Ref. Ares(2020)4506683 - 31/08/2020

780315 — SEMIoTICS — H2020-IOT-2016-2017/H2020-IOT-2017

SEMIoTICS

SEMIoTICS

Deliverable D5.7
Software system integration (Cycle 2)

Deliverable release date	31.08.2020
Authors	1. Arne Broering, Darko Anicic, Jan Seeger (SAG) 2. Eftychia Lakka, Nikolaos Petroulakis, Emmanouil Michalodimitrakis (FORTH) 3. Konstantinos Fysarakis, Iasonas Somarakis, Manolis Chatzimpyrros, Georgia Koutsouri (STS) 4. Domenico Presenza (ENG) 5. Felix Klement, Korbinian Spielvogel, Henrich C. Pöhls (UP) 6. Mateusz Kamiński, Jakub Rola, Michał Rubaj, Bartłomiej Lipa (BS) 7. Prodromos Vasileios (IQU)
Responsible person	Bartłomiej Lipa (BS)
Reviewed by	Bartłomiej Lipa (BS), Mateusz Kamiński (BS), Konstantinos Fysarakis (STS), Emmanouil Michalodimitrakis, Emmanouil Papoutsakis (FORTH), Kostas Ramantas (IQU). Felix Klement (UP)
Approved by	PTC Members (Vivek Kulkarni, Nikolaos Petroulakis, Ermin Sakic, Mirko Falchetto, Domenico Presenza, Christos Verikoukis) PCC Members (Vivek Kulkarni, Nikolaos Petroulakis, Christos Verikoukis, Georgios Spanoudakis, 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

TABLE 1 ACRONYM TABLE

Acronym	Definition
API	Application Programming Interface
CI	Continuous Integration
CD	Continuous Delivery
WP	Work Package
IoT	Internet of Things
KPI	Key Performance Indicator
PERT	Program Evaluation Review Technique
UML	Unified Modelling Language
SPDI	Security & Privacy & Dependability & Interoperability
NFV	Network Functions Virtualization
VNF	Virtualized network function
SME	Small and Medium Enterprises
IIoT	Industrial Internet of Things
REST	Representational state transfer
W3C	The World Wide Web Consortium
GUI	Graphical User Interface
WoT	Web of Things
JSON	JavaScript Object Notification
HTTP	Hypertext Transfer Protocol
JSON-LD	JavaScript Object Notation for Linked Data
URL	Uniform Resource Locator
GW	Gateway
PO	Pattern Orchestrator
ANTLR4	Another Tool for Language Recognition
SDN	Software-Defined Networking
SFC	Service Function Chaining
VIM	Virtualized Infrastructure Manager
OVS	Open vSwitch
OSM	Open Source MAO
OAuth2	Open Standard for Authentication Version 2
BSV	Backend Semantic Validator
GWSM	Gateway Semantic Mediator

SAPB	Semantic API & Protocol Binding
TD	Thing Directory
GE	General Enabler
PEP	Policy Enforcement Point
PDP	Policy Decision Point
XACML	eXtensible Access Control Markup Language
RDF	Resource Description Framework
DB	Database
OSGi	Open Services Gateway initiative
OWL	Web Ontology Language
OSSOSS	Operations Support System
BSS	Business Support System

1. INTRODUCTION

This document is a continuation of Document D5.2 and describes the next cycle of the Task T5.2 which focused on the software integration of the SEMIoTICS framework components.

In general, the integration approach of the SEMIoTICS framework components has not changed. This document describes additional components developed in WP3 and WP4, complementing document D5.2. As a result of this process, the integrated framework provides the basis for evaluating the effectiveness of the SEMIoTICS approach in real-life scenarios and trial operations in domains targeted by the project (T.5.4, T.5.5, and T.5.6). Particular emphasis is given on the automated processes of Continuous Integration and Continuous Delivery (CI/CD) which the integration process is based on.

The deliverable is structured as follows:

- Section 2 covers the approach to the integration taken within this project.
- Section 3 presents the integration flows which have been delivered within the first cycle of the task works
- Section 4 describes the approach for the IoT platforms interoperability
- Section 5 is the validation section where one can see what objectives and KPIs are pertinent to the work presented within this deliverable
- Section 6 features the concluding remarks

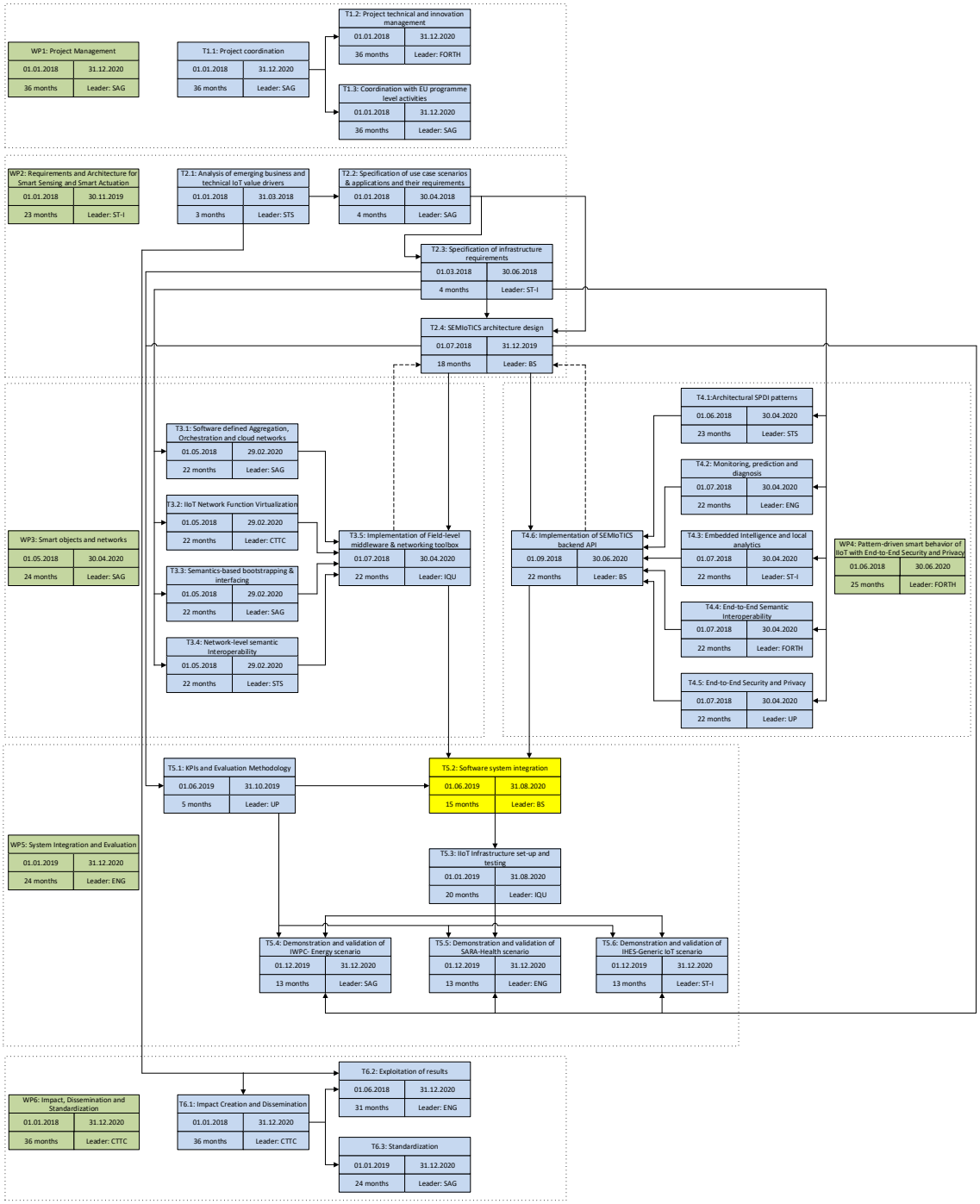
Compared to document D5.2, a new chapter 3.2 was added to describe the integration of the components delivered in Cycle 2.

