SEVENTH FRAMEWORK PROGRAMME

Specific Targeted Research Project

Project Number:	**FP7–SMARTCITIES–2013(ICT)**
Project Acronym:	**VITAL**
Project Number:	**608682**
Project Title:	**Virtualized programmable InTerfAces for innovative cost-effective IoT depLoyments in smart cities**

D3.2.2 Specification and Implementation of Virtualized Unified Access Interfaces V2

Document Id:	VITAL-D322-250615-Draft
File Name:	VITAL-D322-250615-Draft.doc
Document reference:	Deliverable 3.2.2
Version :	Draft
Editor :	John Soldatos, Katerina Roukounaki
Organisation :	AIT
Date :	25 / 06 / 2015
Document type:	Deliverable
Security:	PU (Public)

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V20	John Soldatos	AIT	08/04/2015	Table of Contents, Structured and Enhancement to previous versions
V201	John Soldatos	AIT	02/05/2015	Updates to Section 1
V202	Katerina Roukounaki	AIT	05/05/2015	Section 3 updated
V203	Fabrizio De Ceglia, Paola Dal Zovo	Reply	08/05/2015	Added contributions in Section 6 and Section 7.5
V204	Katerina Roukounaki	AIT	10/05/2015	Removed Section 5, Updates to Section 3, Contributions to Section 6
V205	Riccardo Petrolo, Salvatore Guzzo Bonifacio, Valeria Loscri, Nathalie Mitton	Inria	13/05/2015	Discovery, Filtering, PPI implementation for FIT IoT-Lab
V206	Angelos Lenis	SiLo	14/05/2014	Chapter 4.5: Workflow Management / Orchestrator
V207	Paola Dal Zovo	Reply	18/05/2015	Minor changes and clarifications in Section 5 and 6 based on comments from AIT.
V208	Umut YILDIRIM	ATOS	21/05/2015	Added contributions to Section 4
V209	Martin Serrano	NUIG	23/05/2015	Initial Contributions on DMS architecture
V21	Katerina Roukounaki	AIT	25/05/2015	Minor changes in Section 3
	Katerina Roukounaki	AIT	31/05/2015	Minor changes in Section 3
V211	Riccardo Petrolo	Inria	02/06/2015	Changes in Section 4.2
V212	Aqeel Kazmi	NUIG	08/06/2015	Changes in Section 4.1
V213	Katerina Roukounaki	AIT	09/06/2015	Changes in Section 3, comments in Section 4.2
V214	Aqeel Kazmi	NUIG	10/06/2015	Changes in Section 4.2
V215	Katerina Roukounaki	AIT	11/06/2015	Comments in Section 4.2
	Katerina Roukounaki	AIT	15/06/2015	Comments in Section 4.3
V216	Salvatore Guzzo Bonifacio	INRIA	17/06/2015	Minor fixes in sections 4.2, 4.3 and 6.4

V217	Valeria Loscri	INRIA	19/06/2015	Some minor updates in the sections 4.2, 4.3 and 6.4
V22	Aqeel Kazmi	NUIG	23/06/2015	Some minor updates in section 4.1
V221	Katerina Roukounaki	AIT	24/06/2015	Review section 4.1
V222	Umut YILDIRIM	ATOS	16/07/2015	Changes in Section 4.4
V223	Katerina Roukounaki	AIT	19/07/2015	Review changes in Section 4.4
V224	Aqeel Kazmi	NUIG	31/07/2015	Changes in Section 4.1
V225	Katerina Roukounaki	AIT	03/08/2015	Section 2 Updates to the VITAL architecture
	Katerina Roukounaki	AIT	03/08/2015	Section 3 Updates to the PPI specifications in order to ensure integration with security mechanism and compliance to updated versions of the VITAL ontology
V26	Lorenzo Bracco, Paola Dal Zovo	Reply	05/08/2015	Section 5.1.1 Updates about access control for PUSH interfaces and without Policy Agent
V27	Salvatore Guzzo Bonifacio, Riccardo Petrolo	Inria	06/08/2015	Updated Discovery interface and Technical Review
V28	Miguel Angel Mateo	Atos	21/08/2015	Updated CEP interface and Technical Amendments
V29	Aqeel Kazmi	NUIG	21/08/2015	Updates to DMS section and Quality Control
	Aqeel Kazmi	NUIG	21/08/2015	Circulated for Approval
Draft	Martin Serrano	NUIG	25/08/2015	EC Submitted

COVERVIEW OF UPDATES/ENHANCEMENTS OVER D3.2.1

Section	**Description**
Section 1	Introductory Section Updates to the description of the summary and scope of the second version of the deliverable
Section 2	Updates to the VITAL architecture
Section 3	Updates to the PPI specifications in order to ensure integration with security mechanism and compliance to updated versions of the VITAL ontology
Section 4	Contributions to Complex Event Processing (CEP)
Section 5	Description of the security architecture
Section 6	Description of the PPI for Hi Reply

TABLE OF CONTENTS

DOCUMENT HISTORY 1

COVERVIEW OF UPDATES/ENHANCEMENTS OVER D3.2.1 3

1 INTRODUCTION 8

1.1 Scope 8
1.2 Audience 9
1.3 Summary 9
1.4 Structure 10

2 VIRTUALIZED UNIFIED ACCESS INTERFACES 11

2.1 Purpose and Positioning within the VITAL Architecture 11
2.2 Types of VUAIs and Functional Scope 12

3 PLATFORM PROVIDER INTERFACE SPECIFICATION 13

3.1 Overview 13
3.2 PPI Specification 14
3.2.1 Access to IoT system metadata 14
3.2.2 Access to IoT service metadata 16
3.2.3 Access to sensor metadata 18
3.2.4 Access to sensor observations 21
3.2.4.1 Pull-based mechanism 21
3.2.4.2 Push-based mechanism 23
3.2.5 Configuration 24
3.2.6 Monitoring 26
3.2.6.1 Performance Metric Monitoring 27
3.2.6.2 SLA parameter monitoring 29
3.3 Platform Access and Data Acquisition 31

4 VIRTUALIZED ACCESS TO VITAL MODULES 36

4.1 Interfaces to Data Management Service 36
4.1.1 Overview 36
4.1.2 Specifications 37
4.1.2.1 Store Metadata and Observations 37
4.1.2.2 Access to stored metadata 38
4.1.2.3 Access to stored observations 40
4.2 Interfaces to Service Discovery Module 42
4.3 Interface to Filtering Module 43
4.4 Interfaces to CEP Module 45
4.4.1 Overview 45
4.4.2 Specifications 45
4.5 Interfaces to Workflow Management 49

5 SECURITY OF VIRTUALIZED UNIFIED ACCESS INTERFACES 52

5.1 Overview of VUAI Security Architecture and Implementation 53
5.1.1 Security Modules and Libraries 53
5.1.2 Authentication 56
5.1.3 Authorization 56
5.2 Example: PPI Authentication and Authorization 60
5.3 Example: VUAI Authentication and Authorization 61

6 PPI PROTOTYPE IMPLEMENTATIONS 62

6.1 PPI Implementation for Open Data Sources 62
6.1.1 PPI Implementation for Footfall Data Feed 62
6.1.2 PPI Implementation for Bus Arrival Data Feed 63
6.2 PPI Implementation for X-GSN 64
6.3 PPI Implementation for Xively 65
6.4 PPI Implementation for FIT/IoT-LAB 66
6.4.1 Implementation Overview 66
6.4.2 Examples 67
6.5 PPI Implementation for Hi Reply 68
6.5.1 Implementation Overview 68
6.5.1.1 Hi Reply Client 69
6.5.1.2 Core & Rest API 69
6.5.2 Examples 70

7 CONCLUSIONS AND OUTLOOK TOWARDS FINAL VERSION 71

8 REFERENCES 72

LIST OF FIGURES

FIGURE 1: OVERVIEW OF VITAL ARCHITECTURE. 11
FIGURE 2: TWO MAIN OPTIONS OFFERED BY THE VITAL VIRTUALIZATION LAYER (VUAIS) REGARDING IOT DATA ACCESS. 12
FIGURE 3: OVERVIEW OF THE VUAIS PROVIDED BY THE VITAL PLATFORM. 13
FIGURE 4: PADA POSITION IN VITAL ARCHITECTURE. 32
FIGURE 5: IOT SYSTEM PROVIDER REQUESTS FROM PADA TO REFRESH ALL RELATED METADATA. 34
FIGURE 6: PADA PULLS PERIODICALLY OBSERVATIONS MADE BY A SENSOR. 35
FIGURE 7: IOT PUSHES NEW OBSERVATIONS MADE BY A SENSOR TO PADA. 35
FIGURE 8: DETAILED DESCRIPTION ABOUT INTERFACES USING FLOW DIAGRAMS, CLASS DIAGRAMS. 36
FIGURE 9: DMS POSITION IN VITAL ARCHITECTURE. 37
FIGURE 10: PADA PUSHES METADATA AND DATA TO DMS. 38
FIGURE 11: VITAL MODULES (I.E. ADDED-VALUE FUNCTIONALITIES) ACCESS METADATA. 39
FIGURE 12: VITAL MODULES (I.E. ADDED-VALUE FUNCTIONALITIES) ACCESS OBSERVATIONS. 41
FIGURE 13: EXAMPLE INTERACTION FILTERING AND VUAIS 44
FIGURE 13: BASIC FLOW OF AN SAML 2.0 SSO THROUGH A WEB BROWSER USING HTTP. 52
FIGURE 14: OPENAM IN THE OAUTH 2.0 FLOW. 54
FIGURE 15: POLICY AGENT & OPENAM INTEGRATION. 57
FIGURE 16: POLICY AGENT & OPENAM AUTHENTICATION/AUTHORIZATION FLOW. 58
FIGURE 17: VUAI SECURITY ARCHITECTURE OVERVIEW. 58
FIGURE 18: PPI SECURITY ARCHITECTURE OVERVIEW. 59
FIGURE 19: FIT IOT-LAB REPRESENTATION 67
FIGURE 20: PPI IMPLEMENTATION FOR THE HI REPLY PLATFORM. 68
FIGURE 21: EXAMPLE FLOW 71

LIST OF TABLES

TABLE 1: ACCESS TO GENERAL INFORMATION ABOUT AN IOT SYSTEM. 14
TABLE 2: ACCESS TO CURRENT IOT SYSTEM STATUS. 16
TABLE 3: ACCESS TO METADATA ABOUT THE PROVIDED IOT SERVICES. 16
TABLE 4: ACCESS TO METADATA ABOUT THE MANAGED SENSORS. 18
TABLE 5: ACCESS TO CURRENT SENSOR STATUS. 20
TABLE 6: METADATA SAMPLE FOR AN OBSERVATIONSERVICE THAT SUPPORTS THE PULL-BASED MECHANISM. 21
TABLE 7: ACCESS TO SENSOR OBSERVATIONS. 21
TABLE 8: METADATA SAMPLE FOR AN OBSERVATIONSERVICE THAT SUPPORTS THE PUSH-BASED MECHANISM. 23
TABLE 9: SUBSCRIPTION TO AN OBSERVATION STREAM. 23
TABLE 10: CANCELLATION OF A SUBSCRIPTION TO AN OBSERVATION STREAM. 24
TABLE 11: SAMPLE CONFIGURATIONSERVICE METADATA. 24
TABLE 12: ACCESS TO CURRENT IOT SYSTEM CONFIGURATION. 25
TABLE 13: CHANGES TO CURRENT IOT SYSTEM CONFIGURATION. 25
TABLE 14: MONITORINGSERVICE METADATA SAMPLE. 26
TABLE 15: ACCESS TO SUPPORTED PERFORMANCE METRICS. 27
TABLE 16: ACCESS TO CURRENT VALUES OF PERFORMANCE METRICS. 28
TABLE 17: ACCESS TO SUPPORTED SLA PARAMETERS. 29
TABLE 18: ACCESS TO CURRENT VALUES OF SLA PARAMETERS. 30
TABLE 19: ACCESS TO REGISTERED IOT SYSTEMS. 32
TABLE 20: PUSHING METADATA AND DATA INTO DMS. 38
TABLE 21: ACCESSING METADATA FROM DMS. 39
TABLE 22: ACCESSING OBSERVATIONS FROM DMS. 41
TABLE 23: GET IOTSYSTEM. 42
TABLE 24: GET ICOS 43
TABLE 25: FILTER ON TEMPORAL BASIS 44
TABLE 26: REQUEST FOR AUTHENTICATION TOKEN. 60
TABLE 27: REQUEST FOR A PROTECTED RESOURCE. 60
TABLE 28: TECHNOLOGIES USED IN FOOTFALL DATA FEED PPI IMPLEMENTATION. 63
TABLE 29: TECHNOLOGIES USED IN LIVE BUS ARRIVAL DATA FEED. 64
TABLE 30: TECHNOLOGIES USED IN X-GSN PPI IMPLEMENTATION. 65
TABLE 31: TECHNOLOGIES USED IN XIVELY PPI IMPLEMENTATION. 66
TABLE 32: TECHNOLOGIES USED IN HI REPLY PPI IMPLEMENTATION. 68

TERMS AND ACRONYMS

API	Application Programming Interface
ARIMA	AutoRegressive Integrated Moving Average
CEP	Complex Event Processing
FCAPS	Fault, Configuration, Accounting, Performance, Security
FIT	Future Internet of Things
GSN	Global Sensor Networks
ICO	Internet Connected Object
IoT	Internet-of-Things
JAXB	Java Architecture for XML Binding
JSON-LD	JavaScript Object Notation for Linked Data
LDAP	Lightweight Directory Access Protocol
LSM	Linked Sensor Middleware
OIDC	OpenID Connect
PAP	Policy Administration Point
PDP	Policy Decision Point
PEP	Policy Enforcement Point
PPI	Platform Provider Interface
REST	Representational State Transfer
SAML	Security Assertion Markup Language
SSN	Semantic Sensor Network
SSO	Single Sign-On
TfL	Transport for London
VUAI	Virtualized Unified Access Interface
X-GSN	eXtended Global Sensor Networks

1 INTRODUCTION

1.1 Scope

The third work package of the VITAL project (WP3) deals with the specification and implementation of models and interfaces that could enable virtualization of diverse IoT systems. This virtualization is a key to implement IoT platform agnostic functionalities as part of the VITAL platform, thereby enabling development of integrated IoT applications for smart cities, i.e. applications leveraging data and services from multiple underlying IoT systems. Key elements of the VITAL virtualization infrastructure include:

- A range of platform-agnostic data models (data schemas, ontologies and databases), along with systems for managing their data elements. At the heart of these data models lies the VITAL ontology, which provides a model for representing data for IoT applications in smart cities regardless of the IoT systems used to capture and/or process these data. The VITAL ontology is already specified as part of deliverable D3.1.1 of the project.
- A range of virtualized interfaces, which enable platform agnostic access to the IoT services of diverse heterogeneous IoT systems. These interfaces boost VITAL's integrated virtualized application development paradigm, which promotes a «learn-once-and-use-across-IoT-systems» approach to IoT application development in smart cities.

The purpose of the present deliverable is to provide the specification of the virtualized interfaces of the VITAL platform. In particular, the following types of virtualized interfaces are specified:

- Interfaces to the value-added functionalities of the VITAL platform, such as Complex Event Processing (CEP) and Service Discovery functionalities. These functionalities typically use the VITAL ontology as a means for expressing their results in a way that is independent of the initial data sources (e.g. of the underlying platforms that provide the discovered services).
- Abstract, virtualized interfaces to the functionalities of the underlying IoT systems (platforms and applications), which are classified as PPIs (Platform Provider Interfaces). The notion and the role of PPIs have already been introduced as part of the VITAL architecture specification (described in deliverable D2.3).

All the above interfaces can be classified as VUAIs (Virtualized Unified Access Interfaces). The present version (D3.2.2) of the deliverable represents an enhanced version of the first (D3.2.1). This second version presents the updated VITAL architecture, which reflects updates to the structure and operation of the VUAIs. It documents an initial prototype implementation of PPIs for the four IoT platforms that have been selected (as part of WP2) towards practically showcasing and validating the VITAL platform-agnostic concept. Furthermore, several updates to the semantics of the various VUAIs are presented, including VUAIs that will be implemented in order to access data from the VITAL data management services.

Note that the present version of the deliverable is not the final one. A third release planned for the third year of the project (i.e. D3.2.3) will reflect the final specification and implementation of the VUAIs (including PPIs). The specifications and prototype

implementations that will be included in D3.2.3 will be in-line with the evolution of the VITAL architecture, but also in-line with the final version of the VITAL ontology (to be reflected in the coming deliverable D3.1.2).

1.2 Audience

This deliverable is addressed to the following audiences:

- **IoT applications developers and solution providers**, notably solution providers emphasizing on smart city applications using the IoT paradigm. Application developers and solution providers are expected to be interested into the project's general-purpose interfaces for accessing IoT systems, especially given the fact that the VITAL VUAI specifications attempt to virtually address any IoT system. Furthermore, VUAIs provide a first approach to implement the very topical target of IoT/BigData convergence, since they include a range of abstract data processing functions over platform-agnostic IoT services.
- **IoT researchers,** notably researchers working on abstract data models and service models for IoT applications. To these researchers, VUAIs may serve as a source of information for their research.
- **VITAL developers,** notably individuals engaging in the development of the VITAL added-value functionalities and of the VITAL applications. The former will need to understand the VUAIs in order to make sure that their components/modules support them, while the latter need to gain insights on the VUAIs in order to use them properly as part of their smart city application development tasks.

As already outlined, the present deliverable will be consulted by VITAL developers working in WP4, WP5 and WP6 activities, which will be using PPIs and VUAIs for accessing IoT data and services in a platform-agnostic manner.

1.3 Summary

As already outlined, the VITAL virtualized interfaces can be classified in two different types, namely: (a) abstract interfaces to IoT systems and (b) abstract Interfaces to the added-value functionalities of the VITAL platform. This deliverable includes dedicated parts to the specification of each of the above types of interfaces. Each dedicated part describes the interfaces and their use in the scope of the VITAL platform, i.e. when and how they can be used by «client» applications accessing the VITAL platform. The specification of each interface (regardless of its type) includes a description of the functionalities it provides, as well as a description of the input it expects and the output it produces.

In addition to providing the specifications of the virtualized interfaces, the deliverable elaborates on their positioning and use in the scope of the VITAL architecture. It therefore identifies the consumers of the interface functionalities. For instance, PPIs provide a low-latency interface for accessing data streams and capabilities of the underlying IoT systems. Hence, PPIs are very handy for VITAL application developers, notably developers that are engaging in the integration of real-time or semi real-time applications. As another example, complex event processing interfaces are handy for solution developers focusing on data-intensive applications within the smart city.

Note that several aspects, such as security, are handled within the specification of each interface, rather than horizontally i.e. in a way that transcends all specified interfaces. This is intentional and due to the different scopes and time-scales of operations of the various interface types.

This second release of the deliverable describes the implementation of VUAIs over the VITAL data management services, as well as the prototype implementation of the PPIs for the four platforms that have been selected for practical adaptation to the VITAL architecture and semantically interoperability paradigm.

