Ref. Ares(2020)4506691 - 31/08/2020

SEMIoTICS

Deliverable D5.8
IIoT Infrastructure set-up and testing (Cycle 2)

Deliverable release date	31/08/2020
Authors	1. Ermin Sakic, Darko Anicic (SAG), 2. Nikolaos Petroulakis, Eftychia Lakka, Emmanouil Michalodimitrakis (FORTH), 3. Luis Sanabria-Russo, Jordi Serra, David Pubill, Angelos Antonopoulos and Christos Verikoukis (CTTC), 4. Felix Klement, Korbinian Spielvogel, Henrich C. Pöhls (UP) 5. Prodromos-Vasileios Mekikis, Kostas Ramantas (IQU) 6. Konstantinos Fysarakis (STS), Manolis Chatzimpyrros, Michalis Smyrlis (STS) 7. Piotr Kowalski, Michał Rubaj, Łukasz Ciechomski, Jakub Rola (BLS)
Responsible person	Konstantinos Ramantas (IQU)
Reviewed by	1. Mirko Falchetto (ST) 2. Bartłomiej Lipa (BLS) 3. Jordi Serra (CTTC) 4. Emmanouil Michalodimitrakis (FORTH)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Christos Verikoukis) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Christos Verikoukis, Domenico Presenza, Danilo Pau, Joachim Posegga, Darek Dober, Kostas Ramantas, Ulrich Hansen)
Status of the Document	Final
Version	1.0
Dissemination level	Public

Table of Contents

1 INTRODUCTION

The Internet of Things (IoT) aims to connect everything and everyone, everywhere to everything and everyone else. It enables innovative applications for daily life activities, such as healthcare, industrial automation, smart city administration, etc. Due to the fact that the IoT paradigm is on the way to dominate in the aforementioned use cases, many issues need to be addressed to act in advance of the rapid developments. As an example, automating the networking and backend, improve interoperability among the devices and increase the security and privacy of the applications.

Regarding the networking and backend automation, SEMIoTICS adopts the SDN/NFV technologies that enable network abstraction. SDN seeks to separate network control functions from network forwarding functions, while NFV seeks to abstract network forwarding and other networking functions from the hardware on which it runs. Thus, both depend heavily on virtualization to enable network design and infrastructure to be abstracted in software and then implemented by underlying software across hardware platforms and devices. When SDN executes on an NFV infrastructure, SDN forwards data packets from one network device to another. At the same time, SDN's networking control functions for routing, policy definition and applications run in a virtual machine somewhere on the network. Thus, NFV provides basic networking functions, while SDN controls and orchestrates them for specific uses. The deployment of SDN and NFV under the SEMIoTICS framework together with the performance testing and KPI validation are described in Sections 3.1 and 3.2 of this deliverable, respectively.

Furthermore, interoperability is necessary to bridge the diverse technologies of sensors, actuators and communication hardware at the field layer through a gateway. Novel management frameworks for supporting the entire lifecycle of IoT applications in an automated manner are essential towards resource description and discovery, reservation and bootstrapping, interfacing, experimental control and monitoring. IoT orchestration and management can leverage already mature technologies, such as cloud computing and federated heterogeneous testbed facilities.

An important challenge that arises in this context is the exchange of information about the provided resources with their types and characteristics. Existing works rest upon certain interfaces and syntactic data models with arbitrary extensions and identifiers, which aggravate the management of heterogeneous resources across autonomous testbeds. To tackle this issue, it is proposed that management of the resources be based on their semantics, i.e. their underlying meaning and relations, while specific descriptions, data models and necessary interactions are abstracted. In other words, heterogeneous resources are described in a formalized manner to build a basis for their management. The deployment of semantic bootstrapping, interfacing, and interoperability under the SEMIoTICS framework together with the performance testing and KPI validation are described in Section 3.3 of this deliverable.

Although with the aforementioned technologies it is possible to provide a functional and high-performing IoT platform capable of delivering an outstanding performance in any critical use case, there is still an imperative issue that needs to be thoroughly investigated. As the digital transformation extends into business operations both online and in a world full of physical devices, securing the IoT cannot be an afterthought. IoT growth is helping to drive established digitization trends in mobile, cloud, and data with the ability to get better visibility and control over the physical world – for consumers as well as industrial applications. However, given the IoT-led exposure to the real world, many industries can become vulnerable to security threats creating a massive barrier to digital transformation. Recent analyses of security attacks show:

- >300% increase in malware loaded onto IoT devices (Source: Kaspersky Labs, New Trends in the World of IoT Threats 2018.)
- 600% Increase in IoT device attacks (Source: Symantec Internet Security Threat Report 2018.)

Without proper security in IoT, organizations risk lasting damage to brand reputation, as well as serious business consequences from data breaches and violations of governmental regulations and privacy policies. Thus, SEMIoTICS has proposed an advanced toolset that tackles security and privacy. Moreover, SPDI properties monitoring, in the context of the SEMIoTICS Pattern-driven SPDI monitoring and adaptation, as well as Service Function Chaining, aims to offer SPDI guarantees for the IoT deployments operation.

In this Cycle 2 deliverable we detail the functionality of the aforementioned SEMIoTICS components (already deployed during Cycle 1, as detailed in the first cycle deliverable D5.3), as well as their integration in an overarching testbed (see Section 2). This will be the first step of Task 5.3, with the second being the validation testing of the fully integrated components into a final testbed that delivers the promised advantages under the scope of SEMIoTICS.

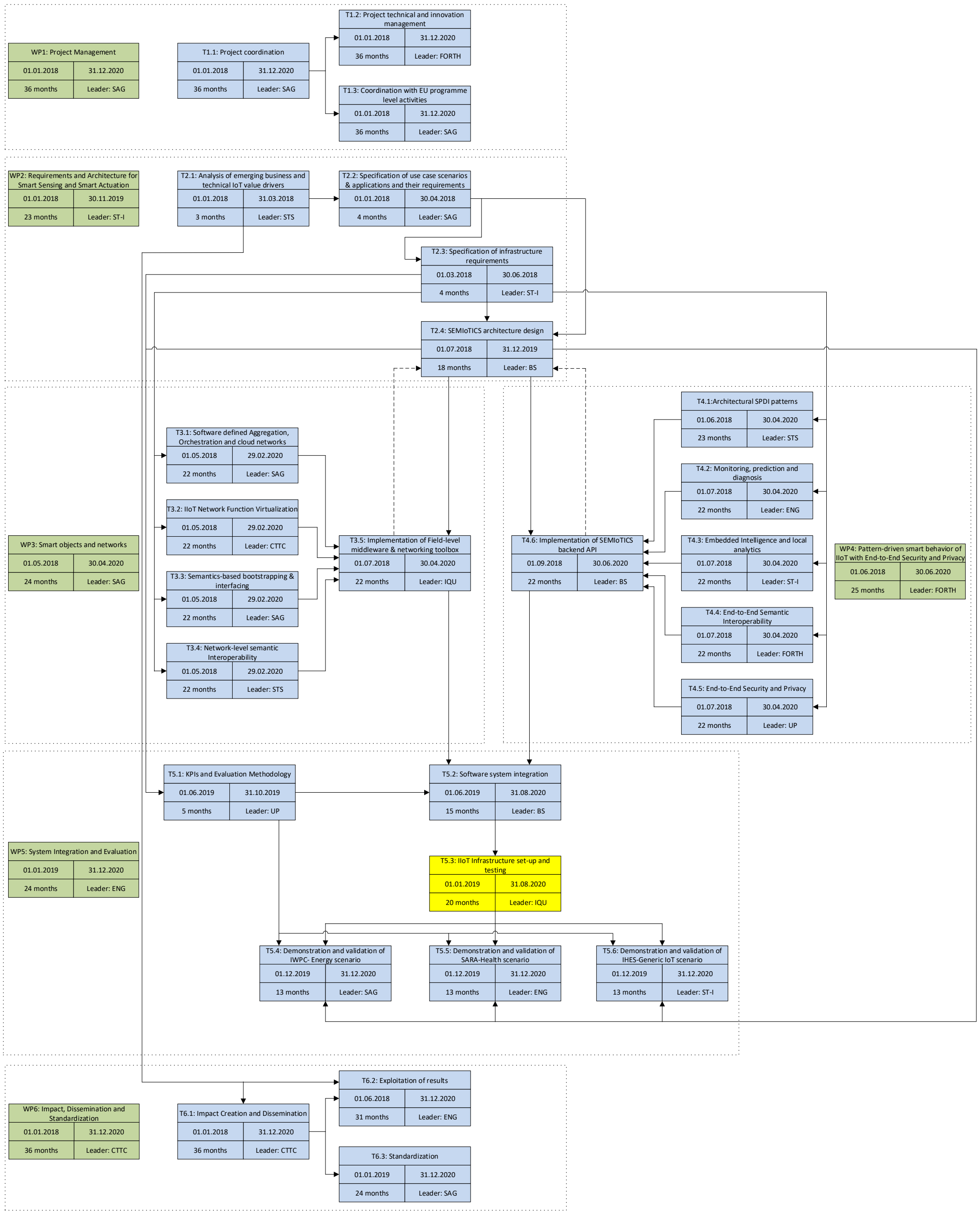
1.1 Updates in the second cycle deliverable

- Section 2.2.3 was added, detailing the integration of testbed components across partner locations through VPN access
- VPN access is showcased in Section 4.2 (UC2).
- Section 3.5 was added, with the Service Function Chaining (SFC) component, its architecture, testing and validation
- While in cycle 1 the pattern orchestrator is tested with a virtual network topology created using Mininet, in cycle 2, we focused on the interaction with the Field Pattern Engine and a physical layer testbed for distributed anomaly vibration monitoring.
- The SEMIoTICS GUI was updated, with new Screens included in section 3.4.4
- Validation Testing results were added throughout the deliverable, including end-to-end tests (see section 3.4.2.3), Service Instantiation results (3.2.3.1), tests related to the Kubernetes application orchestrator (Section 3.4.3.3) and the advanced Attribute Based Encryption (Section 3.4.1.3)

Further details on the Use case specific SEMIoTICS infrastructure set-up and testing can be found in the following documents:

- D5.4 and D5.9 Demonstration and validation of IWPC-Energy (Cycle 1 & 2)
- D5.5 and D5.10 Demonstration and validation of SARA-Health (Cycle 1 & 2)
- D5.6 and D5.11 Demonstration and validation of IHES-Generic IoT (Cycle 1 & 2)

1.2 PERT chart of SEMIoTICS tasks


T1.2: Project technical and innovation management
01.01.2018
31.12.2020
36 months
Leader: FORTH
WP1: Project Management
01.01.2018
31.12.2020
36 months
Leader: SAG
T1.1: Project coordination
01.01.2018
31.12.2020
36 months
Leader: SAG
T1.3: Coordination with EU programme level activities
01.01.2018
31.12.2020
36 months
Leader: SAG
WP2: Requirements and Architecture for Smart Sensing and Smart Actuation
01.01.2018
30.11.2019
23 months
Leader: ST-I
T2.1: Analysis of emerging business and technical IoT value drivers
01.01.2018
31.03.2018
3 months
Leader: STS
T2.2: Specification of use case scenarios & applications and their requirements
01.01.2018
30.04.2018
4 months
Leader: SAG
T2.3: Specification of infrastructure requirements
01.03.2018
30.06.2018
4 months
Leader: ST-I
T2.4: SEMIoTICS architecture design
01.07.2018
31.12.2019
18 months
Leader: BS
T4.1:Architectural SPDI patterns
01.06.2018
30.04.2020
23 months
Leader: STS
T3.1: Software defined Aggregation, Orchestration and cloud networks
01.05.2018
29.02.2020
22 months
Leader: SAG
T4.2: Monitoring, prediction and diagnosis
01.07.2018
30.04.2020
22 months
Leader: ENG
T3.2: IIoT Network Function Virtualization
01.05.2018
29.02.2020
22 months
Leader: CTTC
WP3: Smart objects and networks
01.05.2018
30.04.2020
24 months
Leader: SAG
T3.5: Implementation of Field-level middleware & networking toolbox
01.07.2018
30.04.2020
22 months
Leader: IQU
T4.6: Implementation of SEMIoTICS backend API
01.09.2018
30.06.2020
22 months
Leader: BS
T4.3: Embedded Intelligence and local analytics
01.07.2018
30.04.2020
22 months
Leader: ST-I
WP4: Pattern-driven smart behavior of IIoT with End-to-End Security and Privacy
01.06.2018
30.06.2020
25 months
Leader: FORTH
T3.3: Semantics-based bootstrapping & interfacing
01.05.2018
29.02.2020
22 months
Leader: SAG
T4.4: End-to-End Semantic Interoperability
01.07.2018
30.04.2020
22 months
Leader: FORTH
T3.4: Network-level semantic Interoperability
01.05.2018
29.02.2020
22 months
Leader: STS
T4.5: End-to-End Security and Privacy
01.07.2018
30.04.2020
22 months
Leader: UP
T5.1: KPIs and Evaluation Methodology
01.06.2019
31.10.2019
5 months
Leader: UP
T5.2: Software system integration
01.06.2019
31.08.2020
15 months
Leader: BS
WP5: System Integration and Evaluation
01.01.2019
31.12.2020
24 months
Leader: ENG
T5.3: IIoT Infrastructure set-up and testing
01.01.2019
31.08.2020
20 months
Leader: IQU
T5.4: Demonstration and validation of IWPC- Energy scenario
01.12.2019
31.12.2020
13 months
Leader: SAG
T5.5: Demonstration and validation of SARA-Health scenario
01.12.2019
31.12.2020
13 months
Leader: ENG
T5.6: Demonstration and validation of IHES-Generic IoT scenario
01.12.2019
31.12.2020
13 months
Leader: ST-I
T6.2: Exploitation of results
01.06.2018
31.12.2020
31 months
Leader: ENG
WP6: Impact, Dissemination and Standardization
01.01.2018
31.12.2020
36 months
Leader: CTTC
T6.1: Impact Creation and Dissemination
01.01.2018
31.12.2020
36 months
Leader: CTTC
T6.3: Standardization
01.01.2019
31.12.2020
24 months
Leader: SAG

Please note that the PERT chart is kept on task level for better readability.

2 SEMIOTICS OVERARCHING TESTBED

2.1 SEMIoTICS overarching testbed design

In this section, we present the overarching design of the SEMIoTICS testbed, being composed of a Backend layer, an SDN/NFV Orchestration layer, and a Field layer. Frameworks and APIs designed within WP3 and WP4, which include the NFV, SDN, Semantic Interoperability and Pattern Engine frameworks, are first deployed and evaluated in the SEMIoTICS testbed environment. Afterwards they can be leveraged by use-cases on extended versions of the testbed, to showcase more advanced scenarios. SEMIoTICS' use case applications are built in the form of IIoT services, or VNFs, related to smart monitoring and actuation that are managed autonomically by the SEMIoTICS infra0structure. Thus, functionalities such as establishing connectivity to a service, negotiating transport protocols and networking paths, as well as service scale-out and load balancing functions will be totally transparent for IoT applications. Moreover, they are handled by the respective frameworks of the SEMIoTICS infrastructure under the control of the Pattern Orchestrator (e.g., the networking policies are implemented by the SDN framework, Service policies by the NFV framework, etc.). In what follows, this deliverable contributes the testing methodology and preliminary test results for the main SEMIoTICS components of the overarching SEMIoTICS testbed. Furthermore, the IIoT infrastructures setup at partners' premises, that are involved in these tests, are also detailed as part of this deliverable.

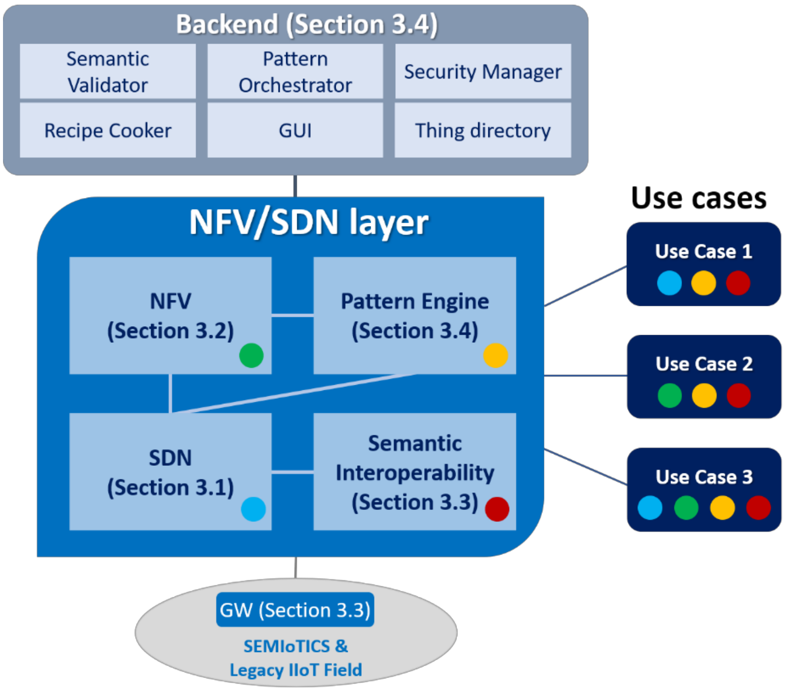


FIGURE 1: SEMIOTICS TESTBED OVERARCHING DESIGN

2.2 SEMIoTICS components integration

Alongside the IIoT infrastructures at partners' premises, a SEMIoTICS integration testbed is also under development, to integrate individual components after their successful validation and verification. The physical infrastructure of the SEMIoTICS integration testbed currently includes the following hardware components and is constantly upgraded:

- One 6-core 64-bit server with 32 GB RAM hosts the OpenStack Controller and Network services, related to Management, Orchestration and SDN control.
- One 4-core 64-bit server with 32 GB RAM hosts the ETSI OSM NFVO management services.

- Two 6-core 64-bit servers with 32 GB RAM act as the Compute Nodes, or Cloud hypervisors, that host all IIoT services and VNFs in dedicated Virtual Machines (VMs).
- One 4-core 64-bit server with 8 GB RAM acts as a resource constrained MEC node that hosts Edge VNFs.
- One Odroid C2 Single-Board Computer (SBCs) acts as the Field layer Virtualized IoT gateway. An 802.15.4 radio module is employed to interconnect Field devices (smart sensors) with the gateway.
- Field layer smart sensors that transmit temperature, humidity, light intensity and vibration values wirelessly over 802.15.4 and BLE. Smart Light actuators are also used for demonstration purposes.
- SDN access switches are employed at the Network layer, to implement the SDN Data plane.

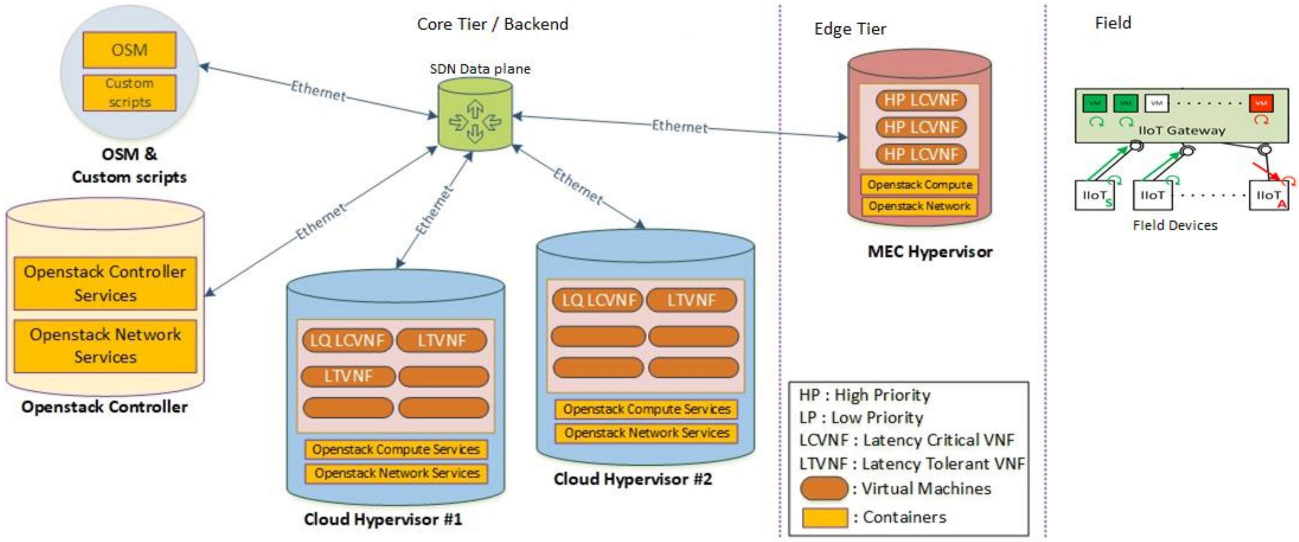


FIGURE 2: SEMIOTICS TESTBED PHYSICAL INFRASTRUCTURE

2.2.1 SDN/NFV ORCHESTRATION LAYER

The SEMIoTICS integration testbed leverages an OpenStack VIM, an OpenDaylight SDN controller, and an ETSI MANO stack. In this testbed, dedicated controller nodes host the VIM, SDN and MANO services in Containers. Containers is an emerging virtualization solution which allows services to run almost to the "bare metal" with minimal performance penalties, but with the requirement that they share the same kernel with the host (in this case the Controller node). IIoT services are implemented in the form of VNFs, that are managed by an ETSI compliant MANO stack, which handles the automatic deployment and lifecycle management of services, based on performance KPIs from a Telemetry system. Moreover, VNFs can be individually scaled, i.e., multiple instances can be deployed to meet user demand and migrated to a different hypervisor for optimization purposes. For example, to meet service KPIs, a VNF may have to be moved to a hypervisor with a lower CPU load, or to an Edge hypervisor to reduce latency. IIoT services are generally assigned dedicated Virtual Tenant Networks (i.e., VTNs) that are managed by the SDN Controller.

2.2.2 FIELD LAYER

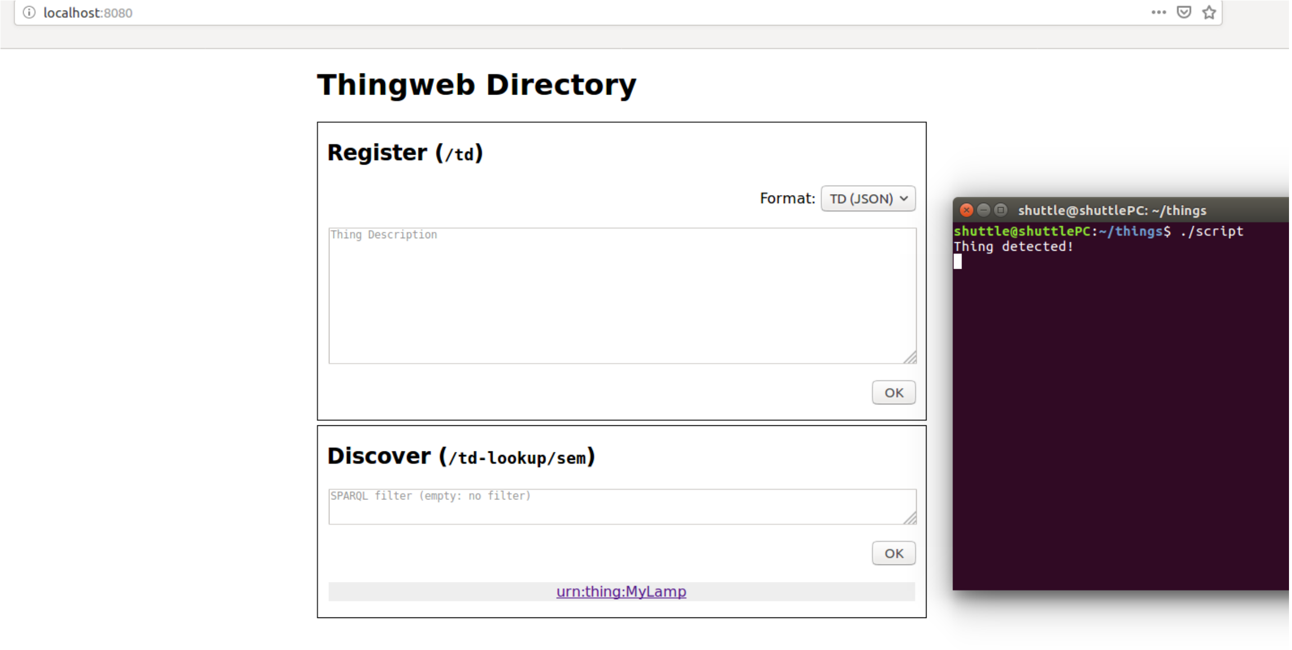
Our testbed Field layer includes a virtualized IIoT gateway that interconnects a set of sensors and actuators with the backend cloud. Our IoT gateway supports KVM virtualization, enabling us to push VNFs down to the gateway tier. This allows services with ultra-low latency requirements to be pushed in very close proximity to the IIoT devices, hence minimizing latency. For the field-layer smart sensors, we employ battery operated 802.15.4 and BLE devices that perform periodic measurement of CO2, Temperature, Vibration and Light (Lux) values. Sensor values are encapsulated in IPv6 packets and transmitted to the IIoT gateway via MQTT. The actuators are commercial Philips Hue Smart Lights that are connected to the IIoT gateway via a Hue bridge.

The Sensors and Actuators are communicating with the respective VNFs that are hosted at the Cloud or IIoT gateway hypervisors. Furthermore, the integration testbed leverages Semantic models, presented in Section 3.3, to annotate data that is exchanged between Things, as well as to describe capabilities of Things in a machine interpretable format. Our gateway serves as a semantic mediator in the task of integrating semantics of brownfield industrial devices and IoT things. More specifically, at the input, the gateway accepts data from diverse field devices. At the output, it provides an API to access semantically-annotated data along with descriptions of capabilities of connected devices. The API is based on the W3C WoT upcoming standard, and Things are specified in the WoT TD format. TD is semantically annotated with iot.schema.org, as it has been thoroughly described in Deliverable 3.3 and Section 3.3.

```
{
  "@context": [ "http://www.w3.org/ns/td",
              {"iot": "http://iotschema.org/"} ],
  "@type" : [
      "Thing", "iot:LightControl", "iot:BinarySwitchControl"
  ],
  "id": "urn:dev:wot:lamp",
  "name": "WirelessLamp",
  "description" : "WirelessLamp uses JSON-LD 1.1 serialization",
  "securityDefinitions": {
      "basic_sc": {"scheme": "basic", "in":"header"}
  },
  "security": ["basic_sc"],
  "properties": {
    "status" : {
      "@type" : "iot:SwitchStatus",
      "type": "string",
      "forms": [{
         "href": mqtt://192.168.1.11:1883/house/lamp/status,
         "mediaType": "application/json"}]
    }
  },
  "actions": {
     "toggle" : {
     "@type" : "iot:ToggleAction",
     "forms": [{
        "href": mqtt://192.168.1.11:1883/house/lamp/toggle,
        "mediaType": "application/json"}]
     }
  },
  "events":{
    "overheating":{
      "@type" : "iot:TemperatureAlarm",
      "data": {"type": "string"},
      "forms": [{
          "href": "mqtt://192.168.1.11:1883/house/lamp/oh",
          "subprotocol": "longpoll"
      }]
    }
  }
}
```

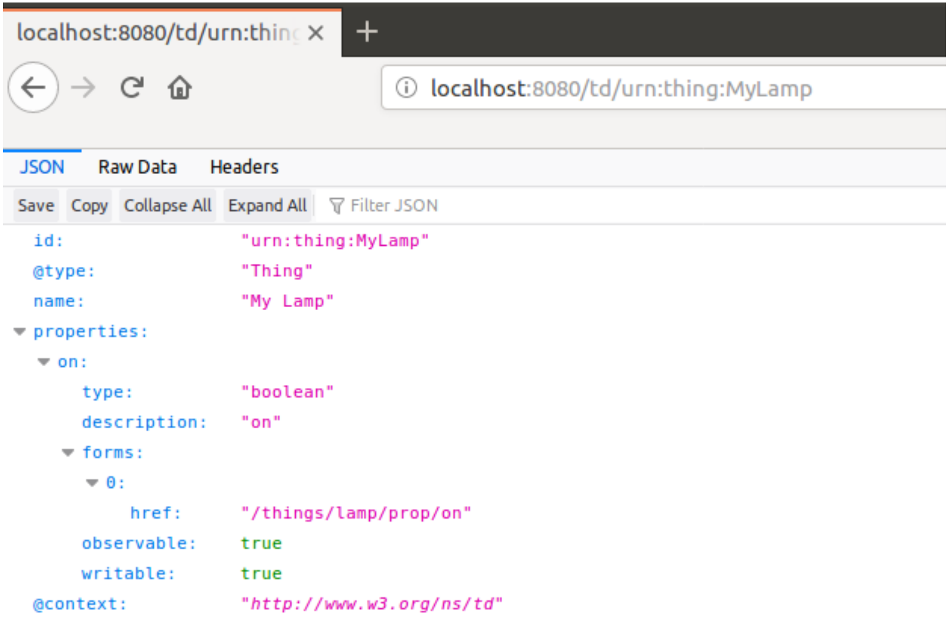
FIGURE 3: THING DESCRIPTION ANNOTATED WITH IOT.SCHEMA.ORG

For verification purposes, in our testbed, we deployed the Smart Light as a Thing that is automatically registered in the database with the reception of an MQTT availability message, as soon as it connects to the network. In detail, a listener at the IIoT gateway receives the availability MQTT message "ON" and retrieves the Thing Description from the local database, as seen in FIGURE 3. The result of the discovery is shown in the Thingweb Directory immediately, as seen in FIGURE 4. Thus, the TD is registered at the Thing Directory that allows searching for a Thing based on its metadata, properties, actions or events. In FIGURE 5, we show the JSON format of the TD and the address that it has been given to the Thing by the Thingweb directory. Through this platform it is also possible to update the TD and even generate a servient based on a discovered Thing.



```
shuttle@shuttlePC:~/things$ ./script
Thing detected!
```

FIGURE 4: THING DISCOVERY



```
id:                    "urn:thing:MyLamp"
@type:                 "Thing"
name:                  "My Lamp"
▼ properties:
  ▼ on:
      type:            "boolean"
      description:     "on"
    ▼ forms:
      ▼ 0:
          href:        "/things/lamp/prop/on"
      observable:      true
      writable:        true
@context:              "http://www.w3.org/ns/td"
```

FIGURE 5: TD OF THE WIRELESS SMART LIGHT

2.2.3 INTEGRATION WITH EXTERNAL COMPONENTS THROUGH VPN ACCESS

The integration of the integration testbed with external SEMIoTICS components (e.g., field layer components that are specific to use-cases), requires that the testbed grants access to its functionalities to external users, services or components. To this end, a VPN service has been enabled. Thereby, in this subsection we will perform the tests to show that external users' can access the NFV testbed along with the NFV functionalities. This is of paramount importance in e.g. SEMIoTICS' SARA Health use case, as the Pattern Orchestrator interacts remotely with the NFV component. Also, in this regard, the IoT GW forwards traffic to the VNFs in the testbed through the VPN that we explain next.

An OpenVPN server (referred to as Router/Firewall in **FIGURE 6**) was setup, and credentials were generated and shared with the SEMIoTICS integrator partner. The topology allowing such workflow is shown in **FIGURE 6**.

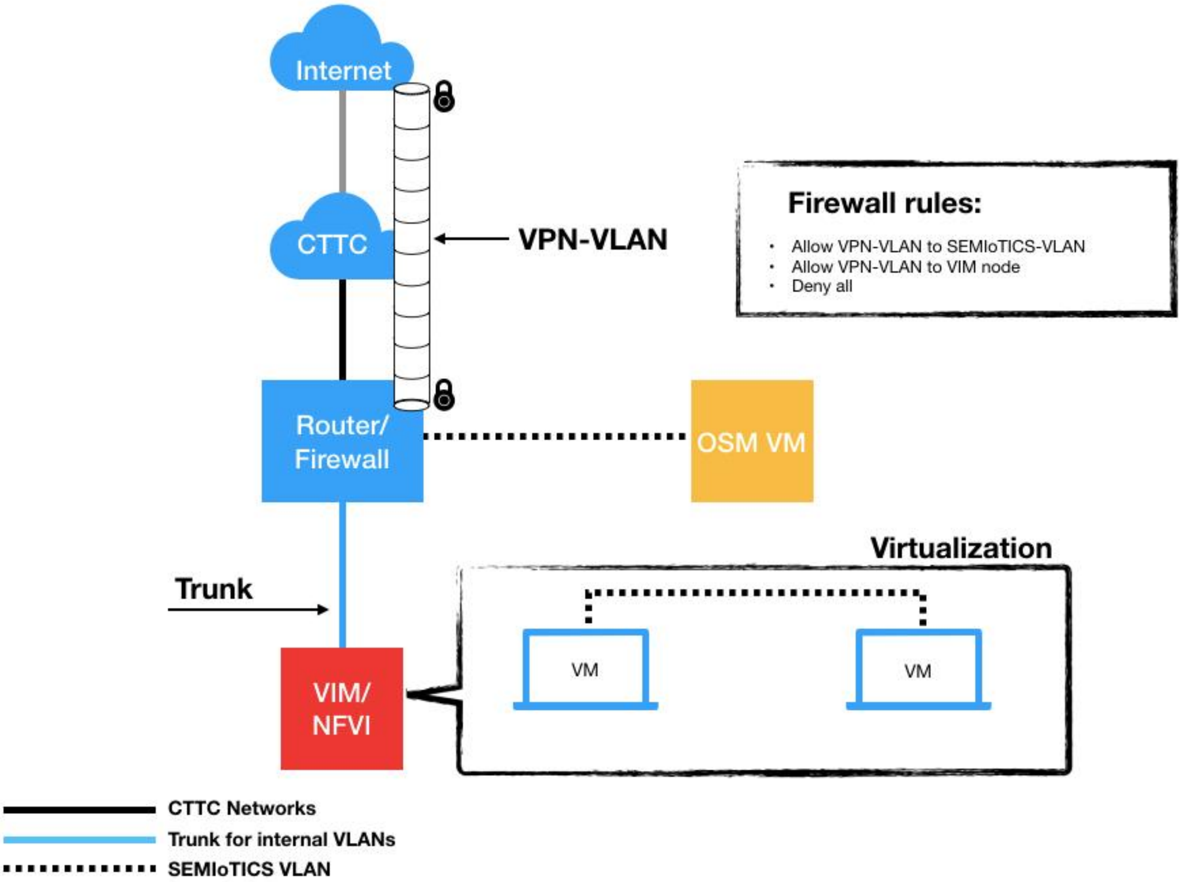
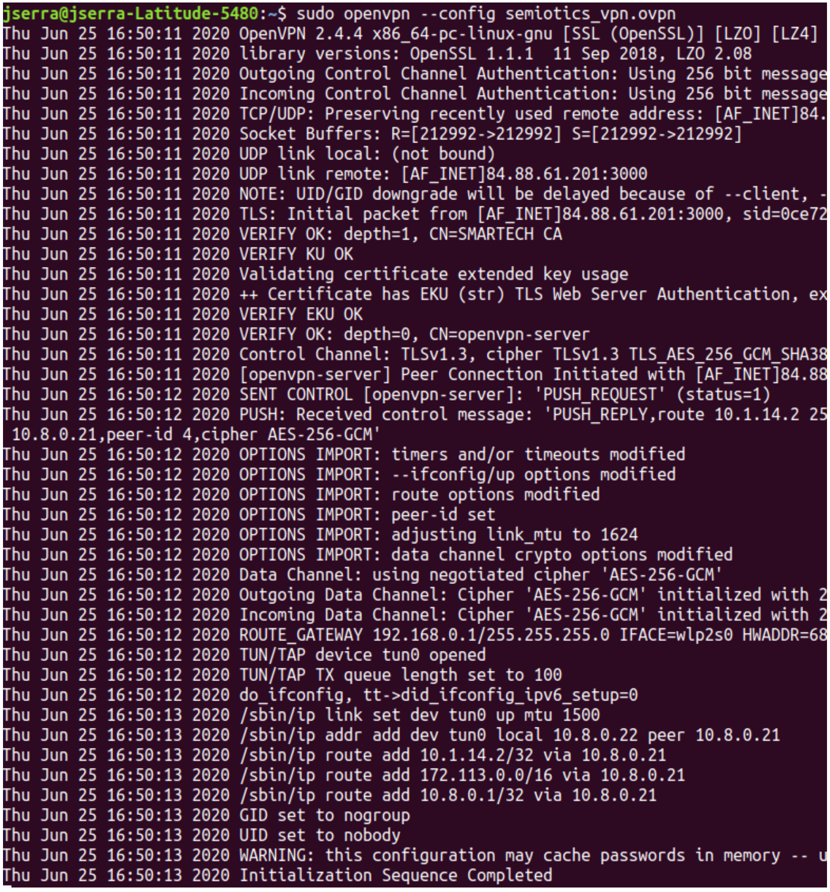


FIGURE 6 OPENVPN TUNNEL TO ALLOW INTEGRATION WITH EXTERNAL COMPONENTS

As can be seen from FIGURE 7, partners may either use their own NFVO and register the local VIM, or SSH through the VPN-VLAN towards OSM VM to use the local NFVO and therefore the whole local SEMIoTICS' NFV Component. IP connectivity to the orchestrated VNFs is also guaranteed.

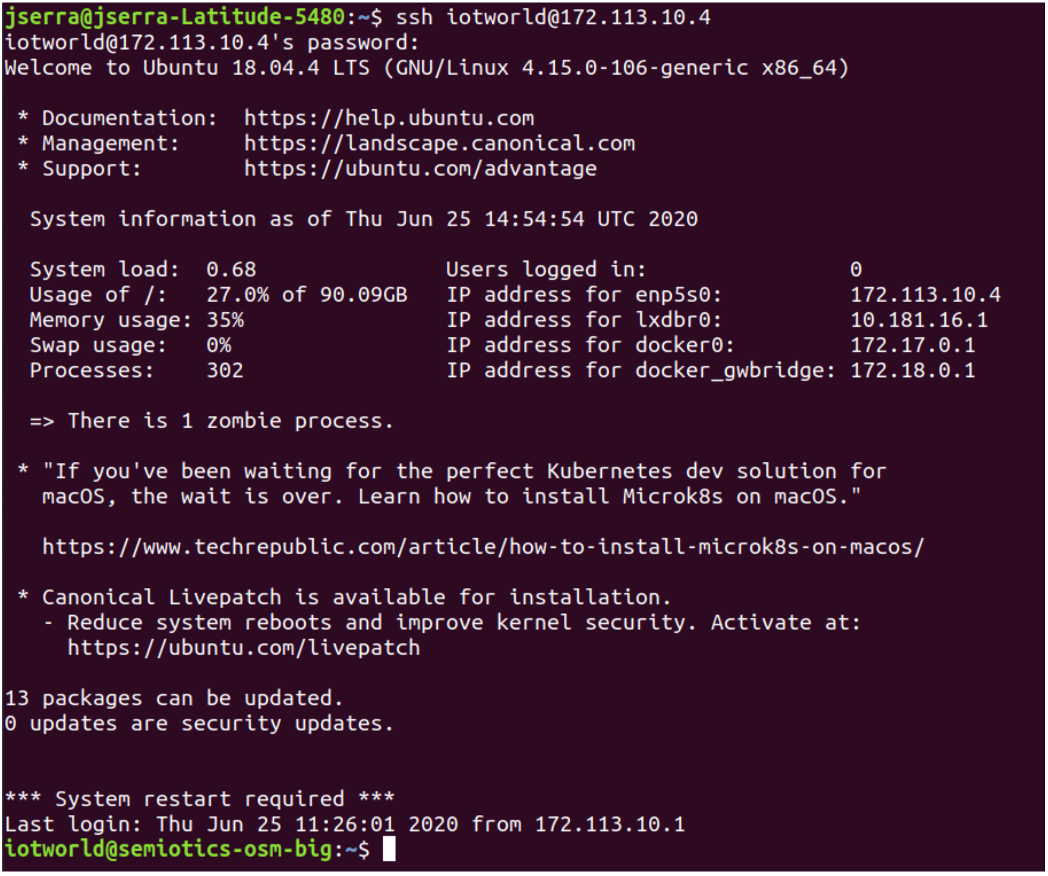
Next, we provide the tests that show that the external users can access the NFV platform and its services by connecting through the VPN, i.e. via the VPN and SSH they will use the whole NFV components located at the CTTC premises. Thereby, first in FIGURE 8FIGURE 7 we show that the execution of the VPN connection instruction, based on a configuration file that sets the VPN restriction rules, is successful.



```
jserra@jserra-Latitude-5480:~$ sudo openvpn --config semiotics_vpn.ovpn
Thu Jun 25 16:50:11 2020 OpenVPN 2.4.4 x86_64-pc-linux-gnu [SSL (OpenSSL)] [LZO] [LZ4]
Thu Jun 25 16:50:11 2020 library versions: OpenSSL 1.1.1  11 Sep 2018, LZO 2.08
Thu Jun 25 16:50:11 2020 Outgoing Control Channel Authentication: Using 256 bit message
Thu Jun 25 16:50:11 2020 Incoming Control Channel Authentication: Using 256 bit message
Thu Jun 25 16:50:11 2020 TCP/UDP: Preserving recently used remote address: [AF_INET]84.
Thu Jun 25 16:50:11 2020 Socket Buffers: R=[212992->212992] S=[212992->212992]
Thu Jun 25 16:50:11 2020 UDP link local: (not bound)
Thu Jun 25 16:50:11 2020 UDP link remote: [AF_INET]84.88.61.201:3000
Thu Jun 25 16:50:11 2020 NOTE: UID/GID downgrade will be delayed because of --client, -
Thu Jun 25 16:50:11 2020 TLS: Initial packet from [AF_INET]84.88.61.201:3000, sid=0ce72
Thu Jun 25 16:50:11 2020 VERIFY OK: depth=1, CN=SMARTECH CA
Thu Jun 25 16:50:11 2020 VERIFY KU OK
Thu Jun 25 16:50:11 2020 Validating certificate extended key usage
Thu Jun 25 16:50:11 2020 ++ Certificate has EKU (str) TLS Web Server Authentication, ex
Thu Jun 25 16:50:11 2020 VERIFY EKU OK
Thu Jun 25 16:50:11 2020 VERIFY OK: depth=0, CN=openvpn-server
Thu Jun 25 16:50:11 2020 Control Channel: TLSv1.3, cipher TLSv1.3 TLS_AES_256_GCM_SHA38
Thu Jun 25 16:50:11 2020 [openvpn-server] Peer Connection Initiated with [AF_INET]84.88
Thu Jun 25 16:50:12 2020 SENT CONTROL [openvpn-server]: 'PUSH_REQUEST' (status=1)
Thu Jun 25 16:50:12 2020 PUSH: Received control message: 'PUSH_REPLY,route 10.1.14.2 25
 10.8.0.21,peer-id 4,cipher AES-256-GCM'
Thu Jun 25 16:50:12 2020 OPTIONS IMPORT: timers and/or timeouts modified
Thu Jun 25 16:50:12 2020 OPTIONS IMPORT: --ifconfig/up options modified
Thu Jun 25 16:50:12 2020 OPTIONS IMPORT: route options modified
Thu Jun 25 16:50:12 2020 OPTIONS IMPORT: peer-id set
Thu Jun 25 16:50:12 2020 OPTIONS IMPORT: adjusting link_mtu to 1624
Thu Jun 25 16:50:12 2020 OPTIONS IMPORT: data channel crypto options modified
Thu Jun 25 16:50:12 2020 Data Channel: using negotiated cipher 'AES-256-GCM'
Thu Jun 25 16:50:12 2020 Outgoing Data Channel: Cipher 'AES-256-GCM' initialized with 2
Thu Jun 25 16:50:12 2020 Incoming Data Channel: Cipher 'AES-256-GCM' initialized with 2
Thu Jun 25 16:50:12 2020 ROUTE_GATEWAY 192.168.0.1/255.255.255.0 IFACE=wlp2s0 HWADDR=68
Thu Jun 25 16:50:12 2020 TUN/TAP device tun0 opened
Thu Jun 25 16:50:12 2020 TUN/TAP TX queue length set to 100
Thu Jun 25 16:50:12 2020 do_ifconfig, tt->did_ifconfig_ipv6_setup=0
Thu Jun 25 16:50:13 2020 /sbin/ip link set dev tun0 up mtu 1500
Thu Jun 25 16:50:13 2020 /sbin/ip addr add dev tun0 local 10.8.0.22 peer 10.8.0.21
Thu Jun 25 16:50:13 2020 /sbin/ip route add 10.1.14.2/32 via 10.8.0.21
Thu Jun 25 16:50:13 2020 /sbin/ip route add 172.113.0.0/16 via 10.8.0.21
Thu Jun 25 16:50:13 2020 /sbin/ip route add 10.8.0.1/32 via 10.8.0.21
Thu Jun 25 16:50:13 2020 GID set to nogroup
Thu Jun 25 16:50:13 2020 UID set to nobody
Thu Jun 25 16:50:13 2020 WARNING: this configuration may cache passwords in memory -- u
Thu Jun 25 16:50:13 2020 Initialization Sequence Completed
```

FIGURE 7 OPENVPN TUNNEL TO ALLOW INTEGRATION WITH THE NFV COMPONENT

To this end, we use the OSM command line interface (cli). Thereby, first it is shown in that the external users can enter via ssh to the VM that hosts OSM (see FIGURE 8).



```
jserra@jserra-Latitude-5480:~$ ssh iotworld@172.113.10.4
iotworld@172.113.10.4's password:
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-106-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Thu Jun 25 14:54:54 UTC 2020

  System load:  0.68               Users logged in:              0
  Usage of /:   27.0% of 90.09GB   IP address for enp5s0:        172.113.10.4
  Memory usage: 35%                IP address for lxdbr0:        10.181.16.1
  Swap usage:   0%                 IP address for docker0:       172.17.0.1
  Processes:    302                IP address for docker_gwbridge: 172.18.0.1

  => There is 1 zombie process.

 * "If you've been waiting for the perfect Kubernetes dev solution for
   macOS, the wait is over. Learn how to install Microk8s on macOS."

   https://www.techrepublic.com/article/how-to-install-microk8s-on-macos/

 * Canonical Livepatch is available for installation.
   - Reduce system reboots and improve kernel security. Activate at:
     https://ubuntu.com/livepatch

13 packages can be updated.
0 updates are security updates.


*** System restart required ***
Last login: Thu Jun 25 11:26:01 2020 from 172.113.10.1
iotworld@semiotics-osm-big:~$
```

.

FIGURE 8 SSH THROUGH THE VPN TO THE VM THAT CONTAINS THE OSM.

3 SEMIOTICS TESTBED COMPONENTS

This is the section of the deliverable that describes the SEMIoTICS components employed at the overarching testbed. After a short description of the component architecture (more details can be found in the respective deliverables of WP3 and WP4), we provide the testing methodology to introduce the reader to the methods that will be employed to undertake the performance evaluation and KPI validation.

3.1 SEMIoTICS Software Defined Networking Controller (SSC)

3.1.1 COMPONENT ARCHITECTURE

SEMIoTICS SDN Controller comprises the architectural components contained as depicted in FIGURE 9. Each of the depicted components was implemented by the time of writing the deliverable, with the majority of components to be used / showcased in SEMIoTICS Use Cases 1 and Use Case 2 (excluding Clustering Manager only, which is to be showcased in a lab environment).

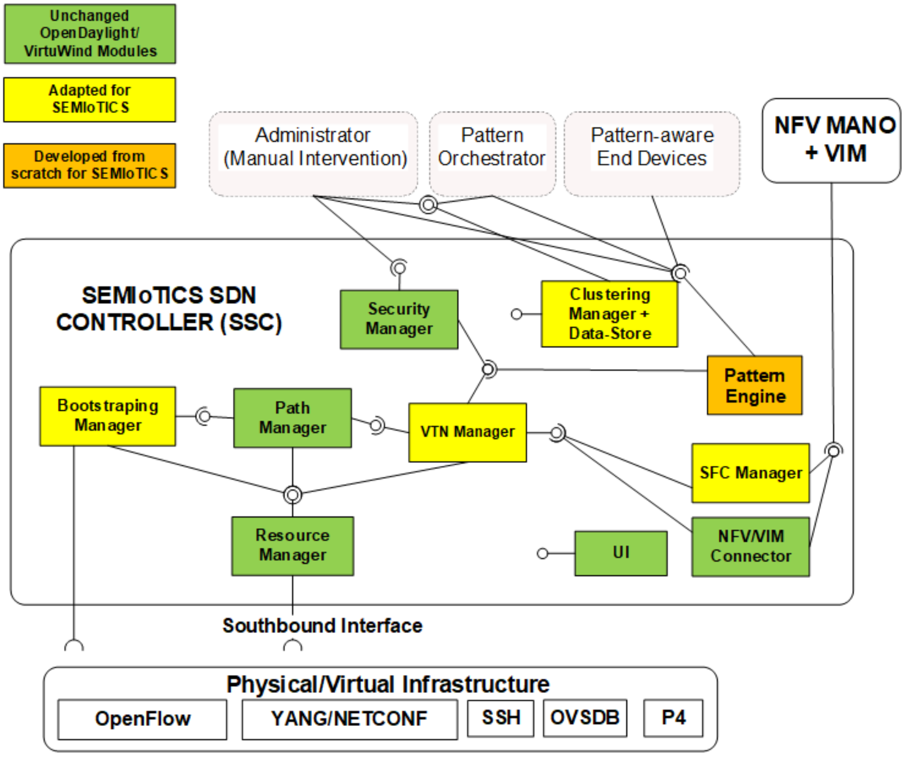

Unchanged OpenDaylight/ VirtuWind Modules
Adapted for SEMIoTICS
Developed from scratch for SEMIoTICS
Administrator (Manual Intervention)
Pattern Orchestrator
Pattern-aware End Devices
NFV MANO + VIM
SEMIoTICS SDN CONTROLLER (SSC)
Security Manager
Clustering Manager + Data-Store
Pattern Engine
Bootstraping Manager
Path Manager
VTN Manager
SFC Manager
Resource Manager
UI
NFV/VIM Connector
Southbound Interface
Physical/Virtual Infrastructure
OpenFlow
YANG/NETCONF
SSH
OVSDB
P4

FIGURE 9: SSC ARCHITECTURE AS PER TASK 3.1 (D3.1)

To test the basic function and evaluate the general performance of functions relevant for the two Use Cases, we focus here on the realization of an exemplary scenario resembling the final Use Case 1 prototype - an integration of the IoT Gateway with corresponding end-points and an exemplary field layer application. We specifically evaluate the Bootstrapping Manager, Pattern Engine, VTN Manager, Path Manager and Resource Manager's operation. Using the interaction between these components, infrastructural services and QoS-enabled end-to-end path configurations are installed with minimal user intervention – apart from the request specification in Pattern Engine.

Note: In the integration tests discussed below, we do not rely on the external Pattern Orchestration instance to feed the SSC with QoS connectivity requests. Instead we rely on requests defined in a few exemplary scripts. To support the complete Use Case 1 workflow, Pattern Orchestrator will be integrated in the final UC1 demonstrator with the related measurement to be documented in D5.9.

3.1.2 TESTING METHODOLOGY

3.1.2.1 INTEGRATION METHODOLOGY

To validate the correctness of implemented modules during implementation, we have deployed an emulated testbed comprising of an arbitrary number of Docker containers hosting individual Open vSwitch instances with OpenFlow 1.3.1 support. Thus, basic functionality such as VTN addition and removal, as well as addition and removal of individual connectivity pattern instances is achieved using unit tests without interaction with the physical setup.

However, the Use Case 1 will rely on deployment of physically attached end-devices and should depict the correct SSC operation when deployed against physical network switching equipment. To this end, we have deployed a 6-switch physical network comprised of 6x OpenFlow 1.3.1 switch instances, deployed using the kernel-space forwarding daemon of Open vSwitch and the corresponding OpenFlow agent. The switch instances are executed on 1 Gbps Banana PI R1 hardware devices. Each switch is equipped with 4 physical RJ45 interfaces, as shown on devices (encircled by green box) in FIGURE 10.

Additionally, we deployed an LTE router attached to switch OF:104, acting as the gateway to backend layer services. The SSC is deployed on the SIMATIC IPC427E industrial PC running Linux, equipped with a recent Intel i7 Processor and 16 GB of DDR4 RAM.

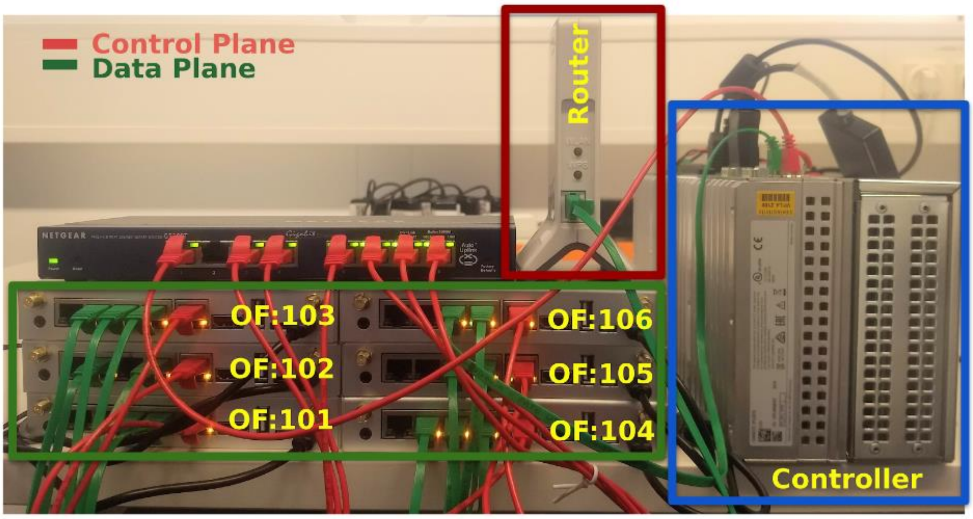


FIGURE 10: INITIAL NETWORKING DEPLOYMENT TOWARDS USE CASE 1 REALIZATION

To evaluate an initial implementation of the SSC, we have compiled and installed its components based on code check-out from December, 2019, with the IoT Gateway solution developed in D3.3 and evaluate the IoT Gateway application providing semantic mediation for inter-connected sensor (camera) and actuator devices (streamer device). An extension of the application demonstrated here is intended for the Use Case 1 deployment in its final form, i.e., interconnecting greenfield sensory devices and a brownfield programmable logic controller (PLC). Nevertheless, discovery of grease leakage based on camera-taken photos, processed in a remote unit, will be shown in the Use Case 1 as well.

To evaluate intermediate integration of current subset of IoT Gateway components (i.e., the Semantic Edge Platform and Semantic API & Protocol binding) we deployed: a) an IP-enabled camera as envisioned end-point sensor; b) the combination of Semantic Edge Platform based on RED-Node and semantic API bindings to expose data using the WoT Thing description interface; and c) the internet gateway used in bootstrapping of the RED-Node framework. Additional details on the tested applications are discussed in Section 3.3.

3.1.2.2 SYSTEM BOOTSTRAPPING

After initial bootup of the devices, the switches must first discover the SSC. We assume the switches are configured with IP address of the SSC and the port which SSC is listening on for new OpenFlow connections. Indeed, the controller/port configurations in the Open vSwitch implementation persists after reboots, hence a single-time configuration was necessary to achieve this functionality. Bootstrapping Manager can alternatively deploy the DHCP server and provide the switches with automatically derived IPv4 addresses but we assume manual IPv4 configuration of management interfaces of the switches for lowered complexity of the demonstration.

Both switches and the SSC's management interfaces are thus configured in the same IP subnet to omit the routing. After the switches have initiated an OpenFlow session to the controller, the Bootstrapping Manager initiates the bootstrapping procedure in the newly switches, i.e., provisioning them with default flow rules used for in-band control channel communication. FIGURE 11 showcases the SSC's UI containing resulting discovered topology (including deployed endpoints). Access to the UI and all other REST-enabled functions of the SSC require HTTPS digest authentication.

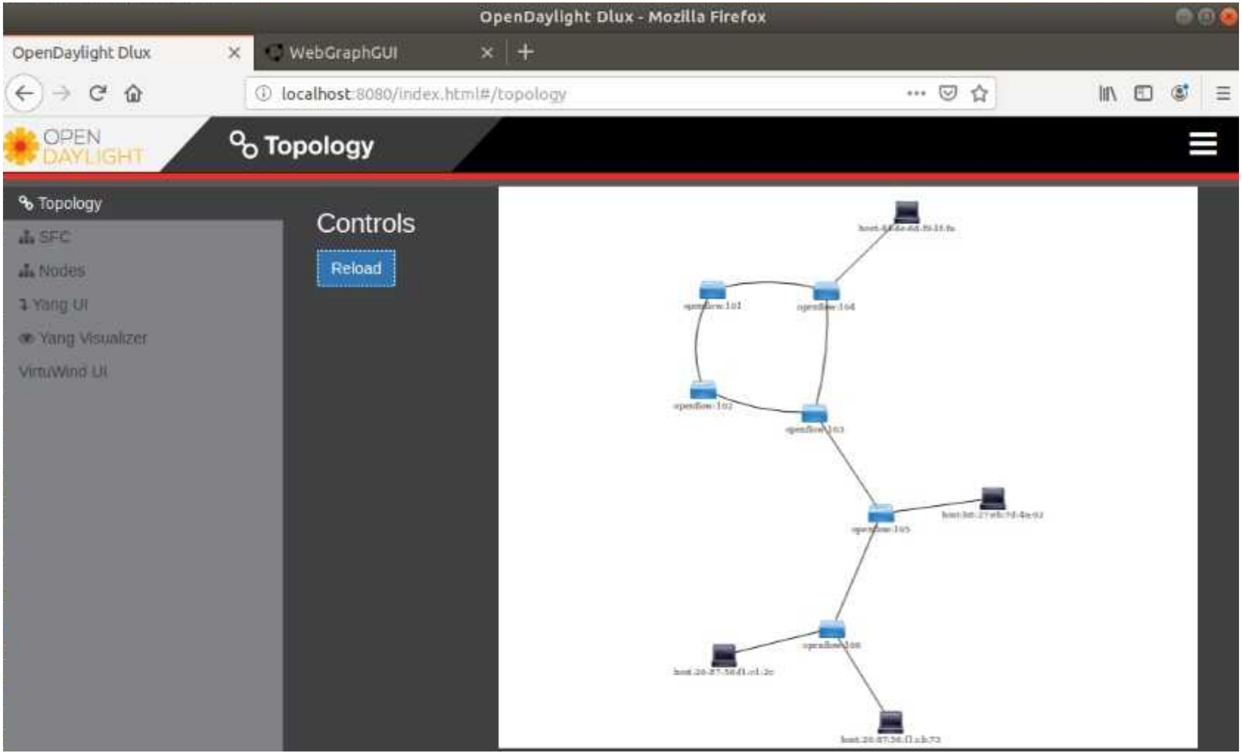


FIGURE 11: The discovered topology and end-devices in SSC after connecting three end-points, booting up the switches and their successful establishment of OpenFlow 1.3.1 sessions with the SSC.

The performance tests related to the total bootstrapping time are presented in Section below.

3.1.2.3 VIRTUAL TENANT NETWORK INSTANTIATION

To interconnect the end-points, we require a Virtual Tenant Network enabling the connectivity of all admitted components of the application: a) an IP-enabled camera; b) the RED-Node based service capability discovery framework and c) the internet gateway used in bootstrapping of the RED-Node framework.

The formulation of the VTN script used in the integration scenario accordingly comprises three according interface definitions, a definition of a shared virtual bridge and the tenant definition. The exemplary VTN establishment script used in the test scenario is as follows (comments contained inline):

```
# VTN Parameters
tenant_name=vtn1
bridge_name=vbr1
interface1=1
interface2=2
interface3=3

#physical mapping
node_1=openflow:104
nodeport_1=1

node_2=openflow:106
nodeport_2=2

node_3=openflow:105
nodeport_3=3

# Creating a VTN tenant.
curl --user "admin":"admin" -H "Content-type: application/json" -X POST \
http://localhost:8181/restconf/operations/vtn:update-vtn -d \
'{
    "input":{
        "tenant-name":"'$tenant_name'"
    }
}'
echo ' '
sleep 0.2

# Creating a VTN bridge.
curl --user "admin":"admin" -H "Content-type: application/json" -X POST \
http://localhost:8181/restconf/operations/vtn-vbridge:update-vbridge -d \
'{
    "input":{
        "tenant-name":"'$tenant_name'",
        "bridge-name":"'$bridge_name'"
    }
}'
echo ' '
sleep 0.2

➔   Repeat for each interface ….

# Creating a vInterface, adding it to a bridge/tenant mapping.
curl --user "admin":"admin" -H "Content-type: application/json" -X POST \
http://localhost:8181/restconf/operations/vtn-vinterface:update-vinterface -d \
'{
    "input":{
        "tenant-name":"'$tenant_name'",
        "bridge-name":"'$bridge_name'",
        "interface-name":"'$interface1'"
    }
}'
sleep 0.2

➔   Repeat for each interface ….

# Mapping a vInterface to a port.
curl --user "admin":"admin" -H "Content-type: application/json" -X POST \
http://localhost:8181/restconf/operations/vtn-port-map:set-port-map -d \
'{
    "input":{
        "tenant-name":"'$tenant_name'",
        "bridge-name":"'$bridge_name'",
        "interface-name":"'$interface1'",
        "node":"'$node_1'",
        "port-id":"'$nodeport_1'"
    }
}'
sleep 0.2
```

The addition of the above VTN is expected to occur once per system deployment, hence the related manual effort is minimal. The establishment of the VTN results in creation of default point to point flows. Using these, the camera and Semantic Edge Platform are capable of exchanging data, i.e., exposing camera's data via the protocol binding API to the external apps.

3.1.2.4 QOS-ENABLED E2E FLOW INSTANTIATION

To establish the QoS-enabled connectivity between declared end-points of the evaluated test scenario, we rely on instantiation of the according pattern instances using the SSC's Pattern Engine. The Pattern Engine interacts with the VTN Manager to evaluate the mapping of end-points to the specified VTN, and if satisfied, notifies the Path Manager module of the path request. The Path Manager then computes the corresponding path fulfilling the QoS constraints and notifies the Resource Manager of the flow rules, as well as the associated queue mapping for each hop on the computed path. Thus, an end-to-end path can be established with deterministic queueing defined individually for each hop on the path. Resource Manager finally installs the flow rules and the end-point connectivity is enabled.

The corresponding QoS pattern instantiation script used in integration testing of a QoS-enabled service path interconnecting the IP camera and IoT Gateway comprises the following content:

```
curl --user "admin":"admin" -H "Content-type: application/json" -X POST \
http://127.0.0.1:8181/restconf/operations/patternengine:addFact -d \
'{
    "input": {
        "recipe_id": "recipe1",
        "fact_id": "factid17",
        "fact_from": "orchestrator",
                "fact_message": "17 True 100 10 1542 44:4e:6d:f9:1f:fa f0:79:59:27:4d:a1 10.50.50.254 10.50.50.2 0",
        "fact_type": "qospathrequest"
    }
}'
echo ' '
sleep 0.2
```

The request properties are contained in the following line "17 True 100 10 1542 44:4e:6d:f9:1f:fa f0:79:59:27:4d:a1 10.50.50.254 10.50.50.2 0" and comprise (in order of appearance):

- The request identifier
- The requirement for bi-directionality of installed paths
- The required bandwidth share in Kbps
- The requested end-to-end delay requirement in milliseconds
- The input traffic burst in max. Kbps
- The source MAC address
- The destination MAC Address
- The source IP address
- The destination IP address
- The requirement for resilient path establishment

After establishment of the above flow, the referenced end-points with given MAC addresses as identifiers in the flow rules in network are guaranteed the requested QoS requirements (bandwidth and delay). Given that the input traffic arrivals are shaped as per promised maximal traffic burst and sending rate and do not exceed the requested rate.

3.1.3 PERFORMANCE TEST AND KPI VALIDATION

We briefly summarize the initial results taken to bootstrap depicted network (which i.e., is also to be used in the final Use Case 1 demonstrator), as well as the time taken to establish the end-to-end path installations with provided QoS guarantees.

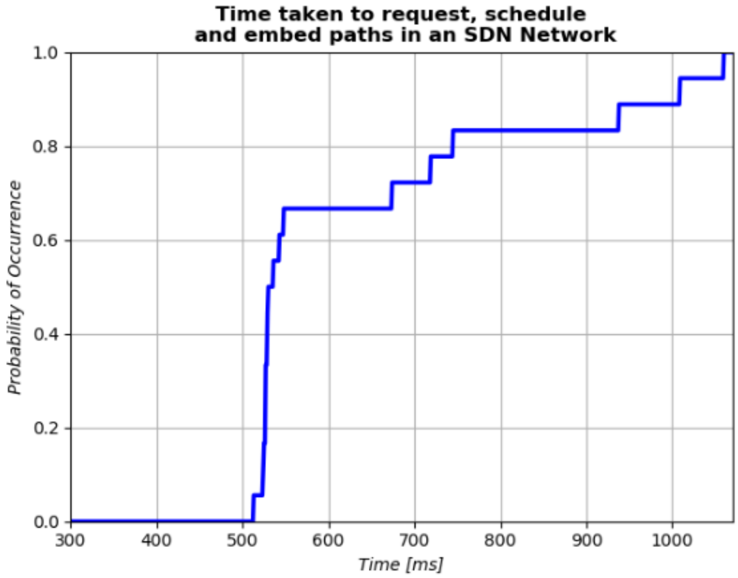


FIGURE 12: OBSERVED PATH COMPUTATION AND INSTALLATION TIME

The ECDF in FIGURE 12 depicts the elapsed time taken by the SDN controller to schedule and/or embed a point - to-point communication service with QoS guarantees. Such a service can be used to serve any generic point-to-point communication relationship in Use Cases 1 and 2. It comprises the time to accept and parse the request, compute the QoS - constrained path inside the Path Manager component, as well as to implement and validate the installation flow rules using the Resource Manager.

The ECDF depicts the probability of embedding the flow rules under a given time -constraint. The portrayed embedding is based on a relatively limited sample size of 20 flow embeddings but should serve as a good representative of the expected controller performance.

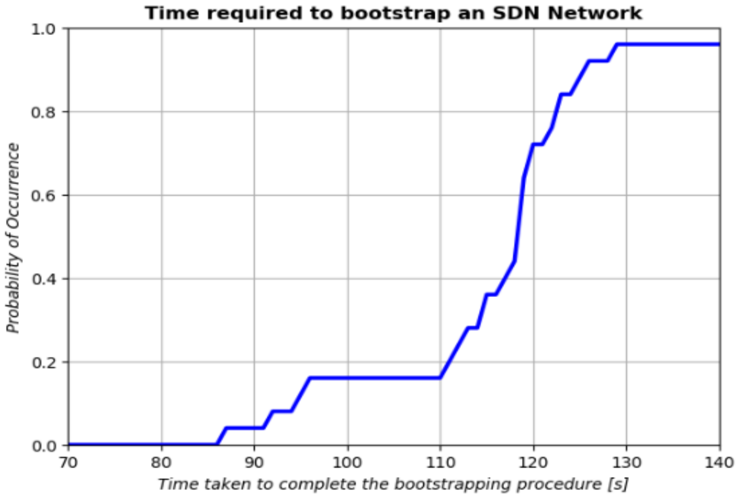


FIGURE 13: OBSERVED NETWORK BOOTSTRAPPING TIME

FIGURE 13 depicts the ECDF of experimental measurements of the total time duration taken to bootstrap the control plane in the Use Case 1 topology in in-band manner. The bootstrapping time comprises the following phases:

1. OpenFlow session establishment and initial rule installation to support LLDP, OpenFlow, SSH, ARP packet propagation
2. Procedure of disabling the (R)STP in safe manner on each OpenFlow switch using SSH channel
3. Procedure of re-routing the control flows for purpose of enabling resilience.

While further tuning of timeouts related to, in particular, phase two of bootstrapping, could improve the total waiting period, the portrayed bootstrapping time is sufficient for industrial networks, that rarely, if ever, experience a total shutdown or reboot following the initial deployment.

The switches are connected to the SSC, and configured one by-one (i.e. the switch closest to the controller is configured first, then the neighboring switch and subsequently all other switches as well). Each switch will try to connect to the SDN controller, independent of the state of connection of its neighbor. In the case of a failure (connect rejection), the interval between two consecutive attempts will increase exponentially until it reaches a maximum value.

3.1.4 RELATION TO NETWORKING REQUIREMENTS

With the above shown connectivity and bootstrapping implementation, we demonstrate the implementation of requirements (mostly revolving around providing the VTN network instantiation, QoS connectivity for interacting flows and network bootstrapping):
R.GP.1, R.GP.3, R.GP.6, R.S.2, R.S.77, R.NL.8, R.BC.12, R.NL.9, R.BC.13, R.NL.1, R.NL.2, R.NL.3, R.NL.4, R.NL.7/R.BC.10, R.UC1.1, R.UC1.3, R.UC1.4, R.UC2.3, R.UC2.15, R.UC2.17

In the remainder of project, with implementation of Use Case 1 Cycle 1 and Cycle 2 demonstrators, we additionally implement, achieve and demonstrate the following requirements:
R.GP.4, R.GP.5, R.GP.7,

Finally, due to missing relation to Use Cases, we do not address the following requirements in Use Cases due to considerable efforts associated with required system development. Nevertheless, we will showcase the initial results to handling these requirements in the lab environment:
R.GP.2 / R.UC1.7, R.UC1.5 & R.UC1.6 (only in lab environment)

3.2 Network Function Virtualization (NFV)

3.2.1 COMPONENT ARCHITECTURE

The SEMIoTICS NFV Component encompasses the NFV Management and Orchestration (NFV MANO) and NFV Infrastructure (NFVI), as defined in FIGURE 14. That is, the set of controllers and managers (NFV MANO), and the set of hardware used for virtualizing network functions (also referred to as compute nodes[1]) at all layers of the SEMIoTICS architecture, respectively.

Together, SDN and NFV are able to realize customizable isolated network environments, where processing endpoints (i.e. VNFs) are dynamically instantiated at precise compute nodes in the SEMIoTICS architecture. Network traffic is then directed towards such VNFs, which may be standalone or part of a custom Service Function Chain (SFC) to reach a desired endpoint, be processed or consumed. In this section, SEMIoTICS

[1] The reference to compute nodes encompasses all hardware capable of providing virtual compute, network and storage resources to the NFV VIM, and therefore are included in the NFVI.

NFV Component is described based on its current implementation on the project. Furthermore, sequence diagrams for common procedures, APIs and their relation to other SEMIoTICS components are overviewed. To conclude, we present:

i) a testing methodology for evaluating the deployment of the component
ii) present a methodology for its eventual integration with other SEMIoTICS off-location components
iii) summarizes the evaluation results through implementation.

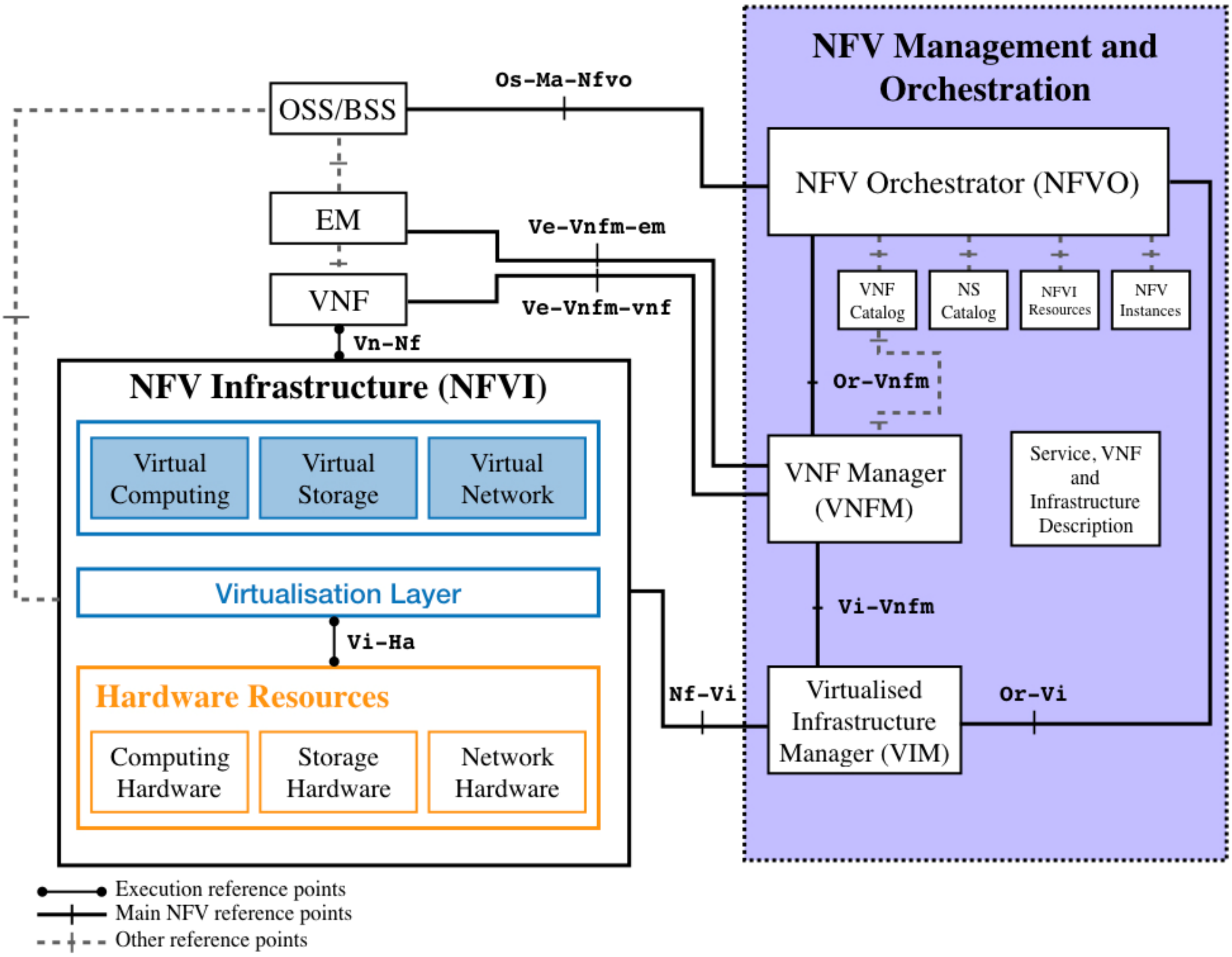


FIGURE 14 ETSI NFV REFERENCE ARCHITECTURE

3.2.1.1 NFV COMPONENT AS HARDWARE

As mentioned in the introduction of this section, here the components defined in FIGURE 14 are mapped to an example network topology. For exemplifying purposes, the network topology describing SEMIoTICS' Use Case 3 (UC 3) will be used, see FIGURE 15. References to the SEMIoTICS architecture and FIGURE 14 will be made, as well as highlights to common NFV Components' APIs.

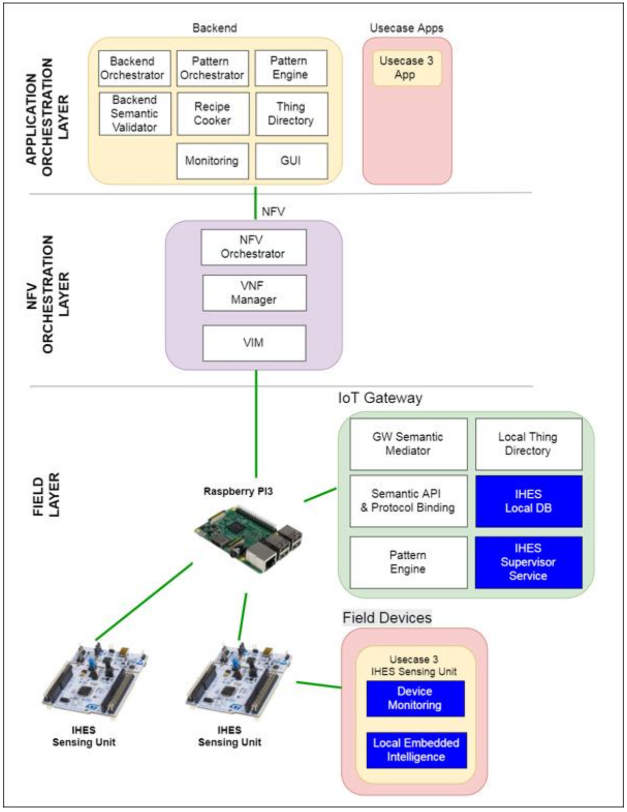


FIGURE 15 USE CASE 3 SYSTEM ARCHITECTURE (FROM D2.4)

Going down an abstraction layer that is from FIGURE 14 to hardware, requires a real use case. For that reason, UC 3 system architecture (shown in FIGURE 15) will serve as a starting point. In it, it is possible to identify SEMIoTICS Architecture's three layers, as well as the components involved in the realization of the UC.

In FIGURE 15, the Field Layer is composed of two type of devices: Field Devices, and IoT Gateway. The formers are sensors equipped with an embedded intelligence component and not virtualization-capable. The latter are indeed virtualization-capable nodes, which are able to run all or some of their services as VNFs. Going up SEMIoTICS System Architecture, the NFV Orchestration Layer summarizes the collection of NFV MANO elements involved in the UC. Finally, the Application Orchestration Layer hosts the backend VNFs supporting the UC.

From the aforementioned overview it is easy to highlight the domain of the NFV Component, i.e. virtualization-capable nodes (NFVI). The following FIGURE 16 maps UC 3 elements and virtualization requirements[2] to a preliminary network topology describing the NFVI of such UC as defined in FIGURE 15. FIGURE 16 also highlights the elements of the SEMIoTICS architecture that would fall within the NFV Component's domain (denoted here as NFVI). Starting from the Field layer, the IoT Gateway hosts the VNFs that enable its functionality in said UCs. Other services at the backend are realized as VNFs at the Application Orchestration Layer. Notice that this layer also hosts NFV MANO and the SEMIoTICS SDN Controller (SSC). FIGURE 16 serves as a reference network topology, mainly because it considers the Field and Application Orchestration Layers for VNF instantiation.

[2] Network, compute, storage.

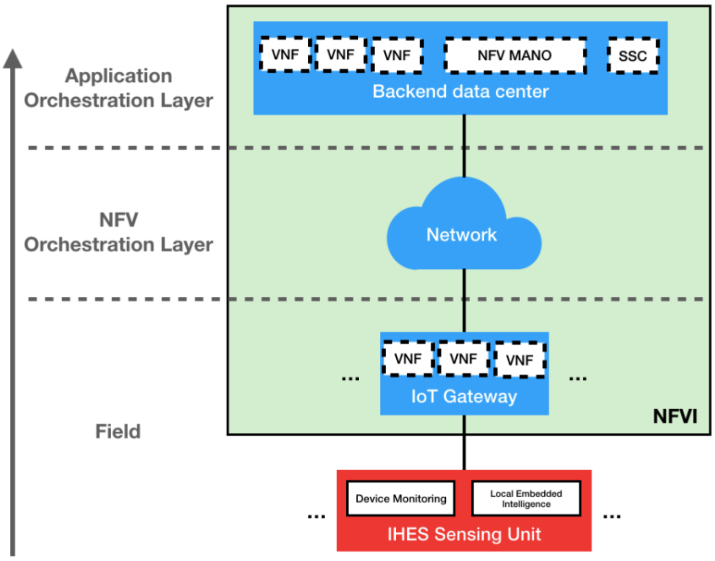


FIGURE 16 ENVISIONED NFVI SUPPORTING UC3

3.2.1.2 NFV OPERATIONS

NFVI is evidenced in FIGURE 16. Virtualization-capable nodes throughout the SEMIoTICS Architecture permit the orchestration of VNFs from a centralized location (i.e. NFV MANO). UC owners, or the Global Pattern Orchestrator may trigger NFV Component's APIs to: onboard descriptors, orchestration, and retrieve NFV telemetry.

- Descriptor onboarding: as described in D3.2, for the NFV Orchestrator (NFVO) to orchestrate a Network Service it requires a blueprint, or descriptor to be loaded (onboarded). Before onboarding, authorized components such as the Pattern Orchestrator may modify the descriptors in order for the to-be-orchestrated NS to comply with a specific pattern or constraint[3]. FIGURE 17 shows a message sequence diagram of a VNF descriptor onboarding.

