



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

D5.1.3 Management Services over Federated IoT Platforms V3

Document Id:	VITAL-D513-200416-Draft
File Name:	VITAL-D513-200416-Draft.pdf
Document reference:	Deliverable 5.1.3
Version:	Draft
Editor:	Lorenzo Bracco, Paola Dal Zovo
Organisation:	Reply
Date:	20/04/2016
Document type:	Deliverable
Security:	PU (Public)

Copyright © 2016 VITAL Consortium

PROPRIETARY RIGHTS STATEMENT

This document contains information, which is proprietary to the VITAL Consortium.
Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with prior written consent of the consortium

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V01	Lorenzo Bracco, Paola Dal Zovo	Reply	20/02/2016	Updated Document Structure based on D5.1.2
	Katerina Roukounaki	AIT	24/02/2016	Review
V02	Angelos Lenis	SiLo	25/03/2016	Updated Sections 3.2.4 -3.2.5, 4.1, 5.2-5.5
V03	Lorenzo Bracco, Paola Dal Zovo	Reply	05/04/2016	Updates overall the document plus Migration Toolkit in sections 4.2.1 and 5.9
V04	Miguel Angel Mateo	Atos	06/04/2016	Updated Sections 3.2.4.3, 5.6
V05	Lorenzo Bracco, Paola Dal Zovo	Reply	05/04/2016	Updates overall the document plus Migration Toolkit in sections 4.2.1 and 5.9
V06	Lorenzo Bracco, Paola Dal Zovo	Reply	06/04/2016	Updates to security management
V07	Lorenzo Bracco, Paola Dal Zovo	Reply	07/04/2016	Update section 5.9 with connections with management platform plus miscellaneous updates
V08	Lorenzo Bracco, Paola Dal Zovo	Reply	07/04/2016	Update section 6
V09	Riccardo Petrolo	Inria	07/04/2016	Update section 3.4.1.2; add section 5.9.2
V10	Miguel Angel Mateo	Atos	11/04/2016	Updated section 4.1.3
V11	Angelos Lenis	SiLo	11/04/2016	Updated sections 4.1.1, 4.2, 5.3 – 5.5, 6
V12	Lorenzo Bracco, Paola Dal Zovo	Reply	11/04/2016	Miscellaneous small updates
V13	Riccardo Petrolo	Inria	12/04/2016	Technical review
V14	Lorenzo Bracco, Paola Dal Zovo	Reply	14/04/2016	Address review comments
V15	Lorenzo Bracco, Paola Dal Zovo	Reply	19/04/2016	Ready for submission
V16	Martin Serrano	NUIG	20/04/2016	Circulated for Approval
Draft	Martin Serrano	NUIG	20/04/2016	EC Submitted

CHANGES & UPDATES FROM D5.1.2

New or Updated Section	Changes & Comments
Section 1 – Introduction	Updated in order to reflect enhancements over D5.1.2; a description of the updated functionalities is included
Sections 4.2.1 and 5.9	Described technical approach and implementation of the Reengineering and Migration Toolkit (Task T5.5)
Section 3.2.4-3.2.5	Updated the association of modules with management, in regards with the last version
Section 4.1, 5.1-5.8	Updated to reflect changes in the last period. Mainly reported the functionalities of the final version of the platform
Section 5 – Prototype Implementation (V3)	Enhancements reflecting the full range of updated functionalities implemented as part of the third release
Sections 3.1, 3.2	Updated APIs and data structure based on changes of interacting components
Section 2.3	Updated VITAL architecture figure
Section 4.1.5 and 5.7	Updated specification and implementation of security management.
Sections 5.1, 5.2	Described additional features of this version, as evolved from the first prototype.
Sections 5.3, 5.4, 5.5, 5.6, 5.7, 5.8	Updated sections to describe in detail how topology, monitoring and configuration are implemented and displayed
Sections 5.6, 5.8	Updated sections with new feature for Event management, SLA is no more part of this deliverable
Section 6	Updated conclusions
Section 4.1.3	Updated to new interface of CEP for alerting
Section 4.1.1	Added a third method for Resource Discovery

TABLE OF CONTENTS

DOCUMENT HISTORY	1
CHANGES & UPDATES FROM D5.1.2	2
 1 INTRODUCTION	 7
1.1 SCOPE	7
1.2 AUDIENCE.....	8
1.3 SUMMARY AND STRUCTURE	8
 2 INCEPTION AND SCOPING OF VITAL MANAGEMENT LAYER	 10
2.1 MANAGEMENT LAYER FOR SMART CITIES	10
2.2 TYPES OF VITAL MANAGED OBJECTS	12
2.3 POSITIONING IN THE VITAL ARCHITECTURE	13
2.4 DESIGN AND IMPLEMENTATION APPROACH.....	14
 3 SPECIFICATION OF MANAGED ENTITIES	 15
3.1 GENERIC MANAGEMENT MEASUREMENT	15
3.1.1 Monitoring Hooks	17
3.1.2 Configuration hooks	19
3.1.3 ICO management.....	21
3.2 MANAGED ENTITIES.....	21
3.2.1 IoT Systems	21
3.2.2 Services	23
3.2.3 Sensors / ICOs.....	24
3.2.4 VITAL Platform Modules	24
3.2.4.1 Service Discovery	25
3.2.4.2 VITAL Data Management Services	25
3.2.4.3 CEP	25
3.2.4.4 VITAL Orchestrator module	26
3.2.5 Service Level Agreements	26
3.2.6 City Specific Information	26
 4 FUNCTIONAL AND TECHNICAL SPECIFICATIONS.....	 27
4.1 MANAGEMENT FUNCTIONALITY SUPPORTED	27
4.1.1 Resource Discovery	27
4.1.2 Monitoring	27
4.1.3 Alerts / Events	28
4.1.4 Configuration.....	32
4.1.5 Security	33
4.2 TECHNICAL ARCHITECTURE AND REQUIREMENTS	33
4.2.1 Reengineering and Migration of existing IoT applications	35

5	PROTOTYPE IMPLEMENTATION (V3)	35
5.1	SUPPORTED FUNCTIONALITY.....	35
5.2	MIGRATION PLAN: FROM PROTOTYPE TO CONCRETE IMPLEMENTATION	36
5.3	DYNAMIC RESOURCE DISCOVERY	37
5.4	PERFORMANCE MONITORING	40
5.5	CONFIGURATION MANAGEMENT	43
5.6	ALERTING AND EVENT MANAGEMENT SERVICES	44
5.7	SECURITY MANAGEMENT FUNCTIONALITIES	45
5.7.1	Security management user interface	46
5.7.2	Security module REST API	52
5.8	SLA MONITORING & MANAGEMENT FUNCTIONALITIES	62
5.9	REENGINEERING AND MIGRATION TOOLKIT.....	62
5.9.1	Case study: Integration of IoT data sources	65
5.9.2	Case study: Integration of IoT gateway	66
5.9.3	Case study: Integration of smart cities services.....	66
6	OUTLOOK AND CONCLUSIONS	67
7	REFERENCES	68

LIST OF FIGURES

FIGURE 1: VITAL ARCHITECTURE.....	13
FIGURE 2: MANAGEMENT LAYER TECHNICAL ARCHITECTURE	33
FIGURE 3 MANAGEMENT UI: LIST OF SYSTEMS.....	39
FIGURE 4: MANAGEMENT UI: LIST OF SENSORS	39
FIGURE 5: MANAGEMENT UI: SYSTEM OVERVIEW – INFO AND STATUS.....	40
FIGURE 6: MANAGEMENT UI: SYSTEM OVERVIEW - PERFORMANCE GRAPHS.....	40
FIGURE 7: MANAGEMENT UI: GEOGRAPHICAL HEALTH-MAP	41
FIGURE 8: MANAGEMENT UI: GEOGRAPHICAL HEALTH-MAP – FOCUS ON ISTANBUL	41
FIGURE 9: MANAGEMENT UI: GEOGRAPHICAL HEALTH-MAP – FOCUS ON CAMDEN	42
FIGURE 10: MANAGEMENT UI: GEOGRAPHICAL HEALTH-MAP - SENSOR DETAILS.....	42
FIGURE 11: MANAGEMENT UI: DMS QUERY TOOL	43
FIGURE 12: MANAGEMENT UI: CONFIGURATION	44
FIGURE 13: ALARM MONITOR SYSTEMS LIST.....	44
FIGURE 14: UI SHOWING ALARMS REPORTED RAISED BY A CEP INSTANCE	45
FIGURE 15 - SECURITY MANAGEMENT ARCHITECTURE.....	45
FIGURE 17 – MANAGEMENT UI MENU INCLUDING SECURITY SECTION	47
FIGURE 18 – USERS LISTING	47
FIGURE 19 – USER CREATION	48
FIGURE 20 – USER EDITING.....	48
FIGURE 21 – GROUP CONTROL.....	49
FIGURE 22 – POLICY CREATION	50
FIGURE 23 – OPENAM APPLICATIONS.....	51
FIGURE 24 – MONITOR INTERFACE	51
FIGURE 25 – ACCESS CONTROL TEST PANEL	52
FIGURE 26 – STRUCTURE OF THE MAVEN ARCHETYPE	63
FIGURE 28 – IOT ADAPTER WEB INTEFACE	64
FIGURE 29 – PPIs LISTED ON THE MANAGEMENT APPLICATION	65
FIGURE 30 – EXAMPLE OF SENSORS SHOWN ON THE MAP FOR TURIN CITY	65

LIST OF TABLES

TABLE 1: JSON-LD SAMPLE FOR THE MANAGEMENT DATA.	15
TABLE 2: IOT SYSTEM SERVICES MANAGEMENT SECTION.	15
TABLE 3: GETSYSTEMSTATUS.	17
TABLE 4: GETSENSORSTATUS.	17
TABLE 5: GETSUPPORTEDPERFORMANCEMETRICS.....	18
TABLE 6: GETPERFORMANCEMETRICS.	18
TABLE 7: GETCONFIGURATION.	20
TABLE 8: SETCONFIGURATION.	20
TABLE 9: METADATA OF THE ALERTINGMANAGEMENTSERVICE	29
TABLE 10: GET ALARMMONITORS.....	29
TABLE 11: GET ALARMMONITOR INTERFACE	30
TABLE 12: CREATE OR UPDATE ALARMMONITOR.....	31
TABLE 13: DELETE ALARMMONITOR	32
TABLE 14: OPENAM ERROR CODES.....	61

TERMS AND ACRONYMS

FCAPS	Fault, Configuration, Accounting, Performance, Security
FIT	Future Internet of Things
GSN	Global Sensor Networks
GUI	Graphical User Interface
ICO	Internet-Connected-Object
IoT	Internet-of-Things
KPI	Key Performance Indicator
QoS	Quality of Service
SLA	Service Level Agreements
ICT	Information and Communication Technologies
CEP	Complex Event Processing
VUAI	Virtualized Unified Access Interfaces
API	Application Programming Interface
PPI	Platform Provider Interface
URI	Uniform Resource Identifier
DMS	Data Management Services
BPM	Business Process Model
UI	User Interface
UX/UI	User Experience/User Interface

1 INTRODUCTION

1.1 Scope

Modern cities are increasingly seeking ICT-driven ways to plan and monitor their sustainable interventions on the urban environment. Such interventions include the development, deployment and operation of IoT-based smart city services. The ability to plan, monitor and manage such IoT based services regardless of the technology platform that empowers them is a key enabler for structuring and organizing city-wide interventions. The VITAL project focuses on the development of a novel smart cities platform, which will enable the integration and semantic interoperability of multiple IoT systems that underpin smart city applications and services. One of the core characteristics of this platform (as specified in WP2 and deliverable D2.3) is its management layer, which aims at providing functionalities for managing in a unified way diverse IoT systems and services, thereby facilitating the city-wide planning and monitoring of relevant interventions. Despite the importance of such a management layer, most of the IoT platforms (including IoT platforms deployed in VITAL such as GSN, Hi Reply and FIT) do not provide relevant functionalities and therefore VITAL aspires to be a pioneer in this respect.

The present deliverable introduces the VITAL management layer, including the drivers behind its development and its main functionalities. In particular, the deliverable presents the VITAL management layer and its role within the VITAL architecture. It also illustrates the various entities and objects that will be managed (i.e. VITAL managed objects), along with the main management functionalities implemented over them. Furthermore, the deliverable illustrates an enhanced implementation of the management layer, which follows the second release presented as part of deliverable D5.1.2. The implementation includes a management application for monitoring and visualizing VITAL managed objects. This third and final version of the application includes several updates and enhancements when compared to the previous release including:

- Search of VUAls through (1) manual configuration (2) the IoT Data Adapter module and (3) the Service Discovery module, which supports more complex queries, are now fully integrated.
- Improved Monitoring services support, mainly through optimizations, no new features are required.
- A new VUAI for alerts management provided by VITAL CEP (Complex Event Processing).
- Security is now fully integrated with the Management application. Furthermore, it saw many major improvements: notably support for fine-grained data access control, better session management through secure cookies (both at the REST API and UI login interface level), addition of a UI section for administrators to test users permissions and much more enhancements all over the module.
- The SLA management module is now part of the Governance Toolkit and is no longer reported in this deliverable.

Despite this being the final version of the management application, it will continue to receive improvements where and when needed alongside the continuous development of the VITAL platform. The management application will be augmented with the Governance toolkit functionalities as to provide a complete management and governance platform.

This version of the deliverable also introduced the migration toolkit, which helps developers in the process of integrating existing services in the VITAL platform.

1.2 Audience

This deliverable is destined for researchers and engineers that are interested in the development of management applications for smart city services and more specifically on IoT based applications and services. It is also expected to be of great interest to development communities working on the OpenIoT, GSN and FIT platforms, given that it provides valuable add-on management functionalities, which are currently missing from these platforms.

The deliverable could be also of interest to operators/administrators of smart city services (including the city authorities), which are the natural end-users for the management services provided by the VITAL management layer in a smart cities context. The VITAL management platform is an independently exploitable result/solution of the project, since it can enable smart city authorities to monitor, configure and manage their infrastructures, including SLA, security and performance aspects.

Note also that the deliverable has also importance as an internal milestone document for the VITAL consortium, since it is closely linked to VITAL results in other (on-going and future) project activities, such as the activities relating to the VITAL applications integration (WP6), but also to activities associated with data/interfaces modelling and implementation (in WP3).

1.3 Summary and Structure

The third version of this deliverable starts with a discussion about the motivation and need behind the implementation of a management tool (and of an associated environment) as part of a smart cities platform. It also illustrates its positioning within the VITAL architecture, including the way it interacts with other modules of the VITAL platform. A significant part of the deliverable is devoted to the specification of the entities/objects to be managed, but also of the type of management operations (e.g., read/write) that are envisaged over them. As part of the deliverable, we also outline the implementation of the VITAL management environment, including an illustration of supported/implemented functionalities and some relevant screenshots.

The deliverable is structured as follows:

- Section 2, following this introductory section, outlines the drivers behind the specification and implementation of the VITAL management environment. It also outlines the positioning of the management environment within the overall VITAL architecture, along with the specified integration interfaces. Finally, this section explains the approaches selected for designing and implementing the management layer within the VITAL project.
- Section 3 provides a detailed list of the entities to be managed (and of their attributes). These entities include IoT systems, ICOs, security components, city-wide information and more.
- Section 4 presents the set of management functionalities that are supported and implemented by the VITAL management environment as part of the third release of the deliverable D5.1. Moreover, this section provides an overview of the technical architecture and design guidelines of the management layer.
- Section 5 presents the prototype implementation of the VITAL management environment, which accompanies this document as part of the third release of the deliverable. The description of this implementation is an extended version of the relevant functionalities description of the earlier release of the deliverable (i.e. D5.1.2). Details about new or updated functionalities are also provided.
- Section 6 is the concluding section of the deliverable. Apart from concluding remarks, it also provides additional improvements that may be implemented.

2 INCEPTION AND SCOPING OF VITAL MANAGEMENT LAYER

2.1 Management Layer for Smart Cities

An IoT platform is a distributed software system with multiple components and subsystems that typically are integrated with external services and sensor networks. In addition to the application software systems involved, an IoT deployment requires infrastructure software (e.g. application servers, databases) and hardware (servers, network equipment) as well as peripheral protocols, APIs, and software systems that provide sensor data streams and related business services. This architectural diversity and operational complexity is the main drive behind the introduction of a management layer that will enable the efficient monitoring of the platform & supported services status and operational readiness.

The need of a management plane/layer is common to all complex ICT systems, especially when the services offered are of a critical nature; however, IoT systems and their applications in the Smart City domain are not just complex systems that support critical functionality but they also provide processed data and reports that may affect large scale urban planning, policy making, regional/national-level strategic decisions or even legislation. Consequently, a Smart City IoT system requires a management layer that:

- enables the monitoring of operational readiness of the system as a whole and the supported services,
- provides early warning on faults and glitches that affect the quality of service offered to the Smart City actors, before the problem escalates into the complete disruption of service or even worse into producing false data,
- provides the FCAPS model functionality [Raman98] of the typical management system, focusing mostly on the IoT services provided and less on the low level systems information (but also be able to drill down into some more detailed management views depending on the specific scope),
- provides friendly and ergonomic user interfaces to the operator and administrators of the Smart City IoT system as well as e-services that can be used by higher level IT Governance Systems used by the city or the regional government,
- and seamlessly integrate with underlying ICT systems that cooperate with the IoT platform in order to minimize delays and communication glitches, thus achieving near real-time information.