1.4 Structure

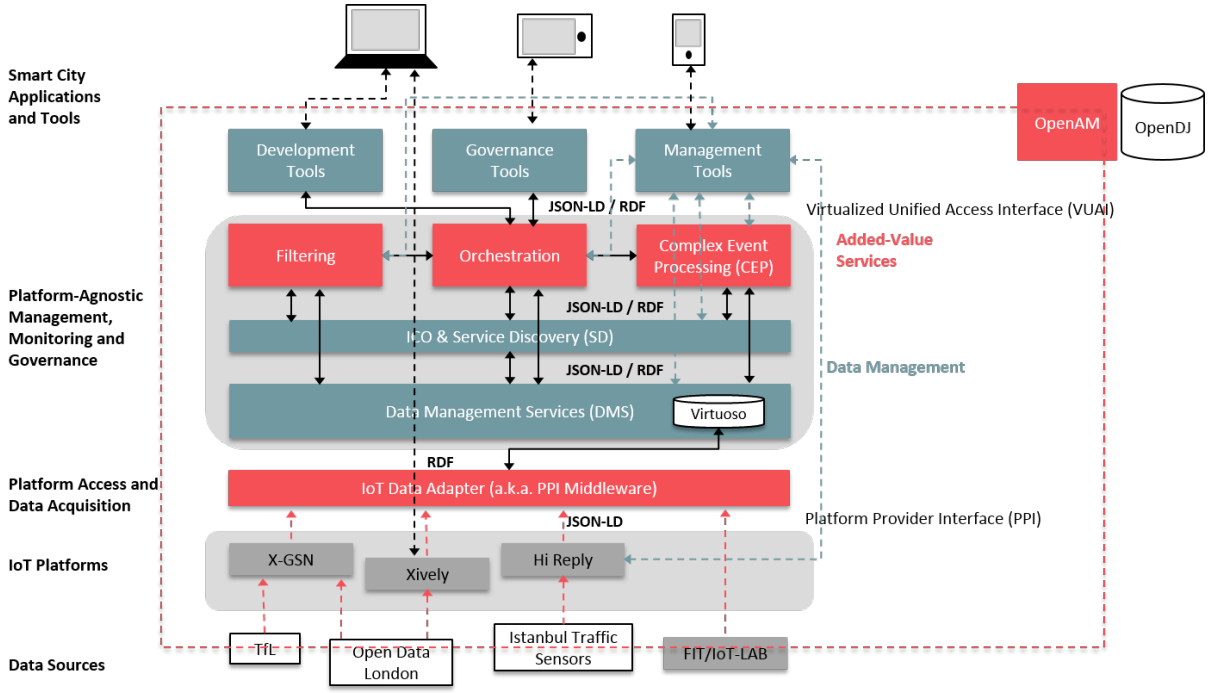
The deliverable is structured as follows:

- Section 2, following this introductory section, illustrates the scope and purpose of the various types of VUAIs. It also discusses their positioning within the VITAL architecture.
- Section 3 is devoted to the specification of PPIs. The specification includes several updates comparing to the earlier version of the deliverable. The updates have been produced as a result of the updates to the VITAL ontology and of additional requirements introduced by several components of the VITAL platform.
- Section 4 focuses on the specifications of VUAIs for accessing the added-value functionalities of the VITAL platform.
- Section 5 illustrates the architecture and implementation of the security mechanisms used to authorize and authenticate interactions through the VUAIs.
- Section 6 describes the early PPI implementations for the FIT/IoT-LAB, Xively, X-GSN, and Hi Reply platforms, but also for various data sources.
- Section 7 is the final and concluding section of this deliverable, which provides also an outlook for the development of the next (and final) release of the latter.

2 VIRTUALIZED UNIFIED ACCESS INTERFACES

2.1 Purpose and Positioning within the VITAL Architecture

The purpose of VUAIs is to facilitate virtualized platform-agnostic access to sensor data and IoT services for smart cities. They are abstract interfaces enabling developers to access data streams and services provided by multiple IoT systems, without the need to deal with the low-level details of the underlying systems. The concept of VUAIs has already been introduced in deliverable D2.3 as part of the presentation of the VITAL architecture.



Required parameters are missing or incorrect.

Figure 1: Overview of VITAL architecture.

Figure 1 illustrates a slightly enhanced version of the (high-level) VITAL architecture provided in deliverable D2.3. It depicts the role of VUAIs as abstract interfaces residing at the top layer of the VITAL architecture, thereby enabling access to added-value data processing and process management functionalities, including CEP, service discovery, filtering, and other functionalities. As already outlined, it also illustrates additional concepts regarding high-performance access to platforms and data sources using PPIs. These concepts are implied, but not detailed in deliverable D2.3. In particular:

- **PPI as high-performance VUAI**: The architecture specifies the ability of accessing IoT data both through VUAIs accessing the VITAL data management services and through direct access to PPIs. The latter option is required in cases where high-performance, low-overhead access to data provided by IoT systems is required (e.g. (near) real-time applications). In this context, the PPI, i.e. the platform agnostic interface for accessing IoT systems, can be considered as a high-performance VUAI for data access. Overall, the two virtualized platform-

agnostic options for accessing IoT systems and IoT data through VUAIs are explained in Figure 2.

- **PPI as an interface for accessing individual data sources**: VITAL enables the federation of IoT platforms (such as the four platforms selected in deliverable D2.2, namely FIT/IoT-LAB, X-GSN, Xively and Hi Reply). Nevertheless, it also acknowledges the possibility of exploiting individual (IoT-related) data sources and datasets (e.g. live feeds stemming from TfL open dataset), based on direct access to their data. Access to individual data sources can be also carried out through a PPI implementation, and more specifically based on a light implementation that does not implement optional functionalities. This concept is also illustrated in Figure 1.

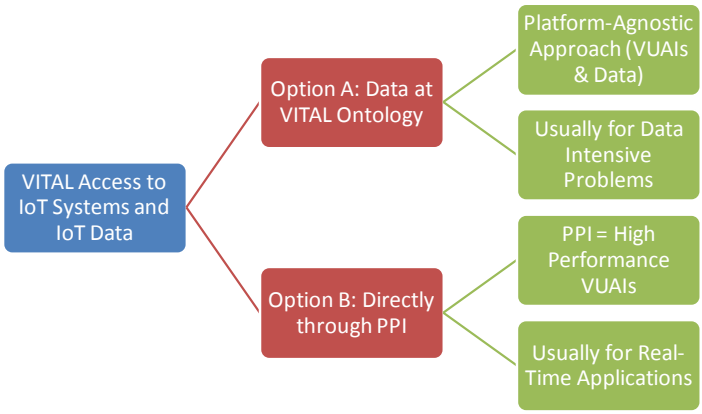


Figure 2: Two main options offered by the VITAL Virtualization Layer (VUAIs) regarding IoT data access.

In addition to virtualized access to IoT data and services, as well as to added-value functionalities (like discovery, filtering and CEP), the high-level VITAL architecture (depicted in the previous figure) prescribes that VUAIs will be accessed by the VITAL development environments and tools. The latter tools are expected to facilitate the development of IoT applications that are based on data processing functionalities (notably data intensive problems). It is therefore envisaged that VUAIs should include interfaces for executing data processing / data mining methods (such as classification and clustering).

2.2 Types of VUAIs and Functional Scope

The previous paragraph has introduced the concept of VUAIs and their positioning in the VITAL architecture. It has also illustrated the fact that PPIs can be considered VUAIs. As a result, VUAIs can be classified in two main categories:

- **VUAIs for direct data access leveraging PPIs**: These are interfaces for platform access, which enable a «learn-once-and-use-across-IoT-systems» discipline for accessing IoT data. They facilitate developers and solution providers to access any VITAL compliant platform via a unified interface. PPIs can also be used to access individual data sources. As illustrated in the following section, the PPI specification includes a range of both mandatory and optional methods, which correspond to the different spectrum of functionalities that are required to access individual data sources or integrated fully fledged IoT platforms.

- **VUAIs accessing VITAL modules**: These are interfaces for accessing the modules residing at the added-value layer of the VITAL architecture, notably the filtering, CEP and service discovery modules, as well as the data from the VITAL data management services (following the VITAL ontology initially specified in deliverable D3.1.1).

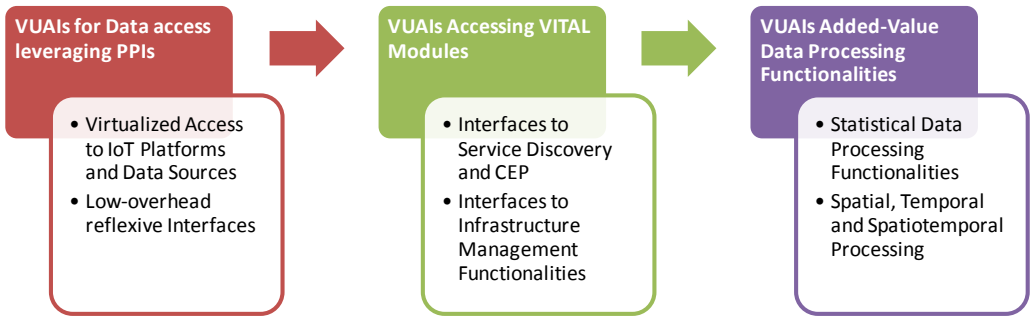


Figure 3: Overview of the VUAIs provided by the VITAL platform.

The two types of VUAIs outlined above are specified in the following paragraphs. The specifications are driven by other VITAL results produced as part of earlier deliverables in particular:

- The VITAL ontology and related data models, which drive the specification of the data elements that are exchanged via the VUAIs. Indeed, we do not specify new data models in this deliverable; rather we rely on the data models prescribed in the first version of the VITAL data modeling deliverable (D3.1.1). Note that subsequent versions of this document will be based on the final release of the VITAL data models (D3.1.2), which is currently in progress.
- The VITAL architecture introduced in deliverable D2.3. The same deliverable has also illustrated requirements associated with the data elements that should be exchanged over VUAIs.

Overall, the present deliverable is in-line with background results contained in deliverables D3.1.1 and D2.3. The readership is advised to consult these documents for more details on data models and architectural concepts that underpin the VUAI specifications in the present document.

It should also be noted that the interfaces (VUAIs) specified in this document will provide valuable inputs to other technical activities of the project, notably activities that will produce technical modules that will consume/use the VUAIs. Several such modules (e.g. CEP, discovery, filtering, data access for FCAPS management) are currently being developed in WP4 and WP5 of the project.

3 PLATFORM PROVIDER INTERFACE SPECIFICATION

3.1 Overview

The Platform Provider Interface (PPI) is a set of RESTful web services, marked as either mandatory or optional, that enable access to IoT systems, as well as to ICOs managed and IoT services provided by those systems. All VITAL compliant IoT

systems are expected to implement and expose at least the web services that are designated as mandatory.

3.2 PPI Specification

PPI is defined as a set of RESTful web services that IoT systems to be integrated into VITAL should expose, and that the VITAL platform can then use in order to retrieve:

- Information about the IoT systems (e.g. their status)
- Information about the IoT services that an IoT system exposes (e.g. how to access them)
- Information about the sensors that an IoT system manages (e.g. what they observe)
- Observations made by the sensors that an IoT system manages.

A detailed description of these web services is given in the following paragraphs and is in-line with information contained in earlier deliverable D2.3. Specifically, in D2.3 we have already provided an overview of data and services that the PPI is expected to provide/expose, and we have also classified these data and services as mandatory or optional. Overall, the detailed PPI specifications contained in the following paragraphs build on earlier specifications and requirements specified as part of WP2 of the project.

3.2.1 Access to IoT system metadata

This section describes the web services that any VITAL compliant IoT system must expose, so that the VITAL platform can use them to obtain metadata about the system. IoT system metadata include:

- general information about the IoT system (e.g. its operator)
- information about any configuration functionalities that the IoT system might provide (e.g. services that allow to retrieve the current configuration of the system and probably change it)
- the current status of the IoT system

The purpose of the web service described in Table 1 is to provide access to general information about the IoT system (e.g. its service area), as well as to the list of the services it provides and the list of the sensors it manages. Access to that information is described as mandatory in D2.3, and thus all VITAL compliant IoT systems must implement and expose this web service.

Table 1: Access to general information about an IoT system.

	Get IoT system metadata	
Description	This service can be used to access metadata about the IoT system.	
URL	PPI_BASE_URL/metadata	
Method	POST	
Request headers	Content-Type	application/json
Request body	**Example** {	

<table>
<tr><td></td><td colspan="2">}</td></tr>
<tr><td>Response headers</td><td>Content-Type</td><td>application/ld+json or application/json</td></tr>
<tr><td>Response body</td><td colspan="2">Example

<pre>{
  "@context": "http://vital-iot.eu/contexts/system.jsonld",
  "id": "http://example.com",
  "type": "vital:VitalSystem",
  "name": "Sample IoT system",
  "description": "This is a VITAL compliant IoT system.",
  "operator": "http://example.com/people#john_doe",
  "serviceArea": "http://dbpedia.org/page/Camden_Town",
  "sensors":
  [
    "http://example.com/sensor/1",
    "http://example.com/sensor/2"
  ],
  "services":
  [
    "http://example.com/service/1",
    "http://example.com/service/2",
    "http://example.com/service/3"
  ],
  "status": "vital:Running"
}</pre></td></tr>
<tr><td>Mandatory</td><td colspan="2">Yes</td></tr>
<tr><td>Notes</td><td colspan="2">• At the moment, the request body is an empty JSON object with no fields, but it can be later used for filtering the returned metadata.
• The context of the response body is the JSON-LD context for systems described in Section 5.1 of D3.1.1.</td></tr>
</table>

PPI_BASE_URL denotes the URL where the PPI implementation of the underlying IoT system can be accessed. Thus, if the PPI implementation for a particular IoT system can be accessed through http://example.com/ppi, then its metadata can be retrieved by issuing a POST request to http://example.com/ppi/metadata.

Note that the above web service returns only the IDs of the services provided and of the sensors managed by the IoT system. Sections 3.2.2 and 3.2.3 describe how to retrieve more information about them.

If an IoT system intends to expose any configuration functionality to the VITAL platform, it must provide a service of type **ConfigurationService**, as described in Section 0. Note that, since the configuration service is in fact one of the IoT services that the corresponding IoT system provides, its ID must be listed in the result of the web service described in Table 1, and its metadata must be available in the same way that the metadata of all IoT services that an IoT system provides are exposed, as described in Section 3.2.2.

Finally, since the status of an IoT system might change throughout its lifecycle, we decided to represent its values as observations of a virtual sensor. Thus, VITAL compliant systems that want to expose their current operational state must manage a virtual sensor of type **MonitoringSensor**. More details about that sensor can be found in Section 3.2.6. As an alternative way of exposing its current status, an IoT system may provide an IoT service of type **MonitoringService** with (at least) an operation of type **GetSystemStatus**. The result of that operation is the current status of the system, and is equivalent to retrieving the last observation made by the sensor of type MonitoringSensor for the property **OperationalState** of the feature of interest

http://example.com (namely the IoT system). Refer to Table 14 for metadata about a monitoring service, and to Table 2 for details about the implementation requirements for the GetSystemStatus operation.

Table 2: Access to current IoT system status.

	Get IoT system status	
Description	This service can be used to access the current status of the IoT system.	
Method	POST	
Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	
	`{`
	`  "@context": "http://vital-iot.eu/contexts/measurement.jsonld",`
	`  "id": "http://example.com/sensor/1/observation/1",`
	`  "type": "ssn:Observation",`
	`  "ssn:observationProperty":`
	`  {`
	`  "type": "vital:OperationalState"`
	`  },`
	`  "ssn:observationResultTime":`
	`  {`
	`  "time:inXSDDateTime": "2014-08-20T16:47:32+01:00"`
	`  },`
	`  "ssn:featureOfInterest": "http://example.com",`
	`  "ssn:observationResult":`
	`  {`
	`  "type": "ssn:SensorOutput",`
	`  "ssn:hasValue":`
	`  {`
	`  "type": "ssn:ObservationValue",`
	`  "value": "vital:Running"`
	`  }`
	`  }`
	`}`	

Mandatory	No	
Notes	• The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.1.	

Based on D2.3, the provision of lifecycle information, part of which are the current status and any configuration functionality, is optional for IoT systems to be integrated into VITAL. Thus, the provision of the necessary operations as part of a configuration and a monitoring service are both optional.

3.2.2 Access to IoT service metadata

IoT systems that want to connect to the VITAL platform can expose to the latter metadata about any IoT services they provide. In order to do that, they can implement and expose the web service described in Table 3, which returns metadata about the provided IoT services that have a specific ID or are of a specific type. Based on D2.3, the provision of the following web service is optional for the IoT systems that want to be PPI compliant.

Table 3: Access to metadata about the provided IoT services.

	Get IoT service metadata
Description	This service can be used to access metadata about IoT services that the IoT system provides.

URL	PPI_BASE_URL/service/metadata	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	{
	}	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	
	[
	{
	"@context": "http://vital-iot.eu/contexts/service.jsonld",
	"id": "http://example.com/service/1",
	"type": "vital:**ConfigurationService**",
	"operations":
	[
	{
	"type": "vital:**GetConfiguration**",
	"hrest:hasAddress": "http://example.com/service/1",
	"hrest:hasMethod": "hrest:GET"
	},
	{
	"type": "**vital:SetConfiguration**",
	"hrest:hasAddress": "http://example.com/service/1",
	"hrest:hasMethod": "hrest:POST"
	}
	]
	},
	{
	"@context": "http://vital-iot.eu/contexts/service.jsonld",
	"id": "http://example.com/service/2",
	"type": "vital:**MonitoringService**",
	"msm:hasOperation":
	[
	{
	"type": "vital:**GetSystemStatus**",
	"hrest:hasAddress": "http://example.com/system/status",
	"hrest:hasMethod": "hrest:POST"
	},
	{
	"type": "vital:**GetSensorStatus**",
	"hrest:hasAddress": "http://example.com/sensor/status",
	"hrest:hasMethod": "hrest:POST"
	},
	{
	"type": "vital:**GetSupportedPerformanceMetrics**",
	"hrest:hasAddress": "http://example.com/system/performance",
	"hrest:hasMethod": "hrest:GET"
	},
	{
	"type": "vital:**GetPerformanceMetrics**",
	"hrest:hasAddress": "http://example.com/system/performance",
	"hrest:hasMethod": "hrest:POST"
	},
	{
	"type": "vital:**GetSupportedSLAParameters**",
	"hrest:hasAddress": "http://example.com/system/sla",
	"hrest:hasMethod": "hrest:GET"
	},
	{
	"type": "vital:**GetSLAParameters**",
	"hrest:hasAddress": "http://example.com/system/sla",
	"hrest:hasMethod": "hrest:POST"
	}	

	<pre>]
	  },
	  {
	    "@context": "http://vital-iot.eu/contexts/service.jsonld",
	    "id": "http://example.com/service/3",
	    "type": "vital:ObservationService",
	    "operations":
	    [
	      {
	        "type": "vital:GetObservations",
	        "hrest:hasAddress": "http://example.com/sensor/observation",
	        "hrest:hasMethod": "hrest:POST"
	      }
	    ]
	  }
	]</pre>

Mandatory	No
Notes	• The request body is a JSON object with the following fields: **id**, whose value is an array of IoT service IDs (e.g. http://example.com/service/1), **type**, whose value is an array of IoT service type IDs (e.g. http://vital-iot.eu/ontology/ns/MonitoringService). Both fields are optional and can be used to filter the IoT services to retrieve metadata for. • The context of the response body is the JSON-LD context for services described in Section 5.2.2 of D3.1.1.

3.2.3 Access to sensor metadata

IoT systems must expose to VITAL information about the sensors they manage (e.g. their type or their location). In order to do that, VITAL compliant IoT systems are expected to implement the RESTful web service that is presented in Table 4. The purpose of this service is to return metadata about the managed sensors that satisfy certain criteria. The current version of the PPI specification supports filtering of the managed sensors based on their ID and type.

Table 4: Access to metadata about the managed sensors.

	Get sensor metadata	
Description	This service can be used to access metadata about sensors that the IoT system manages.	
URL	PPI_BASE_URL/sensor/metadata	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	<pre>{
	}</pre>	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	
	<pre>[
	  {
	    "@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	    "id": "http://example.com/sensor/1",
	    "type": "vital:MonitoringSensor",
	    "name": "System Monitoring Sensor",
	    "description": "A virtual sensor that monitors the operational state of the system, as well as its services and sensors.",
	    "status": "vital:Running",
	    "ssn:observes":</pre>	

	<pre>[
	  {
	  "type": "vital:OperationalState",
	  "id": "http://example.com/sensor/1/operationalState"
	  },
	  {
	  "type": "vital:SysUptime",
	  "id": "http://example.com/sensor/1/sysUptime"
	  },
	  {
	  "type": "vital:SysLoad",
	  "id": "http://example.com/sensor/1/sysLoad"
	  },
	  {
	  "type": "vital:Errors",
	  "id": "http://example.com/sensor/1/errors"
	  }
	  ]
	},
	{
	  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	  "id": "http://example.com/sensor/2",
	  "name": "A sensor.",
	  "type": "VitalSensor",
	  "description": "A sensor.",
	  "hasLastKnownLocation":
	  {
	  "type": "geo:Point",
	  "geo:lat": 53.2719,
	  "geo:long": -9.0849
	  },
	  "ssn:observes":
	  [
	  {
	  "type": "openiot:Temperature",
	  "id": "http://example.com/sensor/2/temperature"
	  }
	  ]
	}]</pre>

Mandatory	Yes
Notes	• The request body is a JSON object with the following fields: **id**, whose value is an array of sensor URIs (e.g. http://example.com/sensor/1), **type**, whose value is an array of sensor type URIs (e.g. http://vital-iot.eu/ontology/ns/VitalSensor). Both fields are optional and can be used to filter the sensors to retrieve metadata for. • The context of the response body is the JSON-LD context for sensors described in Section 3.3 of D3.1.1.