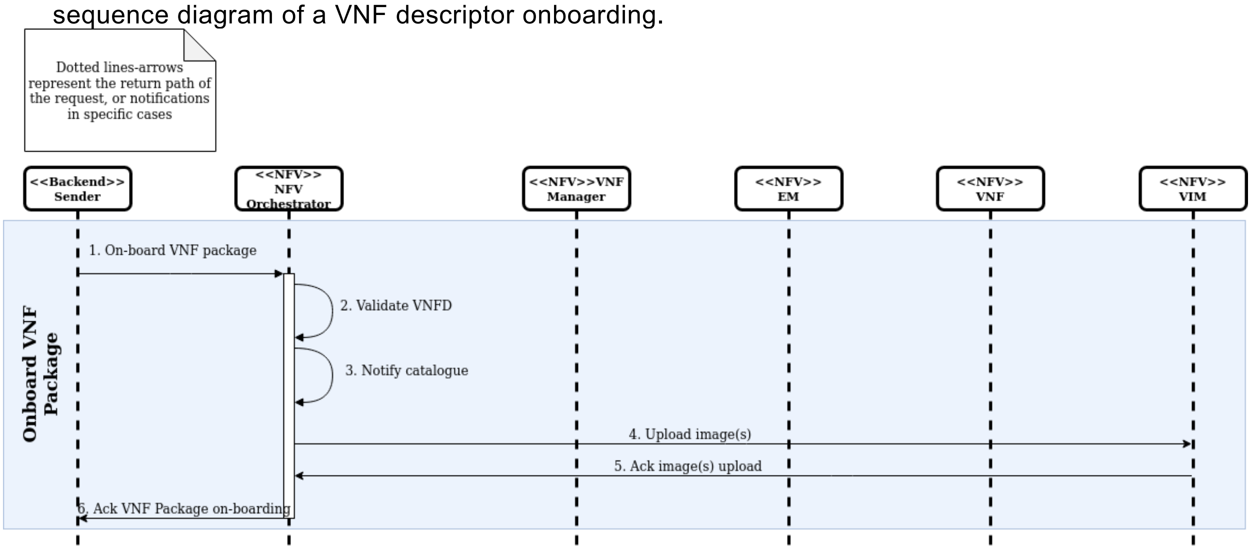


FIGURE 17 VNF DESCRIPTOR ONBOARDING

[3] For ways to modify Network Services, refer to Section 4.7 in D3.2.

- Orchestration: once descriptors are onboarded, NFVO, via its Service and Resource Orchestration functions gather available resources from the VIM and then schedules the instantiation/creation of the required services specified in the descriptors. FIGURE 18, from D3.2, shows the message sequence diagram of a VNF orchestration.

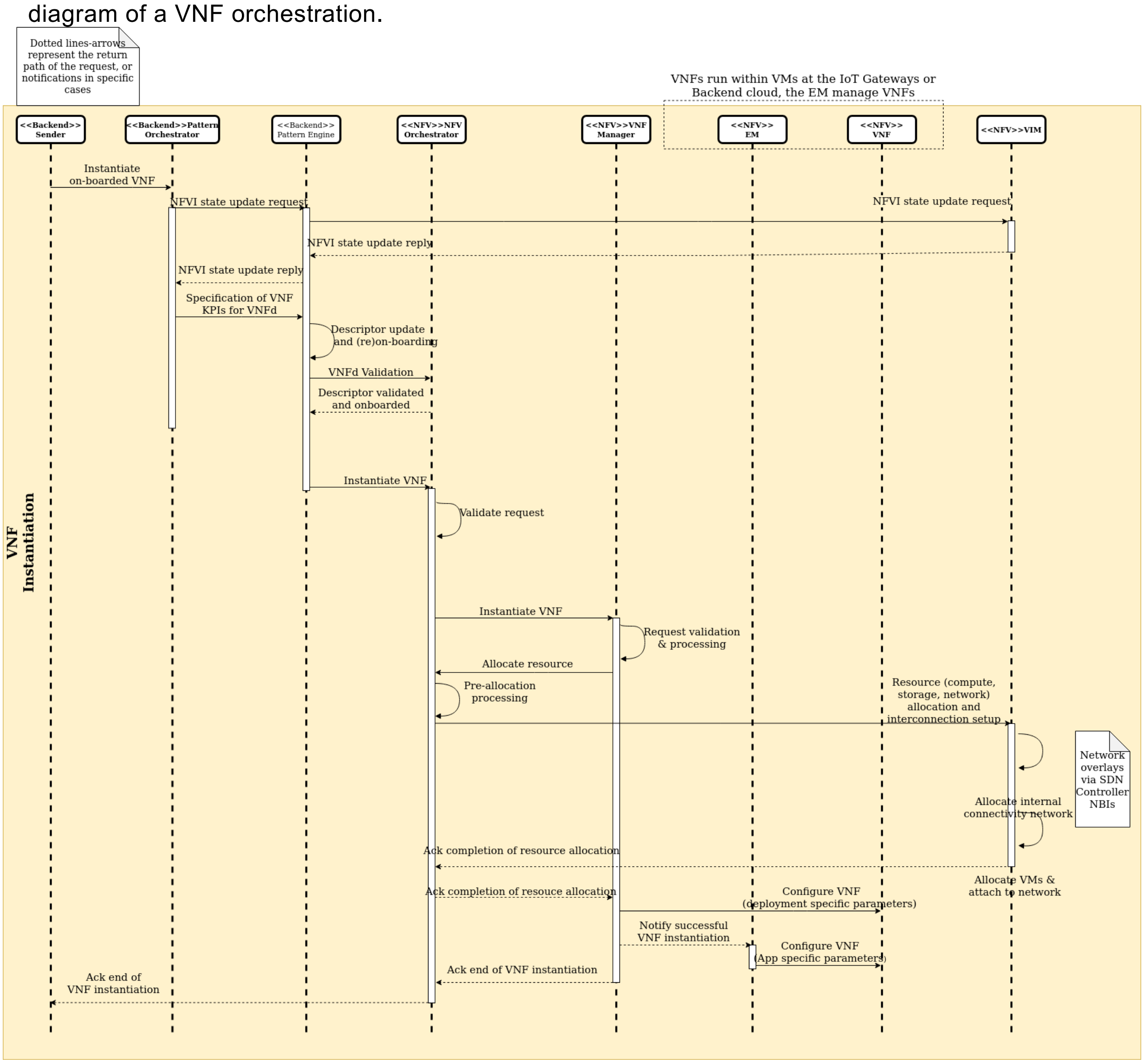
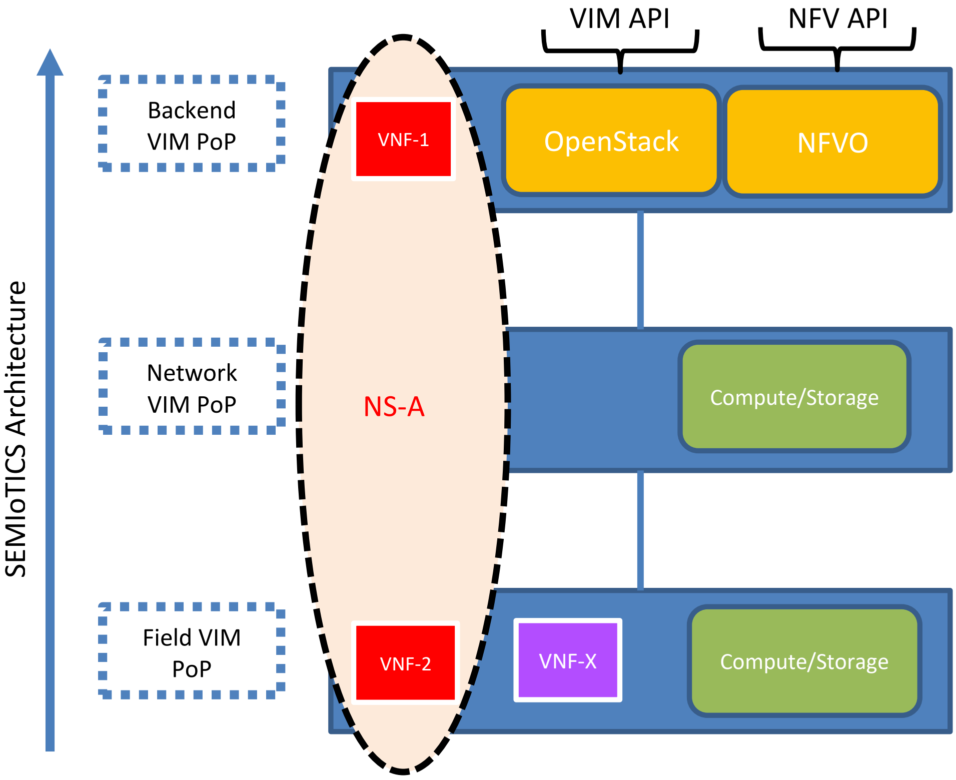

Dotted lines-arrows represent the return path of the request, or notifications in specific cases
VNFs run within VMs at the IoT Gateways or Backend cloud, the EM manage VNFs
<<Backend>> Sender
<<Backend>>Pattern Orchestrator
<<Backend>> Pattern Engine
<<NFV>>NFV Orchestrator
<<NFV>>VNF Manager
<<NFV>> EM
<<NFV>> VNF
<<NFV>>VIM
VNF Instantiation
Instantiate on-boarded VNF
NFVI state update request
NFVI state update request
NFVI state update reply
NFVI state update reply
Specification of VNF KPIs for VNFd
Descriptor update and (re)on-boarding
VNFd Validation
Descriptor validated and onboarded
Instantiate VNF
Validate request
Instantiate VNF
Request validation & processing
Allocate resource
Pre-allocation processing
Resource (compute, storage, network) allocation and interconnection setup
Network overlays via SDN Controller NBIs
Allocate internal connectivity network
Ack completion of resource allocation
Allocate VMs & attach to network
Ack completion of resouce allocation
Configure VNF (deployment specific parameters)
Notify successful VNF instantiation
Configure VNF (App specific parameters)
Ack end of VNF instantiation
Ack end of VNF instantiation

FIGURE 18 VNF ORCHESTRATION

- NFV Telemetry: via well-defined endpoints, authorized components may trigger NFVO's or VIM's APIs in order to gather telemetry metrics from running NS[4], or the NFVI, respectively. The former can be gathered via the *Os-Ma-Nfvo* endpoint at the NFV MANO (see FIGURE 14), while the latter is exposed as a telemetry endpoint at the VIM manager. FIGURE 19 summarizes the physical location of the aforementioned API endpoints, as well as the information they provide.

[4] Subject to metrics definition at descriptor level.



SEMIoTICS leverages NFVO and VIM API endpoints:

- **NFVO API endpoint**: NFVO/VIM/VNF telemetry, orchestrations, onboarding.
- **VIM API endpoint:** custom telemetry, maintenance and configuration.

FIGURE 19 NFVO AND VIM ENDPOINTS FOR TELEMETRY AND OTHER OPERATIONS

The aforementioned operations are achieved through well specified API endpoints. The following excerpt from D2.5 gathers the API enabled in the NFV Component for different operations in SEMIoTICS:

- At NFVO:
  - VNF Telemetry relay to OSS/BSS.
  - VNF/NS instance termination.
  - VNF/NS instantiation.
  - VNF/NS descriptor onboarding.
- At VIM:
  - Terminate VNFs.
  - Remove network connection between VNFs.
  - Configure network connection between VNFs.
  - Create network connection between VNFs.
  - Gather NFVI telemetry.
  - Gather VNF telemetry.
  - NFVI resource allocation/release/update.

Designing procedures leveraging such APIs ensures QoS constraints are met, as well as provide dynamicity and customization to NS (e.g. by other SEMIoTICS components such as the Pattern Orchestrator).

3.2.2 TESTING METHODOLOGY

Testing the NFV Component implies the description of network topologies in terms of NFV NS descriptors, which in turn imply the realization of functionalities via VNFs. At this point in the project, two main tests were performed involving the NFV Component: simple service instantiation, and data processing leveraging tenant networks and multiple clouds. In this section, these two tests are going to be overviewed (as they appear in full detail in D3.2 and D3.5, respectively). Additionally, the foreseeable integration methodology and tests are going to be described outside D3.2 for the first time.

3.2.2.1 SERVICE INSTANTIATION

It is a good practice to describe each NS in terms of a simple drawing. The following FIGURE 20 shows an example diagram describing "*test-cn*", a VNF composed of three Virtualization Deployment Units (VDU), i.e. Virtual Machines.

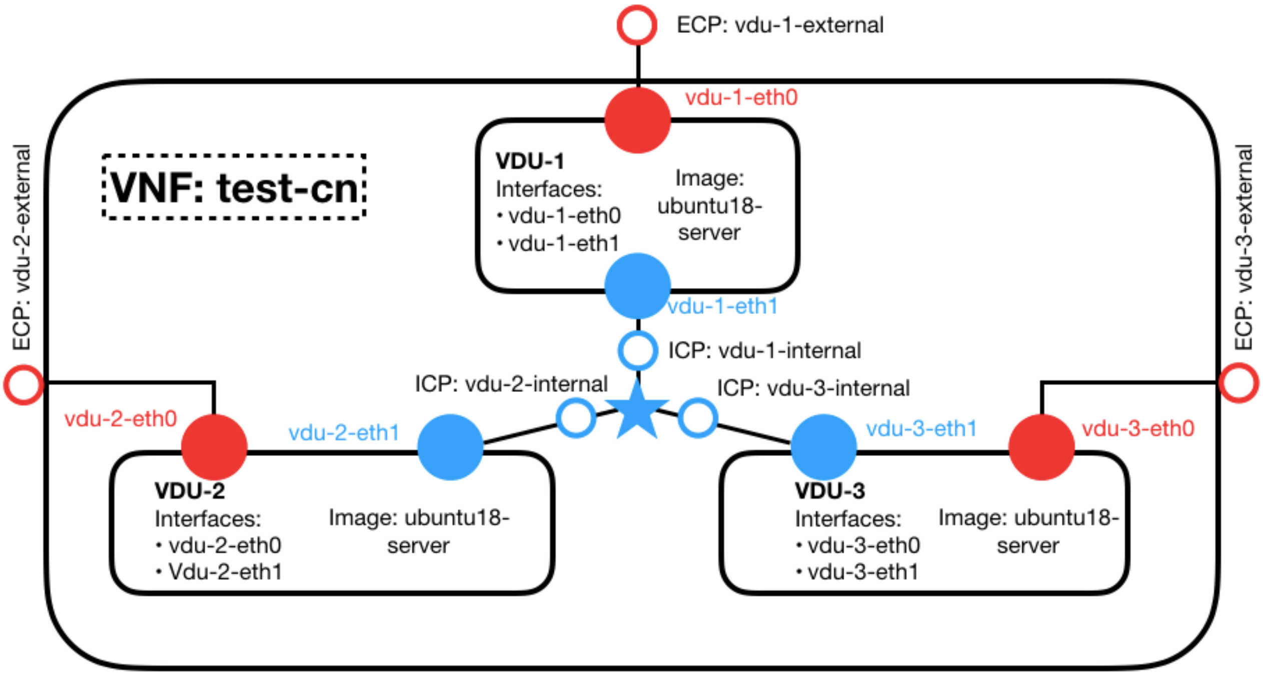


FIGURE 20 EXAMPLE VNF DESCRIPTION DIAGRAM

FIGURE 20 shows three VDUs, each one equipped with two network interfaces. The ones connected from their respective VDU to the External Connection Point (ECP) are referred to as *external* interfaces (e.g. for management purposes, or for exposure to external networks), while the others are connected to Internal Connection Points (ICP) via virtual links. Internal interfaces enable communication among VDUs within a VNF. The figure only shows minor (incomplete) details regarding networking and VDU source image, but more details may be specified, such as: vCPUs, memory, storage, monitoring parameters, scaling parameters and thresholds, etc. These are relevant parameters that need to be known before a descriptor could be written.

Having a descriptor for such VNF requires compliance with SEMIoTICS NFV MANO platform, specifically with the NFVO. SEMIoTICS NFVO, i.e. OSM, provides an ETSI compliant Information Model (IM)[5] for the specification of VNFs and NS via descriptors. An example of VNF and NS descriptor is provided in Section 3.3.1 of D3.2.

3.2.2.2 TENANT NETWORKS AND PLACEMENT

[5] https://osm.etsi.org/wikipub/index.php/OSM_Information_Model

SEMIoTICS NFV Component can be leveraged to place computing agents precisely where they are needed within the architecture. In D3.5, a testbed emulating the layers of SEMIoTICS was deployed to mimic a smart monitoring and actuation scenario. In it, two tenant networks were created, each holding the sensing and actuation VNFs, respectively. Experiments were setup to dynamically modify in real time the attainable throughput in each tenant network (refer to FIGURE 21). Additionally, leveraging precise placement instructions an additional test attempted to register the different network delays between a Field device and a destination VNFs located either at the IoT Gateway (Field Layer or Local Cloud), or in the Cloud.

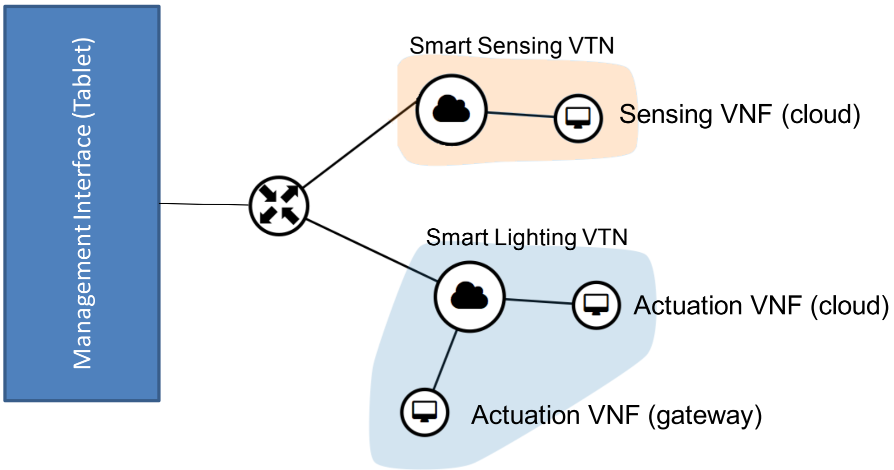


FIGURE 21 IIOT SERVICES AND TENANT NETWORKD (FROM D3.5)

3.2.2.3 INTEGRATION METHODOLOGY

As mentioned at the beginning of this section, SEMIoTICS NFV Component is composed of two elements: a set of manager/controllers (NFV MANO, VIM), and compute nodes (NFVI). Both elements need to have IP connectivity among them. That is, NFV MANO (NFVO and VIM) must have IP connectivity with the rest of the NFVI. Integration tests for the NFV Component may also involve the description of a SEMIoTICS Use Case via NFV descriptors. This way, a centralized NFVO may trigger the destination VIM's API to orchestrate the service on top of a NFVI.

An integration test methodology would involve granting IP connectivity to other SEMIoTICS components to the NFV MANO and NFVI. There are two configurations in which this could be done:

1. Allow an external NFVO to register the local VIM.
2. Allow external access to local NFV MANO.

The first option would allow an external NFVO to orchestrate VNFs in the local NFVI (via the local VIM). This way, NFVO configuration and management would be carried out by an external partner. The second option allows access to a partner (via SSH) to the local NFV Component. This way a single NFVO is maintained, and metrics collection at the NFVO would be gathered more efficiently. In fact this second option is the one that IS adopted in e.g. SEMIoTICS' SARA health use case. Moreover, below in subsection 2.2.3, we show the functional tests that validate the access of external VPN users to the NFV testbed and its services.

3.2.3 FUNCTIONAL TESTS, PERFORMANCE TESTS AND KPI VALIDATION

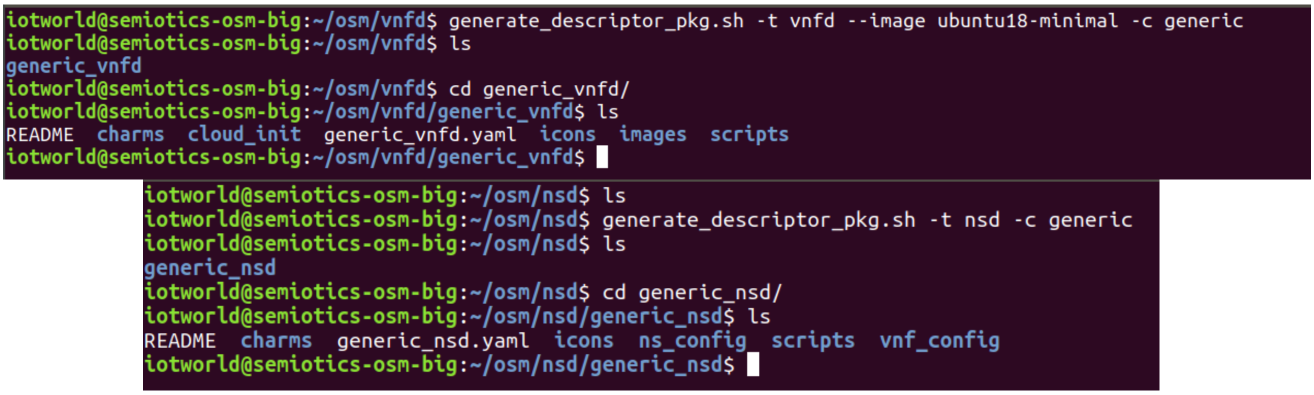
3.2.3.1 SERVICE INSTANTIATION RESULTS

After NS and VNF descriptor onboarding and effective service orchestration procedures, a working NS is shown to be running in Section 3.3.1 of D3.2. The reader is referred to the aforementioned section and deliverable in order to gather more details about descriptors, onboarding, GUI interfaces for OSM including: VIM registered, descriptor onboarding and NS status follow a set of steps to create, instantiate and terminate the generic VNF and the generic NS. Note that, the end user only needs to interact with the OSM, which communicates internally with the OpenStack. Moreover, the interaction with the OSM is through a set of configuration files that define the computing, storage and networking features of the VNF and the NS. These configuration files are so-called VNF descriptor (VNFd) and Network Service descriptor (NSd), respectively. Thereby, the next steps will be followed below to accomplish the following:

- Create generic NSd and VNFd.
- Generate VNF/NS packages.
- Onboard the VNF/NS packages to OSM library.
- Instantiate the NS (and the VNF).
- Check that we can access the VM created for the VNF instantiation.
- Terminate the NS.

Create VNFd, NSd folder structure

First, OSM needs to create a folder structure for the NSd, VNFd and its related files, such as the cloud init, which defines the initial configuration of the VM that will hold the VNF. This is accomplished by using a shell script provided by OSM, it is called “generate_descriptor_pkg”. In FIGURE 22, we show how that folder structure is created correctly.



```
iotworld@semiotics-osm-big:~/osm/vnfd$ generate_descriptor_pkg.sh -t vnfd --image ubuntu18-minimal -c generic
iotworld@semiotics-osm-big:~/osm/vnfd$ ls
generic_vnfd
iotworld@semiotics-osm-big:~/osm/vnfd$ cd generic_vnfd/
iotworld@semiotics-osm-big:~/osm/vnfd/generic_vnfd$ ls
README  charms  cloud_init  generic_vnfd.yaml  icons  images  scripts
iotworld@semiotics-osm-big:~/osm/vnfd/generic_vnfd$
```

```
iotworld@semiotics-osm-big:~/osm/nsd$ ls
iotworld@semiotics-osm-big:~/osm/nsd$ generate_descriptor_pkg.sh -t nsd -c generic
iotworld@semiotics-osm-big:~/osm/nsd$ ls
generic_nsd
iotworld@semiotics-osm-big:~/osm/nsd$ cd generic_nsd/
iotworld@semiotics-osm-big:~/osm/nsd/generic_nsd$ ls
README  charms  generic_nsd.yaml  icons  ns_config  scripts  vnf_config
iotworld@semiotics-osm-big:~/osm/nsd/generic_nsd$
```

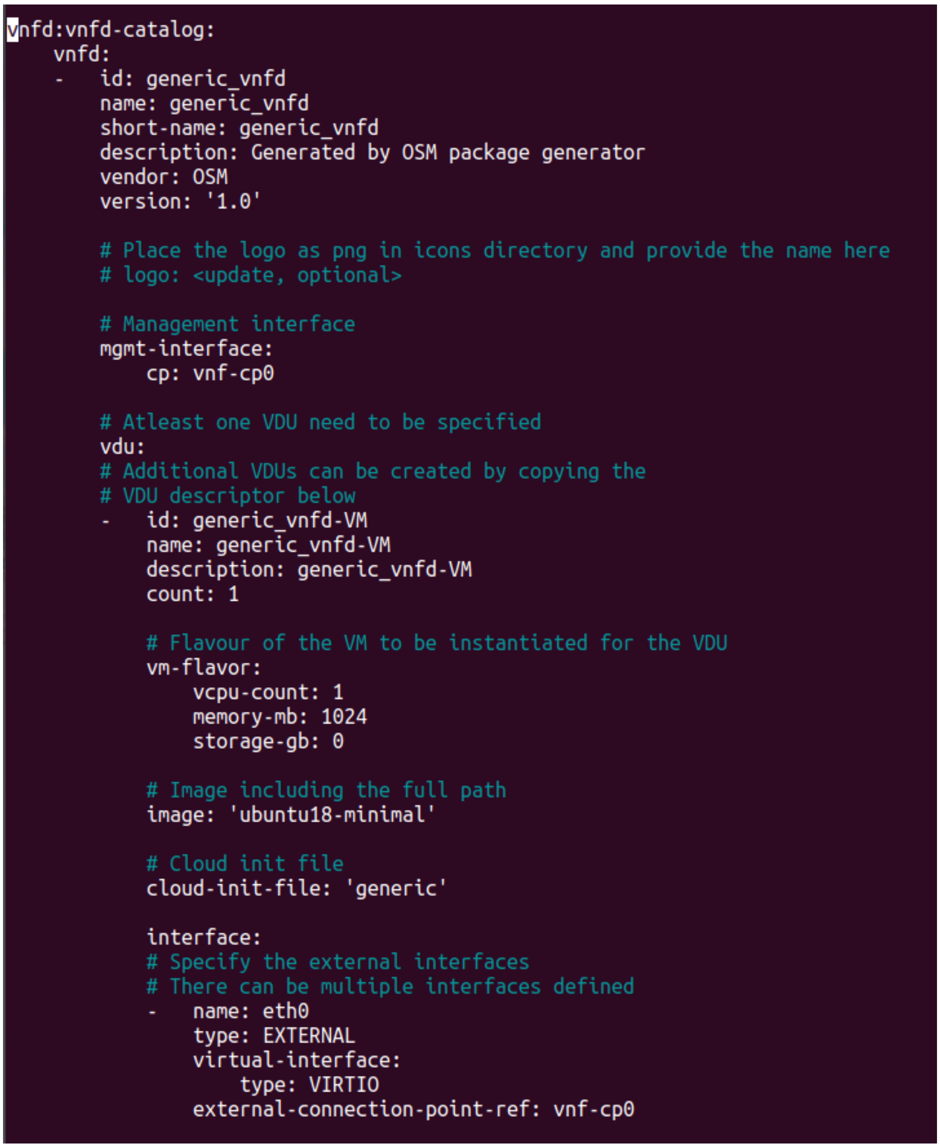
FIGURE 22 CREATION OF THE FOLDER STRUCTURE NEEDED TO CREATE VNFd and NSd.

Edit VNFd

After creating the folder structructre for the VNFd and NSd, we can edit the VNFd, which is a yaml configuration file. In FIGURE 23 we present the snapshot of the yaml file that we have used to specify the VNFd. In the sequel, the important parts of this file are discussed. The tag “id” is the unique identifier for the VNF and it is important to recall it, as it is used in the NSd. The tag “mgmt-interface” is the interface over which the VNF is managed. Moreover, the “cp” within it just specifies the type of management endpoint, in our case “cp” means that we will use a connection point. Another important tag is the “vdu”, which stands for virtual description unit, and it specifies the features of the VM that will host the VNF. The “vm-flavor” indicates the computing, memory and storage features of the VM that will host the VNF. Thereby, note that we define a VM with 1 virtual CPU, 1 GB of RAM and no persistent storage, as the “image” that we discuss next defines enough storage for this test. The tag “image” indicates the image that will be used to create the VM, in our case we will have an Ubuntu OS. The “cloud-init-file” indicates the cloud init file that will be used by the VM. The snapshot for this cloud init file is actually described below. The “interface” tag within the “vdu” tag specifies the interfaces for the vdu.

Cloud init file specification

This file is the one that specifies the initial configuration that we aim for the VM that will host the generic VNF. Note that its name is specified in the vnfd, as commented above. In FIGURE 24, we provide the snapshot of this cloud init file and describe its functionalities. First, note that we have a field called “users”. This allows to add users to the system. Note that we have added a user called “generic”. Finally, within this user, there is an important field to be added, the “ssh_authorized_keys”. This is important, because here we add the public ssh keys of the users that will access the VM that hosts the VNF.

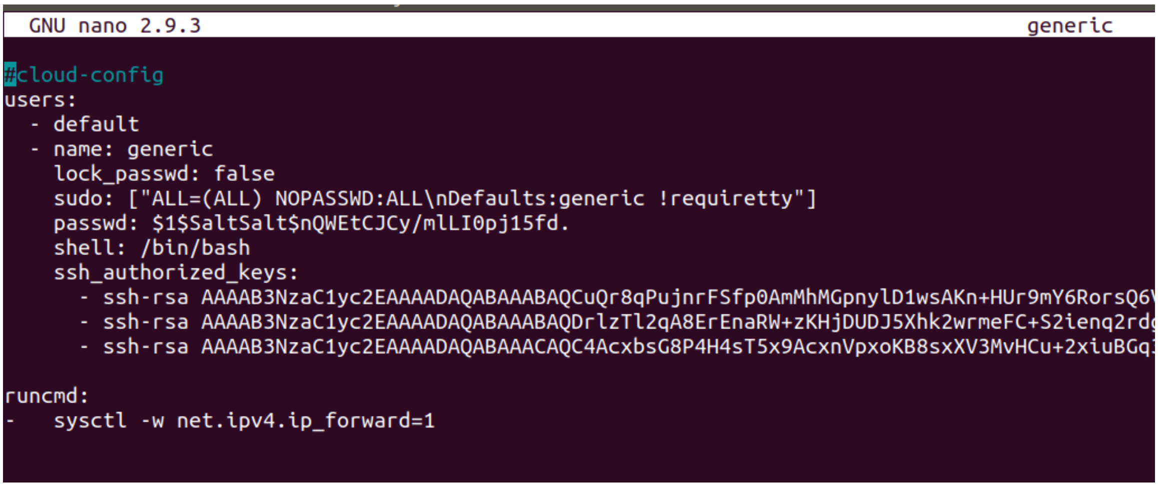


```
vnfd:vnfd-catalog:
    vnfd:
    -   id: generic_vnfd
        name: generic_vnfd
        short-name: generic_vnfd
        description: Generated by OSM package generator
        vendor: OSM
        version: '1.0'

        # Place the logo as png in icons directory and provide the name here
        # logo: <update, optional>

        # Management interface
        mgmt-interface:
            cp: vnf-cp0

        # Atleast one VDU need to be specified
        vdu:
        # Additional VDUs can be created by copying the
        # VDU descriptor below
        -   id: generic_vnfd-VM
            name: generic_vnfd-VM
            description: generic_vnfd-VM
            count: 1

            # Flavour of the VM to be instantiated for the VDU
            vm-flavor:
                vcpu-count: 1
                memory-mb: 1024
                storage-gb: 0

            # Image including the full path
            image: 'ubuntu18-minimal'

            # Cloud init file
            cloud-init-file: 'generic'

            interface:
            # Specify the external interfaces
            # There can be multiple interfaces defined
            -   name: eth0
                type: EXTERNAL
                virtual-interface:
                    type: VIRTIO
                external-connection-point-ref: vnf-cp0
```

FIGURE 23 VNFD THAT DESCRIBES A GENERIC VNF.

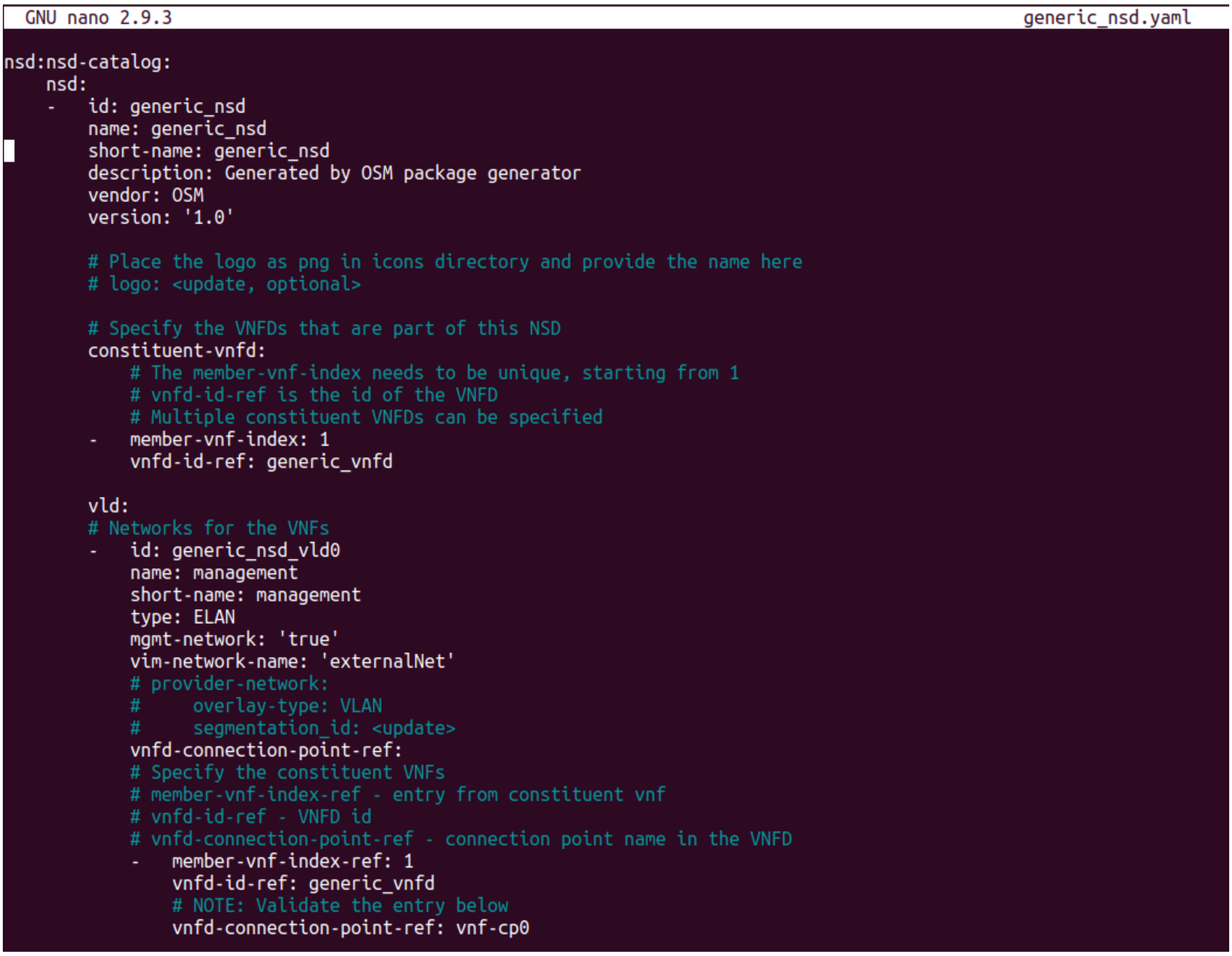
Edit NSd

Next, in FIGURE 25 we display a snapshot of the yaml file that we edited to specify the NSd. The relevant tags are described in the sequel. First, note that the tag “id” determines the unique identifier for the NS. The tag “constituent-vnfd” indicates which VNFs are part of the NS. In our case we have just the generic VNF, whose identification is “generic_vnfd” and is specified through the tag “vnfd-id-ref”. Note that, in the vnfd the “id” tag has to correspond with that value. Then, we have the tag “vld”, which is a description of the virtual links used by the NS for networking connections. In our case, note that the tag “type” is set to “ELAN”. This indicates that the virtual link is a service to connect VNFs. The tag “mgmt.-network” set to “true” means that this is a VIM management network. The tag “vim-network-name” describes the name of the network in the VIM account, in our case “externalNet” is the name that was given such network in the OpenStack framework. Finally, the tag “vnfd-connection-point-ref” describes the connection points for the virtual links towards a vnf. We can see that this is a connection towards our generic VNF as the tag “vnd-id-ref” is set to “generic_vnfd”.



```
  GNU nano 2.9.3                                                                   generic

#cloud-config
users:
  - default
  - name: generic
    lock_passwd: false
    sudo: ["ALL=(ALL) NOPASSWD:ALL\nDefaults:generic !requiretty"]
    passwd: $1$SaltSalt$nQWEtCJCy/mlLI0pj15fd.
    shell: /bin/bash
    ssh_authorized_keys:
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCuQr8qPujnrFSfp0AmMhMGpnylD1wsAKn+HUr9mY6RorsQ6
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDrlzTl2qA8ErEnaRW+zKHjDUDJ5Xhk2wrmeFC+S2ienq2rd
      - ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQC4AcxbsG8P4H4sT5x9AcxnVpxoKB8sxXV3MvHCu+2xiuBGq

runcmd:
-   sysctl -w net.ipv4.ip_forward=1
```

FIGURE 24 CLOUD INIT FILE ASSOCIATED TO THE GENERIC VNF.



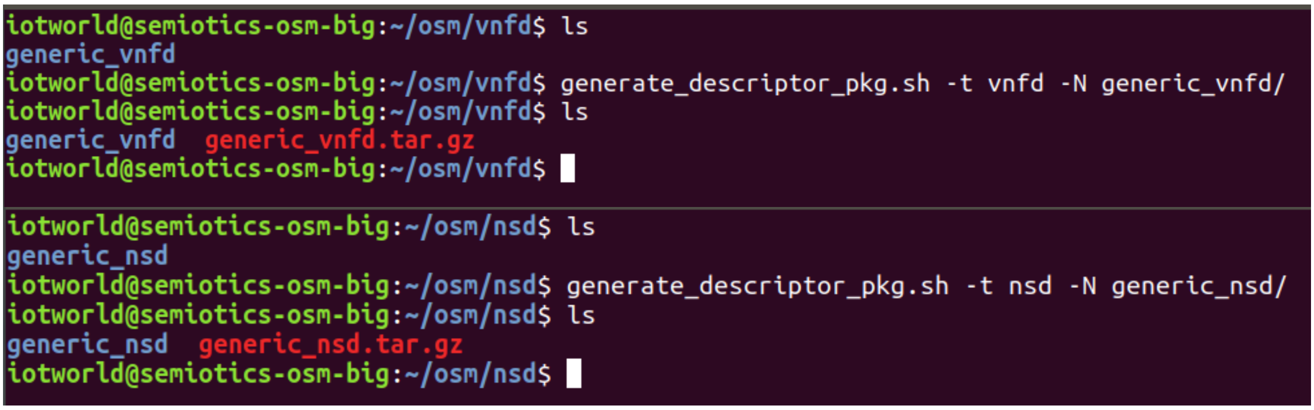
```
  GNU nano 2.9.3                                                                   generic_nsd.yaml

nsd:nsd-catalog:
    nsd:
    -   id: generic_nsd
        name: generic_nsd
        short-name: generic_nsd
        description: Generated by OSM package generator
        vendor: OSM
        version: '1.0'

        # Place the logo as png in icons directory and provide the name here
        # logo: <update, optional>

        # Specify the VNFDs that are part of this NSD
        constituent-vnfd:
            # The member-vnf-index needs to be unique, starting from 1
            # vnfd-id-ref is the id of the VNFD
            # Multiple constituent VNFDs can be specified
        -   member-vnf-index: 1
            vnfd-id-ref: generic_vnfd

        vld:
        # Networks for the VNFs
        -   id: generic_nsd_vld0
            name: management
            short-name: management
            type: ELAN
            mgmt-network: 'true'
            vim-network-name: 'externalNet'
            # provider-network:
            #     overlay-type: VLAN
            #     segmentation_id: <update>
            vnfd-connection-point-ref:
            # Specify the constituent VNFs
            # member-vnf-index-ref - entry from constituent vnf
            # vnfd-id-ref - VNFD id
            # vnfd-connection-point-ref - connection point name in the VNFD
            -   member-vnf-index-ref: 1
                vnfd-id-ref: generic_vnfd
                # NOTE: Validate the entry below
                vnfd-connection-point-ref: vnf-cp0
```

FIGURE 25 NSD ASSOCIATED TO THE GENERIC VNF.

.

Generate VNFd/NSd packages.

At this point, the VNFd and NSd are already edited. Thereby, we can generate the NSd, and VNFd packages, which are required in the onboarding process of the OSM. This means, the process of having our NS and VNF packages available in the OSM library. To generate those packages we only need to execute the shell script "generate_descriptor_pkg" provided by the OSM, which needs the NS and VNF folder structure that we created above in FIGURE 26.



```
iotworld@semiotics-osm-big:~/osm/vnfd$ ls
generic_vnfd
iotworld@semiotics-osm-big:~/osm/vnfd$ generate_descriptor_pkg.sh -t vnfd -N generic_vnfd/
iotworld@semiotics-osm-big:~/osm/vnfd$ ls
generic_vnfd  generic_vnfd.tar.gz
iotworld@semiotics-osm-big:~/osm/vnfd$ 
```

```
iotworld@semiotics-osm-big:~/osm/nsd$ ls
generic_nsd
iotworld@semiotics-osm-big:~/osm/nsd$ generate_descriptor_pkg.sh -t nsd -N generic_nsd/
iotworld@semiotics-osm-big:~/osm/nsd$ ls
generic_nsd  generic_nsd.tar.gz
iotworld@semiotics-osm-big:~/osm/nsd$ 
```

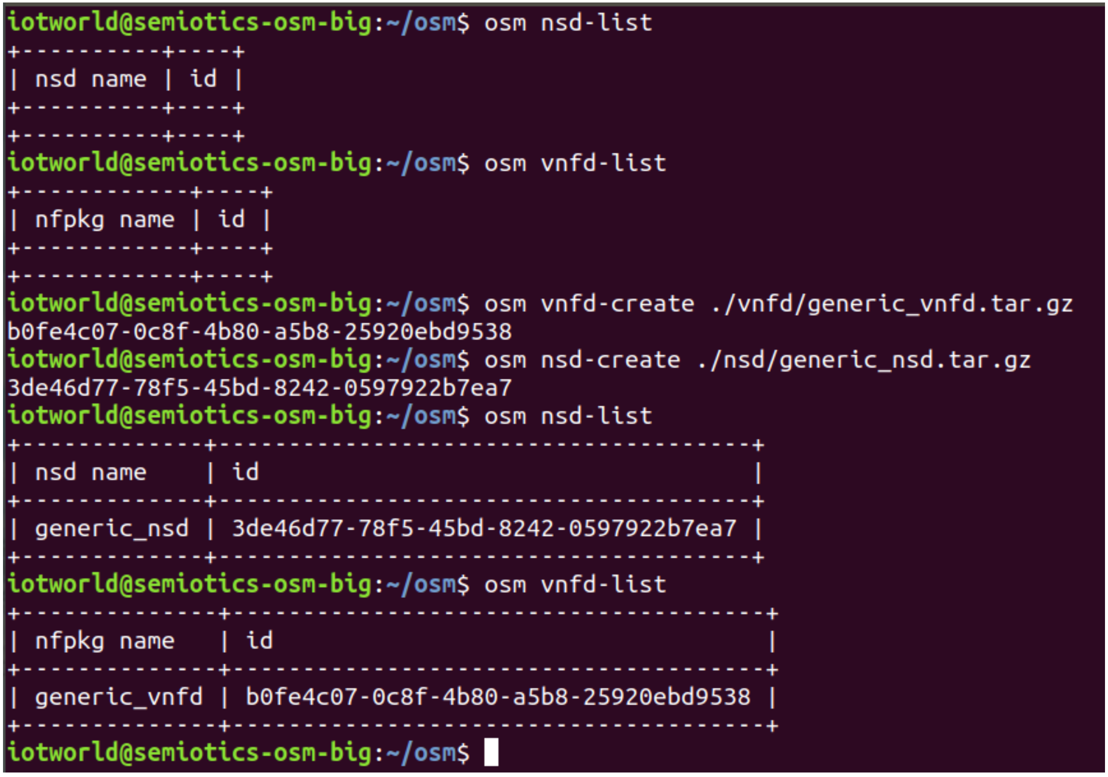
FIGURE 26 GENERATION OF VNFd AND NSd PACKAGES.

Onboard the VNF/NS packages to OSM library.

Next, in FIGURE 27 it is demonstrated that we are able to onboard properly the NSd and VNFd packages into the library of OSM. Recall that these are a set of configuration files that describe the properties of our VNF and the requirements in terms of computing and networking that it has. Also, they describe the features of the VM that will host the VNF and the initial configuration and software packages installations that we need in the VM. We have called the NSd and VNFd as generic_nsd and generic_vnfd, respectively. It can be observed that OSM has onboarded properly the NSd and VNFd, as they appear on the list of available packages in the OSM library. Note that the osm instruction "osm nsd-list" and "osm vnfd-list" were used.

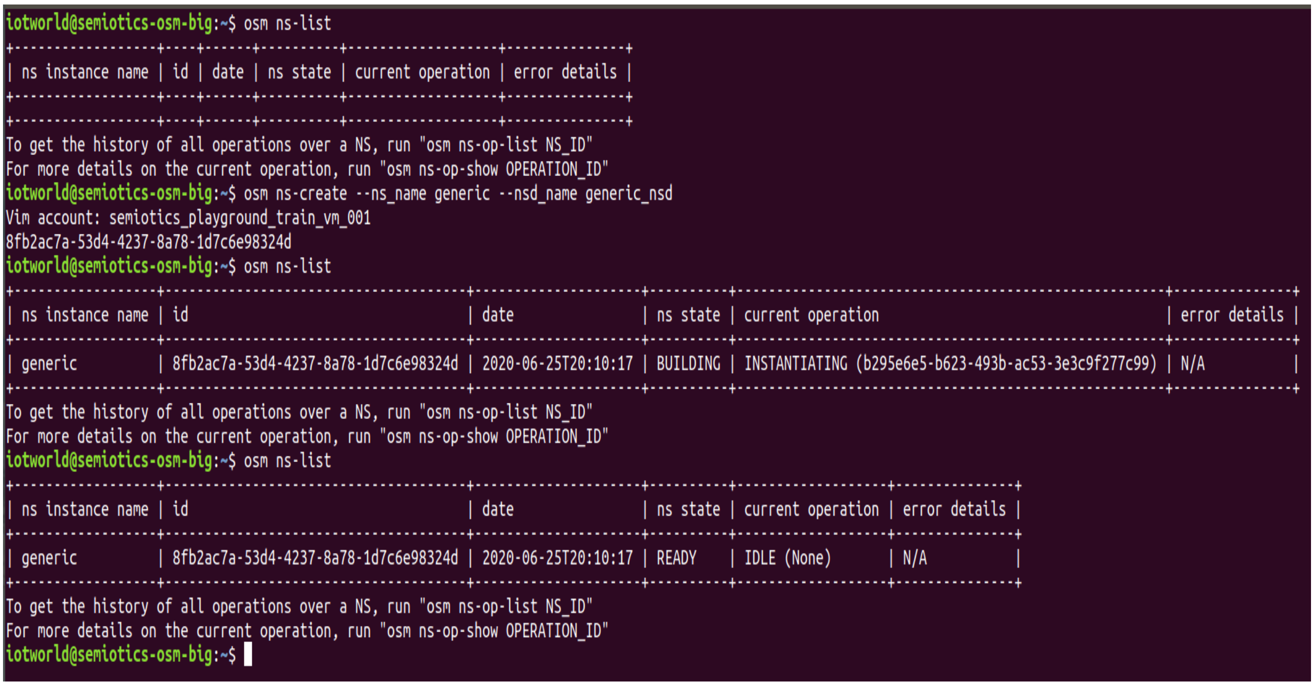
Instantiate the NS.

Then, in FIGURE 28 we trigger the instantiation of the NS that we have onboarded in OSM for our generic VNF. This means that OSM will communicate with the OpenStack controller, which creates the VM that will host our VNF on top of the virtual resources exposed by the OpenStack compute node. For that, obviously, the OSM takes into account the information embedded within the NSd and VNFd. Note that to trigger the NS instantiation we used the OSM command "osm ns-create –ns_name generic –nsd_name generic_nsd". Observe that you must specify for the nsd_name option the name of the NSd that you want to instantiate, otherwise the NS will not be instantiated.



```
iotworld@semiotics-osm-big:~/osm$ osm nsd-list
+----------+----+
| nsd name | id |
+----------+----+
+----------+----+
iotworld@semiotics-osm-big:~/osm$ osm vnfd-list
+------------+----+
| nfpkg name | id |
+------------+----+
+------------+----+
iotworld@semiotics-osm-big:~/osm$ osm vnfd-create ./vnfd/generic_vnfd.tar.gz
b0fe4c07-0c8f-4b80-a5b8-25920ebd9538
iotworld@semiotics-osm-big:~/osm$ osm nsd-create ./nsd/generic_nsd.tar.gz
3de46d77-78f5-45bd-8242-0597922b7ea7
iotworld@semiotics-osm-big:~/osm$ osm nsd-list
+-------------+--------------------------------------+
| nsd name    | id                                   |
+-------------+--------------------------------------+
| generic_nsd | 3de46d77-78f5-45bd-8242-0597922b7ea7 |
+-------------+--------------------------------------+
iotworld@semiotics-osm-big:~/osm$ osm vnfd-list
+--------------+--------------------------------------+
| nfpkg name   | id                                   |
+--------------+--------------------------------------+
| generic_vnfd | b0fe4c07-0c8f-4b80-a5b8-25920ebd9538 |
+--------------+--------------------------------------+
iotworld@semiotics-osm-big:~/osm$ 
```

FIGURE 27 ONBOARDING OF THE VNFD AND NSD PACKAGES TO OSM.

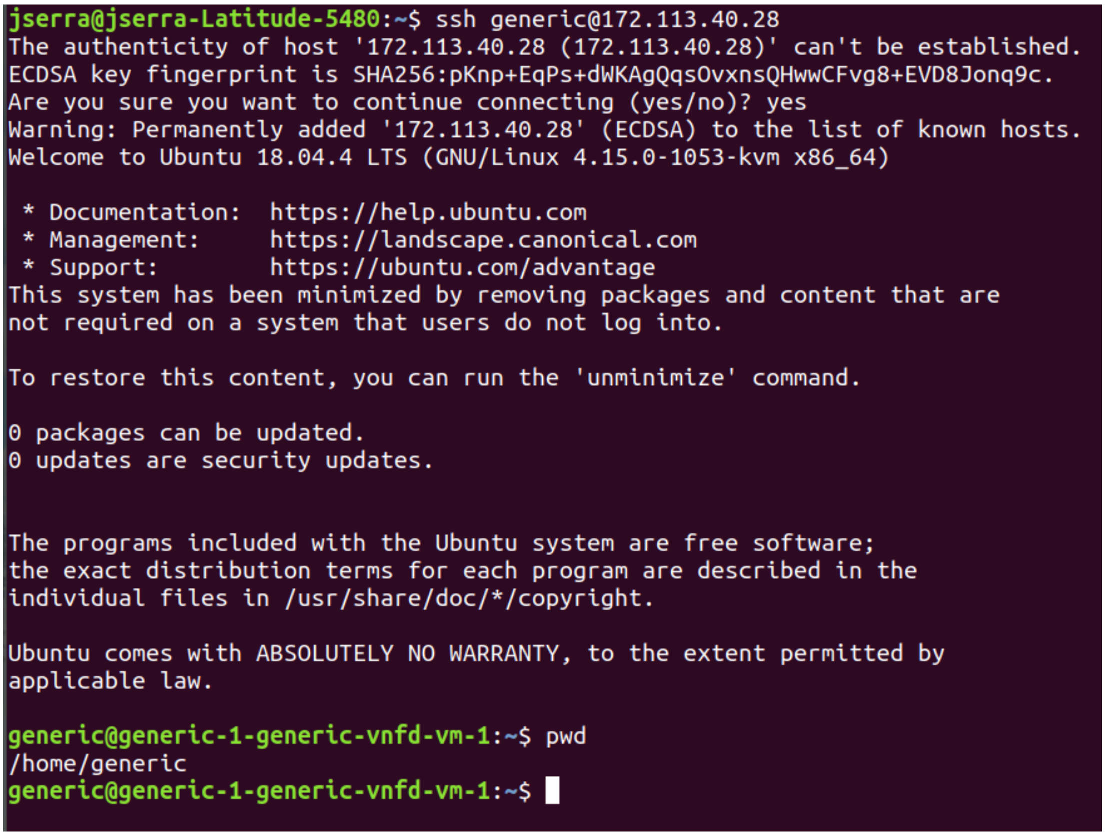


```
iotworld@semiotics-osm-big:~$ osm ns-list
+------------------+----+------+----------+-------------------+---------------+
| ns instance name | id | date | ns state | current operation | error details |
+------------------+----+------+----------+-------------------+---------------+
+------------------+----+------+----------+-------------------+---------------+
To get the history of all operations over a NS, run "osm ns-op-list NS_ID"
For more details on the current operation, run "osm ns-op-show OPERATION_ID"
iotworld@semiotics-osm-big:~$ osm ns-create --ns_name generic --nsd_name generic_nsd
Vim account: semiotics_playground_train_vm_001
8fb2ac7a-53d4-4237-8a78-1d7c6e98324d
iotworld@semiotics-osm-big:~$ osm ns-list
+------------------+--------------------------------------+---------------------+----------+------------------------------------------------------+---------------+
| ns instance name | id                                   | date                | ns state | current operation                                    | error details |
+------------------+--------------------------------------+---------------------+----------+------------------------------------------------------+---------------+
| generic          | 8fb2ac7a-53d4-4237-8a78-1d7c6e98324d | 2020-06-25T20:10:17 | BUILDING | INSTANTIATING (b295e6e5-b623-493b-ac53-3e3c9f277c99) | N/A           |
+------------------+--------------------------------------+---------------------+----------+------------------------------------------------------+---------------+
To get the history of all operations over a NS, run "osm ns-op-list NS_ID"
For more details on the current operation, run "osm ns-op-show OPERATION_ID"
iotworld@semiotics-osm-big:~$ osm ns-list
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
| ns instance name | id                                   | date                | ns state | current operation | error details |
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
| generic          | 8fb2ac7a-53d4-4237-8a78-1d7c6e98324d | 2020-06-25T20:10:17 | READY    | IDLE (None)       | N/A           |
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
To get the history of all operations over a NS, run "osm ns-op-list NS_ID"
For more details on the current operation, run "osm ns-op-show OPERATION_ID"
iotworld@semiotics-osm-big:~$ 
```

FIGURE 28 INSTANTIATION OF THE NS RELATED TO THE GENERIC VNF.

Check that we can access the VM created for the VNF instantiation.

The NS instantiation creates the VM that holds the VNF, on top of the NFVI. Thereby, it is important to check if we have access to such VM. In order to obtain the IP of the VM we can just execute the OpenStack command ”openstack server list”. OSM also provides this IP in the information of the NS instance. In FIGURE 29 we show that an external VPN user can effectively access the VM that holds the generic VNF.

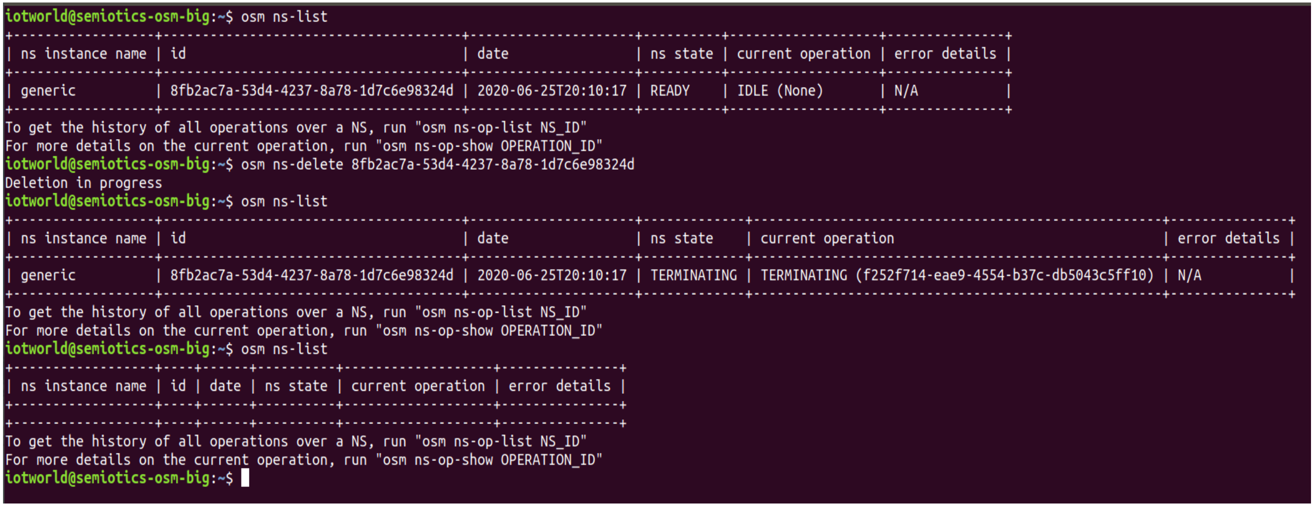


```
jserra@jserra-Latitude-5480:~$ ssh generic@172.113.40.28
The authenticity of host '172.113.40.28 (172.113.40.28)' can't be established.
ECDSA key fingerprint is SHA256:pKnp+EqPs+dWKAgQqsOvxnsQHwwCFvg8+EVD8Jonq9c.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.113.40.28' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-1053-kvm x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

0 packages can be updated.
0 updates are security updates.


The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

generic@generic-1-generic-vnfd-vm-1:~$ pwd
/home/generic
generic@generic-1-generic-vnfd-vm-1:~$ 
```

FIGURE 29 CHECK THE ACCESS TO THE VM CREATED TO HOLD THE GENERIC VNF.

Terminate the NS.

Finally, we show that we can terminate the NS, which shows the overall lifecycle of a NS and its associated VNFs. To this end, we use the OSM command “osm ns-delete” and its argument must specify the id of the NS instance that we want to terminate. In FIGURE 30 we show that we can terminate the NS instance associated to the generic VNF.



```
iotworld@semiotics-osm-big:~$ osm ns-list
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
| ns instance name | id                                   | date                | ns state | current operation | error details |
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
| generic          | 8fb2ac7a-53d4-4237-8a78-1d7c6e98324d | 2020-06-25T20:10:17 | READY    | IDLE (None)       | N/A           |
+------------------+--------------------------------------+---------------------+----------+-------------------+---------------+
To get the history of all operations over a NS, run "osm ns-op-list NS_ID"
For more details on the current operation, run "osm ns-op-show OPERATION_ID"
iotworld@semiotics-osm-big:~$ osm ns-delete 8fb2ac7a-53d4-4237-8a78-1d7c6e98324d
Deletion in progress
iotworld@semiotics-osm-big:~$ osm ns-list
+------------------+--------------------------------------+---------------------+-------------+----------------------------------------------------+---------------+
| ns instance name | id                                   | date                | ns state    | current operation                                  | error details |
+------------------+--------------------------------------+---------------------+-------------+----------------------------------------------------+---------------+
| generic          | 8fb2ac7a-53d4-4237-8a78-1d7c6e98324d | 2020-06-25T20:10:17 | TERMINATING | TERMINATING (f252f714-eae9-4554-b37c-db5043c5ff10) | N/A           |
+------------------+--------------------------------------+---------------------+-------------+----------------------------------------------------+---------------+
To get the history of all operations over a NS, run "osm ns-op-list NS_ID"
For more details on the current operation, run "osm ns-op-show OPERATION_ID"
iotworld@semiotics-osm-big:~$ osm ns-list
+------------------+----+------+----------+-------------------+---------------+
| ns instance name | id | date | ns state | current operation | error details |
+------------------+----+------+----------+-------------------+---------------+
+------------------+----+------+----------+-------------------+---------------+
To get the history of all operations over a NS, run "osm ns-op-list NS_ID"
For more details on the current operation, run "osm ns-op-show OPERATION_ID"
iotworld@semiotics-osm-big:~$ 
```

FIGURE 30 TERMINATE THE NS RELATED TO THE GENERIC VNF.

3.2.3.2 TENANTS NETWORKS AND PLACEMENT RESULTS

As described in the previous section, two tests were to be carried out:

1. Live Network throughput modification.
2. Measurement of delays between local and global cloud.

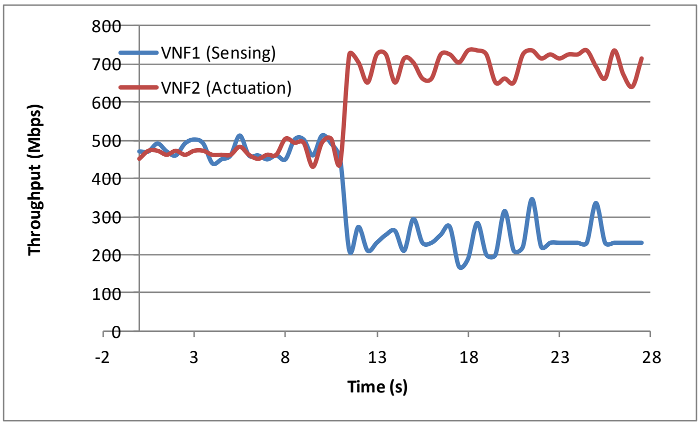


FIGURE 31 THROUGHPUT MEASUREMENT VS. TIME FOR VNF1 AND VNF2

FIGURE 31, extracted from D3.5, shows the results of a network throughput modification at time t=11s. That is, at the beginning of the experiment both VNFs were able to achieve the same throughput towards a cloud destination (1 Gbps external link, therefore 500 Mbps is achieved by each VNF). Nevertheless, at the specified time the VIM's SDN Controller (Neutron) is instructed to modify the distribution of available bandwidth via the corresponding API, effectively lowering VNF1's throughput to around 300 Mbps. As the experiment was performed under saturated traffic, the rest of the available bandwidth is consumed by VNF2.

Regarding VNF placement as a way of lowering delay, measurements of Round-Trip Time (RTT) were performed between a Field device and the local cloud, as well as from the Field device to a cloud instance. Details about the experiment can be found in Section 5.2.2 of D3.5; FIGURE 32 shows RTT in milliseconds for a growing link load. Notice that each curve specifies the location of the destination VNF. Results show that as links get congested, placing the service VNF further away from the Field device results in higher RTT.

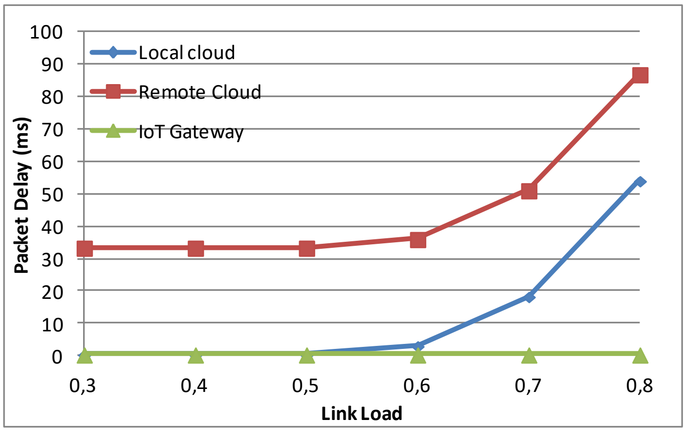


FIGURE 32 PACKET DELAY VS. LOAD FOR DIFFERENT VNF PLACEMENT OPTIONS

3.2.3.3 INTERACTION WITH THE NFV MANO BASED ON RESTFUL NORTH-BOUND INTERFACE (NBI).

In SEMIoTICS, the Pattern Orchestrator needs to interact with the NFV platform to enforce patterns in the underlying VNFs and NS. From the NFV MANO viewpoint, the Pattern Orchestrator can be regarded as Operation Support Systems (OSS) entities. Thereby, the interface between the NFV MANO and the Pattern Orchestrator is well defined through the Os-Ma-NFVO reference point, which is specified by the ETSI standards[6] [7]. In practice, the Os-Ma-NFVO interface can be implemented through RESTful protocols, as suggested by ETSI in the ETSI NFV-SOL specification[8]. Thereby, by means of these RESTful protocols the pattern orchestrator can perform remotely the NFV management operations that control the NS and the VNFs packages or its lifecycle. For instance, onboard a new VNF package, retrieve the information of the available VNF packages or instantiate a new NS. To this end, the RESTful protocol defines:

- The URI resource structure. For instance, the structure that identifies a NSD.
- The HTTP methods that can be applied to the URI resources. For instance, a GET method to consult the information of a given NSD.
- The data structure that we need to specify for a given HTTP method. For instance, a POST method to instantiate a NS requires to specify the identification of the NSD to be instantiated. And this corresponds to a data structure field called “nsdId”, which is specified in the body of the HTTP request.

Next, we provide two tests that allow an external OSS to onboard to the OSM a new VNF package through the Os-Ma-NFVO reference point, i.e. by means of the RESTful protocols mentioned above. It is worth mentioning that a complete sequence of tests is provided in SEMIoTICS’ deliverable D3.8[9]. Thereby, for the sake of brevity, we refer the reader to D3.8 to view all the tests that implement the NS and VNF management control through the RESTful NBI mentioned above.

The tests provided next allow to onboard a VNF package. Namely, the first one allows to create a VNF package resource, whereas the second one uploads the content of a VNF package. First, it is important to note, each management operation done through the Os-Ma-NFVO interface has a Uniform Resource Identifier (URI) structure[8]. Thereby, the URI structure to create a new VNF package resource, has the URI structure

[6] ETSI, "ETSI.org: Network Functions Virtualisation (NFV); Architectural Framework," 10 2013. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf. [Accessed August 2020].

[7] ETSI, "ETSI.org: Network Functions Virtualisation (NFV); Management and Orchestration (ETSI GSNFV-MAN 001)," December 2014. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf. [Accessed August 2020].

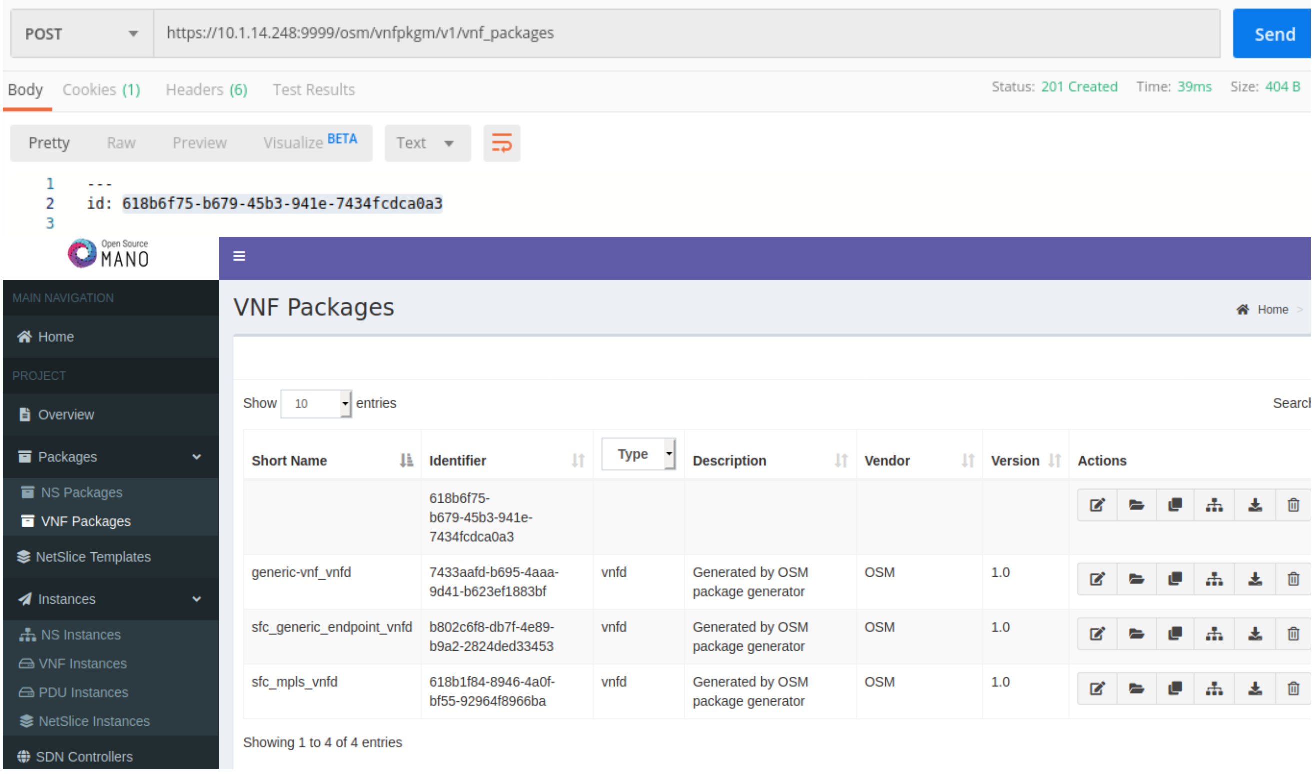
[8] ETSI, "ETSI GS NFV-SOL 005. Network Functions Virtualisation (NFV) Release 2.," ETSI, 2018.

[9] J. Serra et al. “Deliverable D3.8 Network Functions Virtualization for IoT (final)”, SEMIoTICS deliverable D3.8.

"{ApiRoot}/vnfpkgm/v1/vnf_packages". The placeholder {ApiRoot} indicates the scheme ("http" or "https"), the host name and optional port, and an optional prefix path. For instance, in the CTTC testbed, the OSM that implements the NFV MANO has the IP 10.1.14.248, which is accessible through the port 9999. Thereby, the in our case the URI resource structure to create a new VNF package resource has the following expression:

- "https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages"

Moreover, to create a new VNF package resource at the NFV MANO, i.e. the OSM, we need to apply a POST method on the URI resource structure just mentioned above. We used postman software tool[10] to emulate the OSS and to send the RESTful request to the OSM. In FIGURE 33, we display the postman interface that shows the POST query. Observe that it returns an "id" parameter, which corresponds to the identification for the VNF package resource that the OSM has created. FIGURE 33 also shows the web interface of the OSM, where we can observe that effectively a VNF package resource has been created with the id mentioned above. Observe that the VNF package resource is void, as we need to upload the VNF package content. This operation is presented next.



https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages

Status: 201 Created Time: 39ms Size: 404 B

```
---
id: 618b6f75-b679-45b3-941e-7434fcdca0a3
```

VNF Packages

Show 10 entries

Short Name	Identifier	Type	Description	Vendor	Version	Actions
	618b6f75-b679-45b3-941e-7434fcdca0a3					
generic-vnf_vnfd	7433aafd-b695-4aaa-9d41-b623ef1883bf	vnfd	Generated by OSM package generator	OSM	1.0	
sfc_generic_endpoint_vnfd	b802c6f8-db7f-4e89-b9a2-2824ded33453	vnfd	Generated by OSM package generator	OSM	1.0	
sfc_mpls_vnfd	618b1f84-8946-4a0f-bf55-92964f8966ba	vnfd	Generated by OSM package generator	OSM	1.0	

Showing 1 to 4 of 4 entries

FIGURE 33 VNF package resource creation through the RESTful NBI.

In order to upload the content related to a VNF package, i.e. its VNF descriptor, we need to perform a PUT request on the next URI structure:

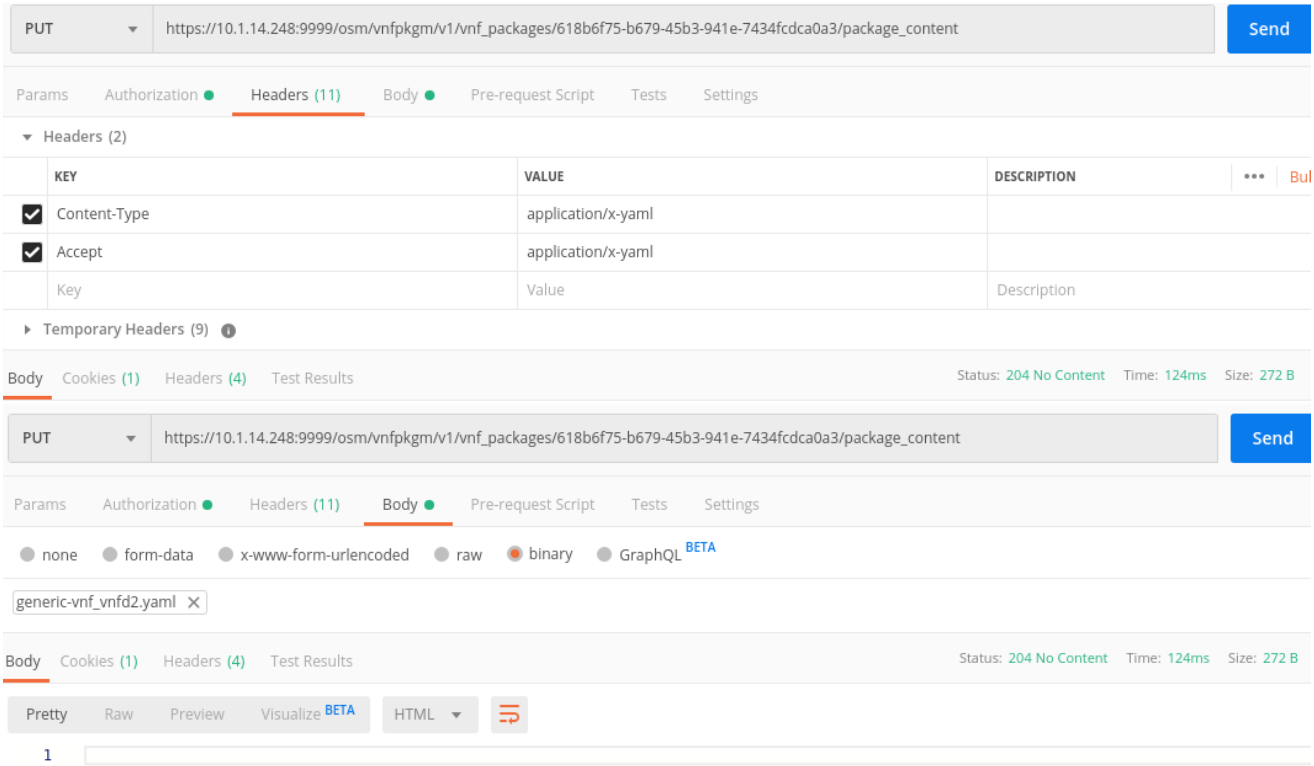
- https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages/{ vnfPkgId }/package_content.

Where the {vnfPkgId} placeholder in the URI resource must be substituted by the identification obtained by the POST operation described above. Observe that according to FIGURE 33 this identification reads "618b6f75-b679-45b3-941e-7434fcdca0a3". Therefore, the complete URI resource for the CTTC testbed reads:

[10] www.postman.com

- https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages/618b6f75-b679-45b3-941e-7434fcdca0a3/package_content.

Also, it is important to note that in the body of the RESTful command we attach the .yaml file that represents the VNF package descriptor, i.e. the content of the VNF package, which will be used by OSM. And another important detail is that in the headers of the RESTful command we must specify that we are sending a .yaml file. We do by setting the "Content-Type" and the "Accept" keys to the "application/x-yaml" value. Thereby, bearing in mind all these considerations, we present in **FIGURE 34** the complete RESTful command that we need to run to upload a VNF package content to OSM. As in the previous case, we did the implementation using postman. In **FIGURE 35** we show the effect that this PUT REST API has on the OSM. Comparing this figure to **FIGURE 33** we can see that now we have content for the VNF package.



PUT https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages/618b6f75-b679-45b3-941e-7434fcdca0a3/package_content

Headers (2)

	KEY	VALUE	DESCRIPTION
☑	Content-Type	application/x-yaml	
☑	Accept	application/x-yaml	
	Key	Value	Description

Temporary Headers (9)

Status: 204 No Content Time: 124ms Size: 272 B

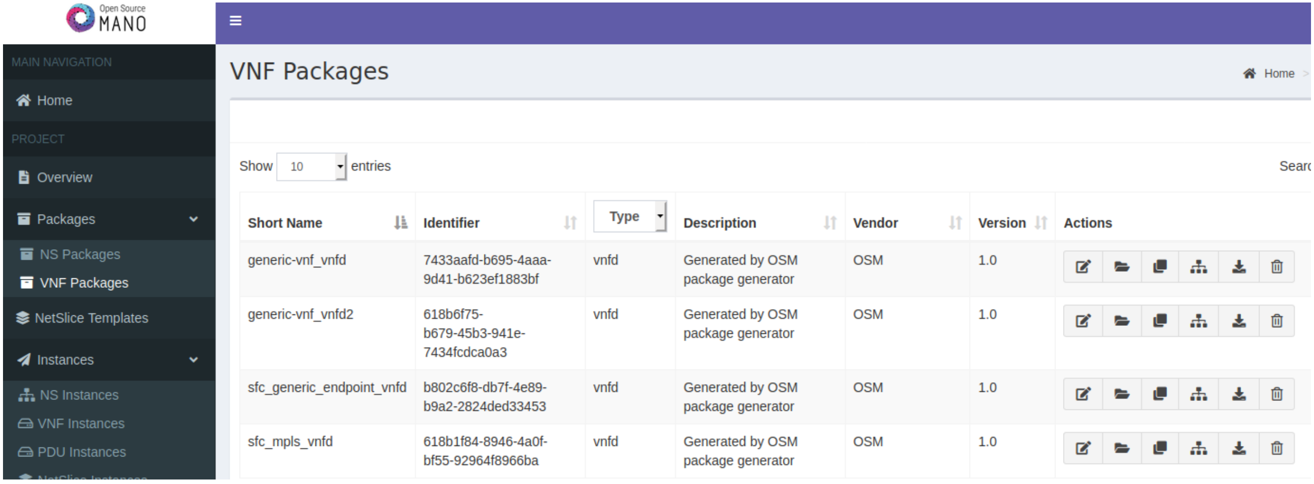
PUT https://10.1.14.248:9999/osm/vnfpkgm/v1/vnf_packages/618b6f75-b679-45b3-941e-7434fcdca0a3/package_content

○ none ○ form-data ○ x-www-form-urlencoded ○ raw ● binary ○ GraphQL BETA

generic-vnf_vnfd2.yaml ✕

Status: 204 No Content Time: 124ms Size: 272 B

FIGURE 34 TRIGGERING OF THE VNF PACKAGE CONTENT UPLOAD THROUGH THE RESTful NBI



VNF Packages

Show 10 entries

Short Name	Identifier	Type	Description	Vendor	Version	Actions
generic-vnf_vnfd	7433aafd-b695-4aaa-9d41-b623ef1883bf	vnfd	Generated by OSM package generator	OSM	1.0	
generic-vnf_vnfd2	618b6f75-b679-45b3-941e-7434fcdca0a3	vnfd	Generated by OSM package generator	OSM	1.0	
sfc_generic_endpoint_vnfd	b802c6f8-db7f-4e89-b9a2-2824ded33453	vnfd	Generated by OSM package generator	OSM	1.0	
sfc_mpls_vnfd	618b1f84-8946-4a0f-bf55-92964f8966ba	vnfd	Generated by OSM package generator	OSM	1.0	

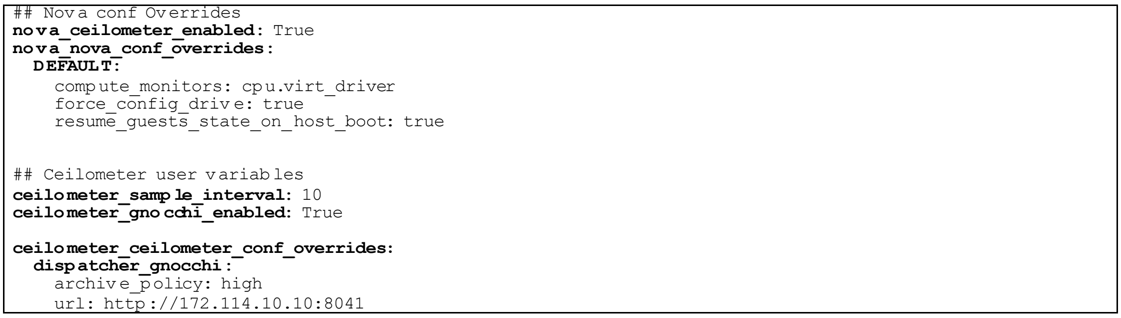
FIGURE 35 VNF PACKAGE CONTENT UPLOADED, THROUGH THE RESTful NBI, TO OSM

3.2.3.4 DYNAMIC SCALING AND TELEMETRY SERVICES

Telemetry services, in an OpenStack based VIM, allow to obtain metrics on the state of the NFVI resources, e.g. CPU or memory usage. Thereby, given those metrics, the NFV MANO can trigger scaling out or scaling in operations of the VNF instances. Note that, after the scaling out operation, new VNF instance replicas are created and the converse in scale in operations.