Further details on the integration of Use case specific SEMIoTICS components 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)

1.1. PERT chart of SEMIoTICS

The PERT chart below provides a graphical representation of the project's timeline, allowing the breakdown of each individual task in the project for analysis.



2. INTEGRATION APPROACH AND METHODOLOGY

SEMIoTICS is a complex framework consisting of various components developed by multiple parties. To allow the software component integration, the state-of-the-art approach for developing complex systems has been used. The components have been assigned to most expert consortium partners in order to coordinate the proper integration.
The above-mentioned approach allows the semi-independent and self-paced development of each partner. However, it also creates the challenge of the integration of all components. The solution for this challenge is the microservices approach architecture and their API. This section describes how the consortium manages the process of integrating all components into one whole working platform on the backend level.

In more detail, the integration process has been divided into three steps:

1. Divide the platform functions into components and assign them to the right partner
2. Define the API of each component
3. Declare and define the communication/dependence among each component

Each of these steps is further elaborated in the subsections that follow.

2.1. Divide the platform functions into components and assign them to the right partner

The first step in the developing process was straightforward. Once the architecture of the platform has been established, each functionality has been divided into small components and assigned to the appropriate partner. The process of the assignment was based on the expertise and technologies brought into the project by each partner.
Figure 2-1 shows the result of the first step. The platform is divided into 3 layers, each layer is divided into components and each component is assigned to the relevant partners.

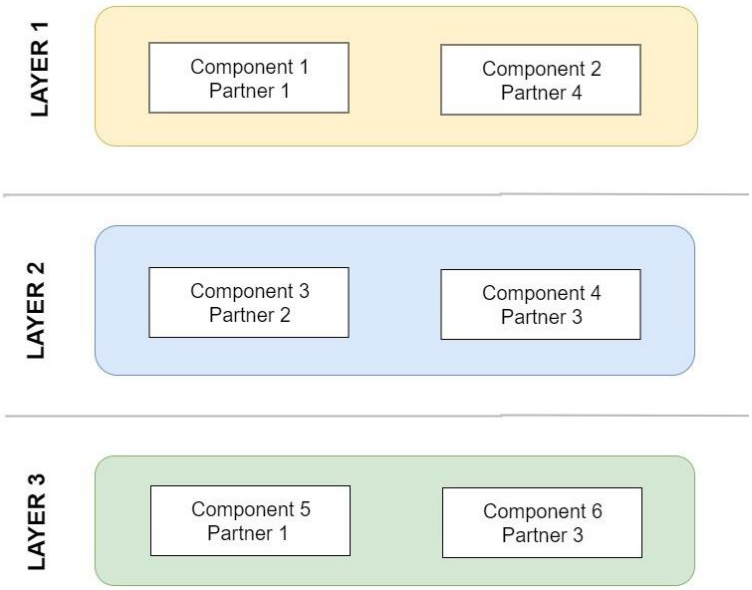


FIGURE 2-1. STATE AFTER STEP 2.1

2.2. Define the interface of each component

While developing each component, partners were defining the API of their part of the platform. To document this, each partner was also updating the corresponding UML component diagram, but without marking the connection between components. This allowed all parties involved in the project to follow the changes in the

API. In cases of a party having doubts or objections to another partner's component interface, the issue was clarified between involved parties. This process was self-organized and self-administrated. The dependencies among the components can be found in a Deliverable D2.5 SEMIoTICS High-Level Architecture (final). The initial API specification without covering component interactions was chosen on purpose; the lack of connections makes the diagram more readable, and simpler. Figure 2-2 below shows this approach of documenting components with their endpoints, but no connections between endpoints.

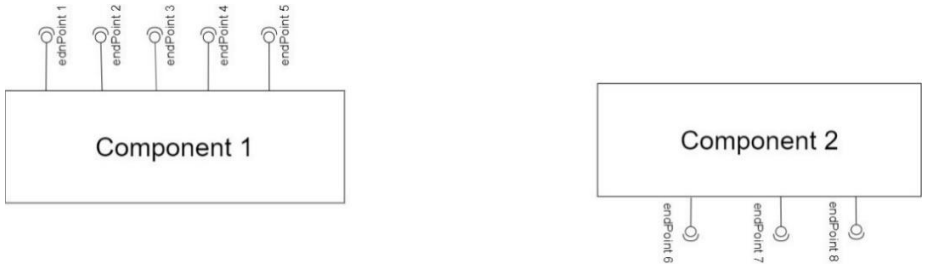


FIGURE 2-2 REPRESENTATIVE COMPONENTS WITH ENDPOINTS

2.3. Declare and define the communication/dependencies among all components

The most challenging part of developing a complex platform based on microservices is to ensure that components are able to interconnect whenever necessary. To make it possible and manageable it was decided to modify the standard UML component diagram. All identified endpoints have been merged into one diagram and the data flow has been shown between components (Figure 2-3). Information about the usage of specific endpoints was shown on separate sequence diagrams developed during working on use cases and discussions between interested partners. This task was the most complex and engaging for every partner in the consortium. The workflow in this task is described below:

1. The appointed partner (coordinator) prepared the first version of the flow diagram based on previously published deliverables and internal project documents.
2. Every partner raises their objections (if any) about the flows to the coordinator
3. The coordinator resolves the conflicts and prepares the new version of the graph
4. Steps 2-3 are repeated until all concerns are addressed

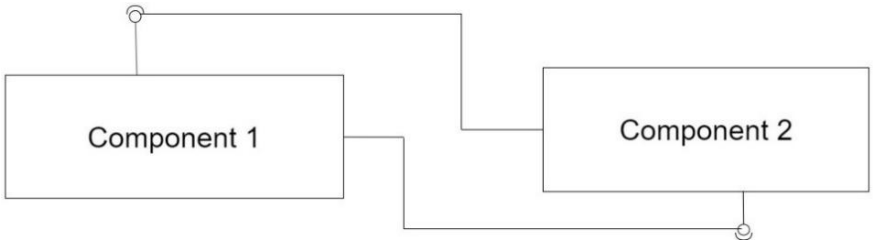


FIGURE 2-3 SAMPLE DIAGRAM OF FLOW BETWEEN TWO COMPONENTS

2.4. Continuous Integration / Continuous Deployment

The microservices structure of the platform allows applying the continuous integration and continuous deployment philosophy. It allows each partner to implement small changes in code and allows for a fast response when changes occur in another partner's requirements.
This philosophy leads to a better code quality and less time spent by introducing automatization in the building and deployment process. It also encourages developers to publish even small improvements in code by simplifying and automatizing the tedious process of testing and deployment.

The opportunity to use the automated pipelines is available for all consortium members. The process is presented in more detail in the subsections that follow, and it can be tailor-made for every partner.

2.4.1. CI/CD TOOLS USED IN SEMIOTICS

The tools used to automatize the process of developing and deployment of the platform include:

- GitLab as the code repository[1]
- Jenkins as a simple CI server[2]
- Docker as a container platform[3]
- GitLab Container Registry as a registry of containers images[4]
- Kubernetes as a runtime environment for containers[5]

2.4.2. CONTINUOUS INTEGRATION (CI) PIPELINE

The CI idea within the SEMIoTICS project is presented in Figure 2-4. The proposed project pipeline is based on the standard CI pipeline. The main difference is that the desired product is a Docker image. At first, the pipeline is started manually. As it is shown in the table below pipeline starts at Jenkins, then the new code is fetched from the GitLab repository, the code is compiled, tested and build. At the end of the process, a docker image is pushed to the Docker or GitLab image registry.