The above are high-level requirements defined by considering the key goals of the Smart City IoT business in general and the best practices and state-of-the-art functional characteristics of the modern management systems (systems and network management, application management, and integrated services management). Although standards and available technologies would allow the management of an IoT system to drill down to excruciating detail throughout the technology stack (e.g. network, system, operating system, DB/app server, application), the management layer should focus on enabling the timely monitoring, and in some cases control, of the health of the overall platform as well as the quality of service delivered to the Smart City actors. The following is a list of functionalities and functional modules that we consider important and critical in this scope:

- Sub-system / component and overall health monitoring: provide a “health map” of the IoT platform with the ability to drill-down to a level that is practical for monitoring sub-systems and component operational status parameters.
- Operations management: provide the GUI tools for supporting operations management, i.e. basic monitoring and configuration of operational parameters, visual status reporting for individual components (in list or geographical views).
- Data/Information strategic planning input: maintain and process historical data in order to provide a data store of structured monitoring data of key operational information, QoS parameters, and service KPIs that can be retrieved via querying/exporting functionalities and to be used in strategic planning and similar activities by the city/regional authorities.
- Critical FCAPS operations: cover the complete span of the FCAPS functional model, but focus on Smart City & IoT/VITAL platform specific requirements:
 - Fault: health map, visual representation of fault events with drill down options, alerting
 - Configuration: geographical map, configuration parameter monitoring and control for platform components and underlying systems (depending on permissions)
 - Accounting: monitor and maintain accounting information, i.e. client application usage of platform resources, for enabling the implementation of fairness control policies, etc. SLA monitoring / management is an important aspect that is associated with this area.
 - Performance: monitor the performance of the platform and individual sub-systems. The goal is not only to monitor QoS but also to provide “early warning insight” on developing faulty conditions before they escalate and/or cascade to an actual fault.
 - Security: monitor operational parameters related to the security and access control of the platform.

The VITAL platform is a complex IoT system that consists of multiple loosely coupled subcomponents (via RESTful APIs), and that by design integrates with multiple and heterogeneous underlying IoT Platforms / Products. VITAL provides to its connected applications a common view of the data & services provided by these underlying systems, unified using a common semantic data model. In a similar fashion the VITAL management layer provides a common management plane that unifies all the management information of all components and systems as well as management functionality that can be performed in a unified way to all parts of the VITAL ecosystem.

More specifically, the VITAL management layer:

- Supports the management of the core VITAL platform and all individual VITAL sub-systems (i.e. software components that are designed and developed by the VITAL project).
- Provides a common, unified management view (monitoring and control if allowed) of the different underlying IoT Systems. It will integrate with any management

hooks provided by these systems and translate data and functionality into the common VITAL management model.

- Manages the relationships between VITAL components and subsystems. This is an extension of the monitoring (and control when possible) of the individual component/sub-system operational / health parameters to the monitoring of the interactions between them.

2.2 Types of VITAL Managed Objects

The VITAL management layer will handle the following types of objects (VITAL managed objects):

- ICOs and data streams: these are the actual “low-level” ICOs that are typically connected to the VITAL platform via an underlying IoT System. The management layer provides operational information of the ICO basic parameters (including position and trajectory for mobile ICOs) as well as the availability and QoS (if available by the IoT systems) parameters of the related data streams. This is information provided by the IoT Systems’ management hooks.
- IoT Systems: these are the underlying IoT Platforms and Applications that are connected to the VITAL platform via the VITAL adapters (PPIs). Depending on what each system supports, the management layer retrieves, processes and uses information on the health, operational parameters, performance, accounting, security and even specific alerts emanating from the connected IoT system.
- VITAL Platform Components: these are the individual loosely coupled sub-systems/components of the VITAL architecture, i.e. the CEP module, the Service Discovery module, the Data Management layer, the Orchestrator module, etc. The management layer monitors and handles health information as well as data related to all the aspects of the FCAPS model (wherever applicable) for each component.
- SLA: this refers to the SLAs set between the VITAL deployment and the connected Platform/Data Providers. The management layer will provide monitoring services as well as historical data on these SLAs as part of the Governance Module.
- Security information: this refers to all security related management data, i.e. data related to authenticated sessions and failed authentication or unauthorized access attempts by the applications operating on top of the VITAL platform. It could also extend to other security related information like authentication errors in the communication of VITAL with the underlying IoT Systems (if they support it).
- City-specific configurations: access to this information is realized via the VITAL Governance toolkit/module (T5.4). Although the T5.4 will produce specific software modules, the management layer will integrate T5.1 and T5.4 functionality via a common GUI.

2.3 Positioning in the VITAL Architecture

The following diagram (Figure 1) provides an overview of the VITAL architecture and outlines the relationships and basic communication between the various components.

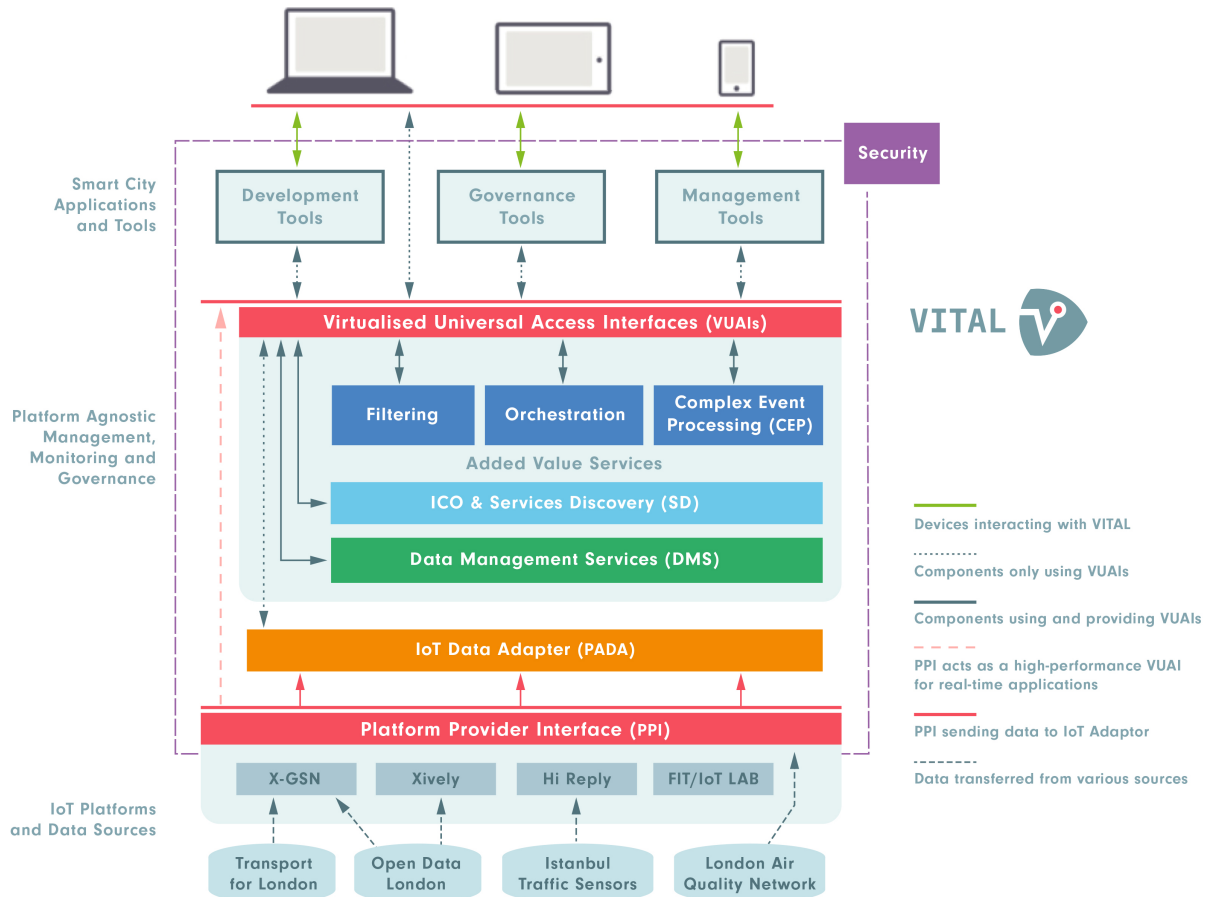


Figure 1: VITAL Architecture

The Management layer (shown as Management Tools), as well as the Governance set of tools, is a high level component in the sense that it interfaces with all other subsystems and components of the VITAL platform architecture. As shown in the diagram, the Management layer interface:

- Uses the VUALs for monitoring & retrieving information on the managed entities (or for pushing configuration changes), with a main focus on VITAL components and VITAL higher-level service components. The VUALs can be also used for retrieving information on ICOs/data sources and underlying IoT Systems.
- Uses directly the PPIs via the VITAL Adapters as an alternative for accessing ICO/IoT System management functionality in use cases where performance is critical (i.e. avoid the delay of going through all the VITAL layers for a configuration change).

In both cases, the Management Layer accesses the RESTful interfaces and the VITAL ontologies and data models, conforming to the JSON-LD syntax defined by VITAL.

In the current version of the design, the Management Layer does not publish services for consumption by the applications built on top of the VITAL platform. It does provide:

- A GUI application for the platform operator / administrator.
- A Restful API that can be used by the platform administrators for integration with Network & System Management Systems.

The following VITAL components interact with the Management Layer as follows:

- Discoverer: provides service directory information (e.g. for the healthmap as well as meta-data for accessing VITAL services) as well as management hooks for monitoring its health.
- Data Services: provides access to the semantically unified structured data and underlying services.
- Adapters/PPIs: provide management hooks to underlying IoT systems (and their services) and ICOs/data streams.
- All VITAL components: provide management hooks for monitoring and configuration (if allowed by each component).

2.4 Design and Implementation Approach

An incremental top-down approach has been adopted for the implementation of the VITAL management environment. This approach is based on the segmentation of the environment into a number of middleware modules and dashboards, which are dedicated to the management of specific smart city entities (such as ICOs, smart cities platforms, smart city applications and more). Following this segmentation, the management environment has gradually incorporated middleware modules that enabled the management of different types of managed entities sets. Also, the management environment offers a range of visualization modules (such as mashups), which are conveniently used in order to present/visualize the values that are associated with managed entities, as well as their evolution (e.g., in time or space). The overall approach is incremental since more bundles of managed entities have been gradually incorporated into the VITAL management environment. While the first release illustrated a (mock-up) prototype implementation of the management environment, the latter two releases of this deliverable elaborated on add-on functionalities associated with the management of additional sets of managed entities.

In addition to incremental, the approach taken towards implementing the prototypes accompanying this deliverable is a top-down approach. It starts with the establishment of a general environment for managing IoT resources, data, platforms and applications for smart cities, which has been gradually refined, detailed and customized to the needs of specific sets/bundles of managed entities. The general environment includes also (re)useable visualization elements for several (rather than a few) middleware modules of the VITAL platform. The following section provides a specification of the entities to be managed.

3 SPECIFICATION OF MANAGED ENTITIES

3.1 Generic Management Measurement

In this section, we illustrate the format for reading management data, e.g. performance throughput, error rate, component load. This is information retrieved by the management layer from the VITAL management instrumentation (modules, IoT subsystems). These are modelled as sensor observations, following the same @context but with their own property types (vital:Errors for example).

For better readability, the 'vital:' prefix is used instead of the full namespace 'http://vital-iot.eu/ontology/ns/'; for example vital:MonitoringService refers to <http://vital-iot.eu/ontology/ns/MonitoringService>.

Table 1: JSON-LD Sample for the management data.

```
[
  {
    "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
    "id": "http://example.iot.system/sensor/monitoring/observation/95",
    "type": "ssn:Observation",
    "ssn:observationProperty": {
      "type": "vital:Errors"
    },
    "ssn:observationResultTime": {
      "time:inXSDDateTime": "2015-06-14T19:37:22Z"
    },
    "ssn:featureOfInterest": " http://example.iot.system",
    "ssn:observationResult": {
      "type": "ssn:SensorOutput",
      "ssn:hasValue": {
        "type": "ssn:ObservationValue",
        "value": 1.0,
        "qudt:unit": "qudt:Number"
      }
    }
  }
]
```

The Management and Governance Layer requires the addition of a new section to the IoT ontology, which is specific to management. This section provides information on the VITAL management instrumentation (service for monitoring and configuration) supported by the specific IoT System connected to the VITAL platform deployment. The structure is in Table2.

Table 2: IoT System services management section.

```
[{
```



```

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

```

The management part of any VITAL system (PPI, CEP, Orchestrator, etc.) that supports it, is composed of two types of services, `vital:MonitoringService` that is responsible for measuring and reporting status and performance related information and `vital:ConfigurationService` that allows changing the configuration of the system.

3.1.1 Monitoring Hooks

The Management & Governance layer utilizes the individual operations of the monitoring services to retrieve status updates and performance metrics. The data format of each operation is displayed in the following tables.

Table 3: GetSystemStatus.

	vital:GetSystemStatus	
Description	Retrieves the current status of the system.	
Method	POST	
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.iot.system/sensor/monitoring/observation/1435652134143", "type": "ssn:Observation", "ssn:observationProperty": { "type": "vital:OperationalState" }, "ssn:observationResultTime": { "time:inXSDDateTime": "2015-06-30T08:15:34Z" }, "ssn:featureOfInterest": "http://example.iot.system", "ssn:observationResult": { "type": "ssn:SensorOutput", "ssn:hasValue": { "type": "ssn:ObservationValue", "value": "vital:Unavailable" } } }</pre>	
Notes	<ul style="list-style-type: none"> The response contains the current status of the system. 	

Table 4: GetSensorStatus.

	vital:GetSensorStatus	
Description	Retrieves the current status of the sensor.	
Method	POST	
Request headers	Content-Type	application/json
Request body	Example <pre>{ "id": ["http://example.iot.system/sensor/5-F"] }</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.iot.system/sensor/monitoring/observation/status/14356</pre>	

	<pre> 52134143", "type": "ssn:Observation", "ssn:observedBy": "http://example.iot.system/sensor/monitoring", "ssn:observationProperty": { "type": "vital:OperationalState" }, "ssn:observationResultTime": { "time:inXSDDateTime": "2015-06-30T08:15:34Z" }, "ssn:featureOfInterest": "http://example.iot.system", "ssn:observationResult": { "type": "ssn:SensorOutput", "ssn:hasValue": { "type": "ssn:ObservationValue", "value": "vital:Unavailable" } } } } </pre>
Notes	<ul style="list-style-type: none"> 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 response contains the current status of a specific sensor of a system.

Table 5: GetSupportedPerformanceMetrics.

	vital:GetSupportedPerformanceMetrics	
Description	VITAL pulls from an IoT system metadata about the supported management metrics.	
Method	GET	
Request headers	Content-Type	application/json
Response headers	Content-Type	application/ld+json OR application/json
Response body	Example <pre> { "metrics": [{ "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" }] } </pre>	
Notes	<ul style="list-style-type: none"> The response contains the hooks for retrieving the monitoring parameters supported by the specific IoT system. Obviously, these parameters are optional and use case specific. The management module is designed to handle missing data (e.g. if the IoT system does not provide a maxRequests value, utilization will not be calculated). 	

Table 6: GetPerformanceMetrics.

	vital:GetPerformanceMetrics	
Description	VITAL pulls from an IoT system actual performance metrics.	
Method	POST	
Request headers		
Request body	Example <pre>{ "metric": ["http://vital-iot.eu/ontology/ns/Errors", "http://vital-iot.eu/ontology/ns/ServedRequests"] }</pre>	
Response headers	Content-Type	application/json
Response body	Example <pre>[{ "@context": "http://vital-iot.eu/contexts/measurement.jsonld", "id": "http://example.iot.system/sensor/monitoring/observation/95", "type": "ssn:Observation", "ssn:observationProperty": { "type": "vital:Errors" }, "ssn:observationResultTime": { "time:inXSDDateTime": "2015-06-14T19:37:22Z" }, "ssn:featureOfInterest": " http://example.iot.system", "ssn:observationResult": { "type": "ssn:SensorOutput", "ssn:hasValue": { "type": "ssn:ObservationValue", "value": 1.0, "qudt:unit": "qudt:Number" } } }, { "@context": "http://vital-iot.eu/contexts/measurement.jsonld", "id": "http://example.iot.system/sensor/monitoring/observation/96", "type": "ssn:Observation", "ssn:observationProperty": { "type": "vital:servedRequests" }, "ssn:observationResultTime": { "time:inXSDDateTime": "2015-06-14T19:37:22Z" }, "ssn:featureOfInterest": " http://example.iot.system", "ssn:observationResult": { "type": "ssn:SensorOutput", "ssn:hasValue": { "type": "ssn:ObservationValue", "value": 5.0, "qudt:unit": "qudt:Number" } } }]</pre>	
Notes	<ul style="list-style-type: none"> The response contains the observation values for the system and for the types specified in the request. 	

3.1.2 Configuration hooks

The Management & Governance layer will use the GetConfiguration to retrieve the exposed configuration parameters of the system. SetConfiguration operation is then

used to set/update one or more configuration parameters with “rw” permission. The IoT system may or may not support the operation (specified as in Table 2).

Table 7: GetConfiguration.

	Get configuration	
Description	VITAL pulls from an IoT system metadata about the supported configuration options.	
Method	GET	
Response headers	Content-Type	application/json
Response body	Example <pre>{ "parameters": [{ "name": "configuration_parameter_1", "value": "some value that will not change", "type": "http://www.w3.org/2001/XMLSchema#string", "permissions": "r" }, { "name": "configuration_parameter_2", "value": "value 2", "type": "http://www.w3.org/2001/XMLSchema#string", "permissions": "rw" }, { "name": "configuration_parameter_3", "value": "value 34", "type": "http://www.w3.org/2001/XMLSchema#string", "permissions": "rw" }, { "name": "configuration_parameter_4", "value": "some value that will not change", "type": "http://www.w3.org/2001/XMLSchema#string", "permissions": "r" }, { "name": "configuration_parameter_5", "value": "value 5", "type": "http://www.w3.org/2001/XMLSchema#string", "permissions": "rw" }]}</pre>	
Notes	<ul style="list-style-type: none"> The response contains the key-value pairs of all configuration parameters exposed to VITAL, along with permissions supported via the PPI interface (rw, r). The VITAL Management & Governance module is agnostic of the keys, but it uses the type parameter for visually rendering these values and providing an editing widget (for parameters with “rw” permission). 	

