http://hl7.org/fhir/Condition/4eaacaec-19e0-4e95-8b8f-be6e3c6e9972> a fhir:Condition ; fhir:nodeRole fhir:treeRoot ; fhir:Condition.category [ fhir:index "0"^^xsd:integer ; fhir:CodeableConcept.coding [ fhir:index "0"^^xsd:integer ; fhir:Coding.code [ fhir:value "diagnosis" ] ; fhir:Coding.display [ fhir:value "Diagnosis" ] ; fhir:Coding.system [ fhir:value "http://www.crowdhealth.eu/hhr-t" ] ] ] ; fhir:Condition.code [ fhir:CodeableConcept.coding [ a sct:441118006 ; fhir:index "0"^^xsd:integer ; fhir:Coding.code [ fhir:value "441118006" ] ; fhir:Coding.display [ fhir:value "Progesterone receptor negative neoplasm (disorder)"* |

```
                "display":"Diagnosis"
              }
            ]
          }
        ],
        "code":{
          "coding":[
            {
              "system":"http://snomed.info/sct",
              "code":"441118006",
              "display":"Progesterone receptor negative neoplasm (disorder)"
            }
          ]
        },
        "subject":{
          "reference":"urn:uuid:80c129ba-dde5-42b8-8cb8-c302f9541e5d"
        }
      }
    }
  ]
}
```

```
          ] ;
          fhir:Coding.system [
             fhir:value "http://snomed.info/sct"
          ]
       ]
    ] ;
    fhir:Condition.subject [
       fhir:link     <http://hl7.org/fhir/urn%3Auuid%3A80c129ba-dde5-42b8-8cb8-c302f9541e5d> ;
       fhir:Reference.reference [
          fhir:value "urn:uuid:80c129ba-dde5-42b8-8cb8-c302f9541e5d"
       ]
    ] ;
    fhir:DomainResource.extension [
       fhir:index "0"^^xsd:integer ;
       fhir:Element.id [
          fhir:value "http://hl7.org/fhir/StructureDefinition/is-primary-disease"
       ] ;
       fhir:Extension.url [
          fhir:value "None"
       ] ;
       fhir:Extension.valueBoolean [
          fhir:value "false"^^xsd:boolean
       ]
    ] ;
    fhir:Resource.id [
       fhir:value "4eaacaec-19e0-4e95-8b8f-be6e3c6e9972"
    ] .

<http://hl7.org/fhir/Condition/4eaacaec-19e0-4e95-8b8f-be6e3c6e9972.ttl>     a owl:Ontology ;
    owl:imports fhir:fhir.ttl .

<http://hl7.org/fhir/Patient/5077b199-0160-4358-be29-fc0b7e10cadd>     a fhir:Patient ;
    fhir:nodeRole fhir:treeRoot ;
    fhir:Patient.identifier [
       fhir:index "0"^^xsd:integer ;
       fhir:Identifier.system [
          fhir:value "CAREACROSS"
       ] ;
       fhir:Identifier.value [
          fhir:value "group/1"
       ]
    ] ;
    fhir:Resource.id [
       fhir:value "5077b199-0160-4358-be29-fc0b7e10cadd"
    ] .
```

| | |
|---|---|
| | *<http://hl7.org/fhir/Patient/5077b199-0160-4358-be29-fc0b7e10cadd.ttl> a owl:Ontology ;* *owl:imports fhir:fhir.ttl .* *<http://hl7.org/fhir/urn%3Auuid%3A80c129ba-dde5-42b8-8cb8-c302f9541e5d> a fhir:Resource .* |

Python fhirtordf script has been integrated in the FHIR server through an interceptor that works at JPA Server Storage level, as shown in Figure 56.

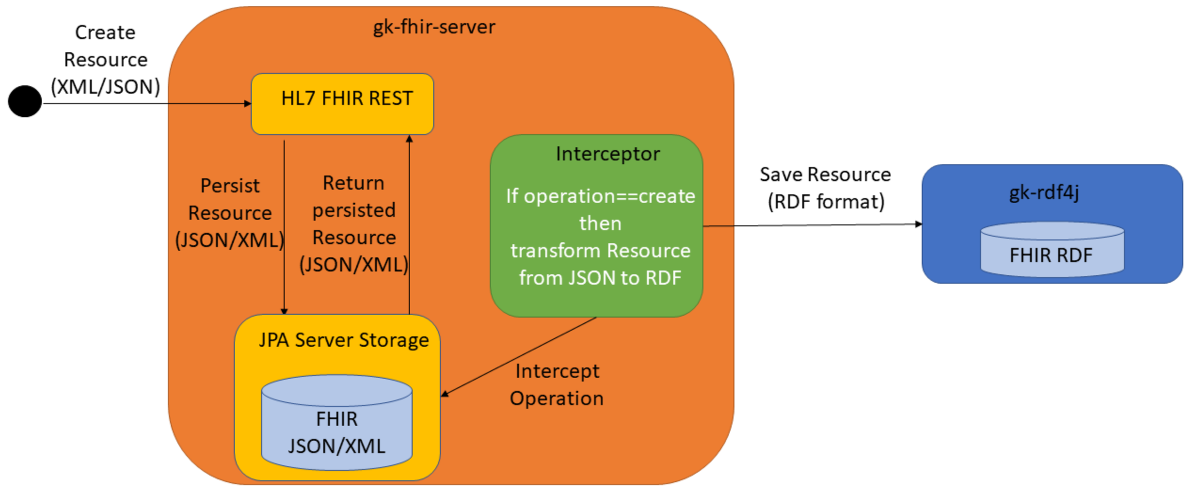

Create Resource (XML/JSON)
gk-fhir-server
HL7 FHIR REST
Persist Resource (JSON/XML)
Return persisted Resource (JSON/XML)
Interceptor
If operation==create then transform Resource from JSON to RDF
Save Resource (RDF format)
gk-rdf4j
FHIR RDF
JPA Server Storage
FHIR JSON/XML
Intercept Operation

Figure 56 gk-fhir-server interceptor

When a REST request is performed on FHIR server, for example the creation of the Resource passing the JSON in body of the request, it is persisted into data base invoking the methods offered by the module named JPA Server Storage. This module returns the JSON of the persisted Resource that will be added in an OperationOutcome resource and returned in the response of the request.

The workflow has been integrated with an interceptor mechanism that catches each request performed on JPA Server Storage retrieving the resource created, updated, or deleted after the operation completed successfully. The interceptor checks if the operation is a “create” and only in this case executes the python script, described above, that transforms data to RDF and invokes the API provided by gk-rdf4j to store them into RDF repository.

## 3.3RDF4J Workbench

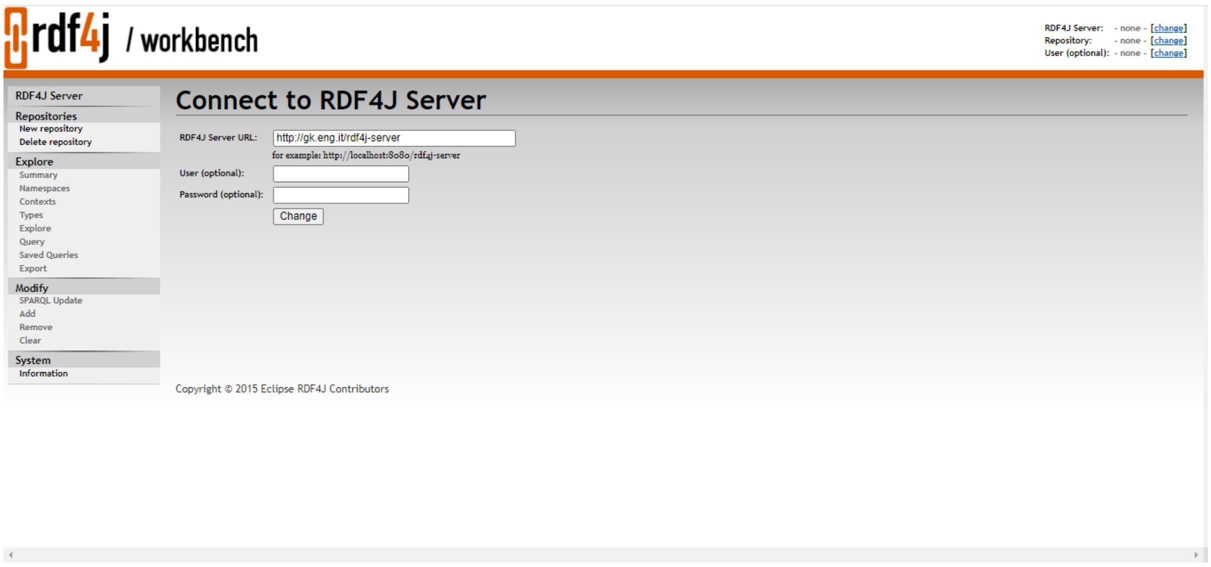
Eclipse RDF4J is an open-source framework for storing, querying, and analysing RDF (Resource Description Framework) data distributed under “Eclipse Distribution License 1.0 (BSD)”. It contains implementations of an in-memory triplestore and an on-disk triplestore, along with two separate Servlet packages that can be used to manage and provide access to these triplestores, on a permanent server. RDF4J supports two query languages: SPARQL and SeRQL, ), as well as a set of fully streaming parsers and writers for most common RDF syntax formats, called Rio.

RDF4J's RDF database API differs from comparable solutions in that it offers a stackable interface through which functionality can be added, and the storage engine (SAIL) is abstracted from the query interface. Many other triplestores can be used through the RDF4J API. Through the stackable interface, functionality can be added to all of these stores.

The current core development team consists of individuals and employees of other commercial software vendors that have an interest in continued maintenance and development of the project.

In addition to its primary use as a set of Java libraries, RDF4J also provides a Server web application that can be accessed as a web service for RDF database access, and a Workbench web application which provides a (web-based) client user interface for an RDF4J Server, with a full SPARQL query editor (with completion features and syntax highlighting), and several convenient ways to manipulate or explore the data in any RDF database/SPARQL endpoint.

- Main feature of this RDF solution are synthesized below:
- full support for SPARQL 1.1 query and update;
- fast and efficient parsing of all common RDF formats through the Rio parser toolkit;
- an easy to use, lightweight, modern Java API for handling RDF in code;
- support for RDF Schema reasoning as well as SHACL validation;
- fast in-memory RDF database with optional file-backed persistence;
- fast Native RDF database with full binary persistence to disk;
- convenient access to third-party RDF database implementations and remote SPARQL endpoints



RDF4J Server: - none - [change]
Repository: - none - [change]
User (optional): - none - [change]

Connect to RDF4J Server

RDF4J Server URL: http://gk.eng.it/rdf4j-server
for example: http://localhost:8080/rdf4j-server
User (optional):
Password (optional):

Copyright © 2015 Eclipse RDF4J Contributors

Figure 1 rdf4j workbench

## 3.4Data Federation & Integration Dockerization

### 3.4.1 Docker overview

All the components consisting of Data Federation and Integration has been migrated to Docker [14]. Docker is a set of platforms as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. All containers are run by a single operating system kernel and therefore use fewer resources than virtual machines. The service has both free and premium tiers. The software that hosts the containers is called Docker Engine [15].