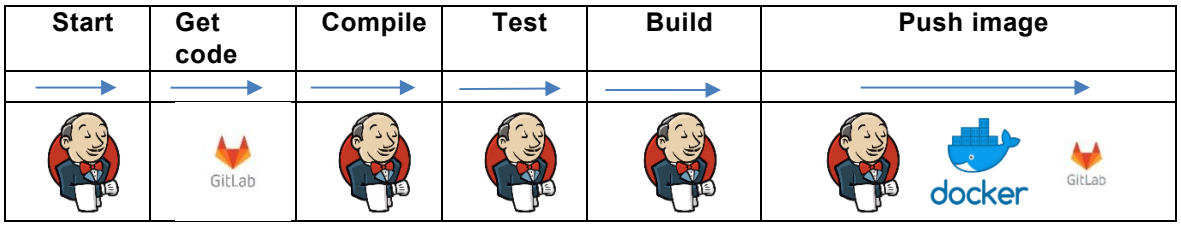


FIGURE 2-4 CI PIPELINE

2.4.3. CONTINUOUS DEPLOYMENT (CD) PIPELINE

The overall CD idea within the SEMIoTICS project is presented in Figure 2-5. At first, the pipeline is started manually. Firstly, in order to start the pipeline, the changes need to be committed to Gitlab. As it is shown in the table below, the pipeline starts at Jenkins, then cluster configuration files are pulled from GitLab. Next, Jenkins plugin plan changes, then apply changes and deploy them on Kubernetes cluster. Kubernetes gets a declarative configuration of the cluster and are then responsible for other actions – e.g. obtaining images from the Docker registry (if a Docker registry is private, the special Secret resource needs to be created to pull the image).

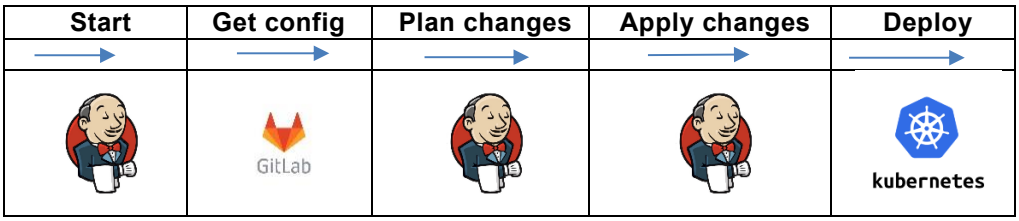


FIGURE 2-5 CD PIPELINE

[1] https://about.gitlab.com
[2] https://jenkins.io
[3] https://www.docker.com
[4] https://about.gitlab.com/blog/2016/05/23/gitlab-container-registry/
[5] https://kubernetes.io

3. INTEGRATION DESCRIPTION AND IMPLEMENTATION PROGRESS

Within cycle one, the consortium decided to focus on the integration development related to the core functionality of SPDI patterns and their distribution across all 3 identified layers of SEMIoTICS framework as well as the pattern definition and visualization. This is a crucial aspect for achieving the objective of multi-layered embedded intelligence and enablers the semi-autonomous operation of the assets within the different layers. Hence, a number of integrations of Pattern Engine and Pattern Orchestrator with components across three layers are described in the following sections.
Field devices bootstrapping as one of the core flows required for any other functionality is described below while the integration flows required for the brownfield devices are going to be detailed in the further cycle.
The second area of focus for cycle one was the semantic interoperability within and externally to the SEMIoTICS framework which means between the integral SEMIoTICS components as well as with external IoT platforms such as CLOE-IOT, MindSphere, OpenHab, and FIWARE. Details of specific integration flows may be found in the subsections that follow.
Within Cycle 2, the partners were to prepare the integrations not performed in the previous cycle. The work focused on preparing communication GUI with the Security Manager and the Monitoring Component. Thanks to the former, data processed in the SEMIoTICS project were protected against unauthorized access.
Moreover, as part of Cycle 2, the partners performed tasks related to integration with Recipe Cooker, Thing Directory and Backend Semantic Validator.

3.1. Integration flows delivered in cycle 1

3.1.1. PATTERN ENGINE INTEGRATION WITH ORCHESTRATORS AT ALL LEVELS

The Pattern Engine is responsible for reasoning on the Security, Privacy, Dependability, and Interoperability (SPDI) properties across all layers of the SEMIoTICS architecture. For this reason, variants of Pattern Engine are implemented in the backend, in the network, and in the field layer. Patterns are inserted, modified, executed or retracted at design as well as at runtime. These interactions are conducted with the help of Pattern Orchestrator. Apart from the interaction of Pattern Orchestrator with the Pattern Engines across all layers, there is also the interaction between NFV Orchestrator and the Backend Pattern Engine. Currently, this interaction is limited only for verifying that any required VNFs are instantiated in order to satisfy a related SPDI property. In Table 2, the interactions of Pattern Engines with the Orchestrators along with a small description are presented.

TABLE 2 PATTERN ENGINE INTERACTIONS WITH ORCHESTRATORS

Pattern Engine	Orchestrators used by Pattern Engine	Description of interactions
Backend Pattern Engine	*Pattern Orchestrator*	Pattern Orchestrator is sending the pattern requirements and receives the status of the requirement after the Pattern Engine has reasoned based on the facts and rules stored in the Pattern Global Repository
	NFV Orchestrator	The Pattern Engine is getting the available VNFs from NFV orchestrator when a related pattern requirement is received.
SDN Pattern Engine	*Pattern Orchestrator*	Pattern Orchestrator is sending the pattern requirements and receives the status of the requirement after the Pattern Engine has reasoned

		based on the facts and rules stored in the SDN Pattern Repository
Field Pattern Engine	*Pattern Orchestrator*	Pattern Orchestrator is sending the pattern requirements and receives the status of the requirement after the Pattern Engine has reasoned based on the facts and rules stored in the Field Pattern Repository

As shown in the sequence diagram of Figure 3-1, the Pattern Orchestrator will choose to send the SPDI requirement to one or more Pattern Engines depending on the orchestration requirements (e.g., sending network-related requirements to the network pattern engine, if such requirements are included within the orchestration specification). This will trigger 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 Global Pattern repository as well. When the requirement is related to some VNFs then interaction with the NFV orchestrator will also occur in order for the final requirement status to be formed.
For the needs of the communication between Pattern Engine and the Orchestrators, POST service requests have been developed such as addFact, insertRule and factUpdate.