Table 8: SetConfiguration.

	Set configuration	
Description	VITAL sends new values for configuration options.	
Method	POST	
Request headers	Content-Type	application/json
Request body	Example <pre>{ "parameters": [{ "name": "configuration_parameter_2", "value": "value 2" }, { "name": "configuration_parameter_3", "value": "value 34" }, {</pre>	

	<pre> "name": "configuration_parameter_5", "value": "value 5" }] } </pre>
Response body	200 OK
Notes	<ul style="list-style-type: none"> The status code of the response can be: <ul style="list-style-type: none"> 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 <p>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.</p>

3.1.3 ICO management

The management of ICOs is supported by the existing design without extra ontologies and services, provided that the Management and Governance layer is interested solely on the status of an ICO and its location data (if supported by the ICO). Configuration options will be supported via the Get/Set Configuration operations provided by the IoT System, as described above.

3.2 Managed Entities

This section identifies the different types of managed entities handled by the VITAL Management Layer.

3.2.1 IoT Systems

IoT Systems are managed through their corresponding PPIs that expose a VUAI interface. VITAL modules that expose a VUAI interface are also considered as IoT Systems by the management layer.

Systems provide metadata information not only for describing their capabilities and setup, but also for measuring performance. The management platform utilizes the subset of the available information that is required for managing the health of the system.

The following is a list of generic parameters of the vital ontology that the management system utilizes:

Property	Type	Description
id	Static	A unique identifier to differentiate entities
type	Static	The type of the entity, for systems this is vital:IoTSystem
name	Static	A name of the system, for displaying on screen
description	Static	A description of the system, for displaying on screen
status	Dynamic	The status of the entity. This can contain info like 'vital:Running', 'vital:Unavailable', or any values that will be added in the

		ontology as required by the systems. In the monitoring screen the status will be the first indicator that the system is working properly or not.
operator	Static	The system's operator's contact information
serviceArea	Static	Describes the area that the system covers, for example a geographic polygon (Basic Geo (WGS84) ontology)
providesService	Dynamic	A list of ids of the available services offered by the system. More specific information for the services are obtained through an extra call to the system, described in the next section
providesSystem	Static	A list of underlying systems, if this IoT system is a wrapper for other systems in a hierarchical topology
hasNetworkConnection	Static	Information on the underlying network infrastructure (wired/wireless, bandwidth etc.)

Below, is a list of performance measurements that should be measured by the IoT system and exposed to the management platform via the vital:MonitoringService of the system. This list is not complete and not all metrics are mandatory. Any system developer should review these and implement the metrics that fit their systems, expanding the list as necessary.

Property	Type	Description
SysUptime	System Metric	The time in msec since the last restart
SysLoad	System Metric	A number indicating the load of the system (similar to the unix load http://en.wikipedia.org/wiki/Load_%28computing%29).
UsedMem	System Metric	The memory in bytes used by the system
AvailableMem	System Metric	The memory in bytes available to the system
ServedRequests	Performance Metric	The total number of requests served by this system. This is an always-increasing number (can be reset when restarting). Requests can be http requests for operation on the system (eg. data retrieval)
PendingRequests	Performance Metric	The number of pending requests at the time of the measurement

MaxRequests	Performance Metric	The number of requests per a period of time (seconds, minutes) the entity can serve simultaneously (e.g. 10 requests/sec). It indicates the maximum operation capabilities of the service.
Errors	Performance Metric	The total number of errors occurred from this entity. This is always an increasing number (can be reset when restarting). Errors can be of any type, they may come from the IoT system or be triggered from inside the PPI; in general any time a request cannot be satisfied this counter is increased.

With the parameters above we can calculate higher-level performance metrics like:

- $\text{throughput} = (\text{request}(t_2) - \text{request}(t_1)) / (t_2 - t_1)$
- $\text{utilization} = \text{throughput} / \text{maxRequests}$
- $\text{error percentage} = \text{error} / \text{requests}$
- $\text{error rate} = (\text{error}(t_2) - \text{error}(t_1)) / (t_2 - t_1)$

Another option would be to integrate these higher-level metrics directly into the system and let the IoT system provide them. Either approach is viable.

3.2.2 Services

Services are provided by systems in the VITAL architecture. In the management platform the list of provided services, as exposed in the metadata information of each system, provide the necessary information to deduct whether the system is manageable (i.e. does it measure and provide monitoring metrics, can it be configured) and how to actually perform management operations.

In addition it displays the extra functionalities of the system, useful for displaying in the topology of Vital. With this we can, for example, filter systems that support the vital:ObservationService.

Property	Type	Description
id	Static	A unique identifier to differentiate entities
type	Static	The type of the service. Important services for the management platform are described in section 3.1
name	Static	The name of the ICO for displaying on screen
description	Static	A description of the system for displaying on screen
operations	Static	A list of operations this service supports. With this information any external application can find the proper way to actually use the service (URL, HTTP Method)

3.2.3 Sensors / ICOs

For a comprehensive list of both the topology and the current state of the overall VITAL deployment, the management system needs information on the specific ICOs / Sensors.

The following is a list of the metadata parameters that the management system utilizes for sensors:

Property	Type	Description
id	Static	A unique identifier to differentiate entities
name	Static	The name of the ICO for displaying on screen
type	Static	The type of the sensor
description	Static	A description of the system for displaying on screen
status	Dynamic	The status of the ICO. This can contain info like 'active', 'disabled', 'malfunctioning', 'low battery', 'network error' etc. In the monitoring screen the status will be the first indicator that the system is working properly or not.
ssn:observes	Static	A list of measurements this ICO performs (e.g. temperature, average traffic speed etc.)
hasLastKnownLocation	Dynamic	The latest geographic location of the ICO on the map. This is useful to display the HealthMap of the ICOs on a geographic map.
hasMovementPattern	Dynamic	Provides information on both the type of the movement pattern (Stationary Mobile Predicted) and some dynamic information on current direction and speed. Useful to track the location of the ICO on the map.
hasNetworkConnection	Static	Information on the underlying network infrastructure (wired/wireless, bandwidth etc.)

ICOs are not associated with performance metrics, since access to ICOs is via IoT/Systems and PPIs that are already monitored and managed.

3.2.4 VITAL Platform Modules

VITAL Platform modules are monitored and managed in the same manner as IoT Systems through their VUAls. All parameters mentioned in 3.2.1 are also applicable for VITAL Modules. However, each module may have additional requirements (or maybe less) and modification to the exposed management information may be necessary. These extensions are described in the following paragraphs.

In addition, some modules provide required functionality to the management platform. For example, the Service Discovery module provides information on the topology of the VITAL deployment (list of IoT Systems, ICOs and VITAL modules) and the Data Management module provides access to measurements without connecting directly to PPIs.

3.2.4.1 Service Discovery

The Service Discovery module provides the means for discovering ICOs, IoT services, and IoT data that are virtualized in the VITAL platform.

The other modules of the architecture (e.g. CEP, Orchestrator) will access the Service Discovery through RESTful interfaces in order to operate on the IoT resources that are needed for their particular business context.

3.2.4.2 VITAL Data Management Services

The VITAL Data Management Services (DMS) are defined as RESTful web services. These services are used to get data from sensors, systems or services. Data could be metadata as well as measurements.

The management platform uses DMS indirectly through Service Discovery to retrieve metadata, and directly to access historical performance metrics.

3.2.4.3 CEP

The VITAL CEP (Complex Event Processing) module is responsible for managing diverse IoT events stemming from different platforms and applications. The management platform can benefit from the CEP functionality, by defining management specific event processing instructions that the CEP will apply. For example, CEP can process streams of ICO status changes from IoT systems and generate aggregated notifications to the management platform (for example “N ICOs report a malfunction”).

In a wide sense, the functionality offered by the CEP is to process data streams, but the aim of this data processing depends on users intentions that are expressed by means of Dolce specifications.

The VITAL CEP module is able to run and manage many instances of the Complex Event Processing engine, All these instances are internally managed and monitored by the CEP administrator.

The administrator's module announces the VITAL CEP service to the management platform via the VUAI's service metadata endpoint and the providesService parameter of systems.

The administrator module of VITAL CEP also provides the vital:MonitoringService as it is required by the Management platform. The specification of the administrator is described in [Vital-D4.3.1-2016] .

Property	Type	Description
providesService	Dynamic	A list of services provided by administrator module of VITAL CEP apart from the Monitoring service.

3.2.4.4 VITAL Orchestrator module

The goal of the VITAL Orchestrator /BPM module is to achieve cross-platform and cross-business-context process integration in sync to the VITAL overall goal to provide an abstract digital layer over the application silos of the modern smart city.

The VITAL Orchestrator implements the VUAI specifications and implements service with types `vital:MonitoringService` and `vital:ConfigurationService`, required by the Management Platform.

While, generic IoT system's service are static in nature, the Orchestrator services are dynamically deployed and undeployed by users of the system. These dynamic services are monitored internally by the Orchestrator and are announced to the management platform, via the VUAI's service metadata endpoint and the `providesService` parameter of the system.

Property	Type	Description
providesService	Dynamic	A list of services offered by the system. Apart from the static ones (<code>MonitoringService</code> , <code>ConfigurationService</code> ...), this list is updated with new services created by users.

3.2.5 Service Level Agreements

Service Level Agreements (SLAs) define the terms of engagement for the participating entities and VITAL Platform. These parameters will provide a fundamental ground for interactions by means of Quality of Service.

SLA management model is in progress as a different deliverable (D5.3.X – Smart Governance Toolkit V1).

3.2.6 City Specific Information

This is directly related to the Governance tools of the VITAL platform. Since the related task has not produced concrete results yet and it is an on-going work at the time of writing this deliverable no specific parameters will be mentioned here. This information is related to deployment specific data that can be configured via the management layer UI for adapting the VITAL platform to the specific needs of the deployment environment (city) derived by the different social, economic and business parameters. The management UI will provide an overview of these data and allow the administrator of the platform to modify them when the need arises.

4 FUNCTIONAL AND TECHNICAL SPECIFICATIONS

4.1 Management Functionality Supported

Scope of the management platform is to provide the necessary tools for ensuring the smooth operation of the overall VITAL architecture. To this end, it supports the following functions:

4.1.1 Resource Discovery

The management platform, with the support of the Service Discovery module, is able to gather and display information on all the entities that participate in the VITAL architecture. The administrator is able to identify on the screen (a) IoT Systems: PPIs and VITAL Modules (b) ICOs/Sensors with enough administrative information on the purpose of each component.

The management platform supports multiple managed entities, discovered through three sources: (1) from a predefined list of VUAI URLs (system endpoints), and direct connections to each system to retrieve their metadata (2) a dynamic list of VUAI URLs retrieved from the IoT Data Adapter module, and again direct connections (3) through queries to the Service Discovery module. All sources are used in parallel with each one augmenting the information from the others.

4.1.2 Monitoring

Apart from the discovery of each module, administrators need to detect **faults** and **performance** bottlenecks. Thus, in addition to the topology a set of metrics are **monitored** and displayed on screen. The metrics for each component are defined in chapter 3.2. Scope of the management platform is to provide health-maps and dashboards to administrators, with which they will be able to identify problems as they occur.

To this end, two functionalities are supported. The first one is a location-based health-map that displays all participating entities in a geographical map, colour coded to indicate their current status. For entities without location information, a grid-based health-map is also available. This is the first level view of the overall system where the administrator can check the overall system at a glance.

The second one is a dynamic dashboard that is tied to specific managed components. Each component is described by a JSON-LD document mapped using the Vital ontology. The dashboard's scope is to map each part of the ontology to specific visualizations and create dynamic and agnostic views for every component. To this end, a set of different widgets have been developed directed to the different parts of the ontology:

Widget	Description
Information	Generic information on the entity, like name, description, to differentiate each entity and describe its high-level functions.
Status	This is the current status of the managed module. A green colour is associated with a functional module, whereas other colours indicate problems.

Services	Applicable to IoT Systems and VITAL Modules, it displays a list of the services and operations this entity supports.
Area	Displays a polygon in a geographic map. This is used in IoT Systems to describe which area they cover.
Location	Displays a marker with the accurate location of the entity on the map. More focused on ICOs that defines a specific location and not an area (in contrast to IoT systems that provide area coverage visualized as a polygon on the geographic map). In future iterations, it will be possible to draw a trajectory if this ICO is mobile.
Hardware	Displays information on the hardware, CPU, memory, hard disk storage etc.
Network	Displays information on network connectivity infrastructure (wired/wireless, delay, bandwidth and).
Errors	This widget fetches the errors and requests served by a system and draws a line graph with the error-rate for a period and a gauge with the error percentage.
Load	Retrieves load information and draws a gauge chart with the latest observation and a line chart with history of the load. An increasing number indicates a performance problem.
Memory	Displays the memory used and the memory available in a pie chart.
Throughput	Based on the requests served in a time period it calculates the throughput and visualizes it on a line chart.
Utilization	Calculates utilization from the throughput and the maximum request capacity of the module. It displays this information in a gauge chart.

4.1.3 Alerts / Events

The management platform supports dynamic alerts both by exploiting the health information of the system and by connecting to the CEP module for higher-level alerts.

Due to the capability of CEP of processing big amounts of data in real time, it can be used to monitor data stemming from ICOs in order to detect inappropriate values or the lack of data that can be a hint of malfunctioning of an ICO. Some of the aspects that can be monitored by CEP are:

- Data not received. It can indicate, for instance, problems in the communication channel of the ICO, ICO is off, etc.
- Data out of thresholds. It can indicate ICO malfunctioning.
- Trends of data not according to the expected behaviour.
- Correlation of data between similar ICOs, which monitor the same aspect or physical property, at the same geographical position.

The interface to set the required Dolce specifications will be a REST API offered by the management module of CEP. It is a new interface that offers the same

functionality as that defined for CEPICO functionality in “D3.2.2 Specification and Implementation of Virtualized Unified Access Interfaces V2” but dedicated exclusively to alerting function. This new interface has two intentions; first to separate data processing functionality from platform alerting, and the second one is to facilitate the internal processing of alerts. Although internally to CEP these alerts are complex events, they need some extra functionality in order to manage them, such as storing, acknowledge and deletion, which is not offered for normal complex events.

Alerts generated in CEP based on this processing will be sent to DMS. It is possible to send the alarms, filtered by content, source, etc., to the other functional modules or applications.

Below is the description of the Alerting service metadata and REST interface

Table 9: Metadata of the AlertingManagementService

```

{
  "msm:hasOperation": [
    {
      "hrest:hasAddress": "http:// example.com/cep/getalarmmonitors",
      "type": "vital:GetALARMMONITORS",
      "hrest:hasMethod": "hrest:GET"
    },
    {
      "hrest:hasAddress": "http://example.com/cep/getalarmmonitor",
      "type": "vital:GetALARMMONITOR",
      "hrest:hasMethod": "hrest:POST"
    },
    {
      "hrest:hasAddress": "http:// example.com/cep/createalarmmonitor",
      "type": "vital:CreateALARMMONITOR",
      "hrest:hasMethod": "hrest:PUT"
    },
    {
      "hrest:hasAddress": "http:// example.com/cep/deletealarmmonitor",
      "type": "vital>DeleteALARMMONITOR",
      "hrest:hasMethod": "hrest:DELETE"
    }
  ],
  "id": "http://example.com/cep/service/cepalarmmonitormanagement",
  "type": "vital:ALARMMONITORManagementService",
  "@context": "http://vital-iot.eu/contexts/service.jsonld"
}

```

Table 10: Get ALARMMONITORS

	Get ALARMMONITORS	
Description	Gets metadata about all ALARM Monitor.	
URL	CEP_BASE_URL/getalarmmonitors	
Method	GET	
Request headers	-	
Request body	-	
Response headers	Content-Type	application/ld+json or application/json

Response body	Example <pre>[{ "@context": "http://vital-iot.eu/contexts/sensor.jsonld", "ssn:observes": [{ "id": "http://example.com/cep/sensor/1/alarmnodata", "type": "vital:ComplexEvent" },], "name": "alarming system A", "description": "ALARM monitor for system A", "source": ["http://example.com/systemA/output",], "id": "http://example.com/cep/sensor/1", "type": "vital:CEPSensor", "status": "vital:Running" }]</pre>
Notes	<ul style="list-style-type: none"> The context of the response body is the JSON-LD context for sensors described in Section 3.3 of D3.1.1.