A Docker image is a read-only template that contains a set of instructions for creating a container that can run on the Docker platform. It provides a convenient way to package up applications and preconfigured server environments, which can be used for own private use or share publicly with other Docker users. A Docker image can be created in one of two ways:

- **Interactive Method**: by running a container from an existing Docker image, manually changing that container environment through a series of live steps and saving the resulting state as a new image.

- **Dockerfile Method**: by constructing a plain-text file, known as a Dockerfile, which provides the specifications for creating a Docker image.

The Dockerfile approach is the method of choice for real-world enterprise-grade container deployments. It is a more systematic, flexible and efficient way to build Docker images and the key to compact, reliable and secure container environments. In short, the Dockerfile method is a three-step process whereby you create the Dockerfile and add the commands you need to assemble the image.

The output of build process of Dockerfile is a Docker Image while a Container is a running Image, as shown in

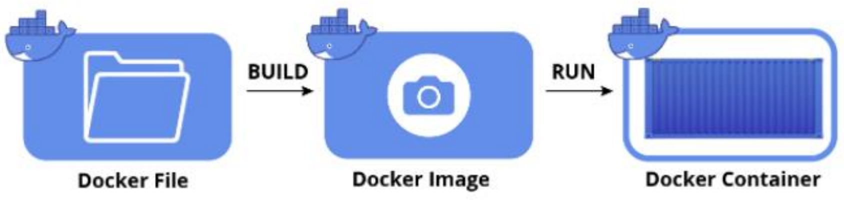


Figure 57 Steps to create a docker container

Compose is a tool for defining and running multi-container Docker applications. With Compose, it is possible to use a YAML file to configure an application's services. Then, with a single command, it is possible to create and start all the services from a custom configuration. Compose works in all environments: production, staging, development, testing, as well as CI workflows. Using Compose is basically a three-step process:

- Define the app's environment with a Dockerfile so it can be reproduced anywhere.
- Define the services that make up the app in docker-compose.yml so they can be run together in an isolated environment.
- Run docker-compose up and Compose starts and runs the entire app.

Container registries are catalogues of storage locations, known as repositories, where it is possible to push and pull container images. The three main types of registry are as follows:

- **Docker Hub**: Docker's own official image resource where it is possible to access more than 100,000 container images, shared by software vendors, open-source projects and Docker's community of users. It is possible also use the service to host and manage your own private images.
- **Third-party registry services**: Fully managed offerings that serve as a central point of access to your own container images, providing a way to store, manage and secure them without the operational headache of running your own on-premises registry.
- **Self-hosted registries**: A registry model favored by organizations that prefer to host container images on their own on-premises infrastructure— typically because of security, compliance or lower latency requirements.

## 3.4.2 Migration of Data Federation & Integration to Docker

The first prototype of Data Federation & Integration is released as Docker microservice where some containers are pulled from the Docker Hub registry while other ones are implemented ad hoc starting from the source code of the application. As shown in Figure 58 the DFI framework consists of six containers that can be launched with a single command using the implemented YAML docker-compose file. The image of keycloak, postgresql, maria db and gk-rdf4j are pulled the public Docker Hub registry while images for containers gk-integration-egine and gk-fhir-server are written from scratch. Following some details about each container.

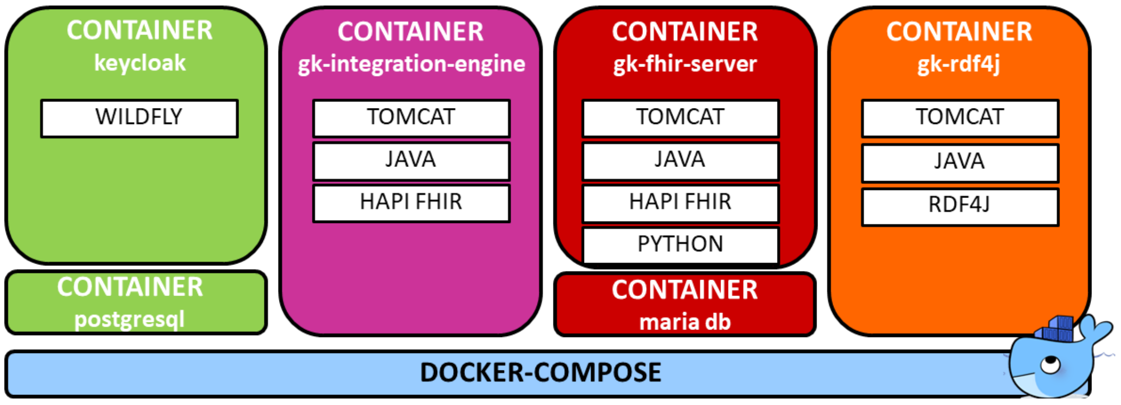


Figure 58 Data Federation & Integration in Docker

The keycloak container uses an instance of Wildfly [16] application server to expose its services and for persisting data into postgresql container, which in turn stores data in a Docker Volume. This is a useful configuration because data generated by postgresql container are not deleted when the container is cancelled.
The image of gk-rdf4j is pulled from the Docker Hub registry and configurated to be integrated into DFI framework. It is written in Java and deployed on an embedded Tomcat application server.
The image of gk-integration-engine is developed by ENG team, it is based on Tomcat, Java and HAPI-FHIR library. The output is a YAML Dockerfile. This image is pushed on the private ENG Docker registry so that it can be pulled and started when the Docker compose is launched.
Also, the gk-fhir-server container is developed by ENG team using the approach of Dockerfile and pushed on the private ENG Docker registry. On the image of this container is installed Tomcat and Java, to run the FHIR server, and python to run the routine to convert data from FHIR Json to RDF. gk-fhir-server uses the container maria-db to store data, such container persists data in a docker volume. Figure 59 shows Dockerfile written for gk-fhir-server.

```
# Start with a base image containing tomcat
FROM tomcat:9.0-alpine

RUN apk upgrade --update && apk add --no-cache python3 python3-dev gcc gfortran freetype-dev musl-dev libpng-dev g++ lapack-dev && apk add openjdk8

# install pip3
RUN pip3 install virtualenv

#install requests
RUN pip3 install requests


# install pip3 rdftofhir
RUN pip3 install fhirtordf

#Add "tests" sources in python installation folder to run fhirtordf library from cli
COPY fhirtordf-depencencies/tests /usr/lib/python3.6/site-packages/tests

# Add a volume pointing to /tmp
VOLUME /tmp

# Make port 8080 available to the world outside this container
EXPOSE 8080

COPY ./target/gk-fhir-server.war /usr/local/tomcat/webapps/

# Add Maintainer Info
LABEL maintainer="gatekeeper_mantainer"

CMD ["catalina.sh", "run"]
```

Figure 59 Dockerfile for gk-fhir-server

The Docker-compose file is developed in order to start all containers with one command. It pulls images of keycloak, postgresql, maria db and rdf4j from Docker Hub registry and the images of gk-integration-engine and gk-fhir-server from the private Docker registry installed on ENG server.
Due to COVID-19 situation, it was not possible to deploy DFI on HPE server, but it was been deployed on ENG server to order to perform some initial integration tests. As soon as the HPE infrastructure will be available, the DFI framework will be migrated to such infrastructure and installed in Kubernetes cluster as described in the next chapter.
PODs, running in the Kubernetes cluster, will use the implemented Docker containers.

## 3.5 Source code

The current prototype of the Data Federation & Integration is shared on the git ENG repository and it is going to be migrated to GK git repository as soon as it will be available. It consists of three projects: gk-integration-engine, gk-fhir-server and gk-docker.

The source code of gk-integration-engine is available at ENG Gitlab https://production.eng.it/gitlab/GTKEEPER_EU/gk-integration-engine. The used technologies are:

- JAVA 1.8 as programming language.
- Framework Spring [17].
- Apache Camel for the routing.
- Keycloak for the security.
- Docker to create the image of the software.
- HAPI FHIR library to implement the conversion from raw data to FHIR.
- RML library to convert raw data to RDF.

The source code of gk-fhir-server is available at ENG Gitlab https://production.eng.it/gitlab/GTKEEPER_EU/gk-fhir-server. It is based on the following technologies:

- JAVA 1.8 as programming language.

- HAPI FHIR library to implement the FHIR Rest APIs.
- Tomcat [18] as application server.
- Maria database as storage.
- Python to write the routine to convert fhir data to RDF.
- Docker to create the image of the software.

The source code of gk-docker is available at ENG Gitlab https://production.eng.it/gitlab/GTKEEPER_EU/gk-docker. It contains the docker-compose file that is start the docker images of the components belonging to DFI: gk-integration-engine, gk-fhir-server, maria database, keycloak, postgress, rdf4j workbench.

Moreover it contains the YAML file to deploy the all services on the Kubernetes Cluster.

# 4 Data federation and Integration V1: deployment environments

## 4.1 KUBERNETES OVERVIEW

The arrival on the market of containers and related microservices based applications, on the one hand enabled applications that quickly scale according to the requests and that could be easily updated, on the other hand meant that software previously managed as a single indivisible piece was split into several dozens of microservices (containers), making it more difficult to manage them.

In this context, the necessity to develop a tool that was able to manage the life-cycle of the microservices (deployment, scaling, and management) arose: such tool was developed by Google with the name of "Project Seven of Nine" and released as open source software in 2014. Today such tool is widely known as Kubernetes [4].

Kubernetes provides:

- **Service discovery and load balancing**
  Kubernetes can expose a container using the DNS name or using its own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.

- **Storage orchestration**
  Kubernetes allows to automatically mounting a storage system of different types, such as local storages, public cloud providers, and more.

- **Automated rollouts and rollbacks**
  You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new containers.

- **Automatic bin packing**
  You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.

- **Self-healing**
  Kubernetes restarts containers that fail, replaces containers, kills containers that do not respond to your user-defined health check, and does not advertise them to clients until they are ready to serve.

- **Secret and configuration management**
  Kubernetes lets you to store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.

### 4.1.1 KUBERNETES RESOURCE TYPES

Kubernetes has five main resource types that can be created and configured using a YAML or a JSON file:

**Pods**
Represent a collection of containers that share resources, such as IP addresses and persistent storage volumes. It is the basic unit of work for Kubernetes.

**Services**
Define a single IP/port combination that provides access to a pool of pods. By default, services connect clients to pods in a round-robin fashion.

**Replication Controllers**
A framework for defining pods that are meant to be horizontally scaled. A replication controller includes a pod definition that is to be replicated, and the pods created from it can be scheduled to different nodes.