Since access to sensor metadata is marked as mandatory in deliverable D2.3, all VITAL compliant IoT systems are expected to implement and expose a RESTful web service that satisfies all requirements outlined in Table 4.

Apart from sensor metadata that are **static** (e.g. their name or the properties they observe), there are also **dynamic** sensor metadata that include:

- the current status of the sensor
- the current location of the sensor, in case the sensor is mobile

IoT systems are expected to expose the current status of a sensor they manage as the last observation of the **MonitoringSensor** (described in Section 3.2.6) for the property **OperationalState** of the managed sensor (i.e. the sensor is the feature of interest). As an alternative way of exposing the current status of the sensors it

manages, an IoT system may provide an IoT service of type **MonitoringService** with (at least) an operation of type **GetSensorStatus**. Refer to Table 14 for metadata about a monitoring service, and to Table 5 for details about the implementation requirements for the GetSensorStatus operation.

Table 5: Access to current sensor status.

<table>
<tr><th></th><th colspan="2">Get sensor status</th></tr>
<tr><td>Description</td><td colspan="2">This service can be used to access the current status of the sensors managed by the IoT system.</td></tr>
<tr><td>Method</td><td colspan="2">POST</td></tr>
<tr><td>Request headers</td><td>Content-Type</td><td>application/json</td></tr>
<tr><td>Request body</td><td colspan="2">Example

<pre>{
  "id":
  [
    "http://example.com/sensor/2"
  ]
}</pre></td></tr>
<tr><td>Response headers</td><td>Content-Type</td><td>application/ld+json or application/json</td></tr>
<tr><td>Response body</td><td colspan="2">Example

<pre>[
  {
    "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
    "id": "http://example.com/sensor/1/observation/2",
    "type": "ssn:Observation",
    "ssn:observationProperty":
    {
      "type": "vital:OperationalState"
    },
    "ssn:observationResultTime":
    {
      "time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
    },
    "ssn:featureOfInterest": "http://example.com/sensor/2",
    "ssn:observationResult":
    {
      "type": "ssn:SensorOutput",
      "ssn:hasValue":
      {
        "type": "ssn:ObservationValue",
        "value": "vital:Running"
      }
    }
  }
]</pre></td></tr>
<tr><td>Mandatory</td><td colspan="2">No</td></tr>
<tr><td>Notes</td><td colspan="2">• The request body is a JSON object with the following fields: id, whose value is an array of sensor IDs (e.g. http://example.com/sensor/1), type, whose value is an array of sensor type IDs (e.g. http://vital-iot.eu/ontology/ns/VitalSensor). Both fields are optional and can be used to filter the sensors to retrieve the current status of.
• The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.1.</td></tr>
</table>

Finally, an IoT system can expose the current status of a sensor in the value of the status field in the metadata retrieved for that sensor through the service described in Table 4.

As far as the current location of a mobile sensor is concerned, there are two ways to obtain it (as described in Section 6.5.1 of D3.1.1):

- through the IoT service that the **hasLocalizer** property contained in the sensor metadata points
- through the **hasLastKnownLocation** property contained in the sensor metadata

The first way is always preferred to the second, since the value of the hasLastKnownLocation property might be outdated.

3.2.4 Access to sensor observations

It is prescribed that the VITAL platform can use both a pull- and a push-based mechanism in order to obtain observations made by a sensor. An IoT system can support one or both of these mechanisms by providing an IoT service of type **ObservationService** with the necessary operations.

Since access to the sensor data of an IoT system is mandatory, all VITAL compliant IoT systems must support at least one of these two mechanisms. This essentially means that IoT providers should implement and expose one RESTful web service for each operation that is required by the mechanism (or the mechanisms) they want to support, as described in the following paragraphs.

3.2.4.1 Pull-based mechanism

In order to support the pull-based mechanism (i.e. in order for VITAL to be able to pull observations from the IoT system), the **ObservationService** must have (at least) an operation of type **GetObservations**. An example for the description of an IoT service of type ObservationService that supports the pull-based mechanism is shown in **Table 6**.

Table 6: Metadata sample for an ObservationService that supports the pull-based mechanism.

```
{
  "@context": "http://vital-iot.eu/contexts/service.jsonld",
  "id": "http://example.com/service/3",
  "type": "vital:ObservationService",
  "operations":
  [
    {
      "type": "vital:GetObservations",
      "hrest:hasAddress": "http://example.com/sensor/observation",
      "hrest:hasMethod": "hrest:POST"
    }
  ]
}
```

As shown in Table 7, the GetObservations operation is expected to return observations that satisfy certain criteria in JSON-LD format based on the JSON-LD context for measurements described in deliverable D3.1.1 (see Section 3.4).

Table 7: Access to sensor observations.

	Get observations

<table>
<tr><td>Description</td><td colspan="2">This service can be used to access observations made by sensors.</td></tr>
<tr><td>Method</td><td colspan="2">POST</td></tr>
<tr><td>Request headers</td><td>Content-Type</td><td>application/ld+json or application/json</td></tr>
<tr><td>Request body</td><td colspan="2">Example

<pre>{
  "sensor":
  [
    "http://example.com/sensor/2"
  ],
  "property": "http://lsm.deri.ie/OpenIot/Temperature"
  "from": "2014-11-17T09:00:00+02:00",
  "to": "2014-11-17T11:00:00+02:00"
}</pre></td></tr>
<tr><td>Response headers</td><td>Content-Type</td><td>application/ld+json or application/json</td></tr>
<tr><td>Response body</td><td colspan="2">Example

<pre>[
  {
    "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
    "id": "http://example.com/sensor/2/observation/1",
    "type": "ssn:Observation",
    "ssn:observedBy": "http://example.com/sensor/2",
    "ssn:observationProperty":
    {
      "type": "http://lsm.deri.ie/OpenIoT/Temperature"
    },
    "ssn:observationResultTime":
    {
      "time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
    },
    "dul:hasLocation":
    {
      "type": "geo:Point",
      "geo:lat": "55.701",
      "geo:long": "12.552",
      "geo:alt": "4.33"
    },
    "ssn:observationResult":
    {
      "type": "ssn:SensorOutput",
      "ssn:hasValue":
      {
        "type": "ssn:ObservationValue",
        "value": "21.0",
        "qudt:unit": "qudt:DegreeCelsius"
      }
    }
  }
]</pre></td></tr>
<tr><td>Mandatory</td><td colspan="2">No</td></tr>
<tr><td>Notes</td><td colspan="2">• The request body is a JSON object with the following fields: sensor, whose value is an array of sensor IDs (e.g. http://example.com/sensor/1), property, whose value is a property type (e.g. http://lsm.deri.ie/OpenIoT/Temperature), and from and to that define a time interval. sensor and property fields are mandatory and determine the sensor and the property, respectively, to return observations for. from and to determine the time interval, when the observations to return were taken. Both to and from are optional. If to is omitted, then all observations taken after from are returned. If both from and to are omitted, then the last observation taken from the specified sensor for the specified property is returned.
• The context of the response is the context for measurements described in</td></tr>
</table>

	Section 3.4 of D3.1.1.

3.2.4.2 Push-based mechanism

In order to support the push-based mechanism (i.e. in order for the IoT system to be able to push observations to the VITAL platform), the **ObservationService** should have (at least) the following operations:

- an operation of type **SubscribeToObservationStream** that creates a subscription to a specific stream of observations
- an operation of type **UnsubscribeFromObservationStream** that cancels the subscription with a specific ID.

An example for the description of an IoT service of type ObservationService that supports the push-based mechanism is shown in Table 8.

Table 8: Metadata sample for an ObservationService that supports the push-based mechanism.

```
{
  "@context": "http://vital-iot.eu/contexts/service.jsonld",
  "id": "http://example.com/service/3",
  "type": "vital:ObservationService",
  "operations":
  [
    {
      "type": "SubscribeToObservationStream",
      "hrest:hasAddress": "http://example.com/observation/stream/subscribe",
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "type": "UnsubscribeFromObservationStream",
      "hrest:hasAddress": "http://example.com/observation/stream/unsubscribe",
      "hrest:hasMethod": "hrest:POST"
    }
  ]
}
```

As described in Table 9, the SubscribeToObservationStream operation is expected to return a subscription ID. As a result of that operation, VITAL periodically receives at the URL, which was specified in the body of the SubscribeToObservationStream request, observations that match the specified criteria (i.e. observations that were obtained by the specified sensor for the specified property). The URL essentially indicates another RESTful web service - one that is implemented and exposed by the VITAL platform - that supports the HTTP POST method, and expects a set of observations in JSON-LD format based on the JSON-LD context for measurements described in Section 3.4 of D3.1.1.

Table 9: Subscription to an observation stream.

	Subscribe to observation stream	
Description	This service can be used to subscribe to an observation stream.	
Method	POST	
Request headers	Content-Type	application/json
Request body	**Example** {	

	<pre>{
	  "sensor": "http://example.com/sensor/1",
	  "property": "http://lsm.deri.ie/OpenIot/Temperature",
	  "url": "http://vital-iot.eu/observation/push"
	}</pre>

Response headers	Content-Type application/json

Response body	**Example**
	<pre>{
	  "subscription" : "d670460b4b4aece5915caf5c68d12f560a9fe3e4"
	}</pre>

Mandatory	No
Notes	• The request body is a JSON object with the following fields: **sensor**, whose value is a sensor ID (e.g. http://example.com/sensor/1), **property**, whose value is a property type (e.g. http://lsm.deri.ie/OpenIoT/Temperature), and **url**, whose value is a URL. All fields are mandatory. **sensor** and **property** fields determine the sensor and the property, respectively, to receive observations for. **url** determines the URL where the IoT system is expected to push to the subscriber any new observations made by the sensor for the property.

As described in Table 10, the UnsubscribeFromObservationStream operation gives back no response, and results in the cancellation of the subscription with the specified ID.

Table 10: Cancellation of a subscription to an observation stream.

	Unsubscribe from observation stream
Description	This service can be used to cancel a currently active subscription to an observation stream.
Method	POST
Request headers	Content-Type application/json

Request body	**Example**
	<pre>{
	  "subscription": "d670460b4b4aece5915caf5c68d12f560a9fe3e4"
	}</pre>

Mandatory	No
Notes	• The request body is a JSON object with a single field, **subscription**. The subscription-id field is mandatory, and its value is the ID of the subscription to cancel.

3.2.5 Configuration

IoT systems connected to the VITAL platform may provide configuration functionalities to the latter. For that purpose, an IoT system can provide an IoT service of type **ConfigurationService**. If VITAL is allowed to retrieve the current configuration of the IoT system, then the service must have an operation of type **GetConfiguration**. If VITAL is also allowed to modify the current configuration of the system, then the service must also contain a second operation of type **SetConfiguration**. Table 11 presents an example for the description of such a service.

Table 11: Sample ConfigurationService metadata.

```
{
  "@context": "http://vital-iot.eu/contexts/service.jsonld",
```

```
  "id": "http://example.com/service/1",
  "type": "vital:ConfigurationService",
  "msm:hasOperation":
  [
    {
      "type": "vital:GetConfiguration",
      "hrest:hasAddress": "http://example.com/service/1",
      "hrest:hasMethod": "hrest:GET"
    },
    {
      "type": "vital:SetConfiguration",
      "hrest:hasAddress": "http://example.com/service/1",
      "hrest:hasMethod": "hrest:POST"
    }
  ]
}
```

As described in Table 12, the GetConfiguration operation is expected to return the current configuration of the IoT system. For each configuration parameter, this operation is expected to return its name, its current value, its type, and the permissions that the VITAL platform has on it.

Table 12: Access to current IoT system configuration.

	Get configuration	
Description	This service can be used to access the current configuration of the IoT system.	
Method	GET	
Response headers	Content-Type	application/json

Response body	**Example**
	
	`{`
	`  "parameters":`
	`  [`
	`  {`
	`  "name": "c1",`
	`  "value": "1",`
	`  "type": "http://www.w3.org/2001/XMLSchema#int",`
	`  "permissions": "rw"`
	`  },`
	`  {`
	`  "name": "c2",`
	`  "value": "v2",`
	`  "type": " http://www.w3.org/2001/XMLSchema#string",`
	`  "permissions": "rw"`
	`  }`
	`  ]`
	`}`	

Mandatory	No	
Notes	• The response contains the key-value pairs for all configuration parameters exposed to VITAL, along with the value types and the associated permissions (rw, r).	

Table 13 contains the functional requirements for the SetConfiguration operation, which receives one or more new values for one or more (writable) configuration parameters, and is expected to update the IoT system configuration accordingly.

Table 13: Changes to current IoT system configuration.

	Change configuration
Description	This service can be used to change the current value of one or more configuration

	parameters of the IoT system.	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	`{`
	`  "parameters":`
	`  [`
	`  {`
	`  "name": "c1",`
	`  "value": "2"`
	`  },`
	`  {`
	`  "name": "c2",`
	`  "value": "v3"`
	`  }`
	`  ]`
	`}`	

Mandatory	No	
Notes	• The status code of the request can be: ◦ **200**, if the IoT system configuration has been successfully updated ◦ **403**, if at least one the specified parameters is read-only ◦ **404**, if at least one of the specified parameters does not exist • Note that if among the specified parameters there is at least one that does not exist or is not writable, then none of the required configuration parameter value changes will be performed.	

Note that the configuration service is one of the IoT services that the corresponding IoT system provides, which implies that its metadata must be available in the same way that the metadata of all IoT services that an IoT system provides are exposed, as described in Section 3.2.2.

3.2.6 Monitoring

The VITAL platform may be able to monitor various aspects of a registered IoT system, depending on whether the latter has exposed any monitoring functionalities to VITAL. More specifically, VITAL can monitor:

- the status of the IoT system
- the status of the sensors that the IoT system manages
- various performance metrics related to the IoT system (e.g. its uptime)
- various SLA parameters related to the IoT system (e.g. the response time for successful requests)

An IoT system can allow VITAL to access and monitor any or all of the above listed information by providing an IoT service of type **MonitoringService**. Table 14 contains an example for metadata about a monitoring service. We have already described how an IoT system can expose its current status and the current status of the sensors it manages in Sections 3.2.1 and 3.2.3, respectively. The rest of the section focuses on the performance- and SLA-related actions of MonitoringService.

Table 14: MonitoringService metadata sample.

```
{
  "@context": "http://vital-iot.eu/contexts/service.jsonld",
  "id": "http://example.com/service/2",
```



```
  "type": "vital:MonitoringService",
  "msm:hasOperation":
  [
    {
      "type": "vital:GetSystemStatus",
      "hrest:hasAddress": "http://example.com/system/status",
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "type": "vital:GetSensorStatus",
      "hrest:hasAddress": "http://example.com/sensor/status",
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "type": "vital:GetSupportedPerformanceMetrics",
      "hrest:hasAddress": "http://example.com/system/performance",
      "hrest:hasMethod": "hrest:GET"
    },
    {
      "type": "vital:GetPerformanceMetrics",
      "hrest:hasAddress": "http://example.com/system/performance",
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "type": "vital:GetSupportedSLAParameters",
      "hrest:hasAddress": "http://example.com/system/sla",
      "hrest:hasMethod": "hrest:GET"
    },
    {
      "type": "vital:GetSLAParameters",
      "hrest:hasAddress": "http://example.com/system/sla",
      "hrest:hasMethod": "hrest:POST"
    }
  ]
}
```

3.2.6.1 Performance Metric Monitoring

In order for VITAL to be able to monitor various performance metrics about an IoT system, the latter must provide a MonitoringService with (at least) an operation of type **GetSupportedPerformanceMetrics** that returns a list of the performance metrics that the IoT system supports. Table 15 contains the specifications for that operation. A list of the performance metrics that IoT systems may support is included in Section 6.7.1 of D3.1.1 and in Section 3.2.1 of D51.1.

Table 15: Access to supported performance metrics.

	Get supported performance metrics	
Description	This service can be used to get a list of the performance metrics that the IoT system supports.	
Method	GET	
Response headers	Content-Type	application/json

Response body	**Example**
	
	`{`
	`  "metrics":`
	`  [`
	`  {`
	`  "type": "http://vital-iot.eu/ontology/ns/SysUptime",`
	`  "id": "http://example.com/sensor/1/sysUptime"`
	`  },`	

	<pre>{
	"type": "http://vital-iot.eu/ontology/ns/SysLoad",
	"id": "http://example.com/sensor/1/sysLoad"
	},
	{
	"type": "http://vital-iot.eu/ontology/ns/Errors",
	"id": "http://example.com/sensor/1/errors"
	}
	]
	}</pre>

Mandatory	No
Notes	• The response body contains all supported performance metrics. For each metric, the type and the ID of the corresponding observed property are provided.

As it is obvious from Table 15, each performance metric corresponds essentially to a property that the **MonitoringSensor** observes. That way an IoT system can expose the current value of a performance metric (e.g. for the system uptime) as the last observation of the MonitoringSensor for the corresponding property (e.g. SysUptime) of the feature of interest that represents the IoT system (e.g. http://example.com).

Alternatively, VITAL compliant IoT systems can expose the current (or last) value of the supported performance metrics by adding an operation of type **GetPerformanceMetrics** to the MonitoringService. Refer to Table 16 for the functional requirements for that operation.

Table 16: Access to current values of performance metrics.

	Get performance metrics	
Description	This service can be used to access the current value of the performance metrics that the IoT system supports.	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	<pre>{
	"metric":
	[
	"http://vital-iot.eu/ontology/ns/SysLoad",
	"http://vital-iot.eu/ontology/ns/SysUptime"
	]
	}</pre>	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	<pre>[
	{
	"@context": "http://vital-iot.eu/contexts/measurement.jsonld",
	"id": "http://example.com/sensor/1/observation/3",
	"type": "ssn:Observation",
	"ssn:observationProperty":
	{
	"type": "vital:SysLoad"
	},
	"ssn:observationResultTime":
	{
	"time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
	},
	"ssn:featureOfInterest": "http://example.com",
	"ssn:observationResult":</pre>	

	<pre>{
	"type": "ssn:SensorOutput",
	"ssn:hasValue":
	{
	"type": "ssn:ObservationValue",
	"value": "80",
	"qudt:unit": "qudt:Percent"
	}
	}
	},
	{
	"@context": "http://vital-iot.eu/contexts/measurement.jsonld",
	"id": "http://example.com/sensor/1/observation/4",
	"type": "ssn:Observation",
	"ssn:observationProperty":
	{
	"type": "vital:SysUptime"
	},
	"ssn:observationResultTime":
	{
	"time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
	},
	"ssn:featureOfInterest": "http://example.com",
	"ssn:observationResult":
	{
	"type": "ssn:SensorOutput",
	"ssn:hasValue":
	{
	"type": "ssn:ObservationValue",
	"value": "800000",
	"qudt:unit": "qudt:MilliSecond"
	}
	}
	}
	]</pre>

Mandatory	No
Notes	• The request body is a JSON object with a single field, **metric**. The value of this field is an array of property type IDs (e.g. http://vital-iot.eu/ontology/ns/SysLoad). • The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.1.

Since an IoT system is not obliged to provide any configuration-related functionality in order to be VITAL compliant, the provision of a monitoring service, as well as the implementation of any of the operations described in this sections are classified as optional.

3.2.6.2 SLA parameter monitoring

For QoS purposes, an IoT system might allow VITAL to monitor certain SLA parameters. An initial version of a list with the SLA parameters that IoT systems may support is included in Section 3.2.4.1 of D5.1.1. In order for an IoT system to enable VITAL to monitor any SLA parameters about it, the former must provide a MonitoringService with (at least) an operation of type **GetSupportedSLAParameters**, which returns a list of the SLA parameters that the IoT system supports, and which is described in Table 17.

Table 17: Access to supported SLA parameters.

	Get supported SLA parameters
Description	This service can be used to get a list of the SLA parameters that the IoT system

	supports.	
Method	GET	
Response headers	Content-Type	application/json

Response body	**Example**
	
	`{`
	`  "parameters":`
	`  [`
	`  {`
	`  "type": "http://vital-iot.eu/ontology/ns/ResponseTime",`
	`  "id": "http://example.com/sensor/1/responseTime"`
	`  },`
	`  {`
	`  "type": "http://www.w3.org/2011/http#StatusCode",`
	`  "id": "http://example.com/sensor/1/statusCode"`
	`  }`
	`  ]`
	`}`	

Mandatory	No	
Notes	• The response body contains all supported SLA parameters. For each parameter, the type and the ID of the corresponding observed property are provided.	