Table 11: Get ALARMMONITOR interface

	Get ALARMMONITOR	
Description	Gets metadata about a ALARMMONITOR with a specific ID.	
URL	CEP_BASE_URL/alarmmonitor	
Method	POST	
Request headers	Content-Type	application/json
Request body	Example <pre>{ "id": "http://example.com/cep/sensor/1" }</pre>	
Response headers	Content-Type	application/ld+json or application/json
Response body	Example <pre>{ "@context": "http://vital-iot.eu/contexts/sensor.jsonld", "ssn:observes": [{ "id": "http://example.com/cep/sensor/1/alarmnodata", "type": "vital:ComplexEvent" },], "name": "alarming system A", "description": "ALARM monitor for system A", "source": ["http://example.com/systemA",], "dolceSpecification": { "id": "XDolce", "event": [{ </pre>	

	<pre> "id": "vital:Speed", "definition": { " use": [{ "dataType": "string", "name": "id" }] }], "complex": [{ "definition": { " payload": [{ "dataType": "string", "name": "sensorId", "assign": "id" },], "detect": ["!vital:Speed"], "group": "id", "in": "60 seconds;" }, "id": " alarmnodata " }], "id": "http://example.com/cep/sensor/1", "type": "vital:CEPSensor", "status": "vital:Running" } </pre>

Table 12: Create or Update ALARMMONITOR

	Create / Update ALARMMONITOR	
Description	Creates or updates a ALARMMONITOR.	
URL	CEP_BASE_URL/ alarmmonitor	
Method	PUT	
Request headers	Content-Type	application/ld+json or application/json
Request body	Example <pre> { "@context": "http://vital-iot.eu/contexts/sensor.jsonld", "description": " ALARM monitor for system B", "name": " alarming system B", "source": ["http://example.com/systemA ",], "dolceSpecification": { "id": "XDolce", "event": [{ "id": "vital:Speed", "definition": { " use": [{ </pre>	

	<pre> "dataType": "string", "name": "id" }] } }, "complex": [{ "definition": { "payload": [{ "dataType": "string", "name": "sensorId", "assign": "id" }], "detect": ["!vital:Speed"], "group": "id", "in": "120 seconds;" }, "id": " alarmnodata" }] } </pre>	
Response headers	Content-Type	application/json
Response body	Example <pre> { "id": "http://example.com/cep/sensor/1" } </pre>	

Table 13: Delete ALARMMONITOR

	Delete ALARMMONITOR
Description	Deletes a ALARMMONITOR with the specified URI
URL	CEP_BASE_URL/deletecepico
Method	DELETE
Input	<pre> { "id": "http://example.com/cep/sensor/1" } </pre>
Output	-

4.1.4 Configuration

For each component, a separate service that exposes its configuration parameters is defined. Simple examples would include enable/disable, numberOfThreads etc.

The “Configuration” widget in the management platform provides a user interface that dynamically adapts to the ontology description of each configurable property and allows the administrator to make changes to the system. Its main purpose is to

display a different type of widget for editing each property, for example a simple textbox for string parameters, or a date-picker for date parameters. At the time of this deliverable, only string parameters were exposed by the existing system implementations, thus only textboxes are generated in the UI.

4.1.5 Security

In terms of security, the management platform is expanded to utilize the underlying infrastructure of VITAL and allow management of users, groups and authorizations policies and display basic statistics about user/application access to provide a monitoring of the access control system availability and possibly detect anomalous behaviours.

4.2 Technical Architecture and Requirements

The following high-level diagram (Figure 2) provides an overview of the technical architecture of the Management Layer of VITAL as specified in T5.1.

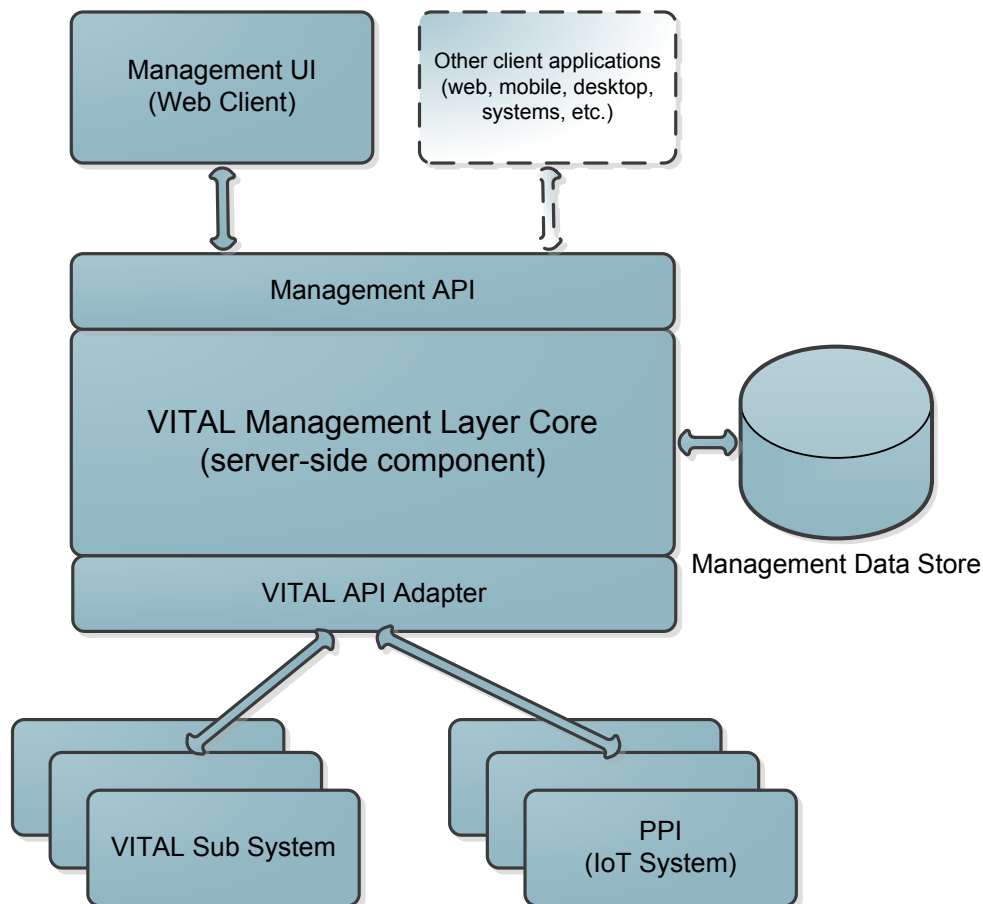


Figure 2: Management Layer technical architecture

The server side part of the Management Layer sub-system consists of the following components:

- The core server process. This is the implementation of the management business logic including scheduling of polling requests, handling of push notifications by other VITAL components, alerts setup and initiation, processing of first-hand management data and calculation of metrics, storage and retrieval of stored data (historical, accounting), event generation, etc. This is the heart of the Management Layer sub-system that controls and coordinates all other components explained below. This has been implemented in Java/JEE and it is deployed in a JEE application server; we selected the WildFly¹ 10.x server, a widely used and open source JEE server.
- Management data store. This component handles storage/search/retrieval of configuration, monitoring and historical data; it can also be used for caching of information for improved performance. Since all data in VITAL are exchanged between components in JSON-LD representation, the Management Layer uses a document store that can natively handle JSON documents. The design and first prototype was based on ElasticSearch². Now it uses MongoDB to exploit a much more robust data storage platform, and to be in line with the rest of VITAL platform, like the DMS implementation that also opted for MongoDB.
- Management UI. This is the web client of the management layer that provides a rich UI to the administrator and implements the functionality specified in this deliverable. It is a Javascript/HTML5/CSS3 single page application based on the AngularJS³ framework; it also relies on the bootstrap⁴ CSS/JS framework for implementing a responsive design / fluid UI that supports both desktop and mobile web browsers with various screen sizes.
- VITAL API Adapter. This is the implementation of the VITAL API client code that the management core uses for retrieving information and communicating with all the components of the VITAL architecture (VITAL sub-systems like the CEP, Service Discovery module, Orchestrator, etc. and the PPI management services). This is also a server-side component implemented in Java/JEE using the JAX-RS API/specification. This component handles dynamic operation calling, proper data formatting and security in all interactions between the management platform and other VITAL services.
- Management API. The server component that implements the RESTful management API through which the Management UI communicates with client applications. Currently this is only used by the Management UI web client application, but future clients or other components of the VITAL ecosystem can use it.

The design and implementation of this architecture should comply with the following requirements:

- Usability. Although the user of the management layer is the platform administrator (i.e. a technical savvy user), and not the end-user of the

¹ <http://wildfly.org/>

² <http://www.elasticsearch.org/>

³ <https://angularjs.org/>

⁴ <http://getbootstrap.com/>

applications or the application developer, the user experience and user interface should be friendly, concise and not overwhelmed with too much information. The goal is to support efficiency and early warning.

- Layered and modular approach. Easy maintenance and extensibility is important as the management layer may evolve rapidly with additional functionality as the platform matures and the supported IoT platforms increase in number and provide more monitoring and configuration hooks. Clear separation of layers and modularity is important in the design and implementation of the management layer component.
- RESTful APIs / JSON-LD. As a VITAL component the management layer must be able to consume and produce REST/JSON-LD data streams that are conform to the VITAL semantic data model.
- Security. The application must be designed in a secure way that will protect its assets from OWASP-10 attacks and unprivileged access in general.

4.2.1 Reengineering and Migration of existing IoT applications

In order to migrate an existing IoT system to VITAL, a PPI must be implemented. This means creating a module able to map data and metadata from the existing system to the JSON-LD format used by VITAL and expose a set of REST APIs to provide the VITAL platform with the mapped information. This set of endpoints is described in D3.2.3. The PPI must then be registered to VITAL through the IoT Adapter UI.

In this way, the data and metadata from the IoT system are automatically available through VITAL services, which can then be exploited by application developers. Notably the developer has access to the Development Tools of the platform.

In order to ease the attachment of IoT systems and sensors to the VITAL platform we provide a Migration Toolkit, the implementation of which is described in next section.

5 PROTOTYPE IMPLEMENTATION (V3)

5.1 Supported Functionality

The prototype implementation focuses on the following functionality:

- Overall UI design. Based on the UI/UX guidelines of the previous version of the prototype, the UI design has been adopted for all additional modules since the first prototype. Furthermore a style update has been performed.
- Health map of a VITAL deployment. The implementation provides a geographical map view based on the OpenStreetMap data feed and demonstrates the Health map approach.
- Monitoring Dashboard. This is a set of entity agnostic management widgets/tools that visualize the VITAL ontologies. These are presented in a dynamic dashboard view as explained in the previous section. Currently the prototype contains implementations of all the widgets described in section 4.1.2, integrated with real live data from existing systems.

- Configuration View. This is a dynamically adapted configuration screen. It displays and allows editing the configuration parameters of systems.
- Security Management. The security management module allows: (1) Registering/managing users, (2) Defining/managing groups/roles, (3) Assigning roles to users or revoking roles from users, (4) Defining/managing permissions, (4) Monitoring number of sessions and policy evaluations, (5) Testing users' permissions over resources.
- Event processing as a mean to build valuable information by correlating data stemming from different sources. The VITAL CEP functionality will provide access to expert programmers through the administration interface and the Dolce Language, and also to non-expert users, through predefined Dolce specifications, that will be offered as services, and internally managed by the Management layer. This management comprises not only the Dolce specifications, but also the on the fly creation/deletion of new instances of the CEP.
- Data model and Communication layer. The implementation understands the VITAL ontologies and processes the JSON-LD syntax as specified by the project at this time.

5.2 Migration Plan: from prototype to concrete implementation

As explained earlier in the text, we adopted a top-down approach, starting with a UI prototype that relied on mock-up data streams and tried to identify specific functional requirements through multiple iterations. The first prototype implementation has been validated within the VITAL consortium by discussing and updating iterations, thus resulting in a first functional UX/UI design as well as the identification of more detailed functional and technical requirements. In addition to that, the prototype led to the initial implementation of the VITAL data model and the VITAL API module that was integrated with the PPIs for end-to-end manipulation and visualization of actual data produced by real IoT systems.

As part of the second release we have continued working in iterations gradually implementing all layers of the management layer architecture and integrating with the management instrumentation as they become available from the implementations of the VITAL core components (e.g. Discoverer, CEP) and the PPIs of the targeted IoT systems. The first iterations have focused on implementing and validating as much as possible of the functionality specified, while the final iteration has been mainly focused on optimizations, debugging and fine-tuning.

As part of the third release of the deliverable the following updates have been implemented:

- Resource Discovery: finalized the mechanism of finding available resources in a Vital installation. Two options are still supported (1) a predefined list of VUAI URLs and direct connection to each VUAI for retrieving metadata (2) Search available VUAIs through the Service Discovery module. The Resource discovery mechanism now supports more complex queries, thus filtering results and being able to support large numbers of sensors.
- Performance Monitoring: no major changes in the performance monitoring aspect in terms of features. All widgets designed from the first prototype are

now fully functional to support real live data from systems, as defined in the VITAL ontology. With new VUAI conformant systems added in this period, the monitoring widget are tested in more use cases and improvements have been performed, mainly in terms of interoperability.

- **Configuration Management:** the management platform exploits the Configuration Service of VUAI systems, as to provide a dynamic user interface, through which administrators can view system properties and adjust them properly. This module is the building block for the Governance's toolkits Smart City management functionality, where more complex operation than editing single system properties will be supported.
- **Security:** the Management Platform is now fully integrated with the security mechanism, both in terms of authenticating and providing access only to administrators, but also with offering them the necessary tools to manage users, groups and policies. A module to monitor and configure security has already been implemented in the previous release. In this release, it has been fully imported in the management system, alongside with resource discovery, monitoring and configuration options. It includes a module interacting with the OpenAM identity provider through REST APIs. An important addition since previous release is support for fine-grained data access control, while at the UI level a new section to test for users' permissions has been added; also, all functionalities have been expanded and improved. In the next phase, the security and security management modules will be tested on the integrated VITAL prototype; functionalities and UI will be evaluated and, where needed, expanded to support simple and effective security configurability and monitoring.
- **Alert/Event Management:** the CEP module has been extended with a new interface specifically designed for alerts.
- **SLA management module:** this module has now become part of the Governance Toolkit and will be reported there.

The functionalities implemented as part of the third version are described in the following paragraphs.

5.3 Dynamic Resource Discovery

As stated Dynamic Discovery is based on three approaches, (a) a predefined list of systems that expose VUAI interfaces with direct interaction to these systems, (b) a dynamic list of systems retrieved by the IoT Data Adapter registry and (c) by exploiting the Service Discovery and DMS modules. In what follows we will describe the algorithm of the first approach, to showcase how systems actually describe their capabilities and how this information is then consumed to fill the topology database and be displayed on the screens of administrators. The second option works in exactly the same manner. Next we will focus on the alternative of exploiting the Service Discovery and DMS modules.

Each VUAI exposes the following required API calls (described in detail in deliverable D3.2.3):

<BASE_URL>/metadata	Returns the system's static information as described in 3.2.1
<BASE_URL>/service/metadata	Returns the system's supported services and operations as described in 3.2.2
<BASE_URL>/sensor/metadata	Returns the sensors that the system provides access to, as described in 3.2.3

The management platform, given the list of BASE_URLs (either statically or from the IoT Data Adapter), connects to each system and performs the following steps:

1. Stores in cache the static information of all components and uses it to generate the System List, Sensor List and Geographical Health-map (for components that contain geo-location data).
2. Searches for the services of the following types: (a) vital:MonitoringService (b) vital:ConfigurationService to detect if the system is manageable.
3. Interacts with the two services to generate performance graphs, configuration hooks and live status updates.

This procedure is repeated every few minutes, to detect changes in the topology and update the management cache accordingly.

Alternatively, instead of connecting to each VUAI directly, the management platform exploits the added value services of VITAL, i.e. the Service Discovery and DMS.

In order to retrieve the different systems connected to VITAL, the Discovery module capabilities for finding systems, services and sensors are used. The procedure is more or less the same; instead of connecting to each system separately to collect metadata, SD provides a full list of all registered systems with their capabilities.

After detection of all static information, performance metrics values are obtained through the systems directly, skipping intermediary services, as to display the latest value. When historical data are necessary, administrators can query DMS through the Management platform and retrieve them.

Following are a few screenshots of the UI, with the dynamically generated list of IoT Systems and Sensors:

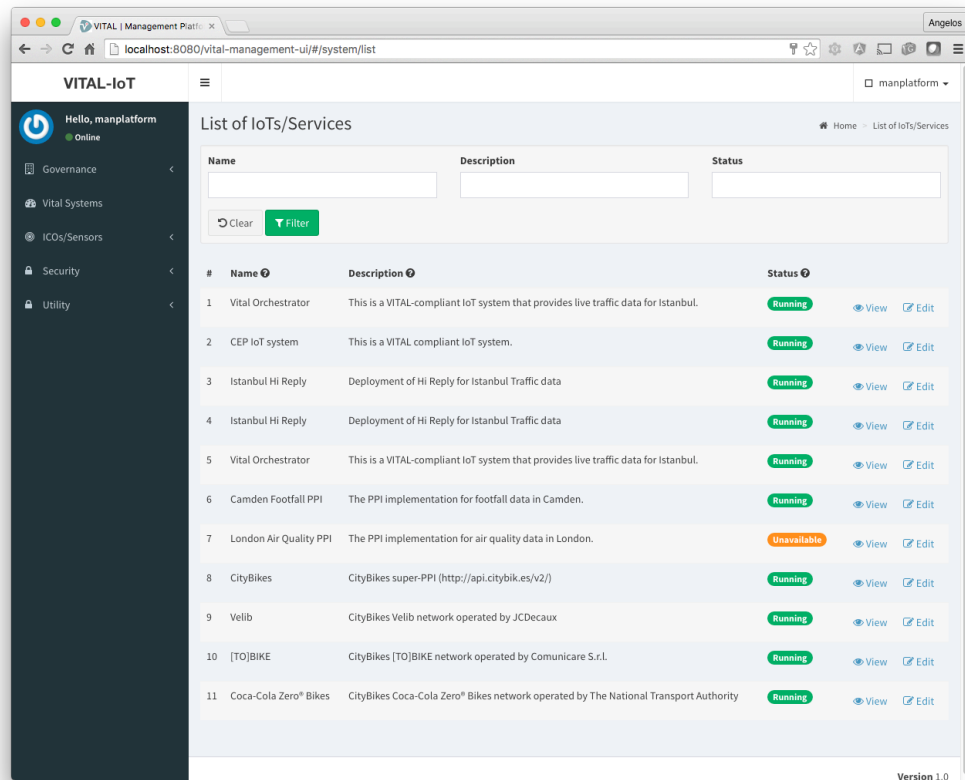


Figure 3 Management UI: List of Systems

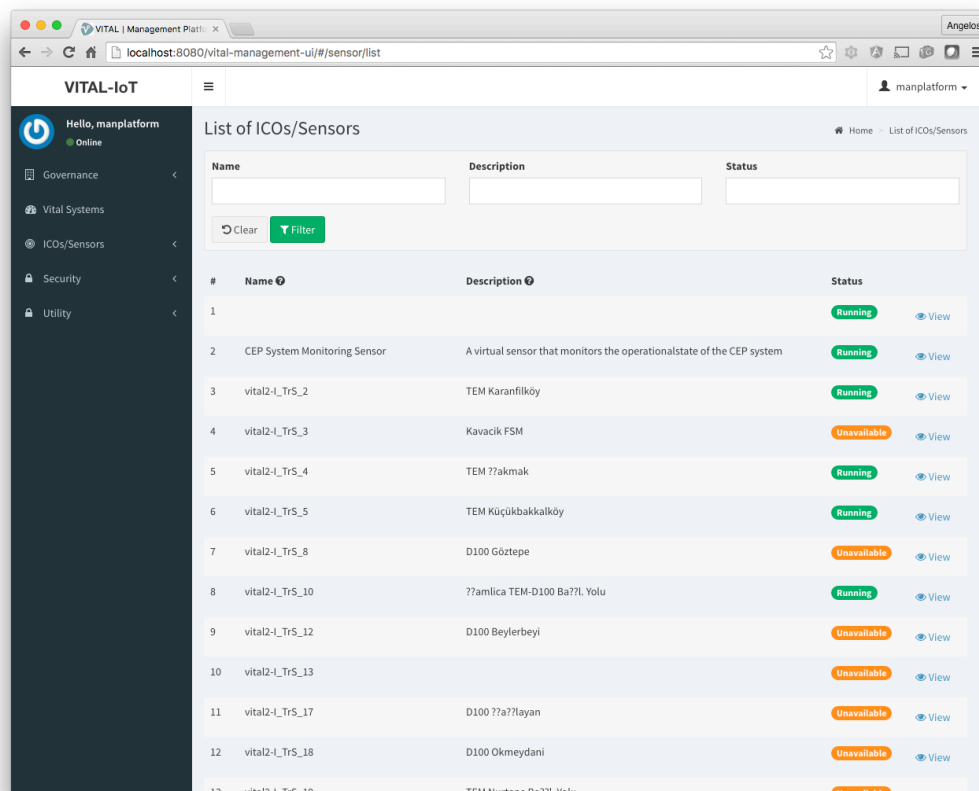


Figure 4: Management UI: List of Sensors

5.4 Performance Monitoring

The following are screenshots from the monitoring dashboard of systems and the overall health-map. These views are generated with a combination of static information and the live performance metrics as obtained by systems.