For the implementation of telemetry services OpenStack ceilometer is leveraged. It is the de facto telemetry service in OpenStack. It deploys agents at compute nodes that periodically poll components for different metrics. Such metrics are then gathered by a central Ceilometer agent which later transmits them to a telemetry storage service for processing and exposure. For this storage and exposure purposes we use Gnocchi. It provides RESTful APIs for metrics monitoring. Thereby, the NFVO can use these RESTful APIs to leverage the metrics in its dynamic scaling operations.

Next, we show a test that we did in the NFV testbed to gather cpu usage metrics and to trigger scaling out operations based on this cpu usage. FIGURE 36 shows the definition of the OpenStack Ansible (OSA) user variables enabling the monitoring of Compute nodes' performance, as well as a pointer for Ceilometer to the Gnocchi endpoint.

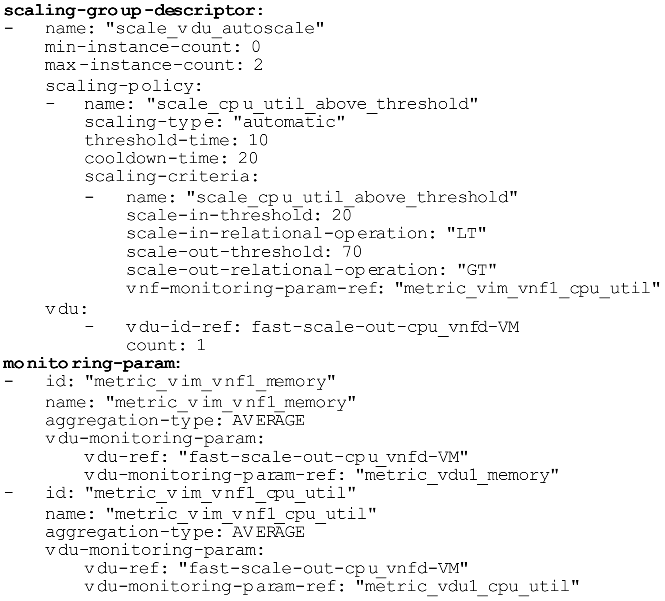


```
## Nova conf Overrides
nova_ceilometer_enabled: True
nova_nova_conf_overrides:
  DEFAULT:
    compute_monitors: cpu.virt_driver
    force_config_drive: true
    resume_guests_state_on_host_boot: true


## Ceilometer user variables
ceilometer_sample_interval: 10
ceilometer_gnocchi_enabled: True

ceilometer_ceilometer_conf_overrides:
  dispatcher_gnocchi:
    archive_policy: high
    url: http://172.114.10.10:8041
```

FIGURE 36 OPEANSTACK ANSIBLE USER VARIABLES TO CONFIGURE TELEMETRY WITH CEILOMETER AND GNOCCHI

The scale out operations are orchestrated by the NFVO, which is instructed before-hand via the VNF descriptors. That is, the VNF descriptors indicate which are the available metrics to look at from the VIM telemetry services and what are the corresponding thresholds that would unleash a scale out, or a scale in.

FIGURE 37 shows a section of a VNF descriptor that specifies the scaling out criteria, as well as the monitoring parameter that is being watched by the NFVO to comply with such criteria. In summary, the scaling out operation is triggered when the **`metric_vim_vnf1_cpu_util`** is greater than (GT) **`scale-out-threshold`** (70%) during **`threshold-time`** (10) seconds. Conversely, a scale in operation is performed on a replica VNF when the aforementioned metric is detected to be lower than (LT) **`scale-in-threshold`** (20%) for **`cooldown-time`** (20) seconds.



```
scaling-group-descriptor:
-   name: "scale_vdu_autoscale"
    min-instance-count: 0
    max-instance-count: 2
    scaling-policy:
    -   name: "scale_cpu_util_above_threshold"
        scaling-type: "automatic"
        threshold-time: 10
        cooldown-time: 20
        scaling-criteria:
        -   name: "scale_cpu_util_above_threshold"
            scale-in-threshold: 20
            scale-in-relational-operation: "LT"
            scale-out-threshold: 70
            scale-out-relational-operation: "GT"
            vnf-monitoring-param-ref: "metric_vim_vnf1_cpu_util"
    vdu:
        -   vdu-id-ref: fast-scale-out-cpu_vnfd-VM
            count: 1
monitoring-param:
-   id: "metric_vim_vnf1_memory"
    name: "metric_vim_vnf1_memory"
    aggregation-type: AVERAGE
    vdu-monitoring-param:
        vdu-ref: "fast-scale-out-cpu_vnfd-VM"
        vdu-monitoring-param-ref: "metric_vdu1_memory"
-   id: "metric_vim_vnf1_cpu_util"
    name: "metric_vim_vnf1_cpu_util"
    aggregation-type: AVERAGE
    vdu-monitoring-param:
        vdu-ref: "fast-scale-out-cpu_vnfd-VM"
        vdu-monitoring-param-ref: "metric_vdu1_cpu_util"
```

FIGURE 37 VNF descriptor for scaling out.

FIGURE 38 shows a sample OSM metrics dashboard containing panels measuring VNF's CPU and Memory usage (as defined in FIGURE 37). It can be seen in the figure how the **`semiotics_scale_out_1_fast-scale-out-cpu_vnfd-VM-1`** CPU utilization metric is maxed out for a period of time, which according to the scaling out rules should trigger the creation of a replica VNF. VM-2 and VM-3 are then created automatically and their respective metrics also appear in the figure[11].

[11] Even though VM-2 and VM-3 metrics appear in the dashboard at the same time, this does not mean the VMs were in fact created simultaneously.

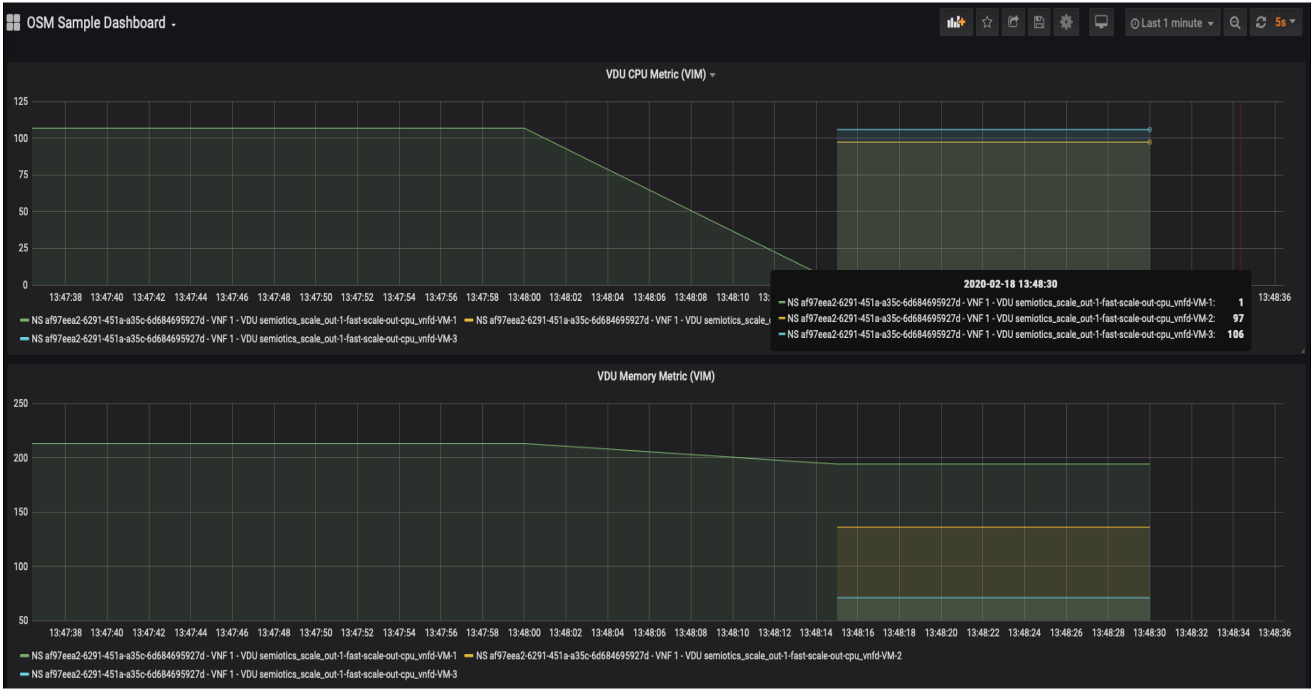

OSM Sample Dashboard
Last 1 minute
5s
VDU CPU Metric (VIM)
125
100
75
50
25
0
13:47:38
13:47:40
13:47:42
13:47:44
13:47:46
13:47:48
13:47:50
13:47:52
13:47:54
13:47:56
13:47:58
13:48:00
13:48:02
13:48:04
13:48:06
13:48:08
13:48:10
13:48:36
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_out-1-fast-scale-out-cpu_vnfd-VM-1
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_out-1-fast-scale-out-cpu_vnfd-VM-3
2020-02-18 13:48:30
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_out-1-fast-scale-out-cpu_vnfd-VM-1: 1
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_out-1-fast-scale-out-cpu_vnfd-VM-2: 97
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_out-1-fast-scale-out-cpu_vnfd-VM-3: 106
VDU Memory Metric (VIM)
250
200
150
100
50
13:47:38
13:47:40
13:47:42
13:47:44
13:47:46
13:47:48
13:47:50
13:47:52
13:47:54
13:47:56
13:47:58
13:48:00
13:48:02
13:48:04
13:48:06
13:48:08
13:48:10
13:48:12
13:48:14
13:48:16
13:48:18
13:48:20
13:48:22
13:48:24
13:48:26
13:48:28
13:48:30
13:48:32
13:48:34
13:48:36
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_out-1-fast-scale-out-cpu_vnfd-VM-1
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_out-1-fast-scale-out-cpu_vnfd-VM-2
NS af97eea2-6291-451a-a35c-6d684695927d - VNF 1 - VDU semiotics_scale_out-1-fast-scale-out-cpu_vnfd-VM-3

FIGURE 38 Scaling out of VNF instances

3.3 SEMIoTICS Field layer and Gateway

3.3.1 COMPONENT ARCHITECTURE

Field layer is, in general, responsible for hosting different types of devices (greenfield- and brownfield-devices). *Semantic-based bootstrapping & interfacing* is a set of components at the field layer that enable bootstrapping and hosting of these different types of devices. This goal is to be achieved over an interface, which is unified (applies to all types of devices) and is semantically described (eases the use of devices' data in applications).

Components related to the semantic-based bootstrapping & interfacing are marked in FIGURE 39. These components are agreed with other SEMIoTICS project partners in the work on SEMIoTICS architecture, see FIGURE 1 and Section 2.3 in SEMIoTICS deliverable D2.4.

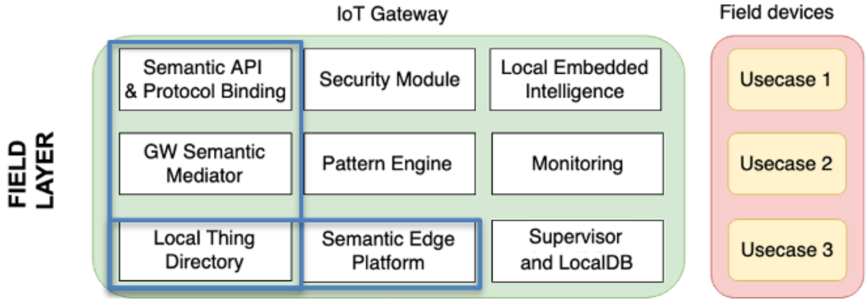


FIGURE 39: MARKED COMPONENTS RELATED TO THE SEMANTICS-BASED BOOTSTRAPPING & INTERFACING

FIGURE 40 shows the sequence diagram for semantic-bootstrapping and interfacing of field devices in SEMIoTICS. In the bootstrapping process we distinguish two different classes of device. The first class consists of devices that already have a Web-based RESTful interface and are described by W3C Thing Description.

The second class comprise of all other devices that yet need to be made accessible over a Web-based RESTful interface. These devices do not have a semantic description, or it exists, but needs to be mapped to standardized semantic IoT models. This is a case, for example, with brownfield devices. In order to realize IoT applications, it is convenient to map these brownfield descriptions into description based on standardized IoT semantic models.

Let us consider now a sequence diagram of activities that occur during the bootstrapping of WoT device, see FIGURE 40. The user performs the first step during the initialization of a new device. This assumes provision of information such as an IP address, device capability, domain of use, location etc. Since the device already has a Thing Description (TD), this information is directly put in its TD. The device can then be registered with SEMIoTICS IIoT Gateway (with GW Semantic Mediator, which is an internal component of the Gateway). The Mediator will create a Device Node for each interaction pattern of the device. Device Node is a programmable component that enables interaction with the device. The Mediator automatically generates Device Nodes, based solely on information from devices' TDs. Generated Device Nodes can then be installed in Semantic Edge Platform. From that moment on, the device can be used in Edge and Cloud-based applications via interactions provided by its Device Nodes. Thus, Device Nodes represent a means for an application to programmatically access device's functionality via Semantic Edge Platform. The Mediator automatically generates Device Nodes, based solely on information from devices' TDs. Generated Device Nodes can then be installed in Semantic Edge Platform. From that moment on, the device can be used in Edge and Cloud-based applications via interactions provided by its Device Nodes.

In comparison to the sequence diagram for bootstrapping and interfacing of field devices in deliverable D3.3, here we have added a component called Semantic Edge Platform (SME), see Section 3.5.8 in deliverable D2.5. SME eases the interaction with SEMIoTICS IoT Gateway (graphic user interface for network scanning and user input, e.g., IP address range etc.), including the interaction with Local Thing Directory too.

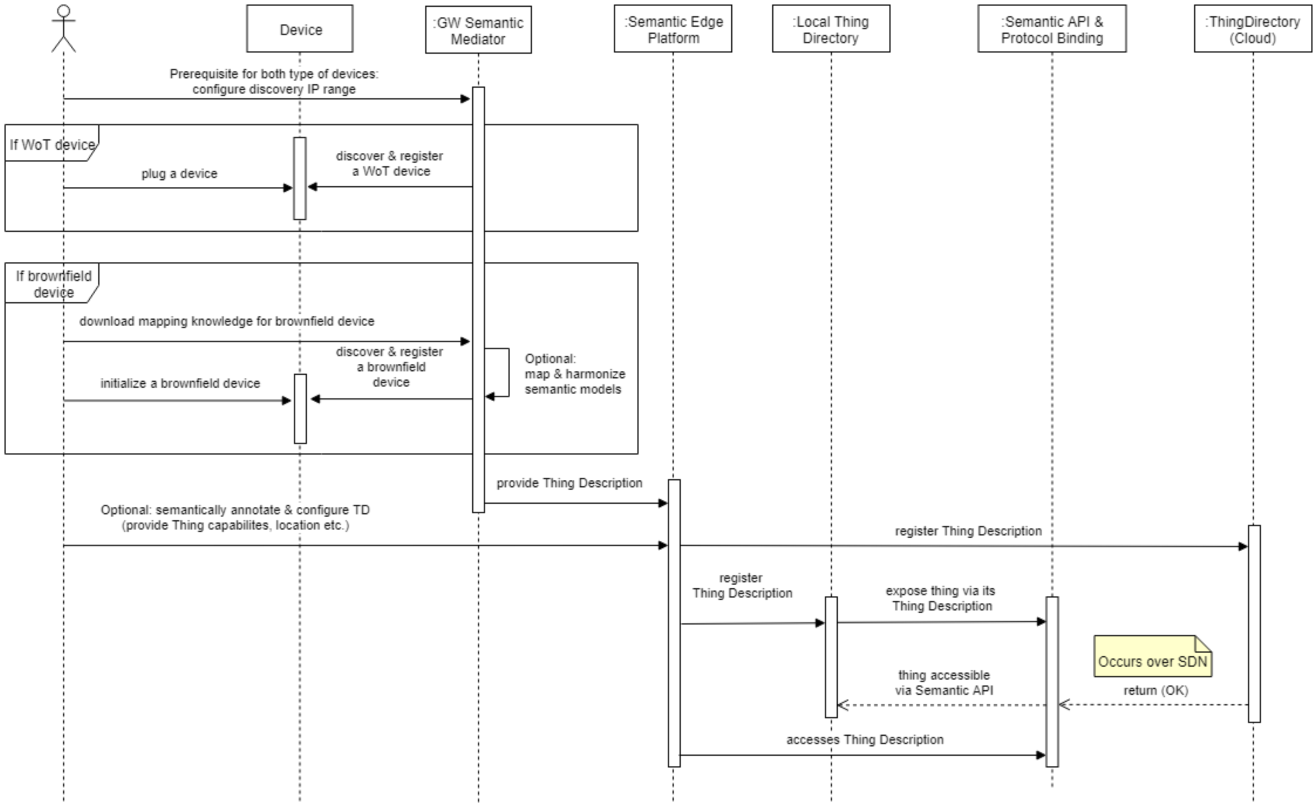

Device
:GW Semantic Mediator
:Semantic Edge Platform
:Local Thing Directory
:Semantic API & Protocol Binding
:ThingDirectory (Cloud)
Prerequisite for both type of devices: configure discovery IP range
If WoT device
plug a device
discover & register a WoT device
If brownfield device
download mapping knowledge for brownfield device
discover & register a brownfield device
initialize a brownfield device
Optional: map & harmonize semantic models
provide Thing Description
Optional: semantically annotate & configure TD (provide Thing capabilites, location etc.)
register Thing Description
register Thing Description
expose thing via its Thing Description
Occurs over SDN
thing accessible via Semantic API
return (OK)
accesses Thing Description

FIGURE 40: SEQUENCE DIAGRAM FOR SEMANTIC-BOOTSTRAPPING AND INTERFACING OF FIELD DEVICES IN SEMIOTICS

If a brownfield device needs to be initialized, then a user in addition to previously mentioned information needs to specify metadata related to the communication protocol and the encoding format. This information will be important part of a Thing Description and is used by SEMIoTICS IoT Gateway to realize the protocol binding. At this point in time brownfield integration has not been provided yet.

It is worth of noting that certain activities in the sequence diagram are accomplished over Software Defined Network (SDN), see FIGURE 40. Thus, we tested the integration of SEMIoTICS IoT Gateway with SEMIoTICS Software Defined Networking Controller, see Section 3.1.

In the following we will detail each component in regard to its currently available and deployable functionality.

3.3.1.1 SEMANTIC API & PROTOCOL BINDING

Semantic API & Protocol Binding is a component responsible for binding different protocol and exposing common semantic API located at the Generic IoT Gateway layer. This functionality is needed in order to integrate brownfield devices into a common IoT access layer. Technology-wise, the functionality is based on W3C Web of Things (WoT) API[12].

In the following we give API that is implemented and tested in SEMIoTICS IoT Gateway (GW). The current implementation is focused only on greenfield devices, i.e., no protocol binding is required yet.

SEMIoTICS greenfield device (e.g., Raspberry Pi with attached an IP camera) implements the following interface for starting the camera in a WoT servient, see TABLE 1. Note that a method for starting a camera is semantically annotated with iotschema.org mark-up for a camera[13].

TABLE 1: WOT SERVIENT – GREENFIELD DEVICE INTERFACE

```
let thing = WoT.produce({
title: "SEMIoTICS Thing",
description: "Camera",
"@context": ["https://www.w3.org/2019/wot/td/v1", {"iot": "http://iotschema.org/" }],
   "@type": "iot:StartRecording",
   "iot:capability": "iot:Camera",
   actions: {
     startCamera: {
        description: "Start recording the video.)"
     }
   }
});
thing.setActionHandler(
   "startCamera",
   (params, options) => {
      // Code that implements an Action, e.g., "startCamera".
      // Removed for the sake of simplicity.
   });
```

The greenfield device does not require the protocol binding. In the next version of this deliverable we will extend the servient to support a protocol binding for a brownfield device.

[12] https://www.w3.org/TR/wot-scripting-api/
[13] http://iotschema.org/Camera

A client, which needs to access a newly plugged (greenfield) device, can access Thing Description (TD) of the plugged device in order to discover its functionality. This can be achieved by providing the IP address of the device in the method `fetch`, see TABLE 2.

TABLE 2: FETCHING A THING DESCRIPTION

```
WoTHelpers.fetch("http://semiotics.things.org:8080/ipcamera").then( async (td) => {
    let thing = await WoT.consume(td);
}).catch( (err) => { console.error("Fetch error:", err); });

```

By examining the fetched TD a client, for example finds an action called `startCamera`. The client can then use the API to invoke the action, see TABLE 3.

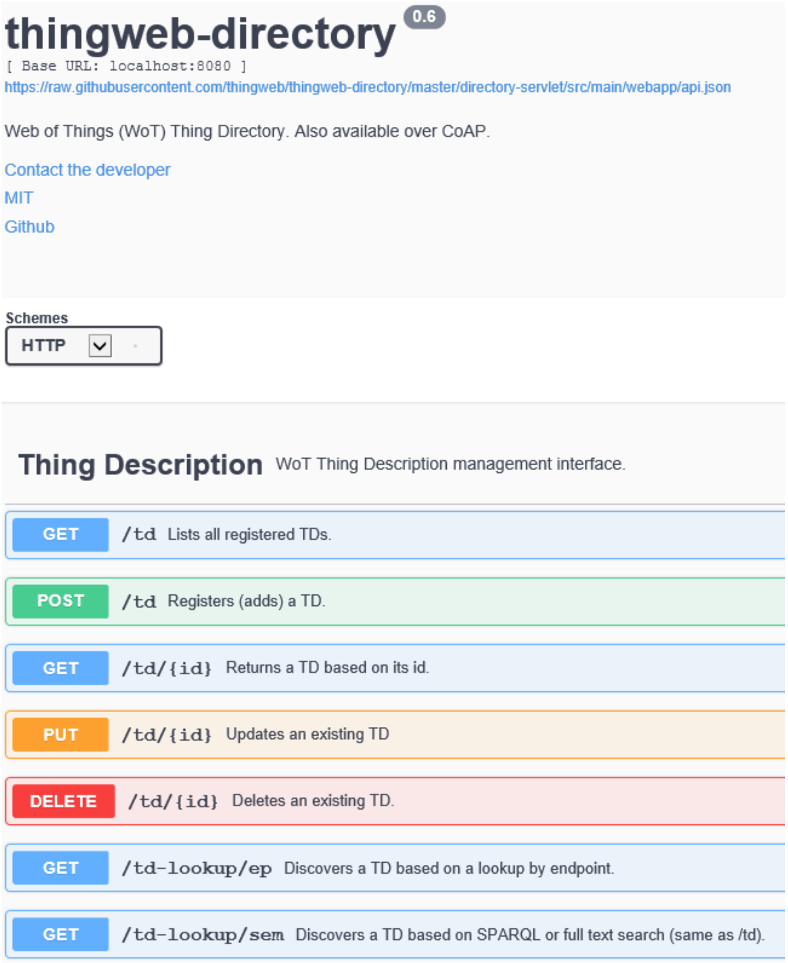
TABLE 3: INTERACTING WITH THING (CAMERA)

```
// start the camera
await thing.invokeAction("startCamera");
```

3.3.1.2 LOCAL THING DIRECTORY

The purpose of Local Thing Directory is to store semantic description of Things locally in the Generic IoT Gateway. In Section 3.3.1.1 we have seen how the gateway (or any other client) can retrieve a Thing Description (TD). During the device registration process, the gateway stores a TD in the Local Thing Directory. For the implementation of this component we use the open source implementation from W3C[14]. FIGURE 41 shows the API available from our local deployment of Thing Directory in the GW.

[14] https://github.com/thingweb/thingweb-directory



thingweb-directory 0.6

[Base URL: localhost:8080]

https://raw.githubusercontent.com/thingweb/thingweb-directory/master/directory-servlet/src/main/webapp/api.json

Web of Things (WoT) Thing Directory. Also available over CoAP.

Contact the developer

MIT

Github

Schemes

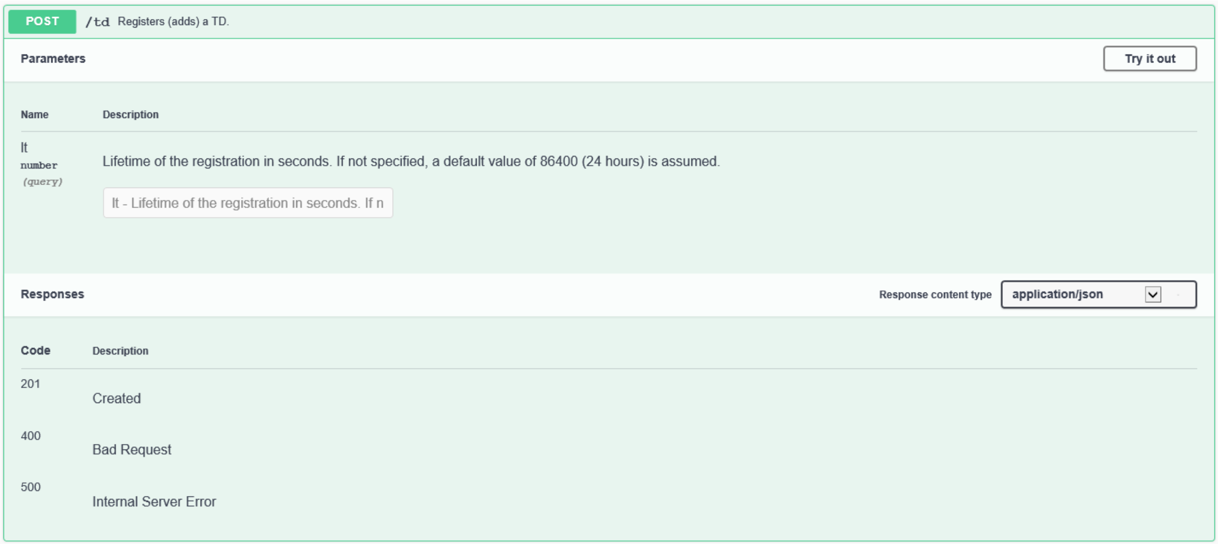
HTTP

Thing Description WoT Thing Description management interface.

GET	/td	Lists all registered TDs.
POST	/td	Registers (adds) a TD.
GET	/td/{id}	Returns a TD based on its id.
PUT	/td/{id}	Updates an existing TD
DELETE	/td/{id}	Deletes an existing TD.
GET	/td-lookup/ep	Discovers a TD based on a lookup by endpoint.
GET	/td-lookup/sem	Discovers a TD based on SPARQL or full text search (same as /td).

FIGURE 41: OVERVIEW OF THING DIRECTORY API

FIGURE 42 details the API related to the Thing Description registration. After fetching a TD (see TABLE 2), the GW uses this method to store a TD in Local Thing Directory.



POST `/td` Registers (adds) a TD.

Parameters

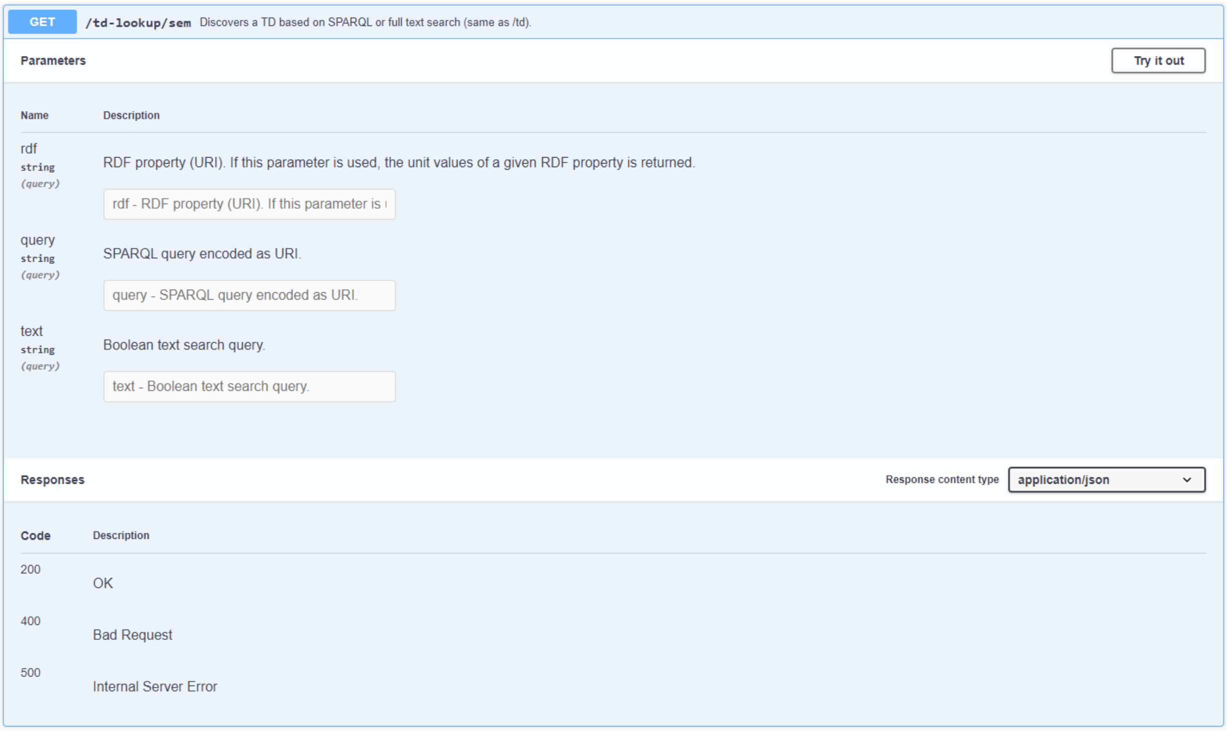
Name	Description
lt number *(query)*	Lifetime of the registration in seconds. If not specified, a default value of 86400 (24 hours) is assumed. lt - Lifetime of the registration in seconds. If n

Responses

Code	Description
201	Created
400	Bad Request
500	Internal Server Error

FIGURE 42: API FOR REGISTRATION OF THING DESCRIPTION

An existing TD can be discovered from Local Thing Directory, e.g., via a SPARQL query, see FIGURE 43.



GET `/td-lookup/sem` Discovers a TD based on SPARQL or full text search (same as /td).

Parameters

Name	Description
rdf string *(query)*	RDF property (URI). If this parameter is used, the unit values of a given RDF property is returned. rdf - RDF property (URI). If this parameter is u
query string *(query)*	SPARQL query encoded as URI. query - SPARQL query encoded as URI.
text string *(query)*	Boolean text search query. text - Boolean text search query.

Responses

Code	Description
200	OK
400	Bad Request
500	Internal Server Error

FIGURE 43: API FOR DISCOVERY OF THING DESCRIPTION

3.3.1.3 GW SEMANTIC MEDIATOR

The goal of semantic integration (see SEMIoTICS deliverable D3.3) is to enable realization of new IoT applications that have not been envisioned at the time of engineering of an existing automation system. In the current implementation (for greenfield devices) the mediator does need to integrate device's semantics as the device already has (semantically annotated) Thing Description. Thus, its function is just to make a device programmatically accessible in accordance with the meta-data from Thing Description. It means that for each interaction pattern from the TD, GW Semantic Mediator will create a Device Node of the device. Device Node is a programmable component that enables interaction with the device. For example, if a TD of a device contains inclinometer property and an action for IP camera, then the mediator may generate two nodes: one for reading the current inclination data, and another one for streaming the video from the camera. So generated Device Nodes can be installed in Semantic Edge Platform and used in Edge and Cloud-based applications. TABLE 4 shows our script that generates and installs Device Nodes.

TABLE 4: CREATING DEVICE NODES

```
npm install
rm -rf ~/.node-red/package-lock.json
rm -rf GeneratedNodes/*
for file in IPshapes/* ; do
    node  NodeGen.js --file=$file
    sleep 2
done
mkdir -p ~/.node-red/SchemaNodes
for d in GeneratedNodes ; do
    cp -R $d ~/.node-red/SchemaNodes/
done
npm install --prefix ~/.node-red ~/.node-red/SchemaNodes/GeneratedNodes/*
npm install
rm -rf ~/.node-red/package-lock.json
rm -rf GeneratedNodes/*
for file in IPshapes/* ; do
    node  NodeGen.js --file=$file
    sleep 2
done
mkdir -p ~/.node-red/SchemaNodes
for d in GeneratedNodes ; do
    cp -R $d ~/.node-red/SchemaNodes/
done
npm install --prefix ~/.node-red ~/.node-red/SchemaNodes/GeneratedNodes/*
```

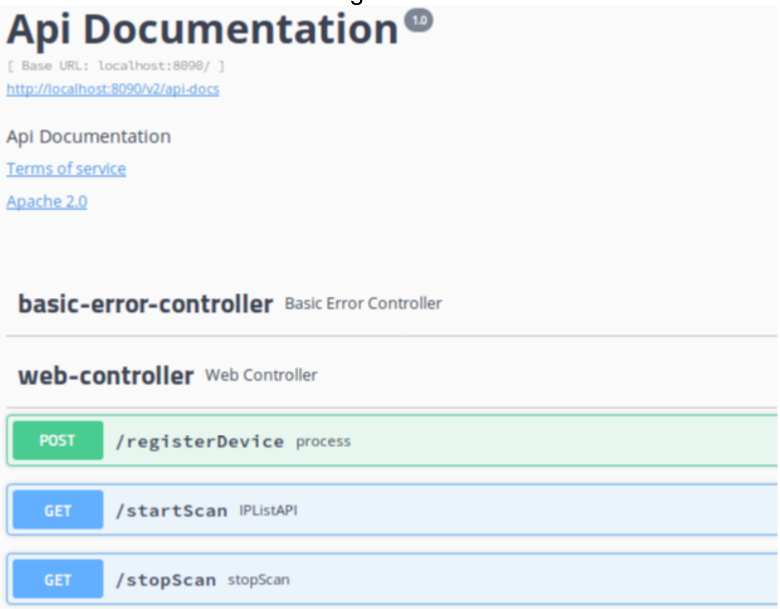
In order to integrate brownfield devices, in the next version of the mediator we will integrate semantics from existing brownfield devices into IoT semantic model (our model is iotschema.org[15]). If a brownfield device does not have any semantic description, then we will need to make sure that iotschema.org covers necessary semantics to create semantic description for that device. The mediator will provide Semantic Nodes (in addition to Device Nodes). Semantic Nodes will be graphic components that are available in Semantic Edge Platform for the purpose of creating semantically annotated Thing Description for a brownfield device. They will be automatically generated from the IoT semantic model (iotschema.org), and will enable a user (without expertise in semantic technologies) to configure a device, i.e., to choose offered values from the semantic model via a graphic component. Once a user has semantically configured a brownfield device over the Semantic Nodes, the mediator will automatically generate a Thing Description. From that moment on, it will be possible to discover a device over its TD and Local Thing Directory. Further on, it will be possible to access the device over API & Protocol Binding. To this goal, we work on a common semantic access layer between brownfield devices and new IoT devices.

3.3.1.4 SEMANTIC EDGE PLATFORM

Semantic Edge Platform (SME) provides a convenient user interface for different components of IoT Gateway, which are accessible over API but not necessarily have a user interface. Thus, for example SME enables a user to configure SEMIoTICS IoT Gateway, choose a network interface, define an IP address range when

[15] http://iotschema.org/docs/full.html

scanning a network for new devices, and initiate the device bootstrapping process. FIGURE 44 shows API of IoT Gateway, which is accessible over Semantic Edge Platform.



Api Documentation 1.0

[Base URL: localhost:8090/]

http://localhost:8090/v2/api-docs

Api Documentation

Terms of service

Apache 2.0

basic-error-controller Basic Error Controller

web-controller Web Controller

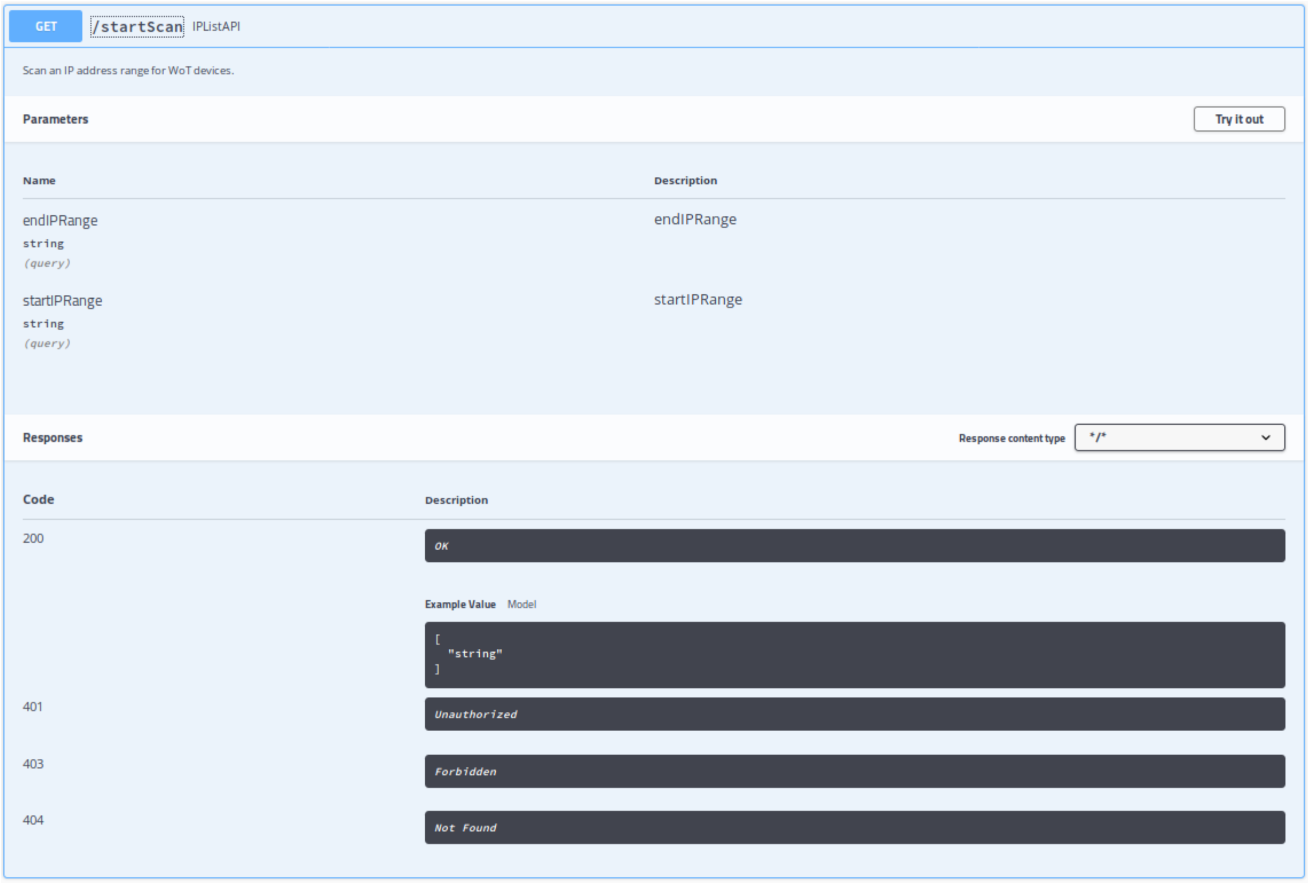
POST /registerDevice process

GET /startScan IPListAPI

GET /stopScan stopScan

FIGURE 44: API OF IOT GATEWAY EXPOSED OVER SEMANTIC EDGE PLATFORM

FIGURE 45 presents the method that is used for scanning a network, where new devices are to be bootstrapped.



Scan an IP address range for WoT devices.

Parameters

Name	Description
endIPRange string *(query)*	endIPRange
startIPRange string *(query)*	startIPRange

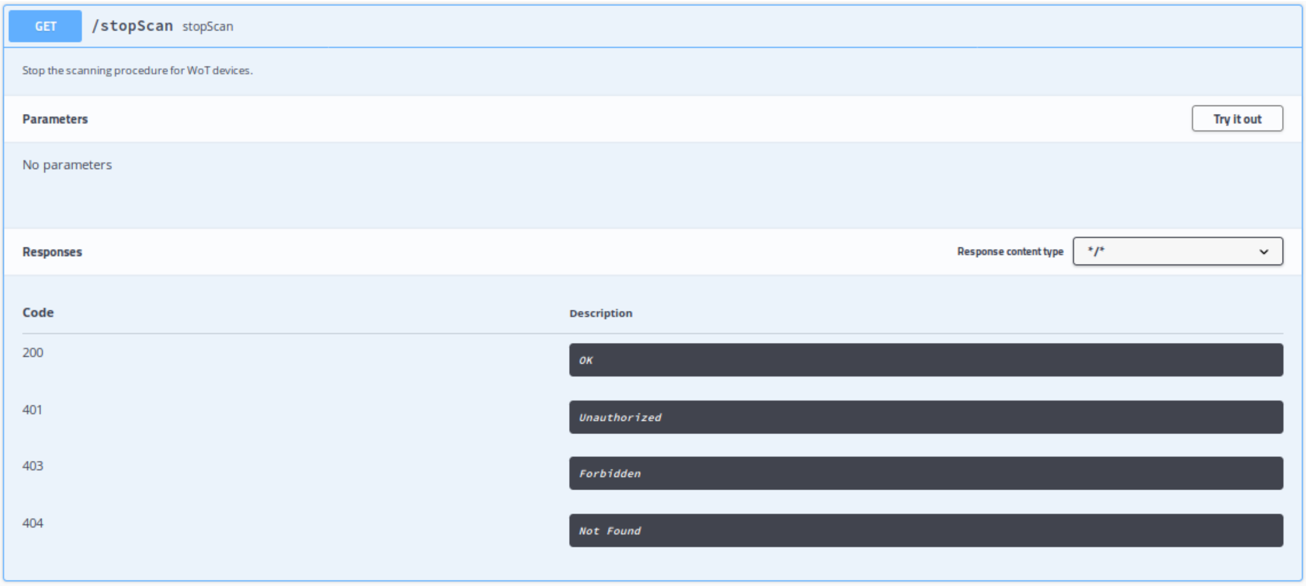
Responses

Response content type */*

Code	Description
200	OK Example Value ["string"]
401	Unauthorized
403	Forbidden
404	Not Found

FIGURE 45: API OF IOT GATEWAY TO START BOOTSTRAPPING

FIGURE 46 shows the API, which enables a user to terminate the network scanning. For large networks this process may take a long time. On the other hand, sometimes a user knows, which devices are supposed to appear during the scan. Thus, as soon as new devices appear, the process may be halted.



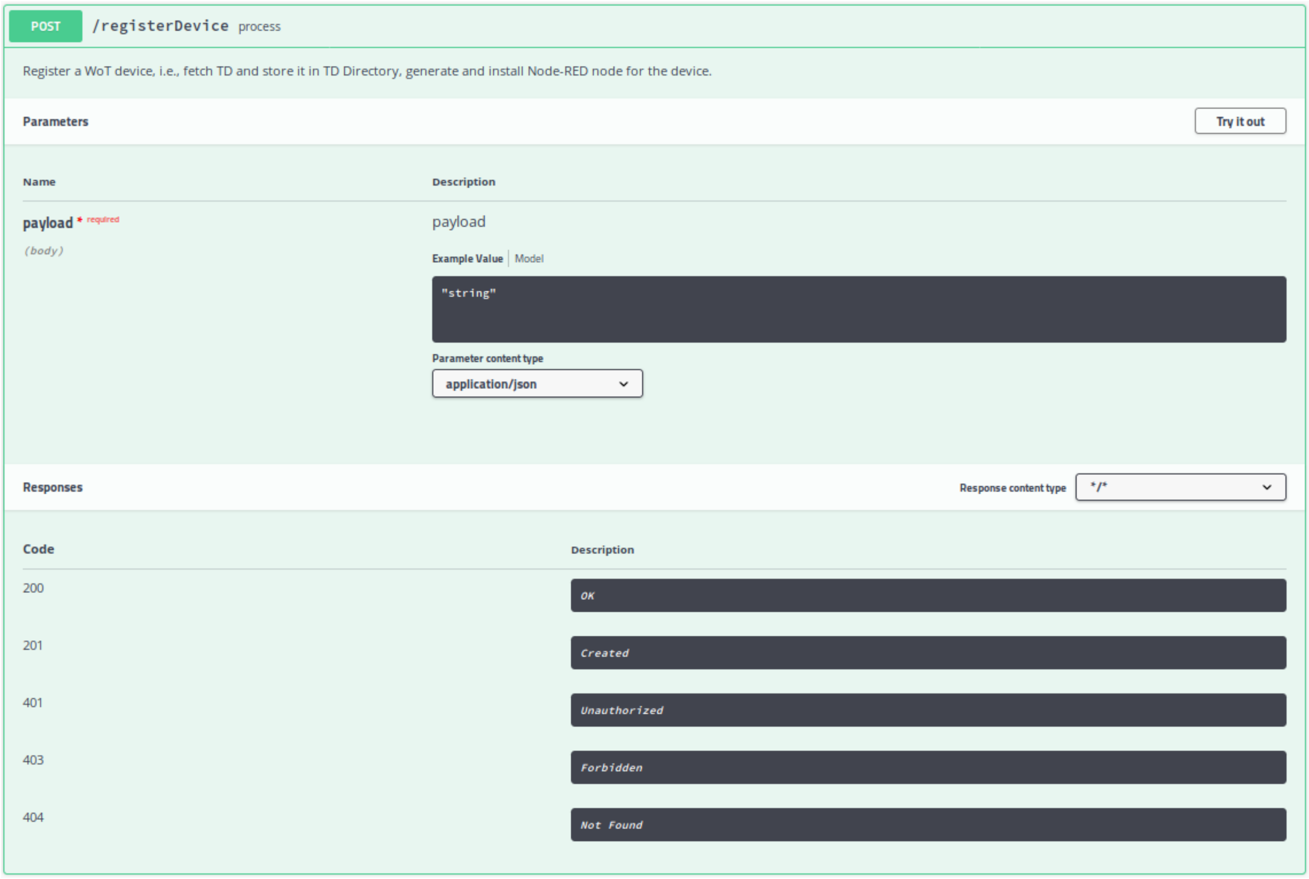
Stop the scanning procedure for WoT devices.

No parameters

Code	Description
200	OK
401	Unauthorized
403	Forbidden
404	Not Found

FIGURE 46: API OF IOT GATEWAY TO STOP SCANNING THE NETWORK

FIGURE 47 depicts the API for registering each new device. The method interacts with other gateway's component, i.e., it fetches device's Thing Description, stores it in TD Directory, invokes the GW Semantic Mediator to create a Device Node for the device, and finally installs the node in SME, thereby making it programmatically accessible in SME over WoT API.



POST /registerDevice process

Register a WoT device, i.e., fetch TD and store it in TD Directory, generate and install Node-RED node for the device.

Parameters

Name	Description
payload * required (body)	payload "string" Parameter content type: application/json

Responses

Response content type: */*

Code	Description
200	OK
201	Created
401	Unauthorized
403	Forbidden
404	Not Found

FIGURE 47: API OF IOT GATEWAY TO REGISTER A NEW WOT DEVICE

3.3.2 TESTING METHODOLOGY

In order to test the IoT Gateway with all its components we have deployed it on Siemens SIMATIC IPC227E (Nanobox), which is an industrial PC with Intel(R) Celeron(R) CPU N2930 @ 1.83GHz × 4, 8 GB RAM, and Ubuntu Linux 18.04. Power Supply makes sure that the Nanobox is powered with enough electricity supply. Further on, a WoT Device, which is to be bootstrapped was implemented on Raspberry Pi 3 B+ with BCM28370B SoC: 1,4 GHz, quad-core ARM-Cortex A53 CPU, 1 GB RAM, and 5 GHz WLAN 802.11 ac, 2,4 GHz WLAN 802.11 b/g/n. WoT Device has an IP 8 Megapixel camera with video 1920 x 1080 Pixel @ 30 fps. Both Nanobox and WoT Device are connected over the same SDN network (see Section 3.1). FIGURE 48 shows this deployment set up.

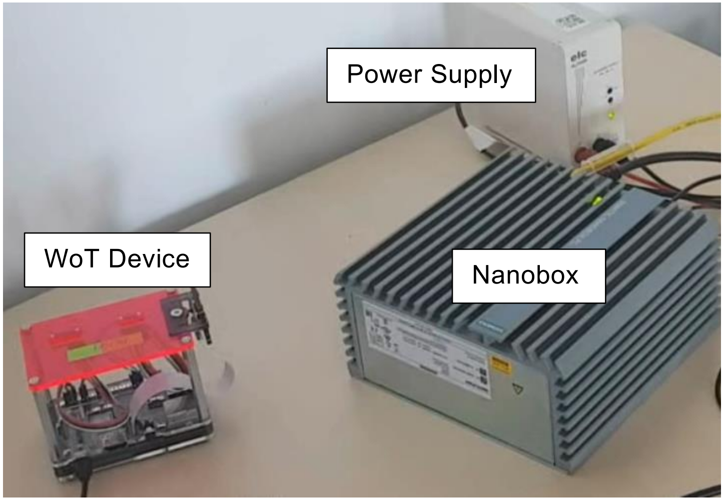


FIGURE 48: INITIAL BOOTSTRAPPING DEPLOYMENT TOWARDS USE CASE 1 REALIZATION

Once the WoT Device is powered up it will connect itself (over WLAN) to the network. Nanobox is also running in the same network and is ready to bootstrap newly plugged devices. In order to test the process, we have developed a set of API tests, see TABLE 5.

TABLE 5: IOT GATEWAY API TESTS

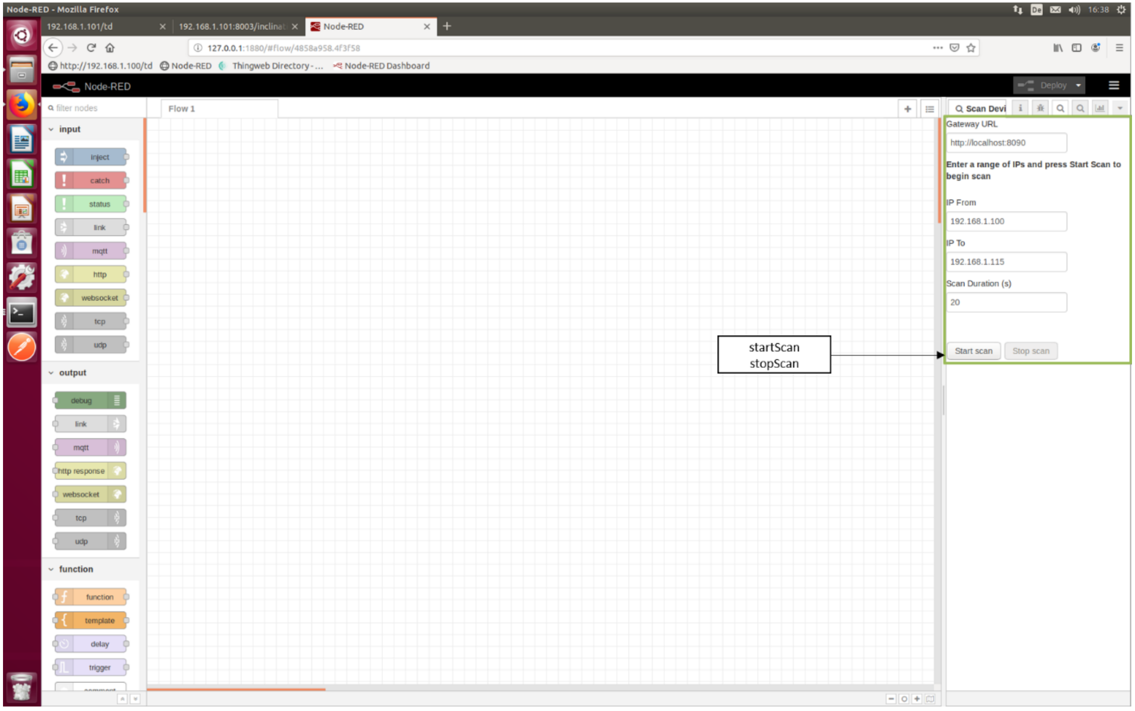
Semantic API & Protocol Binding
fetch
✓ should reject with 404 error when Thing Description is not available on WoT device at default location (IP_Address/td); ✓ should fetch Thing Description from WoT device with 200 OK.
invokeAction
✓ should reject with 404 error if the action does not exist; ✓ should reject with 400 error if the action was not called correctly (e.g., wrong parameters); ✓ should invoke an action on a WoT Device, e.g., start a camera, with 200 OK status.
Local Thing Directory
td
✓ should reject with 400 when attempting to register a device with incorrect Thing Description; ✓ should reject with 500 when attempting to register a device and an Internal Server Error occurs in Local Thing Directory; ✓ should create an entity by id, which points to registered Thing Description (with 201 OK status).
td-lookup/sem
✓ should reject with 400 when attempting to query Local Thing Directory with an incorrect semantic query; ✓ should reject with 500 when attempting to query Local Thing Directory and an Internal Server Error occurs; ✓ should return results in response to a semantic query (with 200 OK status).
GW Semantic Mediator
install
✓ should reject with error FALSE when a new Device Node cannot be created and installed in SME; ✓ should create a Device Node and install it in SME (with status TRUE).
Semantic Edge Platform
startScan
✓ should reject with 401 error if the network scanning is unauthorized;

✓ should reject with 403 error if the network scanning is forbidden (e.g., a request is temporally rejected by the gateway); ✓ should reject with 400 error if the request is badly formed; ✓ should start scanning the network for new devices with the status 200 OK.
stopScan
✓ should reject with 401 error if the network scanning is unauthorized; ✓ should reject with 403 error if the network scanning is forbidden (e.g., a request is temporally rejected by the gateway); ✓ should reject with 400 error if the request is badly formed; ✓ should stop scanning the network for new devices with the status 200 OK.
registerDevice
✓ should reject with 401 error if the access to the device is unauthorized; ✓ should reject with 403 error if the device is not accessible (e.g., temporally not available); ✓ should reject with 400 error if the request is badly formed; ✓ should register a new device with the status 200 OK.

3.3.3 PERFORMANCE TEST AND KPI VALIDATION

In this section we present results of a typical testing workflow, i.e., what happens when a user plugs a new WoT device. These results are not really performance test. As the bootstrapping occurs prior to the run time of an application, the performance of the process (in a sense of run time) is not important. Rather we are interested in in a bootstrapping process that gets completed with minimal user intervention. Thus, in this section we will present a testing procedure we have completed to validate the implementation of IoT Gateway.

A user starts the interaction with the gateway over Semantic Edge Platform. Methods startScan and stopScan, see TABLE 5 and FIGURE 49, are first to be tested. After defining the network range to be scanned, the user hits the “Start scan” button (FIGURE 49) and gets a list of new devices to be registered by the gateway. This occurs if the test goes well. Otherwise the test produces errors as specified in TABLE 5.



Gateway URL

http://localhost:8090

Enter a range of IPs and press Start Scan to begin scan

IP From

192.168.1.100

IP To

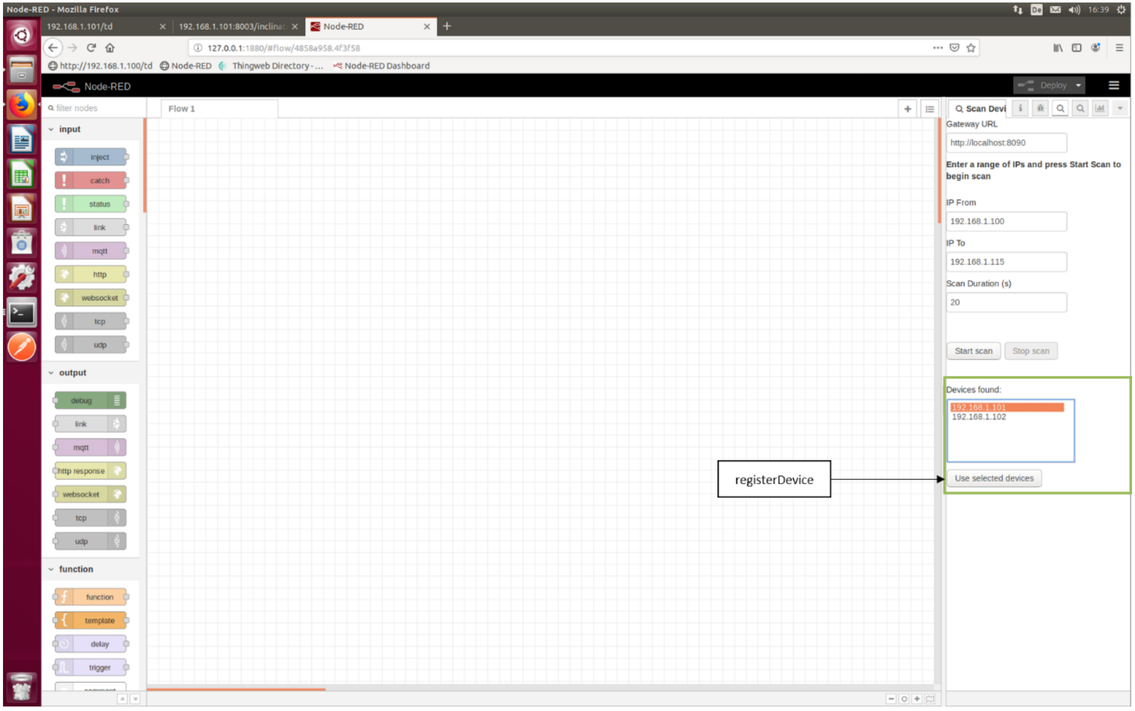
192.168.1.115

Scan Duration (s)

20

FIGURE 49: TESTING IOT GATEWAY FROM SEMANTIC EDGE PLATFORM

FIGURE 50 shows the results from the network scanning. The user may now choose a device and press the “Use selected devices” button. The method to be executed and tested is now registerDevice, see TABLE 5 and FIGURE 49. The method will try to fetch a Thing Description (TD) from the device, pass it to GW Semantic Mediator, and store it in Local Thing Directory. If the process goes well, the status 200 OK will be returned. Otherwise an error will be thrown, see TABLE 5.



Gateway URL: http://localhost:8090

Enter a range of IPs and press Start Scan to begin scan

IP From: 192.168.1.100

IP To: 192.168.1.115

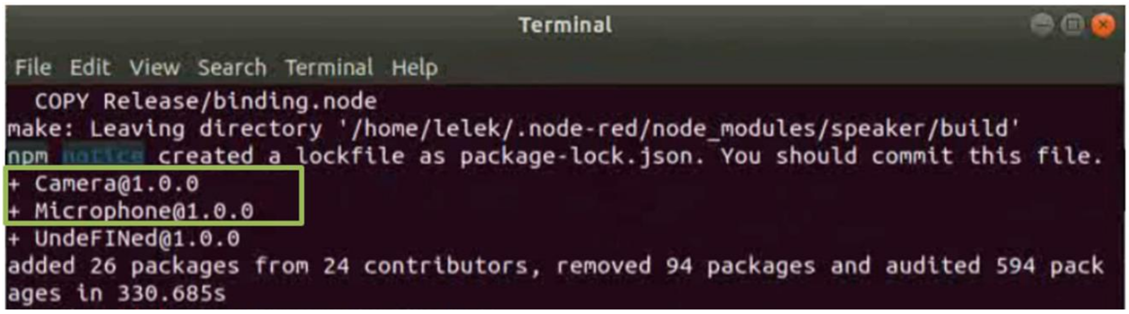
Scan Duration (s): 20

Devices found:
192.168.1.101
192.168.1.102

FIGURE 50: RESULT FROM NETWORK SCANNING

Once GW Semantic Mediator receives a TD, it will try to generate one or more Device Nodes and install them in Semantic Edge Platform. This process is automated as all semantic meta-data needed for Device Nodes generation are contained in the TD. After installation of nodes, a user may start interacting with the device, i.e., to start using it in an application.

The method to be tested during this process is "install", see TABLE 5. Since this component does not have a RESTful interface, the result is shown in command line (see FIGURE 51 after installing a camera and microphone from Use Case 1).

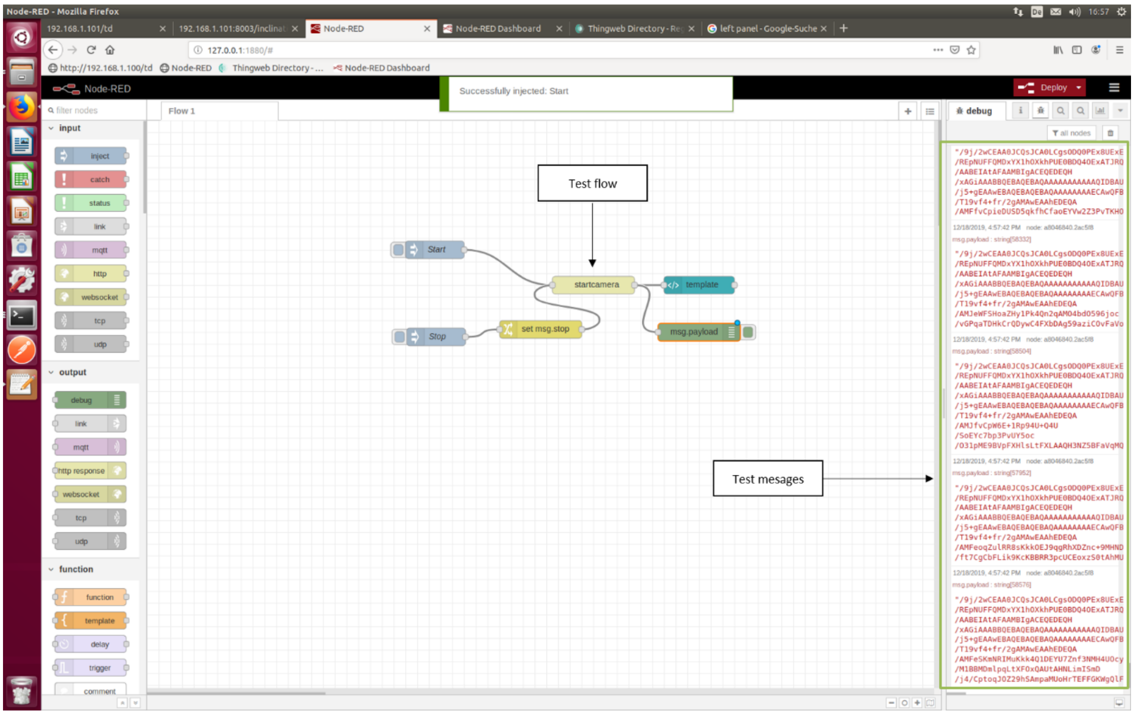


```
  COPY Release/binding.node
make: Leaving directory '/home/lelek/.node-red/node_modules/speaker/build'
npm notice created a lockfile as package-lock.json. You should commit this file.
+ Camera@1.0.0
+ Microphone@1.0.0
+ UndeFINed@1.0.0
added 26 packages from 24 contributors, removed 94 packages and audited 594 pack
ages in 330.685s
```

FIGURE 51: RESULT FROM GW SEMANTIC MEDIATOR

FIGURE 52 shows a Device Node for device that has a camera. After bootstrapping the device, a node called "startcamera" is available and can be tested in an application flow as shown in FIGURE 52. Of course, the node "stopcamera" has been generated and installed as well.

FIGURE 52 also shows test messages after starting the camera, see the debug console. This is a standard way to test a node in Node-RED. Alternately we have tested the camera using the Node-RED Dashboard, see FIGURE 53.



Successfully injected: Start

Test flow

Test mesages

"/9j/2wCEAA0JCQsJCA0LCgsODQ0PEx8UExE
/REpNUFFQMDxYX1hOXkhPUE0BDQ4OExATJRQ
/AABEIAtAFAAMBIgACEQEDEQH
/xAGiAAABBQEBAQEBAQAAAAAAAAAAAQIDBAU
/j5+gEAAwEBAQEBAQEBAQAAAAAAAAECAwQFB
/T19vf4+fr/2gAMAwEAAhEDEQA
/AMFfvCpieDUSD5qkfhCfaoEYVw2Z3PvTKHO
12/18/2019, 4:57:42 PM node: a8046840.2ac5f8
msg.payload : string[58332]
"/9j/2wCEAA0JCQsJCA0LCgsODQ0PEx8UExE
/REpNUFFQMDxYX1hOXkhPUE0BDQ4OExATJRQ
/AABEIAtAFAAMBIgACEQEDEQH
/xAGiAAABBQEBAQEBAQAAAAAAAAAAAQIDBAU
/j5+gEAAwEBAQEBAQEBAQAAAAAAAAECAwQFB
/T19vf4+fr/2gAMAwEAAhEDEQA
/AMJeWFSHoaZHy1Pk4Qn2qAM04bd0596joc
/vGPqaTDHkCrQDywC4FXbDAg59aziCOvFaVo
12/18/2019, 4:57:42 PM node: a8046840.2ac5f8
msg.payload : string[58504]
"/9j/2wCEAA0JCQsJCA0LCgsODQ0PEx8UExE
/REpNUFFQMDxYX1hOXkhPUE0BDQ4OExATJRQ
/AABEIAtAFAAMBIgACEQEDEQH
/xAGiAAABBQEBAQEBAQAAAAAAAAAAAQIDBAU
/j5+gEAAwEBAQEBAQEBAQAAAAAAAAECAwQFB
/T19vf4+fr/2gAMAwEAAhEDEQA
/AMJfvCpW6E+1Rp94U+Q4U
/SoEYc7bp3PvUY5oc
/O31pME9BVpFXHlsLtFXLAAQH3NZ5BFaVqMQ
12/18/2019, 4:57:42 PM node: a8046840.2ac5f8
msg.payload : string[57952]
"/9j/2wCEAA0JCQsJCA0LCgsODQ0PEx8UExE
/REpNUFFQMDxYX1hOXkhPUE0BDQ4OExATJRQ
/AABEIAtAFAAMBIgACEQEDEQH
/xAGiAAABBQEBAQEBAQAAAAAAAAAAAQIDBAU
/j5+gEAAwEBAQEBAQEBAQAAAAAAAAECAwQFB
/T19vf4+fr/2gAMAwEAAhEDEQA
/AMFeoqZulRR8sKkOEJ9qgRhXZnc+9MHND
/ft7CgCbFLik9KcKBBBRR3pcUCEEoxzS0tAhMU
12/18/2019, 4:57:42 PM node: a8046840.2ac5f8
msg.payload : string[58576]
"/9j/2wCEAA0JCQsJCA0LCgsODQ0PEx8UExE
/REpNUFFQMDxYX1hOXkhPUE0BDQ4OExATJRQ
/AABEIAtAFAAMBIgACEQEDEQH
/xAGiAAABBQEBAQEBAQAAAAAAAAAAAQIDBAU
/j5+gEAAwEBAQEBAQEBAQAAAAAAAAECAwQFB
/T19vf4+fr/2gAMAwEAAhEDEQA
/AMFeSKmNRIMukk4Q1DEYU7Znf3NMH4UOcy
/M1BBMDmlpqLtXFOxQAUtAHNLimISmD
/j4/CptoqJOZ29hSAmpaMUTaTEFGKWgUF

FIGURE 52: TEST FLOW FOR CAMERA

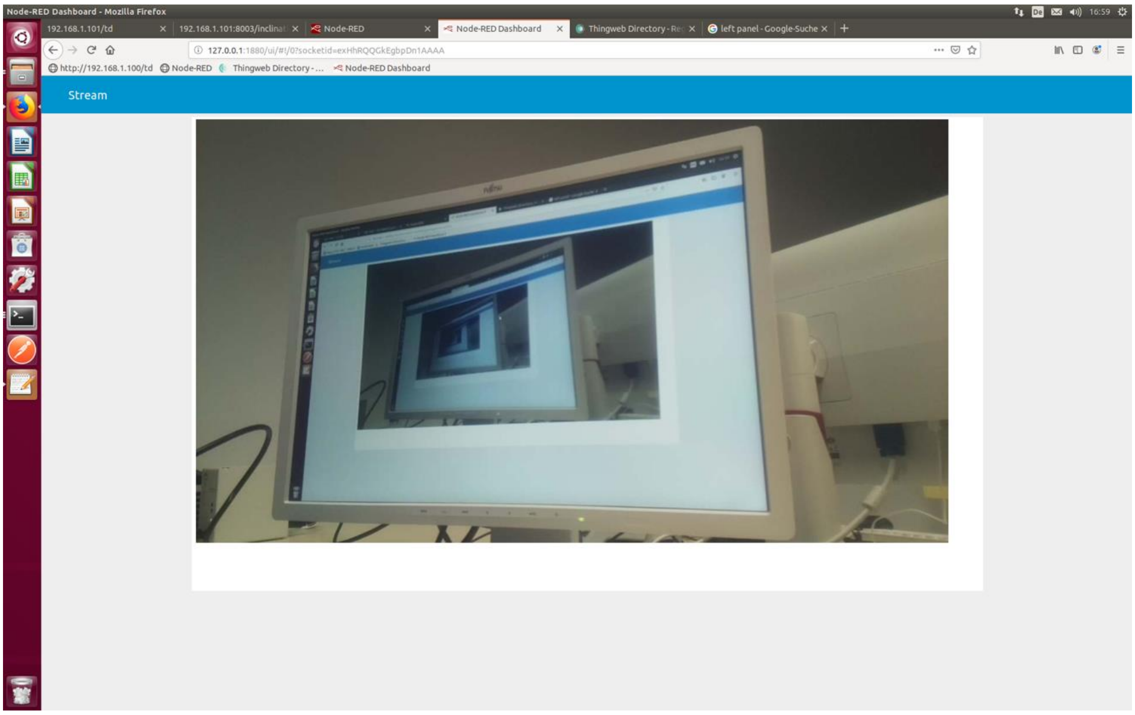
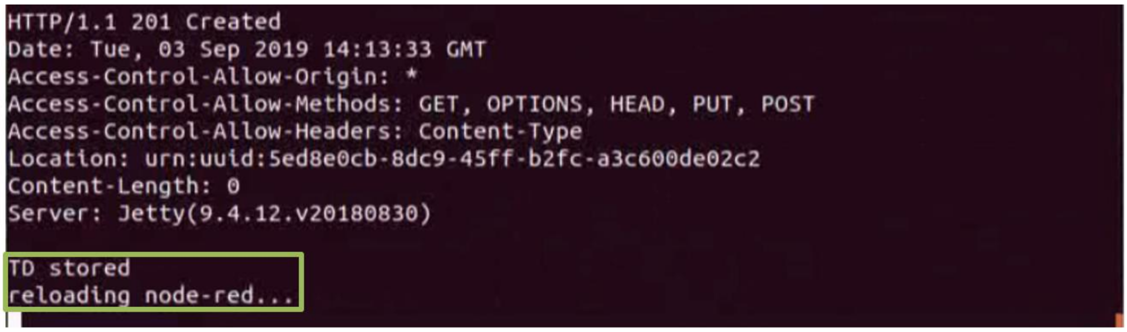


FIGURE 53: CAMERA TESTED IN THE NODE-RED DASHBOARD

The bootstrapping process of a new greenfield device, e.g., a camera, with our approach takes no more than 3 minutes. This includes, the network scanning, discovery, the procedure for creating a programmable interface with the new device, and finally the procedure to make the device discoverable in Local and Global Thing Directory. This process is significantly shorter in comparison to procedures done with traditional engineering tools, where it takes an hour or more to complete the same task. We will provide evaluation results in the final version of this deliverable (once we implement the bootstrapping process for brownfield devices too). But we can already now claim that we meet the KPI 6.1 (Reduce manual interventions required for bootstrapping of smart object in each use case domain by at least 80%).

Finally, let us see tests result from Local Thing Directory. FIGURE 54 shows the command line excerpt after registration of a TD (execution of the “td” method from TABLE 5). The directory creates a new resource for the registered TD with 201 OK status.

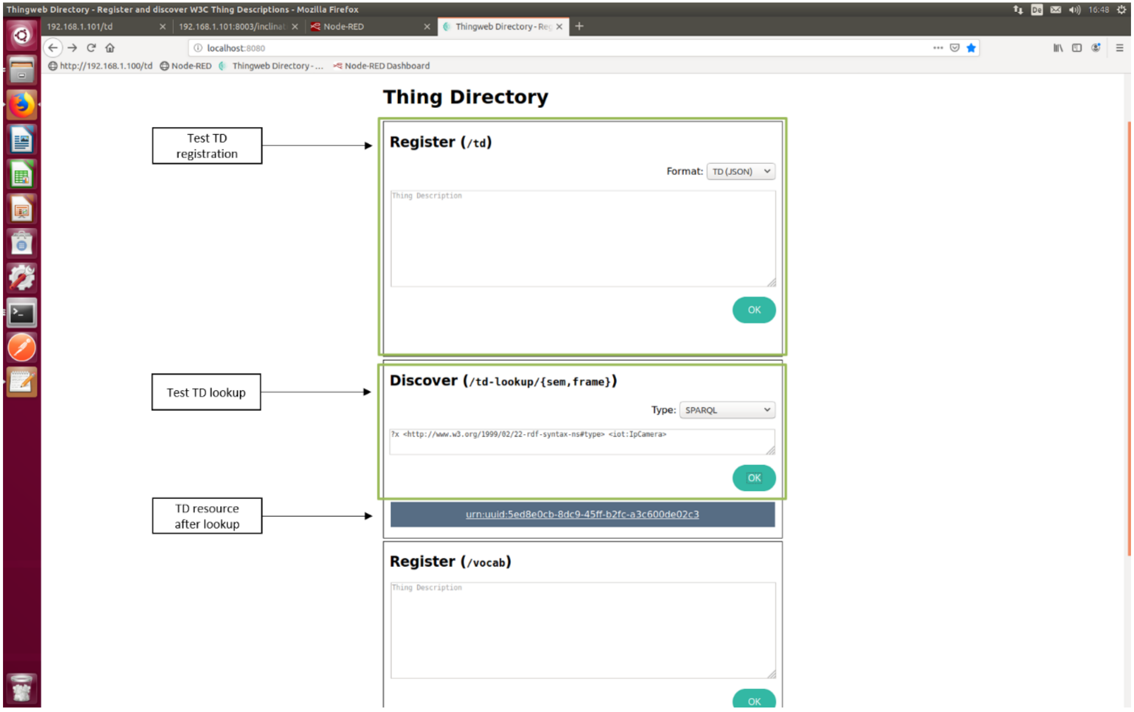


```
HTTP/1.1 201 Created
Date: Tue, 03 Sep 2019 14:13:33 GMT
Access-Control-Allow-Origin: *
Access-Control-Allow-Methods: GET, OPTIONS, HEAD, PUT, POST
Access-Control-Allow-Headers: Content-Type
Location: urn:uuid:5ed8e0cb-8dc9-45ff-b2fc-a3c600de02c2
Content-Length: 0
Server: Jetty(9.4.12.v20180830)

TD stored
reloading node-red...
```

FIGURE 54: TD REGISTRATION

FIGURE 55 shows a user interface of Local Thing Directory. First, it shows an interface for manual registration of a TD (a user may copy/paste a TD and test the “td” method. Second, it shows an interface for semantic lookup, see the method “td-lookup/sem”. FIGURE 55 also shows an example query for discovering all TDs that are annotated with Capability Camera[16]. Finally, FIGURE 55 shows a TD resource, which was retrieved as a query result from Local Thing Directory. If a user clicks on the resource, then it will be displayed in a Web browser (as shown in FIGURE 56). We have created Thing Descriptions for all required field devices in SEMIoTICS. They are semantically annotated with iotschema.org. Thus, we completed KPI 2.1 (Delivery of semantic descriptions for all the 6 types of smart objects which are necessary for the usage scenarios).

[16] http://iotschema.org/Camera



Thing Directory

Register (/td)

Format: TD (JSON)

Thing Description

OK

Discover (/td-lookup/{sem,frame})

Type: SPARQL

?x <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <iot:IpCamera>

OK

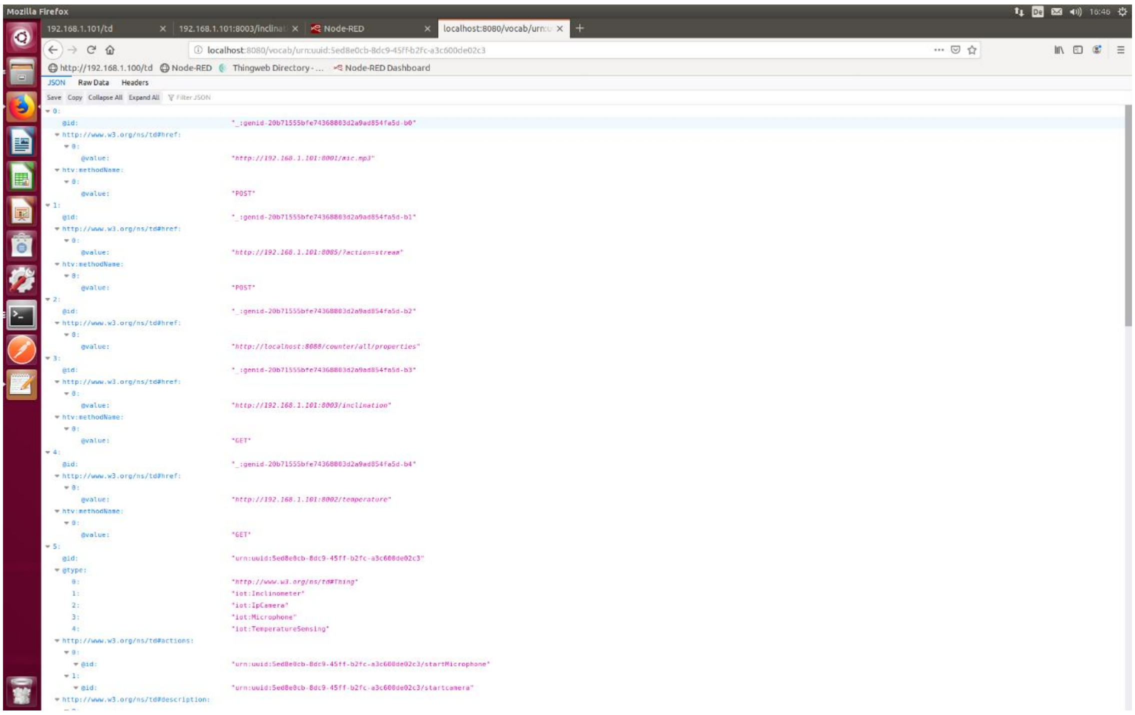
urn:uuid:5ed8e0cb-8dc9-45ff-b2fc-a3c600de02c3

Register (/vocab)

Thing Description

OK

FIGURE 55: TD LOOKUP



```
▾ 0:
    @id:                                   "_:genid-20b71555bfe74368803d2a9ad854fa5d-b0"
  ▾ http://www.w3.org/ns/td#href:
    ▾ 0:
        @value:                            "http://192.168.1.101:8001/mic.mp3"
  ▾ htv:methodName:
    ▾ 0:
        @value:                            "POST"
▾ 1:
    @id:                                   "_:genid-20b71555bfe74368803d2a9ad854fa5d-b1"
  ▾ http://www.w3.org/ns/td#href:
    ▾ 0:
        @value:                            "http://192.168.1.101:8085/?action=stream"
  ▾ htv:methodName:
    ▾ 0:
        @value:                            "POST"
▾ 2:
    @id:                                   "_:genid-20b71555bfe74368803d2a9ad854fa5d-b2"
  ▾ http://www.w3.org/ns/td#href:
    ▾ 0:
        @value:                            "http://localhost:8088/counter/all/properties"
▾ 3:
    @id:                                   "_:genid-20b71555bfe74368803d2a9ad854fa5d-b3"
  ▾ http://www.w3.org/ns/td#href:
    ▾ 0:
        @value:                            "http://192.168.1.101:8003/inclination"
  ▾ htv:methodName:
    ▾ 0:
        @value:                            "GET"
▾ 4:
    @id:                                   "_:genid-20b71555bfe74368803d2a9ad854fa5d-b4"
  ▾ http://www.w3.org/ns/td#href:
    ▾ 0:
        @value:                            "http://192.168.1.101:8002/temperature"
  ▾ htv:methodName:
    ▾ 0:
        @value:                            "GET"
▾ 5:
    @id:                                   "urn:uuid:5ed8e0cb-8dc9-45ff-b2fc-a3c600de02c3"
  ▾ @type:
      0:                                   "http://www.w3.org/ns/td#Thing"
      1:                                   "iot:Inclinometer"
      2:                                   "iot:IpCamera"
      3:                                   "iot:Microphone"
      4:                                   "iot:TemperatureSensing"
  ▾ http://www.w3.org/ns/td#actions:
    ▾ 0:
      ▾ @id:                               "urn:uuid:5ed8e0cb-8dc9-45ff-b2fc-a3c600de02c3/startMicrophone"
    ▾ 1:
      ▾ @id:                               "urn:uuid:5ed8e0cb-8dc9-45ff-b2fc-a3c600de02c3/startcamera"
  ▾ http://www.w3.org/ns/td#description:
```

FIGURE 56: TD RETRIEVED AFTER LOOKUP

The presented work constitutes the contribution towards fulfilling the project’s requirements regarding SEMIOTIC’s objective 2 (development of semantic interoperability mechanisms for smart objects, networks and IoT platforms). The work also contributes to objective 6 (development of a reference prototype of the SEMIoTICS open architecture, demonstrated and evaluated in all use cases, and delivery of the respective open API.). But this work will be continued in D3.9 where the brownfield (domain-specific) integration will be addressed. Additionally, the relevant KPI 2.1 (Delivery of semantic descriptions for all the 6 types of smart objects which are necessary for the usage scenarios) and KPI 6.1 (Reduce manual interventions required for bootstrapping of smart object in each use case domain by at least 80%) are examined. The final validation of objectives and corresponding KPIs will be presented in D3.9 (Bootstrapping and interfacing SEMIoTICS field level devices (final)) and D3.11 (the final version of this deliverable).