Each SLA parameter, like each performance metric, corresponds to a property that the **MonitoringSensor** observes. Based on that an IoT system has two alternative ways of exposing the current value of SLA parameters:

- as the last observation of the MonitoringSensor for the corresponding property (e.g. ResponseTime) of the appropriate feature of interest (e.g. HTTP requests made to a specific URL by a specific user)
- by adding an operation of type **GetSLAParameters** to the MonitoringService

More information about the GetSLAParameters operation can be found in Table 18.

Table 18: Access to current values of SLA parameters.

	Get SLA parameters	
Description	This service can be used to access the current value of the SLA parameters that the IoT system supports.	
Method	POST	
Request headers	Content-Type	application/json

Request body	**Example**
	
	`{`
	`  "parameter":`
	`  [`
	`  "http://vital-iot.eu/ontology/ns/ResponseTime"`
	`  ]`
	`}`	

Response headers	Content-Type	application/ld+json or application/json

Response body	**Example**
	
	`[`
	`  {`
	`  "@context": "http://vital-iot.eu/contexts/measurement.jsonld",`
	`  "id": "http://example.com/sensor/1/observation/5",`
	`  "type": "ssn:Observation",`
	`  "ssn:observationProperty":`
	`  {`	

	<pre>"type": "vital:ResponseTime"
	},
	"ssn:observationResultTime":
	{
	"time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
	},
	"ssn:featureOfInterest":
	{
	"http:methodName": "GET",
	"http:absoluteURI": "http://example.com/system/status",
	"madeBy": "http://vital-example.com/users#243",
	},
	"ssn:observationResult":
	{
	"type": "ssn:SensorOutput",
	"ssn:hasValue":
	{
	"type": "ssn:ObservationValue",
	"value": "80",
	"qudt:unit": "qudt:MilliSecond"
	}
	}
	}
	]</pre>

Mandatory	No
Notes	• The request body is a JSON object with a single field, **parameter**. The value of this field is an array of property type IDs (e.g. http://vital-iot.eu/ontology/ns/ResponseTime). • The context of the response body is the JSON-LD context for measurements described in Section 3.4 of D3.1.1.

The enablement of SLA parameter monitoring is not mandatory for the characterization of an IoT system as VITAL compliant. Thus, the provision of the operations described above as part of a service of type MonitoringService is optional.

3.3 Platform Access and Data Acquisition

Platform Access and Data Acquisition (**PADA**) is the component of the VITAL platform that acts as the intermediary between VITAL and PPI implementations of external IoT systems that want to be integrated into VITAL. Figure 4 shows the position of PADA within the VITAL architecture.

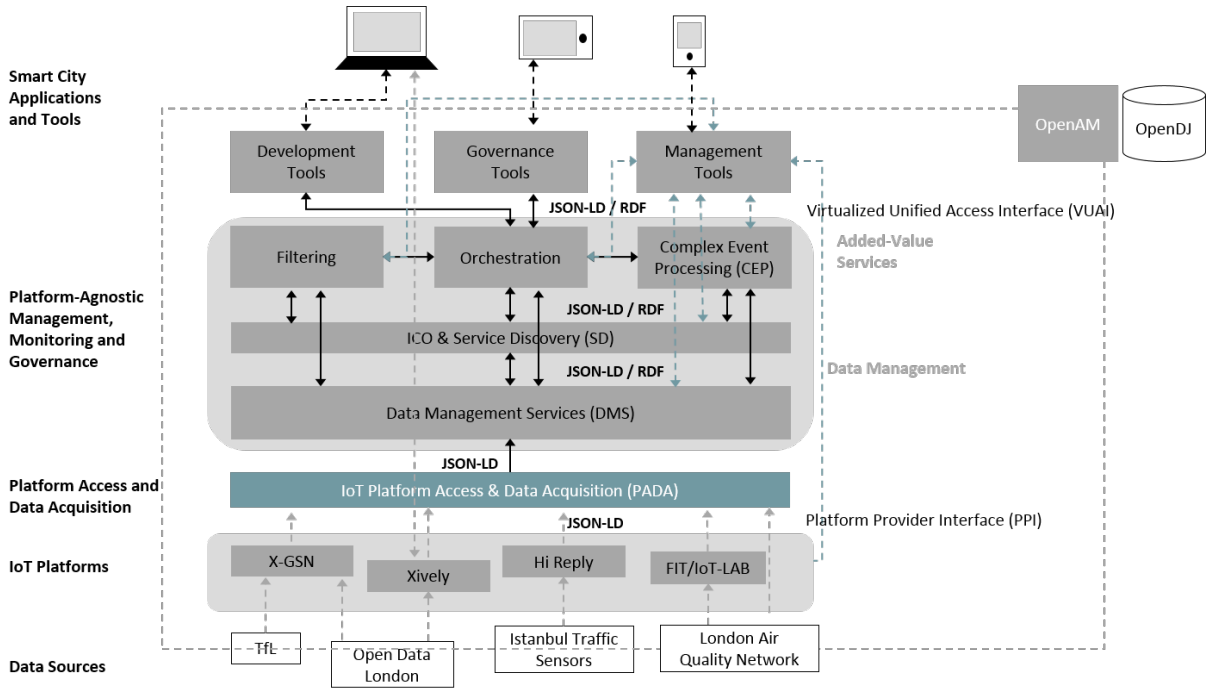


Figure 4: PADA position in VITAL architecture.

The purposes of PADA are:

- to provide a web-accessible user interface that can be used by IoT system providers in order to:
  - register and de-register their systems
  - manage the settings used by VITAL to access their systems (e.g. the URL where the PPI implementation of their system is exposed)
- to collect information from the registered IoT systems, and store this information into a VITAL-specific internal store

As part of the registration and deregistration processes, the IoT system provider is expected to provide:

- the **URI** of the IoT system, which acts as its unique identifier (this is the value of the id field in the IoT system metadata, see Table 1).
- the **base URL**, through which the PPI implementation exposed by the IoT system is accessible.

PADA provides a web service that can be used to retrieve information about the IoT systems that are currently registered in the VITAL platform. Details about this web service are given in Table 19.

Table 19: Access to registered IoT systems.

	Get registered IoT systems	
Description	This service can be used to obtain information about the IoT systems currently registered in VITAL.	
URL	PADA_BASE_URL/system/registered	
Method	GET	
Response headers	Content-Type	application/json
Response body	**Example**	

	<pre>[
	  {
	  "id": "http://example.com/",
	  "ppi": "http://example.com/ppi"
	  }
	]</pre>

Notes	• The response body contains information about all registered systems. For each system, its ID and its base URL are provided.

PADA_BASE_URL denotes the URL that can be used to access the PADA component of an instance of the VITAL platform. The above web service can be used to construct the value of the **providesSystem** property of a **VitalSystem**, as described in Section 6.1.1 of D3.1.2.

PADA is responsible for collecting information about and from the registered IoT systems. As soon as a new IoT system is registered to the VITAL platform, PADA retrieves metadata about the system itself (Section 3.2.1), the IoT services it provides (Section 3.2.2), and the sensors it manages (Section 3.2.3). The retrieved metadata are pushed to DMS through a VUAI the latter exposes for that purpose (Section **Error! Reference source not found.**).

Once an IoT system has been registered to the VITAL platform, its provider can explicitly request VITAL (through the PADA user interface) to refresh the metadata stored about it. PADA is then responsible to retrieve a fresh version of the IoT system, IoT service and sensor metadata, and push them to DMS, using the same VUAI described above. This is how PADA manages information that is exposed by the registered IoT systems and that is considered to be rarely modified (e.g. the name of the system). Figure 5 depicts this sequence of actions.

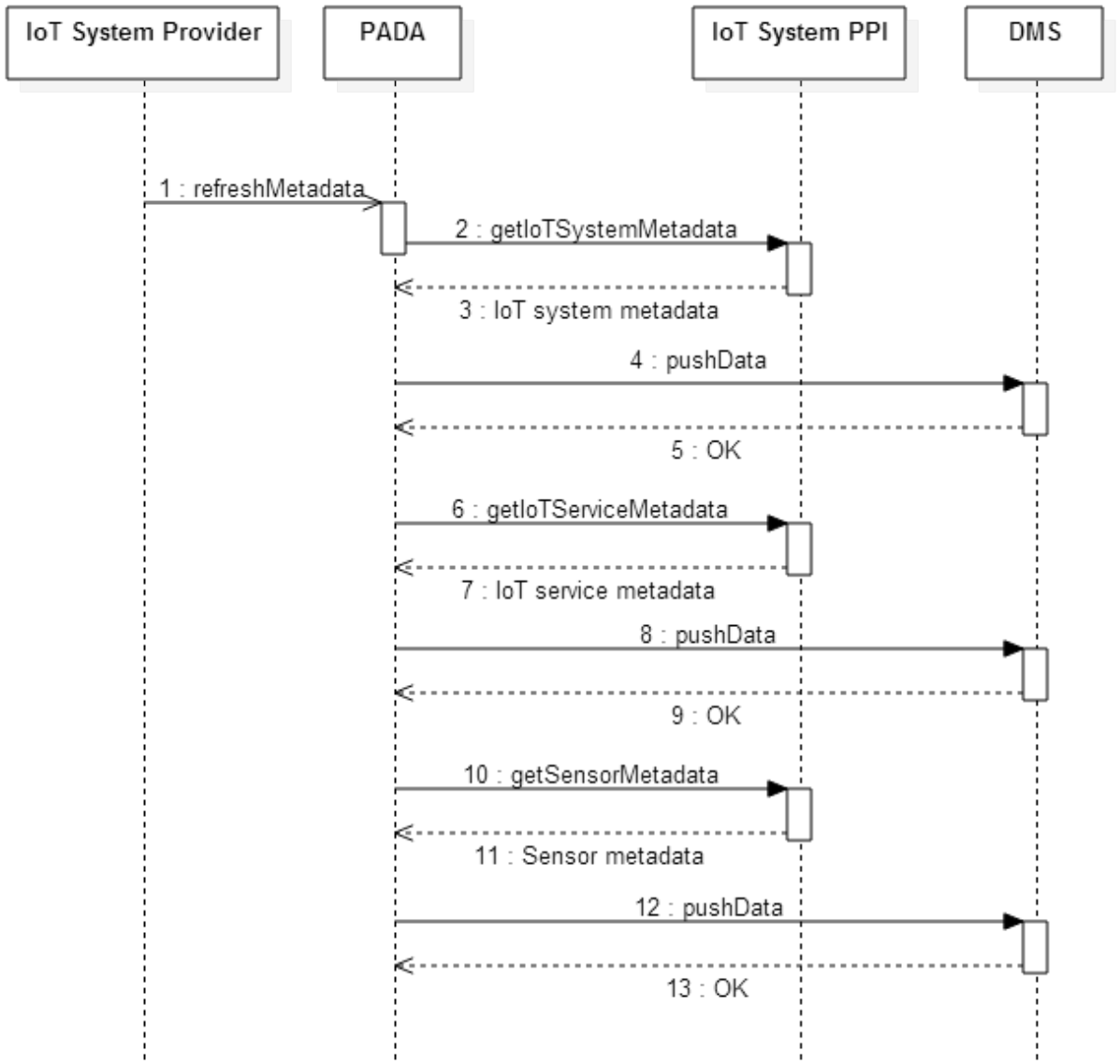


Figure 5: IoT system provider requests from PADA to refresh all related metadata.

Apart from static information, IoT systems provide also access to dynamic information (i.e. information that changes or is generated during the lifetime of the system). At the moment, this dynamic information includes:

- the current status of the IoT system
- the current status of all sensors managed by the IoT system
- the current locations of all mobile sensors managed by the IoT system
- new observations made by sensors managed by the IoT system

In order to guarantee that VITAL has an almost up-to-date version of this dynamic information, PADA pulls periodically this information (using the respective PPI primitives described in Sections 3.2.1, 3.2.3 and 3.2.4), and pushes them to DMS. The frequency with which each one of the above pieces of information is pulled from an IoT system is configurable (through the PADA web interface) and might vary from system to system, depending also on business requirements of the targeted smart

cities applications. Figure 6 illustrates how PADA periodically pulls from the PPI of an IoT system new observations that have been made in the meantime by a sensor.

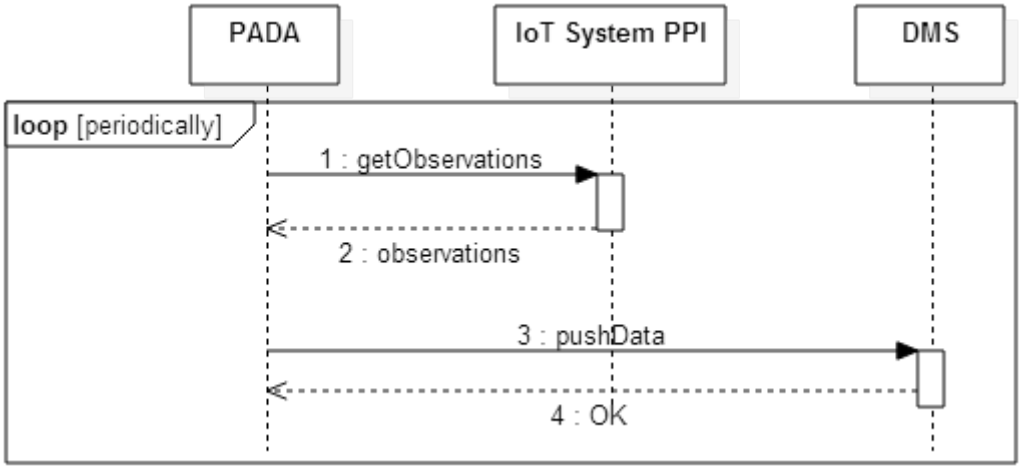


Figure 6: PADA pulls periodically observations made by a sensor.

Finally, PADA always checks for the provision of a push-based mechanism that it can probably use to collect any of the above pieces of information, and prefers it over a pull-based mechanism if both are supported by the corresponding PPI implementation. Figure 7 illustrates how PADA uses the push-based mechanism to get observations from the PPI of an IoT system.

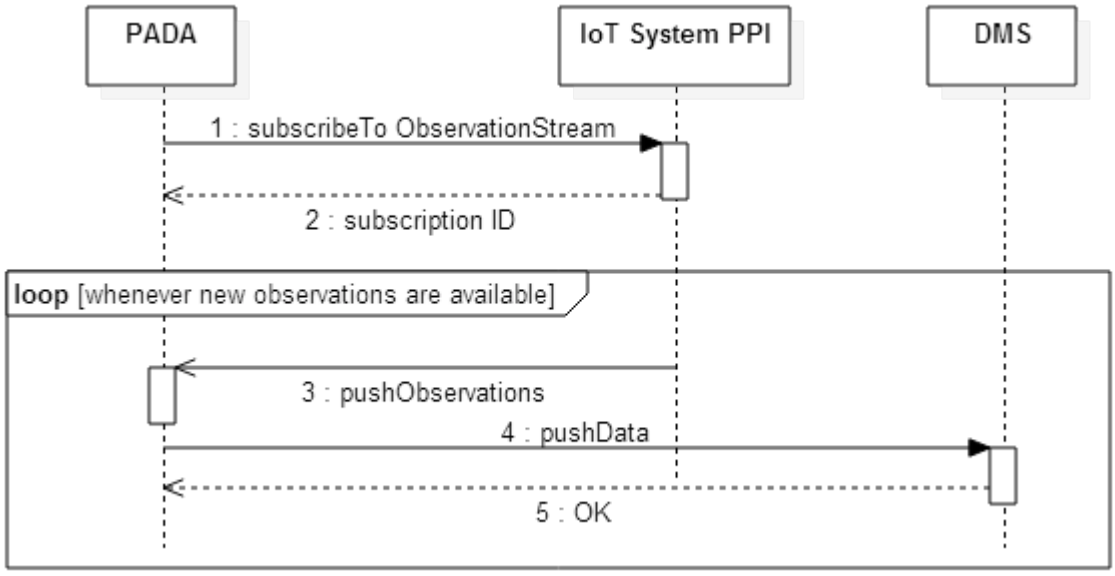


Figure 7: IoT pushes new observations made by a sensor to PADA.

4 VIRTUALIZED ACCESS TO VITAL MODULES

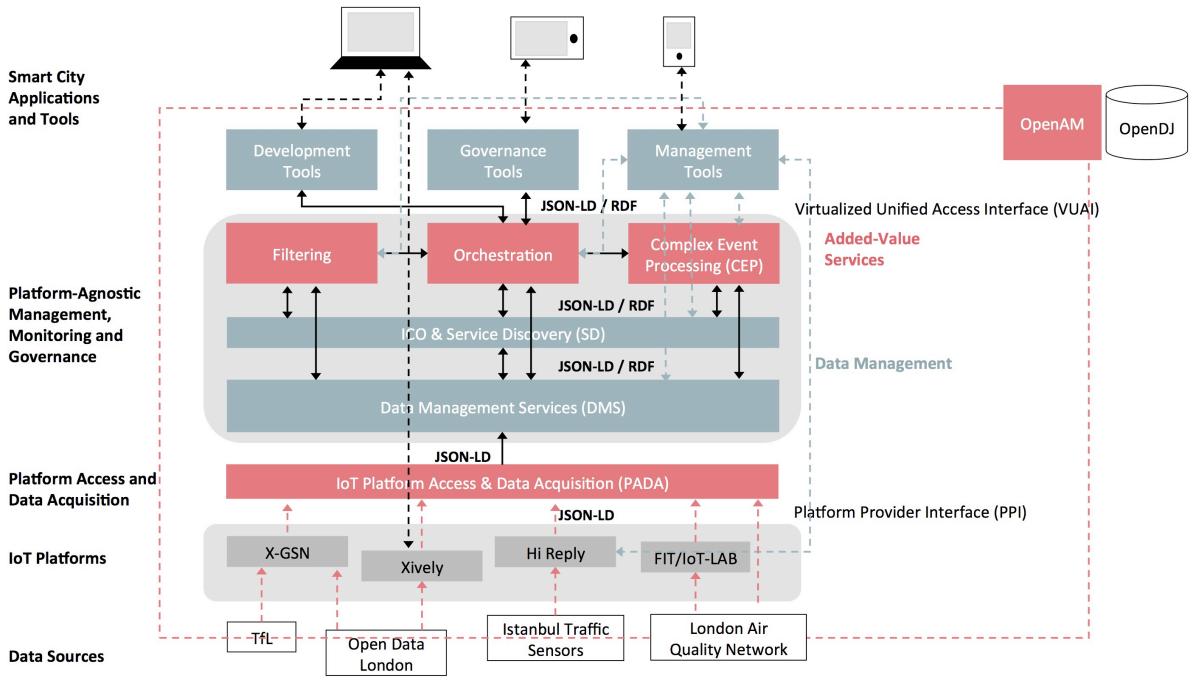


Figure 8: Detailed description about interfaces using flow diagrams, class diagrams.

4.1 Interfaces to Data Management Service

4.1.1 Overview

The VITAL **Data Management Service** (**DMS**) is responsible for storing and allowing access to all kinds of metadata (i.e. metadata about IoT systems, IoT services and ICOs), as well as to historical measurement data. The data is modelled using the VITAL ontology and access to metadata and observations is provided through RESTful interfaces. Furthermore, an interface for VITAL Platform Access and Data Acquisition (PADA) module is provided in order to store metadata and measurement data into DMS. There are three RESTful interfaces to DMS that are used to:

- Store metadata and observations made by ICOs,
- Provide access to metadata and
- Provide access to observations stored in DMS.

A detailed description of these interfaces is given in the following sections. Figure 8 highlights the interaction and positioning of DMS in the VITAL architecture.