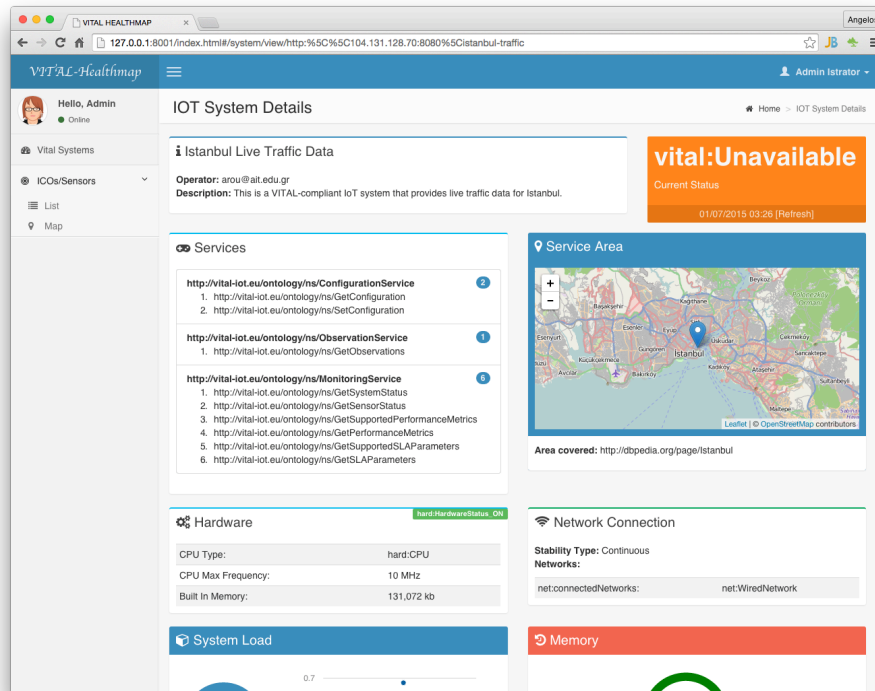


Figure 5: Management UI: System Overview – Info and Status

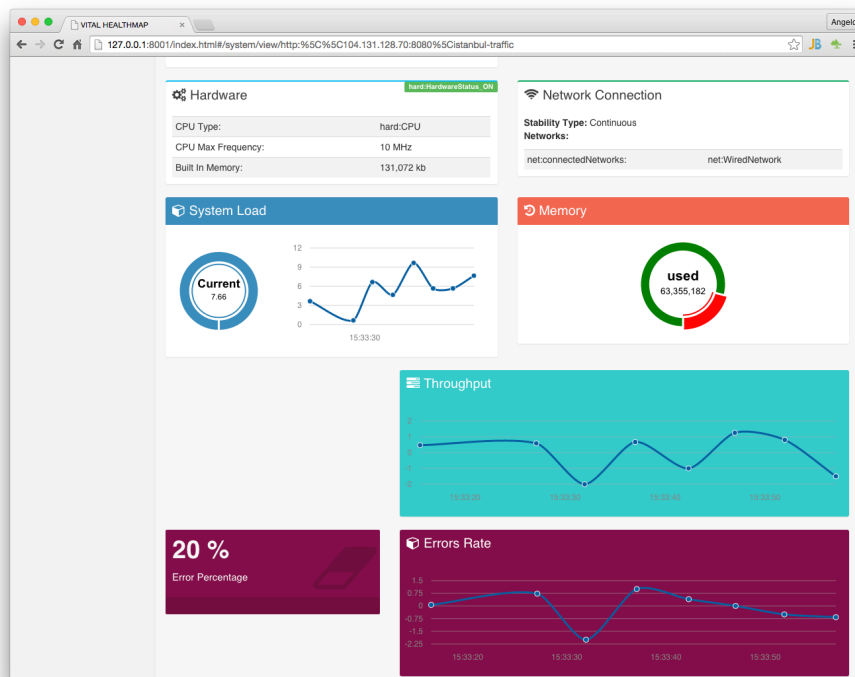


Figure 6: Management UI: System Overview - Performance Graphs

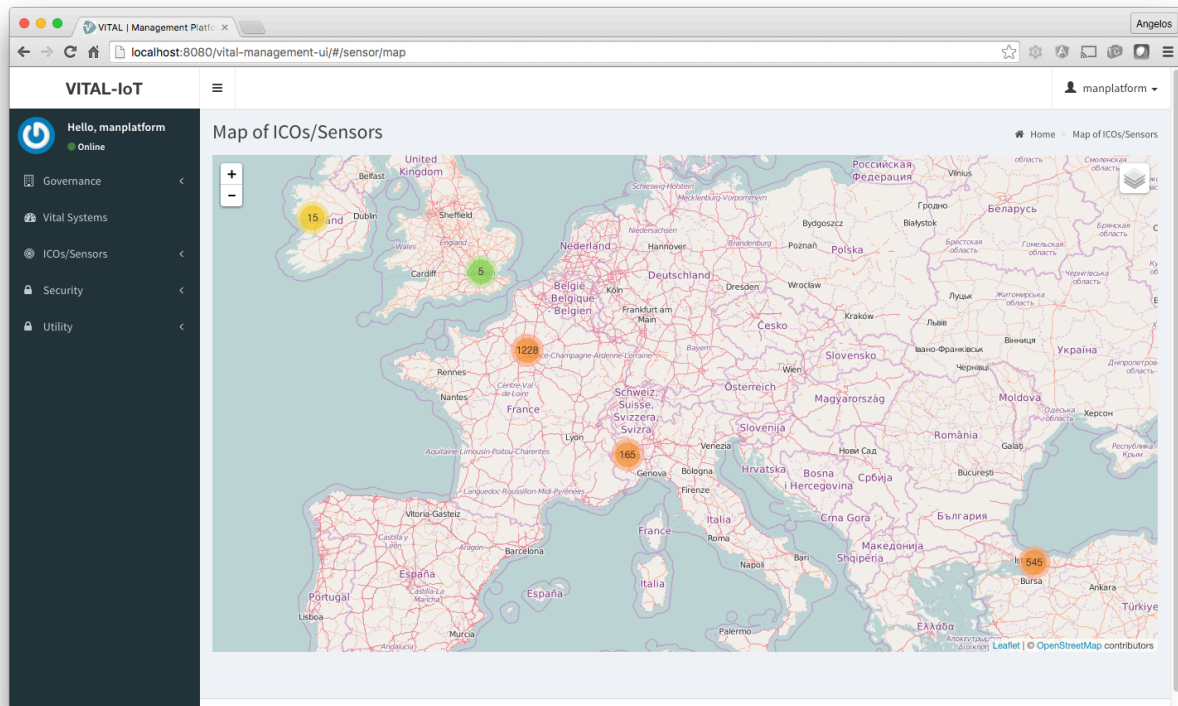


Figure 7: Management UI: Geographical Health-map

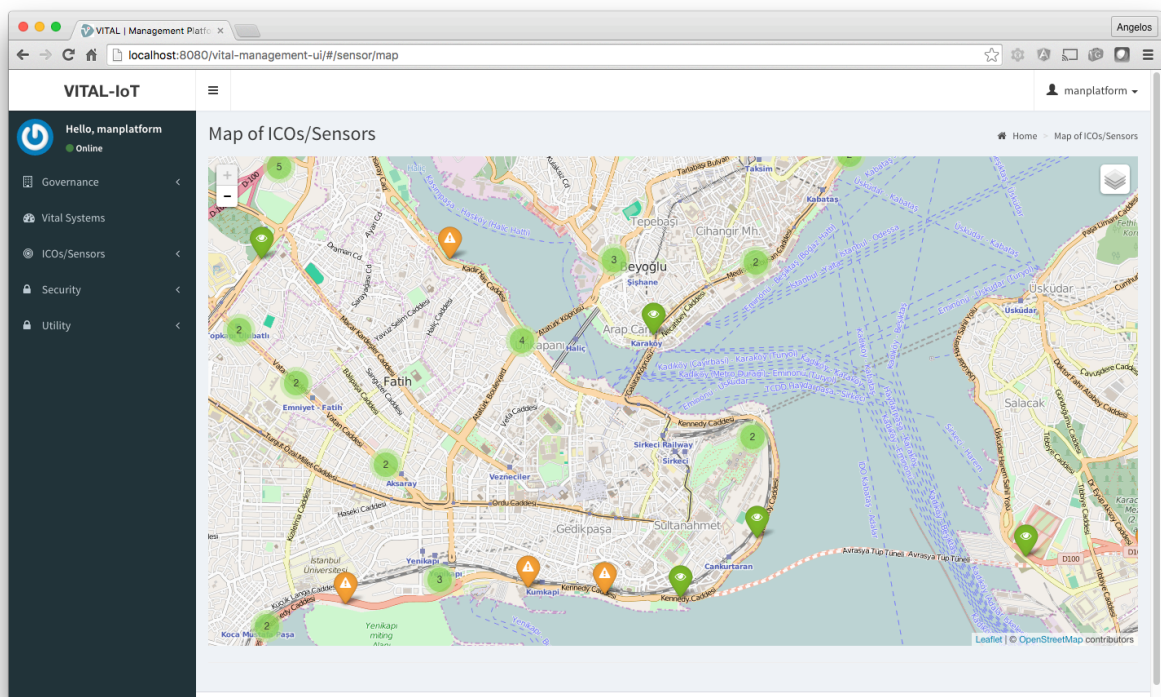


Figure 8: Management UI: Geographical Health-map – Focus on Istanbul

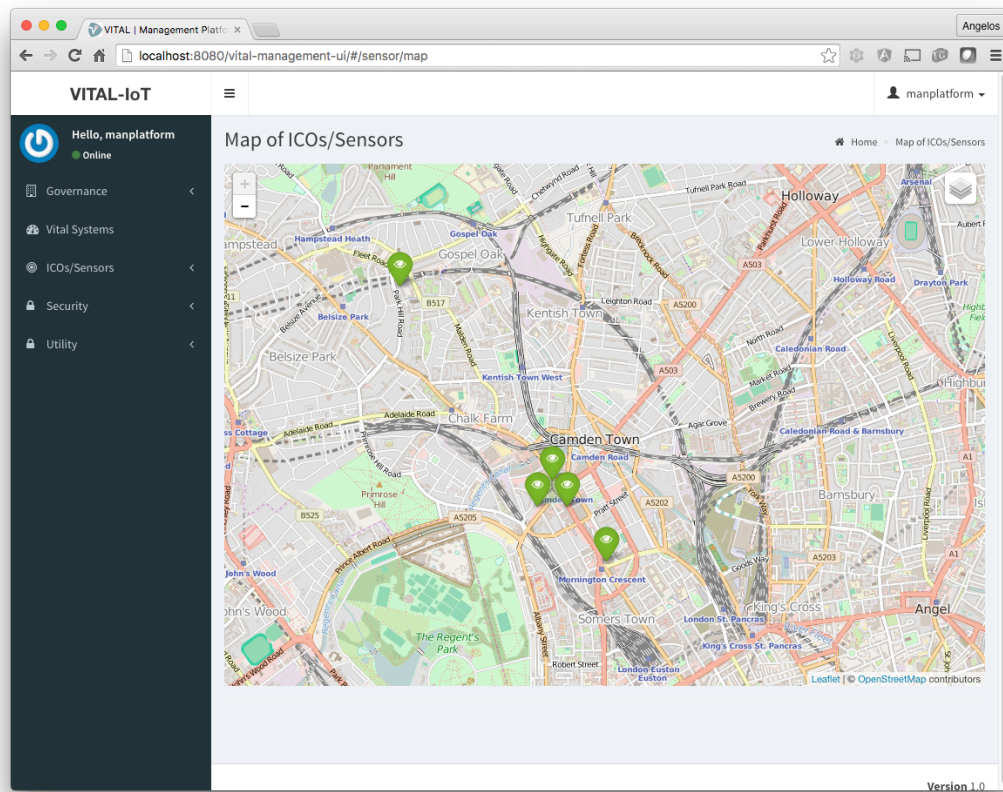


Figure 9: Management UI: Geographical Health-map – Focus on Camden

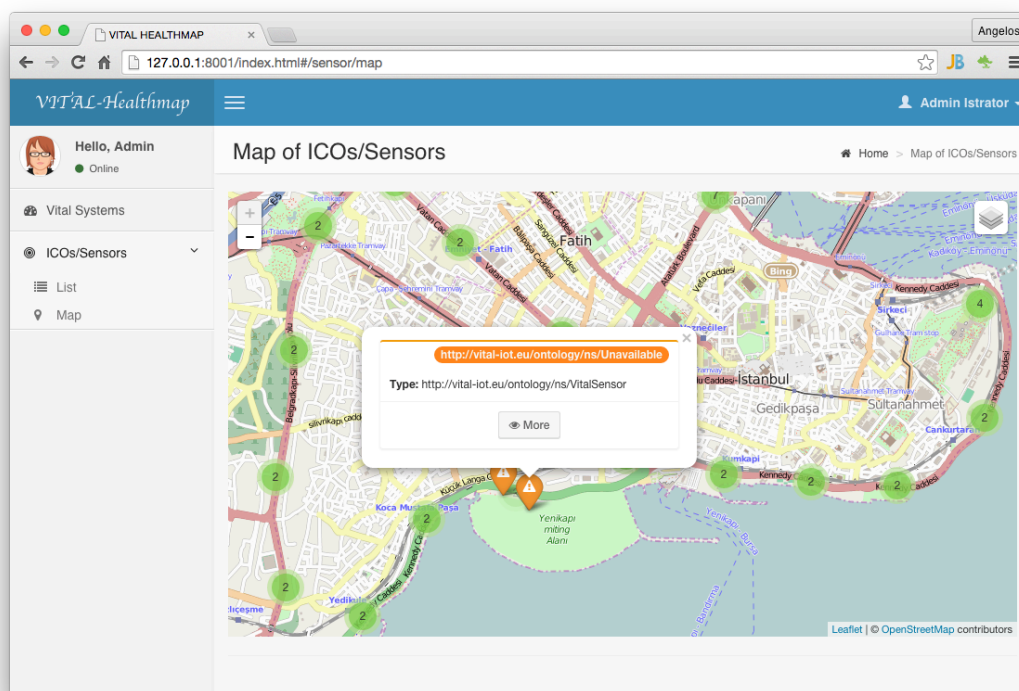


Figure 10: Management UI: Geographical Health-map - Sensor Details

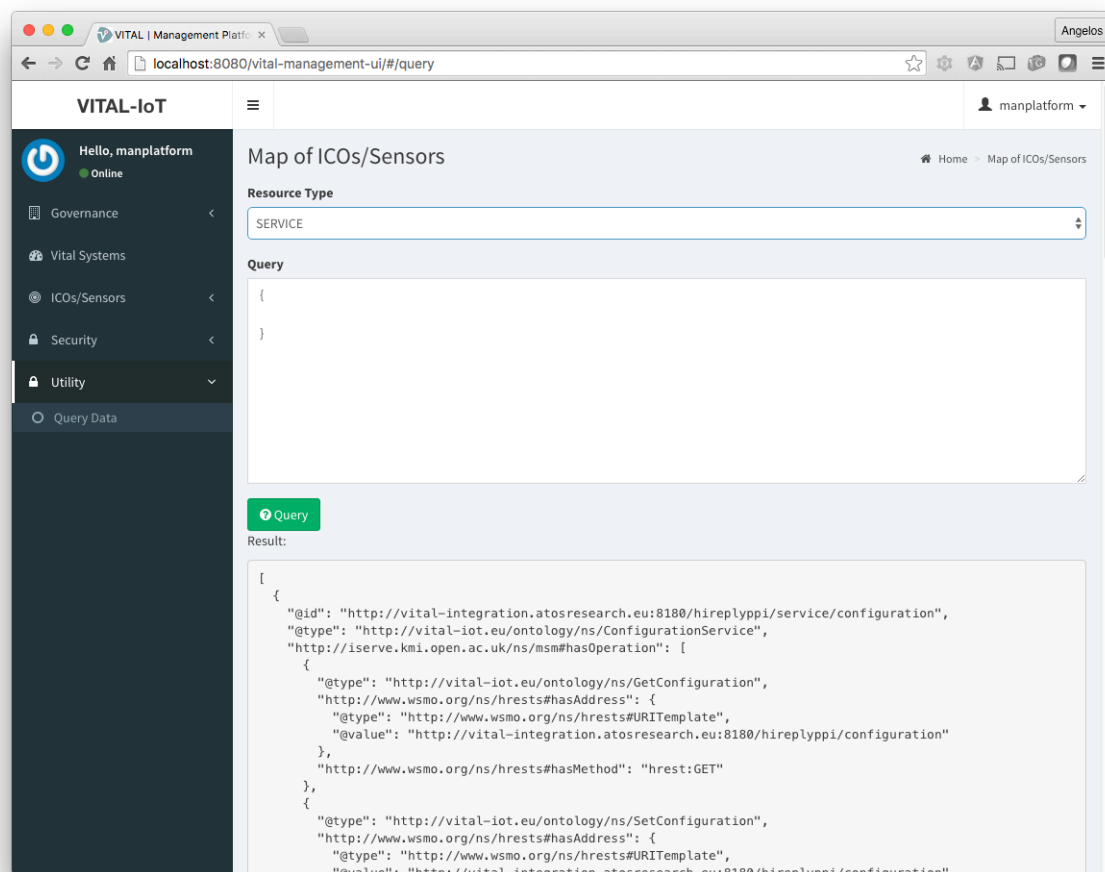


Figure 11: Management UI: DMS Query Tool

5.5 Configuration Management

Figure 12 is a screenshot of the configuration widget, as it is populated with information retrieved by the system's configuration service. Some parameters are editable while others are read-only. Pressing the button "Update" will cause the management platform to directly connect to the system and send the new configuration.