3.3.4 RELATION TO NETWORKING REQUIREMENTS

With the above shown functionality of IoT Gateway, we demonstrate the implementation of requirements (mostly revolving around the bootstrapping and interfacing of SEMIoTICS field level devices for greenfield devices):
R.FD.5, R.FD.6, R.FD.7, R.FD.8, R.FD.12, R.UC1.8, R.UC1.10, R.UC1.12

With the presented implementation we also contribute to tasks that are use case-specific:
R.GP.1, R.UC1.1, R.UC1.11, R.UC2.5, R.UC2.6, R.UC3.2, R.UC3.9, R.UC3.12, R.UC3.13, R.UC3.14, R.UC3.15, R.UC3.16, R.UC3.17.

Finally, requirements that are yet to be addressed in D3.9 (mostly revolving around the bootstrapping and interfacing of SEMIoTICS field level devices for brownfield devices) are:
R.FD.13, R.UC1.9, R.UC1.13.

3.3.5 SEMANTIC INTEROPERABILITY

The interoperability between the SEMIoTICS framework and other IoT platforms, such as FIWARE, MindSphere and CIoE IoT, works in two directions. The **first direction** is originating from other IoT platforms, moving towards the SEMIoTICS framework. In that way, SEMIoTICS is able to use the exposed interfaces of the said IoT platforms, in order to take advantage of IoT devices whose descriptions are available in repositories outside SEMIoTICS framework.

On the other hand, the **second direction** is originating from the SEMIoTICS framework, moving towards other IoT platforms. In a similar way, the said platforms utilize the SEMIoTICS' exposed interfaces of selected components in order to employ IoT smart objects and services.

One of the selected components that has exposed interfaces, is the Thing Directory, which is a global version of the Local Thing Directory described in section 3.3.1.2, above. The said IoT smart objects are called Things and their description resides in the Thing Directory. The interface of Thing Directory, can be used to retrieve the already stored semantic description of Things. The Thing Description of the corresponding Thing that is returned, complies with the iotschema and can be used for consumption from other IoT platforms.

Another component that has exposed interfaces, is the Pattern Orchestrator which along with the Pattern Engines, offers the verification of SPDI/QoS properties as a service to be used from the other IoT platforms. In that way an external IoT platform may utilize the said service, in order to verify the SPDI/QoS properties in an existing workflow that is comprised of IoT smart objects (IoT service workflow). The IoT service workflow in question is described in a dedicated language that the Pattern Orchestrator understands and is given as input. Pattern Orchestrator exchanges information with the Pattern Engines in order to check if a specific property holds throughout the whole workflow and corresponds accordingly.

Finally, SDN controller, another component of SEMIoTICS, is also exposed in the scope of offering a service which provides means of managing available OpenFlow devices in the other IoT platforms.

3.3.5.1 COMPONENT ARCHITECTURE

Regarding the **first direction**, the key components of the SEMIoTICS architecture related to interoperability with external IoT platforms (see FIGURE 57) that are involved in this process are:

- Recipe Cooker which is responsible for cooking (creating) recipes reflecting user requirements,
- Pattern Orchestrator which is in charge of the automated configuration, coordination, and management of different patterns (in this case Interoperability patterns) and their deployment,
- Pattern Engine (Backend) which allows the insertion, modification, execution, and retraction of patterns through the Pattern Orchestrator,
- Backend Semantic Validator (BSV) which resolves semantic interoperability issues and
- Thing Directory (Backend) which is the repository of knowledge containing the necessary Thing models

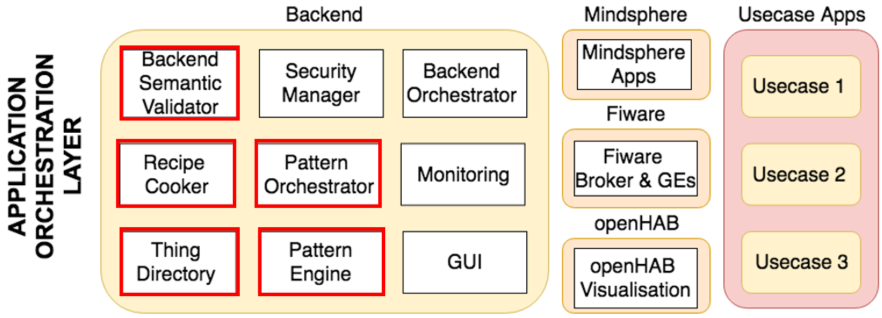


FIGURE 57 KEY COMPONENTS OF THE SEMIOTICS ARCHITECTURE RELATED TO INTEROPERABILITY WITH EXTERNAL IOT PLATFORMS

The below motivating example with FIWARE is used for the description and analysis of the development of the proposed approach. The main concept begins from Recipe Cooker (Backend). During runtime, a recipe/flow can be designed by the user in Recipe Cooker; this flow represents an interaction between two Things i.e. FIWARE Sensor, SEMIoTICS Thermostat. The aim is to check the semantic interoperability between the specific nodes to ensure the aforementioned communication. For that reason, Recipe Cooker sends the “cooked” recipe to the Pattern Orchestrator in order to transform it into architectural patterns (in this case interoperability patterns). The Pattern Engine (Backend) receives the interoperability requirement from Pattern Orchestrator, as it is responsible to enable the capability to insert, modify, execute and retract patterns. The next step of Pattern Engine (Backend) is to examine the semantic interoperability for any links in the recipe/flow (in this example there is only one link/wire, the connection between FIWARE Sensor and SEMIoTICS Thermostat). Thus, for any link, Pattern Engine (Backend) triggers the BSV.

The above approach of the semantic interoperability mechanisms between SEMIoTICS and external IoT platforms is highlighted by sequence diagram, in FIGURE 58.

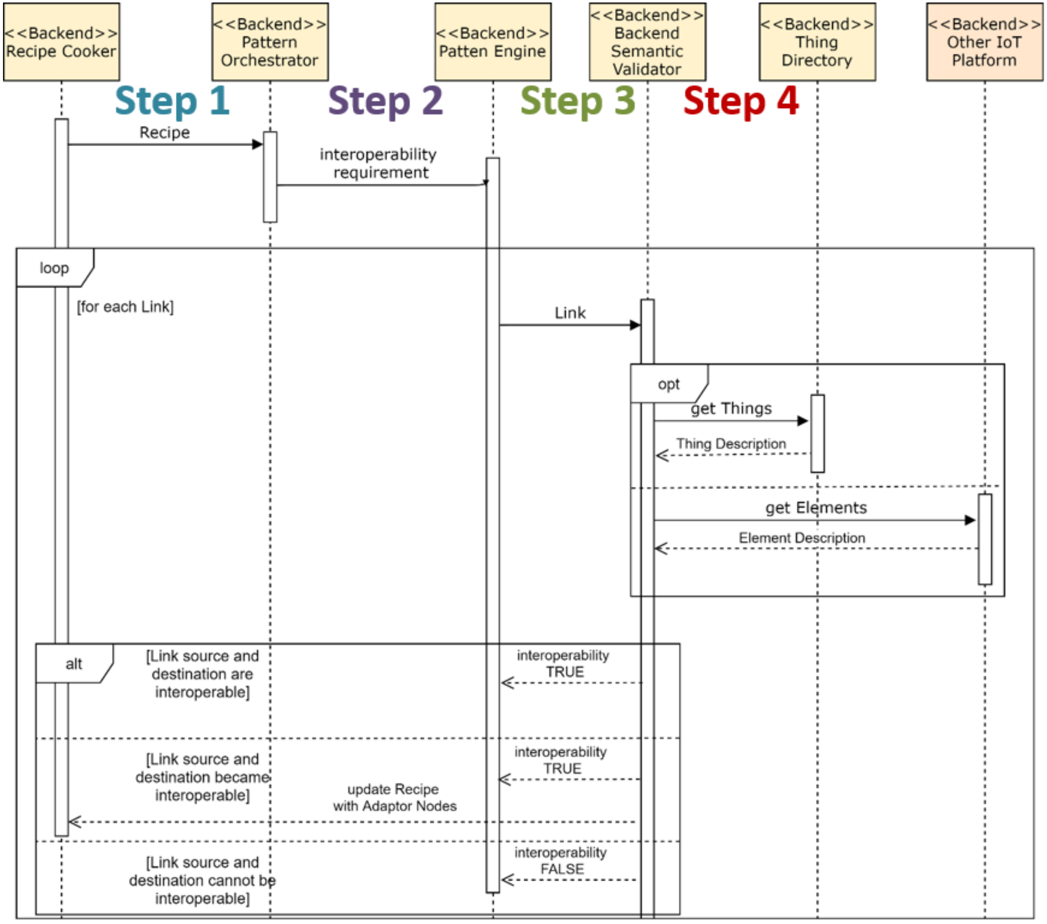
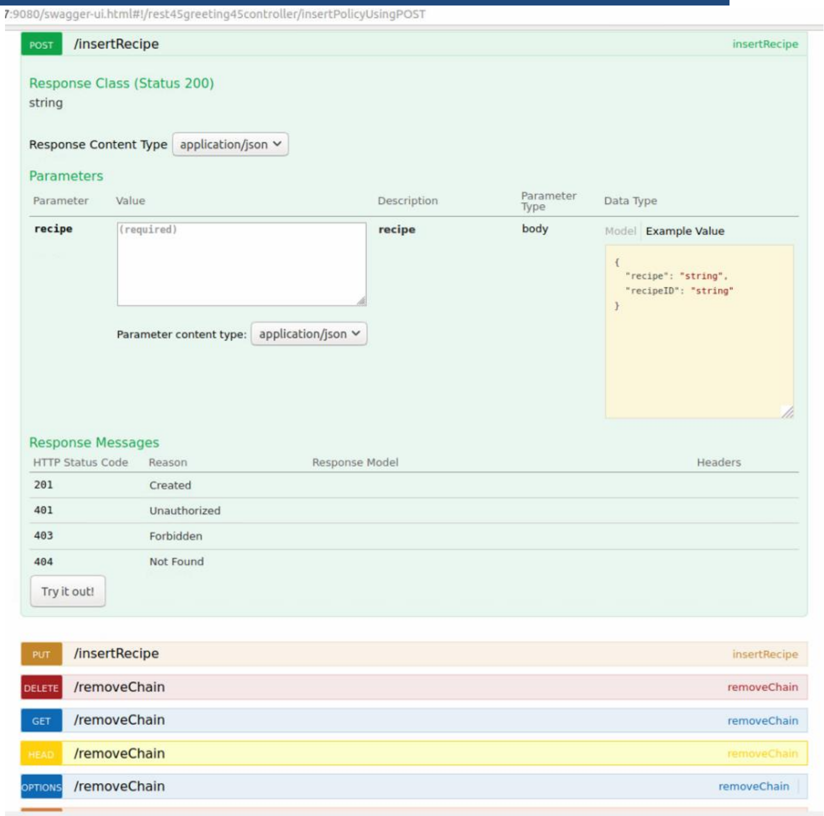

<<Backend>> Recipe Cooker
<<Backend>> Pattern Orchestrator
<<Backend>> Patten Engine
<<Backend>> Backend Semantic Validator
<<Backend>> Thing Directory
<<Backend>> Other IoT Platform
Step 1
Step 2
Step 3
Step 4
Recipe
interoperability requirement
loop
[for each Link]
Link
opt
get Things
Thing Description
get Elements
Element Description
alt
[Link source and destination are interoperable]
interoperability TRUE
[Link source and destination became interoperable]
interoperability TRUE
update Recipe with Adaptor Nodes
[Link source and destination cannot be interoperable]
interoperability FALSE

FIGURE 58 SEQUENCE DIAGRAM OF INTEROPERABILITY WITH EXTERNAL IOT PLATFORMS

The following is a detailed description of each component and API for the interaction between them based on the above sequence diagram.

- **Recipe Cooker (Backend) – Pattern Orchestrator (Backend)** Step 1:

The Recipe Cooker which is responsible for cooking (creating) recipes reflecting user requirements, send the recipe in Pattern Orchestrator using a POST method request. Particularly, this POST parameters includes the recipe/flow (JSON format), as body and the header is application/json. The structure is presented in FIGURE 59.



7:9080/swagger-ui.html#!/rest45greeting45controller/insertPolicyUsingPOST

POST /insertRecipe insertRecipe

Response Class (Status 200)
string

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
recipe	(required)	recipe	body	Model Example Value

```
{
  "recipe": "string",
  "recipeID": "string"
}
```

Parameter content type: application/json

Response Messages

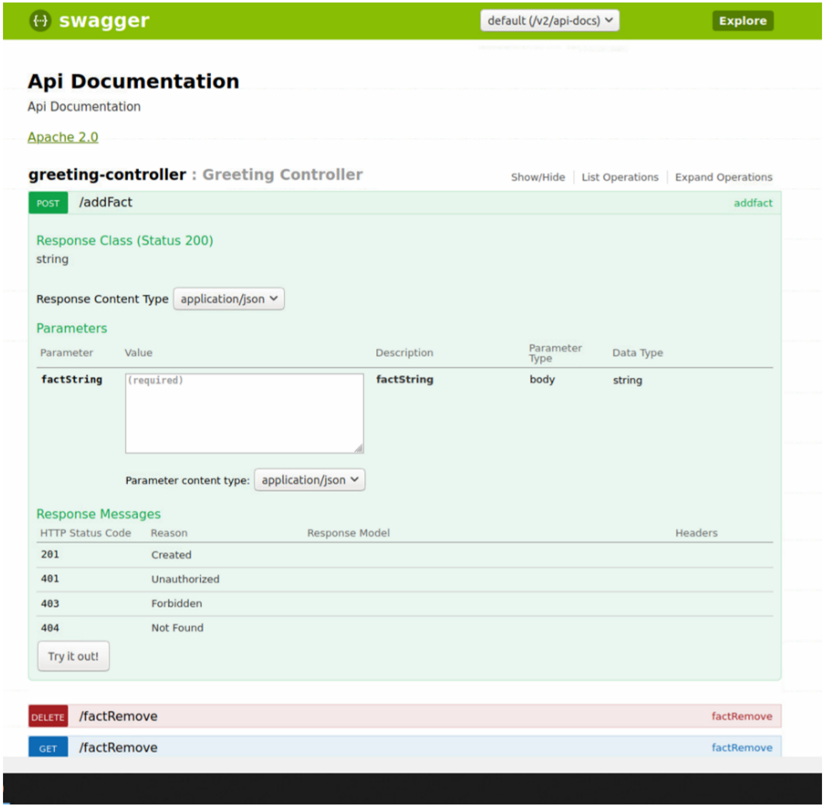
HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

PUT	/insertRecipe	insertRecipe
DELETE	/removeChain	removeChain
GET	/removeChain	removeChain
HEAD	/removeChain	removeChain
OPTIONS	/removeChain	removeChain

FIGURE 59 API INTERACTION BETWEEN RECIPE COOKER (BACKEND) – PATTERN ORCHESTRATOR (BACKEND)

- **Pattern Orchestrator (Backend) – Pattern Engine (Backend)** **Step 2**:

The next step is the request from Pattern Orchestrator to Pattern Engine (Backend) in order to send the Interoperability requirement using a POST method with the following parameters (see FIGURE 60).



Api Documentation

Api Documentation

Apache 2.0

greeting-controller : Greeting Controller

POST /addFact addfact

Response Class (Status 200)
string

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
factString	(required)	factString	body	string

Parameter content type: application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

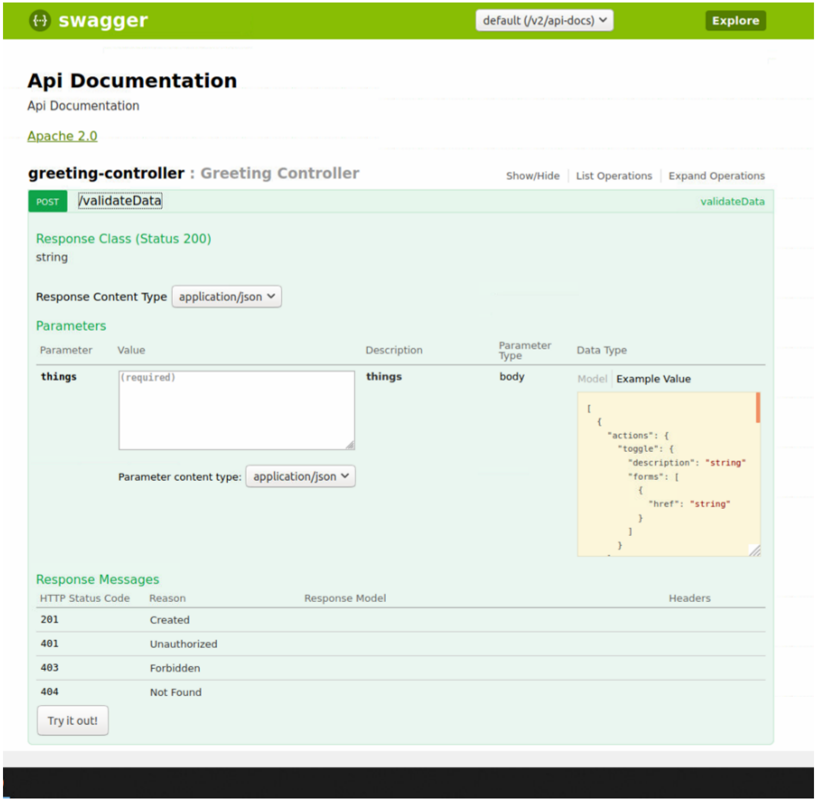
DELETE /factRemove factRemove

GET /factRemove factRemove

FIGURE 60 API INTERACTION BETWEEN PATTERN ORCHESTRATOR (BACKEND) – PATTERN ENGINE (BACKEND)

- **Pattern Engine (Backend) Backend Semantic Validator (Backend)** Step 3:

The BSV component receives a request from the Pattern Engine (Backend) to check the semantic interoperability between two Things (link) in JSON-LD/JSON format. The JSON-LD/JSON Parser is implemented as part of the BSV, to analyze the received input and extract the meaningful information from these set of data. The communication between the said components is achieved using the POST method (FIGURE 61). This step is not yet fully implemented. Currently the BSV is tested using Postman. The requests made by Postman will later be replaced by requests made by the Pattern Engine.



Api Documentation

Api Documentation

Apache 2.0

greeting-controller : Greeting Controller

POST /validateData validateData

Response Class (Status 200)
string

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
things	(required) Parameter content type: application/json	things	body	Model Example Value

```
[
  {
    "actions": {
      "toggle": {
        "description": "string"
        "forms": [
          {
            "href": "string"
          }
        ]
      }
```

Response Messages

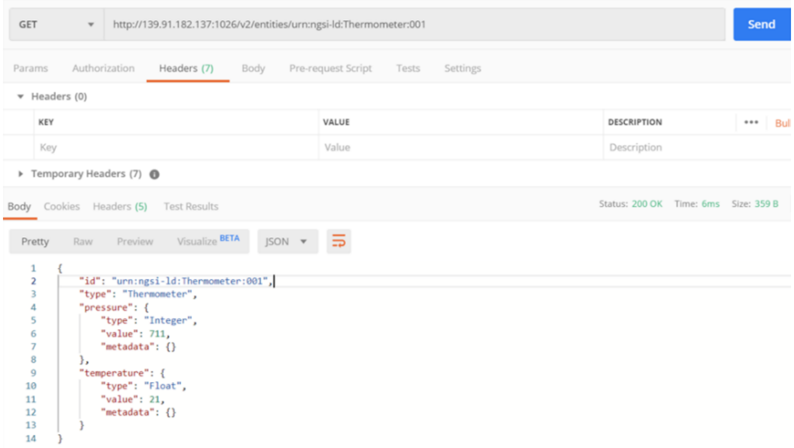
HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

FIGURE 61 API INTERACTION BETWEEN PATTERN ENGINE (BACKEND) – BACKEND SEMANTIC VALIDATOR (BACKEND)

- **Backend Semantic Validator (Backend) – Thing Directory or Other IoT Platform** Step4:

After that, the BSV begins the procedure to tackle the semantic interoperability issues between these two Things from the said recipe/flow. In order to give this answer, the semantic description for any Thing is required (for FIWARE Sensor and SEMIoTICS Thermostat). For that reason, it sends two requests:

1. SPARQL query to Thing Directory in order to receive the Thing Description of SEMIoTICS Thermostat (see FIGURE 43 in Section 3.3) and
2. GET method to the Orion Context Broker FIWARE platform to receive the context data Description of FIWARE Sensor. The query parameters are highlighted in FIGURE 62. The response consists of JSON with the FIWARE Sensor attributes (type, metadata elements).



GET http://139.91.182.137:1026/v2/entities/urn:ngsi-ld:Thermometer:001

KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 6ms Size: 359 B

```
{
    "id": "urn:ngsi-ld:Thermometer:001",
    "type": "Thermometer",
    "pressure": {
        "type": "Integer",
        "value": 711,
        "metadata": {}
    },
    "temperature": {
        "type": "Float",
        "value": 21,
        "metadata": {}
    }
}
```

FIGURE 62 POSTMAN GET REQUEST TO THE ORION CONTEXT BROKER FIWARE PLATFORM

Based on this information, the BSV could decide for the interoperability between the Things and harmonize the semantic model capabilities with the registration of extra Adaptor Nodes in the recipe. Particularly, there are three possible results. Firstly, the link source and destination are interoperable, so the BSV updates the Pattern Engine (Backend) with the TRUE response. Secondly, the link source and destination are not interoperable and the BSV can add Adaptor Nodes in order to guarantee the interoperability. In this case, BSV not only sends the TRUE response in Pattern Engine (Backend), but also updates the recipe in Recipe Cooker using the corresponding Adaptor Nodes. Lastly, the link source and destination are not interoperable and BSV does not have the required information to develop the Adaptor Nodes; hence, the Pattern Engine (Backend) receives the FALSE response by the BSV.

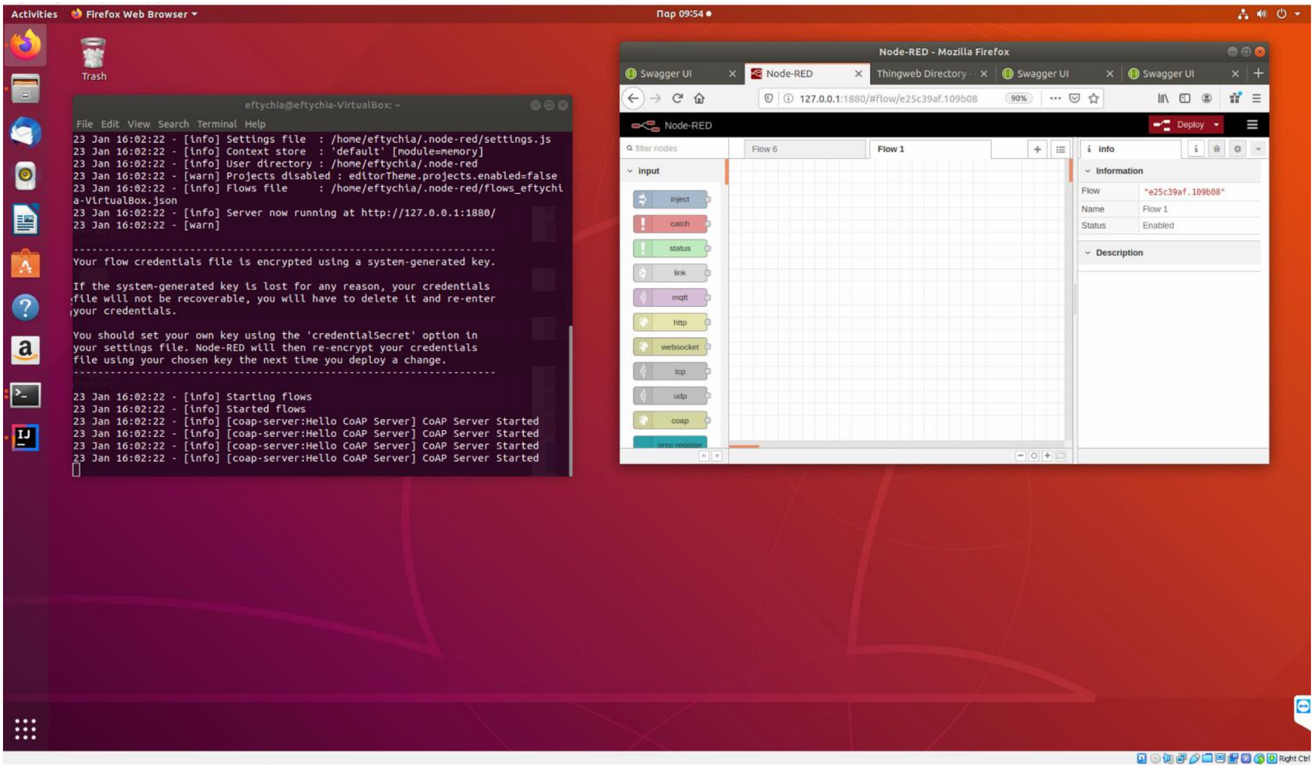
3.3.5.2 TESTING METHODOLOGY

In order to test the functionalities of all the said components of SEMIoTICS, as they are described in previous subsection, we use the Proxmox Virtual environment to create Virtual Machines (VMs). The created VMs have some hardware and software requirements, which are shown in TABLE 6 below.

TABLE 6 VM REQUIREMENTS

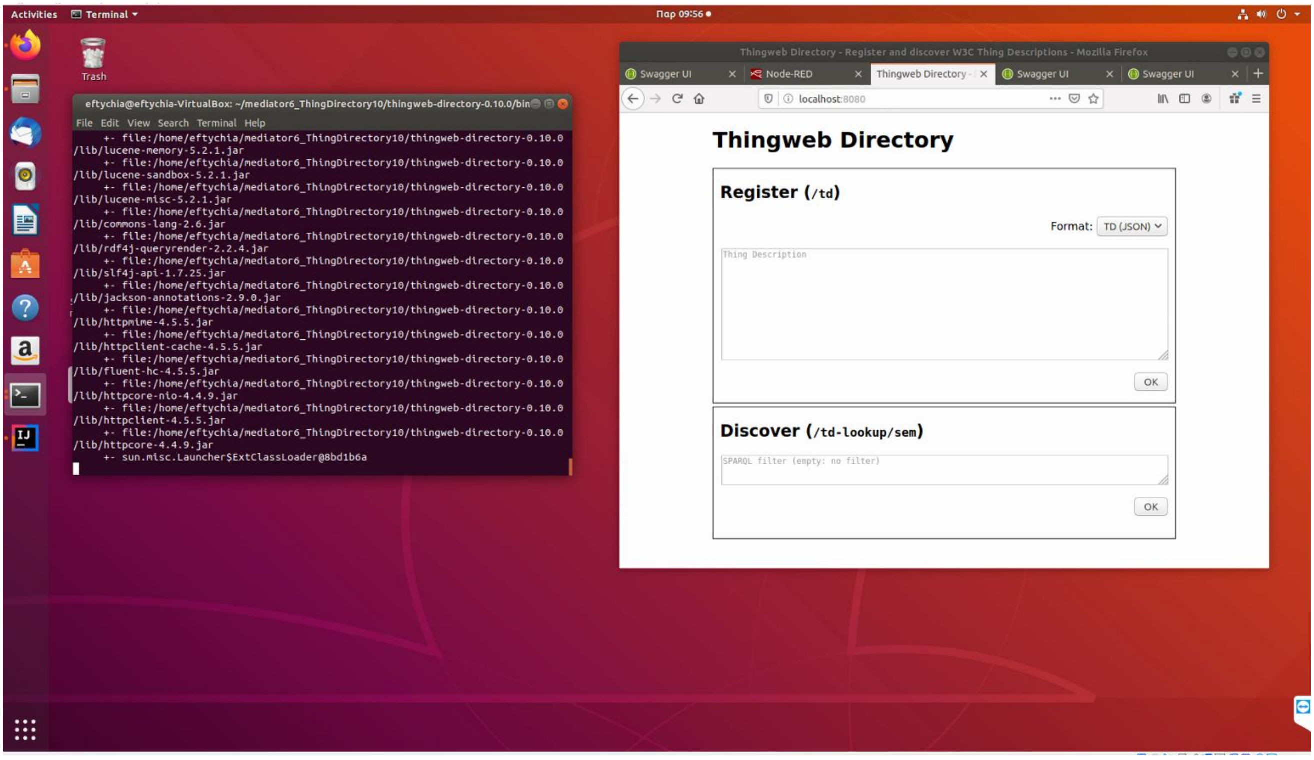
Component	Software	CPU	Memory	Disk
Recipe Cooker	Ubuntu 16.04 LTS	2 cores	4 GB	5 GB
Pattern Orchestrator	Ubuntu 16.04 LTS	2 cores	4 GB	5 GB
Pattern Engine (Backend)	Ubuntu 16.04 LTS	2 cores	4 GB	10 GB
BSV	Ubuntu 16.04 LTS	2 cores	4 GB	10 GB
Thing Directory	Ubuntu 16.04 LTS	2 cores	4 GB	5 GB
Orion Context Broker FIWARE	Ubuntu 16.04 LTS	2 cores	4 GB	5 GB

Thus, every component that is included in the interaction between SEMIoTICS and other IoT platforms, could be run on a modest Ubuntu VM and exchange data using the APIs that have already described in the previous section. The overall deployment is scalable and sufficient for real-time operation. A preliminary version of the proposed setting is implemented, thus FIGURE 63, FIGURE 64, FIGURE 65 and FIGURE 66 present screenshots of running VMs.



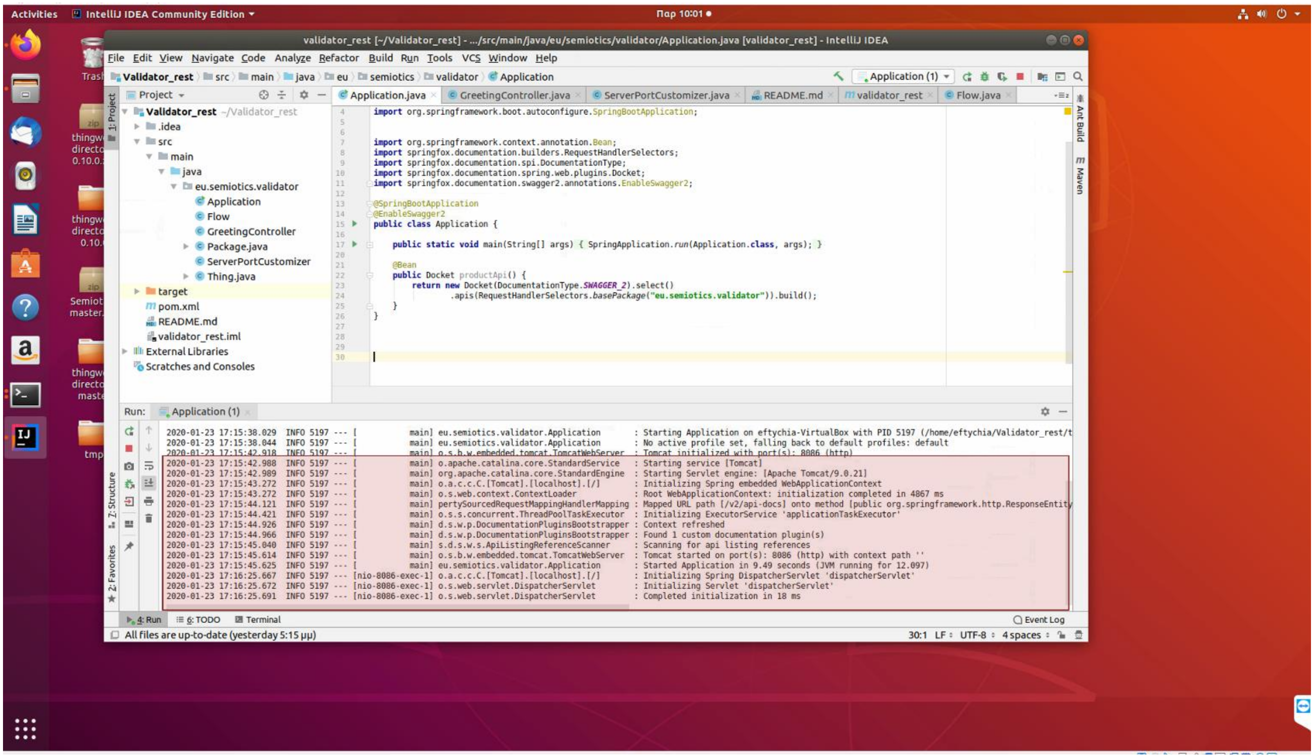
```
23 Jan 16:02:22 - [info] Settings file  : /home/eftychia/.node-red/settings.js
23 Jan 16:02:22 - [info] Context store  : 'default' [module=memory]
23 Jan 16:02:22 - [info] User directory : /home/eftychia/.node-red
23 Jan 16:02:22 - [warn] Projects disabled : editorTheme.projects.enabled=false
23 Jan 16:02:22 - [info] Flows file     : /home/eftychia/.node-red/flows_eftychi
a-VirtualBox.json
23 Jan 16:02:22 - [info] Server now running at http://127.0.0.1:1880/
23 Jan 16:02:22 - [warn]

---------------------------------------------------------------------
Your flow credentials file is encrypted using a system-generated key.

If the system-generated key is lost for any reason, your credentials
file will not be recoverable, you will have to delete it and re-enter
your credentials.

You should set your own key using the 'credentialSecret' option in
your settings file. Node-RED will then re-encrypt your credentials
file using your chosen key the next time you deploy a change.
---------------------------------------------------------------------

23 Jan 16:02:22 - [info] Starting flows
23 Jan 16:02:22 - [info] Started flows
23 Jan 16:02:22 - [info] [coap-server:Hello CoAP Server] CoAP Server Started
23 Jan 16:02:22 - [info] [coap-server:Hello CoAP Server] CoAP Server Started
23 Jan 16:02:22 - [info] [coap-server:Hello CoAP Server] CoAP Server Started
23 Jan 16:02:22 - [info] [coap-server:Hello CoAP Server] CoAP Server Started
```

FIGURE 63 VM RECIPE COOKER


Activities
Terminal
Παρ 09:56
Trash
eftychia@eftychia-VirtualBox: ~/mediator6_ThingDirectory10/thingweb-directory-0.10.0/bin
File Edit View Search Terminal Help
Thingweb Directory - Register and discover W3C Thing Descriptions - Mozilla Firefox
Swagger UI
Node-RED
Thingweb Directory -
Swagger UI
Swagger UI
localhost:8080
Thingweb Directory
Register (/td)
Format:
TD (JSON)
Thing Description
OK
Discover (/td-lookup/sem)
SPARQL filter (empty: no filter)
OK

```
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/lucene-memory-5.2.1.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/lucene-sandbox-5.2.1.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/lucene-misc-5.2.1.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/commons-lang-2.6.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/rdf4j-queryrender-2.2.4.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/slf4j-api-1.7.25.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/jackson-annotations-2.9.0.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/httpmime-4.5.5.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/httpclient-cache-4.5.5.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/fluent-hc-4.5.5.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/httpcore-nio-4.4.9.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/httpclient-4.5.5.jar
     +- file:/home/eftychia/mediator6_ThingDirectory10/thingweb-directory-0.10.0
/lib/httpcore-4.4.9.jar
     +- sun.misc.Launcher$ExtClassLoader@8bd1b6a
```

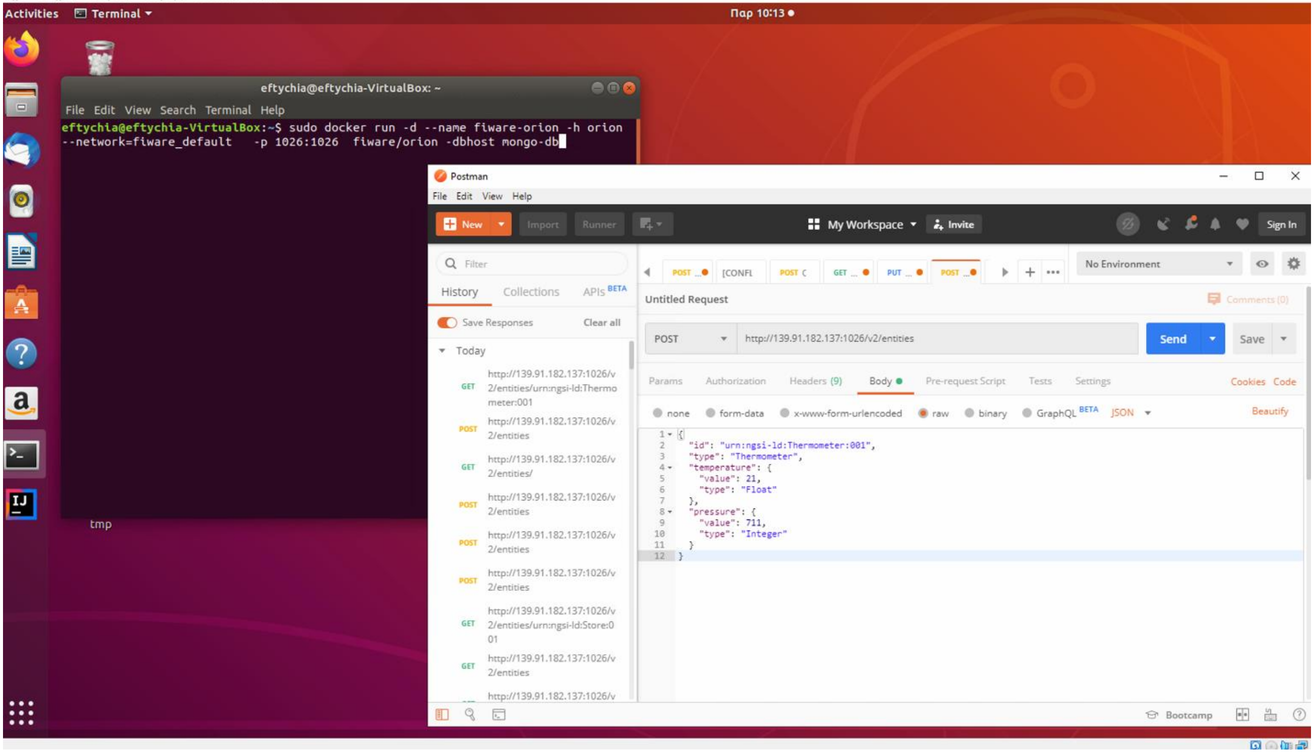
FIGURE 64 VM THING DIRECTORY



```java
import org.springframework.boot.autoconfigure.SpringBootApplication;


import org.springframework.context.annotation.Bean;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@SpringBootApplication
@EnableSwagger2
public class Application {

    public static void main(String[] args) { SpringApplication.run(Application.class, args); }

    @Bean
    public Docket productApi() {
        return new Docket(DocumentationType.SWAGGER_2).select()
                .apis(RequestHandlerSelectors.basePackage("eu.semiotics.validator")).build();
    }
}
```

```
2020-01-23 17:15:38.029  INFO 5197 --- [           main] eu.semiotics.validator.Application       : Starting Application on eftychia-VirtualBox with PID 5197 (/home/eftychia/Validator_rest/t
2020-01-23 17:15:38.044  INFO 5197 --- [           main] eu.semiotics.validator.Application       : No active profile set, falling back to default profiles: default
2020-01-23 17:15:42.918  INFO 5197 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8086 (http)
2020-01-23 17:15:42.988  INFO 5197 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2020-01-23 17:15:42.989  INFO 5197 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.21]
2020-01-23 17:15:43.272  INFO 5197 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2020-01-23 17:15:43.272  INFO 5197 --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 4867 ms
2020-01-23 17:15:44.121  INFO 5197 --- [           main] pertySourcedRequestMappingHandlerMapping : Mapped URL path [/v2/api-docs] onto method [public org.springframework.http.ResponseEntity
2020-01-23 17:15:44.421  INFO 5197 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2020-01-23 17:15:44.926  INFO 5197 --- [           main] d.s.w.p.DocumentationPluginsBootstrapper : Context refreshed
2020-01-23 17:15:44.966  INFO 5197 --- [           main] d.s.w.p.DocumentationPluginsBootstrapper : Found 1 custom documentation plugin(s)
2020-01-23 17:15:45.040  INFO 5197 --- [           main] s.d.s.w.s.ApiListingReferenceScanner     : Scanning for api listing references
2020-01-23 17:15:45.614  INFO 5197 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8086 (http) with context path ''
2020-01-23 17:15:45.625  INFO 5197 --- [           main] eu.semiotics.validator.Application       : Started Application in 9.49 seconds (JVM running for 12.097)
2020-01-23 17:16:25.667  INFO 5197 --- [nio-8086-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-01-23 17:16:25.672  INFO 5197 --- [nio-8086-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2020-01-23 17:16:25.691  INFO 5197 --- [nio-8086-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 18 ms
```

FIGURE 65 VM BSV



```
eftychia@eftychia-VirtualBox:~$ sudo docker run -d --name fiware-orion -h orion
--network=fiware_default    -p 1026:1026  fiware/orion -dbhost mongo-db
```

POST http://139.91.182.137:1026/v2/entities

```json
{
  "id": "urn:ngsi-ld:Thermometer:001",
  "type": "Thermometer",
  "temperature": {
    "value": 21,
    "type": "Float"
  },
  "pressure": {
    "value": 711,
    "type": "Integer"
  }
}
```

GET http://139.91.182.137:1026/v2/entities/urn:ngsi-ld:Thermometer:001
POST http://139.91.182.137:1026/v2/entities
GET http://139.91.182.137:1026/v2/entities/
POST http://139.91.182.137:1026/v2/entities
POST http://139.91.182.137:1026/v2/entities
POST http://139.91.182.137:1026/v2/entities
GET http://139.91.182.137:1026/v2/entities/urn:ngsi-ld:Store:001
GET http://139.91.182.137:1026/v2/entities
http://139.91.182.137:1026/v

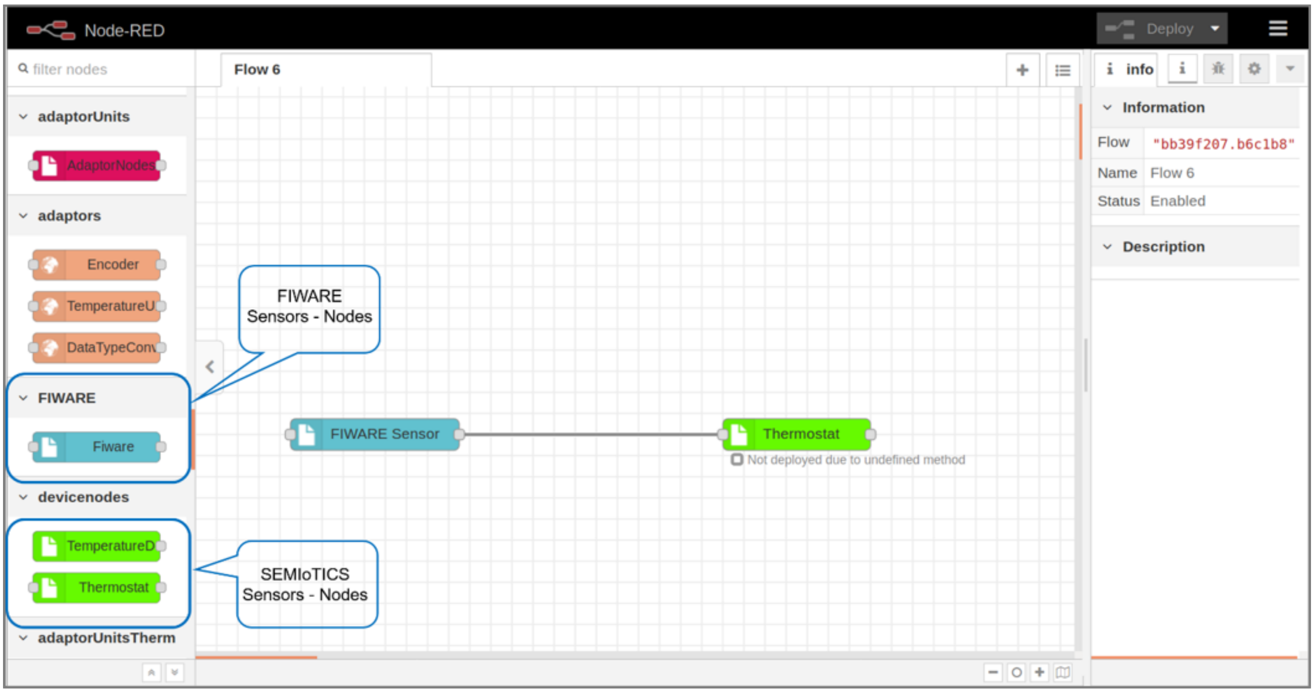
FIGURE 66 VM ORION CONTEXT BROKER FIWARE

3.3.5.3 PERFORMANCE TEST AND KPI VALIDATION

In this section, we present results of a typical testing workflow, based on the methodology that was described in the previous subsections. More specifically, the user designs a recipe/flow in Recipe Cooker; this flow represents an interaction between two Things i.e. FIWARE Sensor, SEMIoTICS Thermostat. The overall functionality is to check the semantic interoperability between these specific nodes to ensure the aforementioned communication between them. For that reason,

- Recipe Cooker sends the "cooked" recipe to the Pattern Orchestrator in order to transform it into architectural patterns (in this case interoperability patterns)
- Pattern Engine (Backend) receives the interoperability requirement from Pattern Orchestrator, as it is responsible to enable the capability to insert, modify, execute and retract patterns. This component should examine the semantic interoperability for the link/wire between FIWARE Sensor and SEMIoTICS Thermostat in the recipe/flow; for that reason, it triggers the BSV
- BSV takes the sematic metadata of two Things using the Thing Directory and Orion Context Broker FIWARE. Based on this information, the BSV could decide for the interoperability between the Things and harmonize the semantic model capabilities with the registration of extra Adaptor Nodes in the recipe and send back to Pattern Engine the corresponding response. Except that, if the link/wire source and destination are not interoperable and the BSV can add Adaptor Nodes in order to guarantee the interoperability, BSV should update the initial recipe in Recipe Cooker.

The following figures demonstrate the initial status of the recipe and the final structure taking to advantage the BSV procedure in which tackles the semantic interoperability issues between these two Things.



Not deployed due to undefined method

Flow "bb39f207.b6c1b8"
Name Flow 6
Status Enabled

FIGURE 67 RECIPE INTERACTION EXAMPLE FIWARE – SEMIOTICS BEFORE SEMANTIC VALIDATION

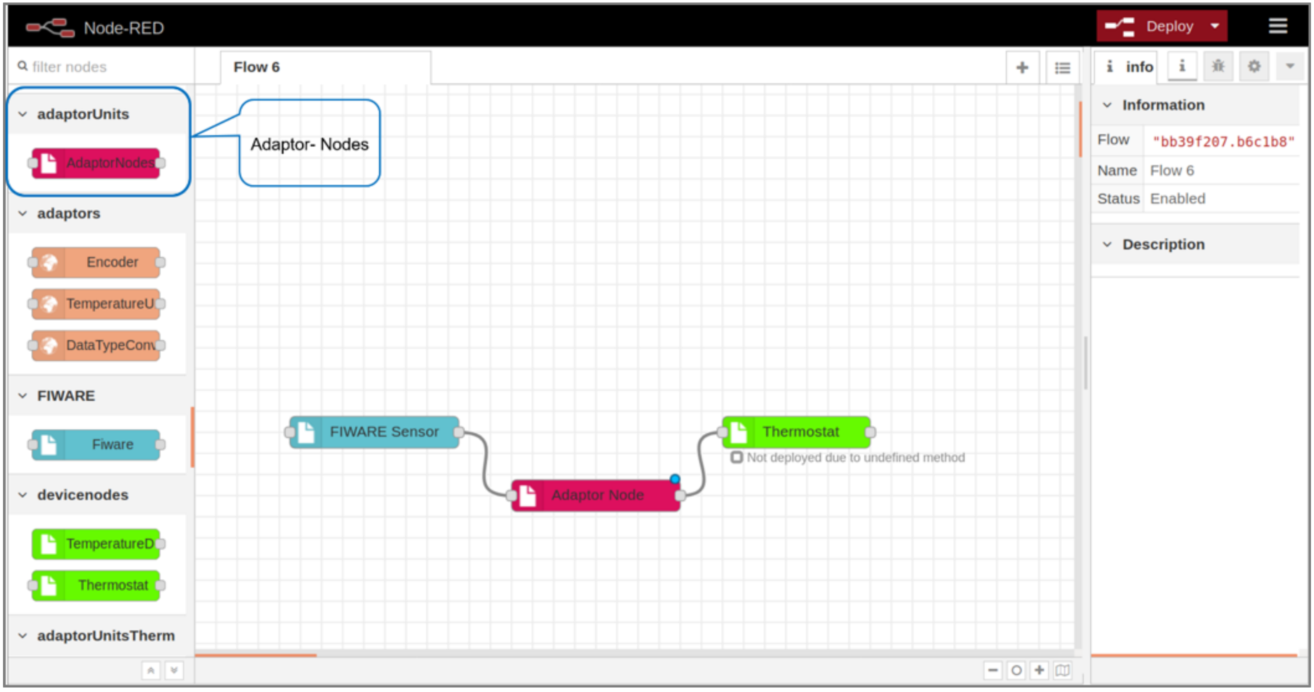


FIGURE 68 RECIPE INTERACTION EXAMPLE FIWARE – SEMIOTICS AFTER SEMANTIC VALIDATION

With the above shown functionality of the interoperability between SEMIoTICS and other IoT platforms, we focus on address some of the related project objectives corresponding the interoperability, such as

- the definition of SEMIoTICS semantic mediator mechanisms, with the purpose of resolving, if possible, conflicts among the semantic models used in the semantic annotations of the patterns,
- the development of data transformation techniques and validation mechanisms to ensure semantic interoperability,
- the definition of the mappings between datatypes used in SEMIoTICS, to ensure that data flow is possible between smart objects that are linked in the composition structure defined by the pattern

The overall procedure constitutes the initial contribution towards fulfilling the project’s requirements regarding SEMIOTIC’s objective 2 (development of semantic interoperability mechanisms for smart objects, networks and IoT platforms). Additionally, the relevant KPI 2.2 (Delivery of data type mapping and ontology alignment and transformation techniques that realize semantic interoperability) and KPI 2.3 (Validated semantic interoperability between the SEMIoTICS framework and 3 IoT platforms, including FIWARE) are examined. The final validation of objectives and corresponding KPIs will be presented in the D4.11 “Semantic interoperability mechanisms for IoT (final)”.

3.4 Backend components

3.4.1 SECURITY AND PRIVACY

3.4.1.1 COMPONENT ARCHITECTURE

The Backend Security Manager was deployed for ensuring security and safety along all other components. The Backend Security Manager helps SEMIoTICS to tackle the security and privacy problems that arise from the multi-tenant scenarios in a variety of levels, i.e., from the networking layer to the application layer. The components of the Security Manager (at the level of the backend and additionally at the network- and field-level) are controlled by the Backend Security Manager component. The components allow SEMIoTICS to achieve the required functionality in order to:

- provide mechanisms to authenticate users and manage their identities,
- provide mechanisms to manage identities of other entities, e.g., sensors,
- support use case applications to enforce access to privacy-sensitive information within the application,
- support use case applications to enforce access to privacy-sensitive information when the data is stored in a cloud server, e.g., by using attribute-based encryption and lightweight encryption algorithms and finally,
- provide mechanisms to configure and manage SEMIoTICS end-to-end secure networking capabilities.

All those requirements are covered and managed by one or more of the different software modules of the Backend Security Manager.

As we have dockerized the complete Backend Security Manager, as described in Deliverable 4.7, it was easily deployed to the Kubernetes cluster.

➔ ENTITY (INCL. USER) AUTHENTICATION

In order to take access control decision, the security manager needs to know which entity is requesting which access. In order to become assured of the entity mechanisms for entity authentication need to be provided by the Security Manager. To handle the authentication requests of users and other entities alike, the Security Manager is additionally an OAuth2 provider. OAuth in Version 2.0 is a standardized (specification and associated RFCs developed by the IETF OAuth WG can be found on https://datatracker.ietf.org/wg/oauth/documents/) framework for user authentication and was published in October 2012. It is considered an industry standard and is state of practice.

The project wide integration was easily manageable, as the other components of SEMIoTICS can use existing client implementations to use SEMIoTICS identity management and authentication services offered by the security manager in the backend. In essence, applications requiring authentication services need to register as an OAuth2 client with the security manager and then can defer users to the SEMIoTICS authentication endpoint. The former approach also helps when users have only access to a browser (or a mobile device), because the OAuth protocol was developed with this in mind. Additionally, applications without explicit user interaction, e.g., batch or cron-jobs, can authenticate towards the security manager by providing the client credentials they have, or by user a username and password tuple for a valid user. The general architecture of the OAuth-related component of the SEMIoTICS Security Manager in the backend is depicted in FIGURE 52.

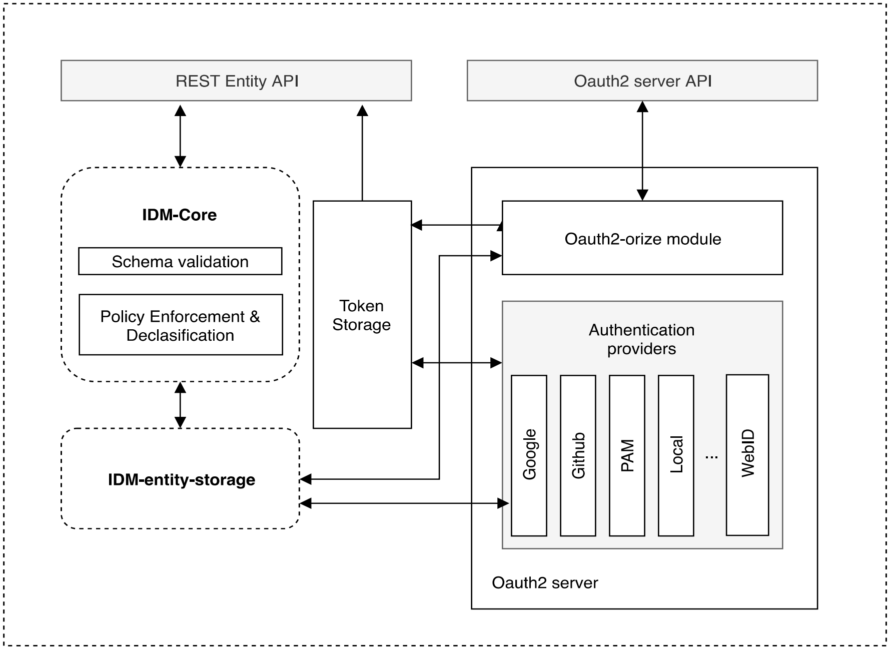


FIGURE 69: SECURITY MANAGER IDM ARCHITECTURE

➔ WORKFLOWS AND INTERACTIONS WITH OTHER SEMIOTICS COMPONENTS

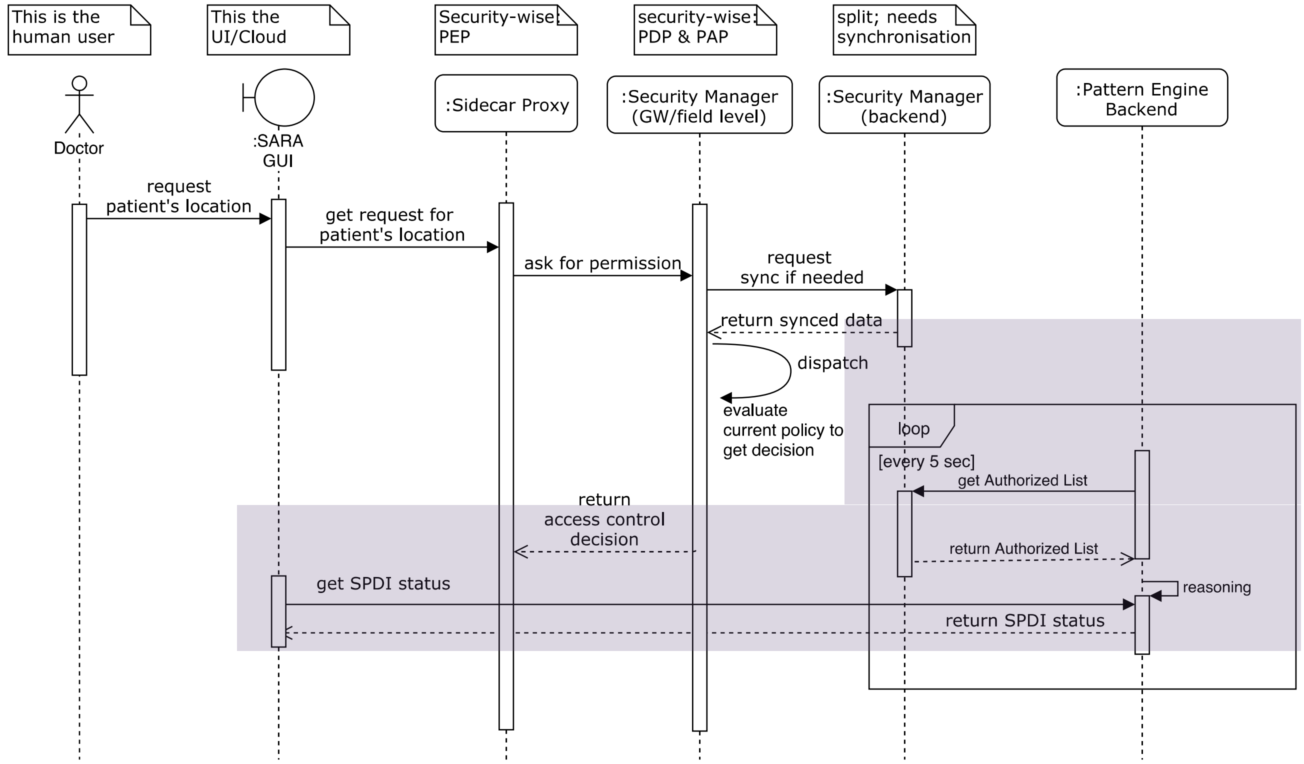


FIGURE 70 WORKFLOW FOR UNAUTHORIZED LOCATION RETRIEVAL

FIGURE 70 depicts in the highlighted area the interaction of the Backend Security Manager with the Pattern engine. The goal is that the Pattern Engine is able to check that the current policy used to make the decisions inside the Backend Security Manager, is conforming to the SPDI Patterns as specified by the application. This is done by the Pattern Engine based on the information about the required SPDI patterns the system should conform to. Pattern Engine periodically makes requests to the Security Manager (like `get Authorized List)` to obtain certain information about the currently enforced policy. Based on the response, it uses rules Drools Engine) specific to the SPDI patterns to reason on the answers received from the Security Manager. The reasoning allows the Pattern Engine to identify if the Policy being enforced in SEMIoTICS is compliant; it is able to reflect this to the outside via an API call that will allow other components to retrieve the `SPDI status`.

What you can see in the non-highlighted area of FIGURE 70 is that the doctor's initial request to a service (not depicted in FIGURE 70) is intercepted by the Sidecar Proxy acting as a policy enforcement point (PEP). This shows that the Sidecar Proxy gets the permission details for the incoming request from the Security Manager. Asking the local Security Manager is only needed if the service would be exposed by the field level and after the synchronization and evaluation of the Security Manager. Based on this information the PEP will then decide what to do with the request or response.

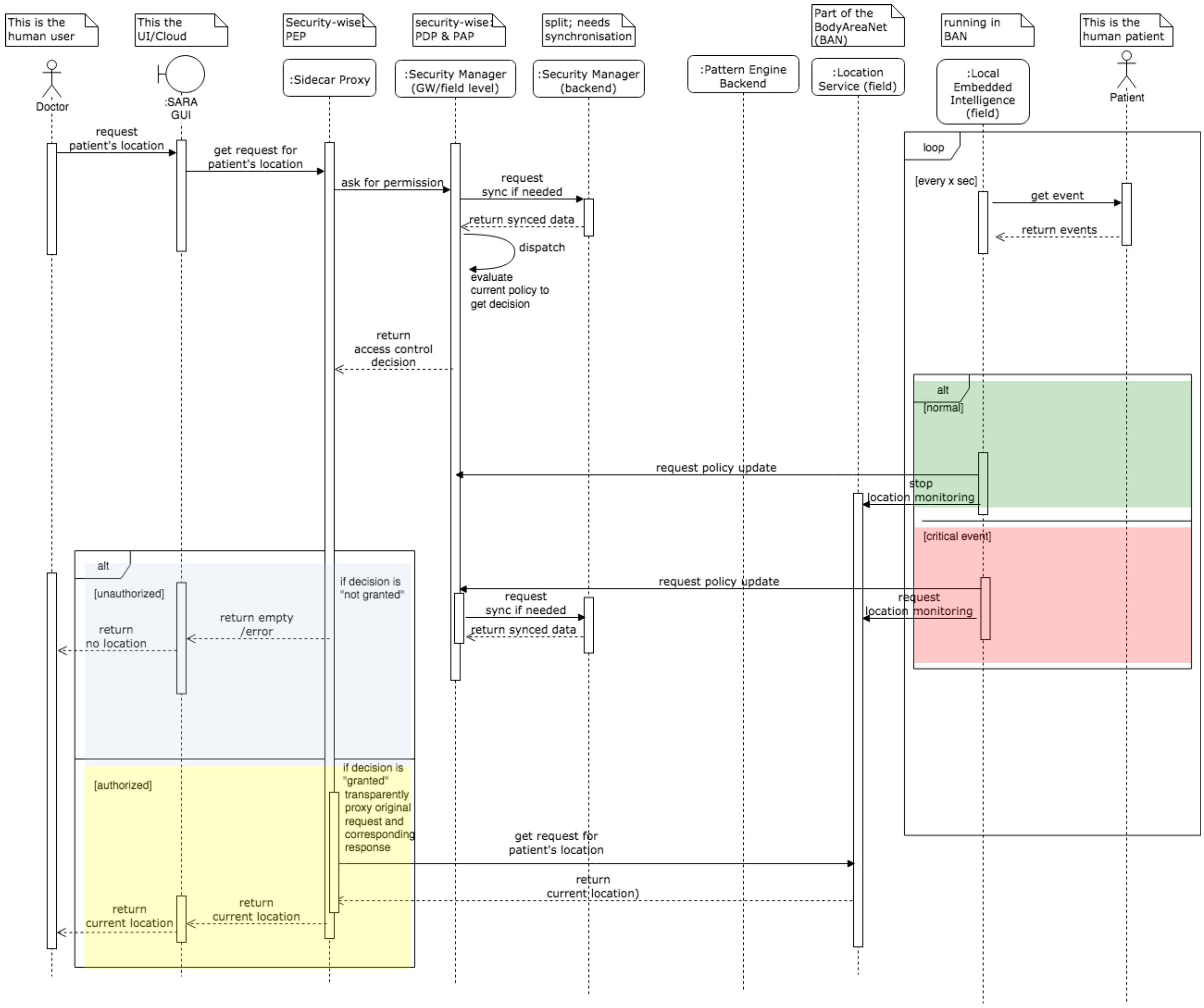


FIGURE 71 WORKFLOW FOR ADJUSTING POLICY DYNAMICALLY TO ALLOW LOCATION RETRIEVAL IN HEALTH CRITICAL EVENTS

FIGURE 71 shows how the local embedded intelligence running in the field level monitors the patient and depending on the status the policy will get updated accordingly to dynamically adapt to current events. In the case of no health critical events being detected and everything is normal, the policy is being kept very strict and compliant to the SPDI patterns (area highlighted in green in FIGURE 71). In case of a critical event the policy is adjusted and certain additional monitoring is triggered regarding the patient's location. The policy update in the case of use case 2 would allow more access, e.g. access to location, by weakening the policy, even to a point where it might not conform to the normal SPDI patterns. The flow depicted in FIGURE 71 in the yellow area then shows that a doctor can be authorized in the alternate case; this shows how SEMIoTICS can dynamically adjust security and privacy policies, e.g. grant access to a previously unauthorized service during a limited time that a critical event is ongoing. In the case the policy will deny access the PDP (the Sidecar proxy) will block the access as depicted in the bluely highlighted area.

➔ IMPLEMENTATION OF AND INTERACTION WITH THE AUTHENTICATION COMPONENT

In the following we quickly provide some overview of how to interact with the SEMIoTICS Backend Security Manager to obtain a token if you are authorized to obtain such a token.

TABLE 7: EXAMPLE OF AUTHENTICATING USING JAVASCRIPT CLIENT AND RECEIVE THE TOKEN

```
function authenticateClient(protocol,host,port,client,secret) {

  var auth = "Basic " + new Buffer(client + ":" + secret).toString("base64");
  request({
        method : "POST",
        url : protocol+"://"+host+":"+port+"/oauth2/token",
        form: {
          grant_type:'client_credentials'
        },
        headers : {
          "Authorization" : auth
       }
  },
  function (error, response, body) {
        if(error)
          throw new Error(error);
        var result = JSON.parse(body);
        var token = result.access_token;
        var type  = result.token_type;
        console.log("kind of token obtained: "+type);
        console.log("token obtained: "+token);
        getInfo(protocol,host,port,token,"client");
        getInfo(protocol,host,port,token,"user");
  });
}
```

To authenticate a client e.g. in Javascript the developer of the other components of SEMIoTICS that want to interact with the Security Manager's IDM-related component simply defines a function that calls the /oauth2/token endpoint with a POST request. To do so they define the protocol, the host and corresponding port of the Security Manager. The authorization variable defined in the header of the request is the based64-encoded client's secret. If the authentication was successful, the Security Manager returns a token for the authenticated client and the token type.
This complete request can also be sent as a simple curl request as shown in TABLE 8 at line 1. The obtained return values - in case the user with the name `MySemioticsClient2` with the password `Ultrasecretstuff` is authenticated successfully - can be seen in the lines 2 – 5: the answer contains the `access_token` as well as the `token_type`.

TABLE 8: EXAMPLE OF AUTHENTICATION USING CURL AND RECEIVE THE TOKEN

```
1.  curl  -X POST -u MySemioticsClient2:Ultrasecretstuff -
    d grant_type=client_credentials http://localhost:3000/oauth2/token
2.  {
3.    "access_token":"1A9HeY99gSYTA2o0MxIhi8pM0UVG ... rWXvrc9nqSdlj1vsEQE3INQyR0bRODEl",
4.    "token_type":"Bearer"
5.  }
```

TABLE 9: UPDATING A POLICY

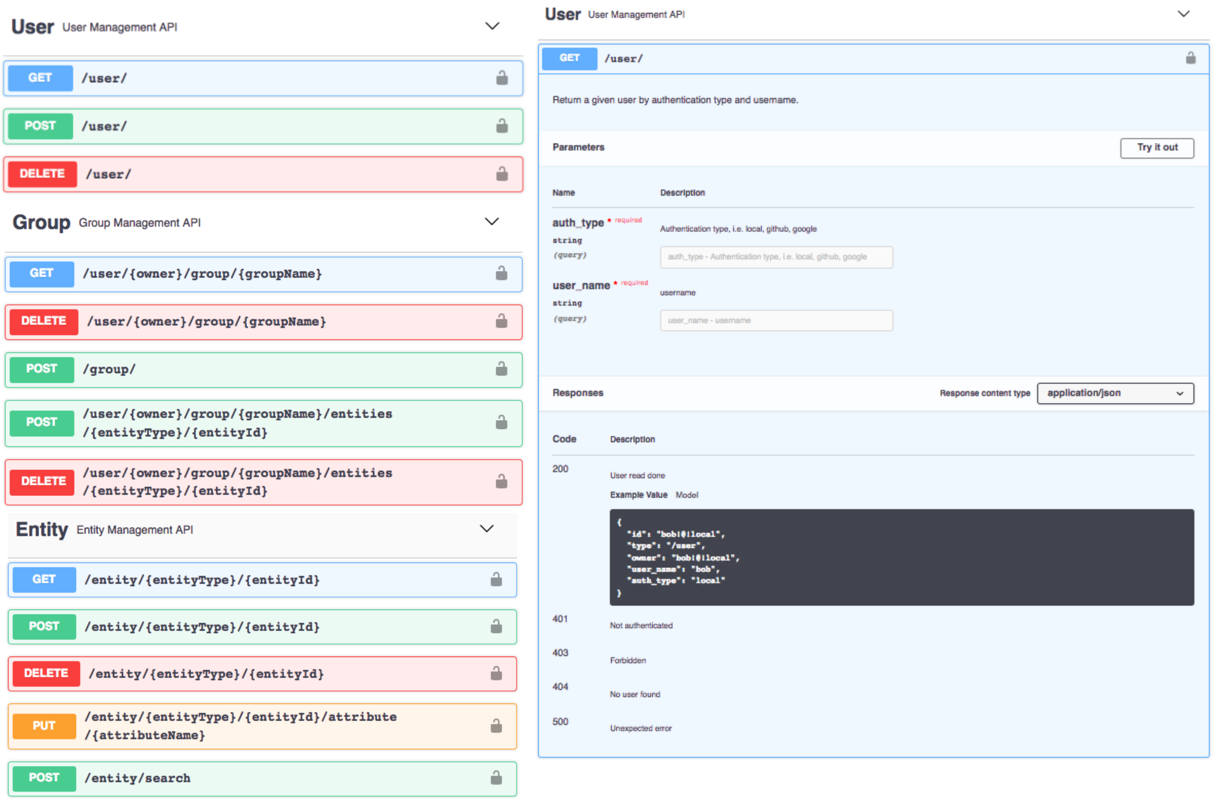
```
tokens.find(username + '!@!' + auth_type, function (_error, token) {
    sm= require('security-manager-sdk')({
        api: conf.api_url,
        idm: conf.idm_url,
        token: token
    });
    sm.policies.pap.set({
        entityId: username + '!@!' + auth_type,
        entityType: 'user',
        field: 'location',
        policy: conf.policies['fallen']
    }).then(function (r) {
        return res.status(200).send({
            text: 'Sucessfully set status to fallen'
        });
    }).catch(function (err) {
        return res.status(err.response.status).send({
            text: err.response.data.error
        })
    });
});
```

TABLE 9 shows how dynamic policy update using an SDK we have internally developed to be used to implement the policy-related functions inside of the SEMIoTICS Security Manager. First the Security Manager reads the provided `token` and checks if it is valid. Then we use the policy set function to update the policy to the given policy in the conf.policies['fallen'] variable, this activates that a policy decision can take this status into account and thus react to the dynamic situation that the patient has fallen. While updating the policy we also must provide the corresponding `entityId` and the `entityType` as well as the field (`location`) for which we intend to update the policy. As the setter function returns a Javascript promise we can use the `.then` and `.catch` clauses to further process our call.

➔ API OF THE BACKEND SECURITY MANAGER IN SWAGGER

We have distributed within SEMIoTICS the complete Interface description (AP) described in swagger using YAML-language. Swagger allows us to define the function names, the variables but also the structure of variables (e.g. lists or arrays) and would allow the software developers to automatically generate code for their clients.

Of course, also the responses and the response codes are fully specified via YAML. In the following Figure we have shown just how some of the information can rendered from the YAML. There are tools like online tool http://editor.swagger.io, which can also automatically render a clickable client in the web browser when supplying the swagger file.



User User Management API

- GET /user/
- POST /user/
- DELETE /user/

Group Group Management API

- GET /user/{owner}/group/{groupName}
- DELETE /user/{owner}/group/{groupName}
- POST /group/
- POST /user/{owner}/group/{groupName}/entities/{entityType}/{entityId}
- DELETE /user/{owner}/group/{groupName}/entities/{entityType}/{entityId}

Entity Entity Management API

- GET /entity/{entityType}/{entityId}
- POST /entity/{entityType}/{entityId}
- DELETE /entity/{entityType}/{entityId}
- PUT /entity/{entityType}/{entityId}/attribute/{attributeName}
- POST /entity/search

User User Management API

GET /user/

Return a given user by authentication type and username.

Parameters

Name	Description
auth_type * required string (query)	Authentication type, i.e. local, github, google auth_type - Authentication type, i.e. local, github, google
user_name * required string (query)	username user_name - username

Responses — Response content type: application/json

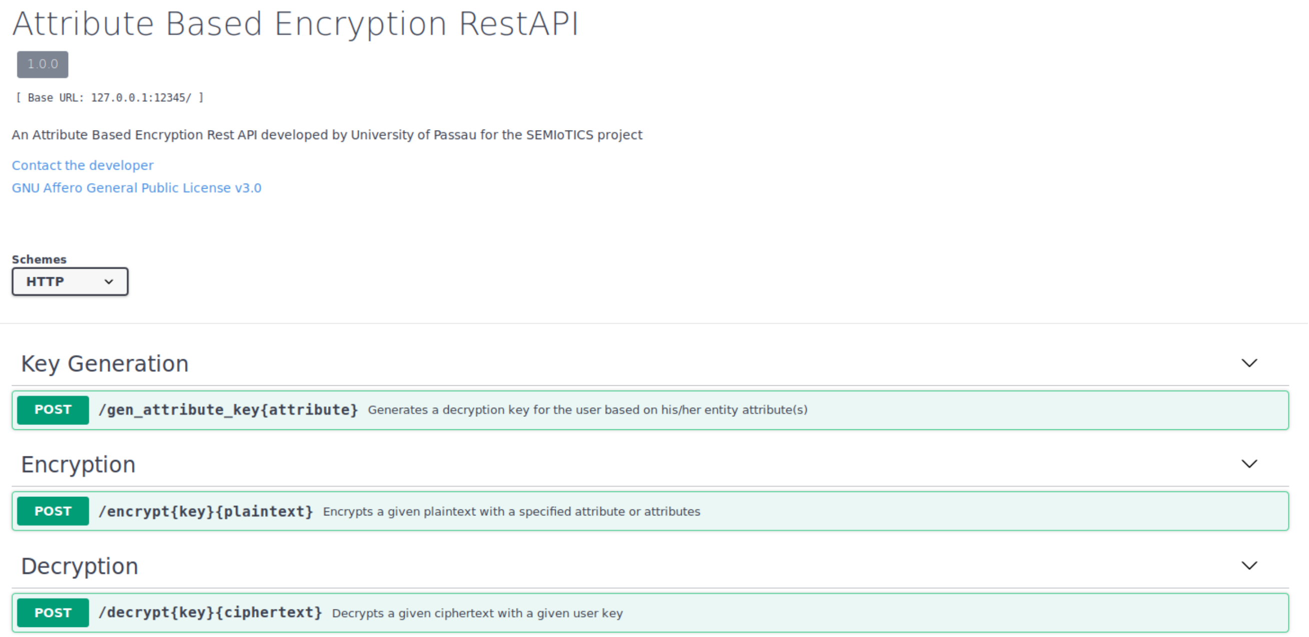
Code	Description
200	User read done Example Value Model
401	Not authenticated
403	Forbidden
404	No user found
500	Unexpected error

```
{
  "id": "bob!@!local",
  "type": "/user",
  "owner": "bob!@!local",
  "user_name": "bob",
  "auth_type": "local"
}
```

FIGURE 72: SECURITY MANAGER IDM ARCHITECTURE

➔ ATTRIBUTE BASED ENCRYPTION

Attribute-Based Encryption (ABE) determines the authorization of a user to decrypt encrypted data based on the user attributes. That means that the decryption of a ciphertext is only possible if the user can present that the user possesses a set of attributes; these attributes are enclosed in the user’s decryption key. Cryptographically the encryption fails unless the decryption keys attributes match the attributes of the ciphertext. This means that the attributes required are encoded during the encryption of the data. UP started implementing a REST API endpoint (as seen in FIGURE 73) to make available the needed ABE functionality; the cryptographic functionality is based upon the open source library OpenABE library that provides a variety of attribute-based encryption algorithms. With this API SEMIoTICS is enabled to seamlessly incorporate ABE technology into the Security Manager. This can then be used where appropriate to secure information. This ensures that the information can only be accessed by a certain entity or by a group of entities with the requested set of attributes, e.g. only entities with the attribute "doctor" are able to access encrypted medical data. At the current state of development cycle, UP focused on integrating the calls to the API endpoint in order to adopt ABE in the SEMIoTICS Security Manager.



Attribute Based Encryption RestAPI

1.0.0

[Base URL: 127.0.0.1:12345/]

An Attribute Based Encryption Rest API developed by University of Passau for the SEMIoTICS project

Contact the developer

GNU Affero General Public License v3.0

Key Generation

POST /gen_attribute_key{attribute} Generates a decryption key for the user based on his/her entity attribute(s)

Encryption

POST /encrypt{key}{plaintext} Encrypts a given plaintext with a specified attribute or attributes

Decryption

POST /decrypt{key}{ciphertext} Decrypts a given ciphertext with a given user key

FIGURE 73 OVERVIEW OF THE ABE REST-API

3.4.1.2 TESTING METHODOLOGY

To provide an overall tested functionality of the API we wrote software-based tests to evaluate the correct behavior. We therefore differentiate between 11 categories which are Entities API (with policies), Entities API, Groups API, List APIs, API for setting policies, Group Actions, Group API with policies, PEP read & write tests and overall API validation tests. Furthermore, the tests also verify the functionality of the Attribute Based Encryption RestAPI.