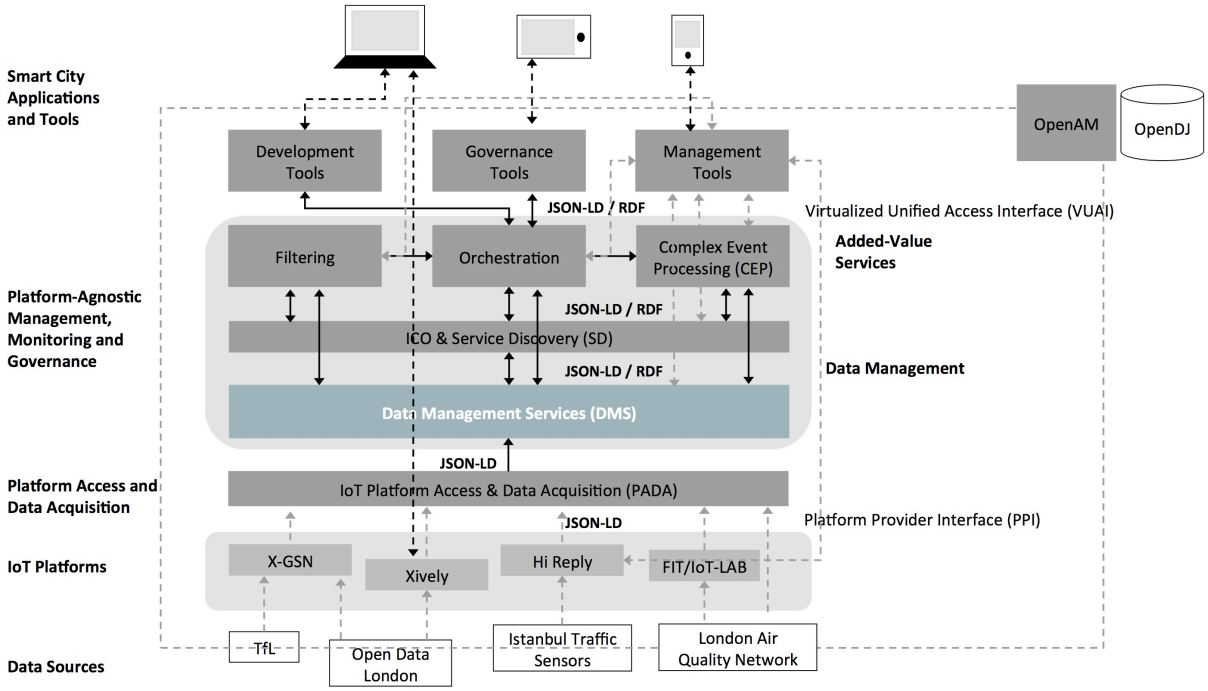


Figure 9: DMS position in VITAL architecture.

4.1.2 Specifications

DMS provides three types of interfaces for storing and retrieving all types of metadata, as well as measurement data acquired from ICOs. As mentioned in previous section, the first interface is provided to the PADA module in order to push metadata and observations into the DMS. The other two interfaces are related to the retrieval of metadata and measurement data. The interface that provides access to the observations uses the SPARQL query language, which allows easy and convenient querying, and supports continuous queries. These interfaces are now discussed in detail.

4.1.2.1 Store Metadata and Observations

In order to allow PADA to store information into DMS, a **pushData** interface is provided. This interface is provided to PADA for pushing data retrieved from various supported platforms via the PPI implementation that each of these platforms exposes. The data will be pushed to DMS by calling the **pushData** interface. This interface requires one field: data that is in JSON-LD format. Figure 9 illustrates the sequence of interactions for pushing data into DMS. Table 20 provides an example of the **pushData** interface.

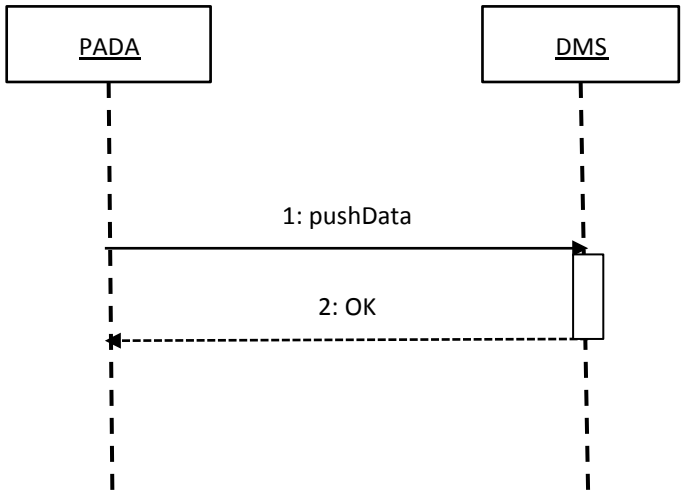


Figure 10: PADA pushes metadata and data to DMS.

Table 20: Pushing metadata and data into DMS.

	Push metadata and observations into DMS	
Description	This interface is used to push metadata and ICO observations into DMS.	
Method	POST	
URL	DMS_BASE_URL/pushData	
Request headers	Content-Type	application/ld+json or application/json

Request body	**Example**
	
	`{`
	`"@context": "http://vital-iot.eu/contexts/measurement.jsonld",`
	`"id": "http://example.com/sensor/1/observation/1",`
	`"type": "ssn:Observation",`
	`"ssn:observationProperty":`
	`{`
	`"type": "vital:OperationalState"`
	`},`
	`"ssn:observationResultTime":`
	`{`
	`"time:inXSDDateTime": "2014-08-20T16:47:32+01:00"`
	`},`
	`"ssn:featureOfInterest": "http://example.com",`
	`"ssn:observationResult":`
	`{`
	`"type": "ssn:SensorOutput",`
	`"ssn:hasValue":`
	`{`
	`"type": "ssn:ObservationValue",`
	`"value": "vital:Running"`
	`}`
	`}`	

Notes	• The request body contains the actual data to be stored by DMS, and is in JSON-LD format. Status code OK indicates if the push data request has succeeded.	

4.1.2.2 Access to stored metadata

In order to allow access to metadata stored in DMS, an interface named **metadata** is provided. VITAL applications and components, such as filtering, CEP, Orchestrator,

and Resource Discovery, use this interface to access stored metadata. The request body is a JSON object that contains the URI of an entity and the preferred type of the returned data (a flag), which can be **RDF/XML, JSON-LD,** or **JSON**. Upon a valid request the interface returns data in RDF/XML, JSON-LD or JSON format. Figure 10 illustrates the sequence of interactions for accessing metadata from DMS. Table 21 provides an example of the metadata interface.

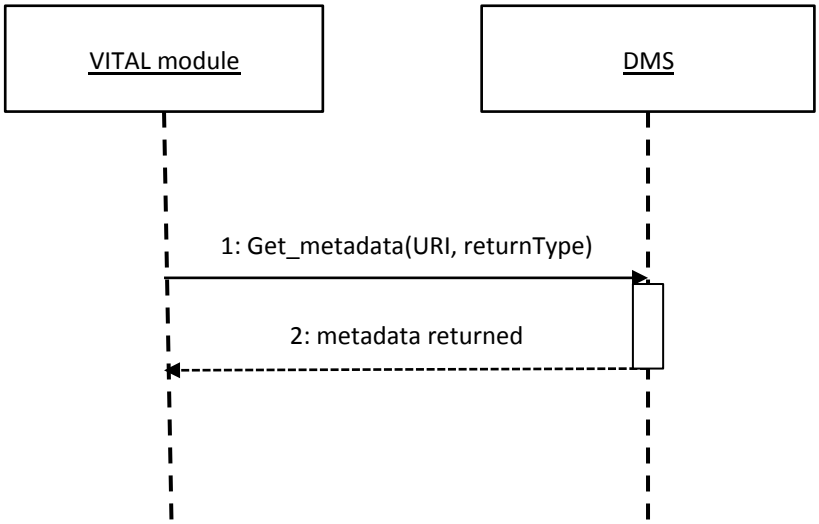


Figure 11: VITAL modules (i.e. added-value functionalities) access metadata.

Table 21: Accessing metadata from DMS.

	Pull metadata from DMS	
Description	This interface can be used to retrieve metadata from DMS.	
Method	POST	
URL	DMS_BASE_URL/metadata	
Request headers	Content-Type	application/json

Request body	**Example**
	`{`
	`  "id" : "http://104.131.128.70:8080/istanbul-traffic",`
	`  "returnType" : "JSON-LD"`
	`}`	

Response headers	Content-Type	application/json or application/ld+json or application/rdf+xml

Response body	**Example**
	`{`
	`  "@id" : "http://104.131.128.70:8080/istanbul-traffic",`
	`  "@type" : "http://vital-iot.eu/ontology/ns/IotSystem",`
	`  "managesSensor" : [ "http://104.131.128.70:8080/istanbul-traffic/sensor/2-F", "http://104.131.128.70:8080/istanbul-traffic/sensor/3-F", "http://104.131.128.70:8080/istanbul-traffic/sensor/monitoring" ],`
	`  "operator" : "zeeshan.jan@insight-centre.org",`
	`  "providesService" : [ "http://104.131.128.70:8080/istanbul-traffic/service/monitoring", "http://104.131.128.70:8080/istanbul-traffic/service/configuration", "http://104.131.128.70:8080/istanbul-traffic/service/observation" ],`	

<table>
<tr><td></td><td><pre>
 "serviceArea" : "http://dbpedia.org/page/Istanbul",
 "status" : "http://vital-iot.eu/ontology/ns/Unavailable",
 "comment" : "This is a VITAL-compliant IoT system that provides live
traffic data for Istanbul.",
 "label" : "Istanbul Live Traffic Data",
 "@context" : {
   "providesService" : {
     "@id" : "http://vital-iot.eu/ontology/ns/providesService",
     "@type" : "@id"
   },
   "status" : {
     "@id" : "http://vital-iot.eu/ontology/ns/status",
     "@type" : "@id"
   },
   "operator" : {
     "@id" : "http://vital-iot.eu/ontology/ns/operator",
     "@type" : "@id"
   },
   "comment" : "http://www.w3.org/2000/01/rdf-schema#comment",
   "managesSensor" : {
     "@id" : "http://vital-iot.eu/ontology/ns/managesSensor",
     "@type" : "@id"
   },
   "serviceArea" : {
     "@id" : "http://vital-iot.eu/ontology/ns/serviceArea",
     "@type" : "@id"
   },
   "label" : "http://www.w3.org/2000/01/rdf-schema#label"
 }
}
</pre></td></tr>
<tr><td>Notes</td><td>• The request body is a JSON object with the following fields: id, whose value is the URI of an entity (of an IoT system, an ICO, or an IoT service), and returnType, which is a flag whose value is RDF/XML, JSON or JSON-LD.</td></tr>
</table>

4.1.2.3 Access to stored observations

In order to allow access to ICO observations stored in DMS, a **sparql** interface is provided. A number of VITAL modules, such as filtering, CEP, Orchestrator, and Resource Discovery, use this interface to access observations stored within DMS. The data is requested by passing a SPARQL query to the SPARQL interface, and the preferred type of the returned data (a flag), which can be **RDF/XML, JSON-LD,** or **JSON**. Upon a valid request the interface returns data in RDF/XML, JSON-LD or JSON format. Figure 11 illustrates the sequence of interactions for accessing data from DMS. Table 22 provides an example of the SPARQL interface.

Note: It is known that the format (JSON-LD) returned by DMS is not accurate. This conversion accuracy is under investigation and will be updated in the next version of this document.

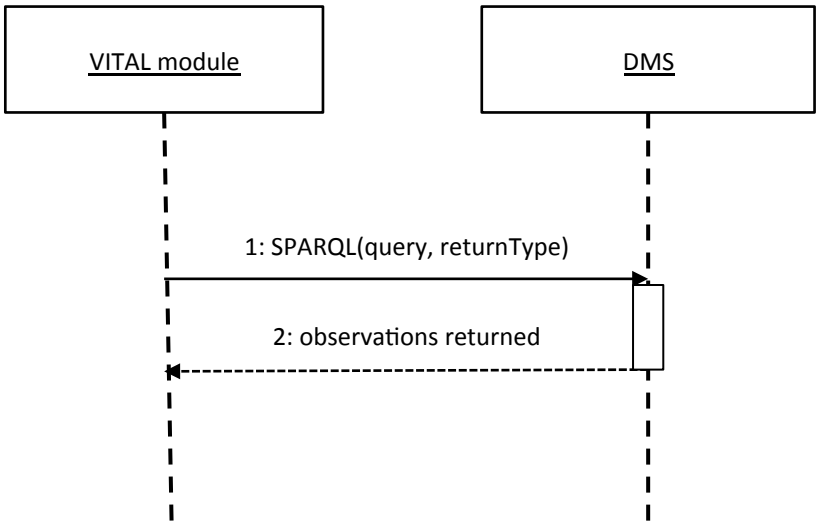


Figure 12: VITAL modules (i.e. added-value functionalities) access observations.

Table 22: Accessing observations from DMS.

	Pull data from DMS	
Description	This interface can be used to pull data from DMS using the SPARQL query language.	
Method	POST	
URL	DMS_BASE_URL/sparql	
Request headers	Content-Type	application/json

Request body	**Example**
	{
	"query": " prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
	SELECT ?uri WHERE { ?uri rdf:type <http://vital-iot.eu/ontology/ns/IotSystem> } ",
	"returnType": "JSON-LD"
	}	

Response headers	Content-Type	application/json or application/ld+json or application/rdf+xml

Response body	**Example**
	{
	"@context": {
	"rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
	"rdfs": "http://www.w3.org/2000/01/rdf-schema#",
	"rs": "http://www.w3.org/2001/sw/DataAccess/tests/result-set#",
	"xsd": "http://www.w3.org/2001/XMLSchema#"
	},
	"@graph": [
	{
	"@id": "_:N4095aab4cb5742319bd0432d9f35be6a",
	"rs:value": {
	"@id": "http://104.131.128.70:8080/istanbul-traffic"
	},
	"rs:variable": "uri"
	},
	{
	"@id": "_:N9a0633bf09314588a12a87c86acc5148",
	"rs:binding": {	

	"@id": "_:N4095aab4cb5742319bd0432d9f35be6a"
	}
	},
	{
	"@id": "_:Na1949a1cd1fc4395a097b5d8b09d4337",
	"@type": "rs:ResultSet",
	"rs:resultVariable": "uri",
	"rs:size": {
	"@type": "xsd:int",
	"@value": "1"
	},
	"rs:solution": {
	"@id": "_:N9a0633bf09314588a12a87c86acc5148"
	}
	}
	]
	}

Notes	• The request body is a JSON object with the following fields: **query**, whose value is a SPARQL query, and **returnType**, which us a flag whose value is either RDF/XML, JSON or JSON-LD.

4.2 Interfaces to Service Discovery Module

The Service Discovery module is in charge of discovering ICOs, data streams, and services horizontally integrated in the VITAL platform. It operates on the IoT resources that are stored by the Data Management Service (DMS) and are requested by other modules in the VITAL Architecture (e.g. CEP and Orchestration).

The Service Discovery module is accessible through a RESTful web service designed and implemented as part of WP4 and based on JavaEE technologies.

In the first version, the module exposes general operations, as the `ConnDMS` operation used to retrieve information on the status of the connection with the (DMS), and other operations that focus on the ICO Discovery as:

- the `getICOs` operation accepts input parameters like `latitude`, `longitude`, `radius`, `observedProperty` and returns the available ICOs in the area.
- the `getServices`, which allows the discovery of the services available in the Vital system;
- the `getSystems` that is in charge of discovering the different IoT Systems.

The following tables (Table **23** - **Table 24Error! Reference source not found.**) illustrate interfaces for the use of the services exposed**Error! Reference source not found.**.

Table 23: Get IoTSystem

	Get IoTSystem
Description	Returns the identifiers of all the IoT Systems registered with VITAL.
URL	`DISCOVERY_BASE_URL/system`
Method	POST
Input	**Example**

	<pre>{
	  "type": "http://vital-iot.eu/ontology/ns/VitalSystem"
	  "serviceArea": "http://example.com/service_area_1"
	}</pre>

Output	List of IoT Systems.

Table 24: Get ICOs

	Get ICOs
Description	Returns the identifiers of the ICOs that match the search criteria.
URL	DISCOVERY_BASE_URL/ico
Method	POST

Input	**Example**
	<pre>{
	"region":
	{
	"latitude": 40.83452,
	"longitude": 5.04737,
	"radius": 1
	},
	"observes": "http://lsm.deri.ie/OpenIoT/Temperature",
	"type": "http://vital-iot.eu/ontology/ns/VitalSensor",
	"movementPattern": "http://vital-iot.eu/ontology/ns/Mobile",
	"connectionstability": "http://vital-iot.eu/ontology/ns/Continuous",
	"hasLocalizer": true,
	"timeWindow": 10
	}</pre>

Output	**Example**
	<pre>[
	{
	"@context": "http://vital-iot.org/contexts/sensor.jsonld",
	"uri": "http://www.example.org/vital/sensor/123"
	},
	{
	"@context": "http://vital-iot.org/contexts/sensor.jsonld",
	"uri": "http://www.example.org/vital/sensor/125"
	},
	{
	"@context": "http://vital-iot.org/contexts/sensor.jsonld",
	"uri": "http://www.example.org/vital/sensor/127"
	}
	]</pre>

All the operations return information formatted in JSON-LD, according to the data models and ontologies illustrated in deliverable D3.1.1.

Furthermore, details about the interface to Service Discovery module will be available in deliverable D4.1.2 of the VITAL project.

4.3 Interface to Filtering Module

The Filtering module offers mechanisms that enable the filtering of data stemming from different sources.

Figure 13 shows an example of the interactions between the Orchetrator module, the Discovery and the Filtering. In this case, the Orchestrator directly activates the Filtering, and the objective is to resample the value of the temperature in the `room 105` every `30 minutes` in a definite period of past time. The filtering will apply its filtering functionalities and will send back the results to the Orchestrator. Moreover, the result of this request will be stored as Virtual ICO on the DMS for future requests, avoiding then performing several times the same task.

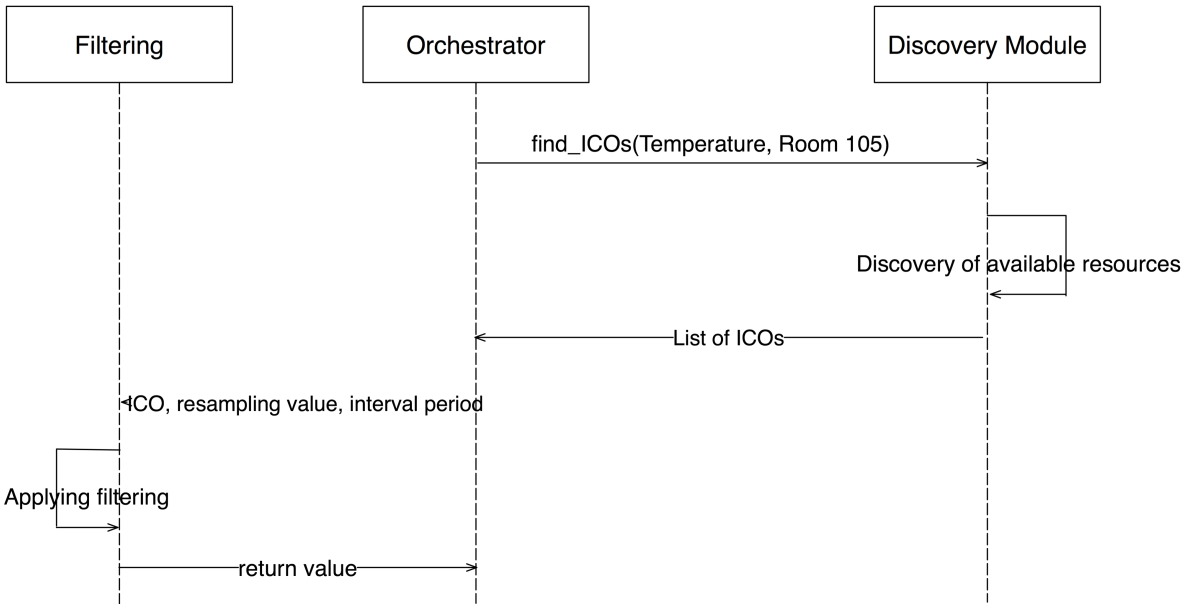


Figure 13: Example interaction Filtering and VUAIs

One first interface (Table 25: Filter on temporal basis) offered by the Filtering is the `Filter on temporal basis`. In this case, the filtering accepts a POST request that has in input the value of the sampling frequency; it will return the resampled data.

Table 25: Filter on temporal basis

	Resampling
Description	Resampling data within a fix interval
URL	base_url/filtering/threshold
Method	POST

Input	`Sampling value.`
	
	Example format:
	{ ICO: “uri”
	resampling: “30 mins”
	interval: “dd/mm/yyyy-dd/mm/yyyy”
	
	}

Output	Value resampled.

Since the filtering functionalities have not been implemented yet (i.e. the respective work is still ongoing), additional details will be available in the Deliverable 4.2.1 of the VITAL Project.

4.4 Interfaces to CEP Module

4.4.1 Overview

The Complex Event Processing module is an added-value mechanism in the VITAL project. It is responsible for managing event processing over observation streams. The Complex Event Processing is accessible through a RESTful web service researched and implemented as part of WP4 and based on JavaEE technologies. In the first version, the module exposes interfaces for managing CEP-generated ICOs (CEPICOs) with event and rule specifications.