**Persistent Volumes (PV)**
Provision persistent networked storage to pods that can be mounted inside a container to store data.

**Persistent Volume Claims (PVC)**
Represent a request for storage by a pod to Kubernetes.

## 4.2 GK Integration Engine

This section describes the architecture of the component gk-integration-engine, migrated to Kubernetes, that it is going to be deployed on HPE server (task 4.1). This configuration pulls the docker image of gk-integration-engine uploaded on the private docker registry of ENG server. Figure 60 shows the design of the architecture where the following k8s elements are used: “Namespace”, “Service”, “Ingress”, and “Deployment”.

The k8s configuration of gk-integration-engine consists of a POD, A Service and an Ingress, all these three components belonging to the same namespace named “datafederation”.

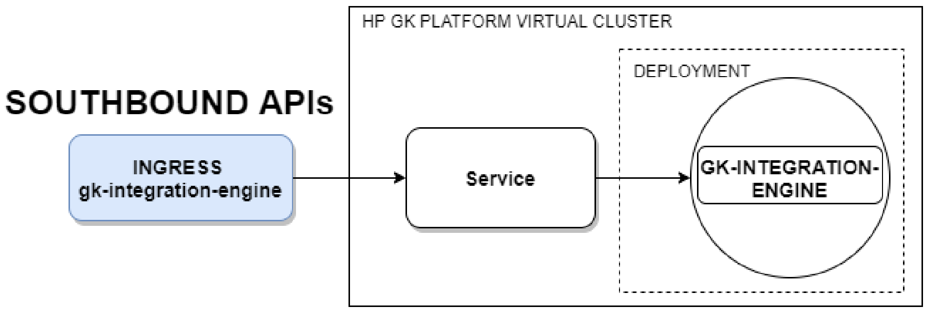
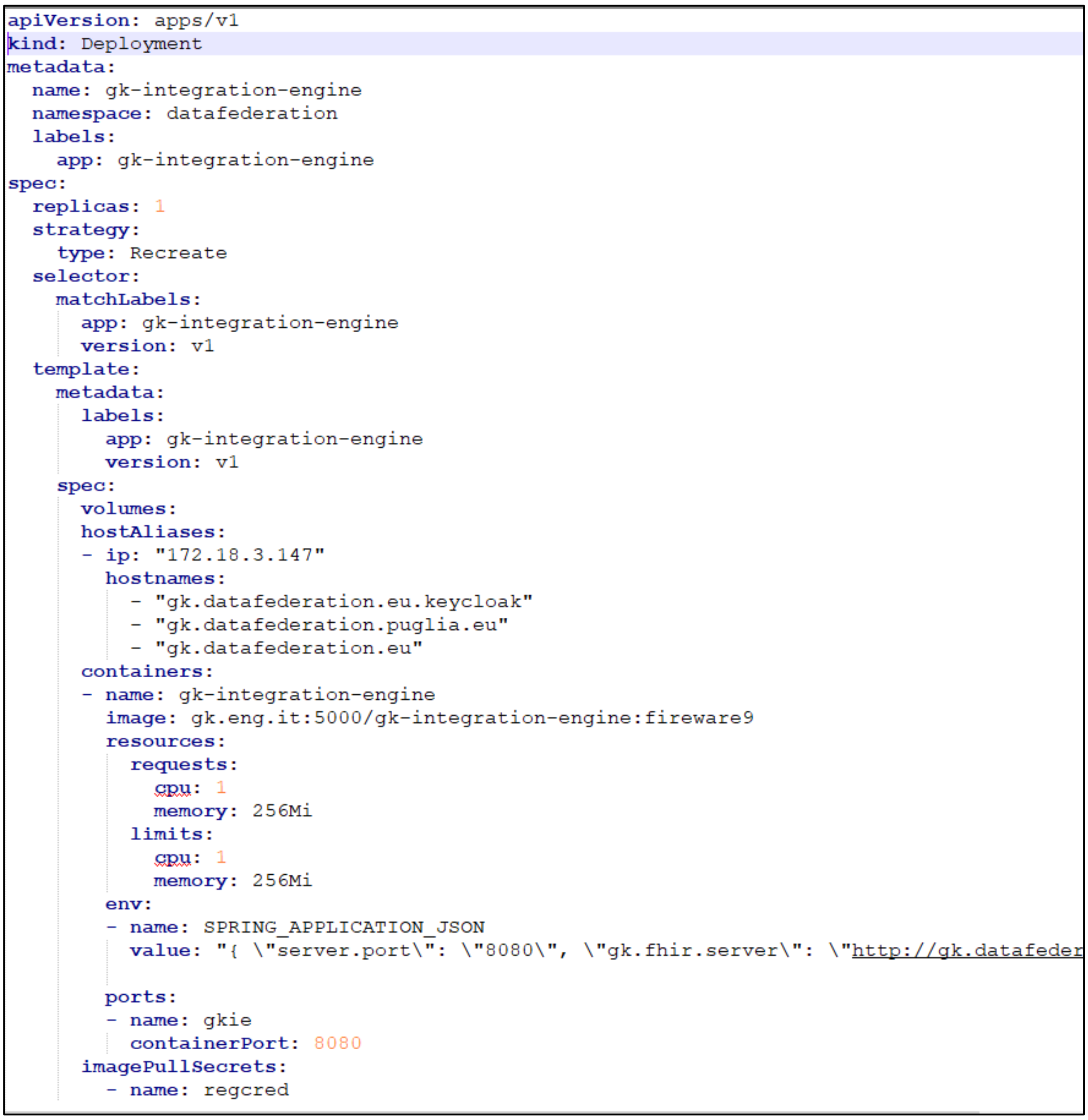


Figure 60 gk-integration-engine on Kubernetes

The POD contains the docker image gk-integration-engine pulled by the docker registry, together with the secret containing the docker registry credentials. In the figure above POD is represented by the cycle while the docker container is represented by the box inside the cycle. The whole configuration is set up inside a k8s deployment YAML file as shown in Figure 61[10]. As it is possible to see from the configuration, such POD does not use any volume, because there are not any persistent data to be store, it reads only one environment variable called SPRING_APPLICATION_JSON which value is a JSON containing needed information in order to run the application such as pilots servers FHIR and rdf4j endpoints for sending transformed data together with the keycloak configuration to verify the authorization and authentication controls.

Section resources contains the max limit of usage for CPU and memory.

[10] This is the local configuration



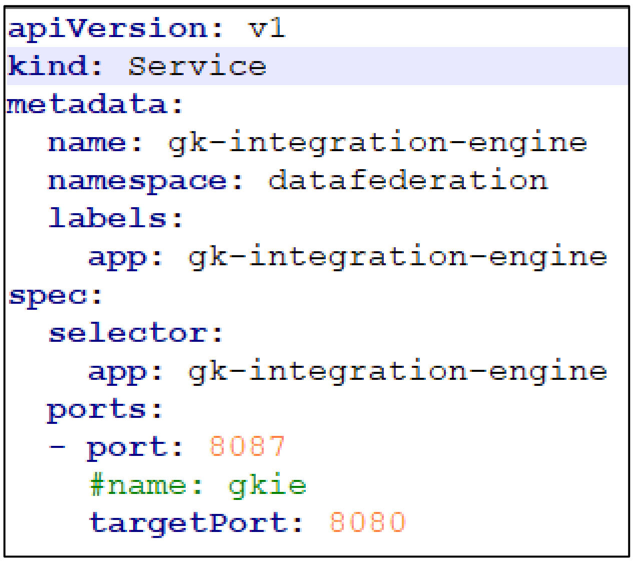
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gk-integration-engine
  namespace: datafederation
  labels:
    app: gk-integration-engine
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: gk-integration-engine
      version: v1
  template:
    metadata:
      labels:
        app: gk-integration-engine
        version: v1
    spec:
      volumes:
      hostAliases:
      - ip: "172.18.3.147"
        hostnames:
          - "gk.datafederation.eu.keycloak"
          - "gk.datafederation.puglia.eu"
          - "gk.datafederation.eu"
      containers:
      - name: gk-integration-engine
        image: gk.eng.it:5000/gk-integration-engine:fireware9
        resources:
          requests:
            cpu: 1
            memory: 256Mi
          limits:
            cpu: 1
            memory: 256Mi
        env:
        - name: SPRING_APPLICATION_JSON
          value: "{ \"server.port\": \"8080\", \"gk.fhir.server\": \"http://gk.datafeder

        ports:
        - name: gkie
          containerPort: 8080
      imagePullSecrets:
        - name: regcred
```

Figure 61 gk-integration-engine k8s deployment

To allow the integration of the PODs running the gk-integration-engine with the other PODs located in the same cluster, a service ClusterIP has been introduced.

A ClusterIP exposes the service on a cluster-internal IP. Choosing this value makes the service only reachable from within the cluster. This is the default simplest type, the default one. It opens access to an application within a cluster, without access from the world. All PODs, in the same cluster, can access to this service by a defined name and port without to specify the CLUSTER-IP that is dynamically assigned.

The Service is represented in the architecture with the rectangle labelled with *service*, and the relative YAML file is shown in Figure 62. The service maps the port 8087 to target port 8080 of the container. Thanks to this configuration, even if the gk-integration-engine is running on the port 8080, it is possible to access to the container by means port 8087 because the last one is mapped on 8080. The configuration of the service says that this service allows to access to the POD named “gk-integration-engine” (represented by file YAML deployment shown in previous figure) by means port 8087 (even in the docker container inside the POD run on port 8080).