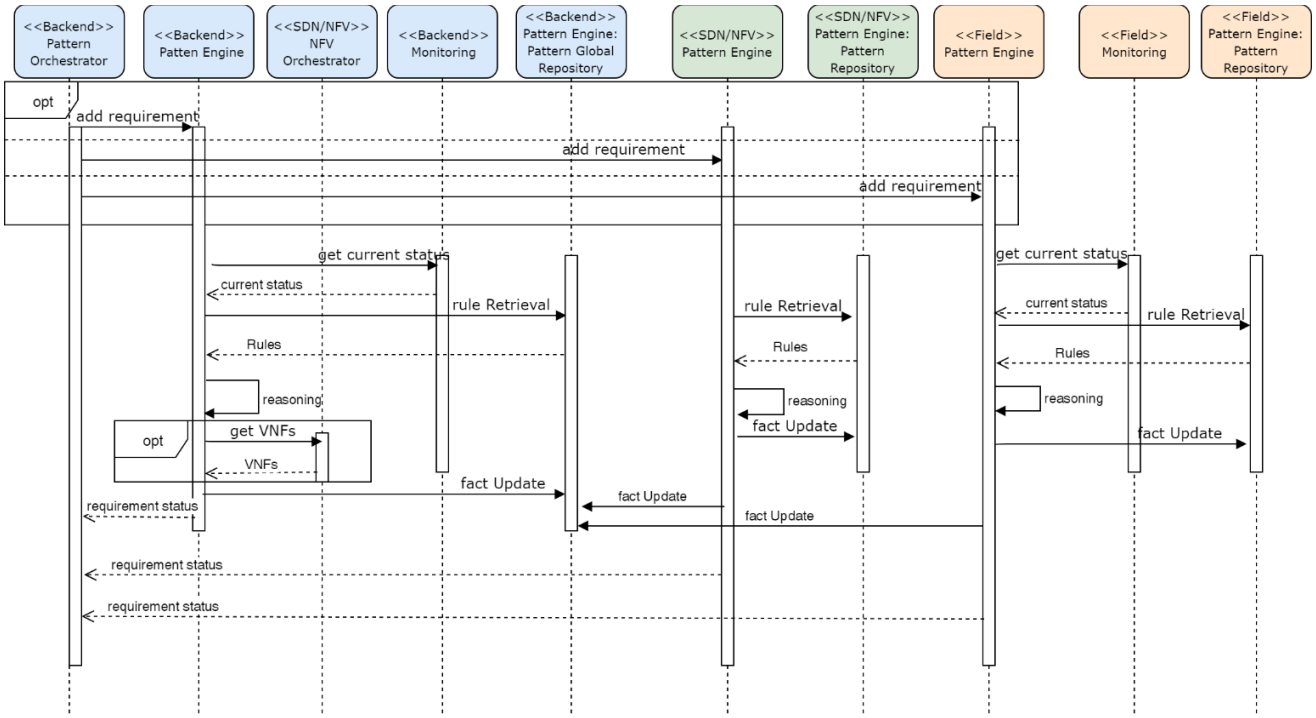


FIGURE 3-1 SEQUENCE DIAGRAM FOR PATTERN ENGINES INTERACTION WITH ORCHESTRATORS

3.1.2. FIELD DEVICES INTEGRATION

Figure 3-2 shows a sequence diagram of activities that occur during the bootstrapping process. The goal of this process is to integrate a new device in the SEMIoTICS platform by using SEMIoTICS IIoT Gateway. Figure 3-2 represents an updated version of a sequence diagram that was presented in the Deliverable D3.3 (see Figure 16). The update is concerned with the introduction of a new component “Semantic Edge Platform”

(SME). The SME has multiple purposes in the architecture: (i) it provides a convenient user interface for configuring SEMIoTICS IoT Gateway; (ii) it provides a convenient development environment for creating new Apps with a newly bootstrapped device, and; (iii) it provides a mechanism to semantically annotate brownfield devices.

Once the process in Figure 3-2 is completed, it is possible to create new applications based on data from the new device, as well as the data from other available devices in the platform. In order to achieve this goal, SEMIoTICS IoT Gateway needs to make the device data accessible, and it has to provide a full semantic description of the device, i.e., semantics about device capabilities, its data, communication protocols, contextual information (e.g., location, a domain of use), etc.

In the bootstrapping process, different classes of the device are distinguished. The first class consists of devices that already have a Web-based RESTful interface and are described by W3C Thing Description. The second class comprises 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. For further details, an interested reader is referred to as SEMIoTICS Deliverable D3.3.

So far, the bootstrapping process has been implemented and demonstrated for the first class of devices. The implementation of the second class is in progress and will be delivered in Cycle 2

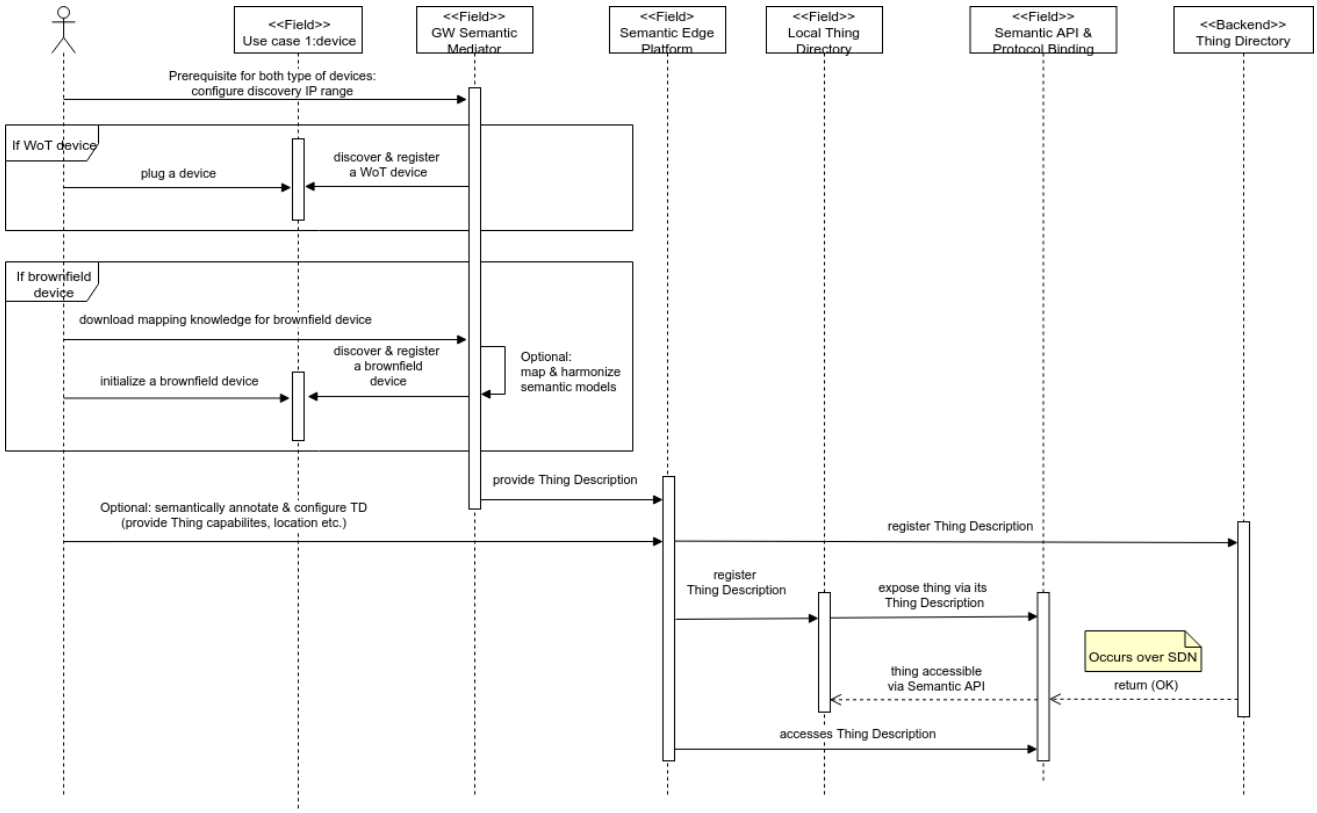


FIGURE 3-2 SEQUENCE DIAGRAM FOR BOOTSTRAPPING AND INTERFACING SEMIOTICS FIELD LEVEL DEVICES

3.1.3. GUI INTEGRATION

The Graphical User Interface is a module that overlays some components of the SEMIoTICS projects. Its main purpose is to support the visualization of individual components and the presentation of collected data in one IoT platform. A detailed description of GUI architecture and interactions between internal and external components were included in D4.7 in section 4.5. According to the project assumptions, GUI integrates with Thing Directory, WoT compliant field devices, and the Pattern Orchestrator. Due to that fact, the description of each integration is provided in a separate subsection below.

3.1.3.1. GUI integration with Thing Directory

This module is responsible for basic visualization of Things currently registered in Thing Directory. A list of all Things is not stored in the GUI database, so only the Thing Directory provides a current state of devices connected to the IoT platform. To receive data, the GUI through an internal component sends HTTP requests to the Thing Directory's API and in the response - the body 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 getting data from Thing Directory, the JSON description needs to be translated into a user-friendly form. For this purpose, mapping to a previously defined object is used, so that the user can easily browse devices with their attributes. Moreover, the GUI provides support for the SPARQL filter for easy searching in Thing Directory. This component also allows for adding new things and remove existing ones directly through the platform. It is not the main way to register new devices to the platform, but it can be additional functionality to support the Thing Directory. Sequence diagrams illustrating interactions between GUI and Thing Directory are depicted in Figure 3-3 and Figure 3-4.