A **CEPICO** is an internally generated virtual ICO that represents a CEP instance that is running user defined DOLCE rules over a specified observation stream. CEPICO generates results of detected complex events and serves them as observations.

4.4.2 Specifications

CEP provides four interfaces to access metadata and manage CEPICOs:

- **getcepicos**: returns the information of previously generated CEPICOs
- **getcepico**: returns the metadata of a CEPICO defined by its URI
- **createcepico**: receives data source URI and dolce specifications as input, and generates a new CEPICO with these specifications. If an ID provided, this operation will update the existing CEPICO with this ID
- **deletecepico**: receives an ID as input and deletes the specified CEPICO

The first interface returns information or metadata of all instantiated CEPICOs, in the information provided by this interface. VitalCEP can manage a big number of CEPICOs processing as many Dolce specification as number of CEPICOs. The Dolce specifications of each CEPICO is not provided in order to do not overload network with long messages and to avoid users dealing with the task of processing long messages, that may provoke to overload their systems.

To get the Dolce specification processed by a CEPICO, users should use the second interface, that returns all metadata of a CEPICO.

Field	Mandatory	Description
source	Yes	This field specifies the URI of the source data stream.
dolceSpecification	Yes	DOLCE specifications.
complex	Yes	Descriptions of “Complex” events in DOLCE.
event	Yes	Descriptions of “Event” section in DOLCE.
external	No	Description of external variables to be used in DOLCE rules.
definition	Yes	Definition section of a DOLCE event or

		complex.
id	Yes	Identifier for a DOLCE complex or event.

	Get CEPICOs
Description	Gets metadata about all CEPICOs created by the user
URL	CEP_BASE_URL/getcepicos
Method	GET
Input	-

Output	**Example**
	[
	{
	"context": "http://vital-iot.eu/contexts/sensor.jsonld",
	"uri": "http://example.com/sensor/1",
	"name": "Traffic Incidents.",
	"type": "vital:CEPSensor",
	"description": "CEPICO for Traffic Incidents.",
	"ssn:observes":
	[
	{
	"type": "vital:ComplexEvent",
	"uri": "http://example.com/sensor/1/cep/complex-event"
	}
	]
	},
	{
	"@context": "http://vital-iot.eu/contexts/sensor.jsonld",
	"uri": "http://example.com/sensor/2",
	"name": "Live Traffic Data.",
	"type": "vital:CEPSensor",
	"description": "CEPICO for Live Traffic Data.",
	"ssn:observes":
	[
	{
	"type": "vital:ComplexEvent",
	"uri": "http://example.com/sensor/2/cep/complex-event"
	}
	]
	}
	]

	Get CEPICO
Description	Gets metadata about a CEPICO with a specific URI created by the user
URL	CEP_BASE_URL/getcepico
Method	POST

Input	{
	"uri": "http://example.com/sensor/2"
	}

Output	**Example** { "context": "http://vital-iot.eu/contexts/sensor.jsonld", "uri": "http://example.com/sensor/2", "name": "Live Traffic Data.", "type": "vital:CEPSensor", "description": "CEPICO for Live Traffic Data.", "ssn:observes": [{ "type": "vital:ComplexEvent",

<table>
<tr><td></td><td>

```
        "uri": "http://example.com/sensor/2/cep"
      }
    ],
    "vital:source": "http://example.com/sensor/1/speed",
    "vital:dolceSpecification": "
      \"complex\": [
        {
          \"definition\":\"payload{int    SensorId    =    sensorId,float
Px=pointX,float Py=pointY,float Speed =speed};detect ImmEvent;\",
          \"-id\": \"TrafficData\"
        },
        {
          \"definition\": \" payload { int SensorID = sensor_id,   int
Valor = value }; detect temperature;\",
          \"-id\": \"SiempreActivo\"
        },
        {
          \"definition\": \" payload { int SENSORID = sensor_id,   int
Valor = value }; detect temperature where (count(temperature) > 1) in
[T_WINDOW];\",
          \"-id\": \"Calooooooor\"
        },
        {
          \"definition\":   \"payload{int   SensorId   =   sensorId,float
Px=pointX,float Py=pointY,float Speed=speed};detect ImmEvent where speed
< 20;\",
          \"-id\": \"TrafficProblem\"
        }
      ],
      \"event\": [
        {
          \"definition\":    \"use{int    sensorId,float    pointX,float
pointY,float speed};\",
          \"-id\": \"ImmEvent\"
        },
        {
          \"definition\": \" use { int sensor_id, int value }; accept {
value > 39 };\",
          \"-id\": \"temperature\"
        }
      ],
      \"external\": [
        {
          \"name\": \"TEMP_ALERT\",
          \"value\": \"39\",
          \"type\": \"int\"
        },
        {
          \"name\": \"umbral01\",
          \"value\": \"21\",
          \"type\": \"int\"
        },
        {
          \"name\": \"T_WINDOW\",
          \"value\": \"1 seconds\",
          \"type\": \"duration\"
        }
      ]"
    }
  }
}
```

</td></tr>
</table>

	Create / Update CEPICO
Description	Creates or updates the specified CEPICO
URL	CEP_BASE_URL/createcepico

Method	PUT

Input	<pre>{
	  "context": "http://vital-iot.eu/contexts/sensor.jsonld",
	  "name": "Live Traffic Data.",
	  "type": "vital:CEPSensor",
	  "description": "CEPICO for Live Traffic Data.",
	  "vital:source": "http://example.com/sensor/1/speed",
	  "vital:dolceSpecification": ”
	  \”complex\”: [
	  {
	  \”definition\”:\”payload{int SensorId = sensorId,float Px=pointX,float Py=pointY,float Speed =speed};detect ImmEvent;\”,
	  \”-id\”: \”TrafficData\”
	  },
	  {
	  \”definition\”: \” payload { int SensorID = sensor_id, int Valor = value }; detect temperature;\”,
	  \”-id\”: \”SiempreActivo\”
	  },
	  {
	  \”definition\”: \” payload { int SENSORID = sensor_id, int Valor = value }; detect temperature where (count(temperature) > 1) in [T_WINDOW];\”,
	  \”-id\”: \”Calooooooor\”
	  },
	  {
	  \”definition\”: \”payload{int SensorId = sensorId,float Px=pointX,float Py=pointY,float Speed=speed};detect ImmEvent where speed < 20;\”,
	  \”-id\”: \”TrafficProblem\”
	  }
	  ],
	  \”event\”: [
	  {
	  \”definition\”: \”use{int sensorId,float pointX,float pointY,float speed};\”,
	  \”-id\”: \”ImmEvent\”
	  },
	  {
	  \”definition\”: \” use { int sensor_id, int value }; accept { value > 39 };\”,
	  \”-id\”: \”temperature\”
	  }
	  ],
	  \”external\”: [
	  {
	  \”name\”: \”TEMP_ALERT\”,
	  \”value\”: \”39\”,
	  \”type\”: \”int\”
	  }
	  ]”
	
	}</pre>

	Delete CEPICO
Description	Deletes a CEPICO with the specified URI
URL	CEP_BASE_URL/deletecepico
Method	DELETE

Input	<pre>{
	  "uri": "http://example.com/sensor/2"
	}</pre>

Output	**-**

4.5 Interfaces to Workflow Management

The Workflow Management / Orchestrator module, is a higher-level module that offers the functionality to combine systems and services in the Vital architecture to create more complex meta-services. To support this, it utilizes workflow descriptions generated by users and modeled in a scripting language. Workflows are then deployed, enabled/disabled, removed and executed as new services.

The REST api exposed is described in the following tables:

	Deploy/Create a service
Description	Deploys a new service based on the workflow
URL	ORCHESTRATOR_BASE_URL`/service`
Method	POST

Input	**Example:**
	{
	"name": "Name of Service",
	"description": "Description of service",
	"access": "PUBLIC", // PRIVATE
	"workflowScript": "script of workflow as a string"
	}

Output	**SUCCESS / 200**
	{
	"id" : 1,
	"url": " ORCHESTRATOR_BASE_URL/execute/1",
	"status" : "DISABLED",
	"access": "PUBLIC", // PRIVATE
	"name": "Name of Service",
	"description": "Description of service",
	"workflowScript": "script of workflow as a string"
	}
	CONFLICT / 409
	{
	"error" : "script evaluation failed"
	}

Notes	**It creates a new service based on the workflow script and attributes a url to be called / executed.**

	Update a service
Description	Updates status or permissions of a deployed service
URL	ORCHESTRATOR_BASE_URL`/service/{:serviceId}`
Method	PUT

Input	**Example:**
	{
	"status" : "ENABLED" // or "DISABLED",
	"access": "PUBLIC", // PRIVATE
	}

Output	**SUCCESS / 200** { "id" : 1, "url": " ORCHESTRATOR_BASE_URL/execute/1",

	"status" : "ENABLED" // or "DISABLED",
	"access": "PUBLIC", // PRIVATE
	"name": "Name of Service",
	"description": "Description of service",
	}
	NOT FOUND / 404
	{
	"error" : "service not found"
	}

Notes	**It updates the service with status value in the input request**

	GET a list of services
Description	Returns the list of deployed services
URL	ORCHESTRATOR_BASE_URL`/service`
Method	GET
Input	

Output	**SUCCESS / 200**
	[{
	"id": 1,
	"url": " ORCHESTRATOR_BASE_URL/execute/1",
	"status": "DISABLED",
	"access": "PUBLIC", // PRIVATE
	"name": "Name of Service",
	"description": "Description of service",
	"workflowScript": "script of workflow as a string"
	}, {
	"id": 4,
	"url": " ORCHESTRATOR_BASE_URL/execute/4",
	"status": "ENABLED",
	"access": "PUBLIC", // PRIVATE,
	"name": "Name of Service",
	"description": "Description of service",
	"workflowScript": "script of workflow as a string"
	}]

Notes	

	Undeploy / Delete a service
Description	Deletes the specified service
URL	ORCHESTRATOR_BASE_URL`/service/{:serviceId}`
Method	DELETE
Input	

Output	**NO-CONTENT / 204**
	
	NOT FOUND / 404
	{
	"error" : "service not found"
	}

Notes	

	Execute a service
Description	Execute a deployed service
URL	ORCHESTRATOR_BASE_URL`/service/execute/{:serviceId}`
Method	POST
Input	Depends on the workflow script

Output	**SUCCESS / 200**
	Depends on the workflow script
	
	NOT FOUND / 404
	{
	"error" : "service not found"
	}
	SERVICE UNAVAILABLE / 503
	{
	"error" : "service disabled"
	}
	INTERNAL SERVER ERROR / 500
	{
	"error" : "<The message of the error generated>"
	}

Notes	**The service execution success depends on the successful completion of all the steps. Failure can occur if for example an IotSystem is not available and the scripts requires observations from this system.**

All calls are secure, i.e. a user must be authenticated before viewing, deploying, updating or undeploying services. Execution of a deployed service is either public or private as set upon creation. Public means that this service can be executed from any user or application, provided he/she knows the URL. Private restricts execution to the user that created the service.

5 SECURITY OF VIRTUALIZED UNIFIED ACCESS INTERFACES

A solution based on an open-source identity and access management system, supporting Single Sign-On (SSO) and various protocols and authentication methods, has been devised for the VITAL security layer.

A common terminology related to security defines the following roles:

- **Identity Provider (IdP)**: Holds all information about the users (for example information stored in an LDAP directory), and its main duty is to authenticate users, and decide what kind of information it shares about them with other parties.
- **Service Provider (SP)**: Is an extra layer in front of the web application. Its job is to authorize page requests, and, if there is no authenticated session at the service provider, to initiate an authentication request to the identity provider.

Identity providers supply user information, while service providers consume this information and give access to secure content.

SAML (Security Assertion Markup Language) is an XML-based, open-standard data format for exchanging authentication and authorization data between parties, in particular, between an identity provider and a service provider. **SAML 2.0** enables web-based authentication and authorization including cross-domain SSO; it is a widely used standard, cross-platform, with many different open-source implementations.

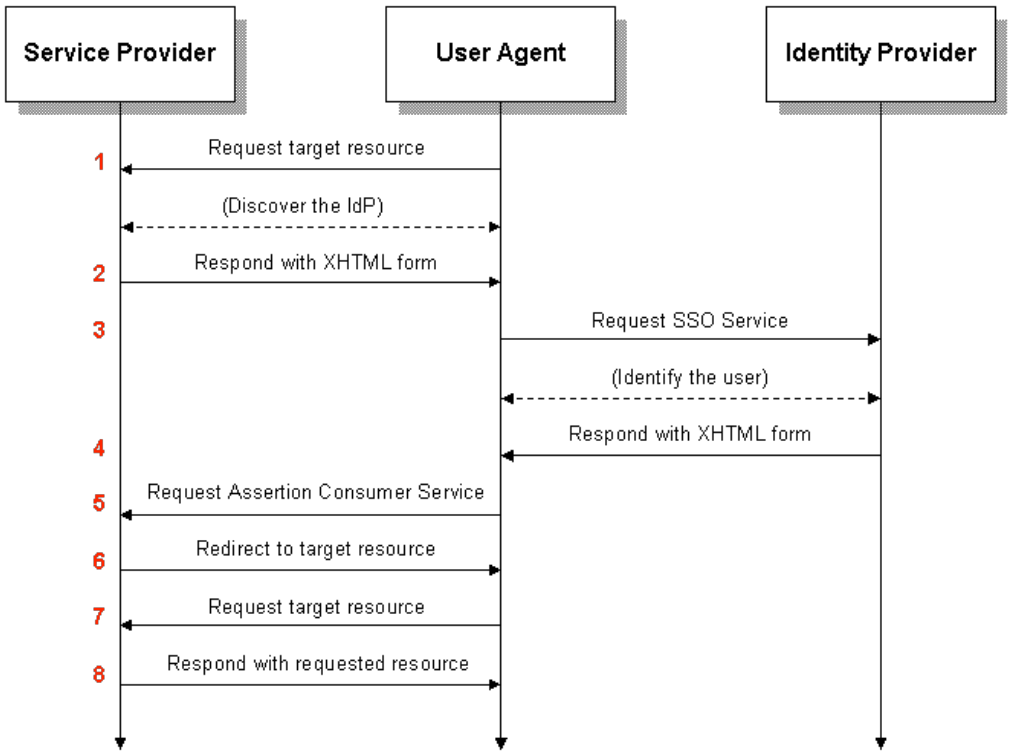


Figure 14: Basic flow of an SAML 2.0 SSO through a web browser using HTTP.

A technology that has been evaluated for use in VITAL is **Shibboleth**. Shibboleth is a free, open-source project that provides “Federated Single Sign-on”, and that implements the HTTP/POST, artifact, and attribute push profiles of SAML, including

both identity provider and service provider components. Standards adoption allows the integration between these Shibboleth components and other solutions.

OpenAM is an open-source product by ForgeRock, started as continuation of the Oracle OpenSSO project, designed to provide services for the Web, the Cloud, mobile devices and things. The code of OpenAM is licensed under the CDDL license. Every release of the OpenAM platform, except the most recent one, is free to download. The latest version is available only after a paid subscription, in addition to a more in-depth customer service. OpenAM has a highly scalable, modular, architecture that supports:

- Authentication
- Single Sign-On (SSO)
- Authorization
- Federation
- Entitlements
- Adaptive authentication
- Strong authentication
- Web service security

It also provides a useful REST API to make these features accessible.

OpenAm can exploit a number of policy agents (e.g. web policy agents, J2EE policy agents), provided by Forgerocks, which are software components enforcing policy for OpenAM. In particular, a **Web Policy Agent** installed in a web server can intercept requests from users trying to access a protected web resource, and denies access until the user has authorization from OpenAM to access the resource.

OpenAM and Shibboleth can be combined in different ways (e.g. OpenAM can be used as both the identity and the service provider, OpenAM identity provider can be combined with Shibboleth service provider) to obtain different setups, each with different characteristics. A setup combining OpenAM identity provider federation with Shibboleth service provider solution has been configured for evaluation purposes.

The technologies selected for the VITAL security layer, based on ForgeRock's open-source components, are described in the following sections.

5.1 Overview of VUAI Security Architecture and Implementation

5.1.1 Security Modules and Libraries

OpenAM can manage access to resources available over the network thanks to a centralized access management service that controls:

- who can access what resource
- when a resource can be accessed
- the conditions under which a resource can be accessed

OpenAM handles both authentication (the process that confirms an identity) and authorization (the process that, using policies defined in OpenAM, decides whether to grant access to someone who has been authenticated).

OpenAM can protect a generic web page, an application, a web service or anything accessible over HTTP. Thanks to decoupling policies from applications, if a policy changes or an issue is found after an application is deployed, the only change to do

is to update the policy definition in OpenAM, and the application can be left untouched.

OpenAM supports also the following authentication/authorization protocols:

- OAuth 2.0
- OpenID Connect
- SAML 2.0 SSO & Federation

In the **OAuth 2.0 flow**, OpenAM can function as:

- **OAuth 2.0 authorization server**: In this role, OpenAM authenticates resource owners and obtains their authorization in order to return access tokens to clients. OpenAM supports the four main grants for obtaining authorization described in the OAuth 2.0 RFC: the authorization code grant, the implicit grant, the resource owner password credentials grant, and the client credentials grant.
- **OAuth 2.0 client**: OpenAM can function as an OAuth 2.0 client for installations where the web resources are protected by OpenAM. When OpenAM functions as an OAuth 2.0 client, OpenAM provides an OpenAM SSO session after successfully authenticating the resource owner and obtaining authorization. This means the client can then access resources protected by policy agents. In this respect the OpenAM OAuth 2.0 client is just like any other authentication module, one that relies on an OAuth 2.0 authorization server to authenticate the resource owner and obtain authorization.

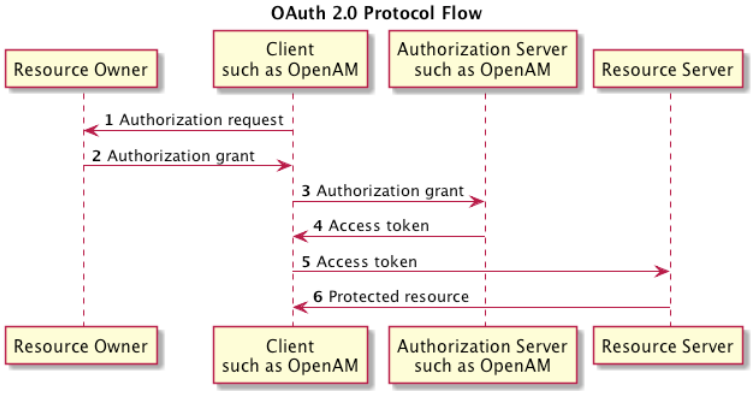


Figure 15: OpenAM in the OAuth 2.0 flow.

In the **OpenID Connect flow**, OpenAM can function as **OpenID Provider**; OpenID Provider holds the user information and grants access. In its role as OpenID Provider, OpenAM:

- allows OpenID Connect relying parties (clients) discover its capabilities
- handles both dynamic and static registration of OpenID Connect relying parties
- responds to relying party requests with authorization codes, access tokens, and user information according to the Authorization Code and Implicit flows of OpenID Connect
- manages sessions

SAML 2.0 SSO is part of federated access management. Federation enables access management cross organizational boundaries. Federation helps organizations share identities and services without giving away their identity information, or the services they provide.

To achieve SAML 2.0 SSO, OpenAM separates identity providers from service providers. These two components are included in a circle of trust:

- An identity provider stores and serves identity profiles, and handles authentication.
- A service provider offers services that access protected resources, and handles authorization.
- A circle of trust groups at least one identity provider and at least one service provider who agree to share authentication information, with assertions about authenticated users that let service providers make authorization decisions. Providers in a circle of trust share *metadata*, configuration information that federation partners require to access each other's services.
- SAML 2.0 SSO maps attributes from accounts at the identity provider to attributes on accounts at the service provider. The identity provider makes assertions to the service provider, for example to attest that a user has authenticated with the identity provider. The service provider then consumes assertions from the identity provider to make authorization decisions.

Using this feature, OpenAM could be combined with other SAML compliant components, such as Shibboleth ones.

In fact, an alternative to the **OpenAM plus Policy Agent** solution is combining OpenAM and Shibboleth in a solution where OpenAM plays the role of the hosted identity provider and Shibboleth plays the role of remote service provider instead of the Web Policy Agent. OpenAM shares information about users with the service provider and handles the authentication flow. Shibboleth offers services that access protected resources, and handles authorization. Shibboleth maps attributes from accounts at the identity provider to attributes on accounts at the service provider. The identity provider makes assertions to the service provider. These attributes are sent along with every assertion and can be used to implement some authorization rules similar to the policy agent ones (e.g. which users or users with what attributes can access a resource).