```
apiVersion: v1
kind: Service
metadata:
  name: gk-integration-engine
  namespace: datafederation
  labels:
    app: gk-integration-engine
spec:
  selector:
    app: gk-integration-engine
  ports:
  - port: 8087
    #name: gkie
    targetPort: 8080
```

Figure 62 gk-integration-engine k8s service

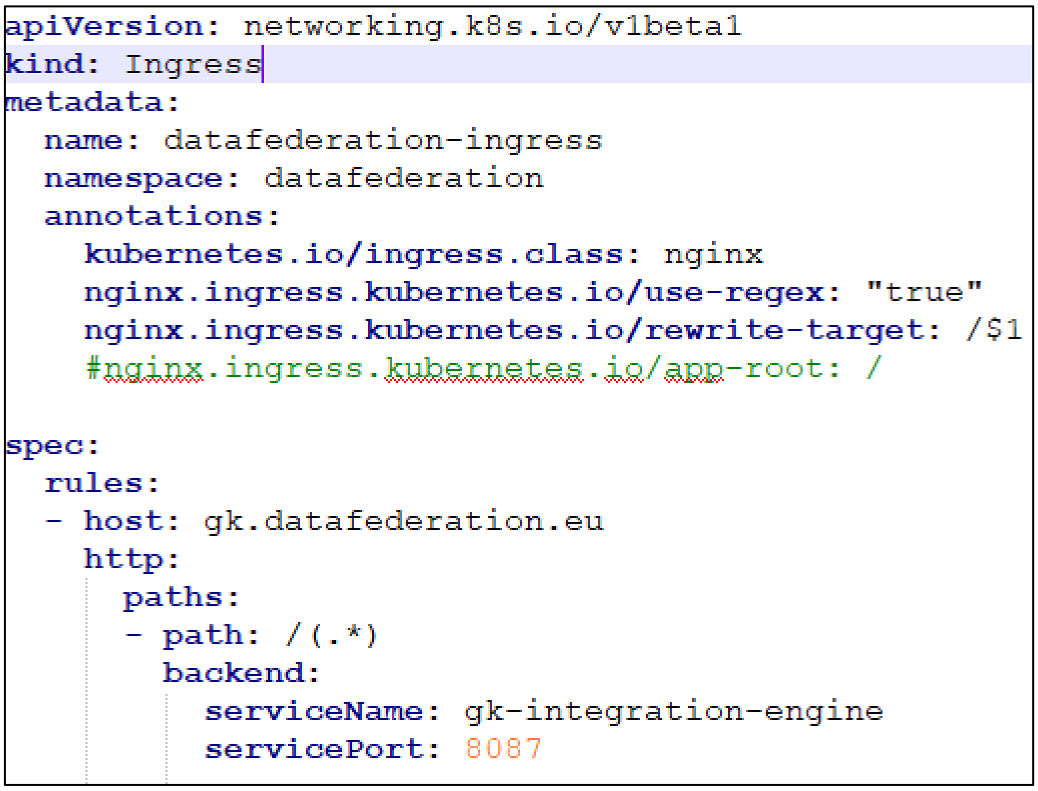
Services enables the communication among several PODs running in the same cluster.

Last element is the Ingress, in Figure 60 represented by a light blue colour rectangle labelled with ingress. In k8s the ingress allows the external application to access a POD ran the cluster using with defined host. Without Ingress and with a service ClusterIP, the POD is not reachable by application that run outside of the clyster.

According to the official documentation [19], an Ingress is an API object that manages external access to the services in a cluster (typically HTTP).

Ingress is not a type of Service, but rather an object that acts as a reverse proxy and single entry-point to the cluster that routes the request to different services. The most basic Ingress is the NGINX Ingress Controller [20], where the NGINX takes on the role of reverse proxy, while also functioning as SSL.

Figure 63 shows the YAML file representing the ingress for gk-integration-engine using a NGINX ingress controller.



```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: datafederation-ingress
  namespace: datafederation
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /$1
    #nginx.ingress.kubernetes.io/app-root: /

spec:
  rules:
  - host: gk.datafederation.eu
    http:
      paths:
      - path: /(.*)
        backend:
          serviceName: gk-integration-engine
          servicePort: 8087
```

Figure 63 gk-integration-engine k8s ingress

**Metadata** contains three attributes:

- **name**: is the name of the component, datafederation-ingress.
- **namespace**: is the namespace where this component (called datafederation-ingress) belongs to.
- **annotations** contain some configurations about the used ingress controller. In details "*kubernetes.io/ingress.class*" specifies the kind of the ingress, in this case "nginx". "*nginx.ingress.kubernetes.io/use-regex*" with value "true" means that is regular expression are enable and finally "*nginx.ingress.kubernetes.io/rewrite-target*" represent the paths that will be rewritten to the provided value[11].

[11] If an ingress has annotation ingress.kubernetes.io/rewrite-target: / and has path /tea, for example, the URI /tea will be rewritten to / before the request is sent to the backend service. Numbered capture groups are supported.

**Spec** section specifies the host name, in this case *gk.datafederation.eu* the gk-integration-engine can be reach by means this hostname since it is mapped to the service *gk-integration-engine* with the port *8087*.

Summarize the gk-integration-engine deployed in Kubernetes consists of docker image running in a POD. This POD can be reached by a service ClusterIP using a specific name and port. This service can be accessed by outside of the cluster with an Ingress component. When application outside of the cluster invokes the gk-integration-engine by means Ingress, such Ingress forwards the request to the relative service ClusterIP and finally service ClusterIP routes the request to specific POD.

Ingress represents the southbound APIs of the Data Federation and Integration that can be invoked by the pilot's applications in order to share their data.

## 4.3 GK FHIR Server and GK RDF4J Workbench

This section describes the design of the components gk-fhir-server and gk-rdf4j in Kubernetes. For each pilot is created a dedicated k8s namespace containing both their own private FHIR and RDF data, further information about this kind of deploy is provided in the next section, here it is described only the internal design of the default pilot virtual cluster (namespace) that will be assigned to each pilot involved in the project.

The k8s pilot namespace architecture consists of three PODs, three services and one ingress, all belonging to the same namespace as shown in Figure 64. Each POD is associated with a dedicated service ClusterIP and each POD can interact with another one by means of the service associated with it. External application can retrieve and stored data from gk-fhir-server and rdf4j through the ingress enabling the access to the cluster where PODs are running.

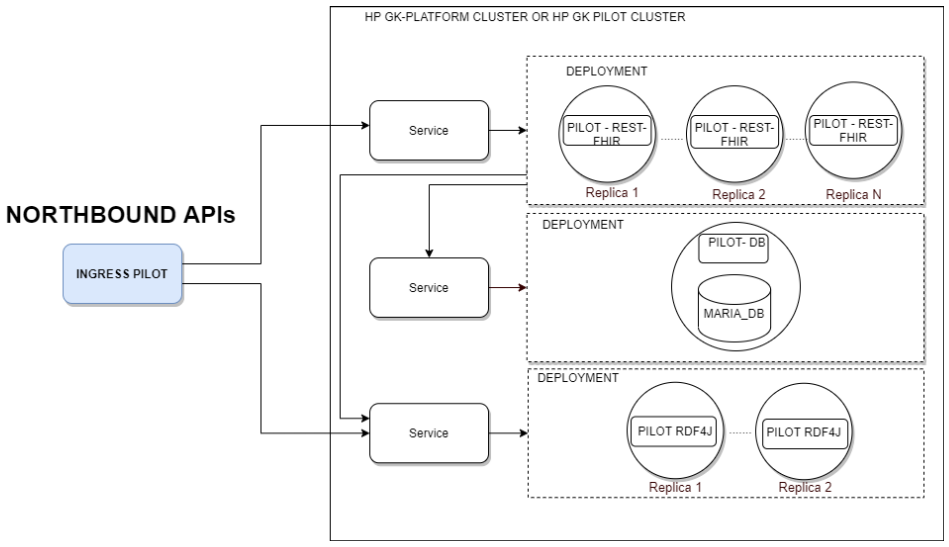


Figure 64 gk-fhir-server server and gk-rdf4j on Kubernetes

The gk-fhir-server consists of two PODs: one containing the docker image implementing the PILOT-REST-FHIR API to store and retrieve data and another one containing the docker image of Maria database [21]. These two PODs is associated with a dedicated service ClusterIP.

The reason why it was decided to have the maria database and fhir server in two different PODs is that the schema of the database is fixed, and it is not expected to change during the progress of the project so there is any needed for which every time the POD containing the PILOT-REST-FHIR is deleted and recreated, also the relative instance of the database must be recreated. This configuration improves performances.
PODs running the docker container for the REST FHIR APIs does not interact with the POD where the instance of maria data base is running but it interacts only by means it associated service ClusterIP. The advance to use service ClusterIP, to interact with the PODs, is that this communication can happen without knowing the IP of the node where the POD is running.

PILOT REST FHIR POD interactions also with PILOT-RDF4J to send transformed data into RDF format, also in this case the communication is realized by means the service CLUSTER IP associated to it avoiding setting the node IP each time this POD is delete and created. According to this architecture to each POD is associated a Service and the interactions among PODs in the same namespace can happen only through the Service and not directly with PODs.

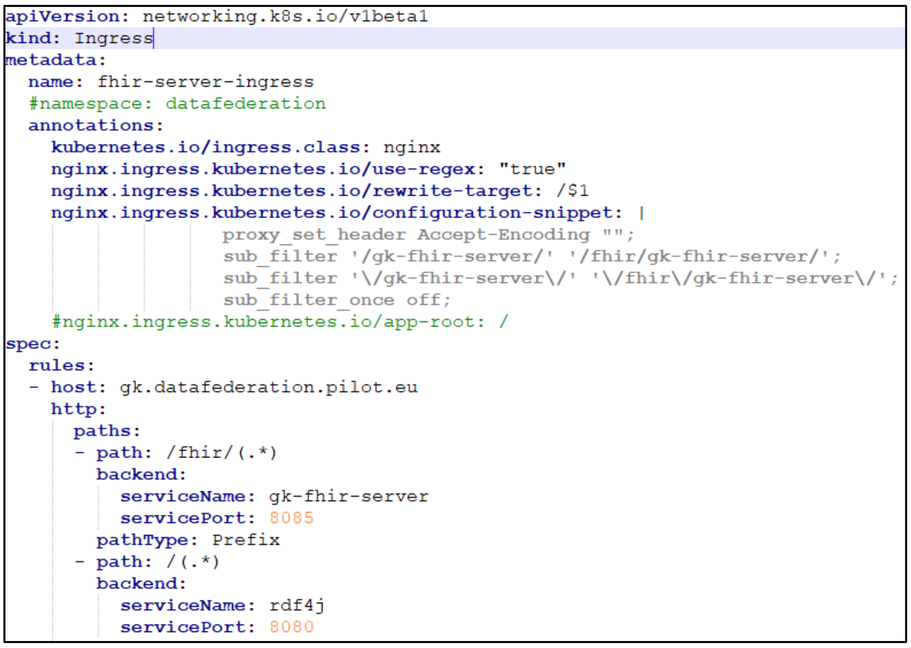
This approach simplifies the configuration of the endpoints with which a POD has to interact because it can be realized setting the name of the Service and port that are statically defined, they do not change even if a POD is delete and created many times.

The k8s environment for FHIR Server consists of four YAML files: PersistentVolumeClaim, ConfigMap, Deployment and Service. PeristentVolumeClam is used to persist in permanent way some logging information about the server fhir, it is used this k8s element so even if PODs is deleted and recreated such logging data are not loosed. It is important to underline that when a POD is deleted all information that are managed in the scope of this POD is deleted. To avoid this situation Kubernetes offers the element PersistentVolumeClam that allows to store persistent data in a volume, so they are never cancelled even if the POD is deleted and created many times. The element ConfigMap contains some configurations variables that are used by POD rest-fhir-server. In particular, it contains the service name and the ports for interacting with maria base (to store and retrieve fhir data) and rdf4j (to store FHIR data in RDF format). Service which defines the ClusterIP note to access the PILOT REST-FHIR POD. Finally, Deployment is responsible for keeping the rest-fhir POD running in the cluster, it pulls the docker image of the service from ENG docker registry.

The k8s environment for maria db consists of three yaml files: PersistentVolumeClaim, Deployment and Service. PersistentVolume is cluster resources that exists independently of POD. This means that the disk and data represented by a PersistentVolume continue to exist as the cluster changes and as Pods are deleted and recreated. PersistentVolume resources can be provisioned dynamically through PersistentVolumeClaims, or they can be explicitly created by a cluster administrator. Data in maria database are stored into a PersistentVolumeClaim so they are not deleted when a POD is dynamically deleted and created. Service enables network access to the POD in the cluster. Finally, Deployment containing all the necessary information for keeping the POD running, it pulls a docker image of mariadb from dockerhub.

Last element is RDF4J. The environment consists of three YAML files: PersistentVolumeClaim, Deployment and Service. PersistentVolumeClaim is used to store RDF data. Service specifies the name the port to access the POD and finally Deployment contains all the configuration to keep the POD running. Deployment pull the rdf4j image from dockerhub.

Last element of this namespace is Ingress. It defines the host paths for PILOT FHIR SERVER and PILOT RDF4J representing the northbound APIs for the Data Federation and Integration as shown in Figure 65. Annotation section contains some filter that are applied to web interfaces of FHIR server. The spec section defines the path that can be used to connect the specific Service. In the figure path http://gk.datafederation.pilot.eu/fhir/ forwards to the backend of the Service of rest-fhir-server POD (name gk-fhir-server and port 8085) while path http://gk.datafederation.pilot.eu/ forwards to the backend of the Service pilot rdf4j (name rdf4j and port 8080).