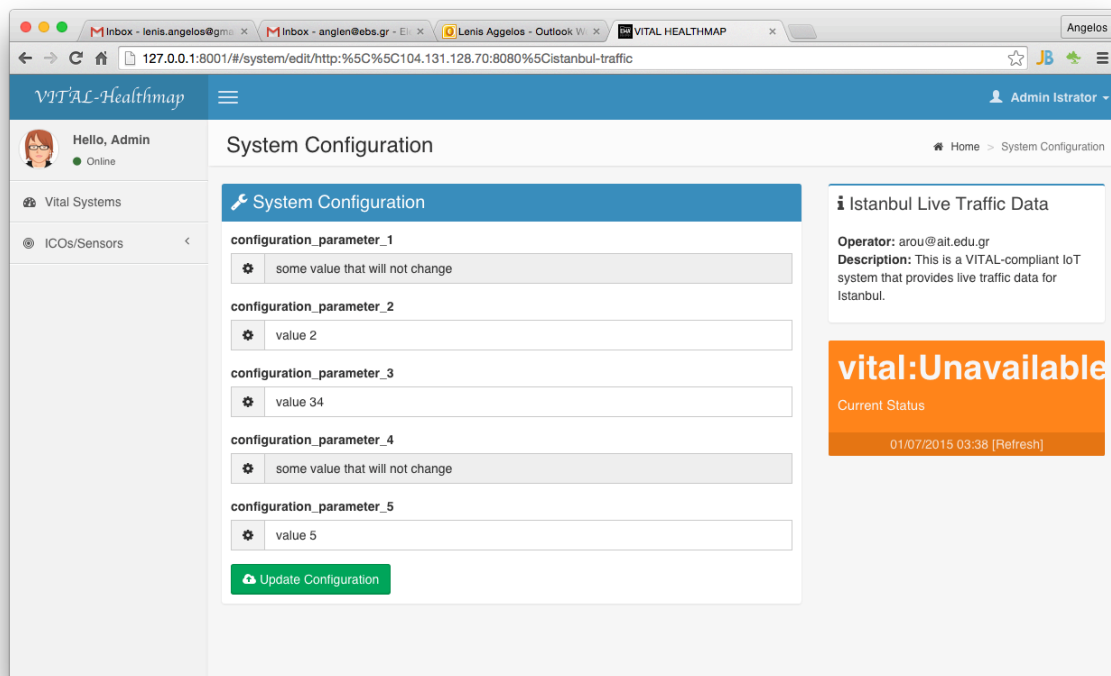


Figure 12: Management UI: Configuration

5.6 Alerting and Event Management Services

Figure 13 and Figure 14 are the screenshots from the alarm management system implemented by using the capabilities of VITAL CEP service; each alarm monitor system is a CEP instance running a specific set of Dolce rules to detect anomalies in a system. The complex events raised by this CEP instances have a special treatment due to their special nature.

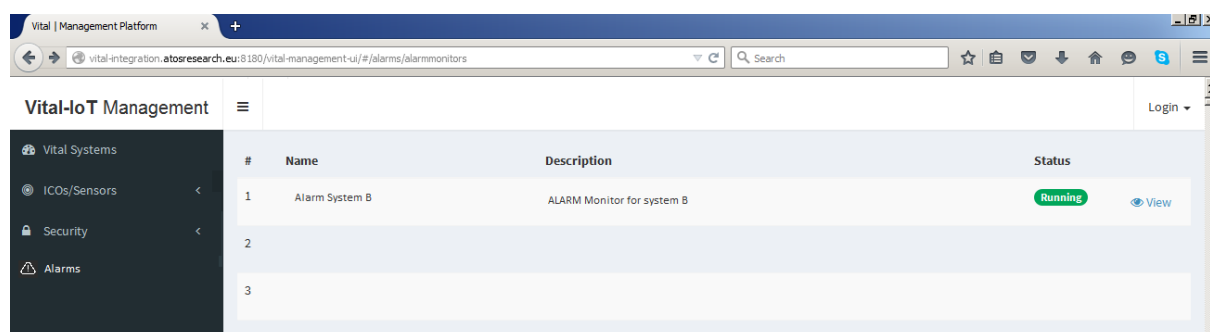


Figure 13: Alarm Monitor systems list

Figure 14 shows the alarms or complex events raised by a CEP instance working as an alarm monitor:

Vital IoT Management Platform

Alarmmonitor id: http://example.com/cep/sensor/1

#	Name	Description	Time
1	http://vital-integration.atosresearch.eu:8081/cep/sensor/1/observation/54325	alarmnodata	2016-01-15T10:15:09+01:00
2	http://vital-integration.atosresearch.eu:8081/cep/sensor/1/observation/54322	alarmnodata	2016-01-15T10:20:12+01:00
3	http://vital-integration.atosresearch.eu:8081/cep/sensor/1/observation/6345	alarmnodata	2016-01-15T10:45:24+01:00
4	http://vital-integration.atosresearch.eu:8081/cep/sensor/1/observation/23451	alarmnodata	2016-02-07T17:14:59+01:00
5	http://vital-integration.atosresearch.eu:8081/cep/sensor/1/observation/3471	alarmnodata	2016-02-07T17:47:03+01:00
6			
8			

Figure 14: UI showing alarms reported raised by a CEP instance

5.7 Security Management Functionalities

The VITAL management layer includes the functionalities needed by administrators to monitor and manage security. The security management module allows:

- Registering/managing users
- Defining/managing groups/roles
- Assigning roles to users or revoking roles from users
- Defining/managing and testing permissions
- Monitoring number of sessions and policy evaluations

Permissions are applied at the group level, CRUD (Create, Read, Update, Delete) actions are possible on users, groups, policies.

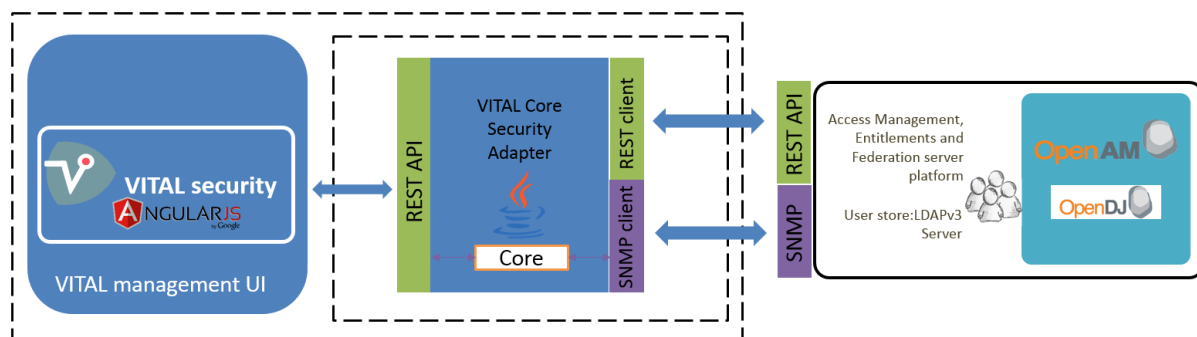


Figure 15 - Security management architecture

The VITAL security management component interacts with the OpenAM identity provider through a security core module that accesses data about users, groups and policies through the REST APIs exposed by OpenAM. The security core module is a Java application deployable in a WildFly application server.

It exposes a REST API used by the VITAL Management user interface. This interface is the 'Security' section of the management web (Javascript/HTML5/CSS3) application based on the AngularJS framework.

It also retrieves information about the access control activity useful to monitor and ensure appropriate performance and service availability; these statistics are retrieved through the OpenAM SNMP (Simple Network Management Protocol) interface over UDP, using Object IDentifiers defined in a Management Information Base (MIB) file. Currently reported figures include:

- Total number of current sessions
- Cumulative number of policies evaluations
- Average rate of policy evaluations

It is noteworthy that besides the information shown in the Monitor section, detailed trails can be found in the log files. OpenAM servers generate two types of log files: audit logs and debug logs. While debug logs are unstructured and mainly intended for debug and troubleshooting purposes, audit logs capture normal operational information about OpenAM usage. Audit file records are structured: they adhere to a consistent, documented file format, are by default logged to flat files but could be sent to relational database tables or a syslog server.

OpenAM log files are named after the service logging the message, with two types of log files per service: *.access* and *.error*. For instance the audit log files for the authentication service are named *amAuthentication.access* and *amAuthentication.error*.

Similarly, OpenAM policy agents also produce agent audit logs.

The following section 5.7.1 describes the functionalities while showing user interface screenshots, while Section 5.7.2 details the REST interfaces provided by the security core module.

5.7.1 Security management user interface

As shown in Figure 16 the Security management functionalities are grouped in six subsections:

- Users
- Groups
- Policies
- Applications
- Monitor
- Access test

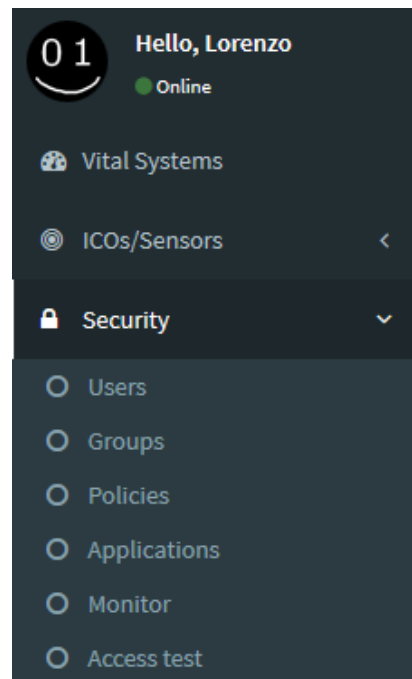


Figure 16 – Management UI menu including security section

The Users control panel allows to list users (Figure 17), edit modifiable users attributes (e.g. email) (Figure 19), delete users or add new users (Figure 18).

From the Users panels it is also possible to manage the membership of a user.

Groups can be specifically managed from the Groups panel, which allows performing CRUD actions.

Groups are intended to represent roles and simplify permissions assignment. It has been decided that permissions cannot be directly assigned to users but must be assigned to a group/role. In this solution, roles and permissions are mapped to the group memberships of users (Figure 20).

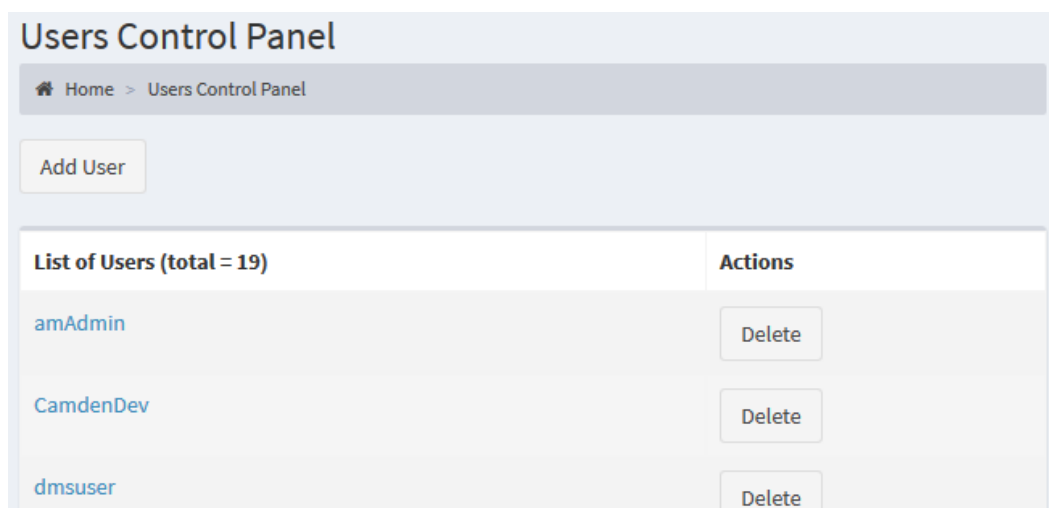


Figure 17 – Users listing

Users Control Panel [Add User](#)

[Home](#) > Users Control Panel

First name

Last name

Username *

Password (at least 8 characters) *

E-mail

[Create User](#)

New user "sconnor" successfully created!

Figure 18 – User creation

Users Control Panel [User Details](#)

[Home](#) > Users Control Panel

[Edit](#)

Successfully updated user details!

Username	Realm	First name	Last name	E-mail	Status
sconnor	/	Sarah	Connor	sarah.connor@terminator.org	Inactive

The user is part of no groups.

Add "sconnor" to group ▾

- Advanced_Users
- can

Figure 19 – User editing

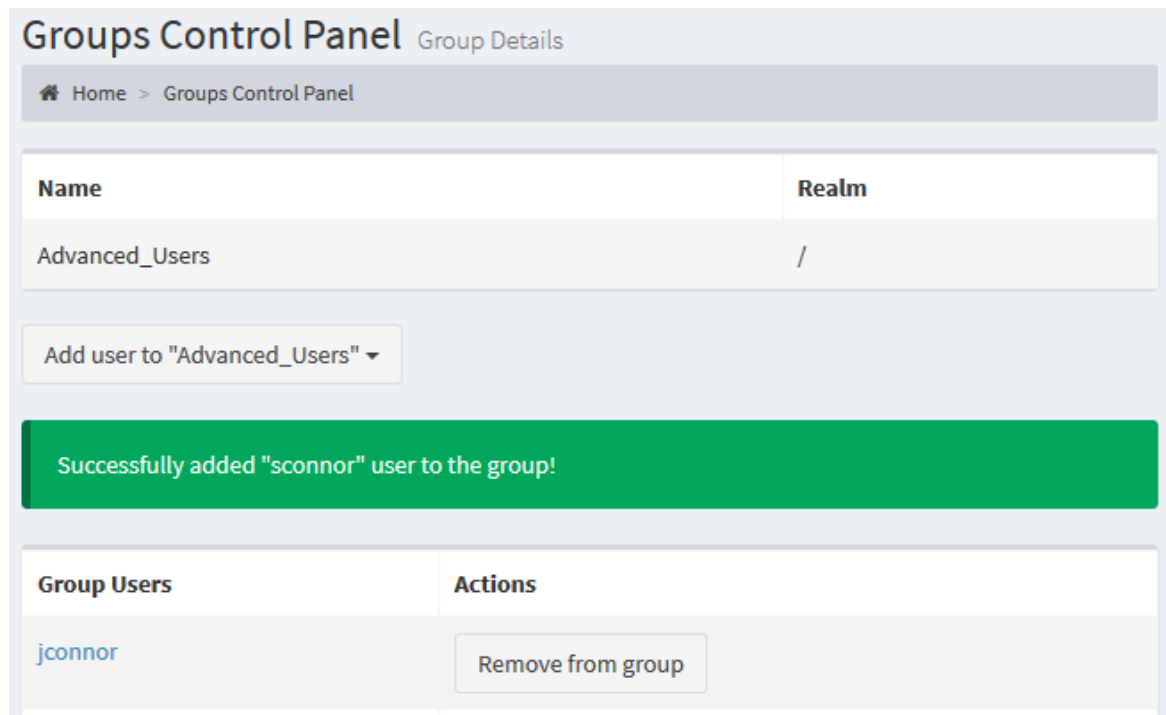


Figure 20 – Group control

The Policies panel, which allows performing CRUD actions on policies, used to manage permissions, is shown in Figure 21.

For specifying the resources in the policy definition, the URL or resource name to protect can be specified individually or by using patterns with the wildcards `*` and `-*`. These wildcards can be used throughout resource patterns to match URLs or resource names (they can be used to match protocols, host names and port numbers too).