The solution using ForgeRock Policy agent has been preferred to the one based on Shibboleth service provider due to the richest REST support. More important, the authorization rules that can be defined in the latter solution are very few and less specific than the rules managed by the Web Policy Agent and related to the web server (e.g. Apache HTTP) on which the service provider is running.

Another solution that has been considered is to use, as a service provider, ForgeRock **OpenIG** (**Open Identity Gateway**), and is based on a reverse proxy architecture. All HTTP traffic to each protected application is routed through OpenIG, enabling inspection, transformation and filtering of each request. By inspecting the traffic, OpenIG is able to intercept requests that would normally require the user to authenticate, obtain the user's login credentials, and send the necessary HTTP request to the target application, thereby logging in the user without modifying or installing anything on the application.

To allow protection of resources retrieved using a PUSH-based mechanism access control can be performed on the subscribe and unsubscribe services (**SubscribeToObservationStream** and **UnsubscribeFromObservationStream** in the case of PPIs); for these services the authorization process is just as in the "normal" PULL-based scenario.

We should also take into account the case where access to resources would not be triggered by HTTP requests that can be intercepted by the Policy Agent and would thus need to evaluate on demand whether the user is allowed to access the resource(s) or not. This might be useful also in requests where the resource URI is contained in the request body (it would be needed to intercept the requests and extract the URIs).

The possibility to perform authorization without the help of a Policy Agent is offered by the Vital Security Adapter described in document D5.1.2. This software module exposes a RESTful interface which allows to specify the resource(s) and the user Token ID (obtained at the time of authentication) and returns a decision about which operations are allowed on the specified resource for that user; this software module contacts the OpenAM server to obtain this information.

5.1.2 Authentication

The authentication can be provided by using a variety of authentication modules connected to identity repositories that store identities and provide authentication services. These identity repositories can be implemented in various technologies, such as LDAP directories, relational databases, one-time password services, other standards-based access management systems and more.

A custom data store and a custom authentication module based on semantic technologies will be considered in the next iteration.

OpenAM allows to you chain together the authentication services used, to configure stronger authentication for more sensitive resources for example. In the proposed architecture, authentication is provided by using OpenDJ as LDAP directory where users and groups are stored.

Clients request authentication to the identity provider. If the user is authenticated to the identity provider, an authentication token is returned to the user. This is an SSO Token value, an encrypted reference to the session stored by OpenAM.

This token has by default a duration of 120 minutes. This duration is configurable.

5.1.3 Authorization

In the proposed architecture, the authorization can be provided by using OpenAM to manage access policies separated from applications and resources, thanks to a **Web Policy Agent**. It can be installed as a plugin in the most common web and application servers and it requests policy decision from OpenAM.

Policy Agent checks the token's validity by querying OpenAM. This allows OpenAM to make policy decisions based on who is authenticated.

Every policy is defined based on:

- **Resources**: The resource definitions constrain which resources, such as web pages or access to the boarding area, the policy applies to.
- **Action**: The actions are verbs that describe what the policy allows users to do to the resources (e.g. GET, POST).
- **Subject conditions**: The subject conditions constrain who the policy applies to (e.g. to all authenticated users, or only to administrators, or to specific groups of users).
- **Environment conditions**: The environment conditions set the circumstances under which the policy applies.
- **Response attributes**: The response attributes define information that OpenAM attaches to a response following a policy decision (e.g. a name, an email address).

When queried about whether to let a user through to a protected resource, OpenAM decides to authorize access or not based on applicable policies. OpenAM communicates its decision to the application using OpenAM for access management. The policy agent installed on the server where the application runs enforces the authorization decision from OpenAM.

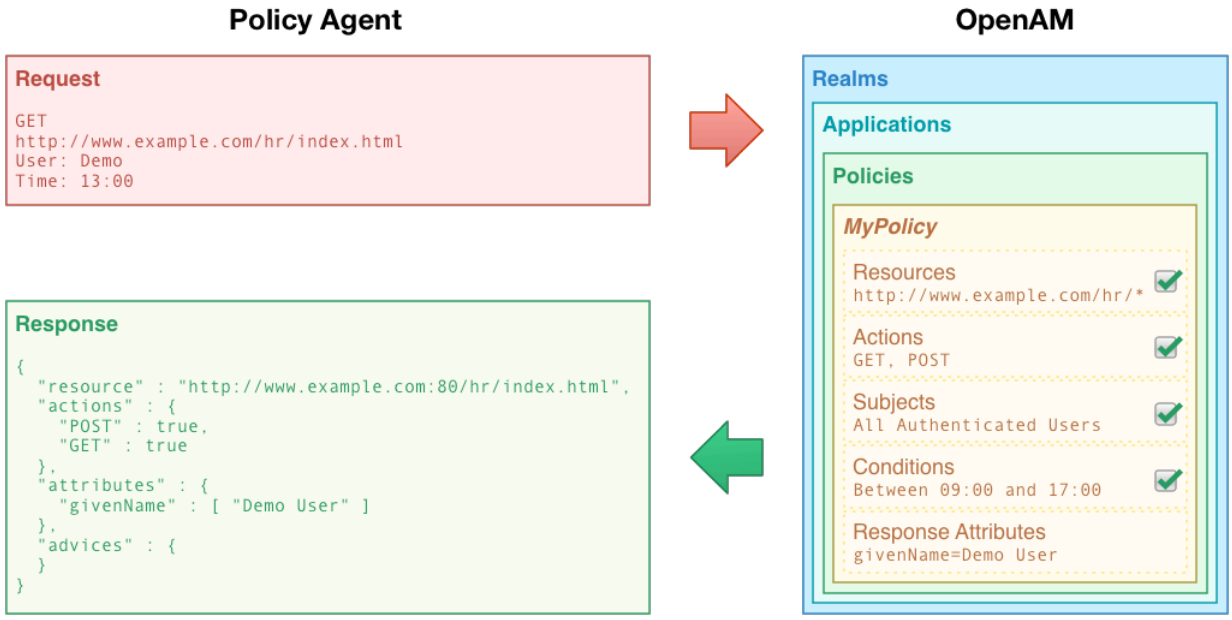


Figure 16: Policy Agent & OpenAM integration.

OpenAM relies on policies to reach authorization decisions, such as whether to grant or to deny access to a resource. OpenAM acts as the **Policy Decision Point** (**PDP**), whereas OpenAM Policy Agent act as the **Policy Enforcement Point** (**PEP**). The policy agent takes responsibility only for enforcing a policy decision made by OpenAM. When applications and their policies are configured in OpenAM, OpenAM is also used as the **Policy Administration Point** (**PAP**).

When a PEP requests a policy decision from OpenAM, it specifies the target resource(s), the application and information about the subject and the environment. OpenAM as the PDP retrieves policies within the specified application that apply to the target resource(s). OpenAM then evaluates those policies to make a decision

based on the conditions matching those of the subject and environment. When multiple policies apply to a particular resource, the default logic for combining decisions is that the first evaluation resulting in a decision to deny access takes precedence over all other evaluations**.** OpenAM only allows access if all applicable policies evaluate to a decision to allow access.

OpenAM communicates the policy decision to the PEP. The concrete decision, applying policy for a subject under the specified conditions, is called an **entitlement**. The entitlement indicates the resource(s) it applies to, the actions permitted and denied for each resource, and optionally response attributes and advice. When OpenAM denies a request due to a failed condition, OpenAM can send advice to the PEP, and the PEP can then take remedial action.

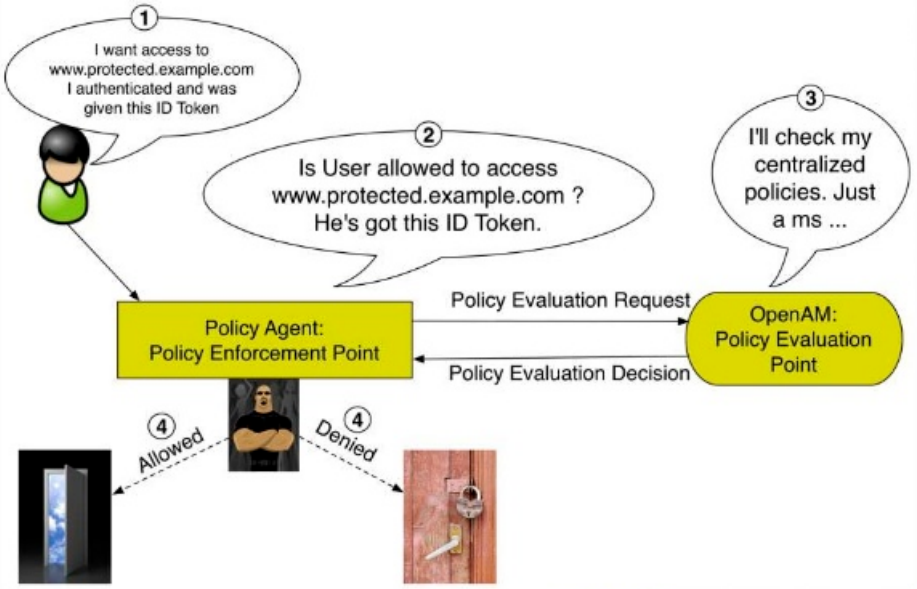


Figure 17: Policy Agent & OpenAM authentication/authorization flow.

The proposed architecture for VUAI security is summarized in Figure 18.

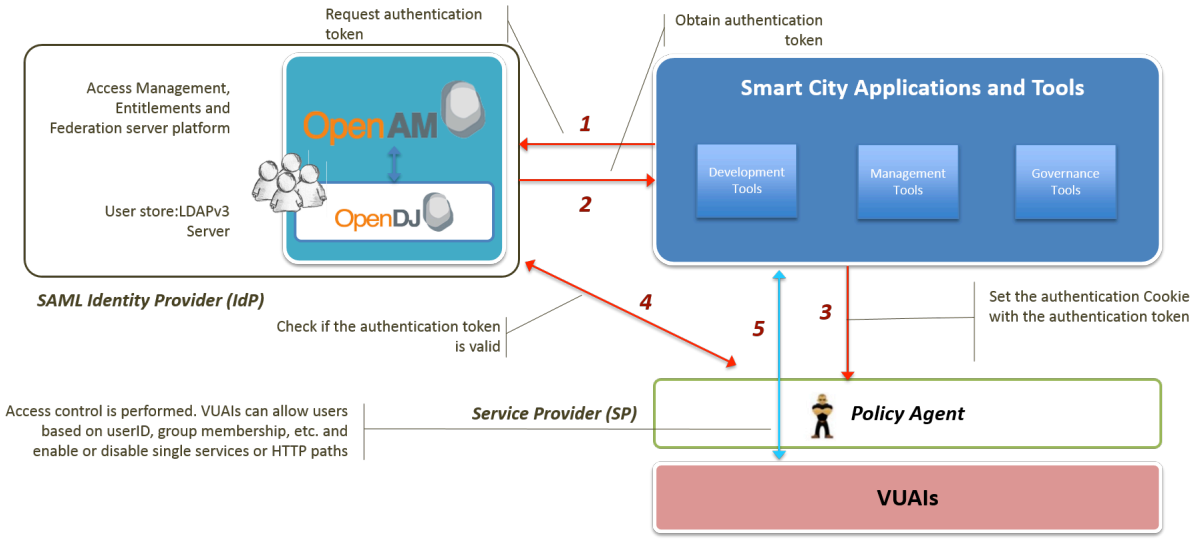


Figure 18: VUAI security architecture overview.

The protection of the VUAIs exposed to the Smart City Applications and Tools follows the same approach as the PPI protection. The policy agent checks the policies defined for a resource, in this case a VUAI endpoint.

Figure 19 shows the proposed architecture for PPI security.

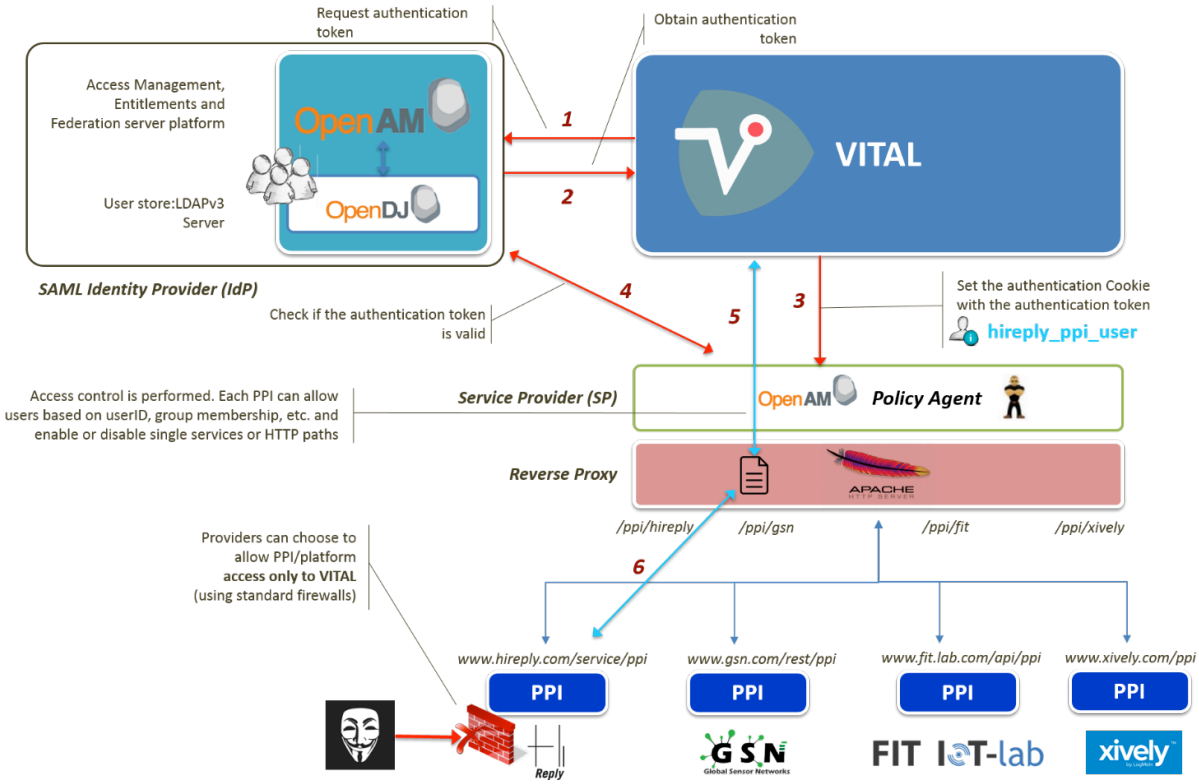


Figure 19: PPI security architecture overview.

The security layer architecture contains three main components:

- **OpenAM identity provider**, described in 5.1.2
- **Policy Agent service provider**, described in 5.1.3
- **Reverse Proxy** that masks the underlying PPI implementations

OpenAM identity provider can use different data stores, namely Oracle Directory Server, Microsoft Active Directory, IBM Tivoli Directory Server, ForgeRock OpenDJ. For VITAL we are currently using OpenDJ data store; OpenDJ is a high performance and highly scalable open source LDAPv3 compliant server. Moreover, it is fully integrated with OpenAM. OpenDJ comes embedded as a configuration store, and it also exposes a RESTful API.

The choice of using a reverse proxy provides high flexibility; with this solution, it is possible to attach different services behind the proxy with simple configurations and a firewall configuration for each service. That way, there is a single point of security enforcement, and the overall system is easily extendable and scalable.

Using OpenAM, VITAL services will authenticate to the identity provider using *ad-hoc technical users* and will obtain the *access token* that the Policy Agent will use to grant or not the access to the specific location requested by the user.

On the reverse proxy, a location is mapped for every PPI.

The main advantage of using OpenAM plus the Web Policy Agent is that it is not needed to modify, redeploy or redesign the applications that need a security layer. Also, OpenAM exposes RESTful APIs that enable JSON or XML over HTTP, allowing users to access authentication, authorization, and identity services from web applications using simple REST clients.

Web Policy Agent is completely integrated with OpenAM, fully compatible and REST friendly. Also, Web Policy Agent make it possible to define very detailed rules for the authorization flow with a high granularity (e.g. rules on group membership, or on user account).

These two components can be used in the same solution, and make it possible to integrate and federate different services in a way that is language- and platform-agnostic and totally REST friendly.

5.2 Example: PPI Authentication and Authorization

OpenAM exposes a REST API for authentication. Username/password authentication returns an authentication token, as shown in Table 26. This must be set in an HTTP header as a cookie, named **vitalAccessToken**, in order to be checked by the service provider, the Policy Agent.

Table 26: Request for authentication token.

	Get authentication token	
Description	This is the authentication endpoint, where authentication token can be requested.	
URL	OPENAM_BASE_URL/idp/json/authenticate	
Method	POST	
Request headers	Content-Type	application/json
	X-OpenAM-Username	vitalUser
	X-OpenAM-Password	vitalUserPassword
Response headers	Content-Type	application/json

Response body	**Example**
	
	{
	"tokenId": "AQICGU4ok.*ANTc3MTM2MjY5Ng..*",
	"successUrl": "/idp/console"
	}	

Set the **vitalAccessToken** cookie in all the requests with the **tokenId** value. Using the **vitalAccessToken** cookie, all the PPI services exposed under the Hi Reply endpoint, for example, will be accessible as shown in Table 27.

Table 27: Request for a protected resource.

	Get metadata about Hi Reply	
Description	This service can be used to get metadata about Hi Reply.	
URL	https://www.hireply.com/metadata	
Method	POST	
Request headers	Content-Type	application/json
	Cookie	vitalAccessToken=AQICGU4ok.*ANTc3MTM2MjY5Ng..*

Request body	**Example**
	
	{
	}	

Response headers	Content-Type	application/json

Response body	**Example**
	{
	. . .
	}	

At the moment of the request, the Policy Agent checks the authentication token and applies the policy defined in the identity provider (as described in Section 5.1.3) to grant or not authorization to the requested resource.

5.3 Example: VUAI Authentication and Authorization

As described in Section 2, VUAIs enable unified access to external IoT systems, and to VITAL components. Authentication and authorization to VUAIs can be provided in the same way as described in Section 5.2. Policies can be defined based on user membership to certain groups, representing roles significant to each VUAI.

6 PPI PROTOTYPE IMPLEMENTATIONS

6.1 PPI Implementation for Open Data Sources

At the moment, we have implemented PPI for the following open data feeds:

- for the footfall data feed, provided by Springboard
- for the bus arrival data feed, provided by Transport for London (TfL)

More details about each implementation are given in the following sections.

6.1.1 PPI Implementation for Footfall Data Feed

Springboard has provided us with access to a **footfall data feed** that contains values collected in an hourly basis by small counting devices installed in several locations across **Camden**. Each value in the data feed contains (among others) the following pieces of information:

- when the value was observed
- where the value was observed (i.e. in which location)
- how many people were counted walking up that location
- how many people were counted walking down that location

In order to read values from that data feed within a specific time range, we need to submit an appropriate HTTP GET request to a pre-defined endpoint. The response to that request contains the requested data in **JSON** format.

We decided that, in the case of the footfall data, each location represents a sensor that observes two properties: (1) the number of people walking up that location, and (2) the number of people walking down that location.

We also manually stored information about the locations, where counting devices are mounted, into **Elasticsearch**.

Then we moved on the actual implementation of PPI, which in essence was (part of) a Java web application deployed in **Wildfly** application server. We first set up a task that pulls values periodically (once every hour) from the data feed, and stores them into Elasticsearch. We then implemented the following PPI services:

- Get IoT system metadata
- Get sensor metadata
- Get sensor measurements
- Get supported performance metrics
- Get performance metrics

We exposed the PPI services as RESTful web services using **RESTEasy**, a JAX-RS implementation. For the sensor measurements, only the pull-based mechanism is supported. As for the performance metrics, the current implementation uses **JavaMelody** to monitor the following performance metrics:

- Available memory

- Errors
- Maximum number of requests
- Pending requests
- Served requests
- System load
- System uptime
- Used memory

Table 28 lists the technologies, libraries, and tools used in the implementation of PPI for Springboard's footfall data feed.

Table 28: Technologies used in footfall data feed PPI implementation.