```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: fhir-server-ingress
  #namespace: datafederation
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /$1
    nginx.ingress.kubernetes.io/configuration-snippet: |
                  proxy_set_header Accept-Encoding "";
                  sub_filter '/gk-fhir-server/' '/fhir/gk-fhir-server/';
                  sub_filter '\/gk-fhir-server\/' '\/fhir\/gk-fhir-server\/';
                  sub_filter_once off;
    #nginx.ingress.kubernetes.io/app-root: /
spec:
  rules:
  - host: gk.datafederation.pilot.eu
    http:
      paths:
      - path: /fhir/(.*)
        backend:
          serviceName: gk-fhir-server
          servicePort: 8085
        pathType: Prefix
      - path: /(.*)
        backend:
          serviceName: rdf4j
          servicePort: 8080
```

Figure 65 K8S Ingress PILOT namespace

## 4.4KEYCLOAK

The section shows how the keycloak [9] component is deployed on Kubernetes framework. The integration and interaction with this tool have been exploited for testing purpose only. In production all invocations to southbound and northbound APIs are expected to be trusted since the interaction with the Data Federation & Integration is mediated by GTA. Figure 66 shows the architecture of Keycloak framework deployed on Kubernetes cluster where all the components belong to the same namespace. Cluster consists of two PODs one for the keycloak web services and one for the database, for each one is implemented a deployment YAML file. The deployment YAML file pulls the keycloak docker image from dockerhub repository and it contains all the information needed to keep the POD running. The interaction with Postgress [22] database is realized by means the service associated to it. The Services created in this namespace are of type ClusterIP, this means that services can be accessed by other PODs in the same cluster, in this way the PODs, keycloak and postgress, expose their services only to the PODs placed in the same cluster. Keycloak POD can access to the postgres database by the ClusterIP node through a static name and port.

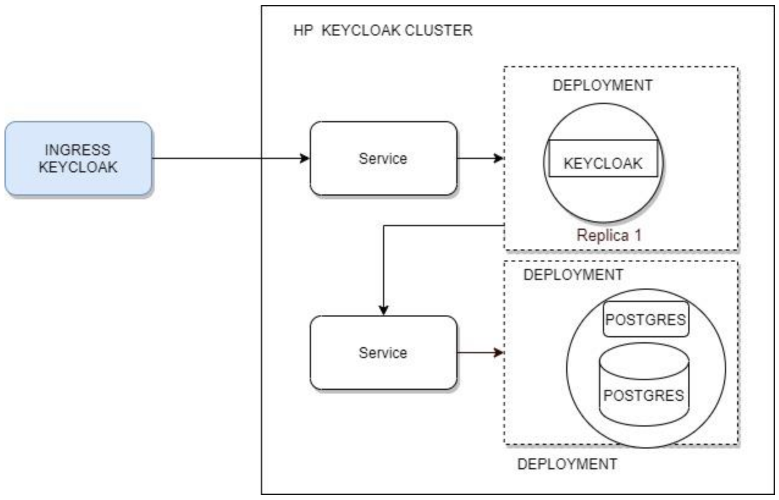


Figure 66 Keycloak on Kubernetes

The deployment of Postgres database consists of three yaml files: Deployment, Service and PersistentVolumeClaim. Deployment pulls the image of postgres database from docker hub registry and contains all the necessary configurations to keep the POD running. PersistentVolumeClaim is used to store in a volume data persisted by the POD keycloak into repository, in this way even if the PODs running the postgres container is deleted, data inside the database are not cancelled. The third element is the Service that allows the interaction with keycloak POD.

Finally, the Ingress component provides routing rules to manage external user's access to the services in the Kubernetes cluster. It provides both Externally reachable URLs for applications deployed in Kubernetes clusters and Name-based virtual host and URI-based routing support.

## 4.5 Deployment scenarios

This section provides an overview of the possible deployment alternatives of northbound API of the Data Federation & Integration into k8s cluster. DFI will be deployed the Kubernetes framework installed on the cloud infrastructure provided by task 4.1. Kubernetes is an open-source system aiming to automatize the deployment, to scale, and to manage of containerized applications.

Figure 68 gives a general overview about how the GK Platform can be deployed on HPE GK CLOUD. All main components of the platform are hosted in GK cloud at HPE data centre in located in Rome in a single reference tenant and connects to pilot sites to fetch data and provide results. Security, updates and maintenance can be managed centrally ensuring the highest level of service. GATEKEEPER Platform will be responsible to ensure separation of data and multitenancy. All the accesses from external applications to GK platform are managed by TMS (task 4.2) and GTA (task 4.5). As stated by the deliverable 3.2 GK offers, to the pilots involved in project, three alternatives of deployment:

1. **Pilots own a private space (virtual cluster) on GK Cloud and share some data with GK Platform.** In case Pilots require to keep part of their data isolated from the other pilots, GK Cloud can provide private storage spaces in dedicated private "*pilot cloud tenants*", while the GK Platform remain centralized. Pilot systems running in the separate spaces interact with the Platform to exploit its services from within GK Cloud. Figure 67 shows this alternative where data shared with

Data Federation & Integration and persisted in fhir and rdf format in a private and dedicated space (virtual cluster). In this configuration it is assigned a private virtual cluster to each pilot containing only their data, such data can be accessed only by the pilots own of the cluster. When a specific pilot invokes sourthbound APIs of DFI to share their data, this one is able to retrieve the pilot sources and accepts date, transform to FHIR and RDF format and finally forward them to the specific virtual cluster belonging to the pilot who sent data. According this configuration each pilot has only access to its private data. The access to the data of other pilots is forbidden.

2. **Pilots own a private space on GK Cloud and share some data with GK Platform.** To ensure a greater isolation, an alternative deployment implies the creation of separated "*pilot cloud tenants*" within the HPE data centre, where replicas of the GATEKEEPER platform are deployed separately (not only storage as in the solution above). In this solution, maintenance of the GK Cloud becomes more complex.

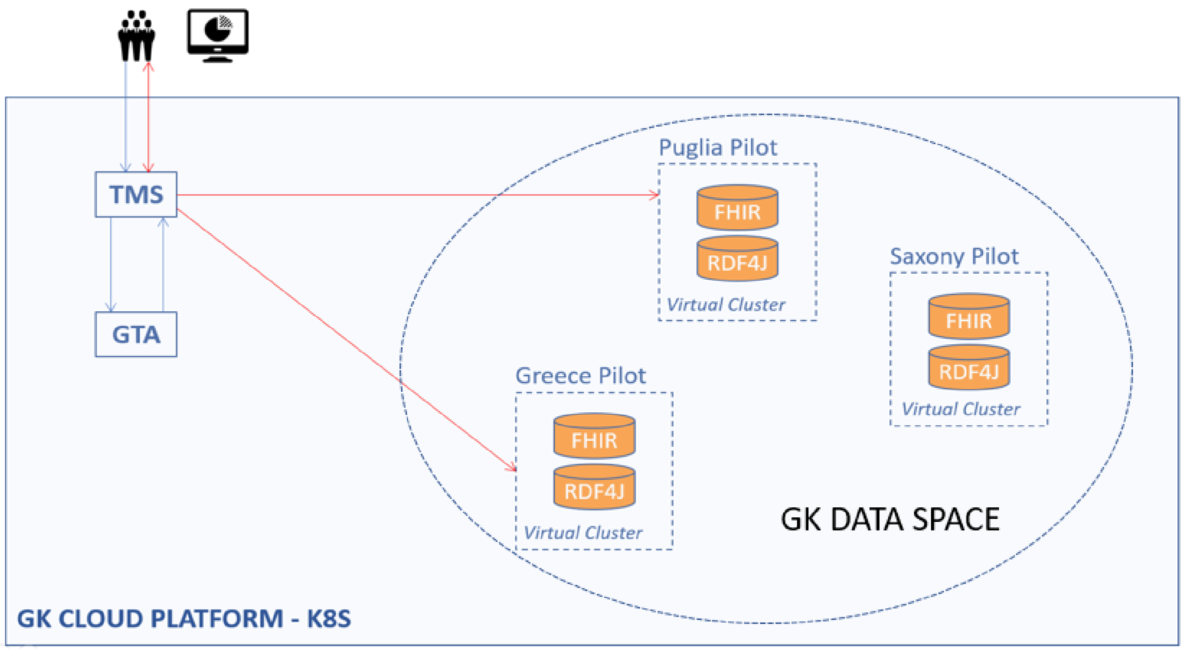


Figure 67 GK CLOUD PLATFORM - K8S

One of the requirements provided by the pilots invoked in the project is to have a dedicated and private space where store their data. This represents the first alternative of the GK deploy provided in the deliverable 3.2. Due to this required the architecture of Data Federation & Integration is designed in order to satisfy this request. Figure 68 shows the final deployment schema of the Data Federation & Integration on K8S cluster.

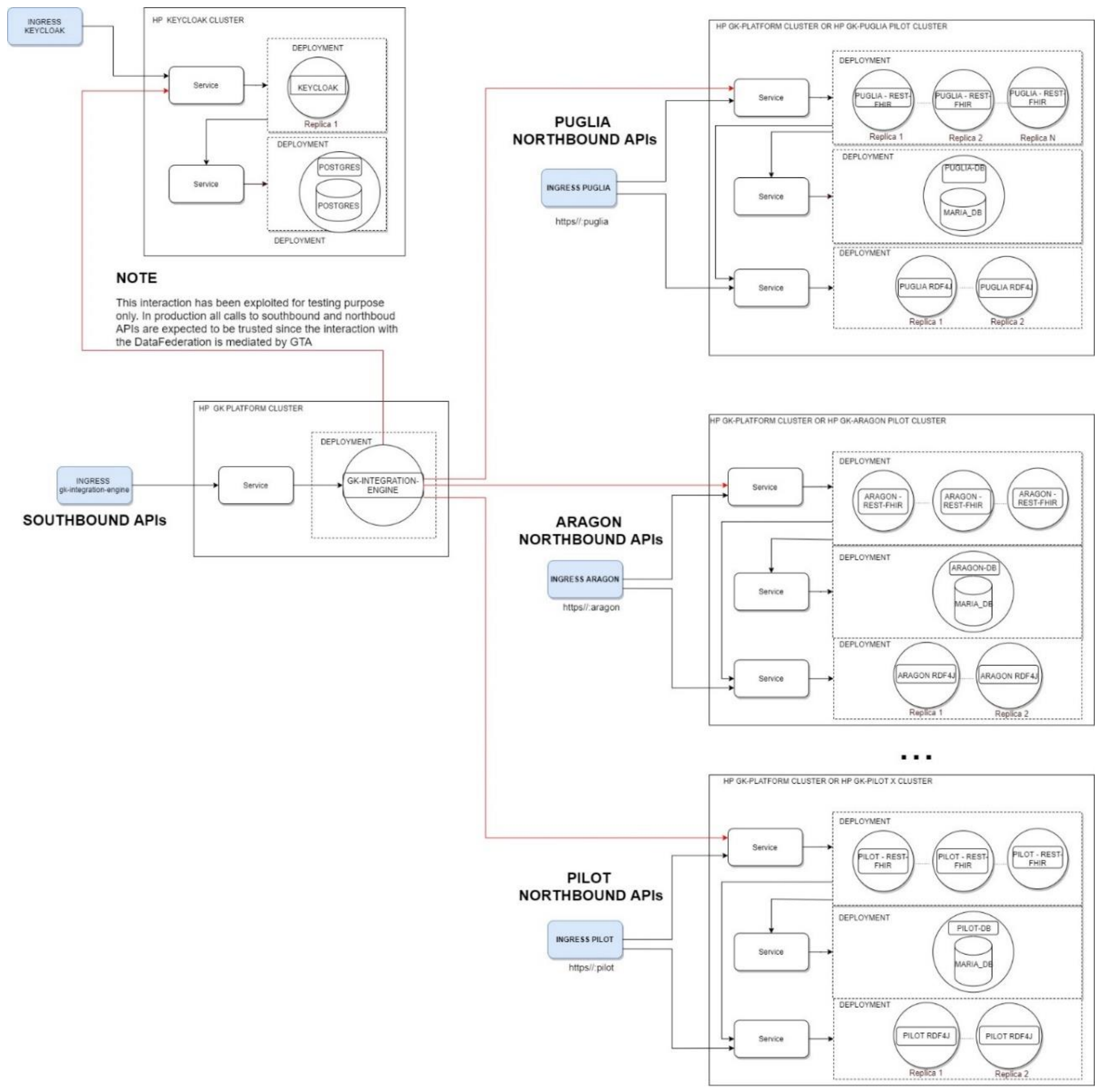


Figure 68 Data Federation & Integration deployed in Kubernetes cluster

The configuration combines all the single components of DFI described in the previous sections, i.e., gk-integration-engine, keycloak, gk-fhir-server and gk-rdf4j. All components are deployed in the same cluster but, for each of them, a dedicated namespace has been defined. Logically it is possible identify the following “virtual cluster”:

- “Virtual cluster” for gk-integration-engine containing the southbound APIs
- “Virtual cluster” for keycloak
- A dedicate “Virtual cluster” for each pilot involved in the project (one for Puglia, one for Aragon and so on) containing the gk-fhir-server and gk-rdf4j representing the northbound APIs of the DFI.

In the Figure 68 on the left there are the “virtual clusters” of gk-integration-engine and keycloak while on the right are point out the “virtual cluster” of pilot Puglia and Aragon[12].

Each “virtual cluster” has a specific k8s namespace, in detail there is a k8s namespace for gk-integration-engine module, a k8s namespace for keycloak module, a k8s namespace for pilot Puglia, a k8s namespace for pilot Aragon, a k8s namespace for pilot Greece, a k8s namespace for pilot Cyprus, a k8s namespace for pilot UK, a k8s namespace for pilot Poland, a k8s namespace for pilot Basque Country and a k8s namespace for pilot Saxony.

The “virtual cluster” of the pilot are replica of the k8s YAML files but with a different namespace. The interaction among all PODs inside the whole cluster is made with ClusterIP node, implemented with k8s Services. A ClusterIP provides a cluster IP address accessible only by other PODs and services within in the cluster. No external IP address is created for the application. To access a POD underlying a cluster service, other applications in the cluster can use the ClusterIP address of the service or send a request using the service name. When reached by requests, the service forwards them to the pods equally, regardless of the clustered IP addresses of the pods and the worker node on which they are deployed. if it is not specified a type in a service YAML configuration file, the ClusterIP type is created by default.

With this configuration all PODs inside the DFI cluster can interaction by mean ClusterIP node using the assigned name and port without setting the IP of the node. The advantage of this configuration is that the configuration YAML file to run the PODs must not be updated each time the IP of the node changes because the interaction among all the PODs is realized using the name and the port associated by the ClusterIP node that are statically defined.

When the POD for gk-integration-engine is started, it reads from the spring_json environment variable the service names and ports of each of POD the gk-fhir-server and gk-rd4j associated to each pilot “virtual cluster” deployed in the HPE cloud infrastructure. Thank to this configuration even if some POD of gk-fhir-server and rdf4j is deleted and recreated, it is not needed to restart the PODs of gk-fhir-server to reload the endpoint of the created POD of fhir-server and rdf4j because they are statically defined into server ClusterIP.

[12] Box labelled with PILOT NORTH BOUND APIs present a generic pilot “virtual cluster”.

The access point among Pilots applications and DFI Cloud platform are the k8s Ingresses. Such ingresses enable the interactions of the PODs running inside the DFI cluster with the external applications using defined host names.

Following there are step performed a pilot X to share its data with DFI and steps followed by FDI to store them in GK cloud Service:

1. Pilot X invokes the APIs defined in K8S ingress to retrieve the bearer token.
    a. Ingress forwarding the request to the service ClusterIP that checks if the credential sent by the PILOT are correct.
2. Keycloak returns the bearer token to the Pilot X (if credential is right).
3. Pilot X invokes one of two defined southbound APIs defined by the ingress of gk-integration-engine passing the bearer token in the header of the request and data in body.
4. Gk-integration-engine interact with the service ClusterIP of keycloak to check if the token passed by pilot X is correct.
5. gk-integration-engine has an internal apache camel routine that is able to identify the pilot that is sending data. Based on this information data are forward to the private "virtual cluster" of the pilot invoking the FHIR REST API by mean the service ClusterIP of the gk-fhir-server.
6. POD rest-fhir-server persists data in maria database in FHIR format, convert them to RDF and invokes the API of service clusterIP of the POD rdf4j that persist them.
7. Pilot X can access to persisted FHIR and RDF data by means the northbound APIs hosted on the Ingress of the "virtual cluster" of pilot X.

As already stated before, the interaction with "virtual cluster" of keycloak has been introduced only for testing purpose. In production all calls to southbound and northbound APIs are expected to be trusted since the interaction with the DataFederation & Integration is mediated by GTA.

The described deployed of Data Federation & Integration in k8s cluster has been tested in a local notebook using a Microk8s distribution [23] since at the moment of writing the deliverable the infrastructure provided by task 4.1 is not yet available. MicroK8s is a powerful, lightweight, reliable production-ready Kubernetes distribution. It is an enterprise-grade Kubernetes distribution that has a small disk and memory footprint while offering carefully selected add-ons out-the-box, such as Istio, Knative, Grafana, Cilium and more.

Next version of this deliverable will describe the full Data Federation & Integration migrated to the Kubernetes framework installed on the HPE infrastructure. At the moment of the writing this deliverable no test is performed in production.

# 5 Conclusion

This deliverable is providing the initial version of Data Federation & Integration as well as the description of how it will interact with the overall Gatekeeper platform. Starting from the requirements collected during the several remote calls made with each pilot involved in the project, the DFI design has been defined together with the first prototype as a microservice framework, consisting of three main components (gk-integration-engine, gk-fhir-server and gk-rdf4j), that expose specific southbound and northbound APIs to collect heterogeneous data coming from electronic health records and devices in order to convert and store such data into FHIR server and RDF repository according to the GK-FHIR-Profile, defined in the task 3.5.

Data Federation & Integration component has been dockerized and deployed on private ENG server to perform some initial integration tests using the Auth 2.0 authentication implemented with keycloak tool.

Thanks to DFI framework a common semantic model, based on HL7-FHIR, is defined that can be used to retrieve and process persisted data, hiding the problem of having them in heterogeneous formats since they come from different applications where each one uses a different model representation. The main advantage of this approach is that each task can play with Gatekeep data only knowing the defined GK-FHIR-Profile and not the specific models used by the pilot's applications. Next version of this deliverable will provide the transformation rules between the data model define in the task 3.4 to the GK-FHIR-Profile defined in the task 3.5.

# 6 References

[1] "HL7-FHIR," [Online]. Available: https://www.hl7.org/fhir/.

[2] "W3C," [Online]. Available: https://www.w3.org/.

[3] "URI," [Online]. Available: https://en.wikipedia.org/wiki/Uniform_Resource_Identifier.

[4] "Dublin Core Elements," [Online]. Available: http://purl.oclc.org/metadata/dublin_core_elements.

[5] "Resource Description Framework (RDF) Model and Syntax," [Online]. Available: http://www.w3.org/RDF/Group/WD-rdf-syntax/.

[6] "cCard Home Page," [Online]. Available: http://www.imc.org/pdi .

[7] "RML," [Online]. Available: https://rml.io/specs/rml/.

[8] K. J. Dittrich, "All Together Now — Towards Integrating the World's Information Systems," *dvances in Multimedia and Databases for the New Century*, p. 109–123, 1999.

[9] "keycloak," [Online]. Available: https://www.keycloak.org/.

[10] "RML," [Online]. Available: https://rml.io/docs/.

[11] "Apache Camel FHIR Component," [Online]. Available: https://camel.apache.org/components/latest/fhir-component.html.

[12] "HAPI FHIR," [Online]. Available: https://hapifhir.io/.

[13] "fhirtordf," [Online]. Available: https://github.com/BD2KOnFHIR/fhirtordf.

[14] "Docker," [Online]. Available: https://www.docker.com/.

[15] "Docker Container," [Online]. Available: https://www.docker.com/resources/what-container.

[16] "Wildfly," [Online]. Available: https://www.wildfly.org/.

[17] "Spring Java," [Online]. Available: https://spring.io/.

[18] "Tomcat," [Online]. Available: http://tomcat.apache.org/.

[19] "Kubernetes ingress," [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/ingress/.

[20] "NGINX Ingress Controller," [Online]. Available: https://kubernetes.github.io/ingress-nginx/.

[21] “Maria DB,” [Online]. Available: https://mariadb.org/.

[22] “PostgressSQL,” [Online]. Available: https://www.postgresql.org/.

[23] “Microk8s,” [Online]. Available: https://microk8s.io/.

[24] “Dublin Core,” [Online]. Available: https://dublincore.org/.

[25] “IMS,” [Online]. Available: http://www.imsglobal.org/metadata/index.html.

# Appendix A

**Instructions to add a new conversion in the Data Federation**

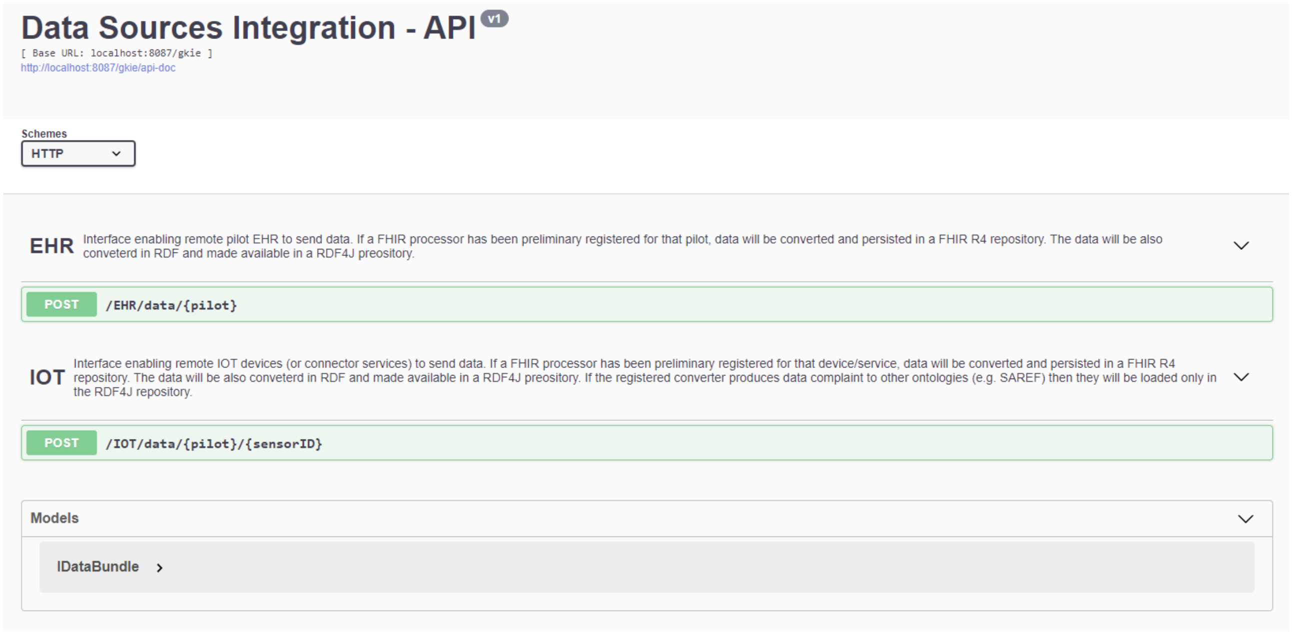
## A.1 General description of Data Federation.

Data Federation is a platform consisting of 3 main components:

- **GK-integration engine:** it is able to accepts data coming from heterogeneous data source, convert them into Bundle FHIR and invoke the GK-FHIR Server APIs to store transformed data.
- **GK-FHIR Server:** FHIR Server providing API according the FHIR standard **version R4**
- **GK-RDFJ4-WORKBENCH:** Repository containing persisted FHIR data in rdf format. It provides a set of API to retrieve information.

## A.2 GK-integration engine

GK-Integration engine is a Maven Java project developed with Spring boot. It provides mainly two kinds of Rest APIs, as showed in the following figure, that accept raw data sent from heterogeneous data sources, convert them into GK-FHIR compliant format and persist such data into the FHIR server for the storage. In the figure below a screenshot of the API documentation provided via Swagger.



Data Sources Integration - API v1