3.4.1.3 PERFORMANCE TEST AND KPI VALIDATION

Entities API (with policies)

#createEntity and readEntity()

✓ should reject with 404 error when data is not there

✓ should create an entity by id and return the same afterwards

#set attribute and read Entity()

✓ should reject with 404 error when attempting to update data that is not there

✓ should update an entity by id and return the proper values afterwards

#delete and readEntity()

✓ should reject with 404 error when attempting to delete data is not there

✓ should delete an entity by id

#search entity by attribute value

✓ should reject with 404 error when there is no entity with attribute value and type

✓ should get an entity based on attribute value and type

✓ should not resolve with an entity when attribute values and type match but entity_type does not

#set and read Policies

✓ set policy for entity

✓ delete policy for entity (46ms)

Entities API

#createEntity and readEntity()

✓ should reject with 404 error when data is not there

✓ should create an entity by id and return the same afterwards

✓ should reject with 400 when attempting create an entity with an undefined attribute type in strict mode

#set attribute and read Entity()

✓ should reject with 404 error when attempting to update data that is not there

✓ should reject with 400 an update when the new attribute is forbidden by the schema in strict mode

✓ should reject with 409 an update when the new attribute is forbidden by the schema

✓ should update an entity by id and return the proper values afterwards

✓ should remove an attribute that is allowed by the schema and return relevant information

✓ should reject with 400 when attempting to remove an attribute that is required by the schema

#delete and readEntity()

✓ should reject with 404 error when attempting to delete data is not there

✓ an entity by id

#search entity by attribute value

✓ should reject with 404 error when there is no entity with attribute value and type

✓ should get an entity based on attribute value and type

✓ should get an entity based on attribute value and type and entity_type

✓ should not resolve with an entity when attribute values and type match but entity_type does not

Groups API

#createGroup and readGroup()

✓ should reject with 404 error when group is not there

✓ should create a group by id and return the same afterwards

#delete and read Group()

✓ should reject with 404 error when attempting to delete data is not there

✓ should delete a group by id

#add entity to group

✓ should reject with 404 error when attempting to add a non existing entity to a group

✓ should reject with 404 error when attempting to add an existing entity to a non existing group

✓ should resolve with a modified entity after adding it to a group

#remove entity from a group

✓ should reject with 409 error when attempting to remove a non existing entity from a group

✓ should reject with 404 error when attempting to remove an existing entity from a non existing group

✓ should resolve with a modified entity without the group after removing the entity from a group where it was

✓ should resolve with two modified entities with one having no group and the other is still in the group, while adding both and removing one of them form the group

List APIs

#list entities by entity type

✓ should reject with 404 error when there is no entity in the database

✓ should get an entity based on its type

#ListGroups()

✓ should reject with 404 error when group is not there

✓ should return a group after its creation

✓ should return all groups

API (set Policies in PAP)

#CreateEntity()

✓ should enable any non-set subfield of actions field in the policy structure to be read and written according to the default policy in actions

✓ should set the highest level for the policy hierarchy to read only (not admin and not owner check)

✓ should enforce meta policies in the configuration of attributes (policies.role) as meta policies

UsedLessThan lock

#readEntity()

✓ should stop resolving with the password, after five actions on the password field

GROUP ACTIONS

✓ should return user entity "elisa" with password attribute for owner elisa

✓ should return user entity "elisa" without password attribute for user admin

✓ should return user entity "elisa" with password attribute for group member bob

Groups API with policies

#createGroup and readGroup()

✓ should reject with 404 error when group is not there

✓ should create a group by id and return the same afterwards

#delete and read Group()

✓ should reject with 404 error when attempting to delete data is not there

✓ should delete a group by id

#add entity to group

✓ should reject with 404 error when attempting to add a non existing entity to a group

✓ should reject with 404 error when attempting to add an existing entity to a non existing group

✓ should resolve with a modified entity after adding it to a group

#remove entity from a group

✓ should reject with 409 error when attempting to remove a non existing entity from a group

✓ should reject with 404 error when attempting to remove an existing entity from a non existing group

✓ should resolve with a modified entity without the group after removing the entity from a group where it was

API (PEP Read test)

#readEntity()

✓ should resolve with a declassified entity for different users (password not there)

✓ should resolve with a declassified entity for different users for nested properties (credentials.dropbox not there)

✓ should resolve with the complete entity when the owner reads it including inner properties (credentials.dropbox)

✓ should resolve with the complete entity when the owner reads it

✓ should resolve with the entity when attempting to create an entity with the proper role

#findEntitiesByAttribute()

✓ should resolve with an array without entities for which the attributes used in the query are not allowed to be read by the policy

API (PEP Write Test)

#createEntity()

✓ should reject with 403 and conflicts array in the object when attempting to create an entity without the proper role

✓ should resolve with the entity when attempting to create an entity with the proper role

#setAttribute()

✓ should reject with 403 and conflicts array when attempting to update an entity's attribute without the proper role and not owner

✓ should reject with 403 and conflicts array when an owner (non-admin) attempts to update his own role

✓ should resolve when attempting to update an entity attribute with the proper role

#deleteEntityAttribute()

✓ should resolve when attempting to update an entity attribute with the proper role

API (Validation test)

✓ should reject with 400 an entity when with an existing type but with an attribute missing

✓ should reject with 409 when attempting to create an entity with a forbidden attribute name

#createEntity()

Attribute Based Encryption (ABE) **AP**I

# Testing Key-Policy ABE (KP-ABE)

✓ Test: Key generation

✓ Test: Encryption

✓ Test: Successful Decryption

✓ Test: Decryption with wrong attributes should fail

# Testing Ciphertext-Policy ABE (CP-ABE)

✓ Test: Key generation

✓ Test: Encryption

✓ Test: Successful Decryption

✓ Test: Decryption with wrong attributes should fail

3.4.2 SEMIOTICS PATTERN ORCHESTRATOR AND ENGINE

3.4.2.1 COMPONENT ARCHITECTURE

FIGURE 74 below depicts the Pattern Engine related components distributed among the three layers of the SEMIoTICS architecture. Pattern Orchestrator and one of the three Pattern Engines are lying at the Application Orchestration Layer, at the Backend. A second Pattern Engine is located at the SDN/NFV orchestration layer in the SDN controller. The last Pattern Engine can be found at the Field layer, at the IoT Gateway.
Pattern Orchestrator is a module that features a semantic reasoner able to understand instantiated Recipes, received from the Recipe Cooker and transform them into composition structures (orchestrations). Backend Pattern Engine enables the capability to insert, modify, execute and retract patterns at design or at runtime. Moreover, it may receive fact updates from the individual Pattern Engines present at the lower layers (Network & Field), allowing it to have an up-to-date view of the SPDI state of said layers and the corresponding components. Pattern Engine at the SDN controller offers the same functionality allowing entities that interact with the controller to be managed based on SPDI patterns at design and at runtime. Pattern Engine at the Field layer, since it is deployed on the IoT/IIoT gateway that has limited capabilities, is a lightweight version of the Backend Pattern Engine.

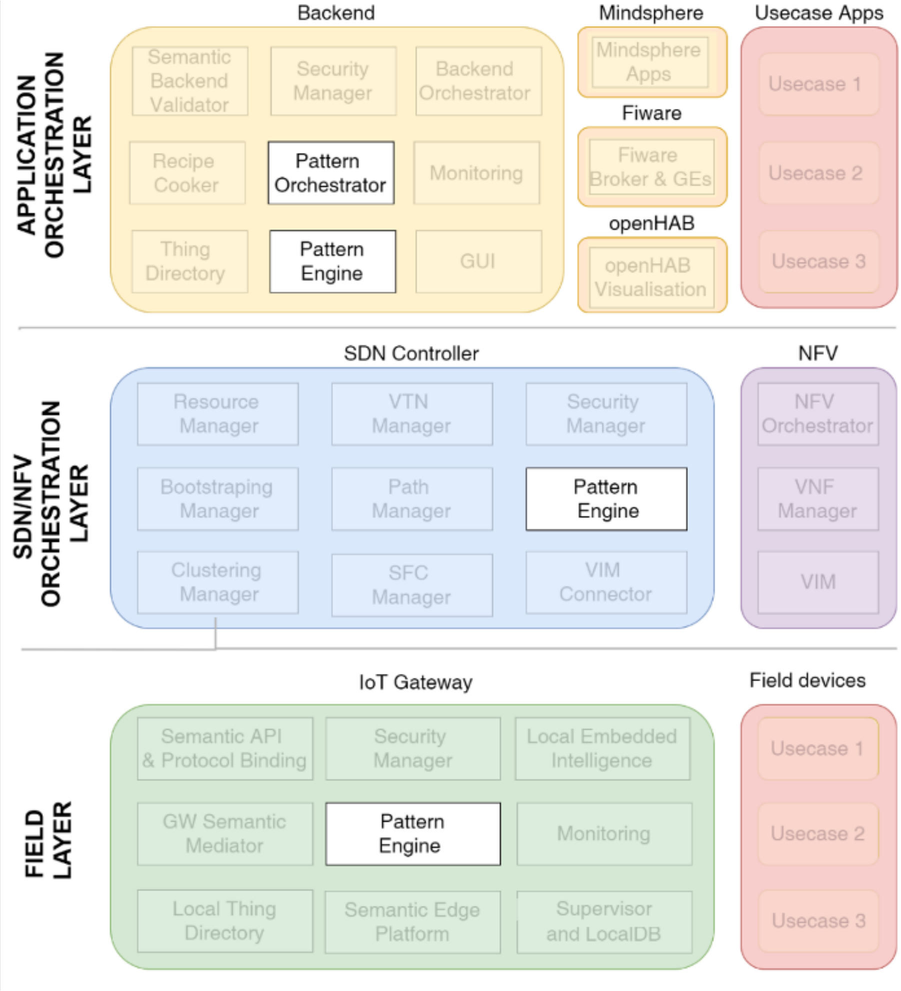


FIGURE 74: PATTERN RELATED COMPONENTS IN SEMIOTICS ARCHITECTURE

3.4.2.2 APIS

FIGURE 75 shows the interactions that take place among the pattern related components. As we can see, Pattern Orchestrator interacts with all the Pattern Engines of the three SEMIoTICS layers. The purpo se of this interaction is to dispatch the created Drools facts and can take place via the common API exposed by the Pattern Engines.

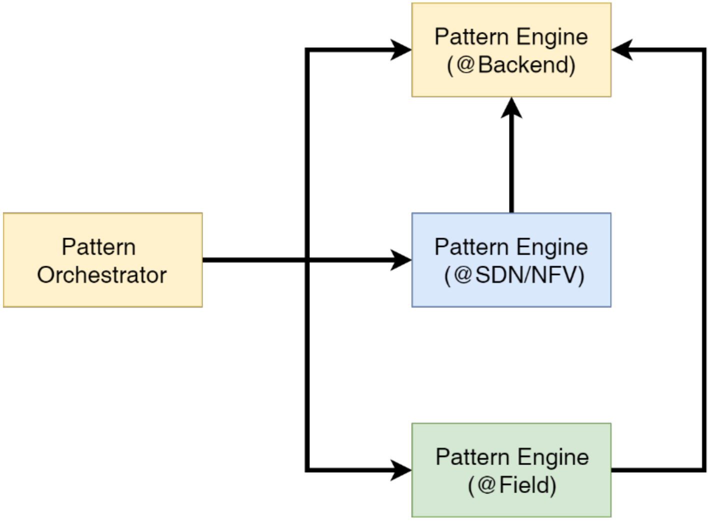
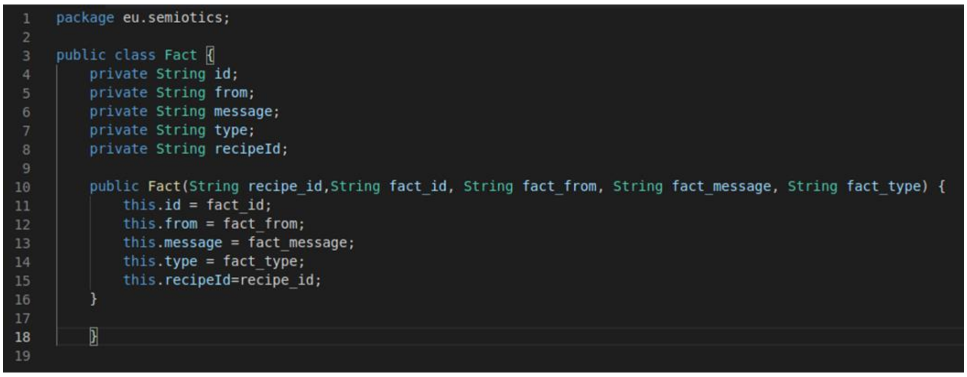


FIGURE 75: INTERACTIONS AMONG PATTERN RELATED COMPONENTS

Moreover, the said API is used by the SDN controller and Field Pattern Engine that send at runtime fact updates to the Backend Pattern Engine, allowing the latter to have an up-to-date view of the SPDI state of SDN layer and the corresponding components. The main web services exposed from the Backend Pattern Engine API are:

- *addFact*
- *factRemove*
- *factUpdate*
- *factStatus*
- *insertRule*

The above correspond to the creation, retrieval, deletion of facts and creation of rules. In more detail, the *addFact* REST service is used by the Pattern Orchestrator for the communication of new Drools facts of a new IoT Service orchestration. It can also be used by the Pattern Engines of the Network and Field layers in the case of new fact discovery. In any case the JSON that is sent, is based on the Fact Java class that can be seen in the code snippet below, in FIGURE 76.



```java
package eu.semiotics;

public class Fact {
    private String id;
    private String from;
    private String message;
    private String type;
    private String recipeId;

    public Fact(String recipe_id,String fact_id, String fact_from, String fact_message, String fact_type) {
        this.id = fact_id;
        this.from = fact_from;
        this.message = fact_message;
        this.type = fact_type;
        this.recipeId=recipe_id;
    }

    }

```

FIGURE 76: FACT JAVA CLASS ATTRIBUTES USED IN REST SERVICES JSON

Moreover, the *factRemove* is used in order for a fact to be deleted from the Drools Memory of the Backend Pattern Engine. The *factUpdate* is used again by the Pattern Orchestrator in case some changes need to be applied to a Drools Fact. The *factStatus* REST service returns the current status of a special type of Drools facts, the instances of Property class. These instances are used to describe SPDI and QoS properties for the components of an IoT Service orchestrator. This REST service could also be used for the visualization of the

SPDI properties of an orchestrator in the SEMIoTICS GUI. Finally, the *insertRule* REST service is used only by the Pattern Orchestrator to communicate Drools Rules to the Pattern Engines for the reasoning of the SPDI and QoS properties.

The interaction among the pattern related and other components (NFV orchestrator, monitoring) can also be seen in the sequence diagram of FIGURE 77.

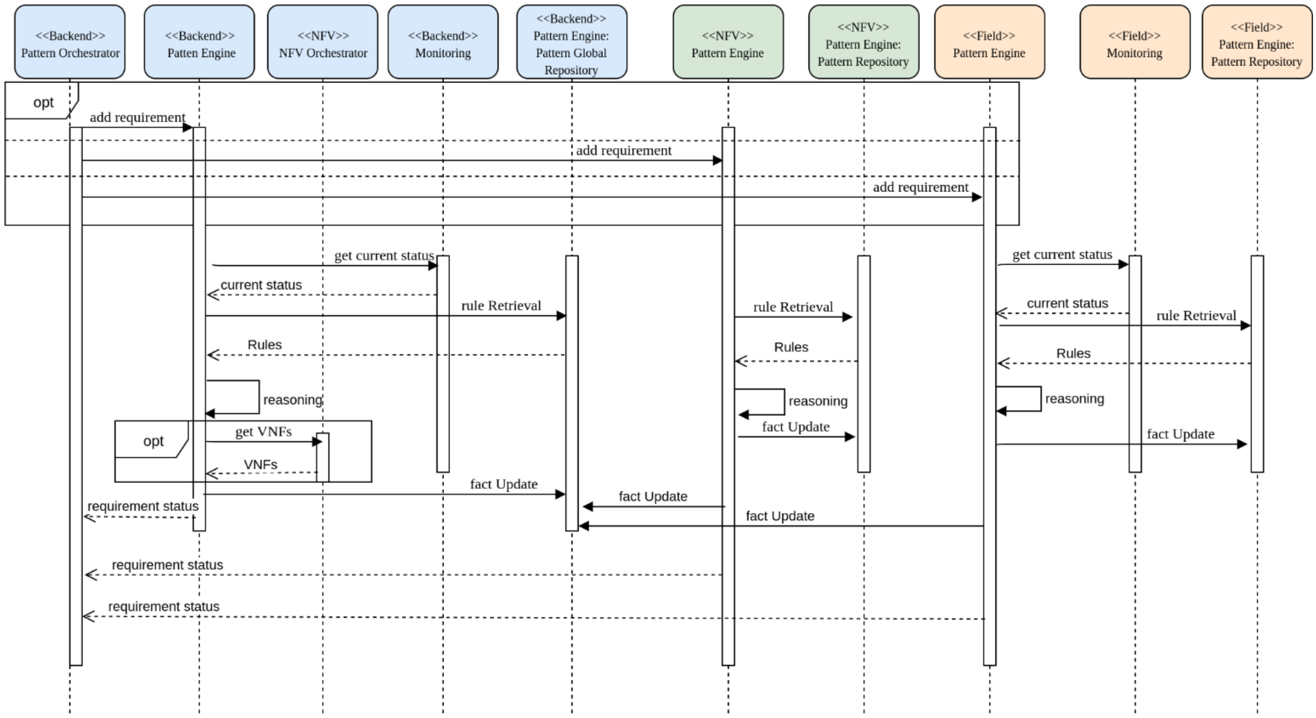


FIGURE 77: PATTERN RELATED COMPONENTS INTERACTIONS

As we can see, the Pattern Orchestrator chooses to send the SPDI requirement to one or more Pattern Engines depending on the case. The requirement is in the form of a property on an orchestration component. This triggers a sequence of events that consists of several steps. Every Pattern Engine uses the available information from the monitoring components in each layer and in combination with the rules and facts already stored in Pattern Repository also in the same layer, reasons for the final status of the said requirement. In addition, the Pattern Engines that exist in the network layer as well as in the field layer, propagate their facts not only to their local Pattern Repository, but at the Pattern repository as well, at the Backend layer. When the requirement is related to some VNFs, interaction with the NFV orchestrator also occurs in order for the final status requirement to be formed.

3.4.2.3 TESTING METHODOLOGY

In order to test the functionalities of all the Pattern related components of SEMIoTICS, as they are described above, we use the Proxmox Virtual environment (FIGURE 79) to create Virtual Machines (VMs) hosted in an INTEL NUC (FIGURE 78). The created VMs have some hardware and software requirements, which are shown in TABLE 10 below.

TABLE 10: VM REQUIREMENTS

Component	Software	CPU	Memory	Disk
Pattern Orchestrator	Ubuntu 18.04 LTS	4cores	4 GB	10 GB

Backend Pattern Engine	Ubuntu 18.04 LTS	4 cores	4 GB	10 GB
SDN Pattern Engine	Ubuntu 18.04 LTS	4 cores	4 GB	10 GB

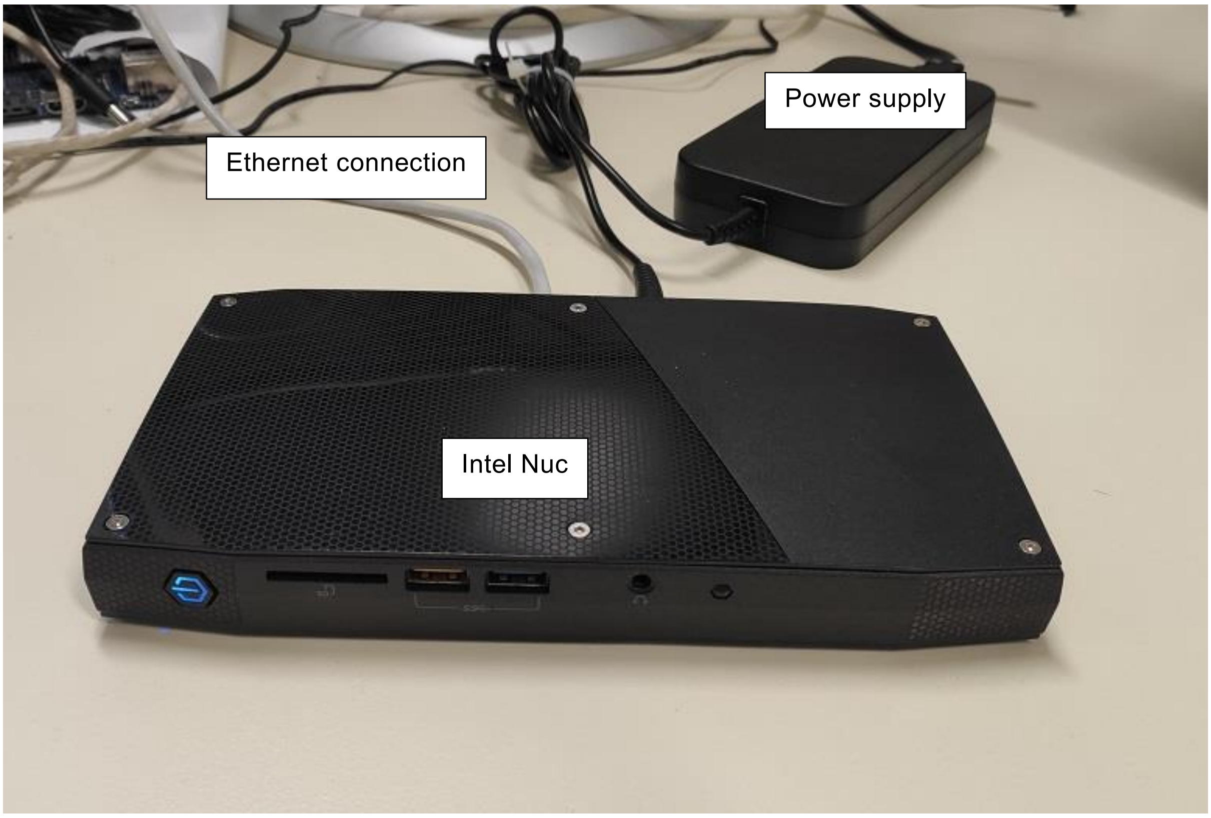
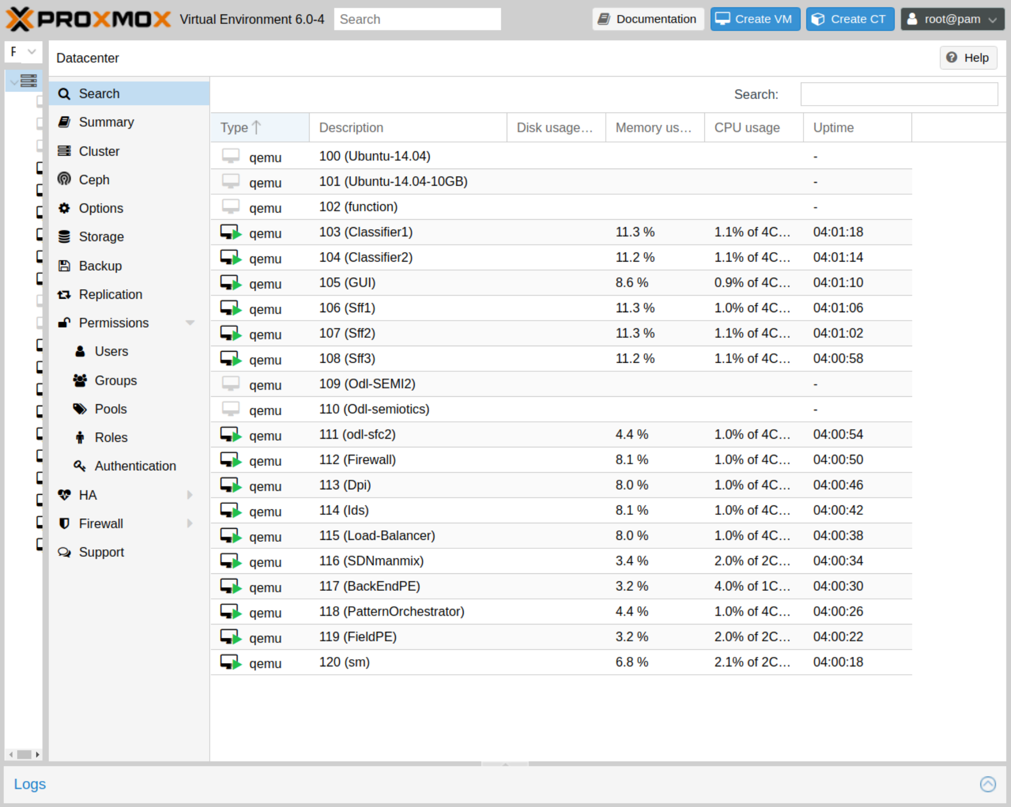


FIGURE 78 INTEL NUC



Type	Description	Disk usage…	Memory us…	CPU usage	Uptime
qemu	100 (Ubuntu-14.04)				-
qemu	101 (Ubuntu-14.04-10GB)				-
qemu	102 (function)				-
qemu	103 (Classifier1)		11.3 %	1.1% of 4C…	04:01:18
qemu	104 (Classifier2)		11.2 %	1.1% of 4C…	04:01:14
qemu	105 (GUI)		8.6 %	0.9% of 4C…	04:01:10
qemu	106 (Sff1)		11.3 %	1.0% of 4C…	04:01:06
qemu	107 (Sff2)		11.3 %	1.1% of 4C…	04:01:02
qemu	108 (Sff3)		11.2 %	1.1% of 4C…	04:00:58
qemu	109 (Odl-SEMI2)				-
qemu	110 (Odl-semiotics)				-
qemu	111 (odl-sfc2)		4.4 %	1.0% of 4C…	04:00:54
qemu	112 (Firewall)		8.1 %	1.0% of 4C…	04:00:50
qemu	113 (Dpi)		8.0 %	1.0% of 4C…	04:00:46
qemu	114 (Ids)		8.1 %	1.0% of 4C…	04:00:42
qemu	115 (Load-Balancer)		8.0 %	1.0% of 4C…	04:00:38
qemu	116 (SDNmanmix)		3.4 %	2.0% of 2C…	04:00:34
qemu	117 (BackEndPE)		3.2 %	4.0% of 1C…	04:00:30
qemu	118 (PatternOrchestrator)		4.4 %	1.0% of 4C…	04:00:26
qemu	119 (FieldPE)		3.2 %	2.0% of 2C…	04:00:22
qemu	120 (sm)		6.8 %	2.1% of 2C…	04:00:18

FIGURE 79 PROXMOX ENVIRONMENT

Moreover, in cycle 1, a virtual network topology is created using Mininet (FIGURE 80). This network topology consists of the OpenDaylight SDN Controller (OSC), one openflow switch (s1) and two hosts. The hosts correspond to two Raspberry Pi boards that are part of the testing orchestration described below.

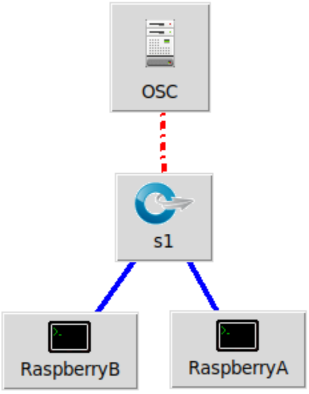
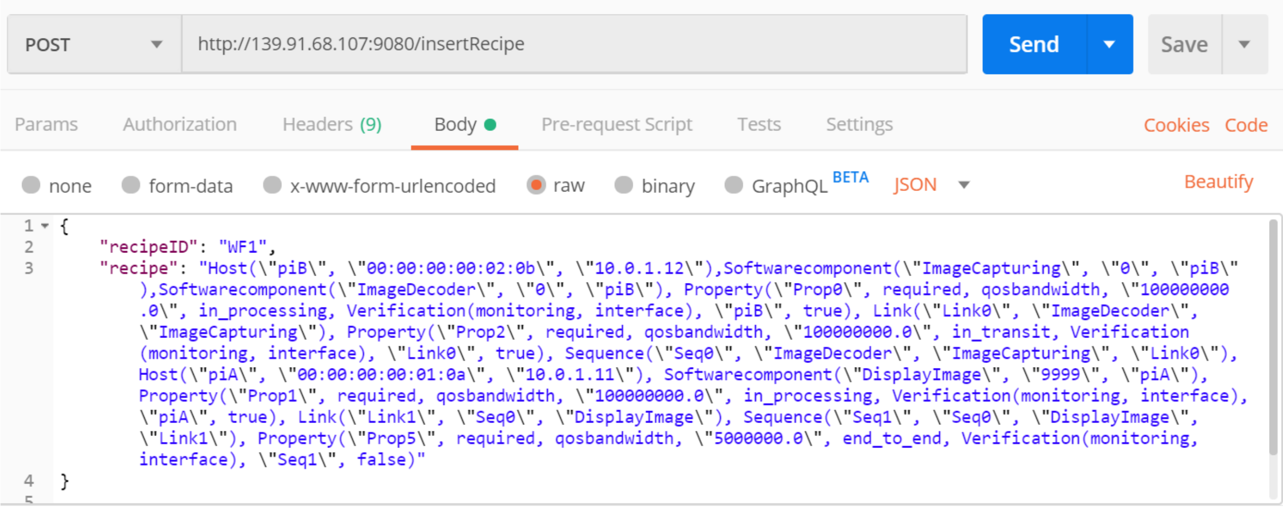


FIGURE 80: MININET NETWORK TOPOLOGY

The first step of our testing methodology is to send a request to Pattern Orchestrator, which is expected to receive instantiated orchestrations of IoT services (i.e. Recipes). We use Postman as an API client to send these requests to the exposed REST API of the Pattern Orchestrator and view the returned responses. Postman allows us to create REST requests with the needed headers and the body we are interested in. FIGURE 81 below is an example of such a REST request.



http://139.91.68.107:9080/insertRecipe

```
{
    "recipeID": "WF1",
    "recipe": "Host(\"piB\", \"00:00:00:00:02:0b\", \"10.0.1.12\"),Softwarecomponent(\"ImageCapturing\", \"0\", \"piB\"
      ),Softwarecomponent(\"ImageDecoder\", \"0\", \"piB\"), Property(\"Prop0\", required, qosbandwidth, \"100000000
      .0\", in_processing, Verification(monitoring, interface), \"piB\", true), Link(\"Link0\", \"ImageDecoder\",
      \"ImageCapturing\"), Property(\"Prop2\", required, qosbandwidth, \"100000000.0\", in_transit, Verification
      (monitoring, interface), \"Link0\", true), Sequence(\"Seq0\", \"ImageDecoder\", \"ImageCapturing\", \"Link0\"),
      Host(\"piA\", \"00:00:00:00:01:0a\", \"10.0.1.11\"), Softwarecomponent(\"DisplayImage\", \"9999\", \"piA\"),
      Property(\"Prop1\", required, qosbandwidth, \"100000000.0\", in_processing, Verification(monitoring, interface),
      \"piA\", true), Link(\"Link1\", \"Seq0\", \"DisplayImage\"), Sequence(\"Seq1\", \"Seq0\", \"DisplayImage\",
      \"Link1\"), Property(\"Prop5\", required, qosbandwidth, \"5000000.0\", end_to_end, Verification(monitoring,
      interface), \"Seq1\", false)"
}
```

FIGURE 81: POSTMAN REST INSERTRECIPE REQUEST

As we can see, the REST request for sending an orchestration along with a requirement uses the POST method. The URL that is used is http://[OrchestratorIP]/insertRecipe. In the body of the request, there is a Recipe object with two field names, the *recipeID* and the *recipe*. The former is used as an identifier for all the Pattern related components, to distinguish each of the incoming recipes. The latter is the description of the recipe itself, which includes three element types. The first element type refers to all the involved components of the orchestration. The second element type refers to the way the said components communicate with each other and can be either links or orchestrations such as sequences, merges, splits and choices. Finally, the last element type refers to the SPDI/QoS properties of the components. When these properties are to be checked whether they hold or not, they are called requirements.

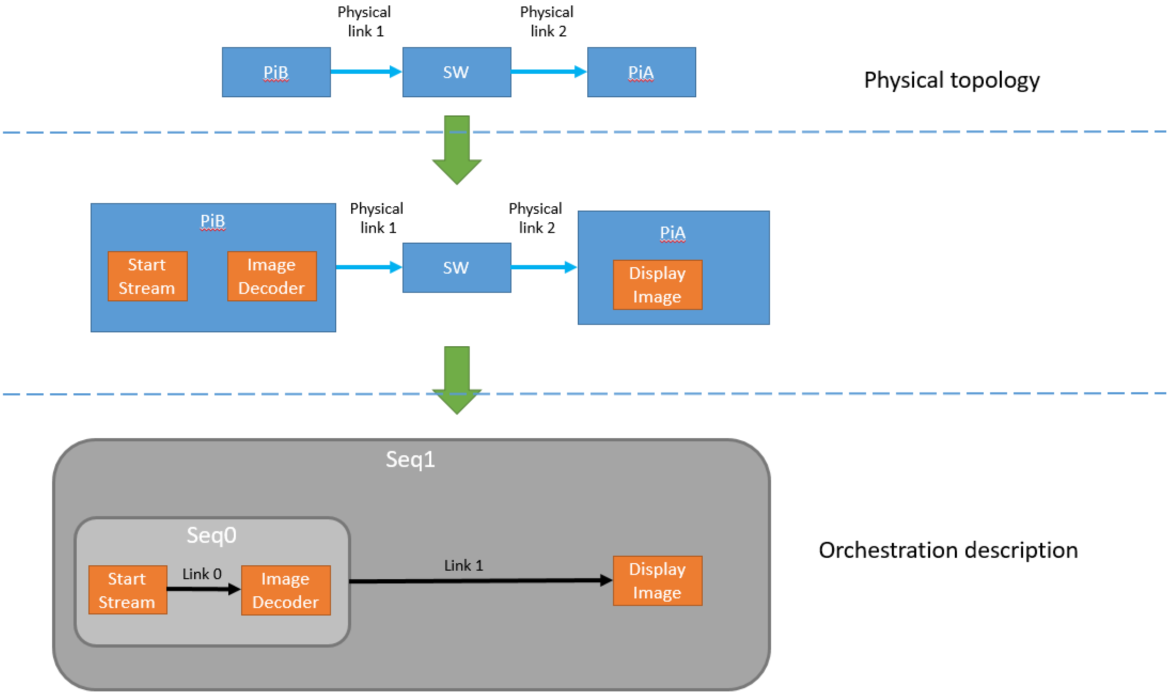


FIGURE 82: TEST ORCHESTRATION

In more detail, the recipe that use in the testing methodology includes two Raspberry Pi boards (Hosts), named *PiB* and *PiA*, which are connected through a switch with two physical links (Physical link1 and physical link2). This is depicted in the first part (*Physical topology*) of the FIGURE 82, which is our test orchestration.

The intermediate part shows the software components that are installed in the two Raspberry Pi boards. Two software components are deployed in the first one, an *ImageCapturing* software and an *Image Decoder* software. As far as the second Raspberry Pi board is concerned, a *Display Image* software is deployed.

The last part of the figure, *Orchestration description*, shows how the software components communicate and the QoS properties they own. Those in *PiB* are connected with an orchestration Link (*Link0*) and they are in Sequence (*Seq0*), meaning that the output of the first becomes the input of the second. *Link0* corresponds to a communication that is achieved due to their deployment at the same host. Regarding their properties, a QoS property for the maximum bandwidth at which they can send and receive data is assigned to both of them, with value 11400000 bps (*qosbandwidth*). This property is also assigned to the Link between the two software components. The *Display Image* software has the same QoS property. Additionally, there is an orchestration Link (*Link1*) between *Seq0* and *Display Image*. This Link does not correspond to a network physical link but to a path between its source and its destination, which may include more than one physical links and other network components among them. SDN Pattern Engine undertakes the assignment to find out the said path to the orchestration Link and additionally, to validate its property. In the orchestration used in our testing methodology the *Link1* corresponds to two physical links and a switch between them as we can see in FIGURE 82. As a result, Seq0 and Display Image connected with Link1 form Seq1. Finally, the last Property that is mentioned in the request is the QoS property that refers to the whole orchestration. According to it, we want to know if the minimum bandwidth throughout the whole test orchestration is 5Mbps.

Patter Orchestrator receives the incoming Recipe and creates instances of the corresponding Java classes (Host, Softwarecomponent, Link, Sequence, Property), which correspond to Drools facts in the Pattern Engine. These Java instances are then sent to the SDN Pattern Engine through one of its REST APIs, called *addFact*. The request for sending a Drools fact uses the POST method and the URL is http://[PatternEngineIP]/ patternengine:addFact. In the body of the request, there is a Fact object with five field names presented in the TABLE 11 below.

TABLE 11: FIELD NAMES OF FACT OBJECT

Name	Description	Valid values
recipeID	the ID of the recipe the fact belongs to	(e.g. “WF1”)
factID	the identifier of the fact object itself	(e.g. “WF1-1”)
from	originated SEMIoTICS component sender	(e.g. “Orchestrator”)
factMessage	the fact itself	(e.g. “DisplayImage”)
type	the object type of the fact	(e.g. “Softwarecomponent”)

Pattern Engine receives all the Java instances sent by the Pattern Orchestrator. Each of the received instances are inserted into the working memory of the Drools Rule Engine, as Drools facts. Being there, they can trigger Drools Rules that are pre-inserted in the Pattern Engine. Based on the orchestration presented in FIGURE 81, the following Drools facts are created:

TABLE 12: DROOLS FACTS DERIVED FROM THE TESTING ORCHESTRATION

#	Type	Description
1	Softwarecomponent	ImageCapturing 0 piB
2	Softwarecomponent	ImageRecorder 0 piB
3	Softwarecomponent	DisplayImage 9999 piA
4	Host	piB b8:27:eb:4b:0a:7c 10.0.1.12
5	Host	piA b8:27:eb:ae:aa:31 10.0.1.11
6	Link	Link0 ImageCapturing ImageRecorder
7	Link	Link1 Seq0 DisplayImage
8	Sequence	Seq0 StartStream ImageRecorder Link0
9	Sequence	Seq1 Seq0 DisplayImage Link1
10	Property	qosbandwidth 1.0E8 in_processing PiB true
11	Property	qosbandwidth 1.0E8 in_processing PiA true
12	Property	qosbandwidth 1.0E8 in_processing Link0 true

13	Property	qosbandwidth 5000000.0 end_to_end Seq1 false

Whenever the SDN Controller Pattern Engine receives a Property that refers to a Host, it creates the corresponding Property for each of the software components that are deployed at that host. In our case, a *qosbandwdith* Property is created for the two software components of Host *PiB* and the software component of PiA (TABLE 13).

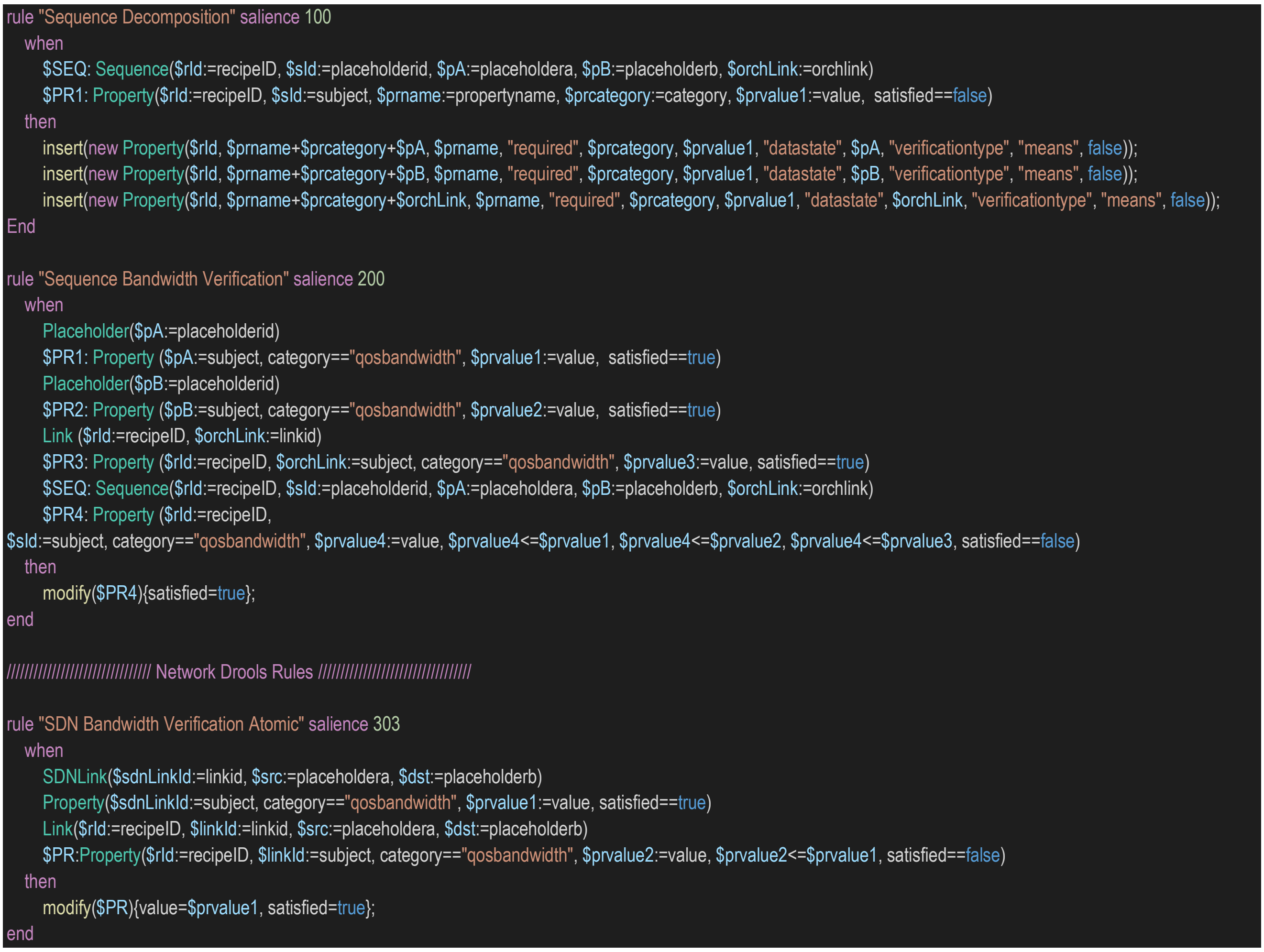
TABLE 13: ADDITIONAL DROOLS FACTS CREATED BY PATTERN ENGINE

#	Type	Description
10	Property	qosbandwidth 1.0E8 in_processing ImageCapturing true
11	Property	qosbandwidth 1.0E8 in_processing ImageRecorder true
12	Property	qosbandwidth 1.0E8 in_processing DisplayImage true

The last Property Drools Fact refers to the whole test orchestration and corresponds to our QoS requirement. Based on this Property, the Drools Rules will fire and the Pattern Engine will reason on the QoS property of the orchestration and will respond to Pattern Orchestrator.

We should mention here that there is no need for the Pattern Orchestrator to create an instance of the corresponding Java class of type Property for *Link1*, although it is necessary for the Drools Rules we use for this test. This Fact is created by the SDN Controller Pattern Engine itself and is added to the working memory of Drools Rule Engine.

The Rules used for our test were pre-inserted and are shown in the code snippet below.



```
rule "Sequence Decomposition" salience 100
  when
    $SEQ: Sequence($rId:=recipeID, $sId:=placeholderid, $pA:=placeholdera, $pB:=placeholderb, $orchLink:=orchlink)
    $PR1: Property($rId:=recipeID, $sId:=subject, $prname:=propertyname, $prcategory:=category, $prvalue1:=value,  satisfied==false)
  then
    insert(new Property($rId, $prname+$prcategory+$pA, $prname, "required", $prcategory, $prvalue1, "datastate", $pA, "verificationtype", "means", false));
    insert(new Property($rId, $prname+$prcategory+$pB, $prname, "required", $prcategory, $prvalue1, "datastate", $pB, "verificationtype", "means", false));
    insert(new Property($rId, $prname+$prcategory+$orchLink, $prname, "required", $prcategory, $prvalue1, "datastate", $orchLink, "verificationtype", "means", false));
End

rule "Sequence Bandwidth Verification" salience 200
  when
    Placeholder($pA:=placeholderid)
    $PR1: Property ($pA:=subject, category=="qosbandwidth", $prvalue1:=value,  satisfied==true)
    Placeholder($pB:=placeholderid)
    $PR2: Property ($pB:=subject, category=="qosbandwidth", $prvalue2:=value,  satisfied==true)
    Link ($rId:=recipeID, $orchLink:=linkid)
    $PR3: Property ($rId:=recipeID, $orchLink:=subject, category=="qosbandwidth", $prvalue3:=value, satisfied==true)
    $SEQ: Sequence($rId:=recipeID, $sId:=placeholderid, $pA:=placeholdera, $pB:=placeholderb, $orchLink:=orchlink)
    $PR4: Property ($rId:=recipeID,
$sId:=subject, category=="qosbandwidth", $prvalue4:=value, $prvalue4<=$prvalue1, $prvalue4<=$prvalue2, $prvalue4<=$prvalue3, satisfied==false)
  then
    modify($PR4){satisfied=true};
end

/////////////////////////////// Network Drools Rules ////////////////////////////////

rule "SDN Bandwidth Verification Atomic" salience 303
  when
    SDNLink($sdnLinkId:=linkid, $src:=placeholdera, $dst:=placeholderb)
    Property($sdnLinkId:=subject, category=="qosbandwidth", $prvalue1:=value, satisfied==true)
    Link($rId:=recipeID, $linkId:=linkid, $src:=placeholdera, $dst:=placeholderb)
    $PR:Property($rId:=recipeID, $linkId:=subject, category=="qosbandwidth", $prvalue2:=value, $prvalue2<=$prvalue1, satisfied==false)
  then
    modify($PR){value=$prvalue1, satisfied=true};
end
```

```
rule "SDN Bandwidth Verification" salience 302
  when
    SDNLink($lId:=linkid, $src:=placeholdera, $node:=placeholderb)
    Property($lId:=subject, category=="qosbandwidth", $prvalue1:=value, satisfied==true)
    Link($rId:=recipeID, $lId2:=linkid, $node:=src, $dst:=dst)
    Property($rId:=recipeID, $lId2:=subject, category=="qosbandwidth", $prvalue2:=value, satisfied==true)
    Link($rId:=recipeID, $lId3:=linkid, $src:=src, $dst:=dst, $lId3!=$lId2)
    $PR:Property($rId:=recipeID, $lId3:=subject, category=="qosbandwidth", $prvalue3:=value, $prvalue3<=$prvalue2, $prvalue3<=$prvalue1, satisfied==false)
  then
    modify($PR){value=Math.min($prvalue1, $prvalue2), satisfied=true};
end

rule "SDN Bandwidth Decomposition" salience 301
  when
    SDNLink($lId:=linkid, $src:=placeholdera, $node:=placeholderb)
    Property($lId:=subject, $prname:=propertyname, $prcategory:=category, $prcategory=="qosbandwidth", $prvalue1:=value, satisfied==true)
    isPath($node, $dst, $prcategory;)
    Link($rId:=recipeID,$sId:=linkid, $src:=src, $dst:=dst)
    Property($rId:=recipeID, $sId:=subject, $prcategory:=category, $prvalue2:=value, $prvalue2<=$prvalue1, satisfied==false)
  then
    Link l1 = new Link($rId, $src+$node, $src+$node, $src, $node, $src, $node);
    insert(l1);
    Property prA = new Property($rId,
$prname+$prcategory+l1.getLinkid(), $prname, "required", $prcategory, $prvalue1, "datastate", l1.getLinkid(), "verificationtype", "means", true);
    insert(prA);
    Link l2 = new Link($rId, $node+$dst, $node+$dst, $node, $dst, $node, $dst);
    insert(l2);
    Property prB = new Property($rId,
$prname+$prcategory+l2.getLinkid(), $prname, "required", $prcategory, $prvalue2, "datastate", l2.getLinkid(), "verificationtype", "means", false);
    insert(prB);
end

query isPath(String a, String b,String c)
  SDNLink(a, b; checkPropertyName(c))
  or
  SDNLink(z,b;checkPropertyName(c)) and isPath(a,z,c;)
end
```

The first rule, *Sequence – Decomposition*, is fired for every Sequence in our testing orchestration that has a SPDI/QoS property and creates a property of the same category and the same value for every component of the Sequence. Every time the second rule, *Sequence Bandwidth Verification*, is fired, verifies a *qosbandwidth* Property of a Sequence. According to the rule, if i) all the components of a Sequence, i.e. two Placeholders and a Link between them, have a Property of *qosbandwidth* category ($PR1, $PR2, $PR3); and ii) the value of the orchestration Property ($prvalue4) is lower than the values of the Properties of the Sequence components ($prvalue1, $prvalue2, $prvalue3), then the corresponding Property of the Sequence in question is verified.

After the initial insertRecipe request, the status of the Properties Drools Facts is as it is shown in step 1 in FIGURE 83. The first rule that is triggered is the *Sequence Decomposition* rule for *Seq1*. The presence of the unverified *qosbandwidth* Property creates Properties of the same category and the same value for the three components *Seq0*, *Link1*, *DisplayImage* (step 2 in FIGURE 83). Our goal is to verify the *qosbandwidth* Property of *Seq1*, and for this we need the *Sequence Bandwidth Verification* rules to run. In order the said rule to run for *Seq0*, we need an unverified *qosbandwidth* Property for *Seq0* and verified *qosbandwidth* Properties for all the three components of *Seq0*, i.e. *ImageCapturing*, *Link0* and *ImageRecorder.* The three verified Properties are provided by the initial insertRecipe request and the run of the previous rule created the unverified *qosbandwidth* Property for *Seq0.* The result of the rule is the verification of *qosbandwidth* Property for *Seq0* (step 3 in FIGURE 83).

The SDN Controller Pattern Engine verifies the *qosbandwidth* Property of *Link1* (step 4 in FIGURE 83). As we have already mentioned above, Link1 corresponds to a path from the source of the link to its destination, which in our test topology consists of two physical links and a switch between them (FIGURE 82). This information becomes available to SDN Controller Pattern Engine, leveraging the Network Drools Rules. The purpose of these rules is to discover the path that forms the said orchestration Link and examine if the SPDI/QoS property of the Link holds for each component of the path. If the above are in effect, the rules verify the SPDI/QoS property of the aforementioned Link.

After that, the *Sequence Bandwidth Verification* rule is triggered again for Seq1 this time. All the three components of *Seq1*, i.e. *Seq0*, *Link1* and *DisplayImage,* have a Property of category *qosbandwidth* that is already verified. The verified Property of *DisplayImage* was part of the initial insertRecipe request. As a result, the *qosbandwidth* Property of *Seq1* is also verified due to the rule triggering (step 5 in FIGURE 83).

At this point, Pattern Engine has reasoned the QoS property of the test orchestration and has produced the result, which is returned as a response back to Pattern Orchestrator.

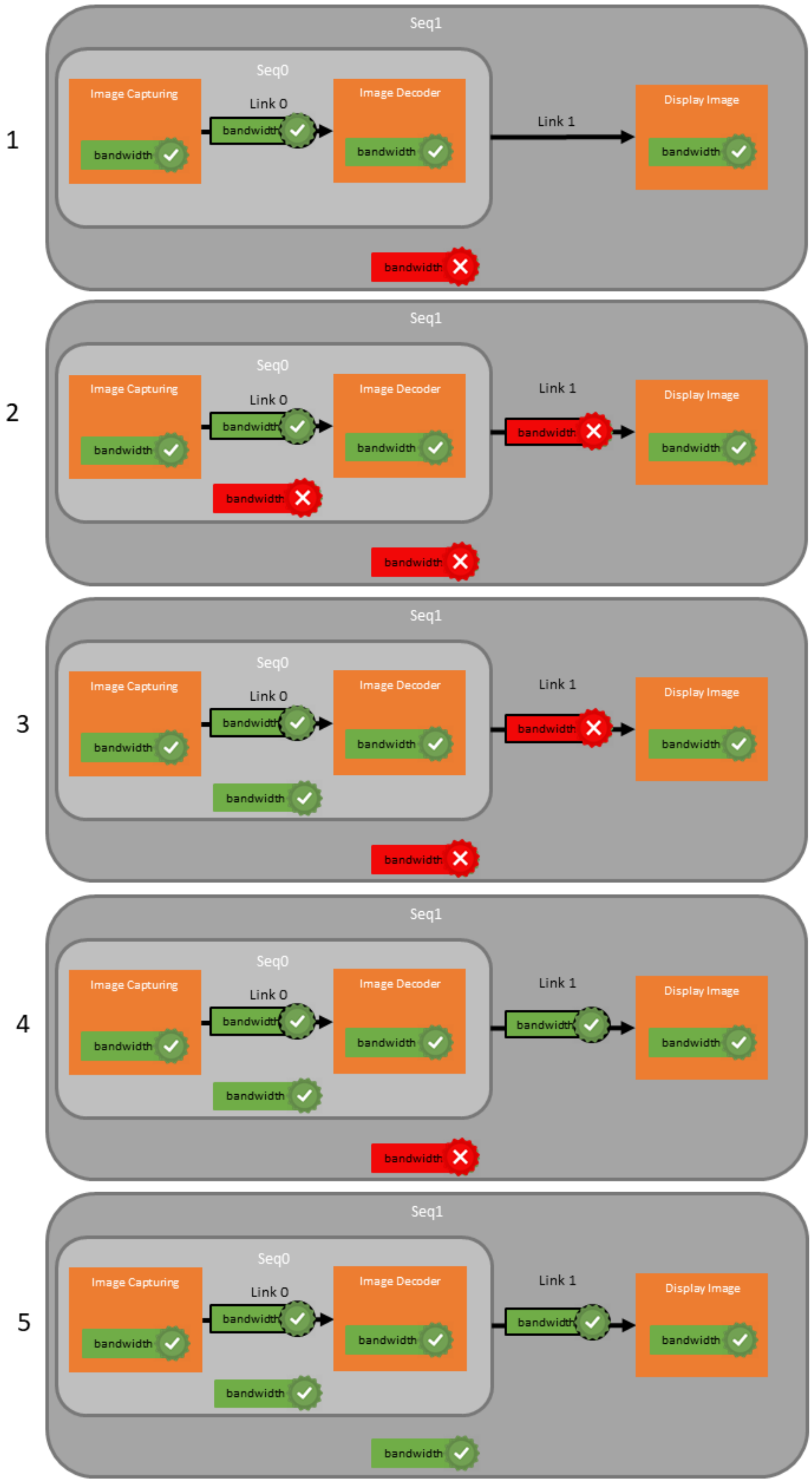


FIGURE 83: SEQUENCE OF DROOLS RULES TRIGGERING

In cycle 2, we focused on the interaction with the Field Pattern Engine. Figure 84 depicts our testing topology for evaluating dependability property and more specifically distributed anomaly vibration monitoring for earthquake detection.

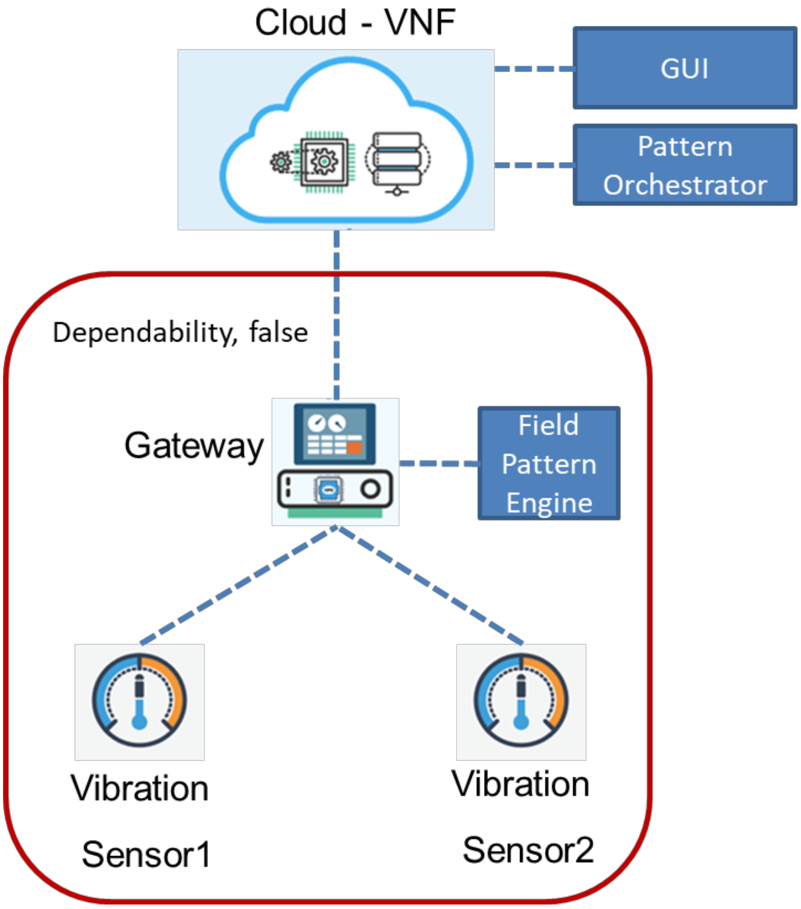


Figure 84: Topology for testing Dependability property

We assume that a Gateway is connected to multiple vibration sensors, which are identical. At any time, all of them are up and running, for redundancy in the monitoring. This redundant topology can be modelled and monitored as the Dependability property, through the appropriately defined pattern rule. Therefore, the infrastructure owner will be able to monitor in real-time the dependability status of his/her deployment, potentially triggering adaptations (e.g., the disabling of one sensor while waiting for a replacement).

In this context, pattern related components are deployed at the field layer, more specifically, a pattern engine runs on the Gateway, able to reason locally about the dependability properties of the anomaly detection setup. Said Pattern Engine will be a lightweight version of the engines deployed in other layers, and will feature appropriate Dependability Pattern Rule to verify that this Dependability property is satisfied and, in case that a sensor fails, will be able to reason and report the failure of the property to the backend. When the sensor is restored, reasoning will verify that the dependability property is restored. Based on the MQTT hierarchy of topics (

Figure 85), Pattern Engine monitors the liveness messages from the sensors (“heartbeats”), via integration with the MQTT broker, leveraging this information to reason about the redundancy of the monitoring system. When one of the sensors fails, the redundancy requirement is violated, and when it is restored (e.g., the sensor comes back online or is restored), the property is satisfied once again.

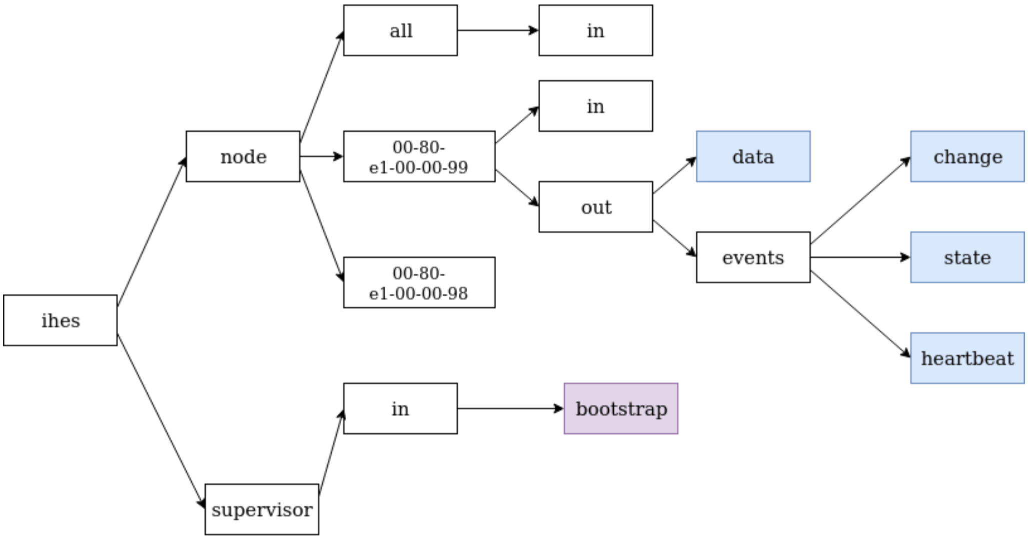
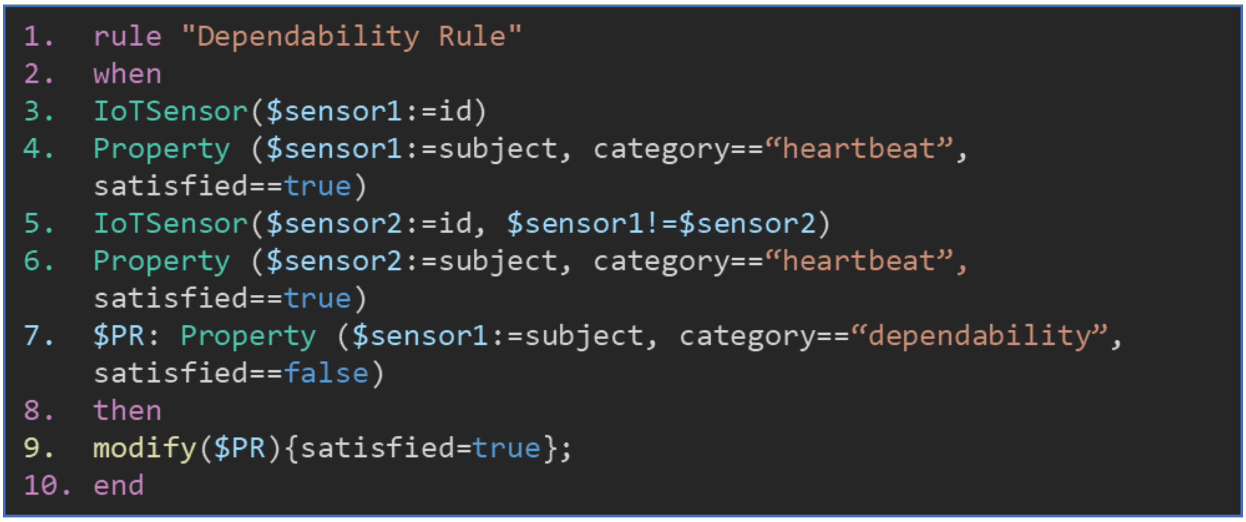


Figure 85: MQTT TOPICS HIERARCHY

Other than the components themselves, an important part is the specification of the associated pattern rule. In terms of the pattern rule to be employed, since dependability is the focus, the Redundancy Pattern will be leveraged, as defined in deliverable D4.8. Adapting said pattern would result in a Drools rule as the one shown in Figure 86.



```
rule "Dependability Rule"
when
IoTSensor($sensor1:=id)
Property ($sensor1:=subject, category==“heartbeat”,
satisfied==true)
IoTSensor($sensor2:=id, $sensor1!=$sensor2)
Property ($sensor2:=subject, category==“heartbeat”,
satisfied==true)
$PR: Property ($sensor1:=subject, category==“dependability”,
satisfied==false)
then
modify($PR){satisfied=true};
end
```

Figure 86: Dependability Drools Rule

The **when** part of the rule specifies:

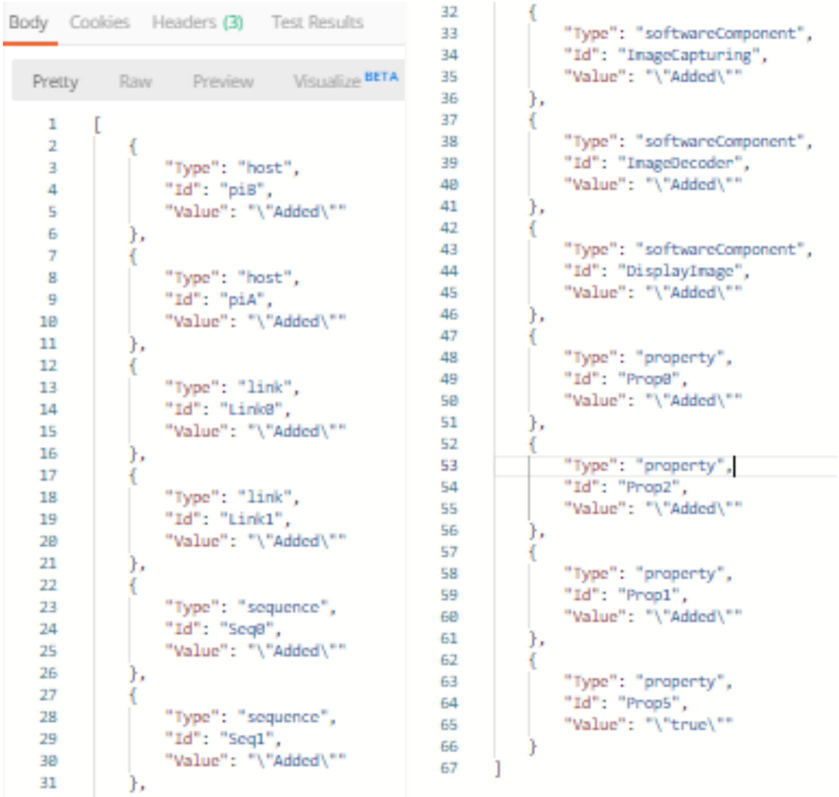
1. the placeholders $sensor1, $sensor2;
2. the extra condition that the sensors are not the same type;
3. the checks that each sensor transmits a heartbeat in a timely manner;
4. the orchestration property that can be guaranteed through the application of the pattern, i.e., the dependability property in this case ($pr).

The **then** part verifies that the orchestration property holds since every essential component is included in the when part (satisfied=true).

3.4.2.4 PERFORMANCE TEST AND KPI VALIDATION

According to the methodology described in subsection 3.4.2.3 in cycle 1, it is expected:

1. the described in the insertRecipe components, communications and properties of the test orchestration to be communicated to the SDN Controller Pattern Engine
2. the qosbandwidth Property of the test orchestration (minimum bandwidth) to be verified according to the response that reaches firstly the Pattern Orchestrator and secondly the Postman client
3. the pre-inserted Drools Rules of the SDN Controller Pattern Engine to be triggered



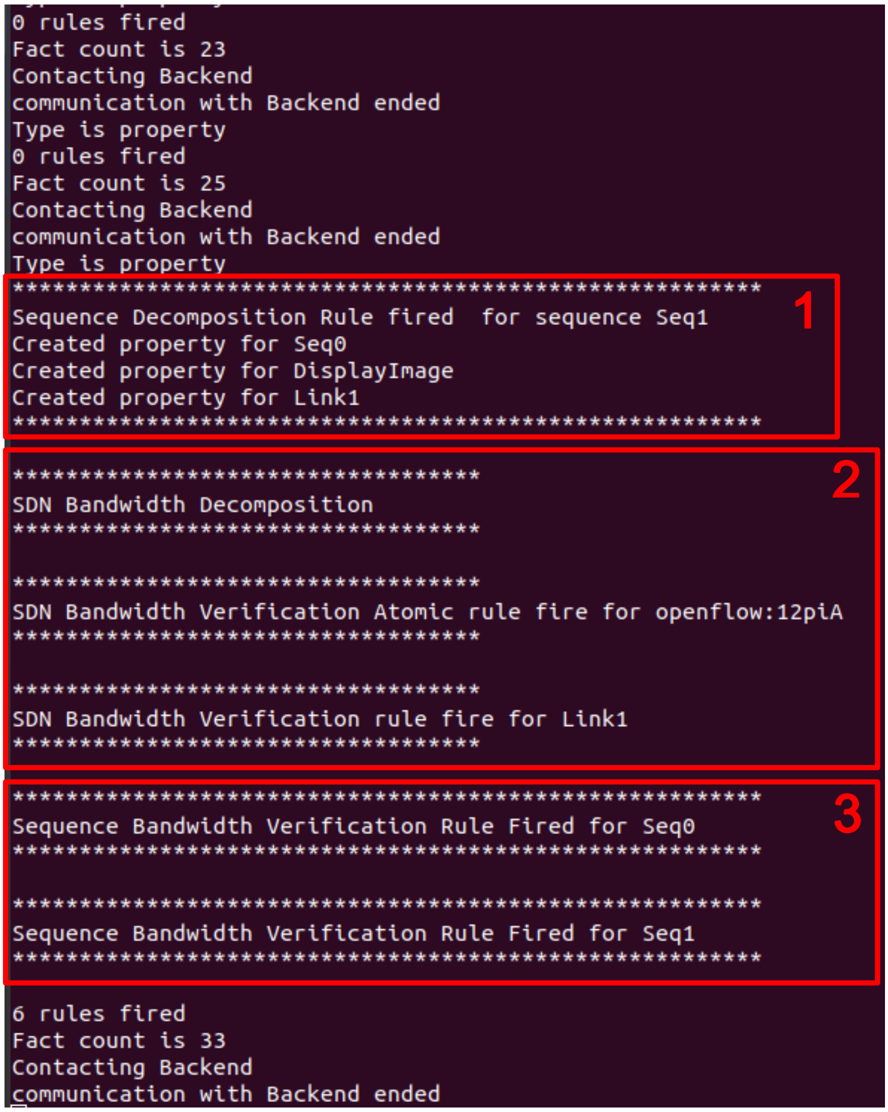
```
[
    {
        "Type": "host",
        "Id": "piB",
        "Value": "\"Added\""
    },
    {
        "Type": "host",
        "Id": "piA",
        "Value": "\"Added\""
    },
    {
        "Type": "link",
        "Id": "Link0",
        "Value": "\"Added\""
    },
    {
        "Type": "link",
        "Id": "Link1",
        "Value": "\"Added\""
    },
    {
        "Type": "sequence",
        "Id": "Seq0",
        "Value": "\"Added\""
    },
    {
        "Type": "sequence",
        "Id": "Seq1",
        "Value": "\"Added\""
    },
    {
        "Type": "softwareComponent",
        "Id": "ImageCapturing",
        "Value": "\"Added\""
    },
    {
        "Type": "softwareComponent",
        "Id": "ImageDecoder",
        "Value": "\"Added\""
    },
    {
        "Type": "softwareComponent",
        "Id": "DisplayImage",
        "Value": "\"Added\""
    },
    {
        "Type": "property",
        "Id": "Prop0",
        "Value": "\"Added\""
    },
    {
        "Type": "property",
        "Id": "Prop2",
        "Value": "\"Added\""
    },
    {
        "Type": "property",
        "Id": "Prop1",
        "Value": "\"Added\""
    },
    {
        "Type": "property",
        "Id": "Prop5",
        "Value": "\"true\""
    }
]
```

FIGURE 87: PATTERN ORCHESTRATOR RESPONSE TO POSTMAN CLIENT

Regarding the first outcome, all the test orchestration components are successfully communicated to the SDN Controller Pattern Engine and this is confirmed by the response that reaches the Postman client. As we can see in FIGURE 87, all hosts, links, softwarecomponents, sequences and properties are “Added” to the working memory of the SDN Controller Pattern Engine. This response is created by the SDN Controller Pattern Engine and is sent to the Pattern Orchestrator for each Drools fact. Pattern Orchestrator assembles all of them and sends them as a response to the Postman client.

Moreover, the same Figure shows the verification of the qosbandwidth Property of the whole test orchestration. The id of the qosbandwidth Property in the request was *Prop5*. The “true” that is depicted at the lowest part of the Figure for the Property with id *Prop5,* verifies that this Property holds.

Finally, the triggering of the Drools Rules is shown in FIGURE 88. As we can see, the *Sequence – Decomposition* rule for *Seq1* is the first rule that is triggered. The output of the rule is shown in the red rectangle labeled with the number 1. After that, the Network Drools Rules run for the verification of the *qosbandwidth* Property of *Link1*. Their output is included in the red rectangle labeled with the number 2. The last rule that is triggered is the *Sequence Bandwidth Verification* rule, which runs twice for *Seq0* and *Seq1* and verifies their properties. Once again, the output of this rule is shown in the red rectangle labeled with the number 2.



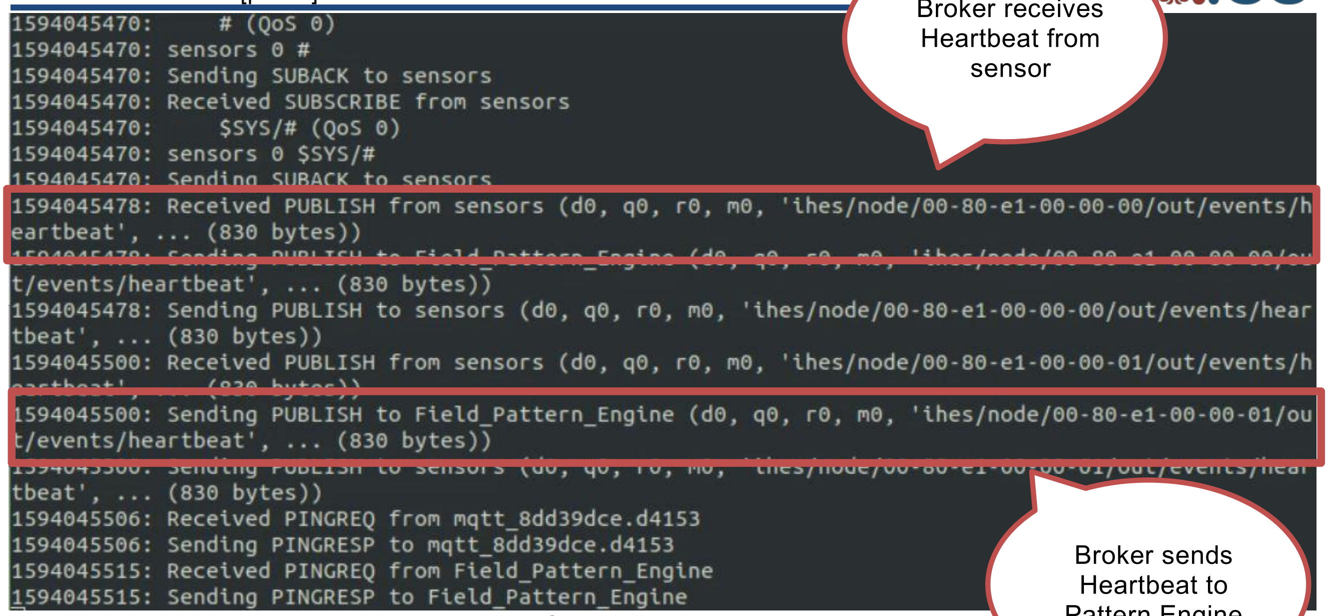
```
0 rules fired
Fact count is 23
Contacting Backend
communication with Backend ended
Type is property
0 rules fired
Fact count is 25
Contacting Backend
communication with Backend ended
Type is property
********************************************************
Sequence Decomposition Rule fired  for sequence Seq1
Created property for Seq0
Created property for DisplayImage
Created property for Link1
********************************************************

***********************************
SDN Bandwidth Decomposition
***********************************

***********************************
SDN Bandwidth Verification Atomic rule fire for openflow:12piA
***********************************

***********************************
SDN Bandwidth Verification rule fire for Link1
***********************************

********************************************************
Sequence Bandwidth Verification Rule Fired for Seq0
********************************************************

********************************************************
Sequence Bandwidth Verification Rule Fired for Seq1
********************************************************

6 rules fired
Fact count is 33
Contacting Backend
communication with Backend ended
```

FIGURE 88: SCREENSHOT WITH THE OUTPUT OF THE TRIGGERED DROOLS RULES

According to the methodology described in subsection 3.4.2.3 in cycle 2, it is expected that:

1. each of the sensors to communicate successfully their heartbeats to the Pattern Engine through the MQTT broker.
2. the Dependability Property of the test orchestration to be verified
3. the pre-inserted Drools Rules of the Field Pattern Engine to be triggered

Regarding the first outcome, in Figure 89 heartbeats from sensors are communicated successfully to the Field Pattern Engine. Based on the Pattern Rule, Figure 90 depicts the triggering of the rule which is the third outcome that also implies the second outcome.

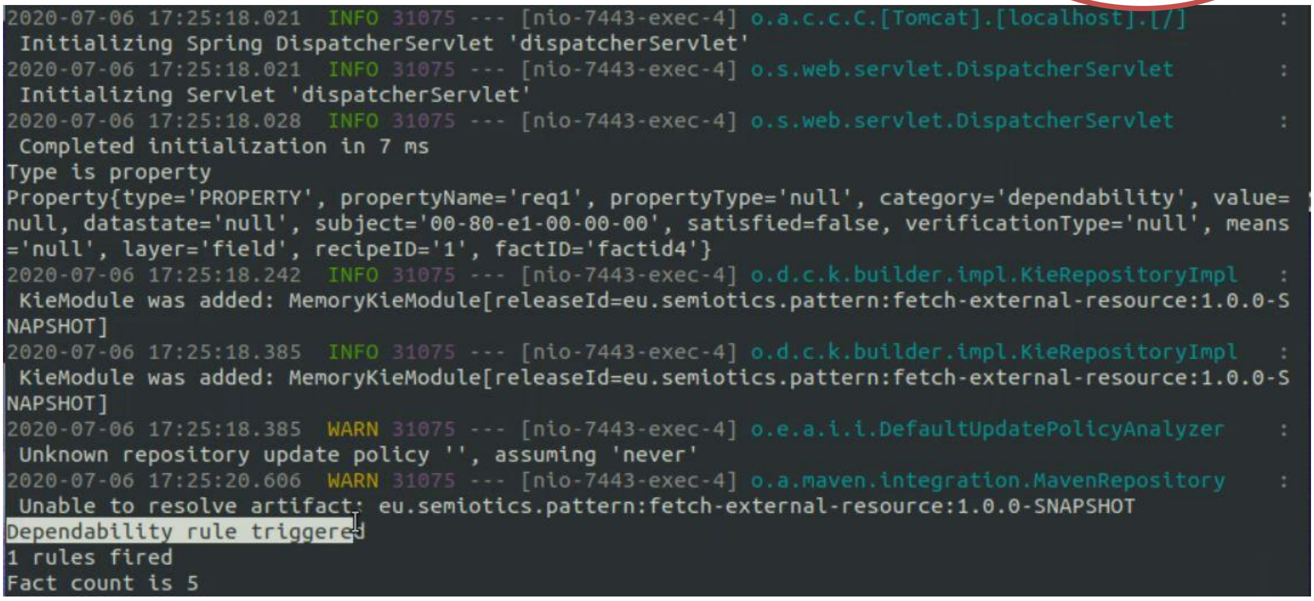


```
1594045470:     # (QoS 0)
1594045470: sensors 0 #
1594045470: Sending SUBACK to sensors
1594045470: Received SUBSCRIBE from sensors
1594045470:     $SYS/# (QoS 0)
1594045470: sensors 0 $SYS/#
1594045470: Sending SUBACK to sensors
1594045478: Received PUBLISH from sensors (d0, q0, r0, m0, 'ihes/node/00-80-e1-00-00-00/out/events/h
eartbeat', ... (830 bytes))
1594045478: Sending PUBLISH to Field_Pattern_Engine (d0, q0, r0, m0, 'ihes/node/00-80-e1-00-00-00/ou
t/events/heartbeat', ... (830 bytes))
1594045478: Sending PUBLISH to sensors (d0, q0, r0, m0, 'ihes/node/00-80-e1-00-00-00/out/events/hear
tbeat', ... (830 bytes))
1594045500: Received PUBLISH from sensors (d0, q0, r0, m0, 'ihes/node/00-80-e1-00-00-01/out/events/h
eartbeat', ... (830 bytes))
1594045500: Sending PUBLISH to Field_Pattern_Engine (d0, q0, r0, m0, 'ihes/node/00-80-e1-00-00-01/ou
t/events/heartbeat', ... (830 bytes))
1594045500: Sending PUBLISH to sensors (d0, q0, r0, m0, 'ihes/node/00-80-e1-00-00-01/out/events/hear
tbeat', ... (830 bytes))
1594045506: Received PINGREQ from mqtt_8dd39dce.d4153
1594045506: Sending PINGRESP to mqtt_8dd39dce.d4153
1594045515: Received PINGREQ from Field_Pattern_Engine
1594045515: Sending PINGRESP to Field_Pattern_Engine
```

Broker receives Heartbeat from sensor

Broker sends Heartbeat to Pattern Engine

Figure 89: MQTT Brocker relaying heartbeats



```
2020-07-06 17:25:18.021  INFO 31075 --- [nio-7443-exec-4] o.a.c.c.C.[Tomcat].[localhost].[/]       :
 Initializing Spring DispatcherServlet 'dispatcherServlet'
2020-07-06 17:25:18.021  INFO 31075 --- [nio-7443-exec-4] o.s.web.servlet.DispatcherServlet        :
 Initializing Servlet 'dispatcherServlet'
2020-07-06 17:25:18.028  INFO 31075 --- [nio-7443-exec-4] o.s.web.servlet.DispatcherServlet        :
 Completed initialization in 7 ms
Type is property
Property{type='PROPERTY', propertyName='req1', propertyType='null', category='dependability', value=
null, datastate='null', subject='00-80-e1-00-00-00', satisfied=false, verificationType='null', means
='null', layer='field', recipeID='1', factID='factid4'}
2020-07-06 17:25:18.242  INFO 31075 --- [nio-7443-exec-4] o.d.c.k.builder.impl.KieRepositoryImpl   :
 KieModule was added: MemoryKieModule[releaseId=eu.semiotics.pattern:fetch-external-resource:1.0.0-S
NAPSHOT]
2020-07-06 17:25:18.385  INFO 31075 --- [nio-7443-exec-4] o.d.c.k.builder.impl.KieRepositoryImpl   :
 KieModule was added: MemoryKieModule[releaseId=eu.semiotics.pattern:fetch-external-resource:1.0.0-S
NAPSHOT]
2020-07-06 17:25:18.385  WARN 31075 --- [nio-7443-exec-4] o.e.a.i.i.DefaultUpdatePolicyAnalyzer    :
 Unknown repository update policy '', assuming 'never'
2020-07-06 17:25:20.606  WARN 31075 --- [nio-7443-exec-4] o.a.maven.integration.MavenRepository    :
 Unable to resolve artifact: eu.semiotics.pattern:fetch-external-resource:1.0.0-SNAPSHOT
Dependability rule triggered
1 rules fired
Fact count is 5
```

Figure 90: FIELD PATTERN ENGINE (SUBSCRIBED TO BROKER; RECEIVING UPDATES & REASONING ON DEPENDABILITY PROPERTY)

The described implementation above contributes to the fulfillment of R.GP.4, R.UC1.1, R.UC1.3, R.UC3.7, R.UC3.16, R.UC3.18 of project requirements as well as KPI-1.1, KPI-1.2, KPI-6.1.