The wildcard `*` matches multiple levels in a URL path, while the wildcard `-*` matches a single level in a path.

For example the pattern `http://www.example.com/-*` would match the URL `http://www.example.com/index.html` but would not match `http://www.example.com/company/images/logo.png`, which has a path composed of more than one level (*company*, *images* and *logo.png*).

The pattern `http://www.example.com/*` would instead match both of the URLs above.

Other particular rules used by OpenAM are:

- The wildcard `*`, used at the end of a pattern and after a `?` character, matches one or more characters, not zero or more characters as usual.
- Duplicate slashes (`/`) in the URL are not considered part of the resource name to match. A trailing slash is considered by OpenAM as part of the resource name.
- Wildcards cannot be escaped.
- Comparisons are not case sensitive (by default).

Action	Specify	Allow
POST	<input checked="" type="checkbox"/>	No ▾
GET	<input checked="" type="checkbox"/>	Yes ▾

Figure 21 – Policy creation

The applications panel (Figure 22) has been added in this third release and allows managing OpenAM “applications”, which allow defining rules for the patterns to be used in policies and at the same time, they act as entities grouping the different types of policies. At this moment, this is used to keep differentiated policies to control access to URL based resources and policies defining fine-grained access control at the level of DMS and PPIs.

The Monitor panel, which allows monitoring of the load and performance of the access control system, is shown in Figure 23.

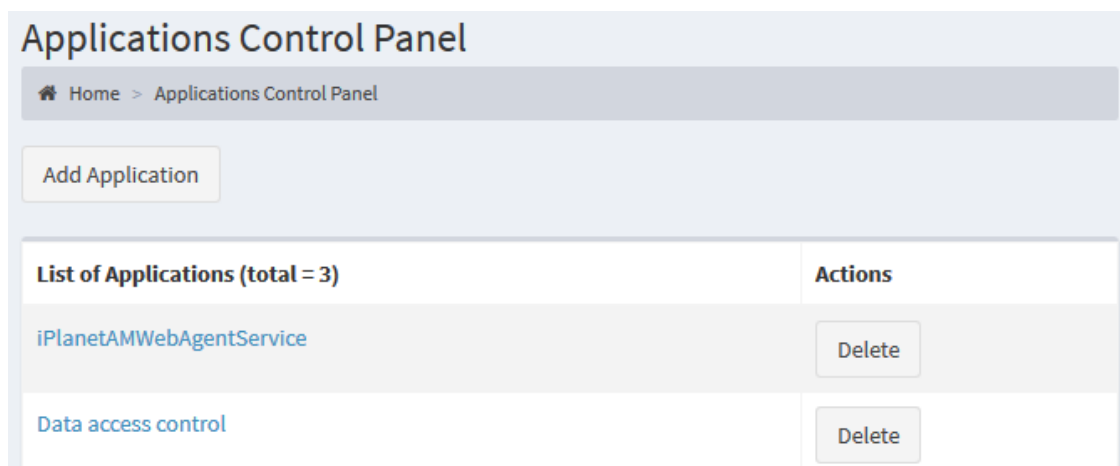


Figure 22 – OpenAM applications

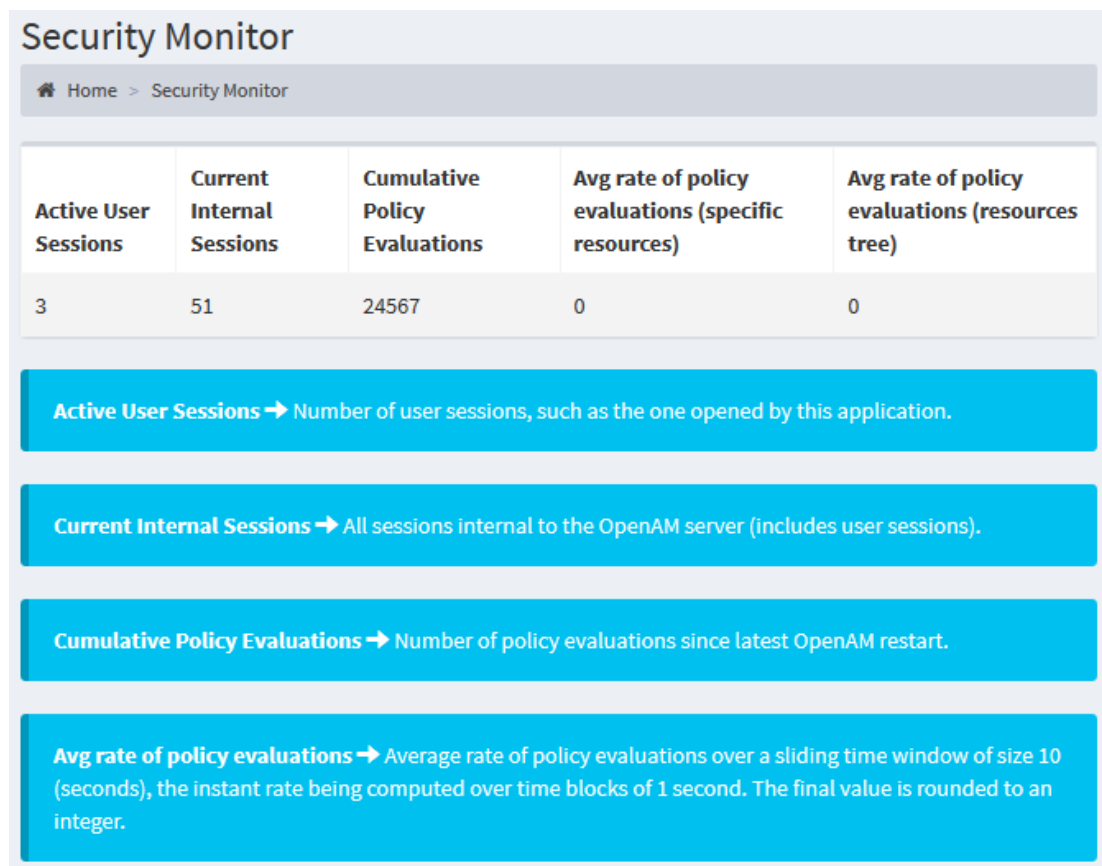


Figure 23 – Monitor interface

The other panel added with the third release is the Access test one, allowing to quickly check the permissions of a user (Figure 24).

Access Control Test Home

User "admin" is now logged in Log out

Data access control

Field	Value	Allowed
RETRIEVE		
@type	*	Yes

Endpoint (is it accessible?)

Evaluate

Action (on " https://vitalsp.cloud.reply.eu/vital10/vital-ppi-citybikes ")	Permitted (for user "admin")
POST	Yes
GET	Yes
DELETE	Yes
PUT	Yes

Resource content (response body to GET request)

Figure 24 – Access Control Test panel

5.7.2 Security module REST API

The core security module exposes a RESTful interface allowing to perform the needed operations described above. The response are in a JSON format.

The list of APIs is detailed in Section **Error! Reference source not found.**, along with responses and status codes. The endpoints respond to successful requests with HTTP status codes in the 2xx range, while they respond with HTTP status codes in the 4xx and 5xx ranges on errors.

The core security module exposes the following APIs. The response bodies are the JSON representations of the requested resources or of the errors coming from OpenAM. More detailed information about the precise formats is provided in the documentation bundled in the open source project repository.

GET endpoints

You may have to pass information in three different ways for GET methods:

1. As part of the path of the endpoint (e.g. <https://vital.com/endpoint/parameter>)
2. As cookies
3. As query parameters (included at the end of the URL, e.g. <https://vital.com/endpoint?par1=val1&par2=val2>)

Path	<i>/user/{id}</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns some info about the user identified by <i>id</i> .

Path	<i>/user/{id}/groups</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns the list of groups having the user identified by <i>id</i> as member.

Path	<i>/group/{id}</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns some info about the group identified by <i>id</i> .

Path	<i>/policy/{id}</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns some info about the policy identified by <i>id</i> .

Path	<i>/application/{id}</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns some info about the application identified by <i>id</i> .

Path	<i>/apptype/{id}</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns some info about the

	application type identified by <i>id</i> .
--	--

Path	<i>/application/{id}/policies</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns some info about the policies part of the application identified by <i>id</i> .

Path	<i>/users</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns the list of users.

Path	<i>/groups</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns the list of groups.

Path	<i>/policies</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns the list of policies with some info for each of them.

Path	<i>/applications</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns the list of applications with some info about each of them.

Path	<i>/apptypes</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns the list of application types with some info about each of them.

Path	<i>/stats</i>
Description	Expects the <i>vitalAccessToken</i> session cookie to be included in the request and returns some statistics about the status of the OpenAM server.

Path	/user
Description	Expects either the vitalAccessToken session cookie or the vitalTestToken one to be included in the request and returns some info useful for session management (whether the user session is still valid or not and some info about the user); if the query parameter testCookie is set to true the info is related to the session of the vitalTestToken cookie, otherwise of the vitalAccessToken cookie.

Path	/validate
Description	Expects the vitalAccessToken and vitalTestToken (the latter is optional) session cookies to be included in the request and returns a single json boolean attribute telling if the session is active or not; if the query parameter testCookie is set to true the info is related to the session of the vitalTestToken cookie, otherwise of the vitalAccessToken cookie. While the above endpoint resets the user idle time, this one does not, but because of an OpenAM bug requires the user corresponding to the vitalAccessToken to be an administrator.

Path	/getresource
Description	Expects either the vitalAccessToken session cookie or the vitalTestToken one to be included in the request and returns the response of a GET request to the URL specified in the query parameter resource ; if the query parameter testCookie is set to true the vitalTestToken cookie is included in the request, otherwise of the vitalAccessToken cookie is included.

Path	/permissions
Description	Expects the vitalAccessToken and vitalTestToken session cookies to be included in the request and returns some info about the user permissions for data access control. This information is in the form of values that the specific attribute of the documents to retrieve are allowed or denied to have. If the query parameter testCookie is set to true the info is related to the user of the vitalTestToken cookie, otherwise of the

	<i>vitalAccessToken</i> cookie.
--	--

POST endpoints

You may have to pass information in three different ways for POST methods:

1. As part of the path of the endpoint (e.g. <https://vital.com/endpoint/parameter>)
2. As cookies
3. As form parameters (included in the body with content type ***application/x-www-form-urlencoded***, e.g. `par1=value1&par2=value2`)

Path	<i>/user/create</i>
Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the following form parameters to be included in the request:</p> <ul style="list-style-type: none"> • <i>givenName</i>, the optional user first name • <i>surname</i>, the optional user last name • <i>name</i>, the mandatory username • <i>password</i>, the mandatory (8 characters or more) user password • <i>mail</i>, the optional user e-mail address <p>It returns some info about the created user.</p>

Path	<i>/user/delete</i>
Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the <i>name</i> form parameter (the user to delete) to be included in the request. Deletes the user.</p>

Path	<i>/group/create</i>
Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the <i>name</i> form parameter (the name of the group to create) to be included in the request. It returns some info about the created group.</p>

Path	<i>/group/delete</i>
Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the <i>name</i> form parameter (the group to delete) to be included in the request. Deletes the group.</p>

Path	<i>/group/{id}/addUser</i>
------	-----------------------------------

Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the <i>user</i> form parameter (the user to add to the group <i>id</i>) to be included in the request. It returns some info about the group updated with the added user.</p>
-------------	---

Path	<i>/group/{id}/delUser</i>
Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the <i>user</i> form parameter (the user to remove from the group <i>id</i>) to be included in the request. It returns some info about the group updated without the removed user (same format as <i>/addUser</i>).</p>

Path	<i>/policy/create</i>
Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the following form parameters to be included in the request:</p> <ul style="list-style-type: none"> • <i>name</i>, the name of the policy to create • <i>description</i>, a textual description of the policy to create • <i>appname</i>, the application name • <i>resources[]</i>, the array of resources the policy will affect • <i>groups[]</i>, the array of user groups the policy will affect • <i>actions[ACTION]</i>, boolean values specifying whether the ACTION (GET, POST, PUT, etc.) is allowed or denied <p>It returns some info about the created policy.</p>

Path	<i>/policy/delete</i>
Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the <i>name</i> form parameter (the policy to delete) to be included in the request. Deletes the policy.</p>

Path	<i>/application/create</i>
Description	<p>Expects the <i>vitalAccessToken</i> session cookie and the following form parameters to be included in the request:</p>

	<ul style="list-style-type: none"> • name, the name of the application to create • type, the name of the application type to use • description, some free text describing the application • resources[], the array of patterns for allowed resources in policies • actions[ACTION], specifying the default boolean value of the ACTION (GET, POST, PUT, etc.) <p>It returns some info about the created application.</p>
--	---

Path	/application/delete
Description	<p>Expects the vitalAccessToken session cookie and the name form parameter (the application to delete) to be included in the request. Deletes the application.</p>

Path	/user/{id}
Description	<p>Expects the vitalAccessToken session cookie and the following form parameters to be included in the request:</p> <ul style="list-style-type: none"> • "givenName", the updated user first name • "surname", the updated user last name • "mail", the updated user e-mail address • "status", "Active" or "Inactive" <p>It returns some info about the user identified by id with the updated fields (please refer to user info GET or creation for response format).</p>

Path	/user/changePassword
Description	<p>Expects the vitalAccessToken session cookie and the following form parameters to be included in the request:</p> <ul style="list-style-type: none"> • userpass, the new password • currpas, the old password <p>It sets the new password userpass for the user corresponding to the session of the vitalAccessToken cookie.</p>

Path	/policy/{id}
------	---------------------

Description	<p>Expects the vitalAccessToken session cookie and the following form parameters to be included in the request:</p> <ul style="list-style-type: none"> • description, the updated policy description • active, new policy status (false/true) • groups[], the updated list of groups to be affected by the policy • nogr, a boolean value which set to false allows to update without including the previous parameter (i.e. groups are not updated), while set to true means that if no group is specified then all groups are removed from the policy • resources[], the updated list of resources to be affected by the policy • nores, a boolean value which set to false allows to update without including the previous parameter (i.e. resources are not updated), while set to true means that if no resource is specified then all resources are removed from the policy • actions[ACTION], updated boolean values specifying if the ACTION (GET, POST, PUT, etc.) is allowed or denied • noact, a boolean value which set to false allows to update without including the previous parameter (i.e. actions are not updated), while set to true means that if no action is specified then all actions are removed from the policy (the policy will have no effect) <p>It returns some info about the policy identified by id with the updated fields (please refer to policy info GET or creation for response format).</p>
-------------	--

Path	/application/{id}
Description	<p>Expects the vitalAccessToken session cookie and the following form parameters to be included in the request:</p> <ul style="list-style-type: none"> • description, the updated application description • type, the updated application type • resources[], the updated list of patterns for resources allowed in policies • nores, a boolean value which set to false allows to update without including the previous parameter (i.e. patterns are not updated), while set to true means that if no pattern is specified then all patterns are removed from the application • actions[ACTION], updated default boolean

	<p>value for ACTION (GET, POST, PUT, etc.)</p> <ul style="list-style-type: none"> • noact, a boolean value which set to false allows to update without including the previous parameter (i.e. actions are not updated), while set to true means that if no action is specified then all actions are removed from the application <p>It returns some info about the application identified by id with the updated fields (please refer to application info GET or creation for response format).</p>
--	--

Path	/authenticate
Description	<p>Expects the following form parameters to be included in the request:</p> <ul style="list-style-type: none"> • name, username • password, user password • testCookie, if false the SSO vitalAccessToken cookie is returned, otherwise the alternative vitalTestToken cookie is included in the response. <p>It returns some info useful for session management.</p>

Path	/logout
Description	<p>Expects either the vitalAccessToken or the vitalTestToken session cookie to be included in the request and performs a logout (destroys the session identified by the cookie and resets the cookie in the response with an empty value); if the form parameter testCookie is set to true the module will use the vitalTestToken cookie, otherwise the vitalAccessToken cookie.</p>

Path	/evaluate
Description	<p>Expects both the vitalAccessToken and the vitalTestToken session cookies and the form parameter resources[] (resources for which user permissions are requested) to be included in the request. If the additional query parameter testCookie is set to true the "vitalAccessToken" cookie is considered corresponding to a user session with the rights to request a policy evaluation while the vitalTestToken</p>

	cookie to the user for whom the policies are to be evaluated, otherwise it is the opposite. It returns for each resource the list of permitted or denied actions.
--	---

Path	<i>/user/register</i>
Description	Expects the form parameter <i>mail</i> . It send an e-mail to the address specified in the parameter with a link to perform user self-registration.

Path	<i>/user/signup</i>
Description	<p>Expects the following form parameters to be included in the request:</p> <ul style="list-style-type: none"> • <i>givenName</i>, the optional user first name • <i>surname</i>, the optional user last name • <i>name</i>, the mandatory username • <i>password</i>, the mandatory (8 characters or more) user password • <i>mail</i>, the optional user e-mail address • <i>tokenId</i>, generated by the above endpoint and included in the e-mail • <i>confirmationId</i>, generated by the above endpoint and included in the e-mail <p>It returns some info about the created user (please refer to user info GET or creation for response format).</p>

The core security module may return three different status codes: 200, 400 or 500, depending on OpenAM errors or internal errors. If the error comes from OpenAM its description is forwarded to the management UI in the response body so that it can display the detailed info to the user. In some cases more specific codes are used directly in HTTP response status.

Possible status codes (from OpenAM) used are described in the following list.