```
[ Base URL: localhost:8087/gkie ]
```

http://localhost:8087/gkie/api-doc

Schemes: HTTP

**EHR** Interface enabling remote pilot EHR to send data. If a FHIR processor has been preliminary registered for that pilot, data will be converted and persisted in a FHIR R4 repository. The data will be also conveterd in RDF and made available in a RDF4J preository.

POST `/EHR/data/{pilot}`

**IOT** Interface enabling remote IOT devices (or connector services) to send data. If a FHIR processor has been preliminary registered for that device/service, data will be converted and persisted in a FHIR R4 repository. The data will be also conveterd in RDF and made available in a RDF4J preository. If the registered converter produces data complaint to other ontologies (e.g. SAREF) then they will be loaded only in the RDF4J repository.

POST `/IOT/data/{pilot}/{sensorID}`

Models

IDataBundle

Figure 69 OpenAPI Data Federation Integration Engine

In order to convert the sent data, the integration engine, internally, retrieves the specific **converter** associated to the specific “data source”. The “data source” identifier corresponds to the {pilot} path parameter for the EHR interface while it is a combination of {pilot} and {sensorID} path parameters for the IOT interface.

### A.2.1How to build a new converter

If it is needed to provide a new **converter** for a new data source (it doesn't matter if such data source is going to call EHR or IOT interface), it is needed to implement a new converter following Data Federation framework guidelines. In order to speed up the process a sample eclipse project, with all the needed dependencies already in place, is provided [1]. Once downloaded and imported in eclipse it is simply needed:

**Step 1**: to provide the java data model used to deserialize data sent by the remote data source.

**Step 2**: to provide the converter which will include the logic for transforming the deserialized data (see step 1) in GK-FHIR compliant format.

**Step 3**: preform a test to check the capability of the new converter to properly work

Here below the details of the two steps.

### A.2.2Step 1 details

Browse the sample project and go to:

```
it.eng.gk.dataintegration.model.<pilot name>
```

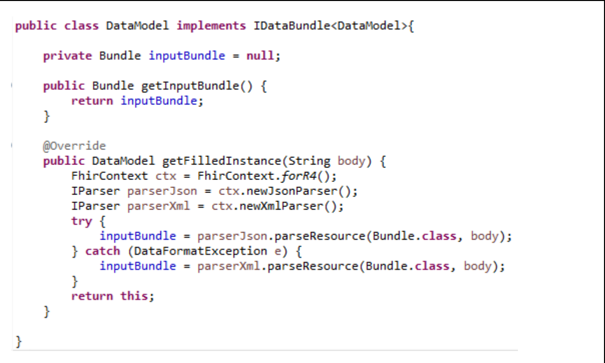
or to

```
it.eng.gk.dataintegration.model.<sensor ID>
```

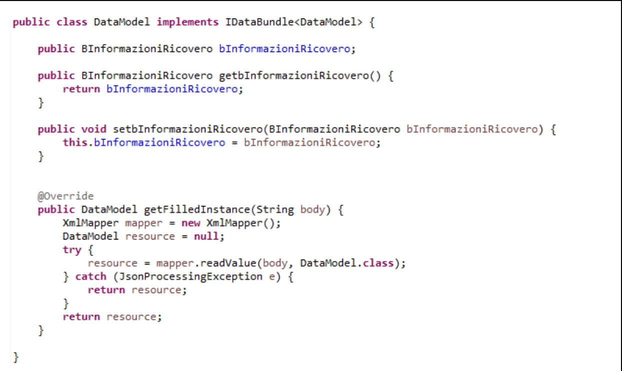
respectively if the data source we want to federate is an EHR or an IoT sensor (or IoT sensor gateway). The sample project already contains such packages based on our knowledge of the project but they could be easily extended in the case. For instance if the data source to federate is  the Samsung IoT gateway android app, then it is needed to access to:

```
it.eng.gk.dataintegration.model.samsung
```

and modify the class (DataModel.java) by overriding the method getFilledInstance that is appointed to return an instance of the model, valorized with the data received in the request body. If the DataModel.java depends on further classes they can also be added in the same package. The important thing is that they include getter and setter methods as for JAVA BEAN specification . Here below two examples. On the left side an implementation in the case the string sent with the body request is itself a FHIR bundle. On the right instead an example when the body is a generic model in XML.



```
public class DataModel implements IDataBundle<DataModel>{

    private Bundle inputBundle = null;

    public Bundle getInputBundle() {
        return inputBundle;
    }

    @Override
    public DataModel getFilledInstance(String body) {
        FhirContext ctx = FhirContext.forR4();
        IParser parserJson = ctx.newJsonParser();
        IParser parserXml = ctx.newXmlParser();
        try {
            inputBundle = parserJson.parseResource(Bundle.class, body);
        } catch (DataFormatException e) {
            inputBundle = parserXml.parseResource(Bundle.class, body);
        }
        return this;
    }

}
```



```
public class DataModel implements IDataBundle<DataModel> {

    public BInformazioniRicovero bInformazioniRicovero;

    public BInformazioniRicovero getbInformazioniRicovero() {
        return bInformazioniRicovero;
    }

    public void setbInformazioniRicovero(BInformazioniRicovero bInformazioniRicovero) {
        this.bInformazioniRicovero = bInformazioniRicovero;
    }


    @Override
    public DataModel getFilledInstance(String body) {
        XmlMapper mapper = new XmlMapper();
        DataModel resource = null;
        try {
            resource = mapper.readValue(body, DataModel.class);
        } catch (JsonProcessingException e) {
            return resource;
        }
        return resource;
    }