3.4.3 BACKEND ORCHESTRATION

3.4.3.1 TEST BED - KUBERNETES

In order to simplify the need for usage of different components in the backend, an integrated testbed for backend in form of Kubernetes cluster was deployed on BlueSoft's premises. Currently, backend components are being deployed one by one and then onboarded on the Jenkins pipelines to ensure that each deployment is correct and free of human error. Thanks to the set-up partners can access APIs of backend components without the need for replicating the component infrastructure on their environments. Kubernetes requires deployment on multiple nodes which can be divided into two categories: master nodes and normal nodes. The main role of the master is control of the whole cluster, all the management tools are deployed there, the master should not host any non-essential applications. The rest of the nodes are typical nodes for different application deployments. Kubernetes handles app to node associations, but the user is still responsible for controlling the available resources (CPU, memory, and storage). Our cluster holds 3 nodes (1 master and 2 app nodes) and can be extended in the future if needed. The specification of the nodes can be found in the table below.

Table 14: Kubernetes cluster technical details

Name	Master	Node 1	Node 2
Total space	1000 GB	230 GB	200 GB
CPU	Genuine Intel, 4 cores, 2.5 GHz, Cache 16 MB	Genuine Intel, 4 cores, 2.500 GHz, Cache 16 MB	Genuine Intel, 4 cores, 2.5 GHz, Cache 16 MB
Operational System	CentOS 7 Linux (Core)	CentOS 7 Linux (Core)	CentOS 7 Linux (Core)
Virtual Machine	Yes	Yes	Yes
RAM	8 GB	8 GB	8 GB

Nodes are using virtual machines deployed on a dedicated physical server. Specification of the server can be found in table below, currently, the not whole server is utilized but it should change in upcoming months.

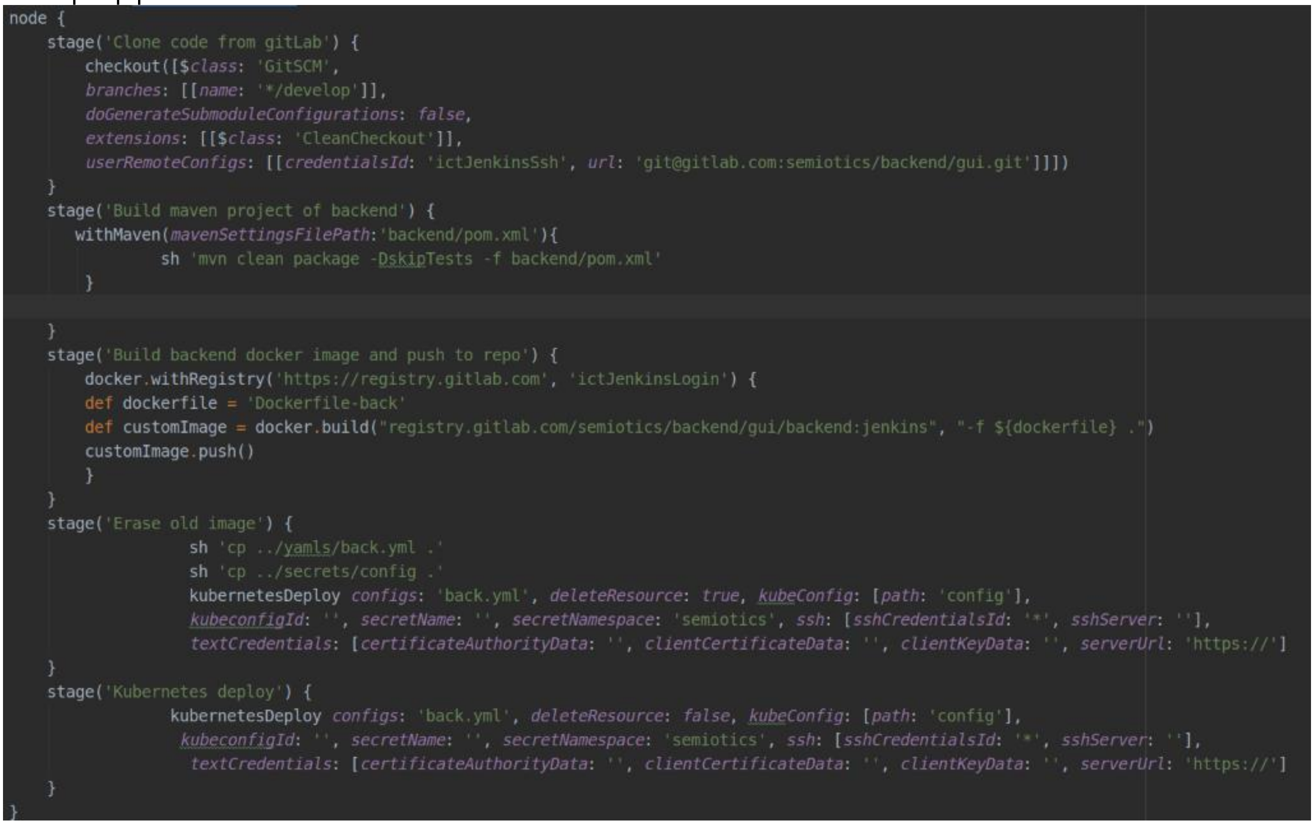
Table 15: Server technical details

Brand and Model	Dell R730
CPU	Intel Xeon 2x E5-2680 v3 2.7 GHz, Cache 20MB
RAM	64 GB
Total space	8 TB

The additional virtual machine was created for Jenkins's deployment. Jenkins takes care of deployment for backend components running pipelines which executes the following steps (FIGURE 91):

- build an application from Gitlab code
- build sidecars for application (if needed)
- run automatic tests if the exists
- prepare docker image from build
- save the image in the registry
- deploy image in Kubernetes

Example pipeline code can be found on the screens below.



```
node {
    stage('Clone code from gitLab') {
        checkout([$class: 'GitSCM',
        branches: [[name: '*/develop']],
        doGenerateSubmoduleConfigurations: false,
        extensions: [[$class: 'CleanCheckout']],
        userRemoteConfigs: [[credentialsId: 'ictJenkinsSsh', url: 'git@gitlab.com:semiotics/backend/gui.git']]])
    }
    stage('Build maven project of backend') {
       withMaven(mavenSettingsFilePath:'backend/pom.xml'){
                sh 'mvn clean package -DskipTests -f backend/pom.xml'
        }

    }
    stage('Build backend docker image and push to repo') {
        docker.withRegistry('https://registry.gitlab.com', 'ictJenkinsLogin') {
        def dockerfile = 'Dockerfile-back'
        def customImage = docker.build("registry.gitlab.com/semiotics/backend/gui/backend:jenkins", "-f ${dockerfile} .")
        customImage.push()
        }
    }
    stage('Erase old image') {
                    sh 'cp ../yamls/back.yml .'
                    sh 'cp ../secrets/config .'
                    kubernetesDeploy configs: 'back.yml', deleteResource: true, kubeConfig: [path: 'config'],
                    kubeconfigId: '', secretName: '', secretNamespace: 'semiotics', ssh: [sshCredentialsId: '*', sshServer: ''],
                    textCredentials: [certificateAuthorityData: '', clientCertificateData: '', clientKeyData: '', serverUrl: 'https://']
    }
    stage('Kubernetes deploy') {
                  kubernetesDeploy configs: 'back.yml', deleteResource: false, kubeConfig: [path: 'config'],
                   kubeconfigId: '', secretName: '', secretNamespace: 'semiotics', ssh: [sshCredentialsId: '*', sshServer: ''],
                    textCredentials: [certificateAuthorityData: '', clientCertificateData: '', clientKeyData: '', serverUrl: 'https://']
    }
}
```

FIGURE 91: AN EXAMPLE OF JENKINS PIPELINE

Such an approach guarantees less deployment time and higher quality deploys with fewer errors thanks to automatic testing. Bellow, there are details for Jenkins's virtual machine.

Table 16: Jenkins machine technical details

Name	Jenkins
Total space	232 GB
CPU	Genuine Intel, 4 cores, 2500 MHz, Cache 16 MB
OS	CentOS 7 Linux (Core)
Virtual Machine	Yes
RAM	6 GB

The deployment is shown in the diagram below. Both Kubernetes and Jenkins are in the same internal BlueSoft network with the firewall in front of it. The firewall opens only required ports and filters incoming traffic. Only white-listed IP addresses can access the instances.

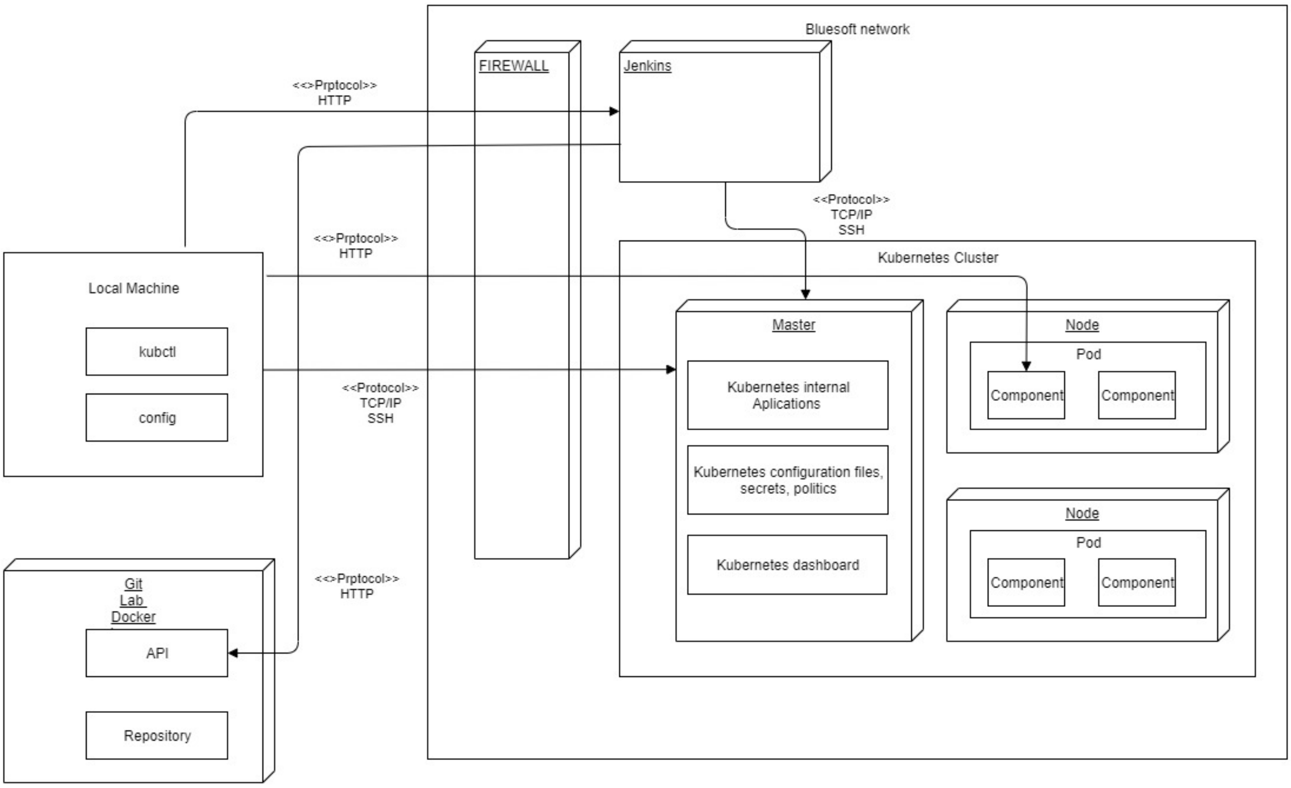


FIGURE 92: DEPLOYMENT DIAGRAM

3.4.3.1.1 ADMINISTRATION

Multiple tools allow for interaction with the Kubernetes cluster. Here are the 3 most popular not requiring additional products installations:

- kubectl - CLI client which allows connecting to Kubernetes, components are created above the API
- Kubernetes API
- Kubernetes dashboard - additional component connecting to Kubernetes API giving a visual representation of the cluster.

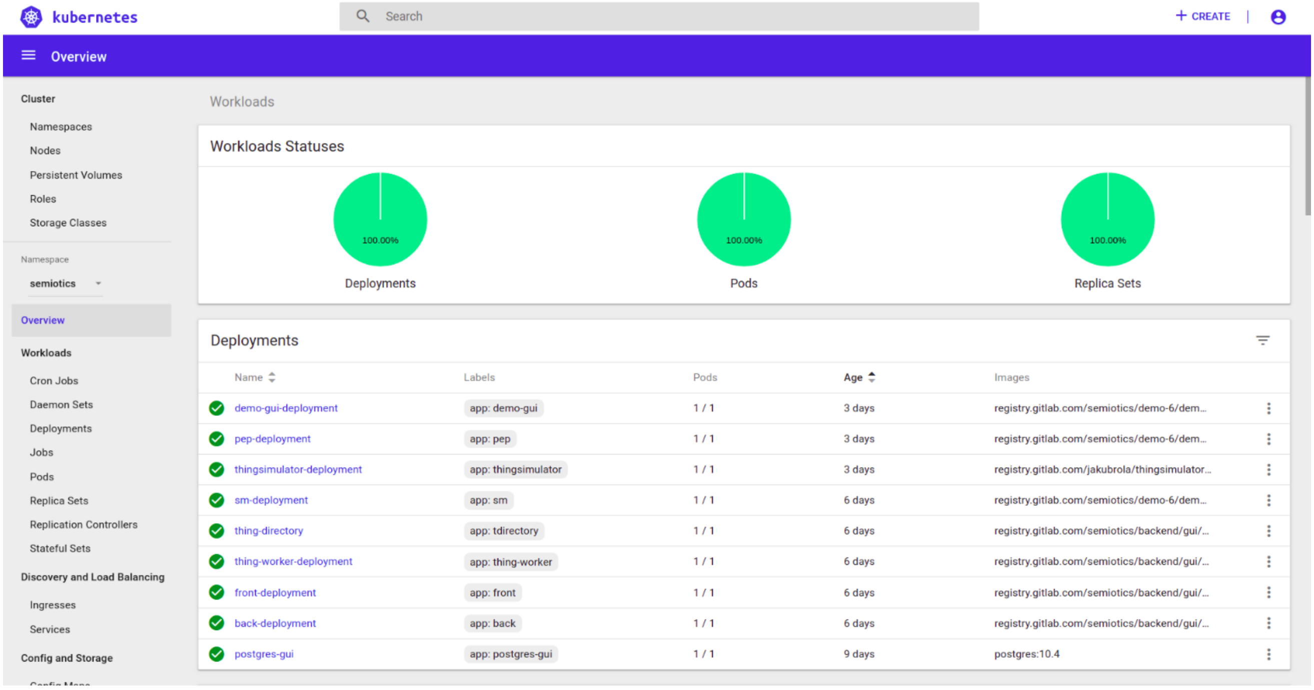
kubectl

The Kubernetes command-line tool, allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs[17]. It also allows remote access to the cluster, with a personal key.

Kubernetes dashboard

[17] https://kubernetes.io/docs/reference/kubectl/overview/

The dashboard is a web-based Kubernetes administration console. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster resources[18]. The dashboard is used to monitor the state of the cluster e.g. state of applications, cluster resources, etc. It also provides information about any errors that occurred. In the figure below the view of the Kubernetes dashboard is presented.



Deployments

Name	Labels	Pods	Age	Images
demo-gui-deployment	app: demo-gui	1 / 1	3 days	registry.gitlab.com/semiotics/demo-6/dem...
pep-deployment	app: pep	1 / 1	3 days	registry.gitlab.com/semiotics/demo-6/dem...
thingsimulator-deployment	app: thingsimulator	1 / 1	3 days	registry.gitlab.com/jakubrola/thingsimulator...
sm-deployment	app: sm	1 / 1	6 days	registry.gitlab.com/semiotics/demo-6/dem...
thing-directory	app: tdirectory	1 / 1	6 days	registry.gitlab.com/semiotics/backend/gui/...
thing-worker-deployment	app: thing-worker	1 / 1	6 days	registry.gitlab.com/semiotics/backend/gui/...
front-deployment	app: front	1 / 1	6 days	registry.gitlab.com/semiotics/backend/gui/...
back-deployment	app: back	1 / 1	6 days	registry.gitlab.com/semiotics/backend/gui/...
postgres-gui	app: postgres-gui	1 / 1	9 days	postgres:10.4

FIGURE 93 STATE OF DEPLOYED APPLICATIONS PRESENTED WITH KUBERNETES DASHBOARD

3.4.3.1.2 SECURITY

The security of the cluster has been taken into account while creating the testbed. Currently, security is implemented on multiple layers to protect the testbed. Access is blocked for different users depending on the layer.

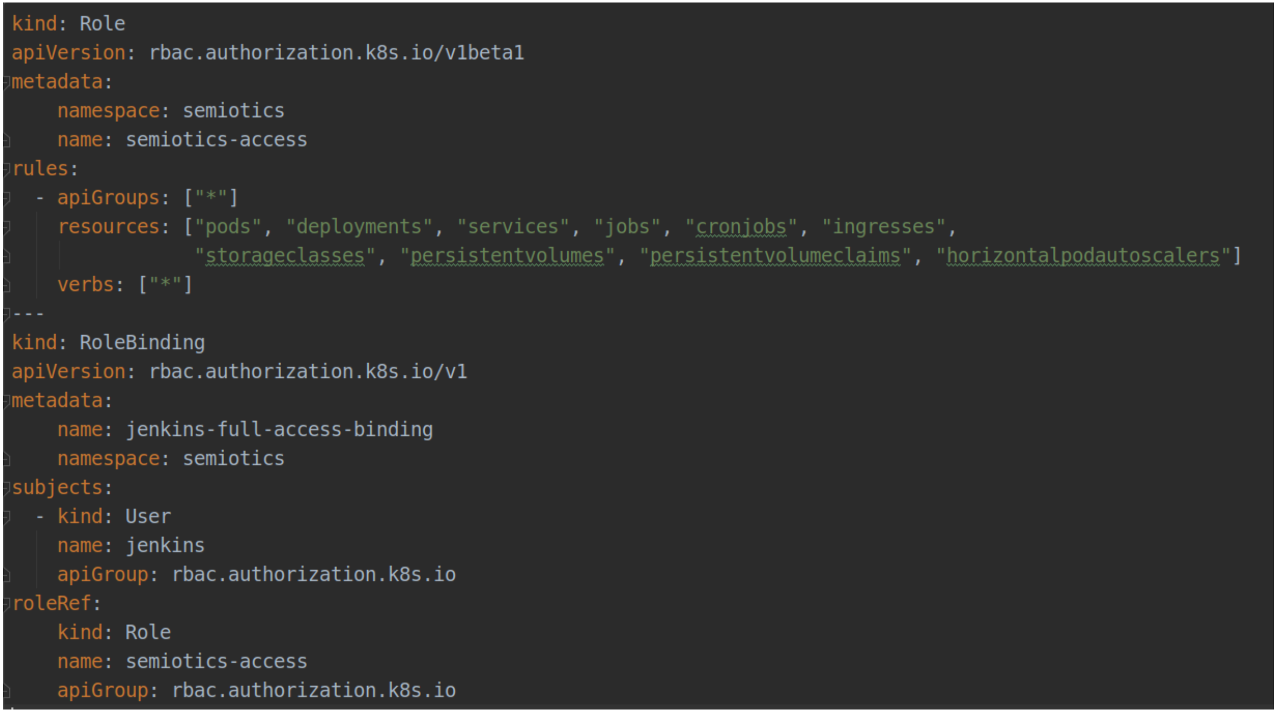
Cluster node access

Nobody from outside of BlueSoft can access cluster node using ssh, and management is currently done by BlueSoft due to our experience with the technology. Each user uses a personal key to log in to the machine so every action can be traced.

Cluster management

To ensure proper access to cluster resources the namespace SEMIoTICS has been created. Cluster management can be done using API or kubectl tool. The installation uses the RBAC model of privileges limited only to SEMIoTICS namespace (FIGURE 94). Access to cluster APIs is hidden behind the aforementioned firewall and granted only to whitelisted IPs or cider blocks of the partner. Additionally, only the required ports are opened, if the port is not used it is blocked on the firewall.

[18] https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/



```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
    namespace: semiotics
    name: semiotics-access
rules:
  - apiGroups: ["*"]
    resources: ["pods", "deployments", "services", "jobs", "cronjobs", "ingresses",
                "storageclasses", "persistentvolumes", "persistentvolumeclaims", "horizontalpodautoscalers"]
    verbs: ["*"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
    name: jenkins-full-access-binding
    namespace: semiotics
subjects:
  - kind: User
    name: jenkins
    apiGroup: rbac.authorization.k8s.io
roleRef:
    kind: Role
    name: semiotics-access
    apiGroup: rbac.authorization.k8s.io
```

FIGURE 94:SCRIPT FOR ADDING PRIVILEGES IN SEMIOTICS NAMESPACE TO USER JENKINS

Application remote access or inner application to application access

An application that is dedicated to the user containing a graphical user interface or API. In both cases, the security manager will be used. API access is done using PEP and Security Manager (these components are extensively described in deliverable 4.5).

- Security Manager – SEMIoTICS component responsible for authentication decisions and necessary security checks at the backend layer. It stores and decides on security policies across all SEMIoTICS components.
- PEP - Policy Enforcement Point. The sidecar component is responsible for intercepting HTTP requests. It is a sidecar application that is deployed as a standalone app next to the primary application and also as a second container in a pod in Backend Orchestrator.

3.4.3.2 COMPONENT ARCHITECTURE

Backend orchestrator (Jenkins + Kubernetes) in SEMIoTICS is responsible for orchestration and management of components in the backend layer. The diagram below shows Backend Orchestrator and all components currently deployed by it.

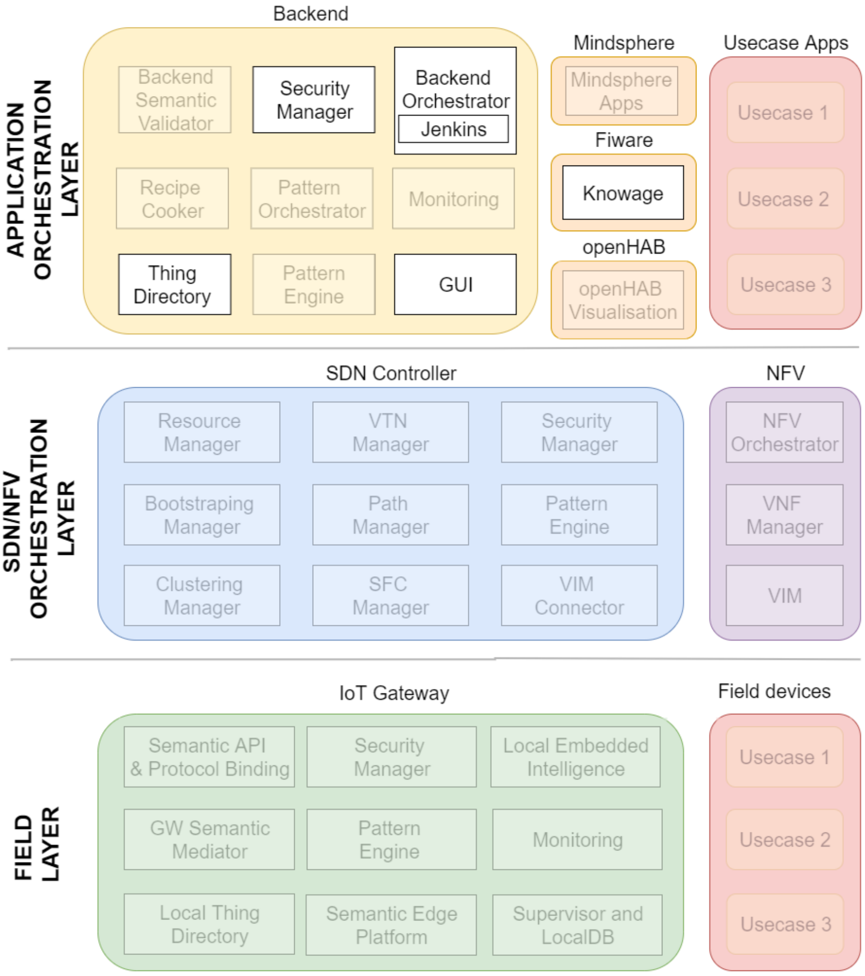
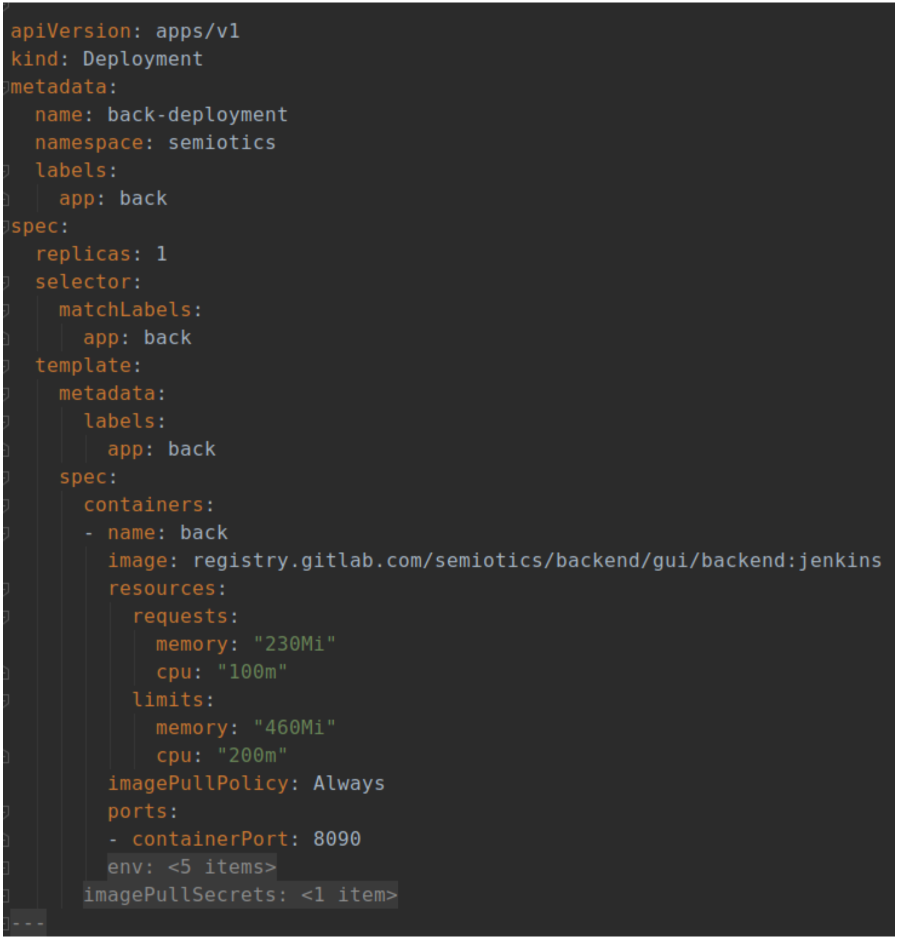


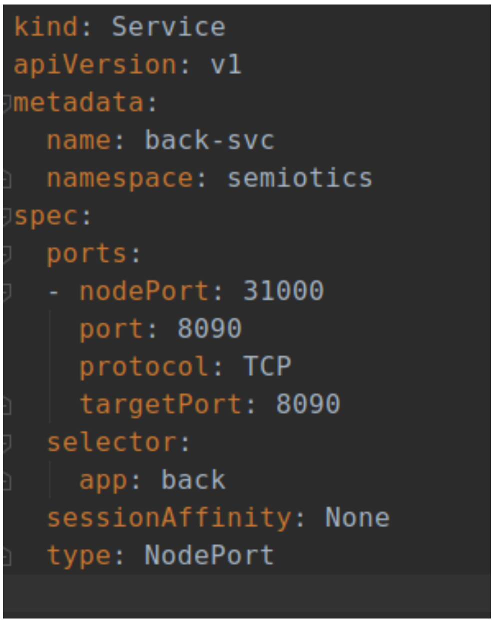
FIGURE 95 COMPONENTS RELATED TO THE BACKEND ORCHESTRATION

Kubernetes deployments are done using YAML files in the specific format required by Kubernetes. Deployment specifies application additionally if components need to expose API additional resources are required for inner API or API which can be accessed outside of Kubernetes. FIGURE 96 and FIGURE 97 depict an example of Kubernetes Deployment and Service resources YAML files. The deployment file specifies the namespace in which the application is going to be deployed, as well as, how many replicas of the pod should be running, which image to use to build container and how much resources allocated to it. The service file specifies the way of how to expose a pod outside of the cluster.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: back-deployment
  namespace: semiotics
  labels:
    app: back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: back
  template:
    metadata:
      labels:
        app: back
    spec:
      containers:
      - name: back
        image: registry.gitlab.com/semiotics/backend/gui/backend:jenkins
        resources:
          requests:
            memory: "230Mi"
            cpu: "100m"
          limits:
            memory: "460Mi"
            cpu: "200m"
        imagePullPolicy: Always
        ports:
        - containerPort: 8090
        env: <5 items>
      imagePullSecrets: <1 item>
---
```

FIGURE 96:THE GUI:BACKEND DEPLOYMENT FILE



```yaml
kind: Service
apiVersion: v1
metadata:
  name: back-svc
  namespace: semiotics
spec:
  ports:
  - nodePort: 31000
    port: 8090
    protocol: TCP
    targetPort: 8090
  selector:
    app: back
  sessionAffinity: None
  type: NodePort
```

FIGURE 97: THE GUI – BACKEND SERVICE FILE

Service resource file specifies the port and the type of service that exposes the pod with the application. In the example above, the application is expected to run on port 31000. After successful deployment, it can be checked by performing an HTTP request to the specified address (FIGURE 98). If the deployment of pod fails, the cause of failure will be shown in pod's logs (FIGURE 101).

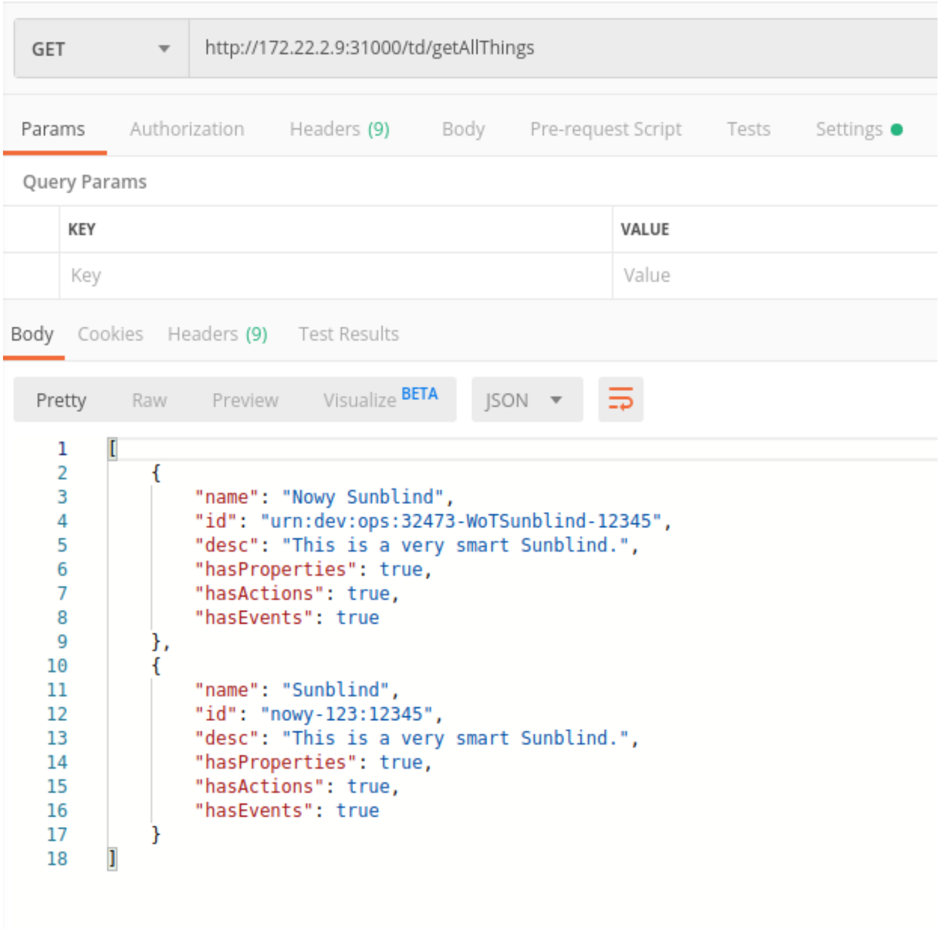
3.4.3.3 TESTS

The functional tests of Backend Orchestrator and Jenkins will be focused entirely on the deployment of components. All tests presume that the cluster is working correctly. As test scenario deployment will be made, which will be further verified by API calls. TABLE 17below holds a list of all the test scenarios.

TABLE 17 FUNCTIONAL TESTS SCENARIOS FOR KUBERNETES AND JENKINS

GUI
/deployment/semiotics/back-deployment, /deployment/semiotics/front-deployment
✓ After a successful deployment, the application is running on the specified address (if it exposes an API) It has to be checked manually using a tool like POSTMAN or even web browser. If the application doesn't have an API, the information about success can be seen in the logs.
✓ After a successful deployment, the response contains information about the deployed application.
✓ After a failed deployment, the response of the HTTP request should be 'Could not get any response'. Because of the variety of factors that have an impact on the success of the deployment, it's impossible to predict the cause of the failure.
Security Manager
/deployment/semiotics/security-manager-deployment
✓ After a successful deployment, the application is running on the specified address (if it exposes an API) It has to be checked manually using a tool like POSTMAN or even web browser. If the application doesn't have an API, the information about success can be seen in the logs.
✓ After a successful deployment, the response contains information about the deployed application.
✓ After a failed deployment, the response of the HTTP request should be 'Could not get any response'. Because of the variety of factors that have an impact on the success of the deployment, it's impossible to predict the cause of the failure.
Thing Directory
/deployment/semiotics/thing-directory-deployment
✓ After a successful deployment, the application is running on the specified address (if it exposes an API) It has to be checked manually using a tool like POSTMAN or even web browser. If the application doesn't have an API, the information about success can be seen in the logs.
✓ After a successful deployment, the response contains information about the deployed application.
✓ After a failed deployment, the response of the HTTP request should be 'Could not get any response'. Because of the variety of factors that have an impact on the success of the deployment, it's impossible to predict the cause of the failure.
KNOWAGE
/deployment/semiotics/knowage-deployment
✓ After a successful deployment, the application is running on the specified address (if it exposes an API) It has to be checked manually using a tool like POSTMAN or even web browser. If the application doesn't have an API, the information about success can be seen in the logs.
✓ After a successful deployment, the response contains information about the deployed application.
✓ After a failed deployment, the response of the HTTP request should be 'Could not get any response'. Because of the variety of factors that have an impact on the success of the deployment, it's impossible to predict the cause of the failure.

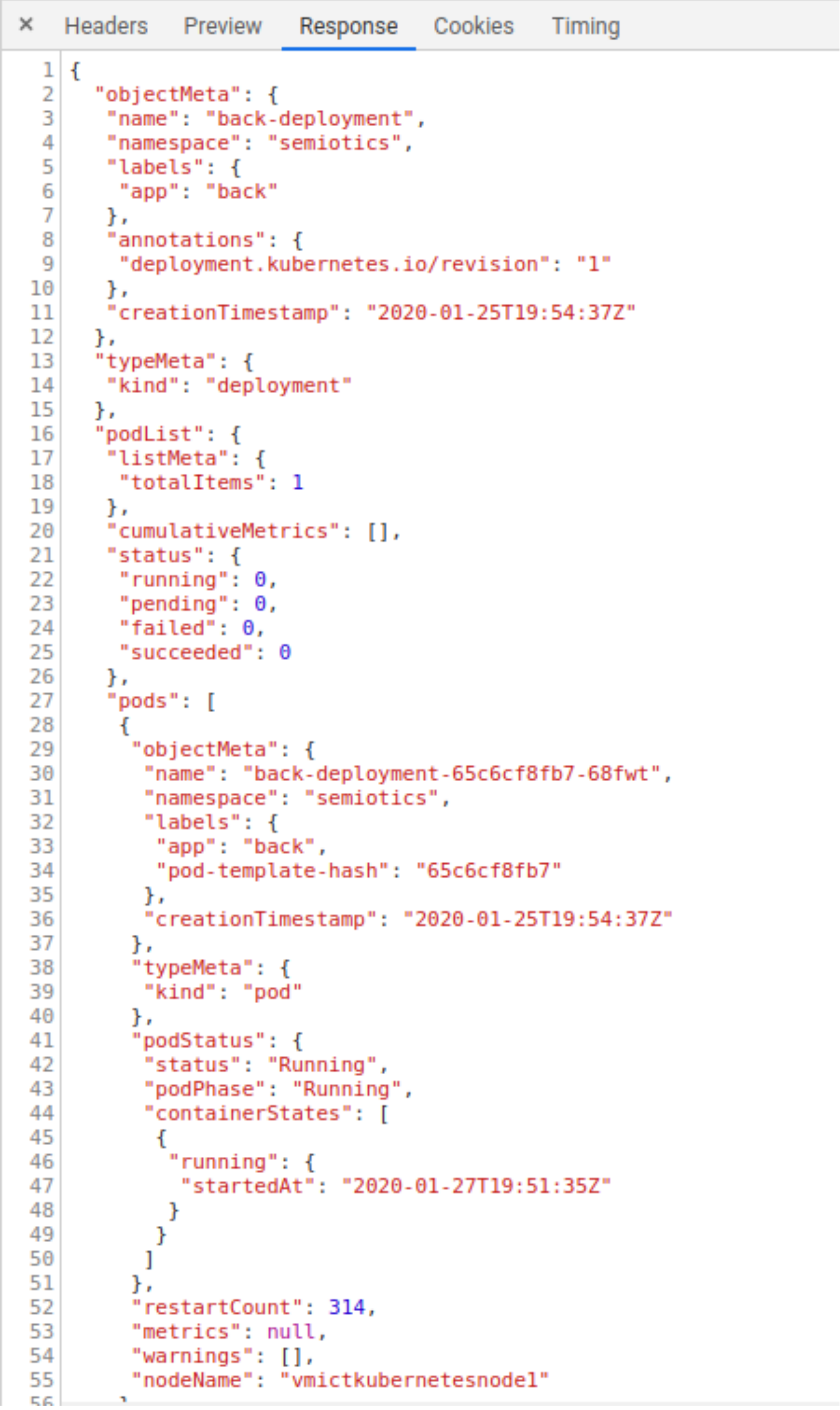
Thing Simulator
/deployment/semiotics/thing-simulator-deployment
✓ After a successful deployment, the application is running on the specified address (if it exposes an API) It has to be checked manually using a tool like POSTMAN or even web browser. If the application doesn't have an API, the information about success can be seen in the logs.
✓ After a successful deployment, the response contains information about the deployed application.
✓ After a failed deployment, the response of the HTTP request should be 'Could not get any response'. Because of the variety of factors that have an impact on the success of the deployment, it's impossible to predict the cause of the failure.
Thing Worker
/deployment/semiotics/thing-worker-deployment
✓ After a successful deployment, the application is running on the specified address (if it exposes an API) It has to be checked manually using a tool like POSTMAN or even web browser. If the application doesn't have an API, the information about success can be seen in the logs.
✓ After a successful deployment, the response contains information about the deployed application.
✓ After a failed deployment, the response of the HTTP request should be 'Could not get any response'. Because of the variety of factors that have an impact on the success of the deployment, it's impossible to predict the cause of the failure.



GET http://172.22.2.9:31000/td/getAllThings

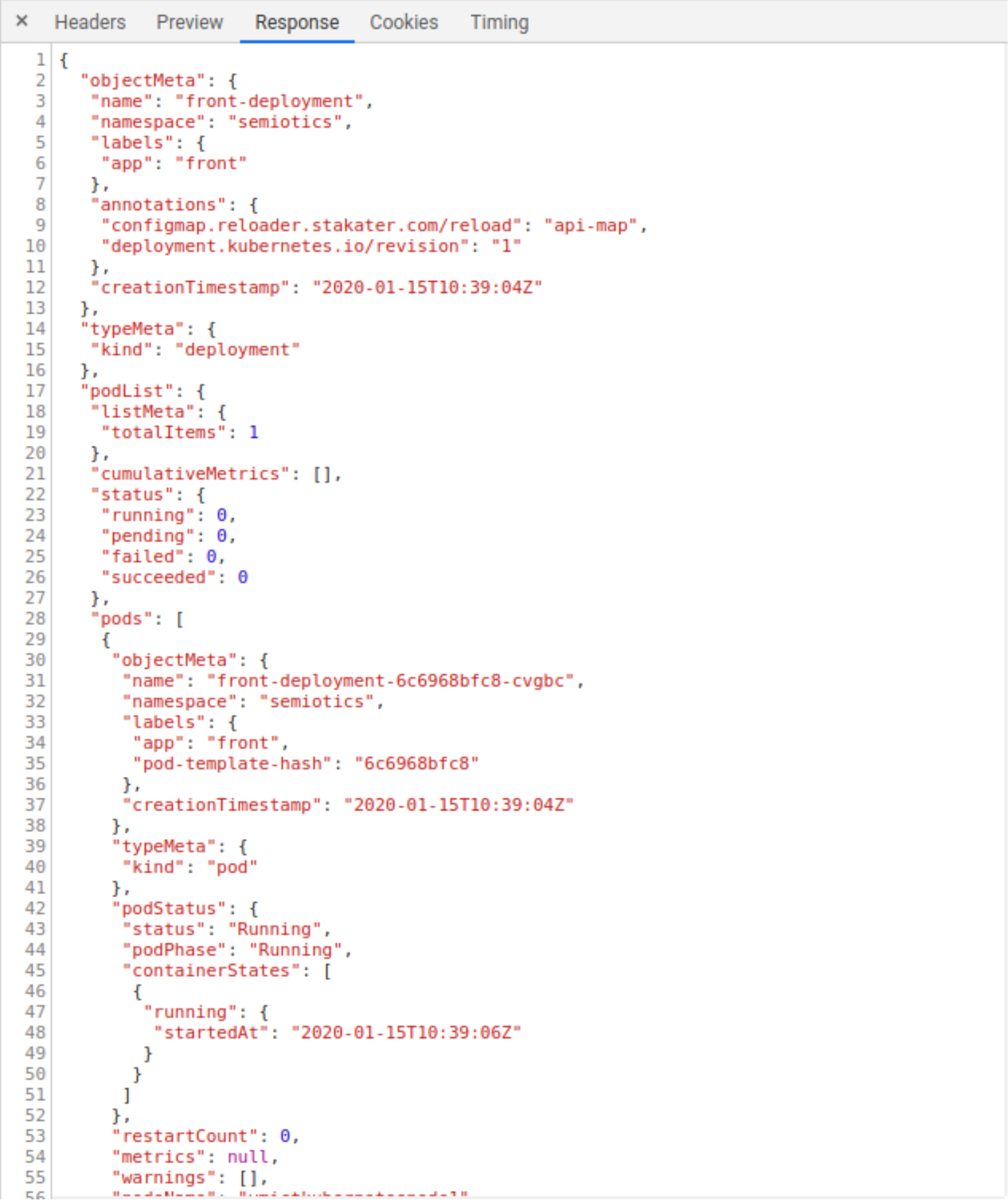
```
[
    {
        "name": "Nowy Sunblind",
        "id": "urn:dev:ops:32473-WoTSunblind-12345",
        "desc": "This is a very smart Sunblind.",
        "hasProperties": true,
        "hasActions": true,
        "hasEvents": true
    },
    {
        "name": "Sunblind",
        "id": "nowy-123:12345",
        "desc": "This is a very smart Sunblind.",
        "hasProperties": true,
        "hasActions": true,
        "hasEvents": true
    }
]
```

FIGURE 98 RESPONSE AFTER SUCCESSFUL DEPLOYMENT



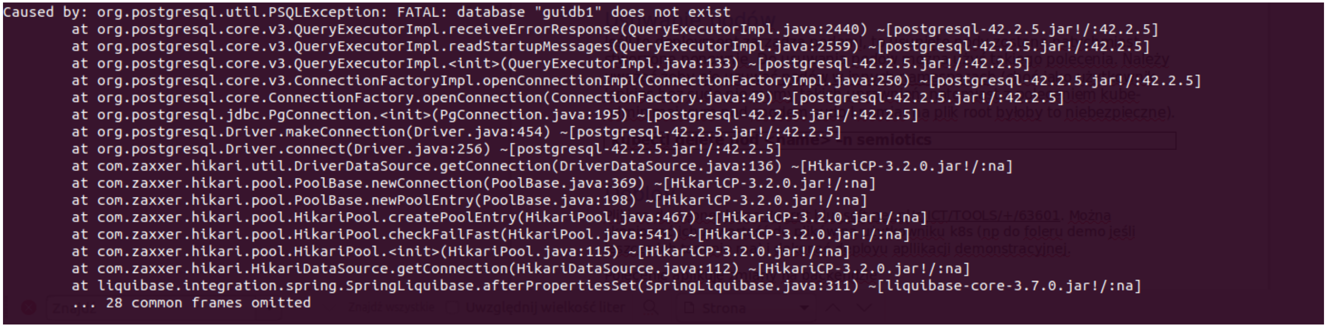
```
{
 "objectMeta": {
  "name": "back-deployment",
  "namespace": "semiotics",
  "labels": {
   "app": "back"
  },
  "annotations": {
   "deployment.kubernetes.io/revision": "1"
  },
  "creationTimestamp": "2020-01-25T19:54:37Z"
 },
 "typeMeta": {
  "kind": "deployment"
 },
 "podList": {
  "listMeta": {
   "totalItems": 1
  },
  "cumulativeMetrics": [],
  "status": {
   "running": 0,
   "pending": 0,
   "failed": 0,
   "succeeded": 0
  },
  "pods": [
   {
    "objectMeta": {
     "name": "back-deployment-65c6cf8fb7-68fwt",
     "namespace": "semiotics",
     "labels": {
      "app": "back",
      "pod-template-hash": "65c6cf8fb7"
     },
     "creationTimestamp": "2020-01-25T19:54:37Z"
    },
    "typeMeta": {
     "kind": "pod"
    },
    "podStatus": {
     "status": "Running",
     "podPhase": "Running",
     "containerStates": [
      {
       "running": {
        "startedAt": "2020-01-27T19:51:35Z"
       }
      }
     ]
    },
    "restartCount": 314,
    "metrics": null,
    "warnings": [],
    "nodeName": "vmictkubernetesnode1"
```

FIGURE 99 EXAMPLE OF SUCCESSFUL RESPONSE USING KUBERNETES API



```
{
  "objectMeta": {
   "name": "front-deployment",
   "namespace": "semiotics",
   "labels": {
    "app": "front"
   },
   "annotations": {
    "configmap.reloader.stakater.com/reload": "api-map",
    "deployment.kubernetes.io/revision": "1"
   },
   "creationTimestamp": "2020-01-15T10:39:04Z"
  },
  "typeMeta": {
   "kind": "deployment"
  },
  "podList": {
   "listMeta": {
    "totalItems": 1
   },
   "cumulativeMetrics": [],
   "status": {
    "running": 0,
    "pending": 0,
    "failed": 0,
    "succeeded": 0
   },
   "pods": [
    {
     "objectMeta": {
      "name": "front-deployment-6c6968bfc8-cvgbc",
      "namespace": "semiotics",
      "labels": {
       "app": "front",
       "pod-template-hash": "6c6968bfc8"
      },
      "creationTimestamp": "2020-01-15T10:39:04Z"
     },
     "typeMeta": {
      "kind": "pod"
     },
     "podStatus": {
      "status": "Running",
      "podPhase": "Running",
      "containerStates": [
       {
        "running": {
         "startedAt": "2020-01-15T10:39:06Z"
        }
       }
      ]
     },
     "restartCount": 0,
     "metrics": null,
     "warnings": [],
```

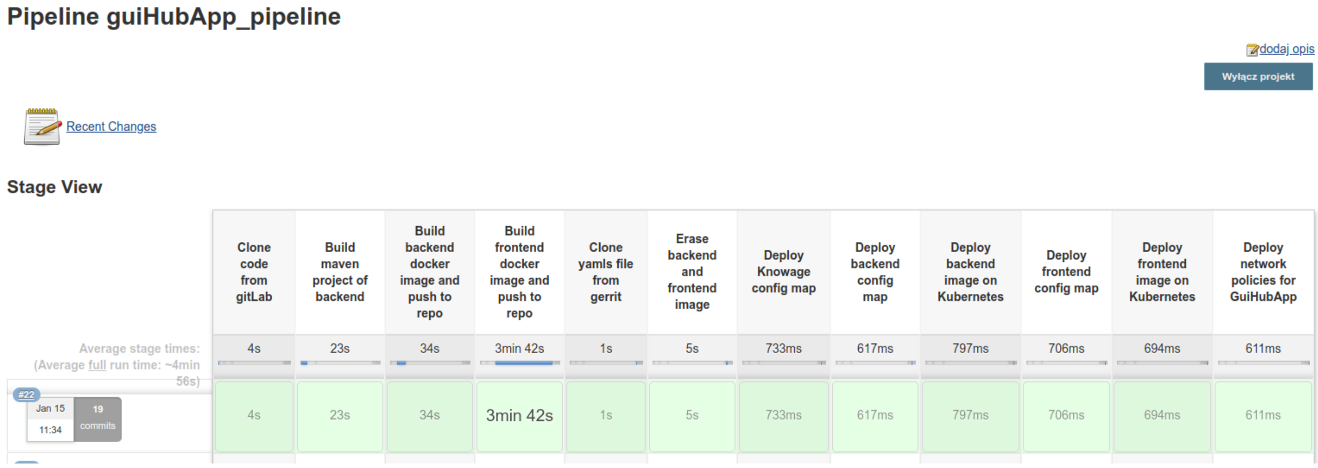
FIGURE 100 EXAMPLE OF A SUCCESSFUL RESPONSE USING KUBERNETES API



```
Caused by: org.postgresql.util.PSQLException: FATAL: database "guidb1" does not exist
	at org.postgresql.core.v3.QueryExecutorImpl.receiveErrorResponse(QueryExecutorImpl.java:2440) ~[postgresql-42.2.5.jar!/:42.2.5]
	at org.postgresql.core.v3.QueryExecutorImpl.readStartupMessages(QueryExecutorImpl.java:2559) ~[postgresql-42.2.5.jar!/:42.2.5]
	at org.postgresql.core.v3.QueryExecutorImpl.<init>(QueryExecutorImpl.java:133) ~[postgresql-42.2.5.jar!/:42.2.5]
	at org.postgresql.core.v3.ConnectionFactoryImpl.openConnectionImpl(ConnectionFactoryImpl.java:250) ~[postgresql-42.2.5.jar!/:42.2.5]
	at org.postgresql.core.ConnectionFactory.openConnection(ConnectionFactory.java:49) ~[postgresql-42.2.5.jar!/:42.2.5]
	at org.postgresql.jdbc.PgConnection.<init>(PgConnection.java:195) ~[postgresql-42.2.5.jar!/:42.2.5]
	at org.postgresql.Driver.makeConnection(Driver.java:454) ~[postgresql-42.2.5.jar!/:42.2.5]
	at org.postgresql.Driver.connect(Driver.java:256) ~[postgresql-42.2.5.jar!/:42.2.5]
	at com.zaxxer.hikari.util.DriverDataSource.getConnection(DriverDataSource.java:136) ~[HikariCP-3.2.0.jar!/:na]
	at com.zaxxer.hikari.pool.PoolBase.newConnection(PoolBase.java:369) ~[HikariCP-3.2.0.jar!/:na]
	at com.zaxxer.hikari.pool.PoolBase.newPoolEntry(PoolBase.java:198) ~[HikariCP-3.2.0.jar!/:na]
	at com.zaxxer.hikari.pool.HikariPool.createPoolEntry(HikariPool.java:467) ~[HikariCP-3.2.0.jar!/:na]
	at com.zaxxer.hikari.pool.HikariPool.checkFailFast(HikariPool.java:541) ~[HikariCP-3.2.0.jar!/:na]
	at com.zaxxer.hikari.pool.HikariPool.<init>(HikariPool.java:115) ~[HikariCP-3.2.0.jar!/:na]
	at com.zaxxer.hikari.HikariDataSource.getConnection(HikariDataSource.java:112) ~[HikariCP-3.2.0.jar!/:na]
	at liquibase.integration.spring.SpringLiquibase.afterPropertiesSet(SpringLiquibase.java:311) ~[liquibase-core-3.7.0.jar!/:na]
	... 28 common frames omitted
```

FIGURE 101 BACKEND ORCHESTRATOR LOGS AFTER DEPLOYMENT FAILURE

After having the certainty that Kubernetes deployment works, the same test scenarios should be performed using Jenkins pipelines. These scenarios are pretty similar to the scenarios above but use Jenkins instead of APIs. Jenkins catches all of the errors on each test step. Jenkins can show the result using a graphical interface which can be found below.

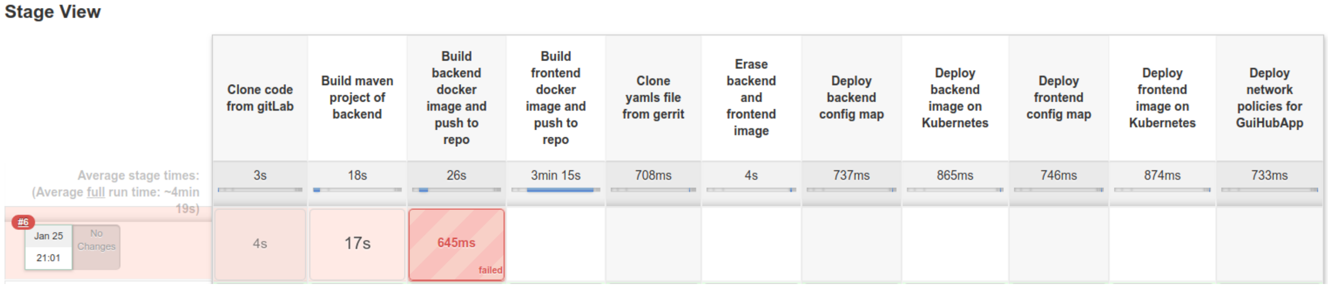


Pipeline guiHubApp_pipeline

Stage View

	Clone code from gitLab	Build maven project of backend	Build backend docker image and push to repo	Build frontend docker image and push to repo	Clone yamls file from gerrit	Erase backend and frontend image	Deploy Knowage config map	Deploy backend config map	Deploy backend image on Kubernetes	Deploy frontend config map	Deploy frontend image on Kubernetes	Deploy network policies for GuiHubApp
Average stage times: (Average full run time: ~4min 56s)	4s	23s	34s	3min 42s	1s	5s	733ms	617ms	797ms	706ms	694ms	611ms
#22 Jan 15 11:34 19 commits	4s	23s	34s	3min 42s	1s	5s	733ms	617ms	797ms	706ms	694ms	611ms

FIGURE 102: A SUCCESSFUL PIPELINE IN JENKINS GRAPHICAL INTERFACE



Stage View

	Clone code from gitLab	Build maven project of backend	Build backend docker image and push to repo	Build frontend docker image and push to repo	Clone yamls file from gerrit	Erase backend and frontend image	Deploy backend config map	Deploy backend image on Kubernetes	Deploy frontend config map	Deploy frontend image on Kubernetes	Deploy network policies for GuiHubApp
Average stage times: (Average full run time: ~4min 19s)	3s	18s	26s	3min 15s	708ms	4s	737ms	865ms	746ms	874ms	733ms
#6 Jan 25 21:01 No Changes	4s	17s	645ms failed								

FIGURE 103:AN UNSUCCESSFUL PIPELINE IN JENKINS GRAPHICAL INTERFACE

3.4.4 DATA VISUALIZATION

The main purpose of data visualization in SEMIoTICS is the graphic presentation of the most important information gathered and generated by the components. More information on how the visualization is done can be found in Deliverable 4.7. Considering the various content of the presented data, the visualization in the SEMIoTICS includes the following parts:

- Thing directory user interface
- Pattern visualization
- Sensor data gathering
- Data visualizations using FIWARE

The GUI is the main module that provides a graphical interface for individual components to show data in on IoT platform. During this Cycle 2 deliverable the GUI was significantly enhanced and upgraded, and the new GUI screens are included in what follows. According to the project assumptions, GUI integrates with Thing Directory, WoT compliant field devices, Pattern Orchestrator and Knowage that is one of the FIWARE Generic Enablers. All types of integration are shortly presented below.

- Integration GUI with Thing Directory
  This integration was created to provide a graphical interface for Thing Directory API to visualize all devices connected to the SEMIoTICS platform. To receive data, GUI through an internal component sends HTTP request to Thing Directory's API and in the response gets JSON with specific information. To avoid problems with the device description, maintain consistency and uniform format in the platform, GUI uses the JSON-LD standard in the above-mentioned communication. After that, the JSON description is translated into a user-friendly form. For this purpose, mapping to a previously defined object is used. It enables a user to watch all devices in table form with the possibility to filter by attributes and properties. Moreover, GUI provides support for the SPARQL filter to search devices directly Thing Directory. This component also allows for adding new things and remove existing ones directly through the platform.

- Integration with Pattern Orchestrator
  This integration aimed to support Pattern Orchestrator in monitoring the current state of SPDI patterns from all recipes and location SPDI patterns in an individual layer. In the Pattern Orchestrator component, a dedicated endpoint was created for a GUI that provides combined data with SPDI patterns and recipes. At this stage of a project, GUI uses a mocked response from Pattern Orchestrator as it is not deployed on Backend Orchestrator yet. GUI translates this data to show it in two possible ways, as patterns with assigned to layers or as a node graph. To avoid problems with the incompatible data, a dedicated JSON model was created.

- Integration for providing sensor data gathering
  A dedicated window was created in GUI to interact with all types of sensors and devices registered in the Thing Directory. This view enables the user to show current values of things properties, collect data with set frequency and actuate actions. Because GUI should communicate not only with Green Field devices but also with Brown Field Devices, so it was integrated with Semantic API & Protocol Binding. This component allows interacting with things that do not support JSON-LD format. As Semantic API & Protocol Binding has not been developed yet, so to test integration a special Thing Simulator component was created. It uses the same methods as Semantic API & Protocol Binding,(GET methods to return properties values and POST methods to control actions).

- Integration with Knowage
  Integration with Knowage, which is one of the FIWARE Generic Enabler, was created to visualize collected data from SEMIoTICS on powerful and efficient dashboards. On these dashboards, users can create various types of widgets like charts, tables, images, documents or own HTML elements. Knowage is embedded in GUI as a frame so does not require redirection to another page. To allow Knowage to have access to the collected data, the thing details view in GUI has been adapted to allow the creation of new datasets when data collection is started. GUI creates dataset through API provided by Knowage and after this operation, datasets are available to use on dashboards. Additionally, GUI

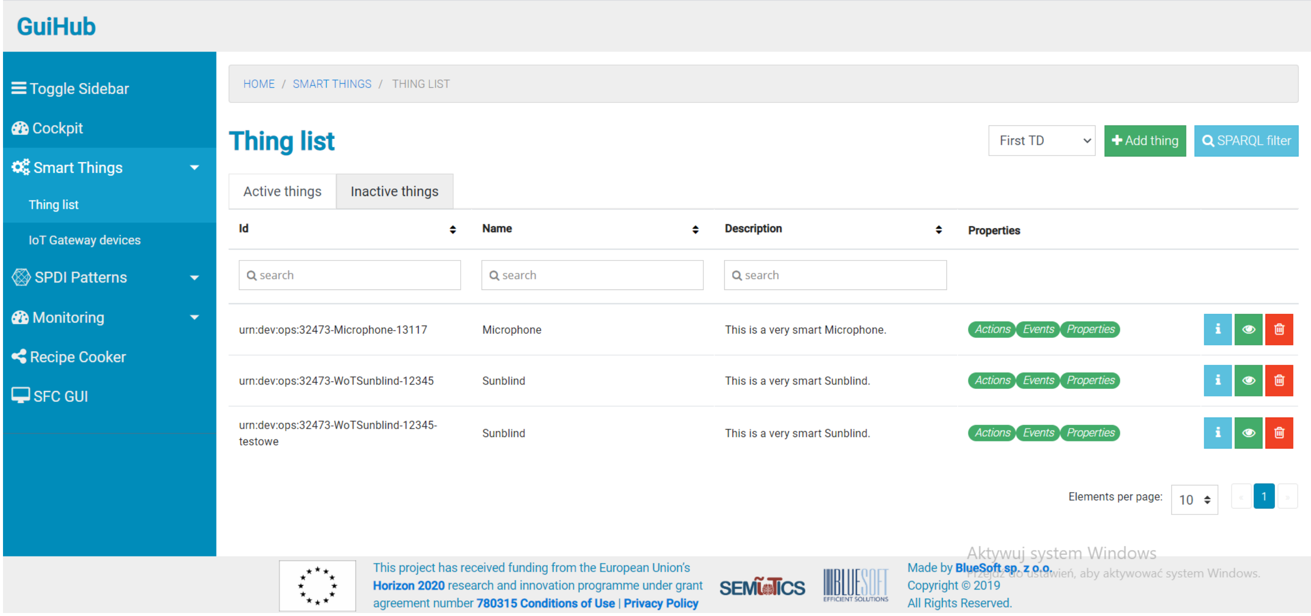
enables the user to create and manage more than one dashboard. A new view with a dashboards list allows the user to create, delete, view, edit and filter dashboards by names or by dynamically created tags.

3.4.4.1 THING DIRECTORY USER INTERFACE

Thing Directory provides multiple APIs to interact with. Below can be found the list of all APIs which are used by GUI component:

- Getting a list of registered devices with description.
- Filtering things using the SPARQL filter.
- Registering new devices using the description in JSON-LD format.
- Deleting devices.

Figures below (FIGURE 104, FIGURE 105, FIGURE 106, FIGURE 107) show the effects of integration between GUI and Thing Directory components.



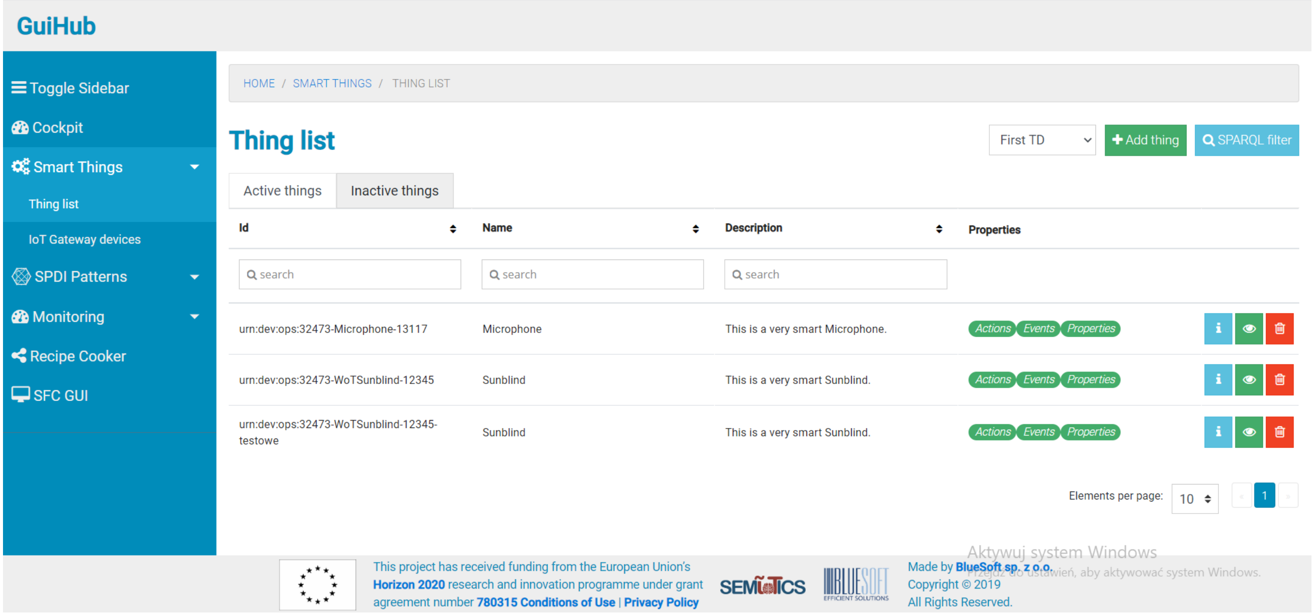
Thing list

Id	Name	Description	Properties	
urn:dev:ops:32473-Microphone-13117	Microphone	This is a very smart Microphone.	Actions Events Properties	
urn:dev:ops:32473-WoTSunblind-12345	Sunblind	This is a very smart Sunblind.	Actions Events Properties	
urn:dev:ops:32473-WoTSunblind-12345-testowe	Sunblind	This is a very smart Sunblind.	Actions Events Properties	

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

Aktywuj system Windows
Przejdź do ustawień, aby aktywować system Windows.



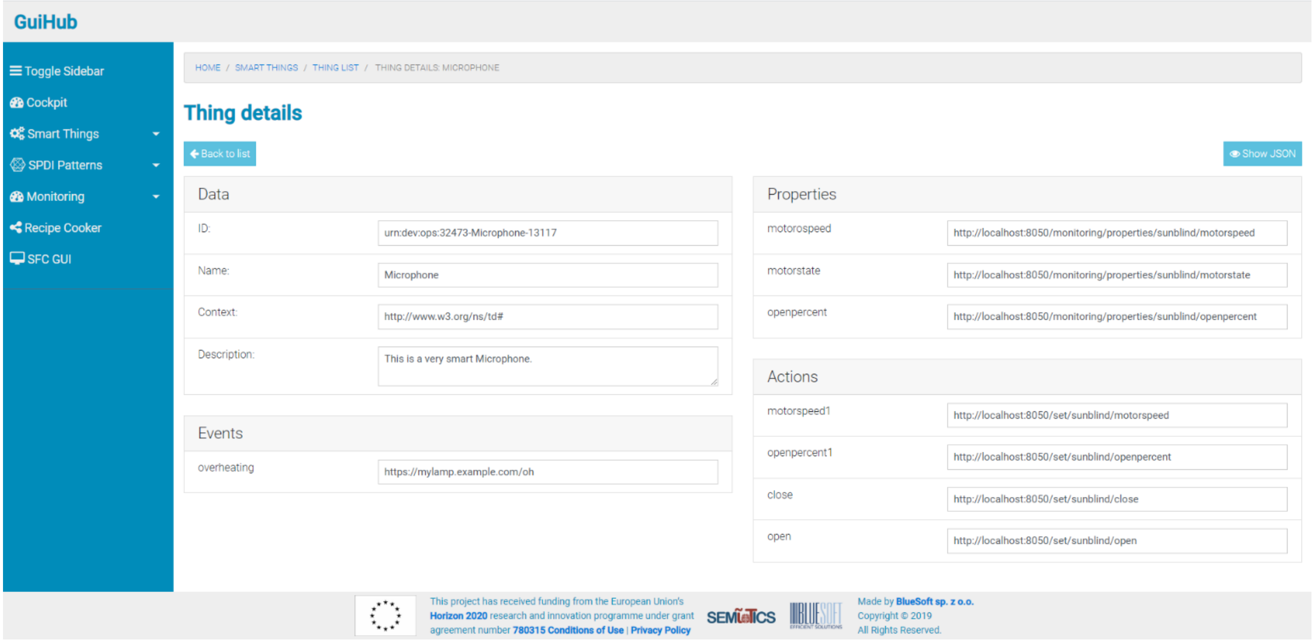
Thing list

Id	Name	Description	Properties
search	search	search	
urn:dev:ops:32473-Microphone-13117	Microphone	This is a very smart Microphone.	Actions Events Properties
urn:dev:ops:32473-WoTSunblind-12345	Sunblind	This is a very smart Sunblind.	Actions Events Properties
urn:dev:ops:32473-WoTSunblind-12345-testowe	Sunblind	This is a very smart Sunblind.	Actions Events Properties

Elements per page: 10 1

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy
Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 104 LIST OF ALL DEVICES REGISTERED IN THING DIRECTORY



Thing details

Data

ID:	urn:dev:ops:32473-Microphone-13117
Name:	Microphone
Context:	http://www.w3.org/ns/td#
Description:	This is a very smart Microphone.

Events

overheating	https://mylamp.example.com/oh

Properties

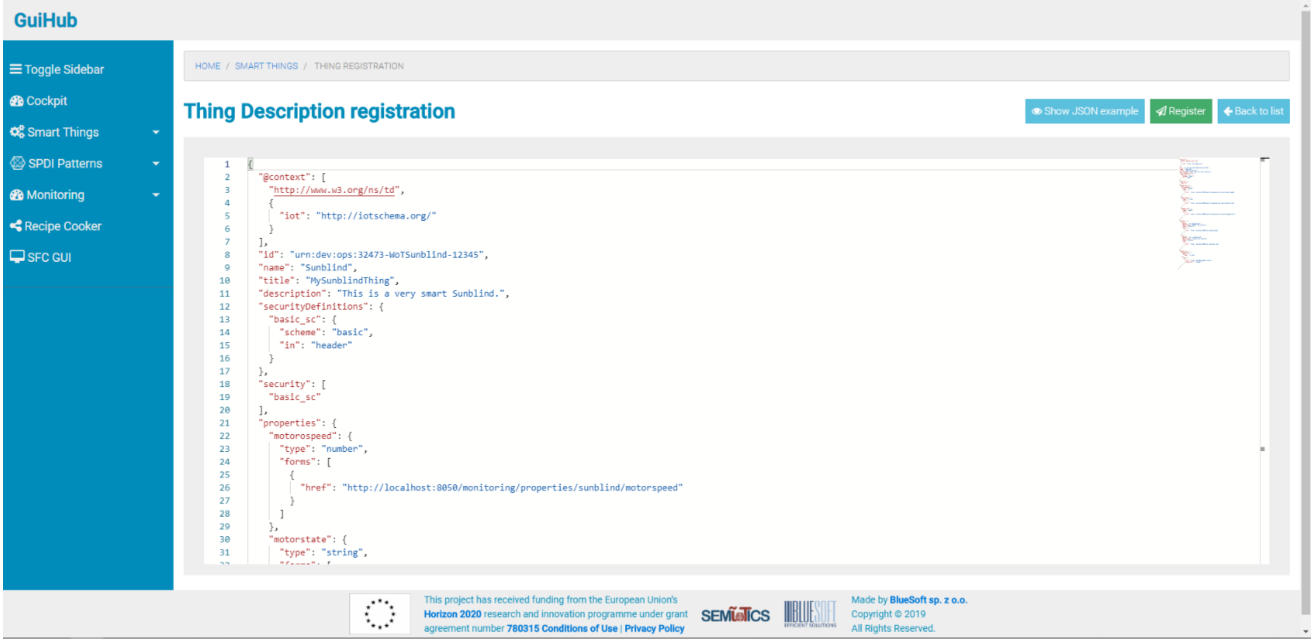
motorospeed	http://localhost:8050/monitoring/properties/sunblind/motorspeed
motorstate	http://localhost:8050/monitoring/properties/sunblind/motorstate
openpercent	http://localhost:8050/monitoring/properties/sunblind/openpercent

Actions

motorspeed1	http://localhost:8050/set/sunblind/motorspeed
openpercent1	http://localhost:8050/set/sunblind/openpercent
close	http://localhost:8050/set/sunblind/close
open	http://localhost:8050/set/sunblind/open

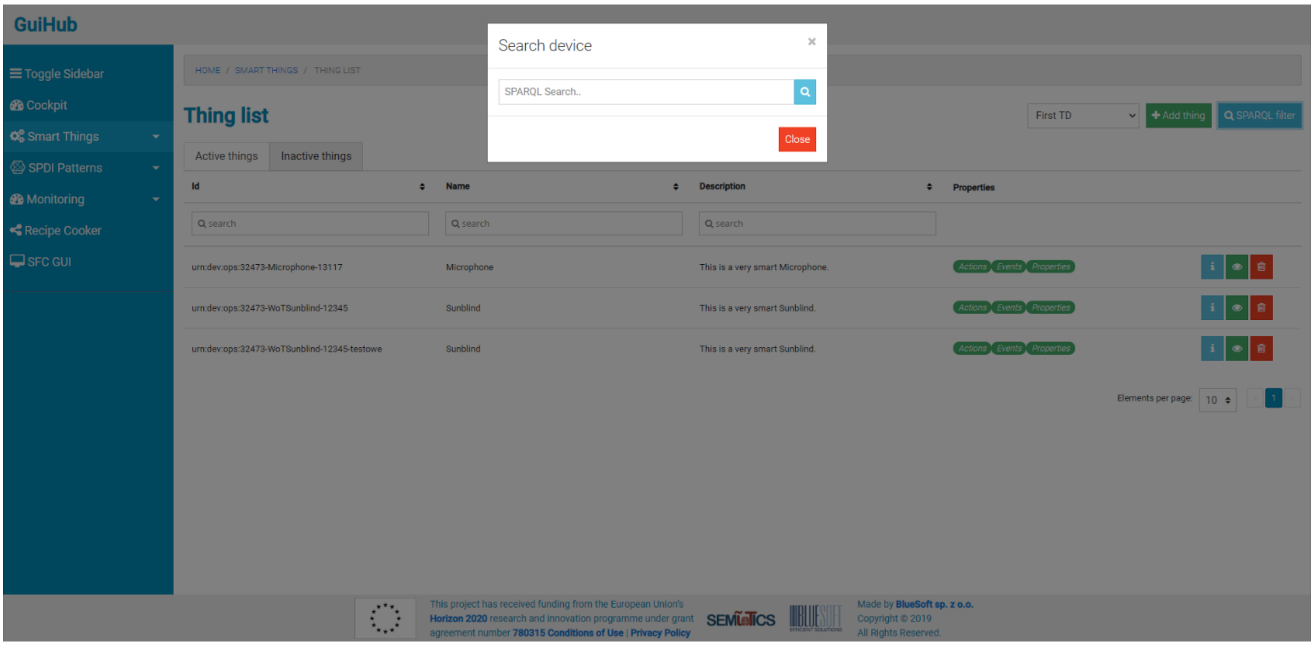
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy
Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 105 THING DETAILS


GuiHub
Toggle Sidebar
Cockpit
Smart Things
SPDI Patterns
Monitoring
Recipe Cooker
SFC GUI
HOME / SMART THINGS / THING REGISTRATION
Thing Description registration
Show JSON example
Register
Back to list
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy
SEMIoTICS
BLUESOFT
Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

```
{
  "@context": [
    "http://www.w3.org/ns/td",
    {
      "iot": "http://iotschema.org/"
    }
  ],
  "id": "urn:dev:ops:32473-WoTSunblind-12345",
  "name": "Sunblind",
  "title": "MySunblindThing",
  "description": "This is a very smart Sunblind.",
  "securityDefinitions": {
    "basic_sc": {
      "scheme": "basic",
      "in": "header"
    }
  },
  "security": [
    "basic_sc"
  ],
  "properties": {
    "motorospeed": {
      "type": "number",
      "forms": [
        {
          "href": "http://localhost:8050/monitoring/properties/sunblind/motorspeed"
        }
      ]
    },
    "motorstate": {
      "type": "string",
```

FIGURE 106 REGISTRATION OF A NEW DEVICE



Id	Name	Description	Properties
urn:dev:ops:32473-Microphone-13117	Microphone	This is a very smart Microphone.	Actions Events Properties
urn:dev:ops:32473-WoTSunblind-12345	Sunblind	This is a very smart Sunblind.	Actions Events Properties
urn:dev:ops:32473-WoTSunblind-12345-testowe	Sunblind	This is a very smart Sunblind.	Actions Events Properties

FIGURE 107 SEARCHING A DEVICE USING SPARQL FILTER

3.4.4.1.1 COMPONENT ARCHITECTURE

Components necessary for providing the Thing Directory user interface are depicted in FIGURE 108. There are highlighted Thing Directory component in the backend layer that provides API mentioned in 3.4.4.1 and GUI component from the backend layer which visualizes this API. A detailed description of the integration between these two components as described in section 3.4.4. Data that is received from Thing Directory is not stored in a GUI database until the user does not start collecting data from things. Only then the simplified description of the thing is saved in the database to identify collected data with them.

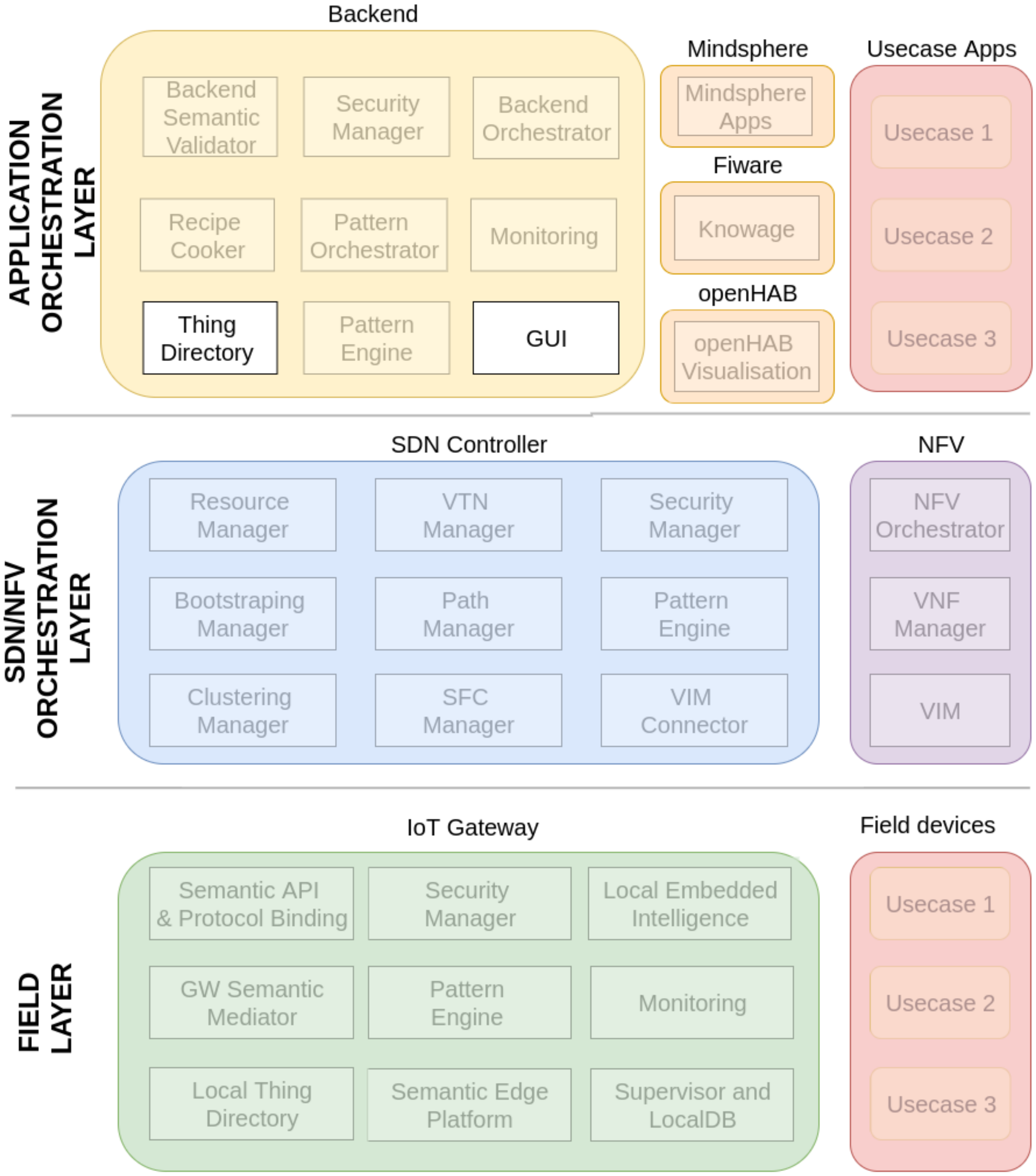


FIGURE 108 COMPONENTS TAKING PART IN INTEGRATION WITH THING DIRECTORY

Sequence diagrams showing integration GUI with Thing Directory are presented below.

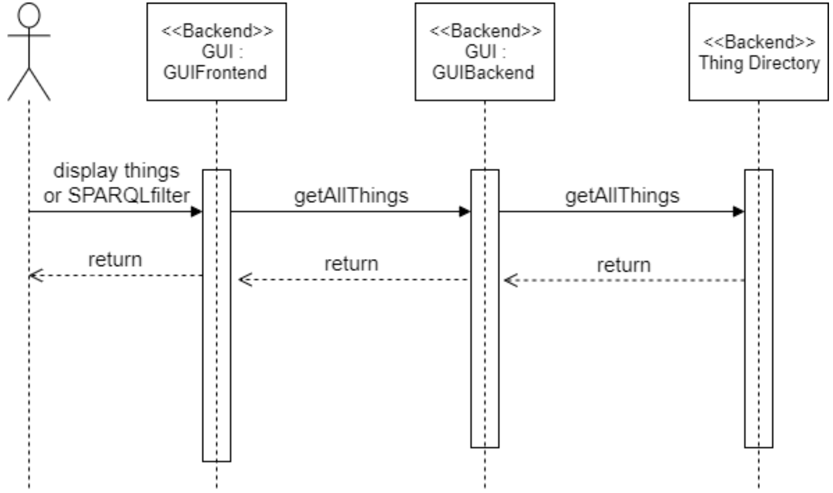

<<Backend>> GUI : GUIFrontend
<<Backend>> GUI : GUIBackend
<<Backend>> Thing Directory
display things or SPARQLfilter
getAllThings
getAllThings
return
return
return

FIGURE 109 SEQUENCE DIAGRAM, DISPLAY ALL DEVICES FROM THING DIRECTORY

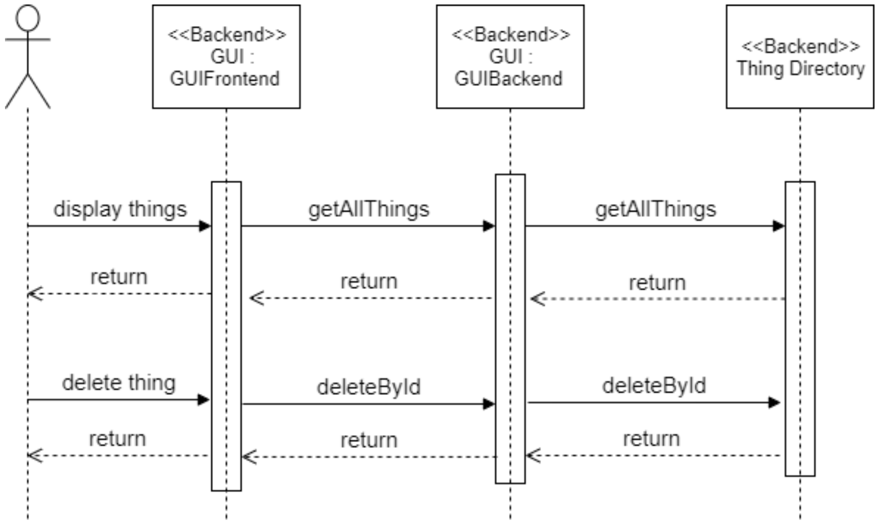


FIGURE 110 SEQUENCE DIAGRAM, DELETE THING FROM THING DIRECTORY

3.4.4.1.1 TESTS

The functional test focuses on calling API exposed by GUI Backend. All the tests

Table 18 Functional tests scenarios for Thing Directory

GUI Integration with Thing Directory
/getAllThings, /filterAllThings
✓ should return results in response when there’s at least one thing description registered in Thing Directory (with 200 OK status)
✓ should return an error if there’s none registered in Thing Directory (with 400 Bad request)
✓ should return an error if there a thing registered in Thing Directory is invalid mapped (with 400 Bad request)

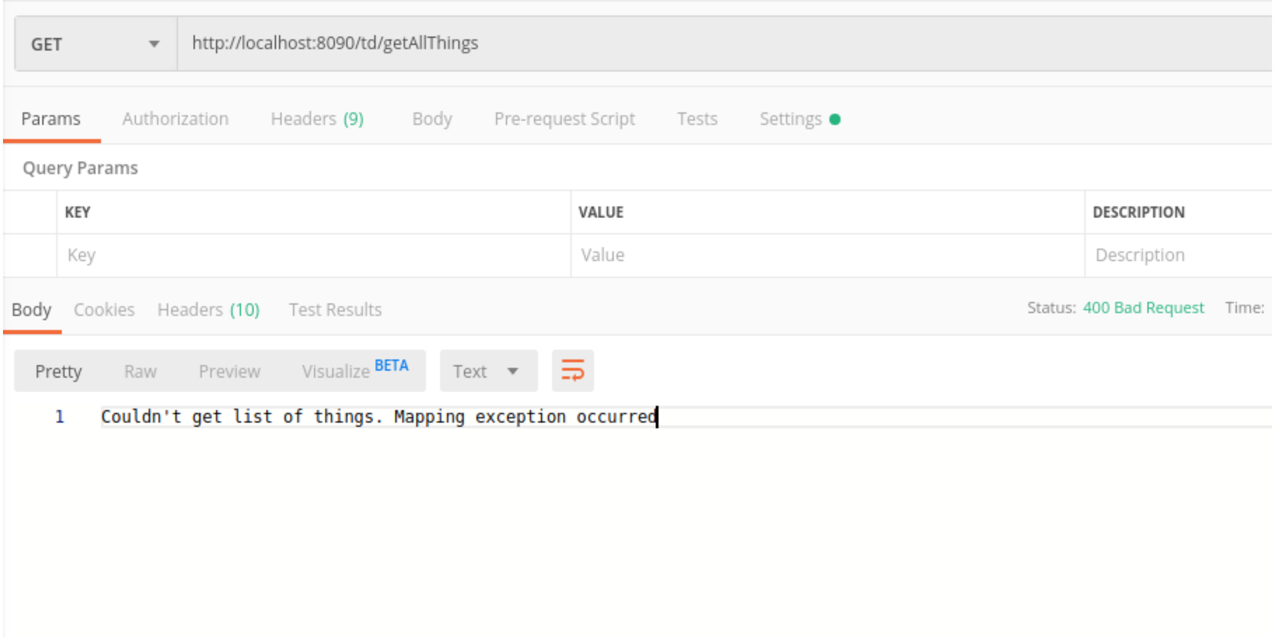
/addThing

✓ should return a success message when request body has got a proper format and its JSON (with 200 OK status)
✓ should return an error when request body has got an improper format or it is not JSON (with 400 Bad request status)

/getThing
✓ should return a result in response if thing description has been registered in Thing Directory (with 200 OK status)
✓ should return an error when a registered thing description has got an improper format (with 400 Bad request status)

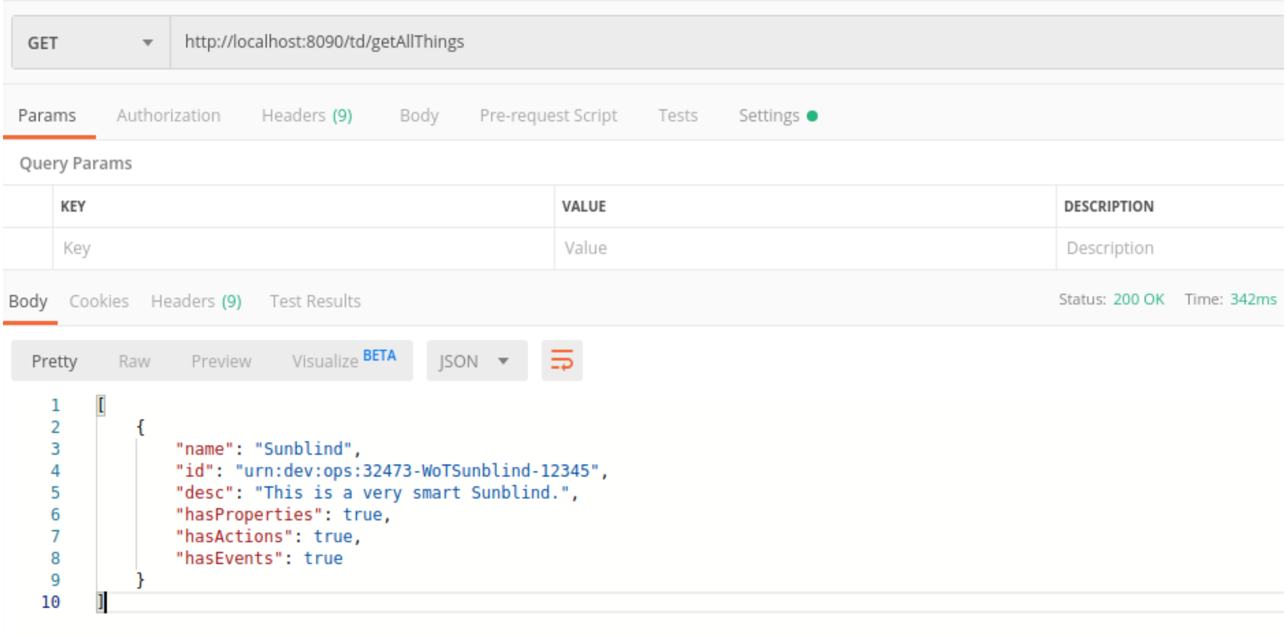
/deleteThing
✓ should return a result in response if the thing with given id has been registered in Thing Directory (with 200 OK status)
✓ should return an error when a given thing id doesn't match with anything description in Thing Directory (with 400 Bad request status)

Integration test are currently being implemented. Below there are examples of API calls.