Table 14: OpenAM error codes

200 OK	The request was successful and a resource returned, depending on the request. For example, a successful HTTP GET on /users/myUser returns a user profile and status code 200, whereas a successful HTTP DELETE returns {"success","true"} and status code 200.
201 Created	The request succeeded and the resource was created.
400 Bad Request	The request was malformed. Either parameters

	required by the action were missing, or incorrect data was sent in the payload for the action.
401 Unauthorized	The request requires user authentication. It can be reported if e.g. the required SSO Token value is missing.
403 Forbidden	Access was forbidden during an operation on a resource, for instance if a regular user tries to read the OpenAM administrator profile.
404 Not Found	The specified resource could not be found.
405 Method Not Allowed	The HTTP method is not allowed for the requested resource.
406 Not Acceptable	The request contains parameters that are not acceptable.
409 Conflict	The request would have resulted in a conflict with the current state of the resource.
410 Gone	The requested resource is no longer available, and will not become available again.
415 Unsupported Media Type	The request is in a format not supported by the requested resource for the requested.
500 Internal Server Error	The server encountered an unexpected condition which prevented it from fulfilling the request.
501 Not Implemented	The resource does not support the functionality required to fulfill the request.
503 Service Unavailable	The requested resource was temporarily unavailable. The service may have been disabled.

5.8 SLA Monitoring & Management Functionalities

SLA Monitoring and Management are part of the Governance Toolkit and will be reported in D5.3.2.

5.9 Reengineering and Migration toolkit

This toolkit is intended to facilitate reengineering processes and the migration of existing services to the VITAL platform, in order to make them available through the VITAL tools.

The implementation of the toolkit is based on components meant to ease the development of PPIs for different target platforms.

The first component, for Linux/Windows developers, is in the form of a **Maven Archetype**. This is a Maven project itself, but can also be distributed as a JAR package. In any case it can be used to generate new projects which contain a basic implementation to start the development from. This is meant for PPIs based on the **Java** language and deployable on a WildFly container (both technologies are used by VITAL modules and many PPIs developed by the consortium). The archetype produces a deployable PPI with all needed functionalities plus some guidelines (in the form of comments) for developers and integrators to realize what is needed to complete the module for their IoT system. In particular:

- Mandatory and some optional PPI endpoints (/metadata, /sensor/metadata, /sensor/observation, etc.) are already implemented, but return generic/empty responses by default.
- JSON parsers/formatters are provided with examples of use.
- JSON-LD examples for the formats used by VITAL are provided; thanks to the use of the “jsonschema2pojo” Maven plugin those result, upon building, in Java classes which can be used to read requests coming from the VITAL platform and to construct responses.
- A Java class (“IoTSystemClient”) is predefined and must be completed with methods to retrieve the needed pieces of information (data and metadata) from the IoT system and return them to be used to properly fill the responses of the PPI. A method is already defined for handling HTTP requests using the Apache HttpClient with timeouts and retries, if needed by the developer.
- The endpoints and code for monitoring (such as statistics about requests and errors) and the retrieval of performance metrics (based on the PPI and not the underlying IoT system) are already implemented and working.

The PPI developer will thus need to implement the methods to retrieve data from the services to be migrated and to complete the PPI endpoints code to fill in the responses with the retrieve data, thus performing the mapping with the VITAL entities.

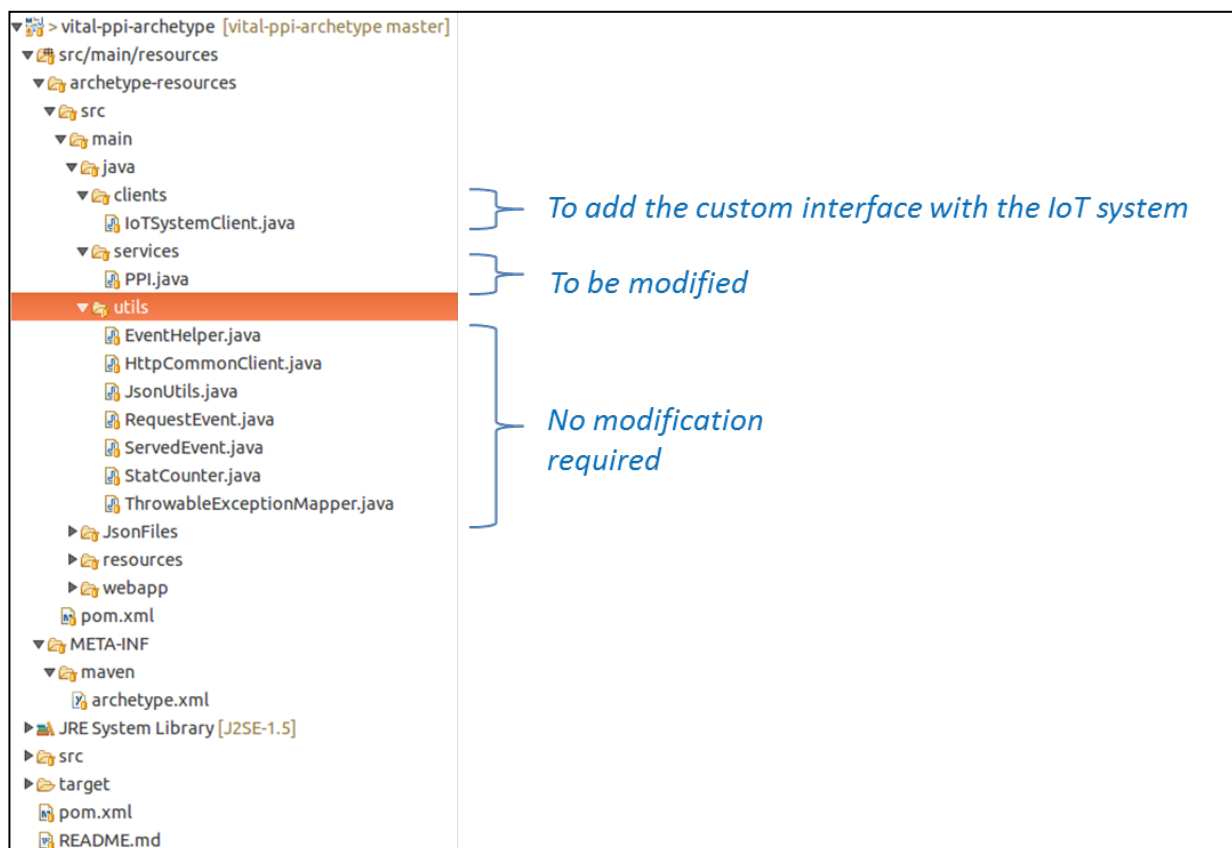


Figure 25 – Structure of the Maven Archetype

The second component of the Migration tooling is a template for the creation of an **Arduino** based PPI. This is a sketch providing a basic implementation of the mandatory PPI endpoints and it has been developed for the Arduino Mega 2560 with the WiFi Shield attached (needed in order to be able to expose the services).


As for the Maven Archetype, this requires the developer to configure and complete some parts; in particular:

- Configure the parameters of the Wi-Fi network used (or other configurations for other connectivity options may be needed if the WiFi Shield is not used)
- Configure the DDNS service - if needed
- Fill the endpoints responses with the right metadata and observations based on the type of sensor(s) attached to the board

Both components of the toolkit are available as repositories part of the VITAL open source release, named respectively “vital-ppi-archetype” and “vital-ppi-arduino-template”.

More modules like these might be developed and made available for different types of platforms.

Once a PPI is ready, in order to attach an IoT system to the platform, it must be registered to IoT Adapter through the web UI (shown in Figure 26).



Registered IoT systems				
These are the PPI-compliant IoT systems that are currently registered with VITAL.				
#	Name	URI		
1	Camden Footfall PPI	http://vital-integration.atosresearch.eu:8180/vital-ppi-camden-footfall	Last metadata refresh: 26.02.2016 13:22 Last data refresh: 29.02.2016 15:05	Refresh Edit Deregister
2	Istanbul Traffic PPI	https://vitalsp.cloud.reply.eu/vital/hireplyppi	Last metadata refresh: 26.02.2016 14:05 Last data refresh: 29.02.2016 15:00	Refresh Edit Deregister
3	London Air Quality PPI	http://vital-integration.atosresearch.eu:8180/vital-ppi-london-air-quality	Last metadata refresh: 29.02.2016 18:05 Last data refresh: 29.02.2016 18:30	Refresh Edit Deregister
4	CityBikes PPI	http://vital-integration.atosresearch.eu:8280/vital-ppi-citybikes	Last metadata refresh: 06.04.2016 11:09 Last data refresh: 06.04.2016 11:31	Refresh Edit Deregister
5	CityBikes [TO]BIKE PPI	http://vital-integration.atosresearch.eu:8280/vital-ppi-citybikes/to-bike	Last metadata refresh: 07.04.2016 10:59 Last data refresh: 06.04.2016 12:00	Refresh Edit Deregister
6	CityBikes Coca-Cola Zero® Bikes PPI	http://vital-integration.atosresearch.eu:8280/vital-ppi-citybikes/galway	Last metadata refresh: 06.04.2016 11:16 Last data refresh: 06.04.2016 12:01	Refresh Edit Deregister
7	CityBikes Veilb PPI	http://vital-integration.atosresearch.eu:8280/vital-ppi-citybikes/veilb	Last metadata refresh: 06.04.2016 11:18 Last data refresh: 06.04.2016 12:01	Refresh Edit Deregister

Register

Figure 26 – IoT adapter web interface

Once the system is registered to the platform the administrators will be able to visualize it in the management platform (Figure 27). All the sensors exposed by the PPI are added to the dedicated views of the management application, both in the form of list and as markers in the map (Figure 28).

7	CityBikes	CityBikes super-PPI (http://api.citybik.es/v2/)
8	Velib	CityBikes Velib network operated by JCDecaux
9	[TO]BIKE	CityBikes [TO]BIKE network operated by Comunicare S.r.l.
10	Coca-Cola Zero® Bikes	CityBikes Coca-Cola Zero® Bikes network operated by The National Transport Authority

Figure 27 – PPIs listed on the management application

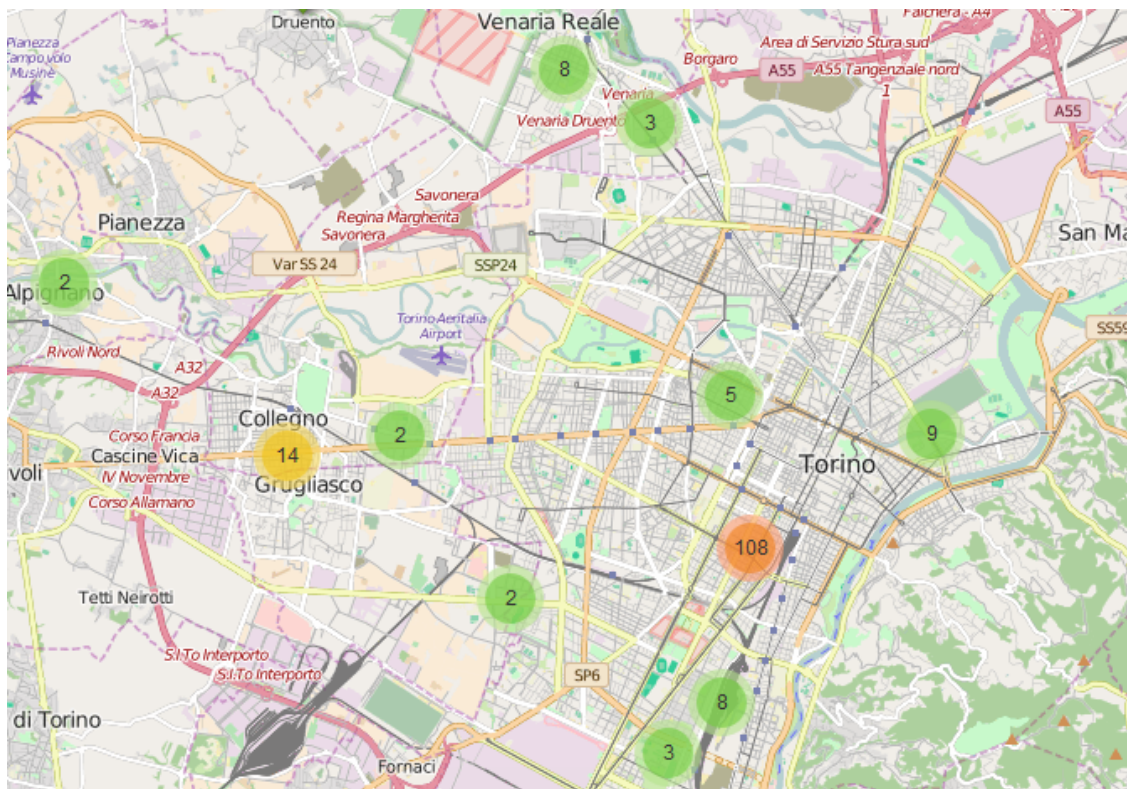


Figure 28 – Example of sensors shown on the map for Turin city

5.9.1 Case study: Integration of IoT data sources

To address the concern of the possibility of integrating single sensors with the VITAL platform and running PPIs on low power resource constrained devices, where Java Runtime Environment is not available, the following case study has been explored.

The first one is an environmental sensor (temperature and humidity) based on **Arduino**. This has also been described in D3.3.2 [Vital-D3.3.2-2016] has been developed together with the Migration toolkit described above. The sensor used is a DHT11, which is connected to the digital pins of the board. To complete the PPI first

the metadata and the structure of the endpoints responses had to be defined in the form of constant strings; then the library for the sensor has been imported and used to read the temperature and humidity values when requested and insert them in the observation endpoint response.

To be able to attach it to VITAL, the Arduino had to be configured to connect to a portable Wi-Fi hotspot, perform requests to a DDNS service and port forwarding has to be enabled on the router (the endpoints must be reachable from the outside).

5.9.2 Case study: Integration of IoT gateway

Beside the integration of single sensors reported in the above section, in the following we report the integration of an entire Sensor Network (SN) by using a gateway. In order to make this connection possible we use a **Raspberry** Pi 2 Model B that, from one side, communicates with the IoT Data Adapter module of the VITAL architecture and, on the other, acts as a “sink” for the SN.

Maxfor nodes sense the physical environment; they use a MSP430 as CPU and CC2420 as radio frequency module; moreover, they are integrated with temperature and light sensors.

The main tasks of the gateway are:

- Managing the sensor network.
- Formatting the information received from sensors nodes based on the VITAL ontology.
- Storing information into a local database.
- Exposing a PPI implementation via the web.

The PPI is developed in Java; to expose the PPI on the Raspberry Pi we used WildFly, while local data is stored by Elasticsearch.

5.9.3 Case study: Integration of smart cities services

Also reported in D3.3.2 is the test case of the CityBikes API, which exposes a big number of networks of bike sharing across the world. This is also the PPI that has been developed alongside the Maven Archetype of the Migration Toolkit.

Compared to the base provided by the Archetype it has the following main additions, custom for the CityBikes system:

- Java classes corresponding to the Json responses of the CityBikes API
- Two methods in the “IoTSystemClient” class to retrieve data for the two endpoints of the API:
 - <http://api.citybik.es/v2/networks>, which returns the list of networks together with some metadata
 - http://api.citybik.es/v2/networks/<network_id>, which returns metadata and observations for a specific network
- The missing code to properly fill the PPI response objects with the retrieve data

6 OUTLOOK AND CONCLUSIONS

The present document represents the third release of the VITAL management services for smart city environments over multiple IoT platforms, which has seen significant updates since the previous versions/releases of the deliverable. While many functionalities have been completed or extended (details in sections 1.1 and 5.2), what is maybe even more important is that the management platform is now fully dynamic and integrated with both security and all other modules of the VITAL platform, discovery, DMS, etc.

This version of the deliverable describes a fully working implementation of the VITAL management platform, which provides a wide range of functionalities/features including:

- A health map of a VITAL deployment, which provides a geographical representation of sensors and data feeds.
- Entity agnostic management widgets and dashboard, which visualize the VITAL ontologies.
- A set of functionalities for dynamic visual configuration of the main parameters of the platform.
- Security management functionalities, including management of users, groups, roles, permissions, sessions and more.
- Event processing functionalities based on the combination and correlation of data streams from different sources.
- Alerts visualization and management based on events (this is an work in progress extension of the platform functionalities)

Overall, this third release of the VITAL management platform has focused on extending functionalities, testing and improving interfaces between module and integration among the components.

The Management Platform will be extended in Task 5.4 – Smart Governance Toolkit with the governance related functionalities of VITAL. These include adding/removing systems, installation wide policies, SLA monitoring etc. These new features will be reported in D5.3.2.

This platform, as already identified as part of deliverable D7.4.1, is one of the main exploitable assets of the project. In particular, it will serve as a basis for building and commercializing semantically interoperable solutions for smart cities.

7 REFERENCES

- [Jin14] Jiong Jin, Gubbi, J.; Marusic, S. ; Palaniswami, M., « An Information Framework for Creating a Smart City Through Internet of Things», IEEE Internet of Things Journal, Volume:1 Issue:2, 2014.
- [Raman98] L. Raman, “OSI Systems and Network Management”, IEEE Communications Magazine, vol. 36, no. 3, IEEE Press, New York, 1998, pp. 46-53.
- [Vital-D2.3-2014] J. Soldatos, J. Kaldis, C. Georgoulis et al. “Virtualization Architecture and Technical Specifications”, June 2014.
- [Vital-D3.2.1-2014] J. Soldatos, K. Roukounaki et al. Vital Project, Deliverable D3.2.1, “Specification and Implementation of Virtualized Unified Access Interfaces V1”, December 2014.
- [Vital-D3.3.2-2016] P. Dal Zovo, L. Bracco et al. Vital Project, Deliverable D3.3.2, “Adaptation and Migration Mechanisms V1”, April 2016.
- [RiRu07] L. Richardson, S. Ruby. “RESTful Web Services”, O'Reilly May 2007.
- [KuRo14] Rafał Kuć, Marek Rogoziński, “Mastering Elasticsearch”, Packt, October 2014.
- [Burke13] Bill Burke, “RESTful Java with JAX-RS 2.0, 2nd Edition: Designing and Developing Distributed Web Services”, O'Reilly, November 2013.

Vital 2016