}
```

## A.2.3Step 2 details

Once the model has been completed it is possible to develop the converter that is appointed to implement the transformation  from the defined JAVA model (see step 1) to BUNDLE FHIR of **type TRANSACTION**.  Browse the sample project and go to:

`it.eng.gk.dataintegration.converters.<pilot name>`

or to

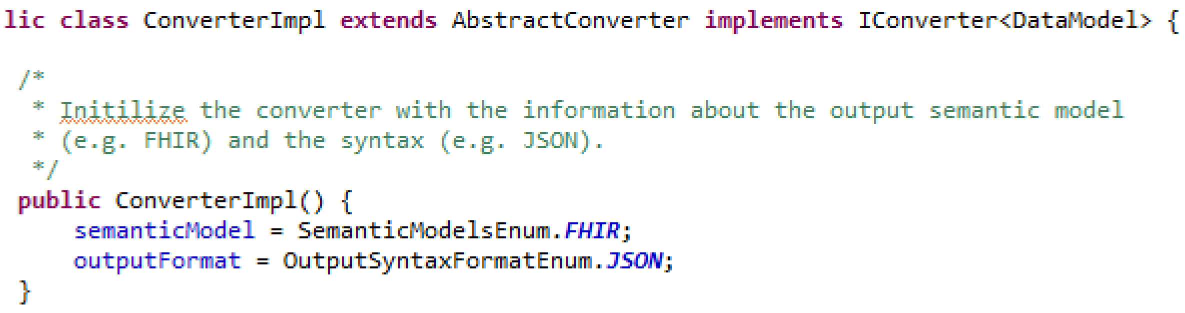
`it.eng.gk.dataintegration.converters.<pilot name>_<source ID>`

respectively if the data source we want to federate is an EHR or an IoT sensor (or IoT sensor gateway). The sample project already contains such packages based on our knowledge of the project but they could be easily extended in the case. For instance if the data source to federate is  the Samsung IoT gateway android app, then it is needed to move to:

`it.eng.gk.dataintegration.converters.saxony_samsung`

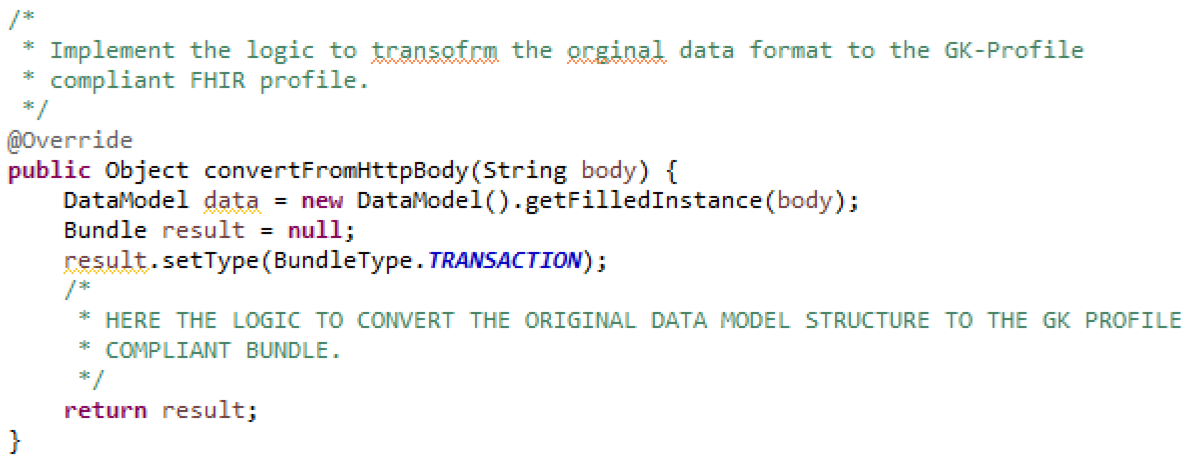
and modify the class (ConverterImpl.java) by:

1. Filling the constructor by initializing the attributes (semanticModel and outputFormat). This information (and related getter(s) methods) are exploited by the engine for routing purposes. Here below an excerpt:



```
lic class ConverterImpl extends AbstractConverter implements IConverter<DataModel> {

 /*
  * Initilize the converter with the information about the output semantic model
  * (e.g. FHIR) and the syntax (e.g. JSON).
  */
 public ConverterImpl() {
     semanticModel = SemanticModelsEnum.FHIR;
     outputFormat = OutputSyntaxFormatEnum.JSON;
 }
```

2. Overriding the method convertFromHttpBody that is appointed to perform the data format transformation. Here below an excerpt:

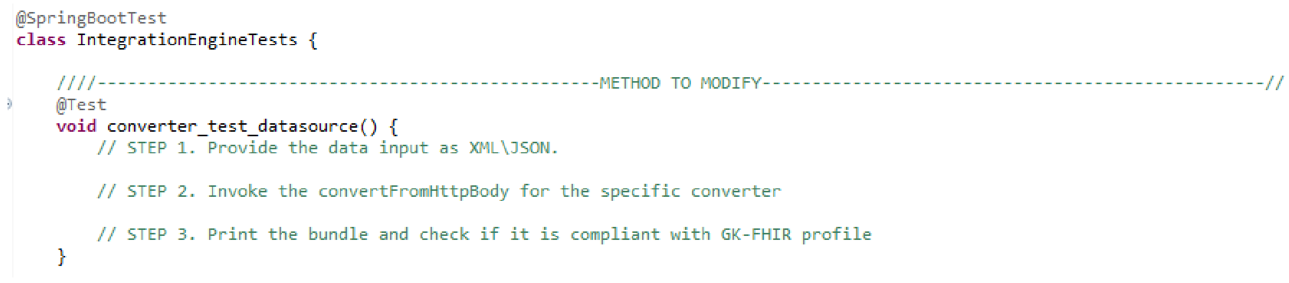


```
/*
 * Implement the logic to transofrm the orginal data format to the GK-Profile
 * compliant FHIR profile.
 */
@Override
public Object convertFromHttpBody(String body) {
    DataModel data = new DataModel().getFilledInstance(body);
    Bundle result = null;
    result.setType(BundleType.TRANSACTION);
    /*
     * HERE THE LOGIC TO CONVERT THE ORIGINAL DATA MODEL STRUCTURE TO THE GK PROFILE
     * COMPLIANT BUNDLE.
     */
    return result;
}
```

## A.2.4Step 3 details

Go to the folder: **src/test/java** and in the **package it.eng.gk.dataintegration**. Here you find a test class (IntegrationEngineTests.java) containing three sections:

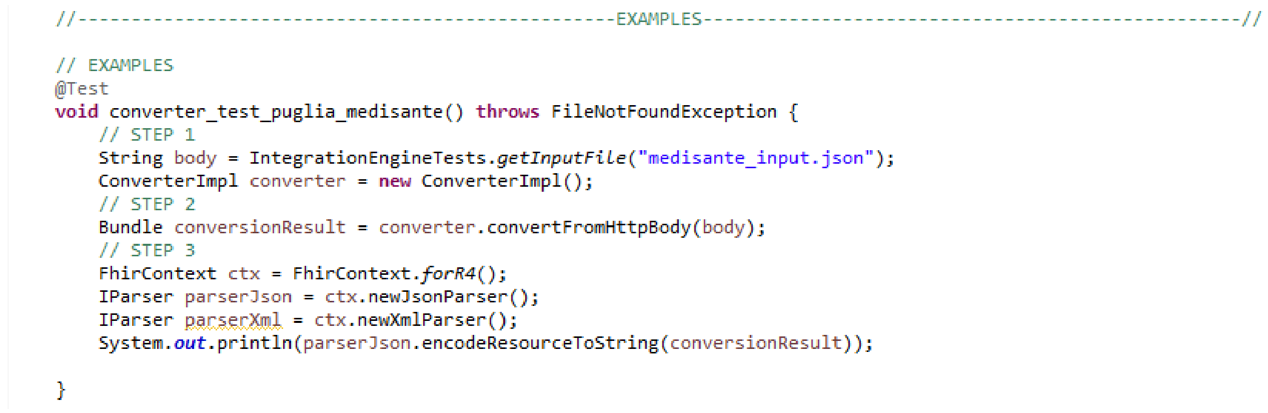
- The method you can override:



```
@SpringBootTest
class IntegrationEngineTests {

    ////-----------------------------------------------METHOD TO MODIFY-----------------------------------------------//
    @Test
    void converter_test_datasource() {
        // STEP 1. Provide the data input as XML\JSON.

        // STEP 2. Invoke the convertFromHttpBody for the specific converter

        // STEP 3. Print the bundle and check if it is compliant with GK-FHIR profile
    }
```

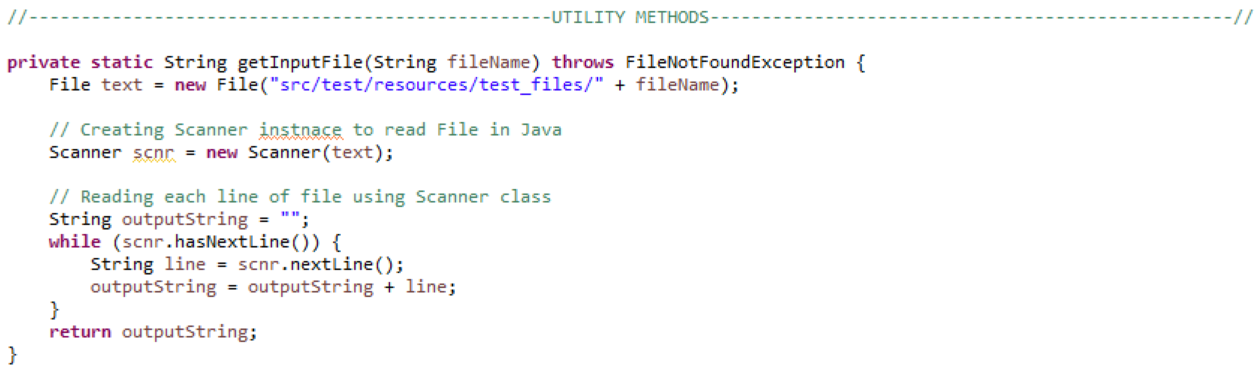
  The compliance must me checked by analyzing the produced FHIR bundle.

- A concrete running example



```
    //-----------------------------------------------EXAMPLES-----------------------------------------------//

    // EXAMPLES
    @Test
    void converter_test_puglia_medisante() throws FileNotFoundException {
        // STEP 1
        String body = IntegrationEngineTests.getInputFile("medisante_input.json");
        ConverterImpl converter = new ConverterImpl();
        // STEP 2
        Bundle conversionResult = converter.convertFromHttpBody(body);
        // STEP 3
        FhirContext ctx = FhirContext.forR4();
        IParser parserJson = ctx.newJsonParser();
        IParser parserXml = ctx.newXmlParser();
        System.out.println(parserJson.encodeResourceToString(conversionResult));

    }
```

- an utility method to load the input file you have uploaded in the local folder (src/test/resources/test_files/).



```java
//--------------------------------------------------UTILITY METHODS--------------------------------------------------//

private static String getInputFile(String fileName) throws FileNotFoundException {
    File text = new File("src/test/resources/test_files/" + fileName);

    // Creating Scanner instnace to read File in Java
    Scanner scnr = new Scanner(text);

    // Reading each line of file using Scanner class
    String outputString = "";
    while (scnr.hasNextLine()) {
        String line = scnr.nextLine();
        outputString = outputString + line;
    }
    return outputString;
}
```