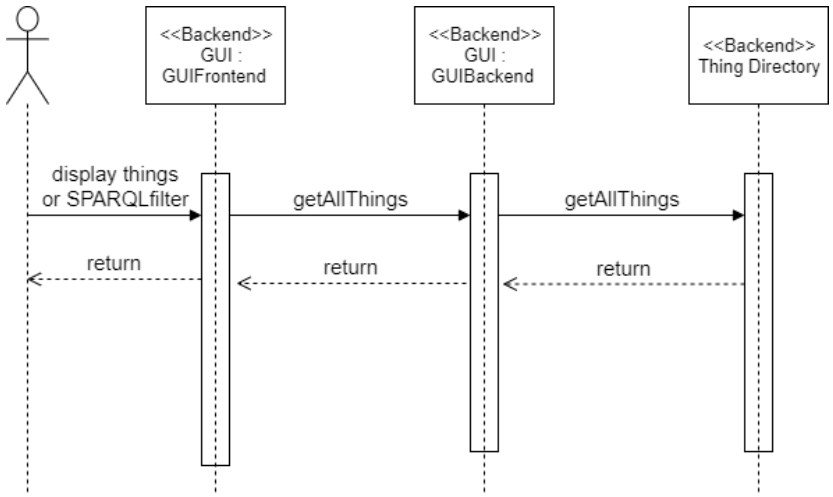


FIGURE 3-3 SEQUENCE DIAGRAM, DISPLAY ALL DEVICES FROM THING DIRECTORY

As shown in Figure 3-3, when the user wants to show or filter devices a GET request from the GUI is sent to Thing Directory and the returned data is translated from JSON-LD format to model that can be presented in the platform. A similar flow occurs when the user registers a new thing through a dedicated window in the GUI. The definition of Thing Description in JSON-LD standard is sent by the POST method directly to Thing Directory where is validated and added to the existing list.

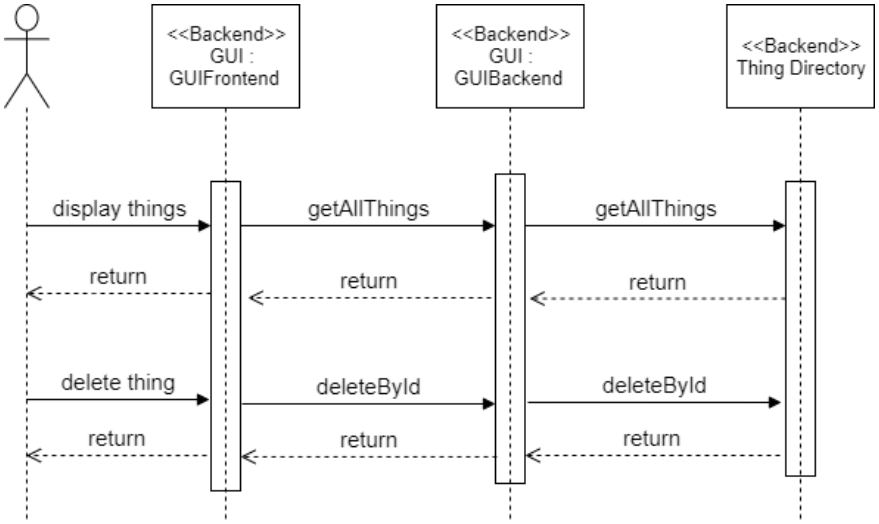


FIGURE 3-4 SEQUENCE DIAGRAM, DELETE THING

Figure 3-4 demonstrates the process of removing a “Thing” from the Thing Directory. The user needs to get a list of things registered in Thing Directory as mentioned in the previous diagram and then can select a list of the device to delete. After confirmation, for each thing, the GUI sends one by one POST requests with Thing id as parameter. When the operation is completed, the user gets a message with success or errors that have occurred.

3.1.3.2. GUI integration with WoT compliant field devices and Semantic API & Protocol Binding

The need to integrate GUI with Semantic API & Protocol Binding resulted from the ability to connect Brown Field Devices to the SEMIoTICS platform. For this type of device, receiving real-time data or triggering actions is not possible without a special mediator that can provide an endpoint to get different requests (GET methods to return properties values and POST methods to control actions). This component is not used to connect with WoT devices which have their endpoints and GUI can receive data directly without using additional components. As it was mentioned above, before reading values from devices in real-time, actuate any action and collect data at a set frequency, GUI component must get URL addresses of each endpoint. For this purpose, a thing description from Thing Directory in the JSON-LD standard is translated to assign actions and properties of the device to the correct URL address. As a result of the mapping, a new data object is created, what ensures quick communication with the Semantic API & Protocol Binding component. Figure 3-5 and Figure 3-6 present interactions between the described components.