Framework/Library	Version
Elasticsearch	1.5.2
Java Platform, Enterprise Edition	8
JavaMelody	1.56.0
jsonld-java	0.6.0-SNAPSHOT
Log4j	2.2
Maven	3.3.3
RESTEasy	3.0.9.Final
WildFly	9.0.0.CR1

6.1.2 PPI Implementation for Bus Arrival Data Feed

Transport for London (**TfL**) provides (among others) a live data feed with bus arrival information. Each value in that data feed contains (among others) the following:

- the bus stop where a bus is expected to arrive
- the route (i.e. the line) of that bus
- the destination of that bus
- the direction of that bus
- when that bus is expected to arrive at that bus stop

The data feed is refreshed every 30 seconds. By submitting an HTTP GET request to a specific endpoint, we can retrieve bus arrival information for the next 30 minutes in CSV format.

In this case, we decided to map each bus stop to a sensor that observes a single composite property: the line, the direction and the arrival time of each bus arrival that involves it.

We implemented PPI as part of a Java web application that we deployed in **Wildfly** application server. In order to feed the application with data, we manually stored information about bus stops (provided also by TfL), and we also set up a task to fetch data from the feed every 10 minutes. We used **Elasticsearch** to store all this data.

Finally, we implemented and exposed as RESTful web services (using RESTEasy) only the mandatory PPI primitives, namely:

- Get IoT system metadata
- Get sensor metadata
- Get sensor measurements

Table 32 summarizes the technologies, libraries, and tools used in the implementation of PPI for TfL's live bus arrival data feed.

Table 29: Technologies used in live bus arrival data feed.

Framework/Library	Version
Elasticsearch	1.5.2
Java Platform, Enterprise Edition	8
jsonld-java	0.6.0-SNAPSHOT
Log4j	2.2
Maven	3.3.3
RESTEasy	3.0.9.Final
WildFly	9.0.0.CR1

6.2 PPI Implementation for X-GSN

X-GSN (**eXtended Global Sensor Networks**) constitutes an enhanced version of the GSN platform that has been developed as part of the OpenIoT project. The main enhancements introduced by X-GSN are:

- the semantic annotation of both sensors and measurements
- the ability to connect to a data management service, like the **Linked Sensor Middleware** (**LSM**) component of OpenIoT, that will store both data and metadata into the Cloud

Since both X-GSN and VITAL use the **Semantic Sensor Network** (**SSN**) ontology to semantically annotate sensors and the observations collected by them, there was no need for any mapping between X-GSN and VITAL entities.

The PPI implementation for X-GSN is a Java web application that we deployed in **Wildfly** application server. In this first prototype, we implemented only the services mandated by the PPI specification, namely:

- Get IoT system metadata
- Get sensor metadata
- Get sensor measurements

The implementation of the last two PPI services involved only the execution of the appropriate SPARQL queries to the **Virtuoso** instance where LSM stores sensor data and metadata. In these cases, the X-GSN PPI implementation acts in effect as an LSM client.

Table 30 lists the technologies, libraries, and tools used in the implementation of PPI for the X-GSN platform.

Table 30: Technologies used in X-GSN PPI implementation.

Framework/Library	Version
OpenLink Virtuoso	7.2.0.1
Java Platform, Enterprise Edition	8
jsonld-java	0.6.0-SNAPSHOT
Log4j	2.2
Maven	3.3.3
RESTEasy	3.0.9.Final
WildFly	9.0.0.CR1

As implied above, the PPI implementation we provided for X-GSN assumes that X-GSN is connected to LSM for data management purposes.

6.3 PPI Implementation for Xively

Xively is an IoT **Platform as a Service (PaaS)** that enables its users to store their data, query them, and even receive notifications when certain conditions are met (e.g. when a new value has been added). Xively exposes information about its resources (e.g. feeds, datastreams, devices) through a **REST** API in XML, JSON and/or CSV formats.

We decided to map each feed (which in Xively is connected with a single device that is in turn connected with a single product) to a VITAL sensor that observes one property for each datastream that has been added to that feed. Metadata about a sensor are retrieved from the attributes of the corresponding product, device and feed, whereas metadata about a property are retrieved from the attributes of the corresponding datastream. As it is already obvious, datapoints in datastreams are mapped to sensor measurements in VITAL.

We included the PPI implementation for the Xively platform into a Java web application that we deployed in **Wildfly** application server. As part of this implementation, we implemented and exposed (through REST) the following PPI services:

- Get IoT system metadata

- Get sensor metadata
- Get sensor measurements
- Subscribe to observation stream
- Unsubscribe from observation stream

For the implementation of all PPI primitives we used the REST API exposed by Xively. Especially for implementing the push-based access to sensor measurements, we used triggers that Xively provides support for.

Table 31 summarizes the technologies, libraries, and tools used in the implementation of PPI for the Xively platform.

Table 31: Technologies used in Xively PPI implementation.

Framework/Library	Version
Java Platform, Enterprise Edition	8
jsonld-java	0.6.0-SNAPSHOT
Log4j	2.2
Maven	3.3.3
RESTEasy	3.0.9.Final
WildFly	9.0.0.CR1

6.4 PPI Implementation for FIT/IoT-LAB

6.4.1 Implementation Overview

FIT IoT-LAB[1] is a platform that provides a very large scale infrastructure facility suitable for testing small wireless sensor devices and heterogeneous communicating objects. FIT IoT-LAB testbeds are located in six different sites across France, which gives forward access to overall 2728 wireless sensor nodes.

Within the VITAL context, the FIT's PPI is a Java application subdivided in 3 logical layers, REST interface, Core and Sensors Communication.

[1] http://iot-lab.info/

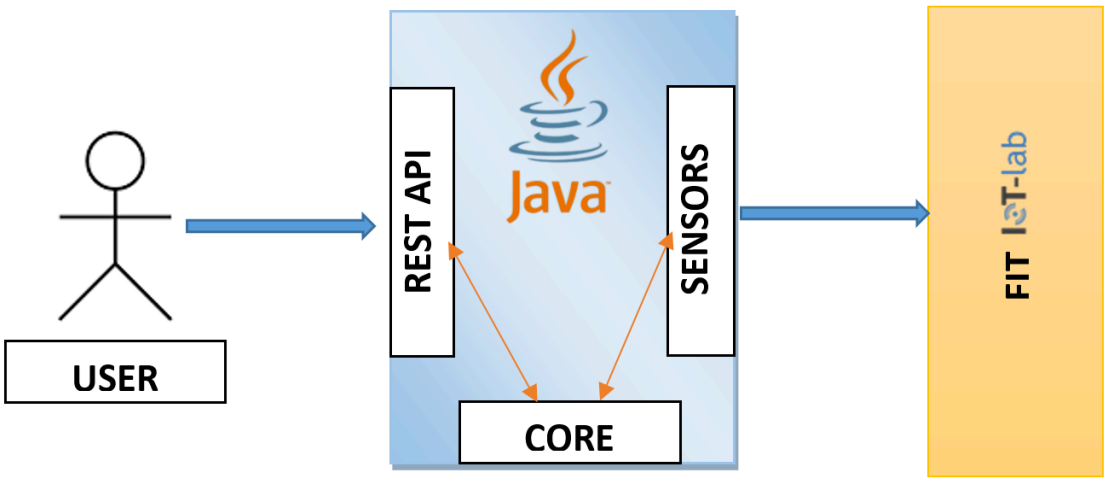


Figure 20: FIT IoT-Lab Representation

REST interface is used in order to allow clients (through a pull-based mechanism) to query the system. We implemented for the PPI the mandatory functions regarding:

- Get IoT system metadata
- Get sensor metadata
- Get sensor measurements

The results for those requests are returned in JSON-LD format, compliant to the format considered by all the PPI data representations.

To the follow we provide a practical example to clarify step sequences involved.

6.4.2 Examples

One possible example of interaction between the FIT's PPI and a client could be the request of a list of ICOs registered in the platform.

In this case, the client sends a request :

1. Wildfly intercepts the request.
2. The method associated to the request is invoked and invokes the *Core.*
3. The *Core* component analyzes the request and forwards a command to the *Sensor Communication Layer* (SCL).
4. The SCL communicates with FIT platform in order to retrieve the requested parameters.
5. The result is returned back to the *Core* Layer which applies the conversion to JSON-LD format.
6. The result is then returned back to the client that performed the request.

6.5 PPI Implementation for Hi Reply

6.5.1 Implementation Overview

The PPI implementation for the Hi Reply platform is a Java application, whose architecture is shown in Figure 21.

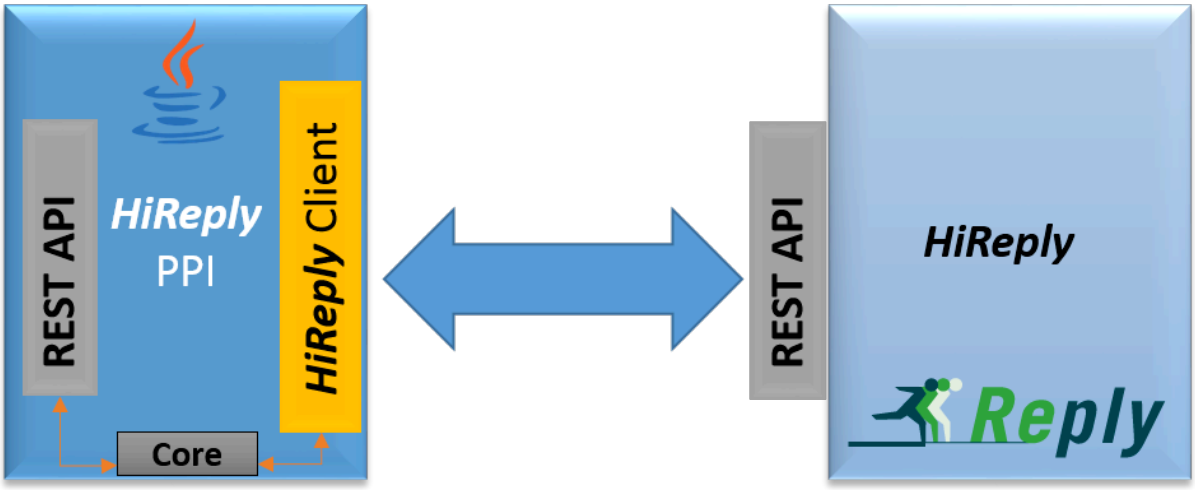


Figure 21: PPI implementation for the HI Reply platform.

Hi Reply exposes REST endpoints that allows clients to query information about:

- data (e.g. sensor measurements)
- the Hi Reply system (e.g. metrics)

The responses returned by Hi Reply are in XML format. All data are retrieved by a pull-based mechanism.

Table 32 lists the libraries and technologies used for the Hi Reply PPI implementation.

Table 32: Technologies used in Hi Reply PPI implementation.

Framework/Library	Version
Grizzly HTTP Server	2.3.16
Jackson	2.0
Java Platform, Enterprise Edition	7
JAXB2	1.6
Jaxen	1.1.6
JAX-RS	2.3.2
jsonschema2pojo	0.4.5
Log4j	2.0.2
Maven	3.3.1

Hereafter, Hi Reply PPI components are described.

6.5.1.1 Hi Reply Client

The **HiReply Client** component implements methods to query the Hi Reply platform by issuing HTTP GET requests to the defined endpoint where Hi Reply is running.

Every XML response is mapped to a Java POJO class thanks to JAXB and Jaxen libraries. POJO classes are generated with the JAXB Maven plugin; this plugin, starting from an XSD file, that defines the schema of Hi Reply XML models, generates a Java class that maps the XML structure defined in the XSD file.

Thus, the Hi Reply Client wraps all the requests and responses to/from Hi Reply. The implemented methods in this client are used by the upper level Core component to prepare data to be exposed via PPI.

6.5.1.2 Core & Rest API

The **Core** component implements all the methods mandated by the PPI specification. Here, all arrived requests are analyzed, and then the Hi Reply Client is in charge to retrieve the requested information.

The Core component uses JAX-RS, a Java API that provides support for creating web services according to the **REpresentational State Transfer** (**REST**) architectural pattern. Using annotations, methods defined in a Java class can be exposed at the defined resource path.

PPI uses JSON-LD to represent data. In order to map information retrieved from Hi Reply in JSON-LD format, two libraries have been used:

- **Jackson**: Serializes a POJO to a JSON string or deserializes a JSON string to the POJO that this JSON string represents.
- **jsonschema2pojo**: A Maven plugin that generates Java classes starting from JSON schemas.

The Core component uses these libraries to parse each request, execute the proper Hi Reply Client operation, and return the correct response.

Grizzly has been used to expose the JAX-RS resources on the specified host and port. Grizzly searches for JAX-RS resources in the package that is specified at the instantiation time.

In order to retrieve information about performance metrics, a class implementing Jersey's RequestEventListener interface has been created in the Core component. This class is notified whenever request-related events occur and tracks the entire lifecycle of requests. In particular, every time a resource is requested (i.e. an event occurs), this class intercepts the event state, which can be:

- **FINISHED**: The request and response processing has finished.
- **MATCHING_START**: The matching of the resource and resource method has started.
- **ON_EXCEPTION**: Exception has been thrown during the request/response processing.

- **REQUEST_MATCHED**: The matching has finished and container request filters are going to be executed.
- **RESOURCE_METHOD_FINISHED**: Resource method execution has finished.
- **RESOURCE_METHOD_START**: Resource method is going to be executed.
- **START**: The request processing has started.

Using the event state information captured by the event handler, performance metrics can be calculated at PPI level. In particular the following metrics are computed:

- Active requests
- Errors
- Served requests

6.5.2 Examples

This section provides an example of the execution flow of a request to the PPI.

Suppose that a client sends an HTTP requests to http://*www.hireply.com/iot/hireply/perf/upTime*.

1. Jersey intercepts the request.
2. The method associated to the request is invoked.
    a. The Core component invokes the Hi Reply Client that in turn issues to the Hi Reply platform the request to retrieve the timestamp when the platform started running.
    b. The Hi Reply Client returns an object that maps the XML response.
    c. The timestamp field is retrieved from the object returned by the client.
    d. Timespan (in ms) between the start time and the request time is calculated.
    e. A POJO that will map to the expected JSON-LD response (defined in3.2.6.1) is then instantiated and filled with the correct data.
    f. Using the Jackson library, the created POJO is serialized to a JSON-LD string.
    g. The string is returned by the method.

In the meantime, the event handler registers the event lifecycle. After the START state, the event is considered as a “pending request” as long as it has not reached the FINISHED state. After reaching the FINISHED state, the event is pulled from the “pending request” queue and the counter of the served requests is incremented. If the event throws an exception during his execution, the total error number is incremented.

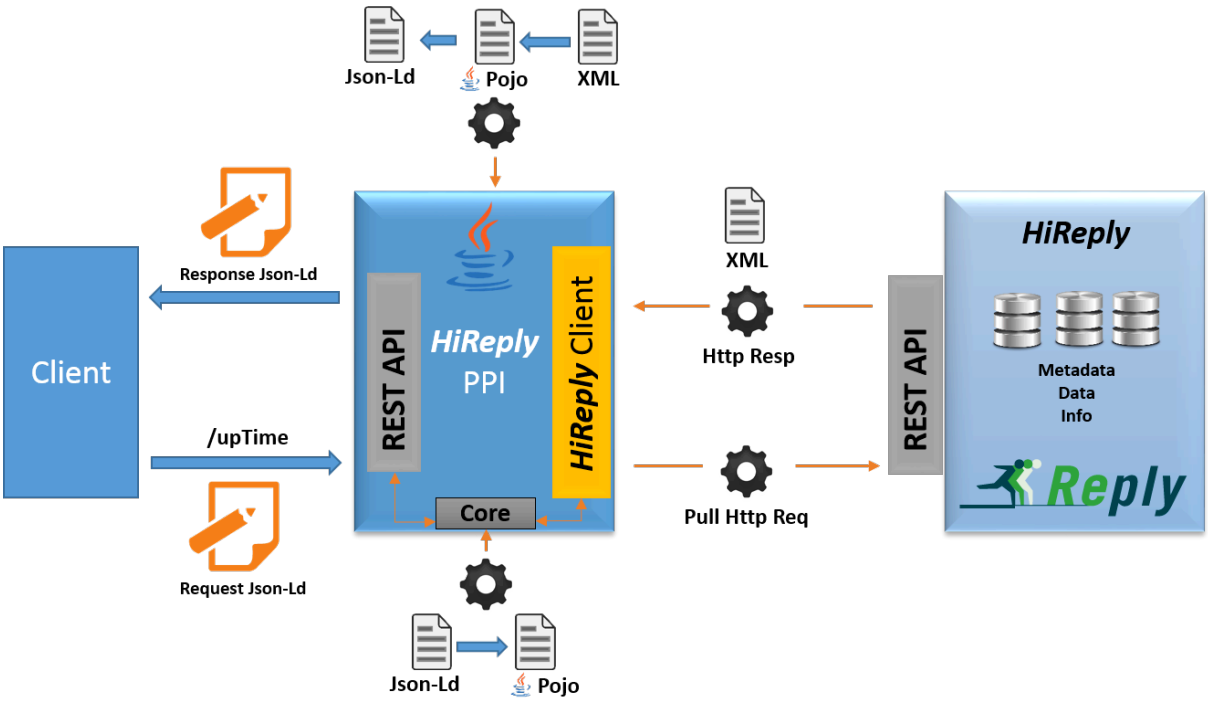


Figure 22: Example flow

7 CONCLUSIONS AND OUTLOOK TOWARDS FINAL VERSION

VITAL relies on a number of abstract virtualized interfaces to access (IoT-based) data streams and services in smart cities regardless of the IoT system (platform, data sources) that enables these functionalities. Earlier sections have provided an initial specification of VITAL's abstract interfaces for accessing IoT data and services in a platform-agnostic way. These interfaces have been classified in three categories: namely 1) interfaces to platforms, 2) interfaces to added-value functionalities of the VITAL platform and 3) interfaces to data processing functions. For each of the categories we have presented the interfaces. In the case of platform and added-value functionality access, the interfaces have been described at a very fine level of detail, i.e. down-to-implementation detail. This is not the case with the specification of the data processing functions, which is still in less mature state, both in terms of completeness and in terms of detail.

In parallel with the specification of the above-listed abstract interfaces, the partners have undertaken a significant step towards the realization of the interfaces. This is for example the case with the implementation of the PPIs over the four IoT platforms selected (in WP2): PPI implementations have been already realized (over X-GSN, FIT, Hi REPLY and Xively.com) and they are under testing and validation. Moreover, the implementation of the interfaces for the added-value functionalities is also in progress, as part of the implementation of the respective modules in WP4 and WP5 of the project. As expected, the implementation of the data processing interfaces is still in its infancy, given the need to finalize the selection of the open-source tools that will empower the realization of the VITAL statistical and BigData processing functionalities.

The present deliverable represents the first version/release of the VITAL abstract interfaces. A second and (final) release is planned six months later according to the project's work plan. This final release will report on updates to the specification of VITAL VUAIs, while also comprising their final implementation. In particular, the final release of the deliverable will be incremental to the present one and it will (additionally) contain:

- Updates to and fine-tuning of the PPI specifications, including any revisions that will be required in order to be in-line with the final version of the VITAL ontology.
- Updates to and fine-tuning of the interfaces to added-value functionalities, based on feedback from the actual implementation of these functionalities in WP4 and WP5 of the project.
- A more thorough (down to implementation detail) specification of the abstract interfaces to data processing, statistical processing and BigData functionalities of the VITAL platform.
- The implementation of the various abstract interfaces. Note that the deliverable will focus on the final implementation of the PPIs, given that the implementation of the rest of the interfaces (e.g. interfaces to service discovery, interfaces to management functionalities, interfaces to data mining etc.) will be implemented as part of the work packages where the respective modules are being implemented.

Overall, we envisaged that the conclusion of the present deliverable will form a sound basis for VITAL solution developers to access diverse platforms and data sources in a well-defined, versatile and effective way.

8 REFERENCES

[Di Ciaccio12] Agostino Di Ciaccio, Mauro Coli, Jose Miguel Angulo Ibanez Eds.) «Advanced Statistical Methods for the Analysis of Large Data-Sets», Series: Studies in Theoretical and Applied Statistics, Subseries: Selected Papers of the Statistical Societies 2012, XIII, 484p.

[Marz14] Nathan Marz, James Warren, «Big Data: Principles and Best Practices of Scalable Real-time Data Systems», Feb 2014, Paperback ISBN13: 9781617290343, ISBN10: 1617290343

[Sioshansi11] Fereidoon P. Sioshansi, «Smart Grid: Integrating Renewable, Distributed & Efficient Energy», November 2011, ISBN-10: 0123864526 | ISBN-13: 978-0123864529.

Vital 2015