```
Couldn't get list of things. Mapping exception occurred
```

FIGURE 111: EXAMPLE OF AN UNSUCCESSFUL HTTP REQUEST



GET http://localhost:8090/td/getAllThings

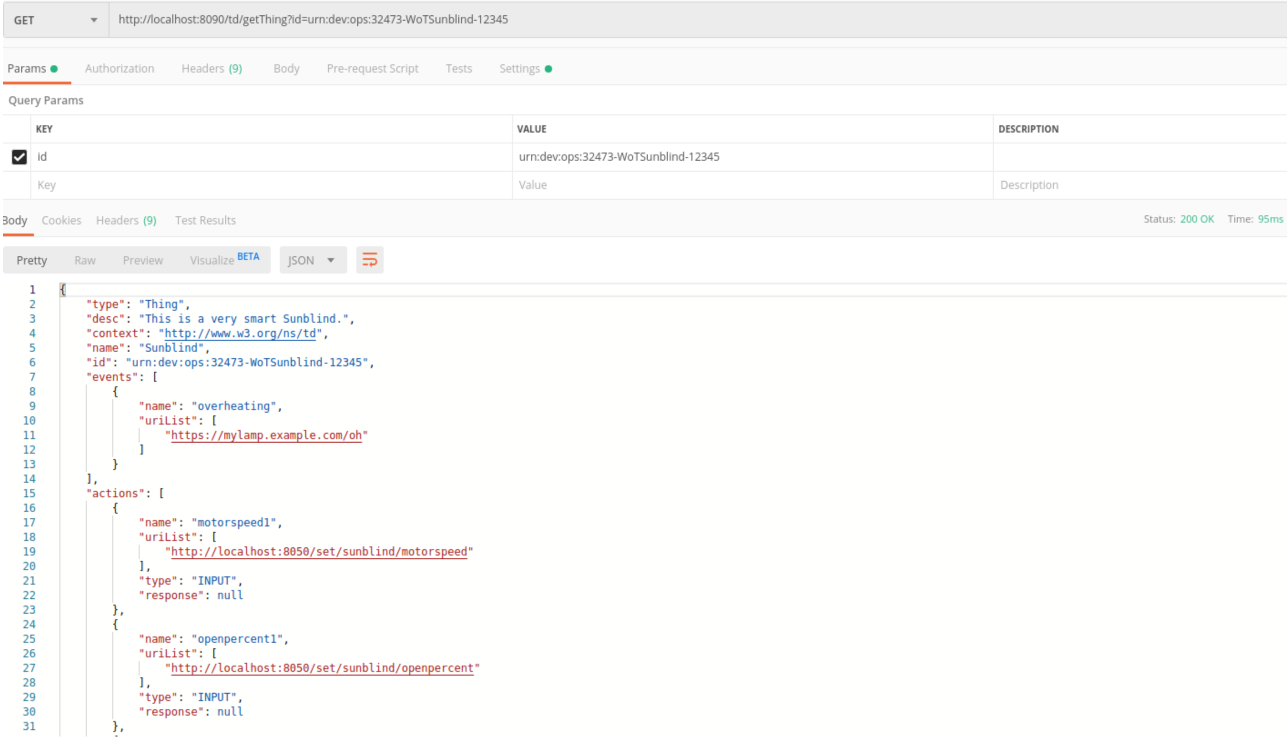
Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Status: 200 OK Time: 342ms

```
[
    {
        "name": "Sunblind",
        "id": "urn:dev:ops:32473-WoTSunblind-12345",
        "desc": "This is a very smart Sunblind.",
        "hasProperties": true,
        "hasActions": true,
        "hasEvents": true
    }
]
```

FIGURE 112:EXAMPLE OF A SUCCESSFUL HTTP REQUEST



GET http://localhost:8090/td/getThing?id=urn:dev:ops:32473-WoTSunblind-12345

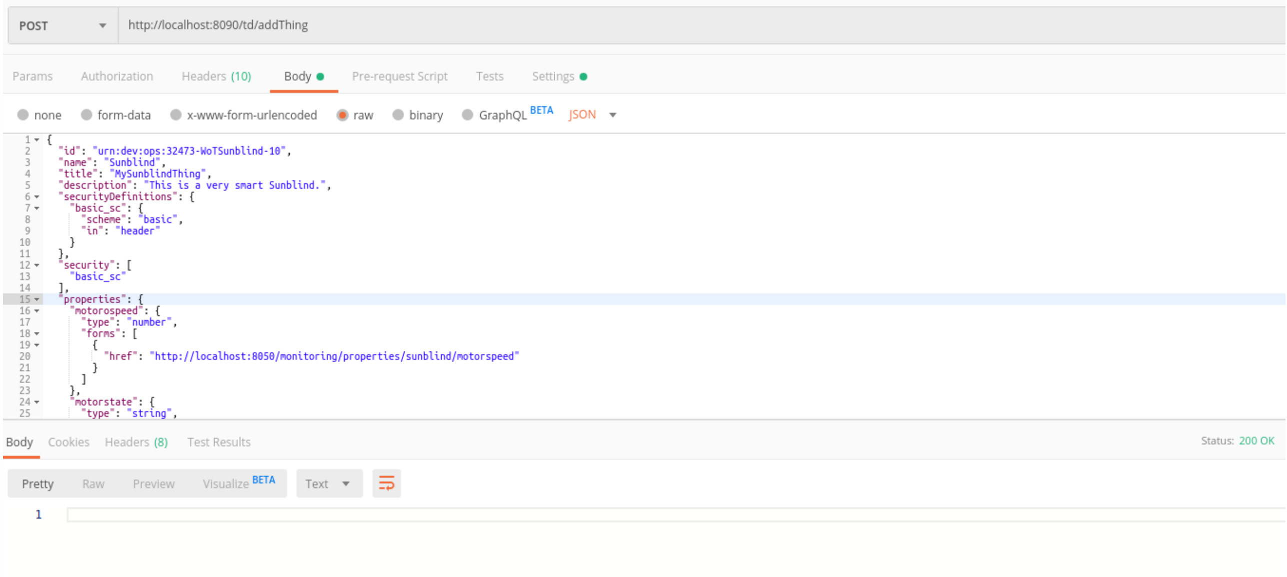
Query Params

	KEY	VALUE	DESCRIPTION
☑	id	urn:dev:ops:32473-WoTSunblind-12345	
	Key	Value	Description

Status: 200 OK Time: 95ms

```
{
    "type": "Thing",
    "desc": "This is a very smart Sunblind.",
    "context": "http://www.w3.org/ns/td",
    "name": "Sunblind",
    "id": "urn:dev:ops:32473-WoTSunblind-12345",
    "events": [
        {
            "name": "overheating",
            "uriList": [
                "https://mylamp.example.com/oh"
            ]
        }
    ],
    "actions": [
        {
            "name": "motorspeed1",
            "uriList": [
                "http://localhost:8050/set/sunblind/motorspeed"
            ],
            "type": "INPUT",
            "response": null
        },
        {
            "name": "openpercent1",
            "uriList": [
                "http://localhost:8050/set/sunblind/openpercent"
            ],
            "type": "INPUT",
            "response": null
        },
```

FIGURE 113: EXAMPLE OF A SUCCESSFUL HTTP REQUEST



POST http://localhost:8090/td/addThing

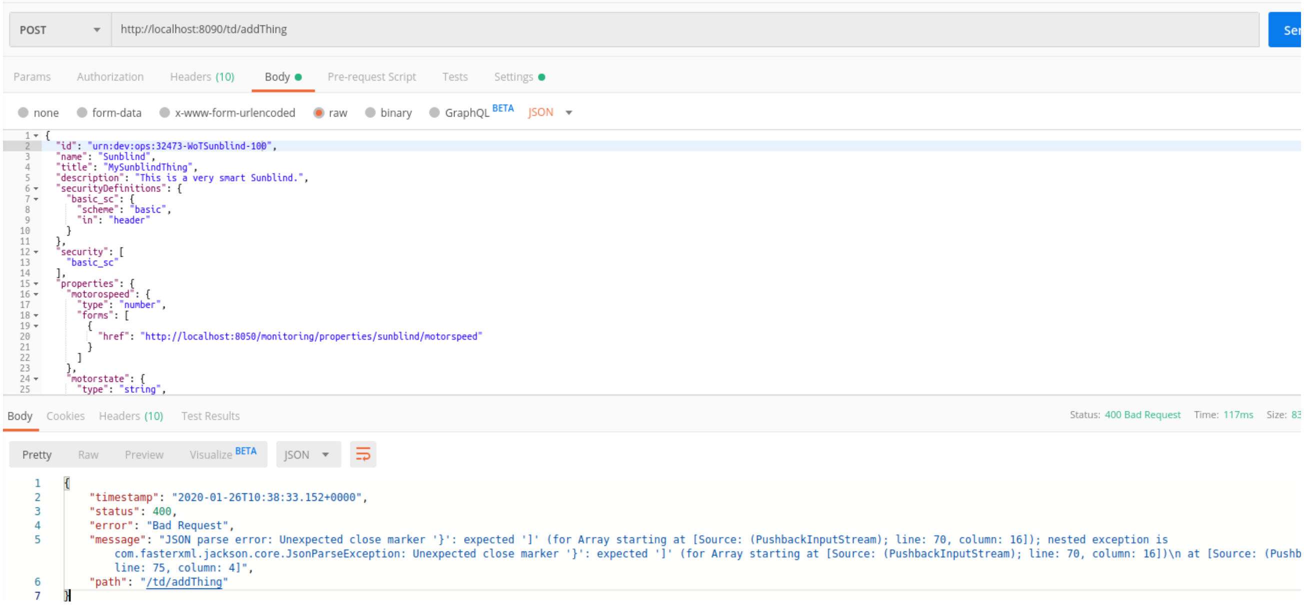
Body: ○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL BETA JSON

```
{
  "id": "urn:dev:ops:32473-WoTSunblind-10",
  "name": "Sunblind",
  "title": "MySunblindThing",
  "description": "This is a very smart Sunblind.",
  "securityDefinitions": {
    "basic_sc": {
      "scheme": "basic",
      "in": "header"
    }
  },
  "security": [
    "basic_sc"
  ],
  "properties": {
    "motorospeed": {
      "type": "number",
      "forms": [
        {
          "href": "http://localhost:8050/monitoring/properties/sunblind/motorspeed"
        }
      ]
    },
    "motorstate": {
      "type": "string",
```

Status: 200 OK

```
1
```

FIGURE 114:EXAMPLE OF A SUCCESSFUL HTTP REQUEST



POST http://localhost:8090/td/addThing

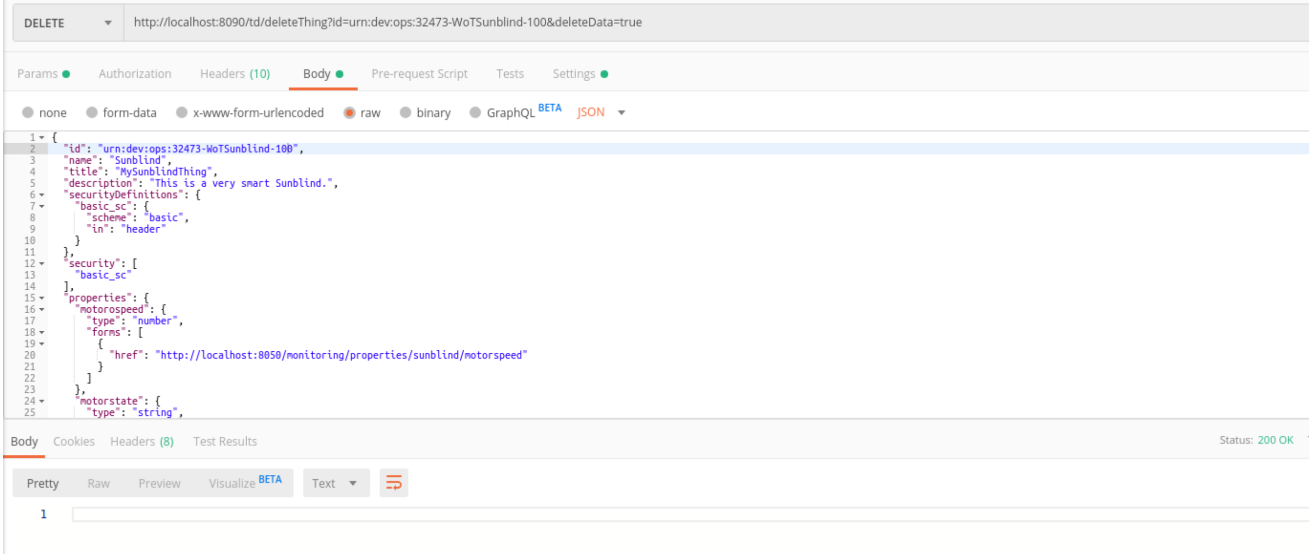
Body: ○ none ○ form-data ○ x-www-form-urlencoded ● raw ○ binary ○ GraphQL BETA JSON

```
{
  "id": "urn:dev:ops:32473-WoTSunblind-100",
  "name": "Sunblind",
  "title": "MySunblindThing",
  "description": "This is a very smart Sunblind.",
  "securityDefinitions": {
    "basic_sc": {
      "scheme": "basic",
      "in": "header"
    }
  },
  "security": [
    "basic_sc"
  ],
  "properties": {
    "motorospeed": {
      "type": "number",
      "forms": [
        {
          "href": "http://localhost:8050/monitoring/properties/sunblind/motorspeed"
        }
      ]
    },
    "motorstate": {
      "type": "string",
```

Status: 400 Bad Request Time: 117ms Size: 83

```
{
    "timestamp": "2020-01-26T10:38:33.152+0000",
    "status": 400,
    "error": "Bad Request",
    "message": "JSON parse error: Unexpected close marker '}': expected ']' (for Array starting at [Source: (PushbackInputStream); line: 70, column: 16]); nested exception is
        com.fasterxml.jackson.core.JsonParseException: Unexpected close marker '}': expected ']' (for Array starting at [Source: (PushbackInputStream); line: 70, column: 16])\n at [Source: (Pushb
        line: 75, column: 4]",
    "path": "/td/addThing"
}
```

FIGURE 115: EXAMPLE OF AN UNSUCCESSFUL HTTP REQUEST

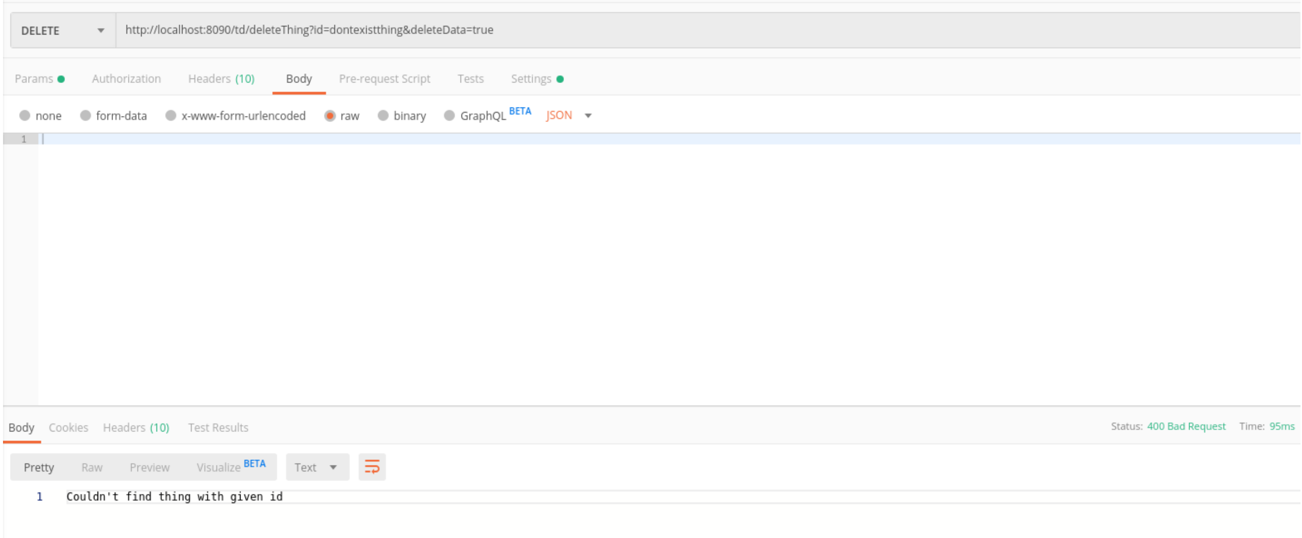


```
http://localhost:8090/td/deleteThing?id=urn:dev:ops:32473-WoTSunblind-100&deleteData=true
```

```
{
  "id": "urn:dev:ops:32473-WoTSunblind-100",
  "name": "Sunblind",
  "title": "MySunblindThing",
  "description": "This is a very smart Sunblind.",
  "securityDefinitions": {
    "basic_sc": {
      "scheme": "basic",
      "in": "header"
    }
  },
  "security": [
    "basic_sc"
  ],
  "properties": {
    "motorospeed": {
      "type": "number",
      "forms": [
        {
          "href": "http://localhost:8050/monitoring/properties/sunblind/motorspeed"
        }
      ]
    },
    "motorstate": {
      "type": "string",
```

Status: 200 OK

FIGURE 116:EXAMPLE OF A SUCCESSFUL HTTP REQUEST



```
http://localhost:8090/td/deleteThing?id=dontexistthing&deleteData=true
```

Status: 400 Bad Request Time: 95ms

```
Couldn't find thing with given id
```

FIGURE 117:EXAMPLE OF AN UNSUCCESSFUL HTTP REQUEST

3.4.4.2 PATTERN VISUALIZATION

The aim goal of pattern visualization was to present and monitor in GUI SPDI patterns stored by Pattern Orchestrator. These patterns are presented in two forms, as a view with layer assignment and as graph representation. To get data, the GUI component communicates with Pattern Orchestrator using one single API, which at this stage of the project is mocked by the JSON object response. It enables to create the entire pattern handling mechanism.

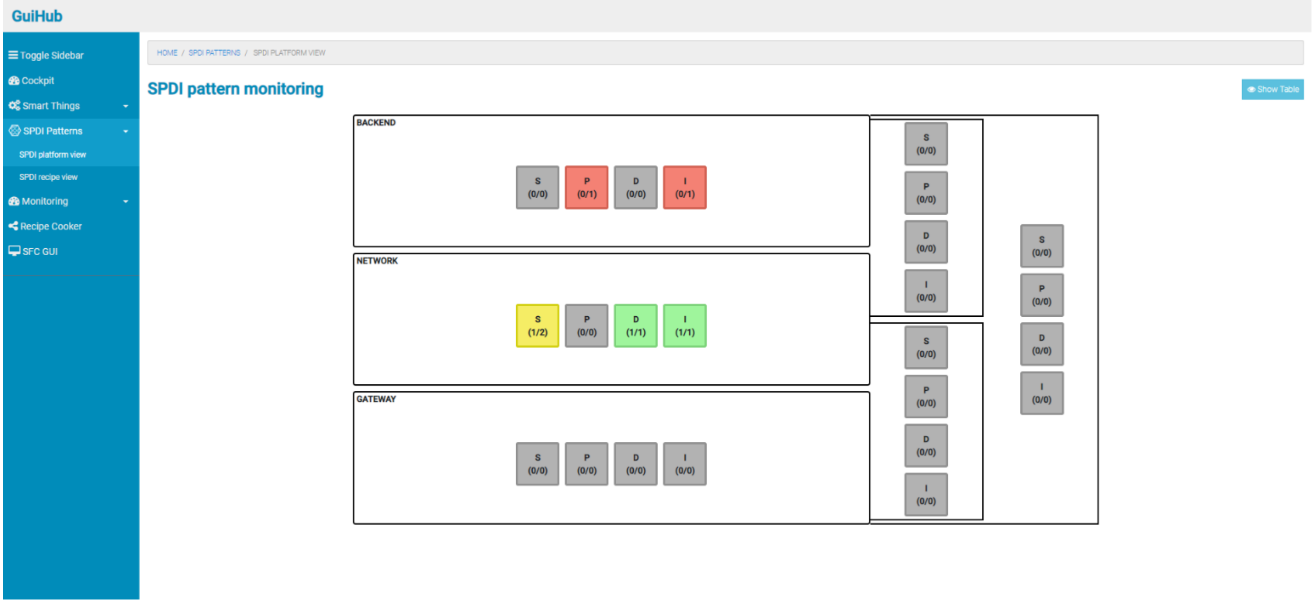
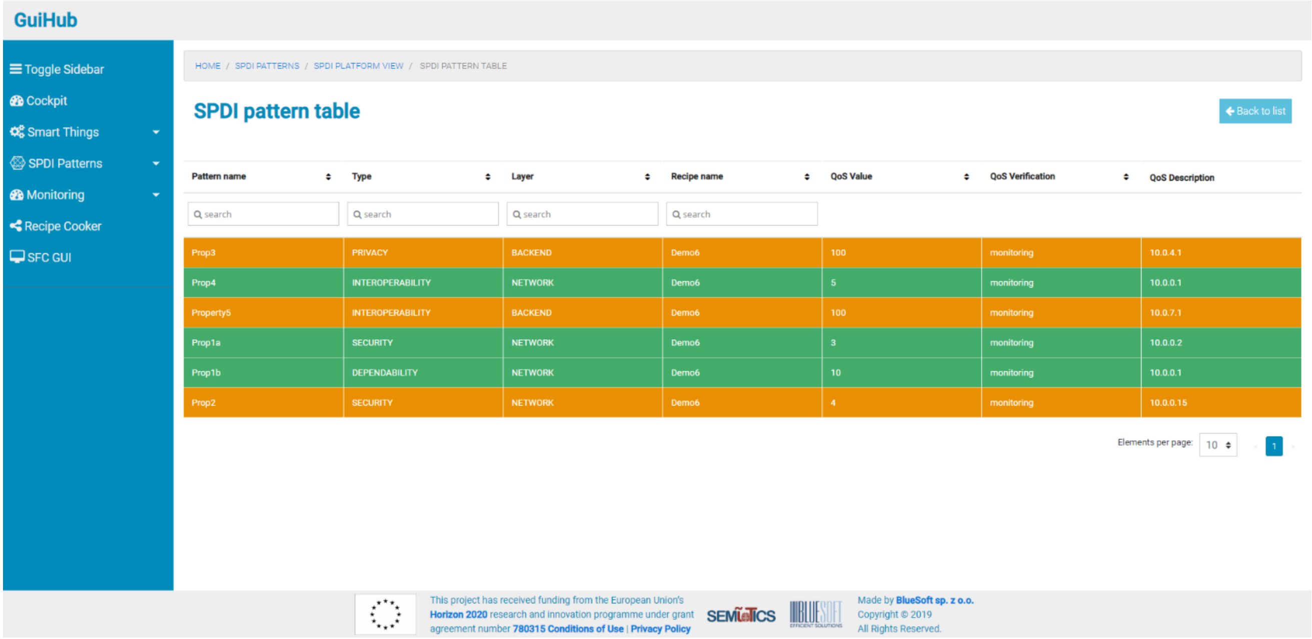


FIGURE 118 SPDI PATTERN MONITORING VIEW



SPDI pattern table

Pattern name	Type	Layer	Recipe name	QoS Value	QoS Verification	QoS Description
search	search	search	search			
Prop3	PRIVACY	BACKEND	Demo6	100	monitoring	10.0.4.1
Prop4	INTEROPERABILITY	NETWORK	Demo6	5	monitoring	10.0.0.1
Property5	INTEROPERABILITY	BACKEND	Demo6	100	monitoring	10.0.7.1
Prop1a	SECURITY	NETWORK	Demo6	3	monitoring	10.0.0.2
Prop1b	DEPENDABILITY	NETWORK	Demo6	10	monitoring	10.0.0.1
Prop2	SECURITY	NETWORK	Demo6	4	monitoring	10.0.0.15

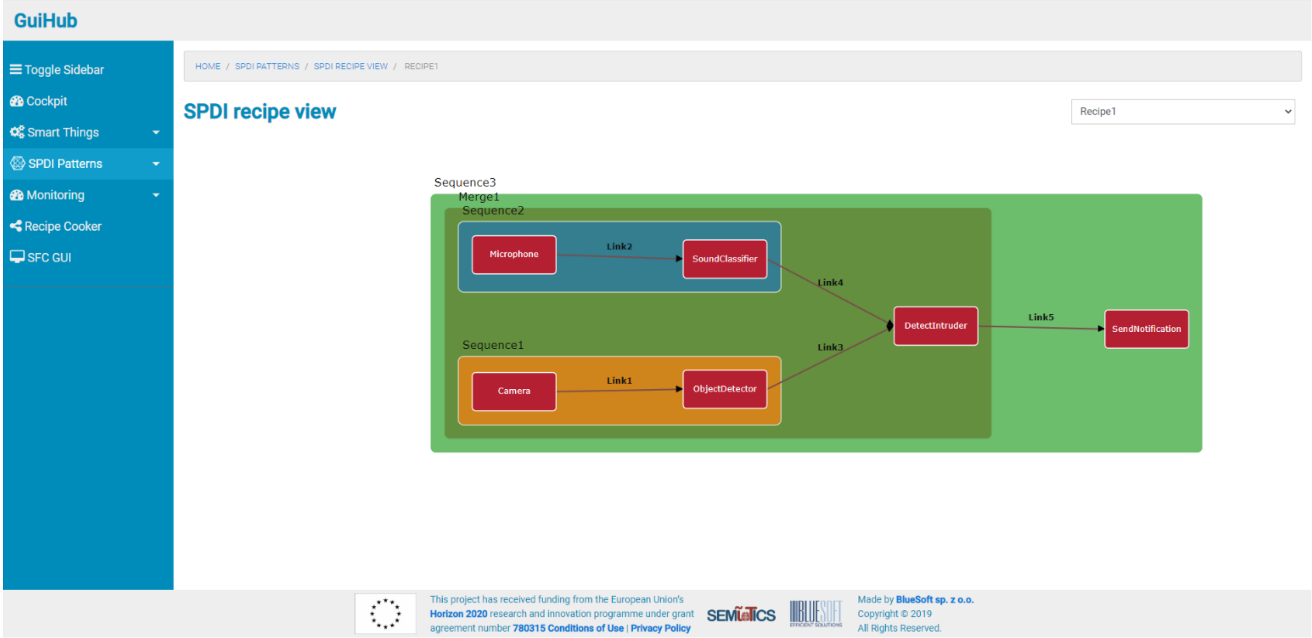
Elements per page: 10

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 119 SPDI PATTERN TABLE

Graph visualization was done using the open source WebCola library. A converted JSON response from Pattern Ochestrator is next passed into javascript function and using the aforementioned library and D3.js, a graph is drawn. The graphs are drawn in real-time and require a 3 second window to completely initialize. This is to ensure proper readibilty as well as full interactivity meaning a single element can be clicked in order to open a detailed node view.



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 120 SPDI GRAPH VIEW

3.4.4.2.1 COMPONENT ARCHITECTURE

FIGURE 121 presents components responsible for pattern monitoring. In this case, the communication is between two components from the backend layer, GUI and Patter Orchestrator. As it was mentioned in section 3.4.4 GUI sends a request to single Pattern Orchestrator API to receive extended JSON description to visualize patterns and recipes in graph form. As this API provides combined data from Pattern Orchestrator and Recipe Cooker, GUI sends only one request instead of two to different components. This reduces the risk of possible errors related to the integration of subsequent components. The data received is immediately processed by subcomponents of GUI and shown to the user. Information about patterns and recipes are not stored in the GUI database.

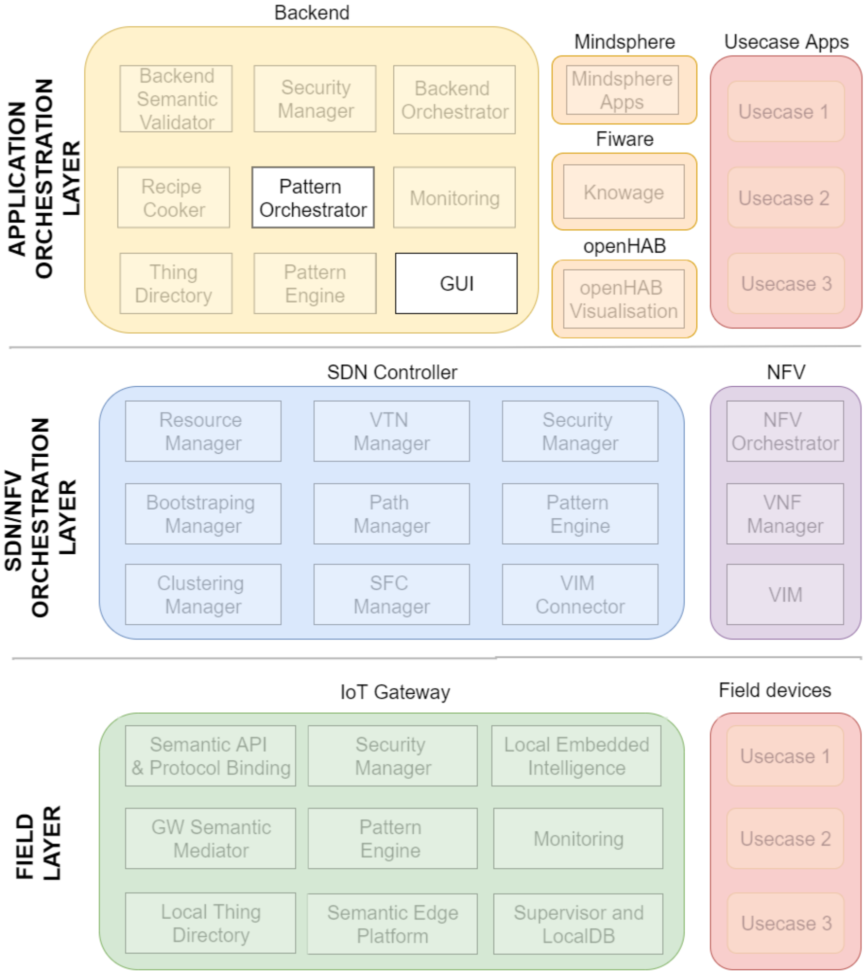


FIGURE 121 COMPONENTS RELATED TO PATTERN VISUALIZATION

Sequence diagram that present communication between GUI and Pattern Orchestrator to visualize patterns is depicted in FIGURE 122.

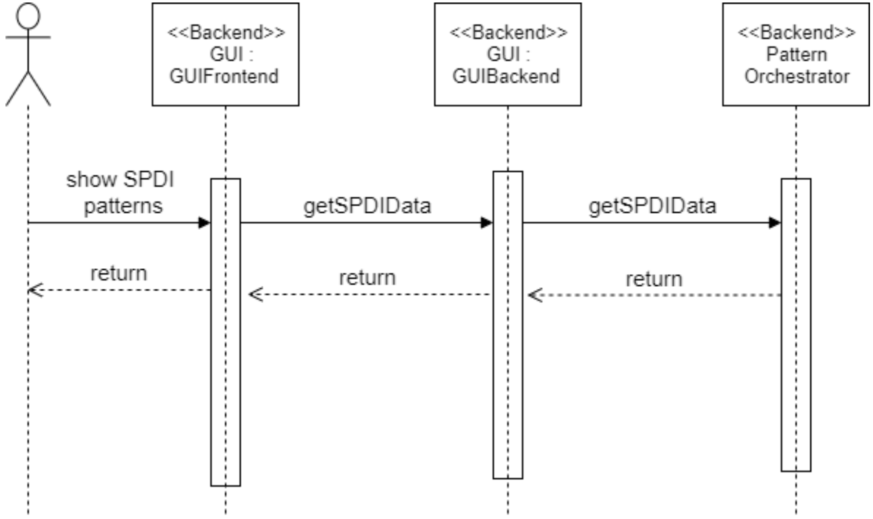


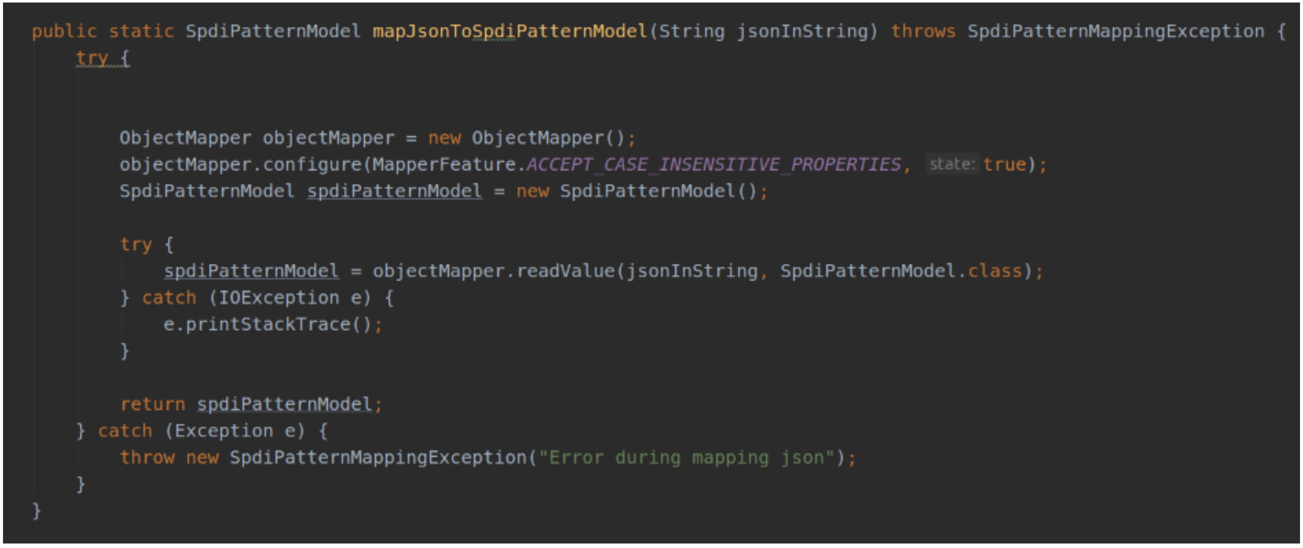
FIGURE 122 SEQUENCE DIAGRAM, SHOW SPDI PATTERNS

3.4.4.2.2 TESTS

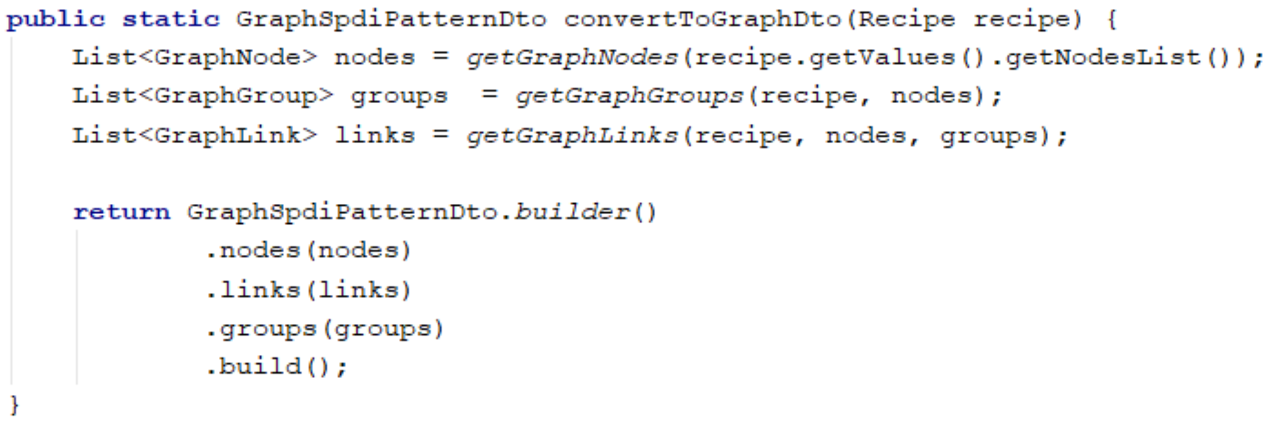
Integration between GUI and Pattern Orchestrator is still in progress because the PO is not deployed in Backend Orchestrator yet. API for getting patterns is prepared in both components as well as a common format for data exchange. The method to map JSON in the GUI is depicted in FIGURE 101. At this stage, the GUI component visualizes a mocked response from PO API and an example of JSON is presented in FIGURE 102.

Table 19 Functional tests scenarios for Pattern Orchestrator

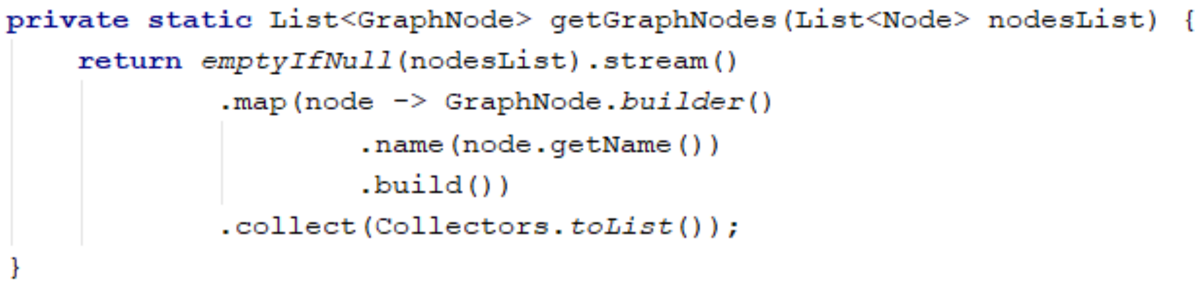
Pattern Orchestrator (integration in progress)
#getSPDIPatterns
✓ **Should reject with 403 error if user attempt to get patterns without proper role**
✓ **Should reject with 404 error if is not able to return patterns**
✓ **Should reject with 400 error if request is incorrect**
✓ **Should reject with 500 error if internal error occurs**
✓ **Should return JSON with patterns with 200 status**



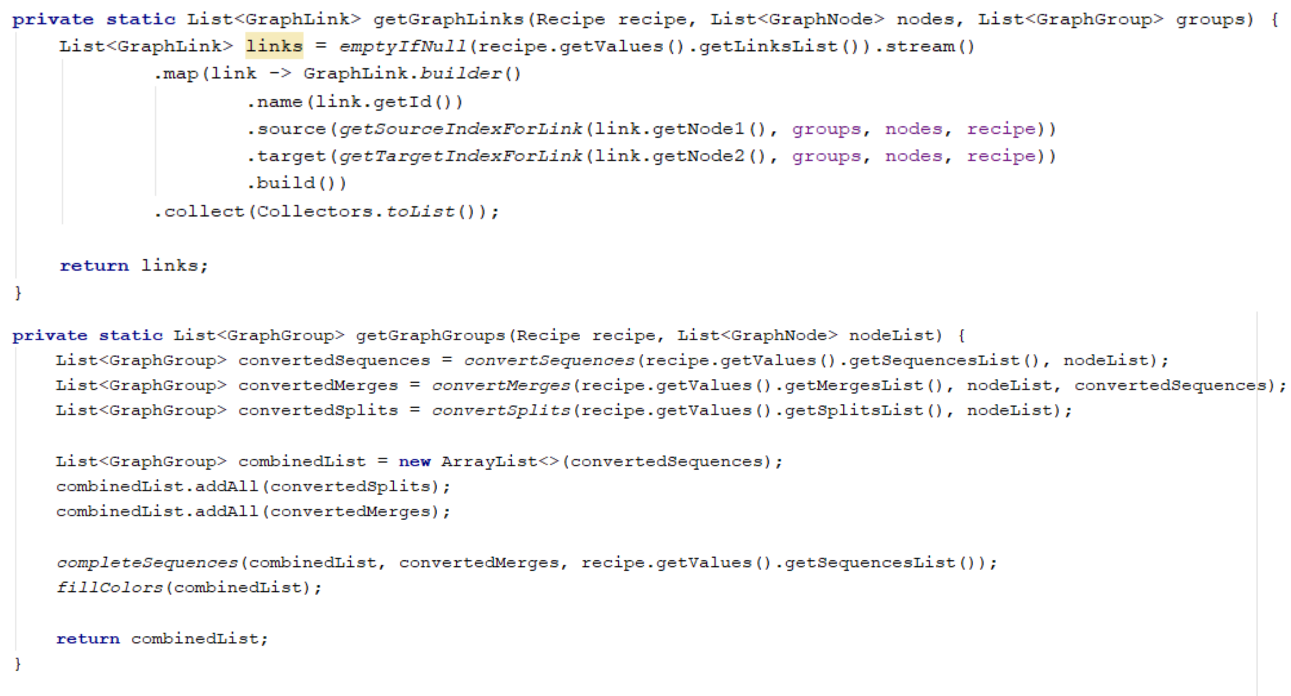
```
public static SpdiPatternModel mapJsonToSpdiPatternModel(String jsonInString) throws SpdiPatternMappingException {
    try {


        ObjectMapper objectMapper = new ObjectMapper();
        objectMapper.configure(MapperFeature.ACCEPT_CASE_INSENSITIVE_PROPERTIES, state: true);
        SpdiPatternModel spdiPatternModel = new SpdiPatternModel();

        try {
            spdiPatternModel = objectMapper.readValue(jsonInString, SpdiPatternModel.class);
        } catch (IOException e) {
            e.printStackTrace();
        }

        return spdiPatternModel;
    } catch (Exception e) {
        throw new SpdiPatternMappingException("Error during mapping json");
    }
}
```



```
public static GraphSpdiPatternDto convertToGraphDto(Recipe recipe) {
    List<GraphNode> nodes = getGraphNodes(recipe.getValues().getNodesList());
    List<GraphGroup> groups  = getGraphGroups(recipe, nodes);
    List<GraphLink> links = getGraphLinks(recipe, nodes, groups);

    return GraphSpdiPatternDto.builder()
            .nodes(nodes)
            .links(links)
            .groups(groups)
            .build();
}
```

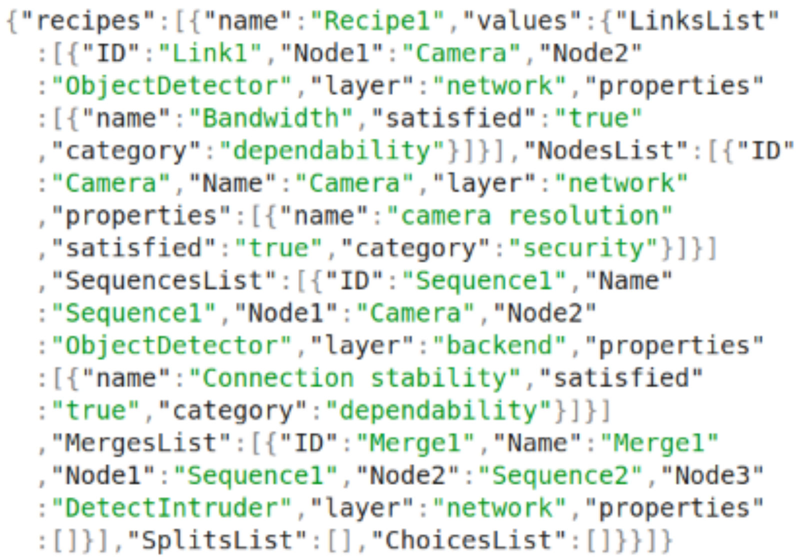


```
private static List<GraphNode> getGraphNodes(List<Node> nodesList) {
    return emptyIfNull(nodesList).stream()
            .map(node -> GraphNode.builder()
                    .name(node.getName())
                    .build())
            .collect(Collectors.toList());
}
```



```
private static List<GraphLink> getGraphLinks(Recipe recipe, List<GraphNode> nodes, List<GraphGroup> groups) {
    List<GraphLink> links = emptyIfNull(recipe.getValues().getLinksList()).stream()
            .map(link -> GraphLink.builder()
                    .name(link.getId())
                    .source(getSourceIndexForLink(link.getNode1(), groups, nodes, recipe))
                    .target(getTargetIndexForLink(link.getNode2(), groups, nodes, recipe))
                    .build())
            .collect(Collectors.toList());

    return links;
}

private static List<GraphGroup> getGraphGroups(Recipe recipe, List<GraphNode> nodeList) {
    List<GraphGroup> convertedSequences = convertSequences(recipe.getValues().getSequencesList(), nodeList);
    List<GraphGroup> convertedMerges = convertMerges(recipe.getValues().getMergesList(), nodeList, convertedSequences);
    List<GraphGroup> convertedSplits = convertSplits(recipe.getValues().getSplitsList(), nodeList);

    List<GraphGroup> combinedList = new ArrayList<>(convertedSequences);
    combinedList.addAll(convertedSplits);
    combinedList.addAll(convertedMerges);

    completeSequences(combinedList, convertedMerges, recipe.getValues().getSequencesList());
    fillColors(combinedList);

    return combinedList;
}
```

FIGURE 101 METHODS TO MAP JSON IN GUI

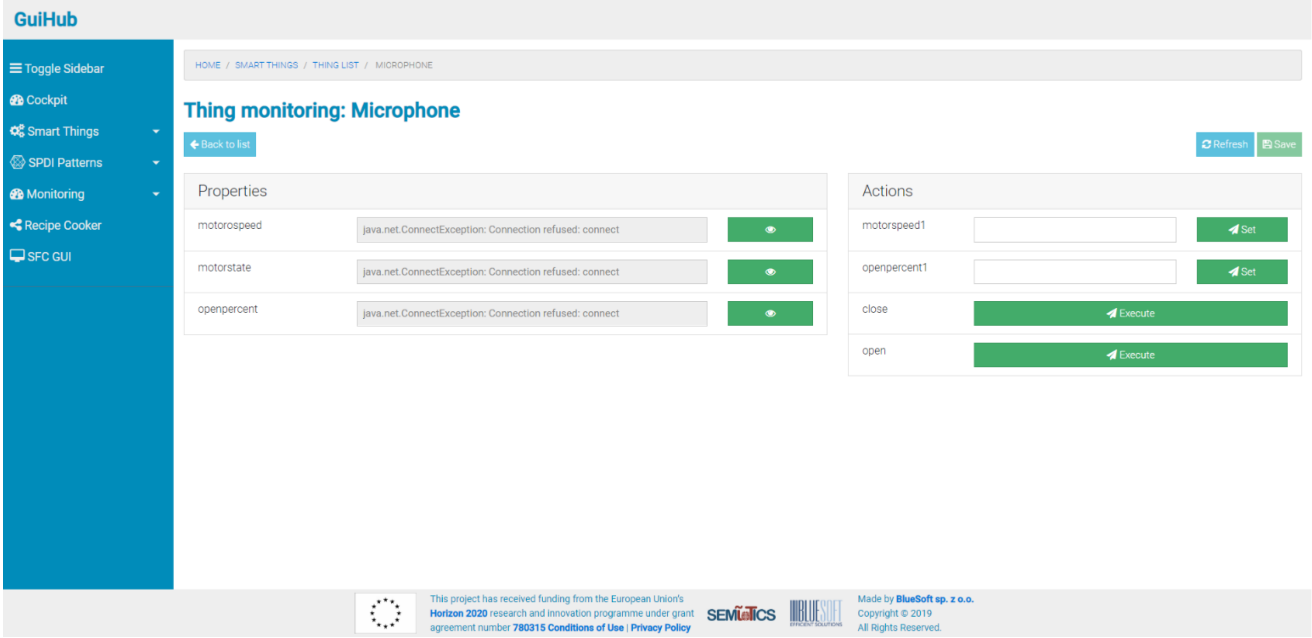


```
{"recipes":[{"name":"Recipe1","values":{"LinksList"
  :[{"ID":"Link1","Node1":"Camera","Node2"
  :"ObjectDetector","layer":"network","properties"
  :[{"name":"Bandwidth","satisfied":"true"
  ,"category":"dependability"}]}],"NodesList":[{"ID"
  :"Camera","Name":"Camera","layer":"network"
  ,"properties":[{"name":"camera resolution"
  ,"satisfied":"true","category":"security"}]}]
  ,"SequencesList":[{"ID":"Sequence1","Name"
  :"Sequence1","Node1":"Camera","Node2"
  :"ObjectDetector","layer":"backend","properties"
  :[{"name":"Connection stability","satisfied"
  :"true","category":"dependability"}]}]
  ,"MergesList":[{"ID":"Merge1","Name":"Merge1"
  ,"Node1":"Sequence1","Node2":"Sequence2","Node3"
  :"DetectIntruder","layer":"network","properties"
  :[]}],"SplitsList":[],"ChoicesList":[]}}]}
```

FIGURE 102 EXAMPLE OF JSON OBTAINED FROM PO

3.4.4.3 SENSOR DATA GATHERING

This integration was done to control all types of devices (Brown Field Devices and Green Field Devices) connected to the SEMIoTICS platform. It allows starting collecting data from a property of a selected device with the frequency set by the user and with a limit date and viewing the current property values of devices or sensors. It also enables to actuate different types of actions defined for the thing. FIGURE 123 below depicts the view for the management of data gathering.



Thing monitoring: Microphone

Properties	
motorospeed	java.net.ConnectException: Connection refused: connect
motorstate	java.net.ConnectException: Connection refused: connect
openpercent	java.net.ConnectException: Connection refused: connect

Actions	
motorspeed1	Set
openpercent1	Set
close	Execute
open	Execute

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 123 VIEW TO THE MANAGEMENT OF DATA GATHERING

3.4.4.3.1 COMPONENT ARCHITECTURE

Architecture for sensor data gathering that is presented in FIGURE 124 contains GUI component from backend layer, Thing Directory from backend layer, Semantic API & Protocol Binding from field layer and Simulated Sensor from field layer which simulates physical sensor. This integration enables to fully control devices which means read real-time data (FIGURE 125) and actuate actions (FIGURE 126). GUI component communicates with devices in two different ways, depending on the type of device as it was presented in section 3.4.4. For greenfield devices, GUI sends requests directly to things that provide a single API for each action and property. For brownfield devices, GUI communicates with Semantic API & Protocol Binding from the field layer which needs GW Semantic Mediator to translate data from thing. At this stage of project physical devices are mocked by Java simulator e.g. Simulated Sensor. Data from mocked devices is collected by the GUI subcomponent (Thing Worker) and stored in the database when the user decides to start gathering information. The sequence diagram showing collecting data is depicted in FIGURE 127.

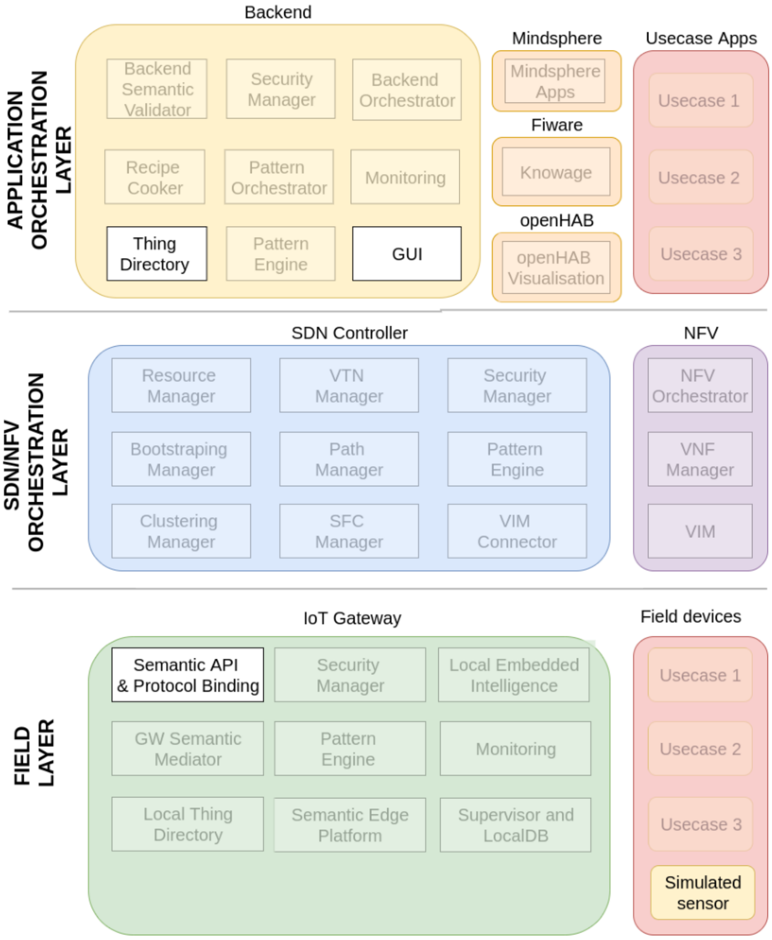


FIGURE 124 COMPONENTS RELATED TO SENSOR DATA GATHERING

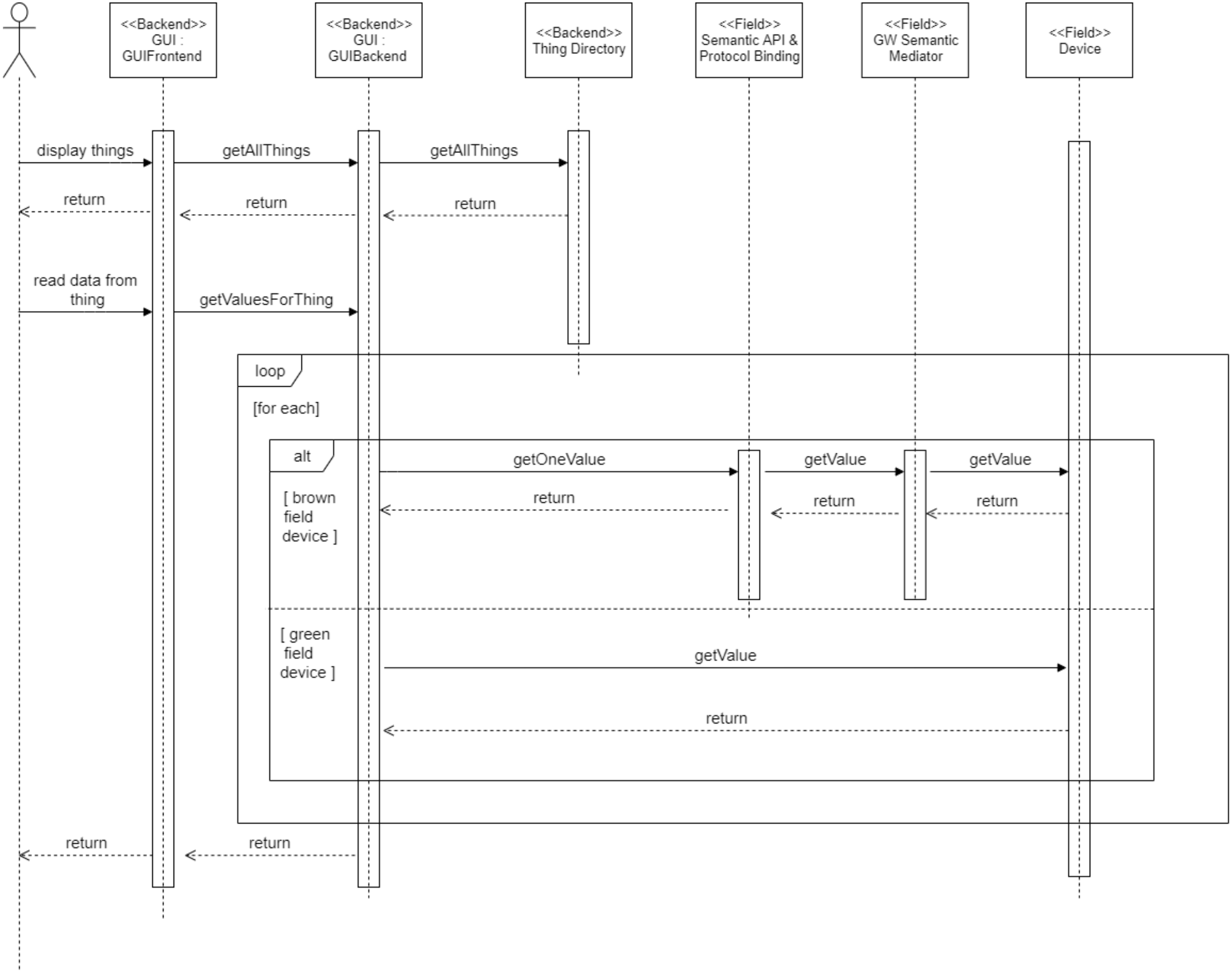


FIGURE 125 SEQUENCE DIAGRAM, READ REAL-TIME DATA FROM DEVICE

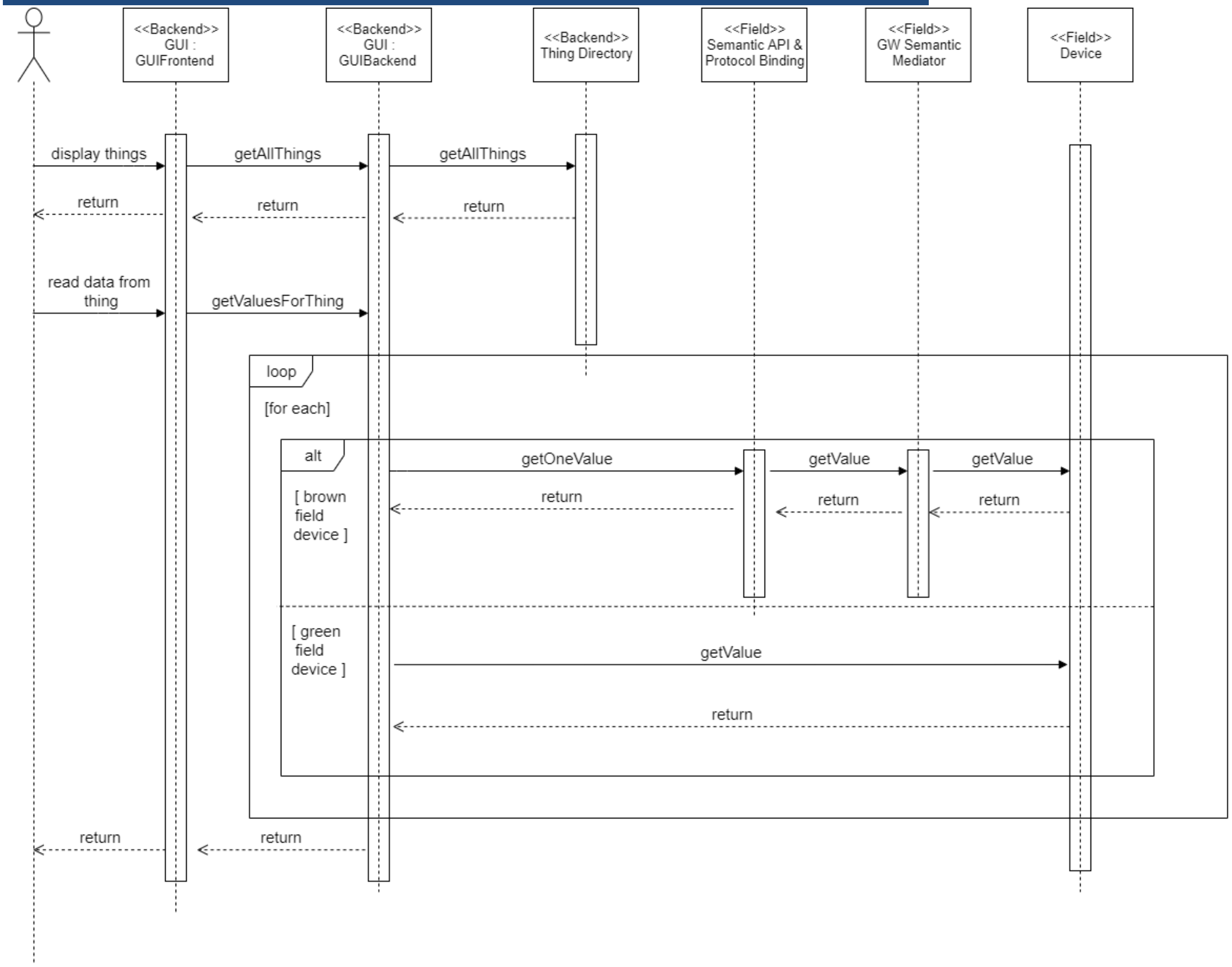


FIGURE 126 SEQUENCE DIAGRAM, ACTUATE ACTION

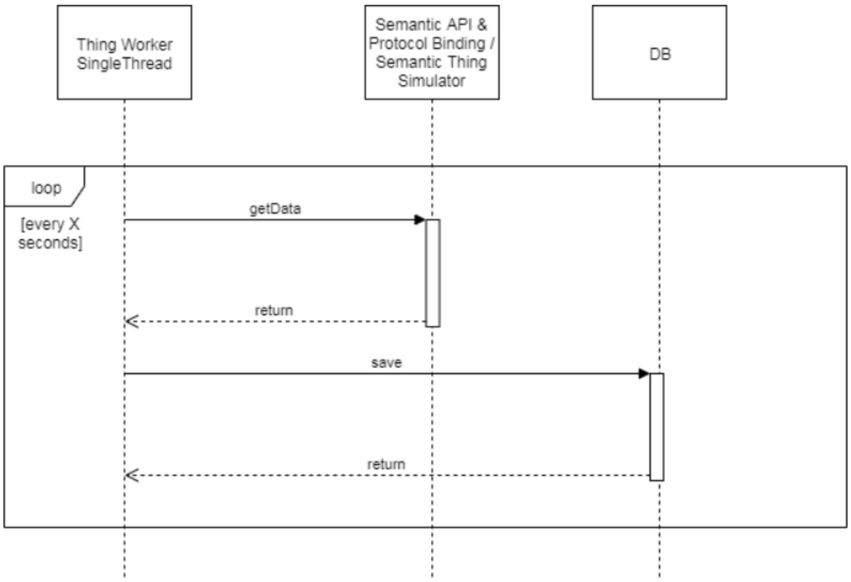

Thing Worker SingleThread
Semantic API & Protocol Binding / Semantic Thing Simulator
DB
loop
[every X seconds]
getData
return
save
return

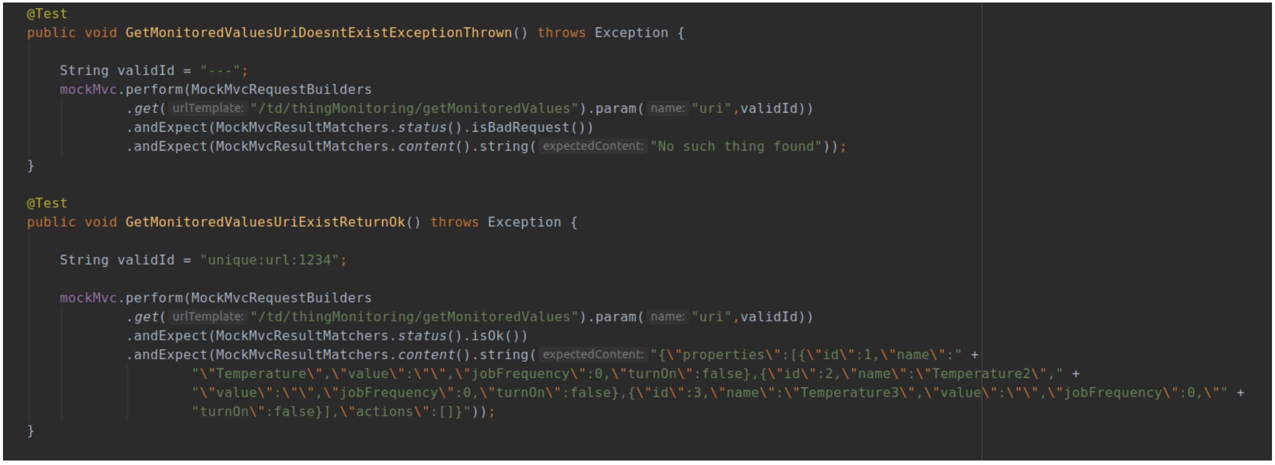
FIGURE 127 SEQUENCE DIAGRAM, COLLECT DATA FROM DEVICE

3.4.4.3.2 TEST

Table 19 Functional tests scenarios for Thing Simulator.

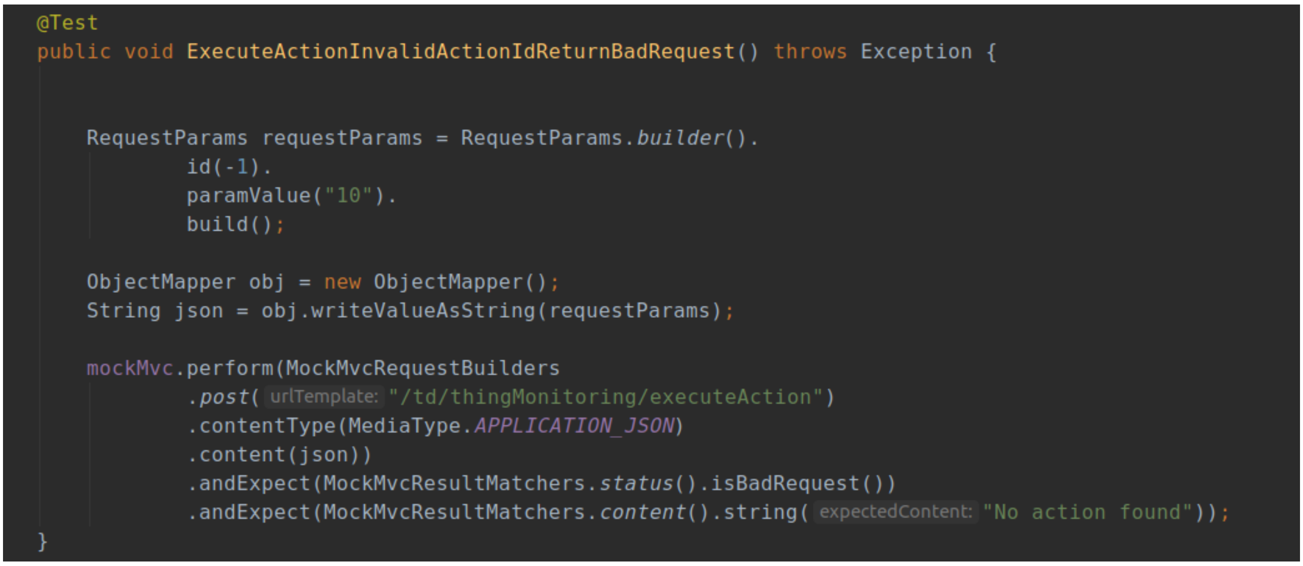
GUI Integration with Thing Simulator
/getMonitoredValues
✓ should return results in response when address is valid (with 200 OK status) ✓ should return null when address is invalid (with 200 OK status)
/executeAction
✓ should return results in response when address is valid (with 200 OK status) ✓ should return an error message in response when address is invalid or request body is invalid (with 400 Bad request status)

Functional tests have been made using Mockito library. The tests check GUI and Thing Simulator API. The integration with brownfield devices through Semantic API & Protocol Binding will be implemented in the next scope.



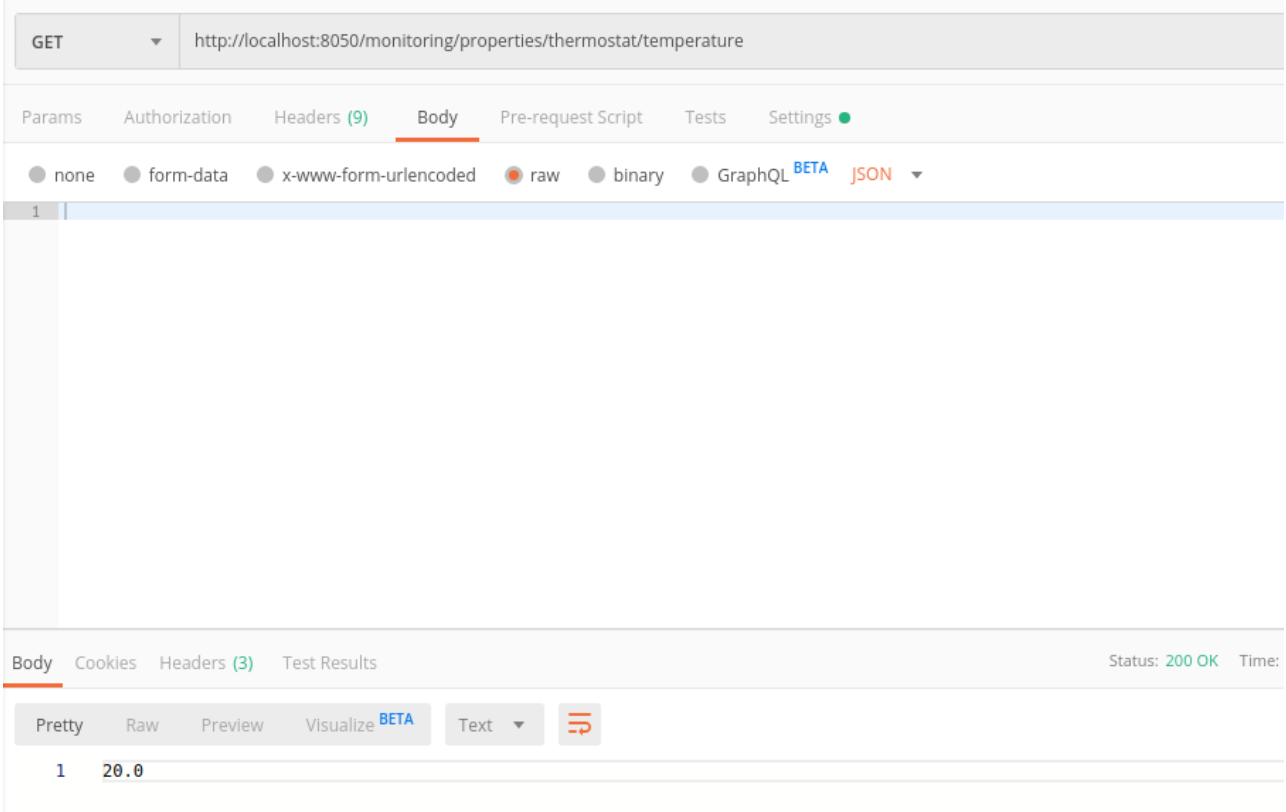
```
@Test
public void GetMonitoredValuesUriDoesntExistExceptionThrown() throws Exception {