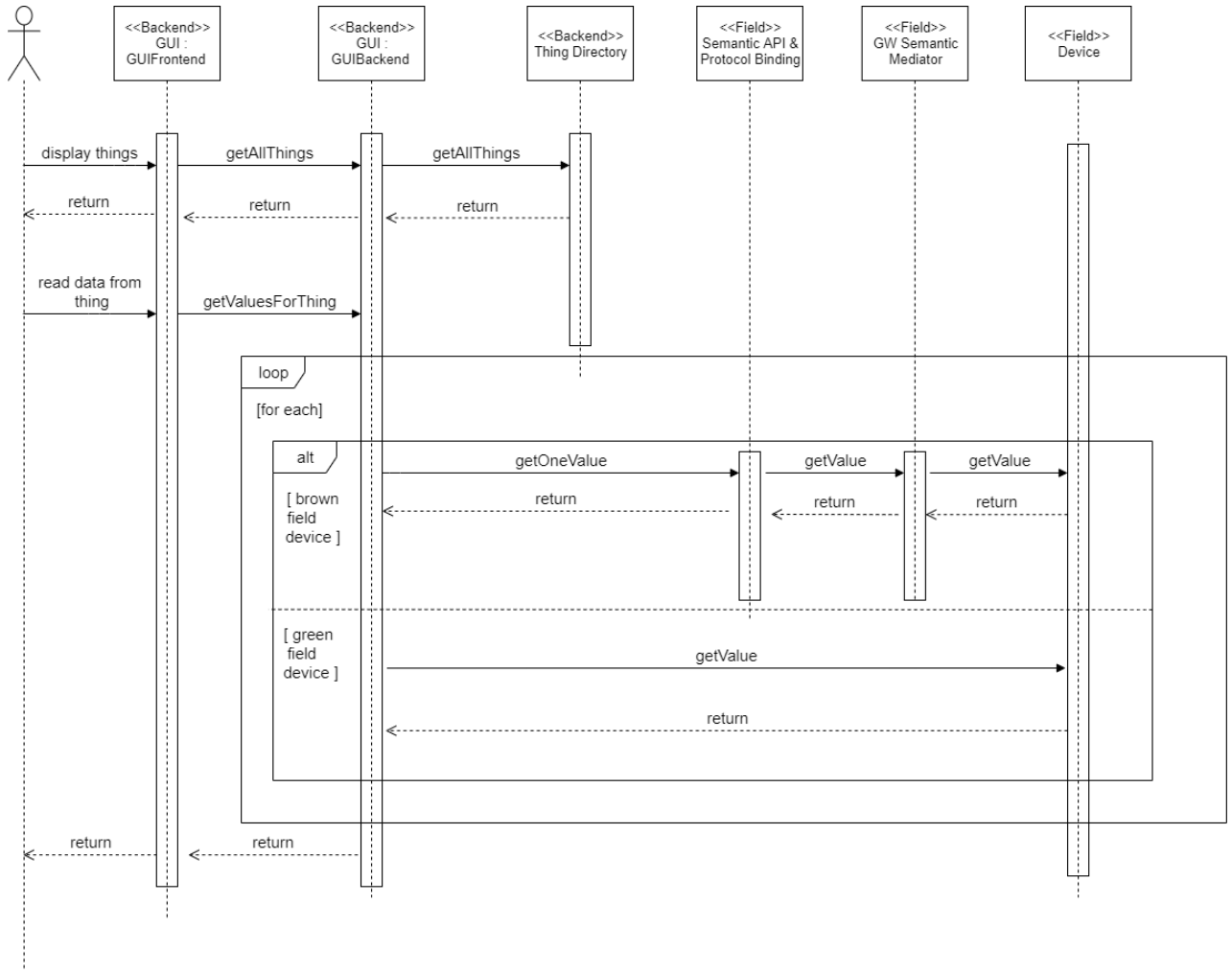


FIGURE 3-5 SEQUENCE DIAGRAM, READ REAL-TIME DATA FROM DEVICE

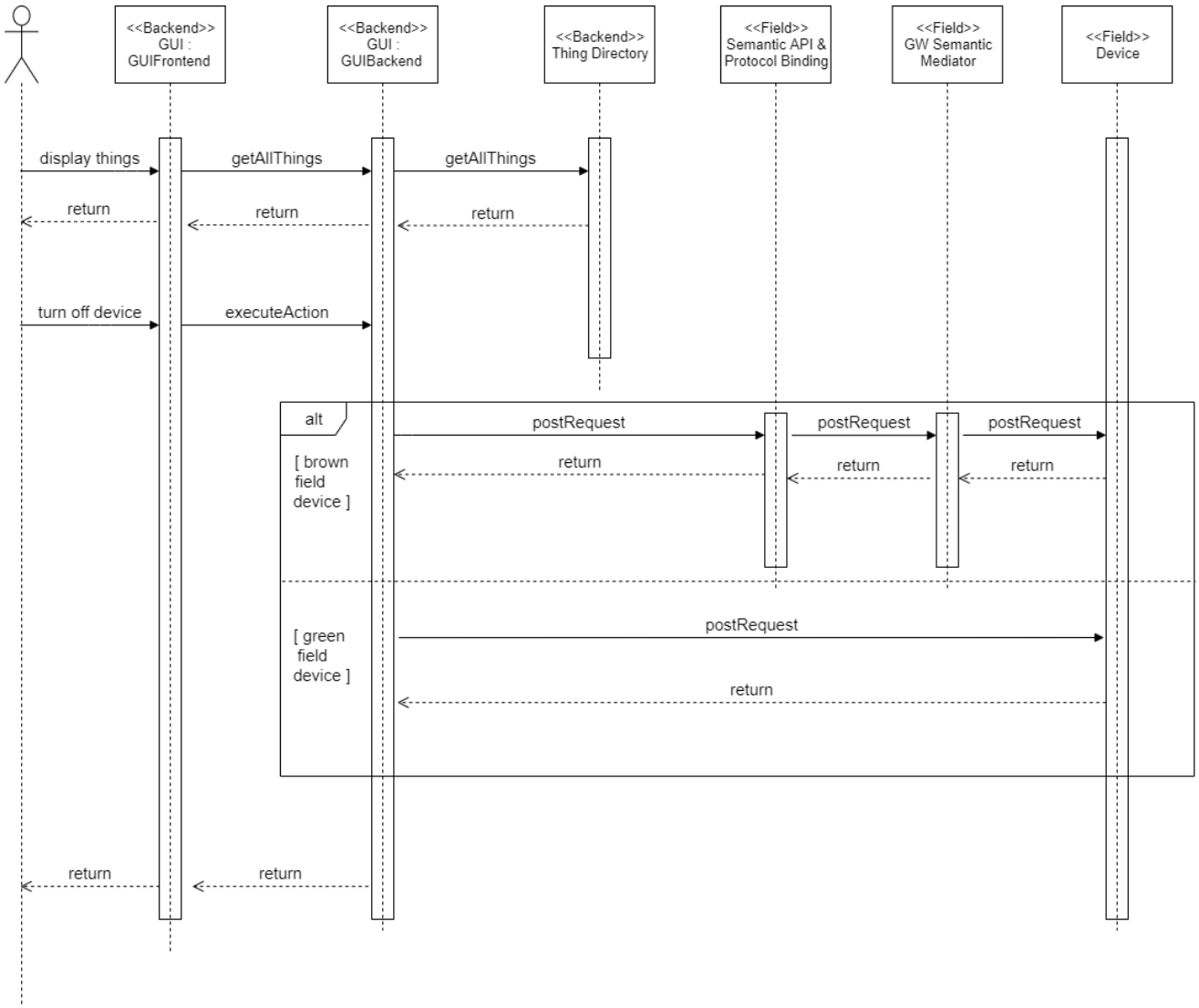


FIGURE 3-6 SEQUENCE DIAGRAM, CONTROL ACTION

3.1.3.3. GUI interaction with Pattern Orchestrator (PO)

The work on integrating the GUI with Pattern Orchestrator started with determining the JSON model to send data between them. It was a crucial step to enable parallel work by partners. The aim of this integration was to support Pattern Orchestrator in monitoring the current state of SPDI patterns from all recipes and location SPDI patterns in an individual layer. In Pattern Orchestrator component, a dedicated endpoint was created for GUI that provides combined data with SPDI patterns and recipes. An example of the JSON model that was created for this communication is depicted in Figure 3-7.

```
{
  "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"
              }
            ]
          },
          {
            "ID": "ObjectDetector",
            "Name": "ObjectDetector",
            "layer": "network",
            "properties": [
              {
                "name": "Memory",
                "satisfied": "false",
                "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 3-7 EXAMPLE OF JSON RETURNED FROM PATTERN ORCHESTRATOR

As shown in Figure 3-7, the model data contains a list of defined recipes with all nodes (e.g. links, sequences, nodes) combined with SPDI patterns defined for them. All patterns are assigned to one of the possible layers (backend, network, gateway) or to one of three cross layers that are between standard layers. To receive data from PO special HTTP method called getSPDIData was developed. When PO receives a request, it merges data from the external component (e.g. Recipe Cooker) and returns a response in JSON format. The GUI then translates this data to show it in two possible ways, as patterns with assigned to layers or as a node graph. Creating a graph from a JSON description required the implementation of new algorithms to be able to show the graph in a similar form to Recipe Cooker. Detailed descriptions with example views can be found in deliverable D4.7 (section 4.3.4), while the interaction between components is depicted in Figure 3-8.

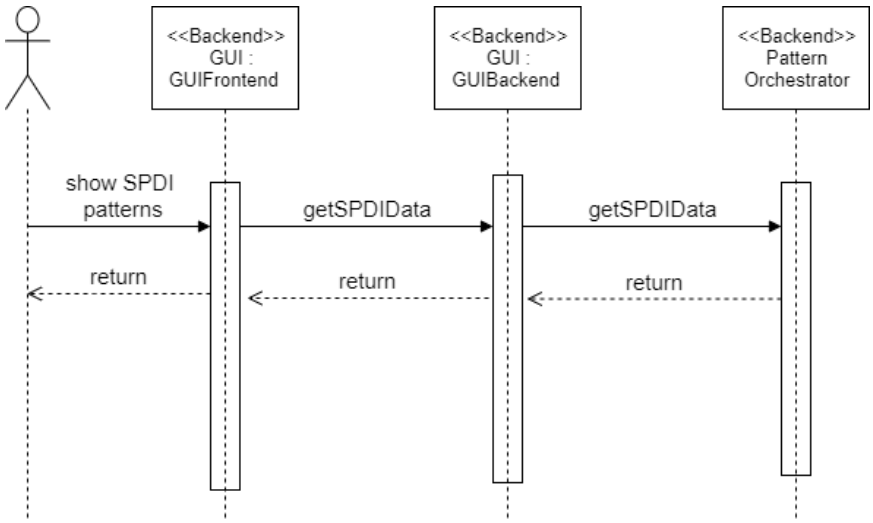


FIGURE 3-8 SEQUENCE DIAGRAM, SHOW SPDI PATTERNS

3.1.4. PATTERN ORCHESTRATOR INTEGRATION WITH RECIPE COOKER

This integration is responsible for translating IoT and service orchestrations, which represent concrete recipes, into patterns and passing them to pattern engines on each layer. The Pattern Orchestrator module features an underlying semantic reasoner able to understand the internal components of IoT Service orchestrations expressed using the pattern language (see deliverable D4.1, Section 3.3), received from the Recipe Cooker module and transform them into architectural patterns. The patterns that are created are then communicated to the corresponding Pattern Engines (as defined in the Backend, Network, and Field layers), taking into consideration the components under their control (e.g. passing Network-specific patterns to the Pattern Engine present in the SDN controller). As a result, automated configuration, coordination, and management of the SEMIoTICS patterns are achieved across different layers and service orchestrations.