    String validId = "---";
    mockMvc.perform(MockMvcRequestBuilders
            .get( urlTemplate: "/td/thingMonitoring/getMonitoredValues").param( name: "uri",validId))
            .andExpect(MockMvcResultMatchers.status().isBadRequest())
            .andExpect(MockMvcResultMatchers.content().string( expectedContent: "No such thing found"));
}

@Test
public void GetMonitoredValuesUriExistReturnOk() throws Exception {

    String validId = "unique:url:1234";

    mockMvc.perform(MockMvcRequestBuilders
            .get( urlTemplate: "/td/thingMonitoring/getMonitoredValues").param( name: "uri",validId))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.content().string( expectedContent: "{\"properties\":[{\"id\":1,\"name\":" +
                    "\"Temperature\",\"value\":\"\",\"jobFrequency\":0,\"turnOn\":false},{\"id\":2,\"name\":\"Temperature2\"," +
                    "\"value\":\"\",\"jobFrequency\":0,\"turnOn\":false},{\"id\":3,\"name\":\"Temperature3\",\"value\":\"\",\"jobFrequency\":0,\"" +
                    "turnOn\":false}],\"actions\":[]}"));
}
```

FIGURE 128:EXAMPLE OF API FUNCTIONAL TEST



```
@Test
public void ExecuteActionInvalidActionIdReturnBadRequest() throws Exception {


    RequestParams requestParams = RequestParams.builder().
            id(-1).
            paramValue("10").
            build();

    ObjectMapper obj = new ObjectMapper();
    String json = obj.writeValueAsString(requestParams);

    mockMvc.perform(MockMvcRequestBuilders
            .post( urlTemplate: "/td/thingMonitoring/executeAction")
            .contentType(MediaType.APPLICATION_JSON)
            .content(json))
            .andExpect(MockMvcResultMatchers.status().isBadRequest())
            .andExpect(MockMvcResultMatchers.content().string( expectedContent: "No action found"));
}
```

FIGURE 129: EXAMPLE OF API FUNCTIONAL TEST



GET http://localhost:8050/monitoring/properties/thermostat/temperature

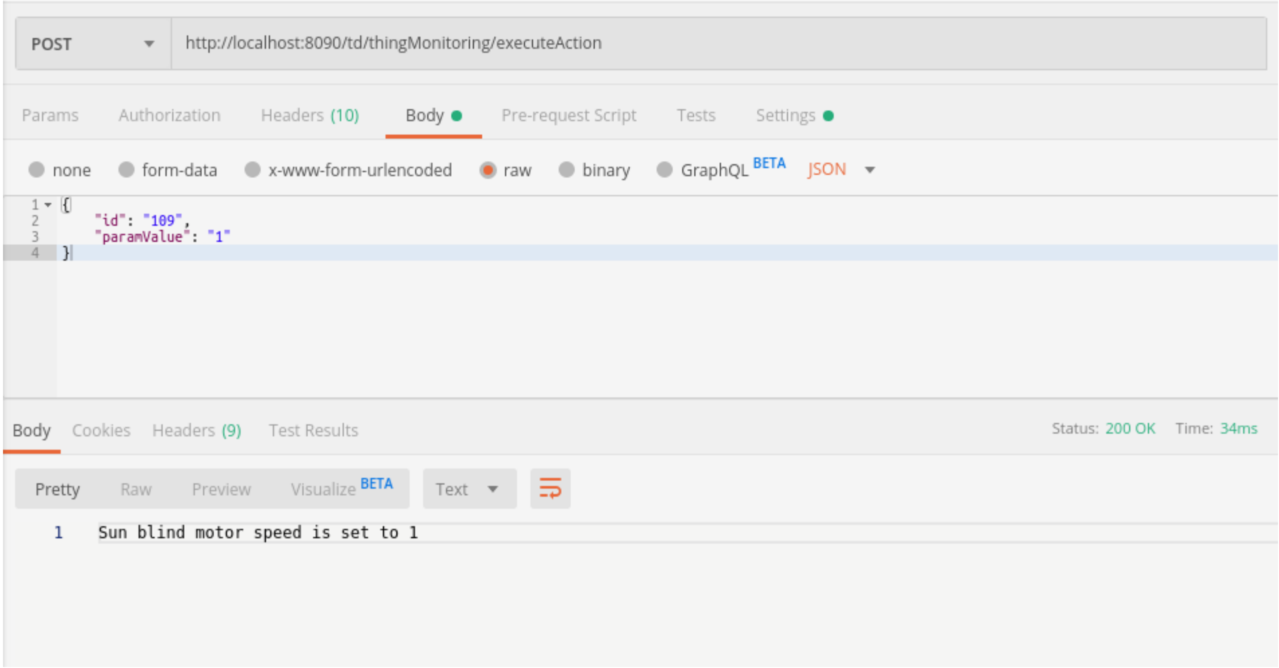
● raw JSON

Status: 200 OK

Text

```
20.0
```

FIGURE 130:EXAMPLE OF GUI - THING SIMULATOR INTEGRATION



POST http://localhost:8090/td/thingMonitoring/executeAction

● raw JSON

```
{
    "id": "109",
    "paramValue": "1"
}
```

Status: 200 OK Time: 34ms

Text

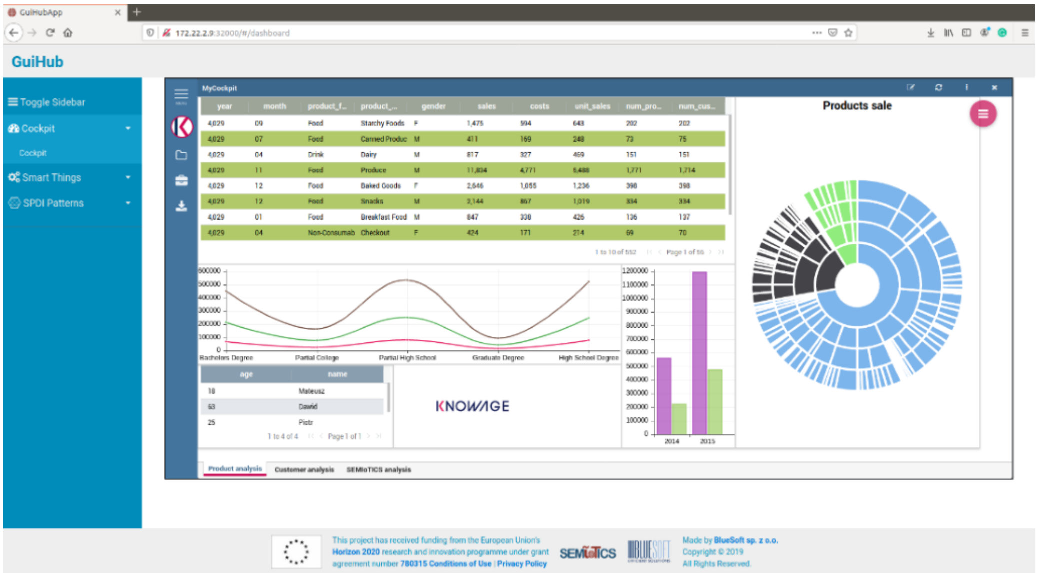
```
Sun blind motor speed is set to 1
```

FIGURE 131: EXAMPLE OF GUI - THING SIMULATOR INTEGRATION

3.4.4.4 DATA VISUALIZATIONS USING FIWARE

Integration GUI component with FIWARE Knowage was created to visualize all data collected from devices and sensors on extended dashboards. It allows the user to create an unlimited number of dashboards form a wide selection of ready-made widgets. Data on dashboards are used from datasets created during gathering data from sensors. Figures below (FIGURE 132, FIGURE 133) show two dashboards, first with random data to show Knowage capabilities and second based on data from SEMIoTICS.

.



year	month	product_f...	product_...	gender	sales	costs	unit_sales	num_pro...	num_cus...
4,029	09	Food	Starchy Foods	F	1,475	594	643	202	202
4,029	07	Food	Canned Produc	M	411	169	248	73	75
4,029	04	Drink	Dairy	M	817	327	469	151	151
4,029	11	Food	Produce	M	11,834	4,771	5,488	1,771	1,714
4,029	12	Food	Baked Goods	F	2,646	1,055	1,236	398	398
4,029	12	Food	Snacks	M	2,144	867	1,019	334	334
4,029	01	Food	Breakfast Food	M	847	338	426	136	137
4,029	04	Non-Consumab	Checkout	F	424	171	214	69	70

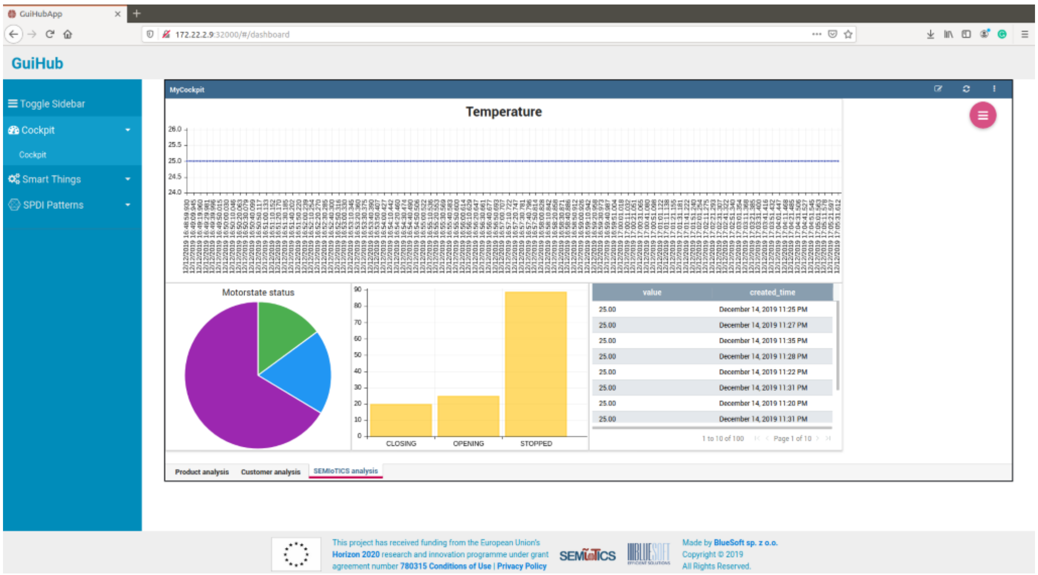
1 to 10 of 552 Page 1 of 56

age	name
18	Mateusz
53	Dawid
25	Piotr

1 to 4 of 4 Page 1 of 1

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy
Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 132 EXAMPLE OF THE DASHBOARD SHOWING COLLECTED DATA



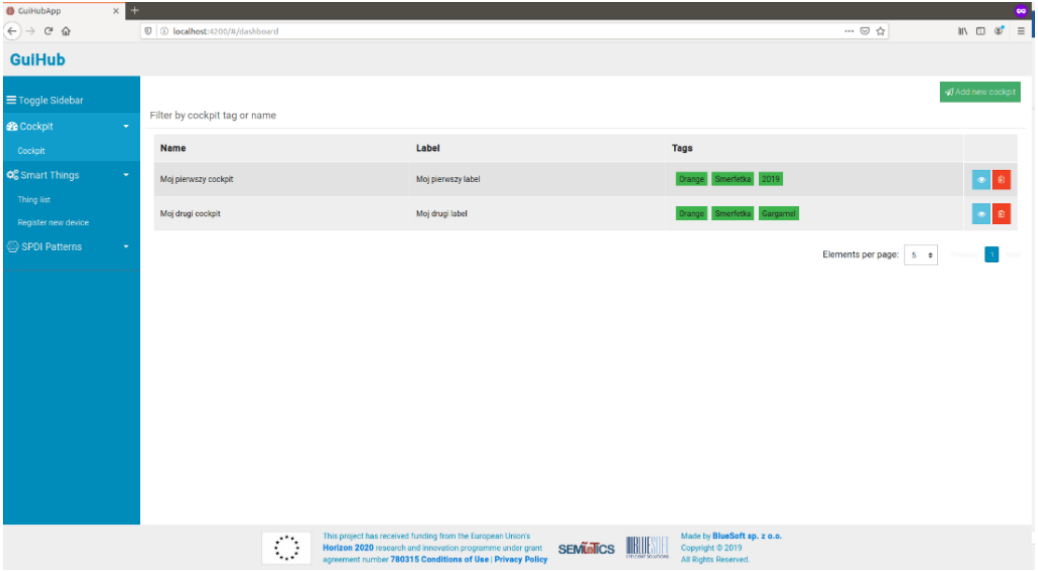
value	created_time
25.00	December 14, 2019 11:25 PM
25.00	December 14, 2019 11:27 PM
25.00	December 14, 2019 11:35 PM
25.00	December 14, 2019 11:28 PM
25.00	December 14, 2019 11:22 PM
25.00	December 14, 2019 11:31 PM
25.00	December 14, 2019 11:20 PM
25.00	December 14, 2019 11:31 PM

1 to 10 of 100 Page 1 of 10

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy
Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 133 EXAMPLE OF THE DASHBOARD SHOWING COLLECTED DATA FROM SENSORS

The user can create several dashboards, edit or delete them and list, as shown in the FIGURE 134.



Name	Label	Tags
Moj pierwszy cockpit	Moj pierwszy label	Orange Smerfetka 2019
Moj drugi cockpit	Moj drugi label	Orange Smerfetka Gargamel

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement number 780315 Conditions of Use | Privacy Policy

Made by BlueSoft sp. z o.o.
Copyright © 2019
All Rights Reserved.

FIGURE 134 LIST OF ALL DASHBOARDS

3.4.4.4.1 COMPONENT DIAGRAM

The architecture of integration with FIWARE (FIGURE 135) includes two components, GUI from the backend layer and Knowage also from the backend layer which is one of the FIWARE Generic Enablers. The main aim of this integration was to visualize all collected data on powerful and efficient Knowage dashboards. For this purpose, Knowage was deployed to the Backend Orchestrator and then embedded in GUI as an HTML frame that was detailed described in section 3.4.4. After installation, it was also necessary to configure Knowage to use data from the GUI database. Knowage provides multiple APIs to interact with them. List of APIs used by GUI component is presented below:

- Lista all dashboards
- Adds a new dashboard
- Update the dashboard
- Delete the dashboard
- Add a dataset
- Login

The first four abovementioned APIs are dedicated to supporting dashboards functionalities and allows them to manage dashboard by the GUI component. ‘Add a dataset’ API is used to create a dataset when the user starts collecting data from a sensor or device. The Last API enables to authorize the user in Knowage to get data according to the permission it has.

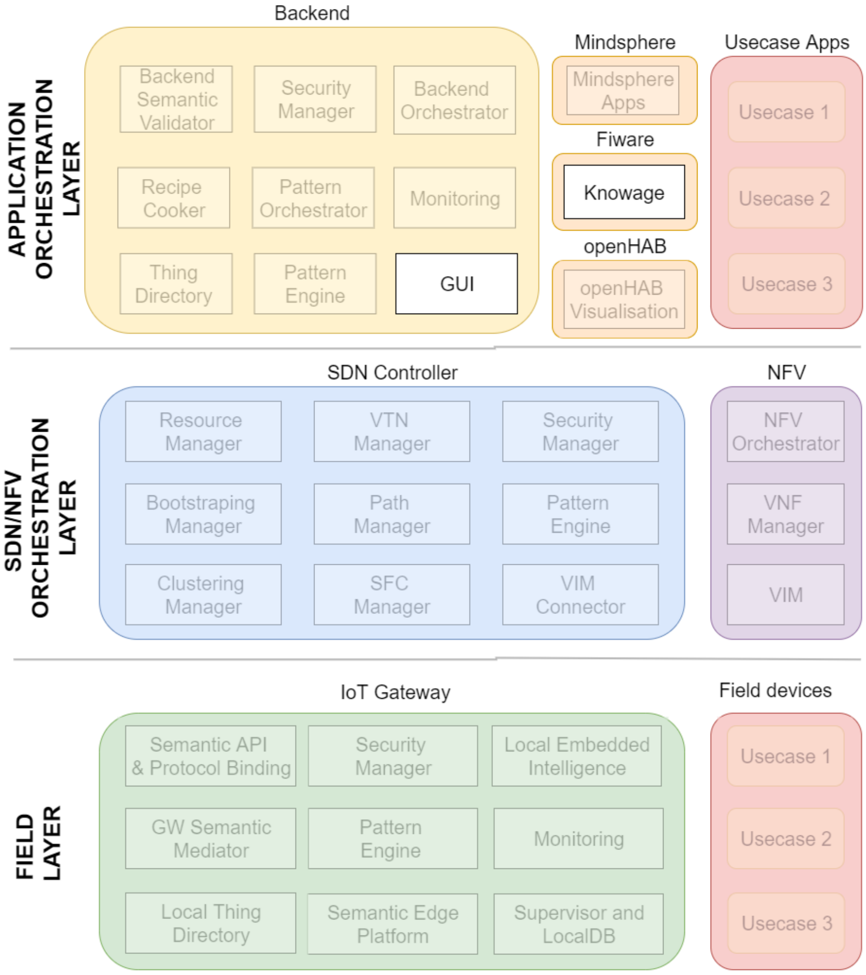


FIGURE 135 COMPONENTS RELATED TO THE DATA VISUALIZATION USING KNOWAGE

3.4.4.4.2 TESTS

Users Cockpit feature in SEMIoTICS platform is provided by FIWARE General Enabler named KNOWAGE. In order to integrate this component to GUI, its capabilities had to be tested. KNOWAGE API[19] has been tested with creating HTTP request with Postman.

Table 20 Functional tests scenarios for Knowage

Documents API
#GET /documents
✓ should reject with 401 error when an authorization credential is invalid ✓ should return JSON format file with documents(Cockpits) list
#GET /documents/document_label
✓ should reject with 401 error when an authorization credential is invalid ✓ should reject with 404 error when attempting to document data is not there ✓ should return JSON format file with cockpit details
#DELETE /documents/document_label

[19] https://knowage.docs.apiary.io/#introduction/errors

✓ ✓ should reject with 401 error when an authorization credential is invalid ✓ should reject with 404 error when attempting to document data is not there ✓ should delete a document by label
#UPDATE /documents/document_label
✓ should reject with 401 error when an authorization credential is invalid ✓ should reject with 404 error when attempting to document data is not there ✓ should update a document by label
Dataset API
#GET /datasets
✓ should reject with 401 error when an authorization credential is invalid ✓ should return JSON format file with datasets list
#GET /datasets/dataset_label
✓ should reject with 401 error when an authorization credential is invalid ✓ should reject with 404 error when attempting to document data is not there ✓ should return JSON format file with datasets details
#DELETE /datasets/dataset_label
✓ ✓ should reject with 401 error when an authorization credential is invalid ✓ should reject with 404 error when attempting to document data is not there ✓ should delete a dataset by label
#UPDATE /datasets/dataset_label
✓ should reject with 401 error when an authorization credential is invalid ✓ should reject with 404 error when attempting to document data is not there ✓ should update a dataset by label

3.5 Service Function Chaining

3.5.1 COMPONENT ARCHITECTURE

Figure 136 below depicts the SFC related components distributed among the layers of the SEMIoTICS architecture. Pattern Orchestrator and one of the three Pattern Engines are located at the Application Orchestration Layer, at the Backend. Also at the SDN/NFV orchestration layer, the NFV orchestrator, VNF Manager and VIM are located.

As already described in D5.5 section 4.2, Pattern Orchestrator is responsible for automated configuration, coordination, and management of different patterns and their deployment. Moreover, it is used for processing a received requirement in order to translate it to Drools facts. The result of the said processing enables the Pattern Orchestrator to choose which Pattern Engine should receive the corresponding Drools facts in order to reason with them.

Pattern Engine at the application layer, includes the patterns in the form of Drools rules, responsible for verifying and instantiating VNFs and SFCs. With the said rules the Pattern Engine is able to reason whether a new VNF is needed to be instantiated in order to complete a specific SFC. Moreover if an SFC is not instantiated at all, the rules allow the instantiation of the SFC after having gathered all the necessary VNFs. Due to the fact that the Pattern Engine at the application layer, has a global view of the pattern facts across all layers, it makes it appropriate for providing up to date information on a graphical interface designed specifically for SFC management in a higher level. This interface is called SFC GUI and extracts all the information depicted directly from the Pattern Engine.

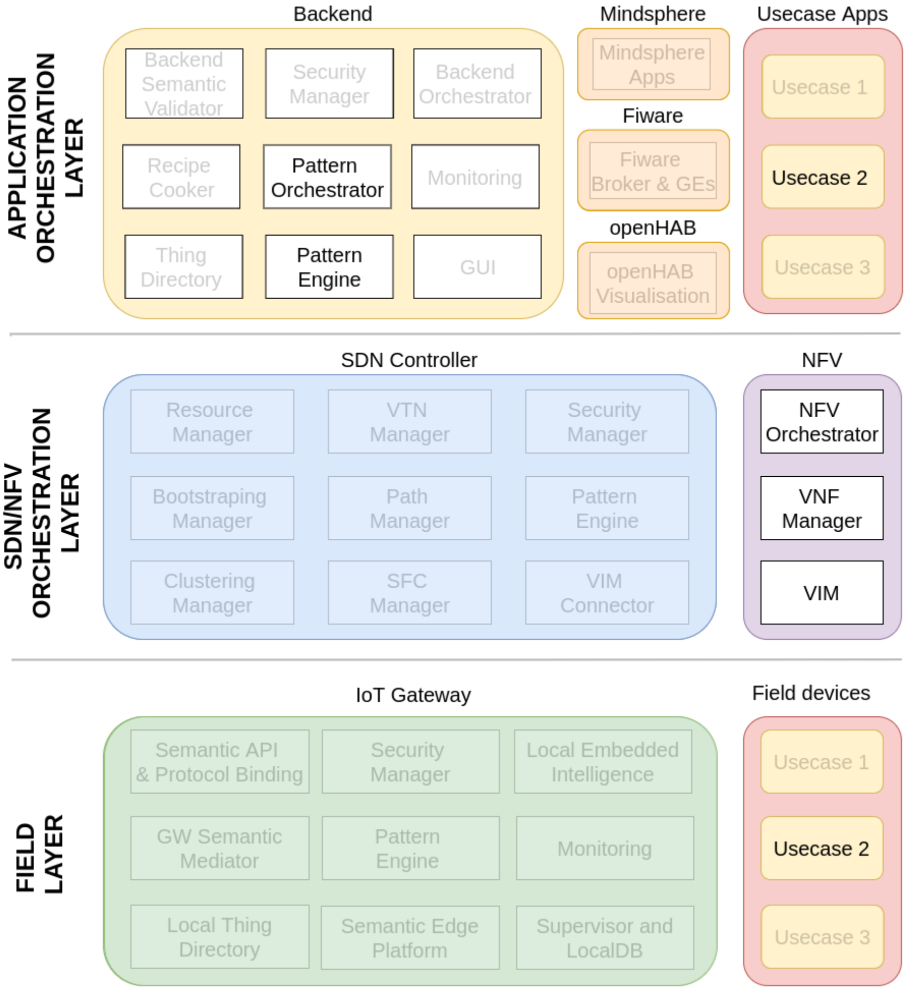
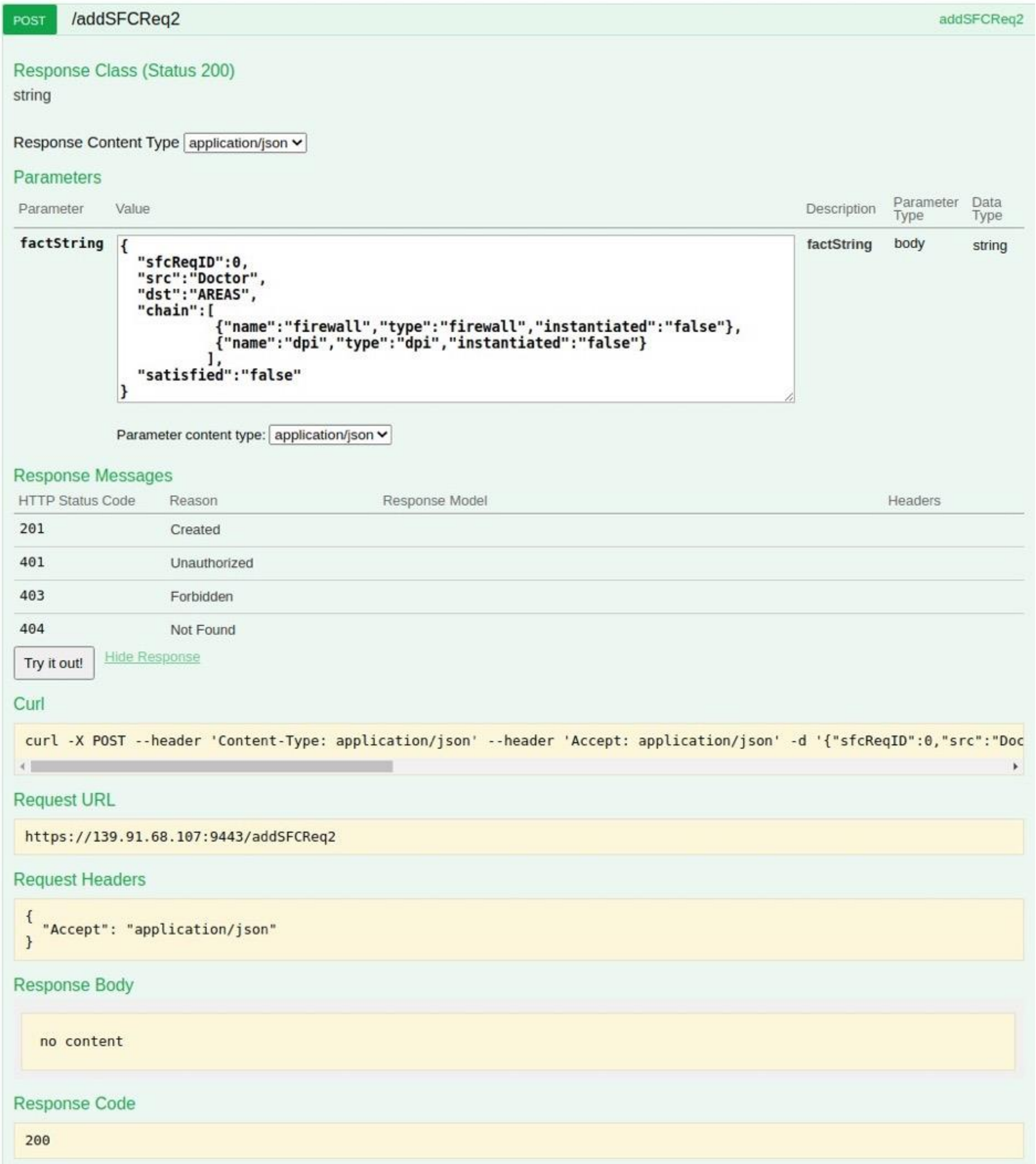


Figure 136 SFC Related Components in SEMIoTICS Architecture

The NVF related components are already described in section 3.2

3.5.2 APIS

Two dedicated REST APIs (and Figure 138) have been developed at the Backend Pattern Engine to facilitate and automate operations of the SFC. The first endpoint (Figure 137), is used by the SFC GUI in order to create and sent to the Pattern Engine a new Pattern Requirement which represents a service function chain.



Response Class (Status 200)

string

Response Content Type application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
factString		factString	body	string

```
{
   "sfcReqID":0,
   "src":"Doctor",
   "dst":"AREAS",
   "chain":[
            {"name":"firewall","type":"firewall","instantiated":"false"},
            {"name":"dpi","type":"dpi","instantiated":"false"}
           ],
   "satisfied":"false"
}
```

Parameter content type: application/json

Response Messages

HTTP Status Code	Reason	Response Model	Headers
201	Created		
401	Unauthorized		
403	Forbidden		
404	Not Found		

Curl

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: application/json' -d '{"sfcReqID":0,"src":"Doc
```

Request URL

```
https://139.91.68.107:9443/addSFCReq2
```

Request Headers

```
{
  "Accept": "application/json"
}
```

Response Body

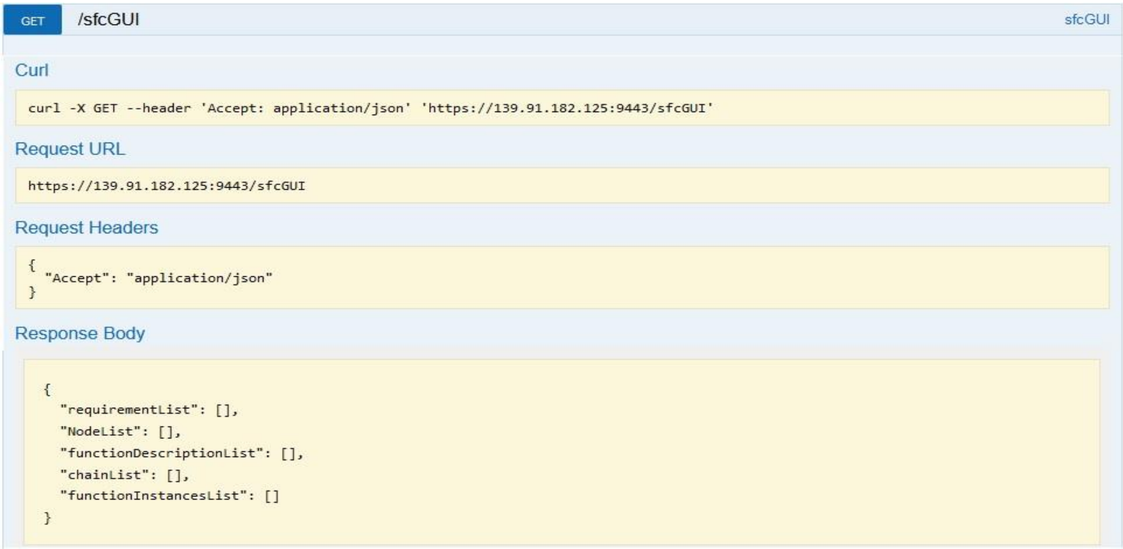
```
no content
```

Response Code

```
200
```

Figure 137 PATTERN ENGINE REST FOR SUBMIT BUTTON OF SFC GUI

The second endpoint (Figure 138), is used by the SFC GUI to enable the visualization of the information gathered from the Pattern Engine.



Curl

```
curl -X GET --header 'Accept: application/json' 'https://139.91.182.125:9443/sfcGUI'
```

Request URL

```
https://139.91.182.125:9443/sfcGUI
```

Request Headers

```
{
  "Accept": "application/json"
}
```

Response Body

```
{
  "requirementList": [],
  "NodeList": [],
  "functionDescriptionList": [],
  "chainList": [],
  "functionInstancesList": []
}
```

FIGURE 138 PATTERN ENGINE REST FOR SFC GUI

3.5.3 TESTING METHODOLOGY

In order to test the functionalities of all the SFC related components of SEMIoTICS, as they are described above, we use the Proxmox Virtual environment to create Virtual Machines (VMs) hosted in an INTEL NUC (Figure 139) with 32GB RAM, 500GB storage and a CPU i7-6770HQ 2.60GHz with 8 cores. Additionally a VPN connection is established with CCTC's premises in order to have access to the NFV related components.
The created VMs have some hardware and software requirements, which are shown in the table below.

Table 21

Component	Software	CPU	Memory	Disk
Pattern Orchestrator	Ubuntu 18.04 LTS	4 cores	4 GB	10 GB
Backend Pattern Engine	Ubuntu 18.04 LTS	4 cores	4 GB	10 GB
SFC GUI	Ubuntu 18.04 LTS	4 cores	2 GB	20 GB

Figure 139 **Intel NUC**

The approach followed by this work, is the dynamic instantiation of SFCs based on the predefined SFC patterns. When there is a request for an SFC instantiation containing service functions, the depicted in Figure 140 procedure should be followed. If the SFC does not exist, the instantiation of the respective SFC is deployed through the identification of the requested VNFs. If the VNFs exist in the service nodes, the SFC is updated including these VNFs. If the VNFs do not exist, the service node with the available resources is requested to instantiate the respective VNFs. The procedure is ended when all the requested VNFs are included in the SFC.

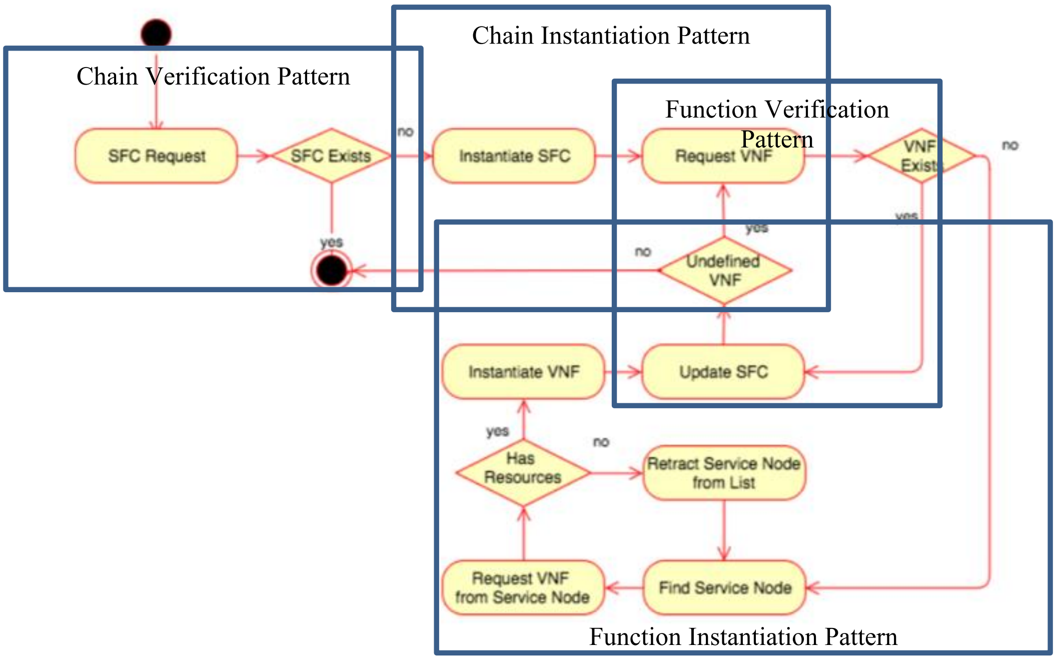

Chain Instantiation Pattern
Chain Verification Pattern
Function Verification Pattern
SFC Request
SFC Exists
no
yes
Instantiate SFC
Request VNF
VNF Exists
no
yes
yes
no
Undefined VNF
Instantiate VNF
Update SFC
yes
Has Resources
no
Retract Service Node from List
Request VNF from Service Node
Find Service Node
Function Instantiation Pattern

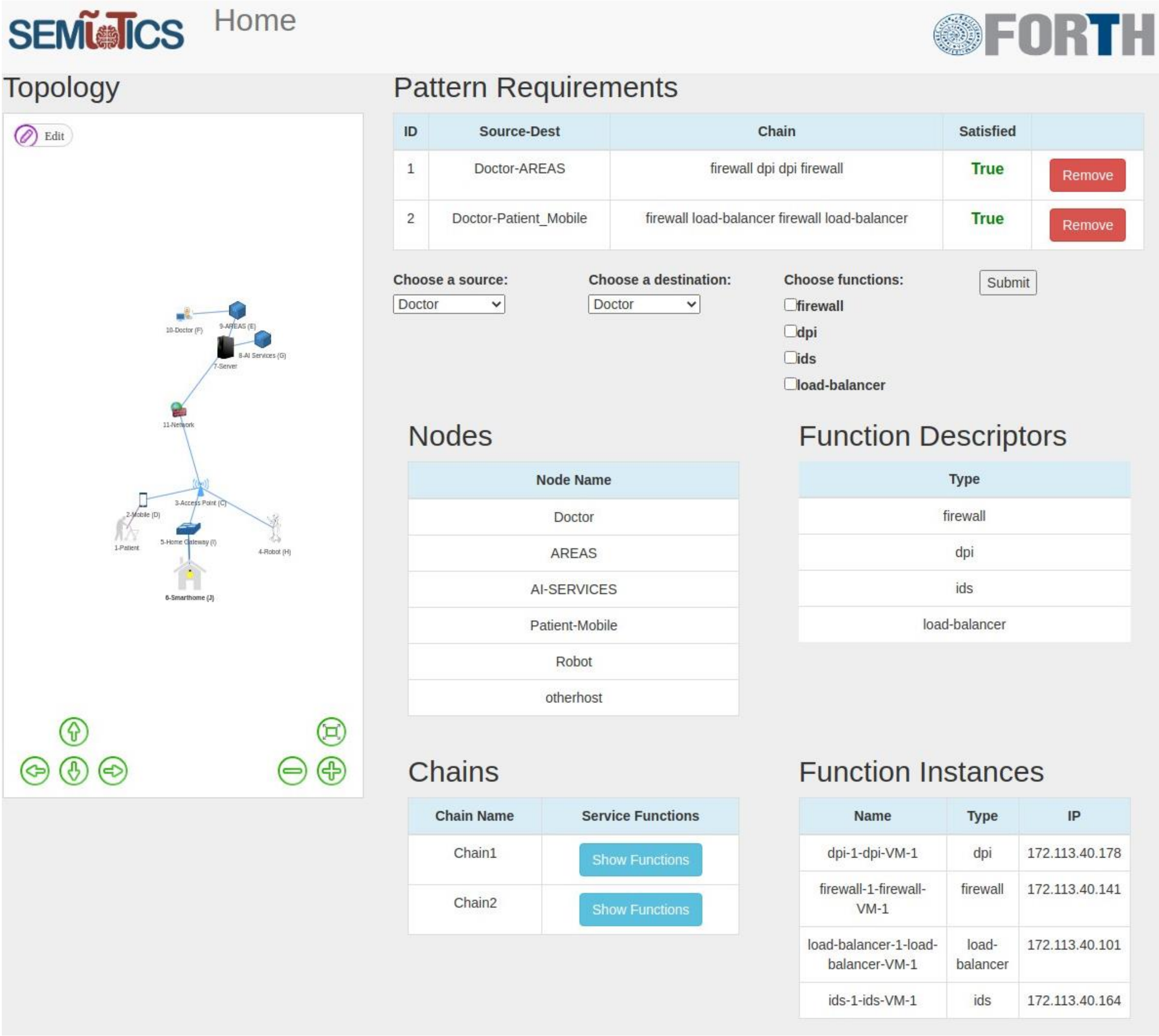
FIGURE 140 VNF INSTANTIATION BASED ON SFC REQUEST

3.5.4 PERFORMANCE TESTING & KPI VALIDATION

All information for the testing is depicted in SFC GUI (Figure 141) which visualizes the data from Backend Pattern Engine with a simple http-GET request (Figure 138). Backend Pattern Engine gathers information from:

- Thing Directory (Nodes)
- OSM (Function Descriptors and Function Instances)
- SFC Requests (Chains)
- Pattern Requirement (user input)

The user requests for specific NFVs to be applied in a flow of a specific source and destination. The NFVs that the user can choose from, is dynamically populated with information provided by the OSM. Initially we choose as source the Doctor, destination the AREAS, a NFV of a firewall and an NFV of a dpi. Upon clicking on the submit button a new Pattern Requirement is created that is sent directly to the Pattern Engine and is consumed by the addSFCReq endpoint. After the new requirement has been consumed by the Pattern Engine, the pattern rules are fired causing the verification and instantiation of VNFs and SFCs (**FIGURE 142**)



Pattern Requirements

ID	Source-Dest	Chain	Satisfied	
1	Doctor-AREAS	firewall dpi dpi firewall	True	Remove
2	Doctor-Patient_Mobile	firewall load-balancer firewall load-balancer	True	Remove

Choose a source: Doctor

Choose a destination: Doctor

Choose functions:
☐ firewall
☐ dpi
☐ ids
☐ load-balancer

Submit

Nodes

Node Name
Doctor
AREAS
AI-SERVICES
Patient-Mobile
Robot
otherhost

Function Descriptors

Type
firewall
dpi
ids
load-balancer

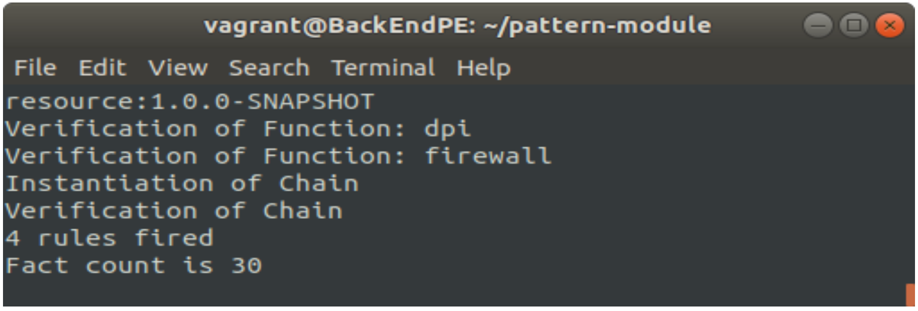
Chains

Chain Name	Service Functions
Chain1	Show Functions
Chain2	Show Functions

Function Instances

Name	Type	IP
dpi-1-dpi-VM-1	dpi	172.113.40.178
firewall-1-firewall-VM-1	firewall	172.113.40.141
load-balancer-1-load-balancer-VM-1	load-balancer	172.113.40.101
ids-1-ids-VM-1	ids	172.113.40.164

FIGURE 141 SFC GUI


vagrant@BackEndPE: ~/pattern-module
File Edit View Search Terminal Help

```
resource:1.0.0-SNAPSHOT
Verification of Function: dpi
Verification of Function: firewall
Instantiation of Chain
Verification of Chain
4 rules fired
Fact count is 30
```

FIGURE 142 PATTERN ENGINE FIRING SFC RELATED RULES

The requirement is immediately satisfied due to the pattern rules, and that is depicted in the SFC GUI. Similarly we choose as source the Doctor, destination the Patient_Mobile, an NFV of a firewall and an NFV of a load balancer.

The described implementation above contributes to the fulfillment of R.GP.3, R.GP.4 R.UC2.3 of project requirements as well as KPI-1.1, KPI-1.2, KPI-2.1, KPI-3.1, KPI-6.1, KPI-5.2.

4 USE-CASE SPECIFIC DEMONSTRATORS

The following demonstrators, that showcase use-case specific functionalities, were presented as part of the Mid-Term Review.

4.1 Use Case 1 demonstrator

A demonstration scenario that relies on the SEMIoTICS pattern-driven network interface and its capabilities was designed and developed around Use Case 1, i.e. industrial IoT environments, and more specifically oil leakage detection in wind turbines through video monitoring. This was also demonstrated during the Mid-Term Review. The overarching aim of the scenario is to distribute a complex application (composed of multiple tasks) to a network of IoT/Edge device and specify constraints (through patterns) on the network / orchestration. In this context, the developed scenario also leverages user-friendly design and deployment of IoT orchestrations through a custom-built, distributed version of Node-RED[20]. The two key research innovation of the scenario and associated demonstration relate to: 1) True distribution of application flows over multiple devices and representing the network perspective in Node-RED, and; 2) Automated enforcement of network / orchestration constraints by defining them as SEMIoTICS patterns.

In terms of the actual setup, it involves transmission of video between two Raspberry Pi credit-card sized embedded devices (from “piA” to “piB”), coordinated by Node-RED running on a Nanobox (industrial PC), while monitoring of QoS constraints with patterns. This setup is depicted in FIGURE 143.

[20] https://nodered.org/

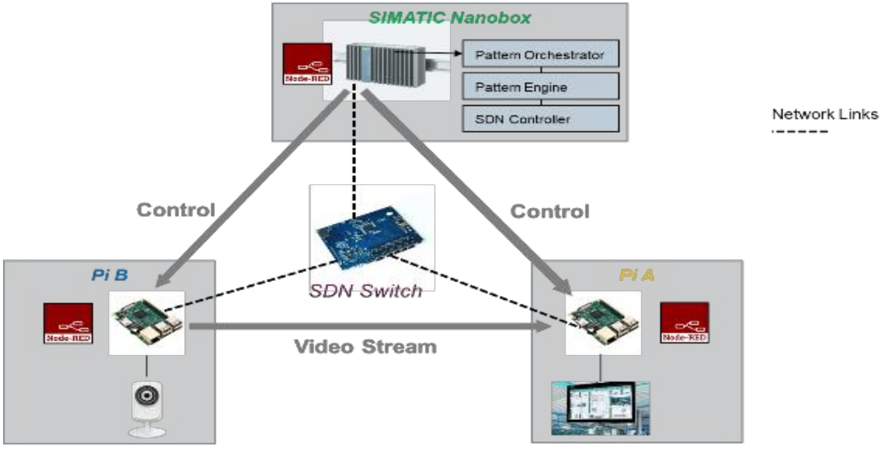


FIGURE 143: PATTERN-ENABLED IOT ORCHESTRATIONS LEVERAGING THE PATTERN-DRIVEN NETWORK INTERFACE

In the above, other than the user-friendly, graphical interface and distributed nature of defining the IoT orchestrations involved (including where / on which devices parts of a flow are deployed), we also want to define SPDI and QOS between these deployments (see FIGURE 144 and FIGURE 145).

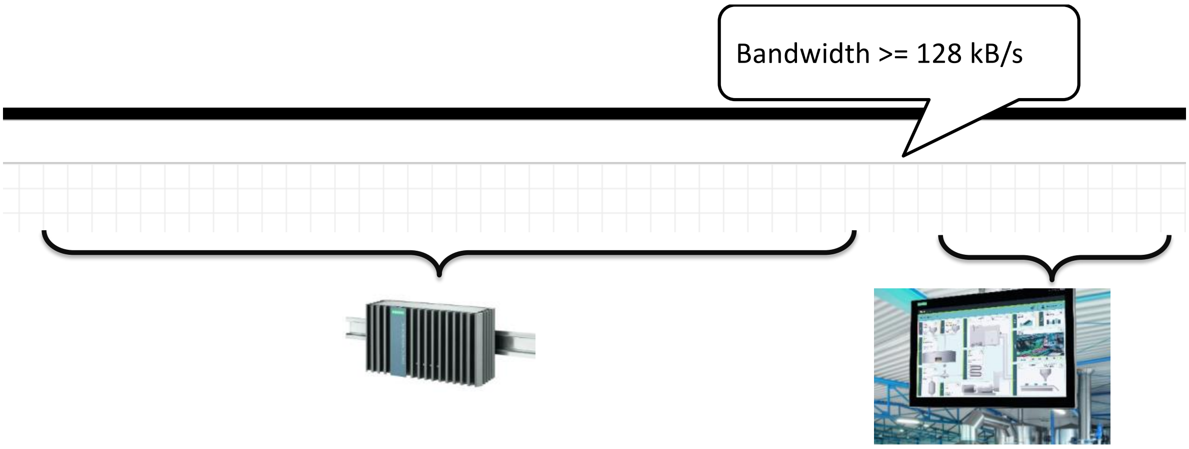
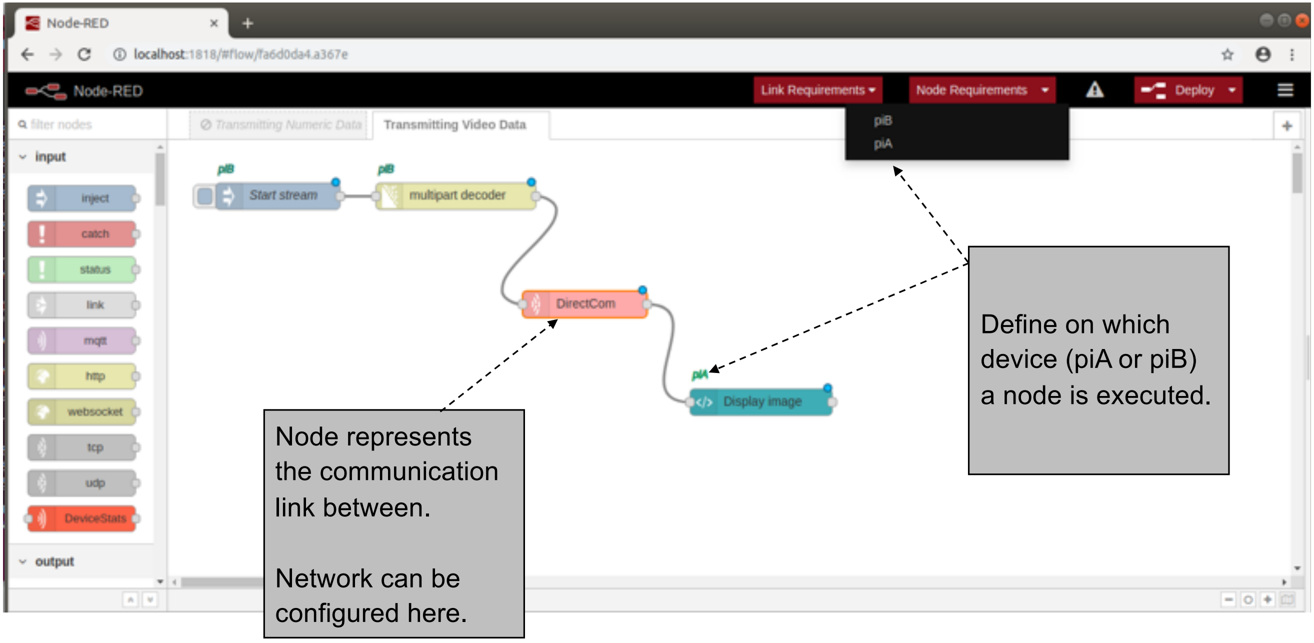


FIGURE 144. GRAPHICAL IOT ORCHESTRATION DEFINITION



Define on which device (piA or piB) a node is executed.

Node represents the communication link between.

Network can be configured here.

FIGURE 145: THE CUSTOMISED NODE-RED GUI AND SCENARIO ORCHESTRATION DEFINITION

Focusing on the network aspects, while maintaining the high-level abstractions needed for user-friendliness, a “Network Link” node enables direct communication between distributed Node-RED instances. Said “Network Link” node enables definition of QoS constraints (e.g., minimum bandwidth, latency) and the whole orchestration specification (a “Recipe”) and the QoS constraints are translated into the SEMIoTICS pattern language and sent to Pattern Orchestrator. From the latter, the information is relayed to the network (SDN) Pattern Engine. A high-level view of this process is shown in FIGURE 146.

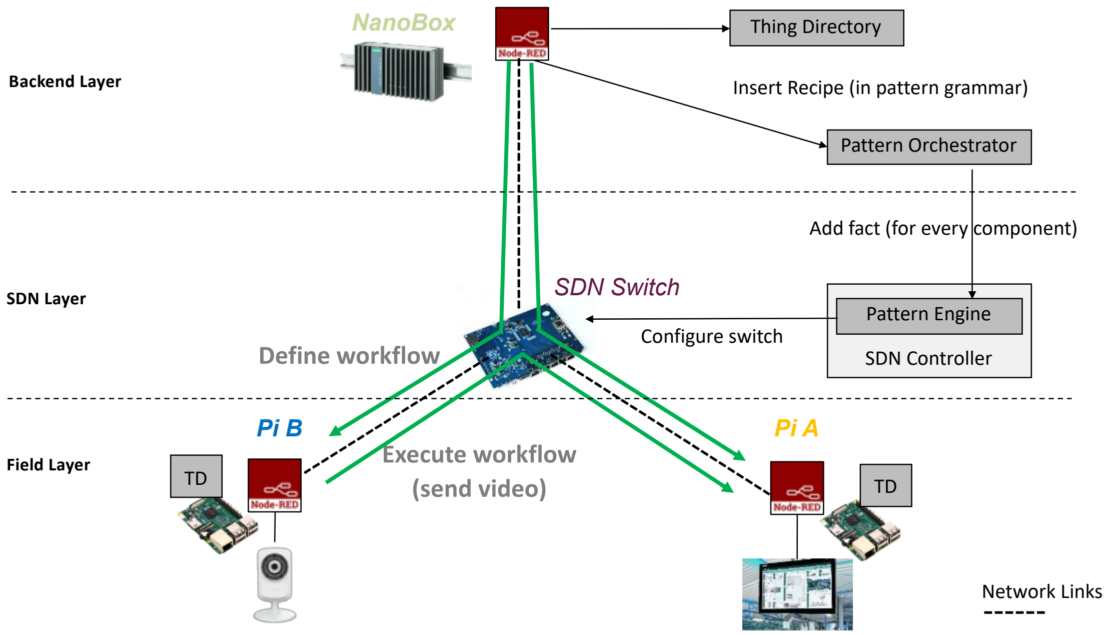


FIGURE 146: HIGH LEVEL VIEW OF SCENARIO IMPLEMENTATION SEQUENCE AND INVOLVED COMPONENTS

4.2 Use Case 2 demonstrator

E-health monitoring systems situated at homes can facilitate the monitoring of patients' activities and enable the remote provision of healthcare services. They improve the quality of elder population well-being in a non-obtrusive way, allowing greater independence, maintaining good health, preventing social isolation for individuals and delay their placement in institutions such as nursing homes and hospitals. In this context, the second use case of SEMIoTICS focuses on an ambient assisted living scenario, whereby a smart home environment.

Part of the SEMIoTICS testbed was demonstrated during the Mid-Term review. The main focus of this demo was to provide an extension of the current SARA use case where the SEMIoTICS framework can be applied in order to support the following service function chaining to guarantee security and dependability based on the defined Security, Privacy, Dependability and Interoperability (SPDI) patterns instantiating the required i) Virtual Network Functions (VNFs) and ii) SFC for assuring the SPDI requirements. Traffic classification is based on the predefined SFC for providing secure chains to forward the different kind of traffic of this use case. The procedure of instantiation and the identification of the respective SFCs and the VNFs based on the patterns is depicted in the Figure below:

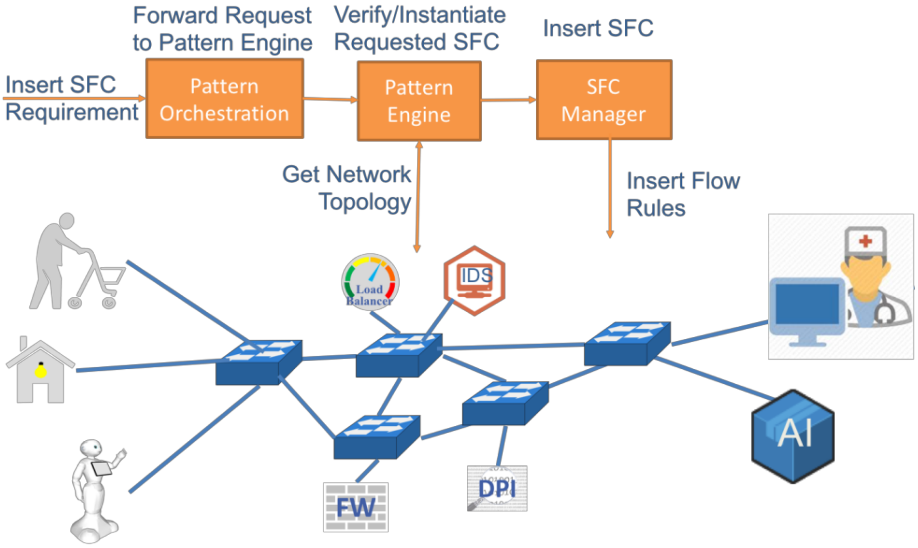
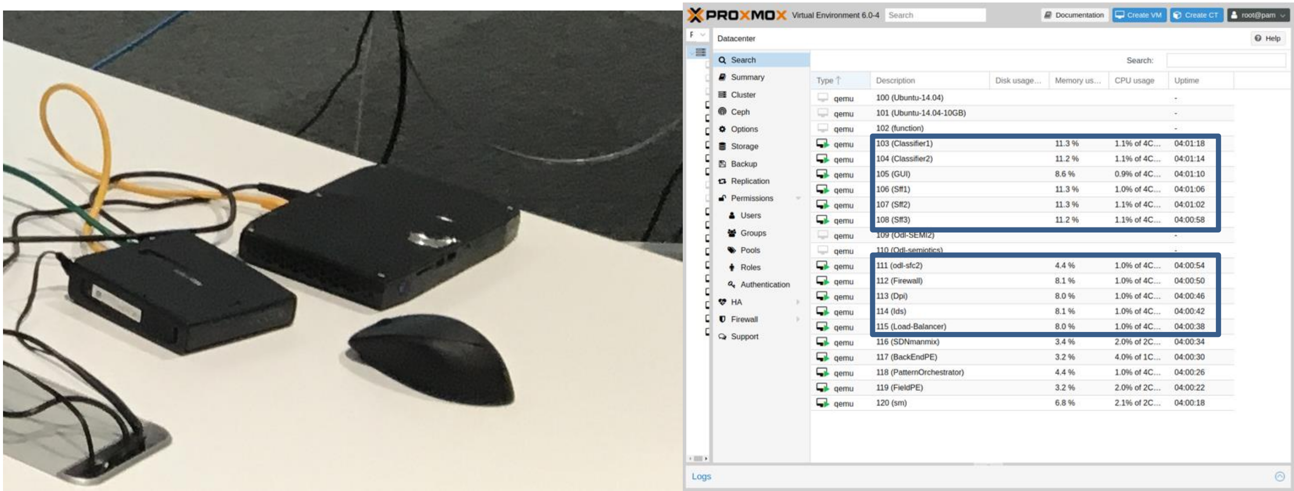


FIGURE 147 INSTANTIATION OF VNFS AND SFC

This demo demonstrated the use case in the following steps:

1) Present the legacy use case as the starting point.
2) Attach this use case to the SEMIoTICS architecture.
3) Install intermediate switches to forward traffic.
4) Identify and/or instantiate VNFs (virtually or physically) attached to the respective switches through the Pattern Engine in the Backend.
5) Instantiate SFCs based on the respective VNFs through the Pattern Engine in the SDN Controller
6) Active demonstration of the proactive control flow instantiation and data traffic classification in the developed network emulator.

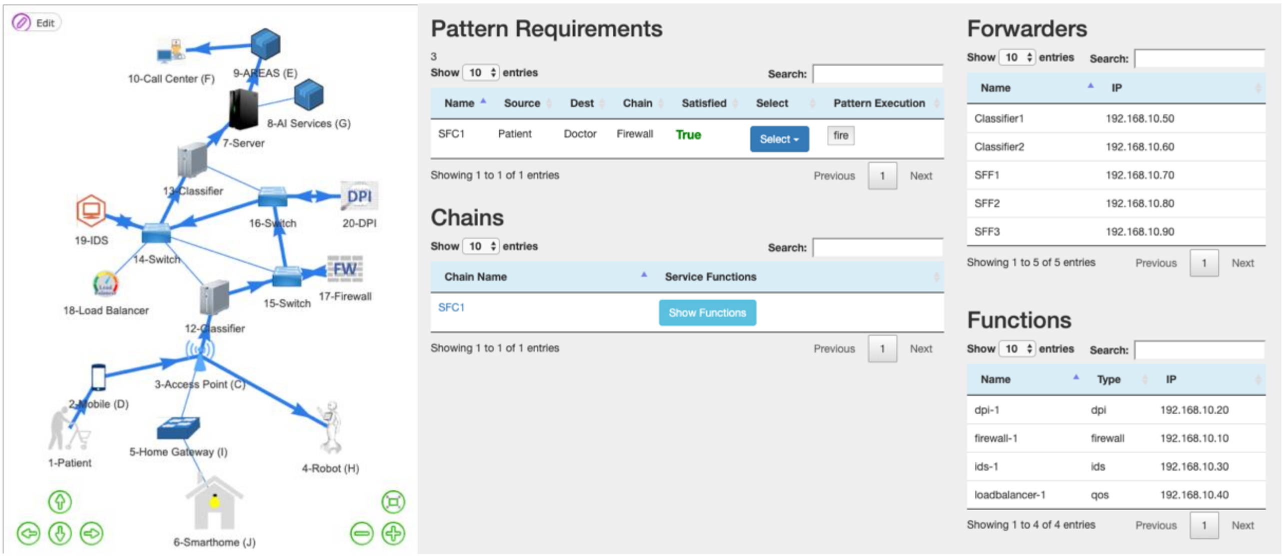
The setup of the testbed was based in the Proxmox where the different VMs hold the different service functions. (firewall, ids, dpi, load balancer), the virtual switches (Classifier1, Classifier2, SFF1, SFF2, SFF3) and the SDN controller as can be seen in Figure 148.



Type	Description	Disk usage...	Memory us...	CPU usage	Uptime
qemu	100 (Ubuntu-14.04)				-
qemu	101 (Ubuntu-14.04-10GB)				-
qemu	102 (function)				-
qemu	103 (Classifier1)		11.3 %	1.1% of 4C...	04:01:18
qemu	104 (Classifier2)		11.2 %	1.1% of 4C...	04:01:14
qemu	105 (GUI)		8.6 %	0.9% of 4C...	04:01:10
qemu	106 (Sff1)		11.3 %	1.0% of 4C...	04:01:06
qemu	107 (Sff2)		11.3 %	1.1% of 4C...	04:01:02
qemu	108 (Sff3)		11.2 %	1.1% of 4C...	04:00:58
qemu	109 (Odl-SEMI2)				-
qemu	110 (Odl-semiotics)				-
qemu	111 (odl-sfc2)		4.4 %	1.0% of 4C...	04:00:54
qemu	112 (Firewall)		8.1 %	1.0% of 4C...	04:00:50
qemu	113 (Dpi)		8.0 %	1.0% of 4C...	04:00:46
qemu	114 (Ids)		8.1 %	1.0% of 4C...	04:00:42
qemu	115 (Load-Balancer)		8.0 %	1.0% of 4C...	04:00:38
qemu	116 (SDNmanmix)		3.4 %	2.0% of 2C...	04:00:34
qemu	117 (BackEndPE)		3.2 %	4.0% of 1C...	04:00:30
qemu	118 (PatternOrchestrator)		4.4 %	1.0% of 4C...	04:00:26
qemu	119 (FieldPE)		3.2 %	2.0% of 2C...	04:00:22
qemu	120 (sm)		6.8 %	2.1% of 2C...	04:00:18

FIGURE 148 SFC TESTBED IN THE MIDTERM REVIEW

Under this test-bed, the SFC deployment of the SEMIoTICS framework in the use case 2 was presented (Figure 149).



Pattern Requirements

3
Show 10 entries Search:

Name	Source	Dest	Chain	Satisfied	Select	Pattern Execution
SFC1	Patient	Doctor	Firewall	True	Select	fire

Showing 1 to 1 of 1 entries Previous 1 Next

Chains

Show 10 entries Search:

Chain Name	Service Functions
SFC1	Show Functions

Showing 1 to 1 of 1 entries Previous 1 Next

Forwarders

Show 10 entries Search:

Name	IP
Classifier1	192.168.10.50
Classifier2	192.168.10.60
SFF1	192.168.10.70
SFF2	192.168.10.80
SFF3	192.168.10.90

Showing 1 to 5 of 5 entries Previous 1 Next

Functions

Show 10 entries Search:

Name	Type	IP
dpi-1	dpi	192.168.10.20
firewall-1	firewall	192.168.10.10
ids-1	ids	192.168.10.30
loadbalancer-1	qos	192.168.10.40

Showing 1 to 4 of 4 entries Previous 1 Next

FIGURE 149 SERVICE FUNCTION CHAINING IN USE CASE 2 TOPOLOGY

This use case also leverages CTTC's NFV testbed to host the VNFs that process the traffic stemming from the SARA's IoT GW. Moreover, the NFV testbed interacts with the SEMIoTICS pattern orchestrator as well, which indicates how the incoming traffic is processed by the SFCs according to the inferred patterns. To allow the external interaction with the NFV tested, and the use of its services, a VPN has been deployed, as detailed in Section 3.2.3.3. Also, a REST northbound interface (NBI) is enabled by the NFV testbed, which allows the pattern orchestrator to manage the NFV Orchestrator, i.e. onboarding and lifecycle control of VNF and NS. The integration tests to validate de VPN connection and the REST NBI are provided above in sections 2.2.3 and 3.2.3.3.

Also, it is worth mentioning that the NFV blocks are implemented in CTTC's NFV testbed by using the next software:

- OSM (Open Source MANO) [21]. It implements the NFVO along with the VNFM blocks.
- OpenStack[22]. The OpenStack controller implements the VIM, the OpenStack compute node implement the NFVI.

The NFV testbed has the topology described in **FIGURE 150**. The NFV MANO consists of two functional blocks. On the one hand, the OSM implements the NFVO and VNFM. On the other hand, the NFV MANO contains another sub block, the OpenStack controller node, which implements the VIM. The NFVI is composed of one functional block, an OpenStack compute node that exposes its virtualized resources to instantiate the VNFs. A switch is leveraged for the communications between all the functional blocks mentioned above.

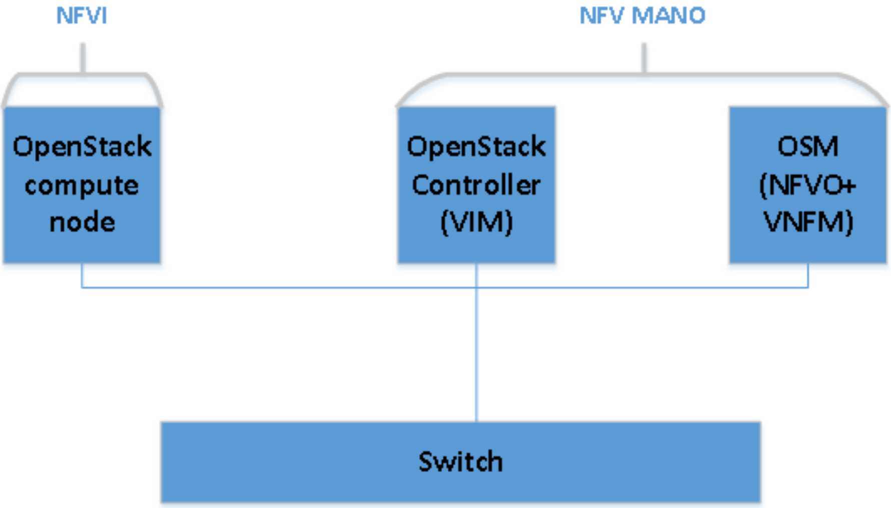


FIGURE 150Topology of the CTTC's NFV testbed.

The OSM and OpenStack are implemented as VM within the CTTC servers.

4.3 Use Case 3 demonstrator

In UC3 scenario we will provide a mapping of the SEMIoTICS architecture towards an "edge computing" approach by tailoring specific scenarios in order to demonstrate its feasibility and integrate the capabilities of the Generic IoT System within SEMIoTICS technology stack. The specific UC3 SEMIoTICS infrastructure adopted in order to bring "Edge computing" approach into reality and the actual mapping of the SEMIoTICS components vs the HW platform testbed developed in task 5.6 is presented in Figure 151. The system presented is an intelligent scalable architecture integrated in SEMIoTICS ecosystem, for the detection and validation of critical events that does not require any prior information the environment to be monitored (IHES, Intelligent Heterogeneous Embedded Sensors). This system is composed of several capable intelligent nodes (IHES Sensing Units) that obtain environmental data, analyze them and detect anomalies through the use of both neural, statistical and autoregressive models (see final deliverable D4.10 for a complete overview of the algorithms deployed as part of Embedded Analytics Component in SEMIoTICS). These smart nodes are connected to a supervisor (IHES Supervisor) deployed at IoT Gateway level, whose purpose is to coordinate the connected nodes. In more detail, each IHES sensing unit is composed by an STM32 CortexM4 80Mhz MCU, an on-board X-NUCLEO-IDW01M1 Wi-Fi adapter and an X-NUCLEO IKS01A2 environmental + inertial sensors expansion board, all stacked together on a single miniaturized PCB board named "CLOUD-JAM". Each MCU is programmed with a dedicated FW stack implementing the UC3 designed analytics algorithms

[21] https://osm.etsi.org

[22] https://www.openstack.org

discussed in D4.3 and D4.10 together with the communication stack (Wi-Fi + MQTT client) based on legacy STM32 SW middleware.

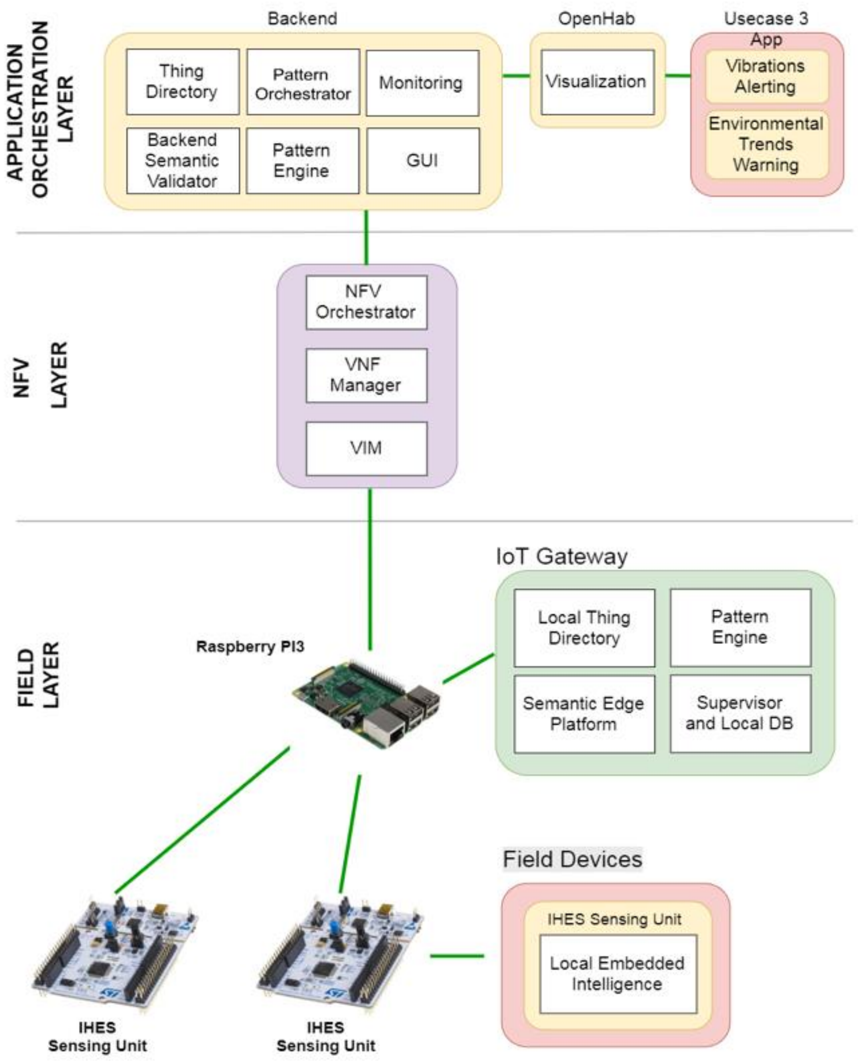


Figure 151. UC3 System vs Components Mapping

The UC3 demonstrator will leverage on this specific technology and is currently under integration within the SEMIoTICS architecture as part of T5.6 activities by following an incremental approach from bottom to top, with associated self-contained storylines and sub-use cases to showcase the incremental functionalities available, as presented in D5.6. Sub use case 1 is devoted to show specific local analytics deployed at device level to enable smart autonomous learning devices following the "Edge Computing" approach. Sub use case 2 will be focused on how to bring the results of the data analytics on the upper level of the infrastructure, and the scalable infrastructure needed for it in order to scale / open-up/scale to backend services. Finally sub use case 3 will deploy end-to-end specific UC3 SPDI patterns exploiting SEMIoTICS ecosystem, to allow a full integration into SEMIoTICS architecture. Currently milestone related to the Sub use case 1 (deployment of Local Embedded Analytics in SEMIoTICS at field device level) has been successfully achieved, and now all the efforts are related to a scale-up of the UC3 infrastructure mapping at Gateway Lavel, and finally on last cycle activities to the cloud level with the interaction of the demo with the IoT platform for the visualization and relevant UC3 patterns mapping.

Please refer to D5.6 for any detailed evidence of the work done so far on UC3 and the current status of the demonstrator.

5 VALIDATION

This section describes the validation features of SEMIoTICS that are related with the implementation of the components and the rest topics that are presented in this document.

5.1 Related Project Objectives and Key Performance Indicators (KPIs)

The following table presents the task objectives and appropriate sections addressing those while Table presents the KPIs and objectives that are relevant for Task 5.3.

TABLE 22: TASK 5.3 OBJECTIVES

T5.3 Objectives	D5.8 Sections
• Implementation of an overarching SEMIoTICS testbed that integrates technologies and components implemented during Cycle 2	0
• Provide the SEMIoTICS SDN controller architecture along with the testing methodology and the KPI validation and evaluation for Cycle 2.	3.1
• Provide the SEMIoTICS NFV architecture along with the testing methodology and the KPI validation and evaluation for Cycle 2.	3.2
• Present the SEMIoTICS advances at the field and gateway layer including semantic bootstrapping, interfacing and interoperability. Showcase the components architecture, provide the testing methodology, and evaluate the performance through KPI validation.	3.3
• Ensure security and safety along all other components through a backend security manager and a pattern orchestrator. Evaluate the performance and validate the KPIs.	3.4.1, 3.4.2

The KPIs and their respective SEMIoTICS objectives that are related to T5.3 are described in the following TABLE 23.

TABLE 23: KPIS AND OBJECTIVES

Objective		KPI-ID	Description	Related task
2	Semantic Interoperability	KPI-2.1	Semantic descriptions for 6 types of smart objects	T3.3
2	Semantic Interoperability	KPI-2.3	Semantic interoperability with 3 IoT platforms	T3.4, T4.4

3	Monitoring Mechanisms	KPI-3.1.1	Generating monitoring strategies in the 3 targeted IoT platforms	T4.1, T4.2
4	Multi-layered Embedded Intelligence	KPI-4.6	Development of new security mechanisms/controls	T4.5
5	IoT-aware Programmable Networks	KPI-5.1	Deployment of a multi-domain SDN orchestrator	T3.1
5	IoT-aware Programmable Networks	KPI-5.2	Service Function Chaining (SFC) of a minimum 3 VNFs	T3.2, T4.1
6	Development of a Reference Prototype	KPI-6.3	Delivery of 3 prototypes of IIoT/IoT applications	T3.5, T4.6, T5.2, T5.3

5.2 SEMIoTICS implementation requirements

The relevant SEMIoTICS requirements that are covered by the presented implementation of SEMIoTICS components are summarized in the next table. The full scope of requirements mapping is available in D2.5.

TABLE 24: REQUIREMENTS´STATUS

Requirements (D5.8)	Description	Related task	Status
R.GP.1	End-to-end connectivity between the heterogeneous IoT devices (at the field level) and the heterogeneous IoT Platforms (at the backend cloud level)	T5.4-6	Delivered
R.GP.3	Scalable infrastructure due to the fast-paced growth of IoT devices	T5.4-5	Delivered
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern	T5.4-5	Delivered
R.GP.5	Interaction between SDN controller and IoT backend cloud through a dedicated interface (called northbound software interface)	T5.4	Delivered
R.GP.6	Interaction between SDN controller and network nodes (e.g. switches, routers or IoT Gateways) through dedicated interface (called southbound software interface)	T5.4	Delivered
R.FD.1	Field devices SHOULD be able to get data from the environment through sensors (sensors).	T5.6	Delivered
R.FD.2	Field devices SHOULD be able to process data in near real time (process units).	T5.6	Delivered
R.FD.3	Field devices SHOULD be able to control (at least) a mechanism / system (actuators).	T5.4	Delivered
R.FD.4	Field devices SHOULD use a global clock for time synchronization.	T5.6	Delivered
R.FD.5	Field devices SHOULD be able to interact with SEMIoTICS IIoT/IoT gateway dedicated components	T5.4-6	Delivered

R.FD.6	Field devices MUST interoperate using a standard communication protocol like Rest APIs, COAP, MQTT.	T5.4-6	Delivered
R.FD.7	Field devices MUST use standardize interoperable message format (e.g. JSON, etc.).	T5.4-6	Delivered
R.FD.8	Field devices MUST support secure bootstrapping / registration protocol.	T5.5	Delivered
R.FD.9	Field devices MUST be able to communicate with the IIoT Gateway / other architectural components.	T5.5	Delivered
R.FD.10	Field devices SHOULD minimize data traffic.	T5.5-6	Delivered
R.FD.11	Field devices SHOULD minimize energy consumption.	T5.5-6	Delivered
R.FD.12	Greenfield device is expected to expose its capability over a W3C Thing Description, which semantically describes field resources, and to be computationally powerful enough to run a node-wot servient (that exposes the TD).	T5.5	Delivered
R.FD.13	Brownfield device is assumed to consist of a sensor/actuator and a controller (PLC). The controller is expected to expose capability of its sensor/actuator over a native brownfield protocol (without the need for IIoT Gateway to interact directly with them).	T5.5	Delivered
R.FD.14	The field layer must feature SPDI pattern reasoning local embedded intelligence capabilities	T5.5-6	Delivered
R.S.1	The confidentiality of all network communication MUST be protected using state-of-the-art mechanisms.	T5.5	Delivered
R.S.2	Authentication and authorization of the stakeholders MUST be enforced by the Network controller, e.g. through access and role-based lists for different levels of function granularities (overlay, customized access to service, QoS manipulation, etc.)	T5.5	Delivered
R.S.3	Sensors SHALL be identifiable (e.g. by a TPM module/smartcard) and authenticated by the gateway.	T5.5	Delivered
R.S.4	All components from gateway, via SDN Controller, to cloud platforms and their users MUST authenticate mutually.	T5.5	Delivered
R.S.5	Before sensitive data is being transmitted, the respective components SHALL be authenticated as defined by requirements R.S.3 and R.S.4	T5.5	Delivered
R.S.6	Sensors SHALL be able to encrypt the data they generate, i.e. their CPU and memory SHALL be sufficient to perform these cryptographic operations.	T5.5	Delivered
R.S.20	Cloud platforms MUST be protected by a firewall against network-based attacks.	T5.5	Delivered

R.P.1	The collection of raw data MUST be minimized.	T5.5-6	Delivered
R.P.2	The data volume that is collected or requested by an IoT application MUST be minimized (e.g. minimize sampling rate, amount of data, recording duration, different parameters).	T5.5-6	Delivered
R.P.3	Storage of data MUST be minimized.	T5.5-6	Ongoing
R.P.4	A short data retention period MUST be enforced and maintaining data for longer than necessary avoided.	T5.5-6	Ongoing
R.P.5	As much data as possible MUST be processed at the edge in order to hide data sources and not reveal user related information to adversaries (e.g. user's location).	T5.5-6	Ongoing
R.P.6	Data MUST be anonymized wherever possible by removing the personally identifiable information in order to decrease the risk of unintended disclosure.	T5.5-6	Delivered
R.P.7	Data granularity MUST be reduced wherever possible, e.g. disseminate a location-related information (i.e. area) and not the exact address.	T5.5	In progress
R.P.8	Data MUST be stored in encrypted form.	T5.5	Delivered
R.P.9	Repeated querying for specific data by applications, services, or users that are not intended to act in this manner SHALL be blocked.	T5.5	Delivered
R.P.10	Wherever possible, information over groups of attributes or groups of individuals SHALL be aggregated (e.g. 'the majority of people that visited the examined area in this time interval were young students' this is sufficient information for an advertising application of a nearby shop, without requiring to process raw data from the personal IoT devices).	T5.5	Delivered
R.P.11	The data principal SHALL be sufficiently informed regarding which data are collected, processed, and disseminated, and for what purposes	T5.5	Delivered
R.P.12	During all communication and processing phases logging MUST be performed to enable the examination that the system is operating as promised	T5.4	Delivered
R.P.13	The user SHALL be able to control the privacy mechanisms (i.e. redemption period, data granularity and dissemination, and anonymization technique)	T5.4	Delivered

6 CONCLUSIONS

This deliverable provides the status at the end of Cycle 2 of the infrastructure setup and testing of the SEMIoTICS solution. In the beginning, we discussed about the overarching testbed that includes various components, being composed of a Backend layer, an SDN/NFV Orchestration layer, and a Field layer, as well as a VPN solution for integrating external components. Then, we provided the architecture of each component by giving focus on the main advancements occurred during the implementation and providing specific details for the characteristics that will be leveraged by each of the three use cases. Moreover, in order to provide an accurate and detailed performance evaluation for each component, we demonstrated the testing methodology and, then, we presented the results of the tests along with the KPI validation for each component, where applicable. The components were validated at the SEMIoTICS testbed environment, leveraging a VPN solution to interconnect partners’ facilities. This setup will support the SEMIoTICS use cases.