The components of the SEMIoTICS architecture that are involved in the process described above are the Recipe Cooker, the Pattern Orchestrator and a translator component between them. The main aim of this translator component is to express an instantiated recipe in a way that is understandable by the Pattern Orchestrator. For that reason, the IoT application model, as described in D4.1, has been created, based on an orchestration-based approach, where the interactions between application components are specified as orchestrations of activities (supporting Sequences, Merges, Choices, Splits etc.). A high-level view of the key components and their interfacing is depicted in Figure 3-9, while the interactions of the aforementioned components are visualized in the sequence diagram in Figure 3-10.

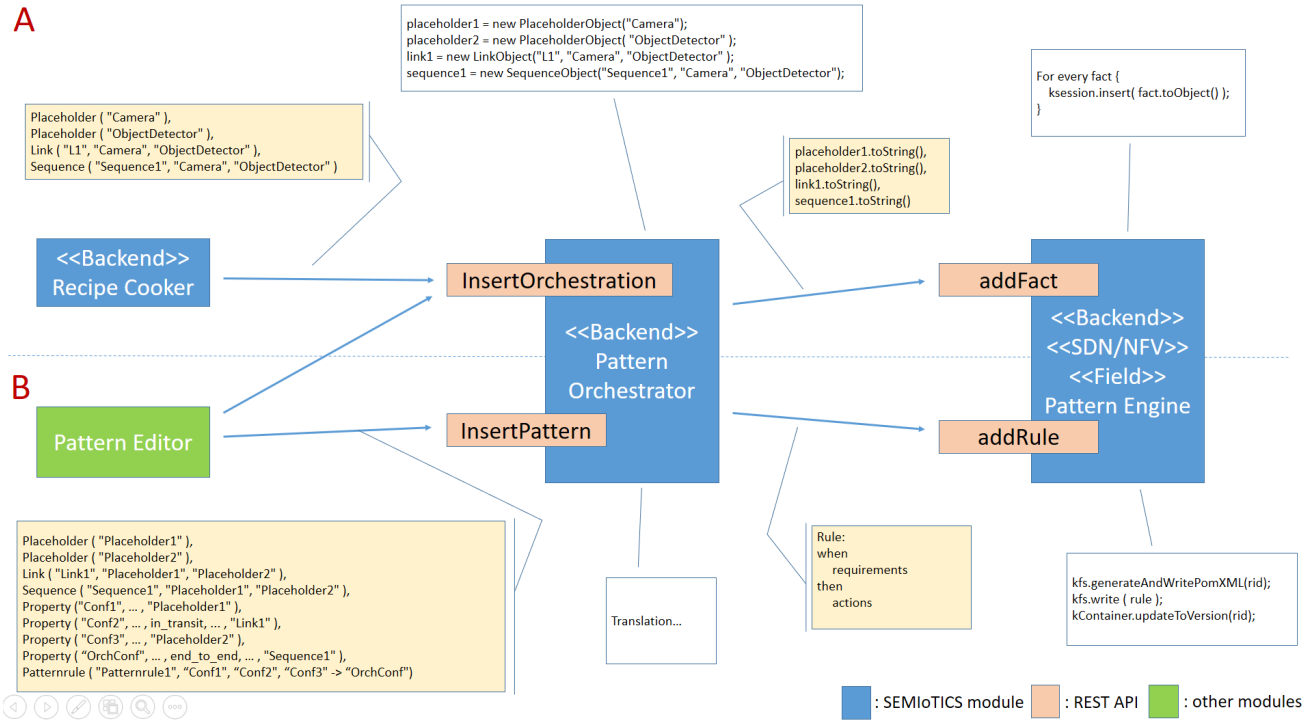


FIGURE 3-9 PATTERN ORCHESTRATION; KEY INTERFACES AND COMPONENT INTERACTIONS

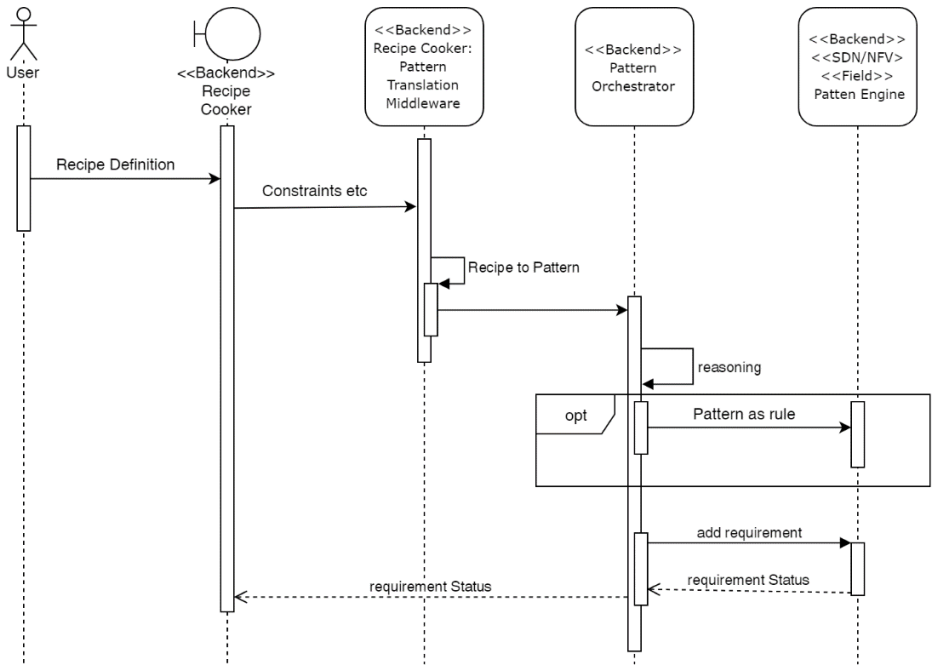


FIGURE 3-10 SEQUENCE DIAGRAM, COMMUNICATION BETWEEN RECIPE COOKER AND PATTERN ORCHESTRATOR

As shown in the sequence diagram above, the user defines the recipe (i.e., the application flow) and specifies the expected capabilities of ingredients, such as input and output data types. The Recipe Cooker tool is utilized for this specification. After this step, the instantiation of the recipe takes place. “Instantiation” refers to the replacement of abstract components with concrete available components. The recipe is then deployed. The recipe deployment triggers the transmission of the recipe instance to the Pattern Translation Middleware. What follows is the description of the recipe instance in terms of the pattern language. This procedure is depicted in the sequence diagram as a self-call to the Pattern Translation Middleware activation, labelled as “Recipe to pattern”. Translation from Node-RED JSON format into the pattern language is realized through a series of graph transformation steps, where nodes from the recipe are collapsed into an orchestration of the pattern language (Sequence, Merge, etc.), until the graph has only a single node left. The transformation steps are then translated into the pattern language.

In sequence, the recipe expressed as the pattern is transmitted to Pattern Orchestrator. For that purpose, a POST service request has been developed. It is called insertRecipe request. Pattern Orchestrator receives a request from Recipe Cooker, which includes a recipe description in JSON format. Such a request is depicted in Figure 16. Under “recipeID” a unique string that acts as an identifier is provided, while under “recipe” label lays the recipe description itself. The recipe instance depicted in Figure 16 is very simple and consists of two software components that are placed in sequence, which means that the output of the former is consumed as input by the latter.


```
{
    "recipeID": "Demo2WF1",
    "recipe": "Softwarecomponent(\"f5a474bd4cd818\", \"0\", \"piB\"), Softwarecomponent(\"f86e905a107f1\", \"0\", \"piB\"),
        Sequence(\"Seq0\", \"f86e905a107f1\", \"f5a474bd4cd818\", \"Link0\")"
}
```


FIGURE 3-11 INSERT RECIPE REQUEST

Eventually, the IoT deployments described using the pattern language will be sent and stored in the Pattern Engines of the three layers (Backend, Network, and Field). For that reason, they need to be translated to Drools; to achieve this they are used as input to an ANTLR4[6] lexer, parser and listener, which is part of Pattern Orchestrator. These programs create a Drools fact for every orchestration activity, control flow operation and property. The Drools facts are then inserted in the KnowledgeBase of Drools, a repository of all the application’s knowledge definitions. Sessions are created from the KnowledgeBase in which data can be inserted and process instances started. A knowledge session is a way to interact with Drools and the core component to fire Drools rules. Rules themselves are also held in a Knowledge session. The information that is stored in the KnowledgeBase is used for reasoning.

During the first step of the translation of an IoT application orchestration to Drools facts, the ANTLR4 lexer recognizes keywords and transforms them into tokens. The created tokens are used by the ANTLR4 parser for creating the logical structure, i.e. the parse tree. Next, the ANTLR4 listener allows communication with Drools every time a node in the parse tree is entered. The listener takes information from the tokens and sends it to Drools. For this communication, a POST request has been created, named “addFact”. This procedure is depicted in the sequence diagram as a synchronous invocation to the Pattern Engines’ activation, labelled as “add requirement”.

As soon as the Drools facts reach one of the Pattern Engines, instances are created from the corresponding Java classes and the received information is stored at the class attributes. During the last step, the created java instances are inserted as facts into the knowledge session. These Drools facts are used by Drools rules, which are fired when a condition is met.

[6] https://www.antlr.org/

The requirement status is returned by the Pattern Engines as an answer to Pattern Orchestrator for every “add requirement” invocation. The received answer is then transmitted by the Pattern Orchestrator to the Recipe Cooker.

3.1.5. PATTERN ORCHESTRATOR INTEGRATION WITH THE SDN/NFV FOR SERVICE FUNCTION CHAINING

One of the scopes of SEMIoTICS is to provide security guarantees through the traffic forwarding via different network security functions by applying the Service Function Chaining (SFC; as detailed in deliverable D2.5 and D3.2). Considering the different types of traffic reaching the backend where the chaining of services will take place, a variety of intricacies can be observed such as of low trust and low priority, low bandwidth and latency, medium trust but high priority, medium trust and of low priority, and finally high trust and high priority, as low latency and relatively high bandwidth. To achieve this goal, the SEMIoTICS framework has integrated a number of different software components in all the layers as can be seen in Figure 3-12 . Apart from the layer separation (application, network or field), the involved components can be separated into two types, expressed also with different colours, with red the design of the control flow components and with blue the runtime data flow involved components.

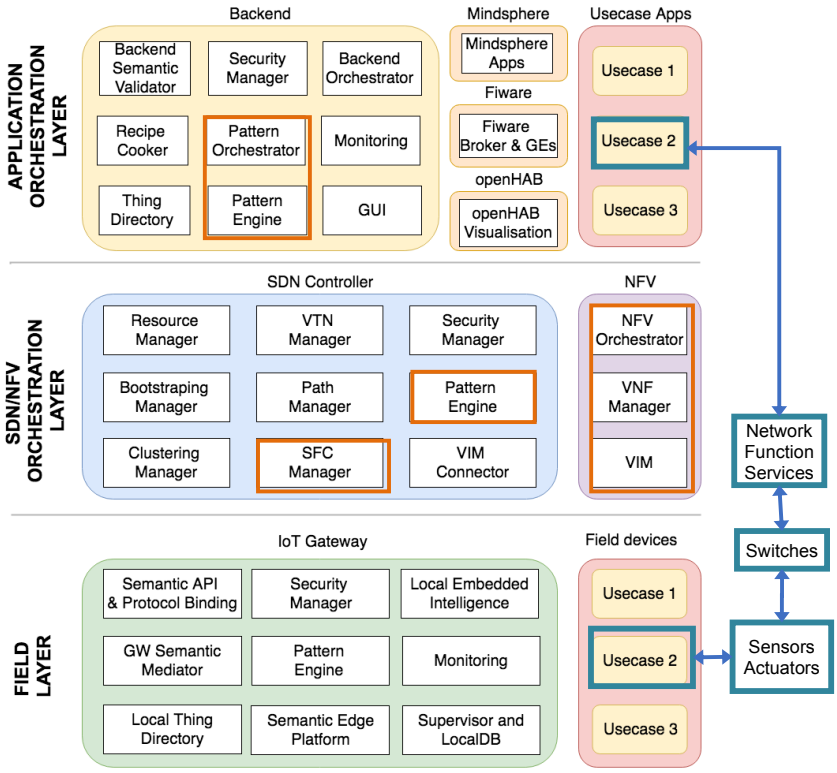


FIGURE 3-12 INTEGRATION OF PATTERN FRAMEWORK AND SERVICE FUNCTION CHAINING

The design of an efficient control flow mechanism is required to be used not only to verify SFC and VNFs but also to instantiate them for assuring the SPDI requirements (**KPI 2.1**) based on the enforcement of the respective SPDI patterns. When an SFC cannot be verified, the required VNFs are requested by the VIM via NFV Orchestrator to identify them or to instantiate them if they do not exist. More specifically, the components which are involved in the control flow are:

- The **Pattern orchestrator** is able to forward pattern rules and trigger the SFC requirements to the pattern engine
- The **Pattern engine** (backend and SDN) for enabling the pattern rules to address the SFC requirements such as instantiate or verify SFCs
- The **SFC manager** in the SDN controller to identify and configure service function chains
- The **SDN Controller:** is responsible to interact with the switches and the VNFs together with the pattern engine and the SFC manager.
- The **NFV orchestrator** to identify available VNFs as instantiated in the **VIM**.

The procedure of instantiation and the identification of the respective SFCs and the VNFs based on the patterns is depicted in Figure 3-13 including the following interactions with the components of the SEMIoTICS architecture. The Pattern orchestrator forwards a specific chain request to the pattern engine for forwarding the traffic between entities through a specific chain of functions. Pattern engine forwards this request to the SFC manager which is located in the SDN controller responding to the pattern engine whether the chain exist or not. If the chain exists, then a respond of the chain satisfaction is returned to the pattern orchestrator. If the chain does not exist, then a requested is forwarded to the VIM asking whether the service functions exist or not. If functions exist in the VIM, then the chain can be instantiated in the SFC Manager and a respond of the chain satisfaction is returned to the pattern orchestrator. If functions do not exist in the VIM then, a function instantiation request is forwarded to the NFV Orchestrator, which is responsible to instantiate them in the VIM. Then, the chain can be instantiated in the SFC Manager and a respond of the chain satisfaction is returned to the pattern orchestrator.

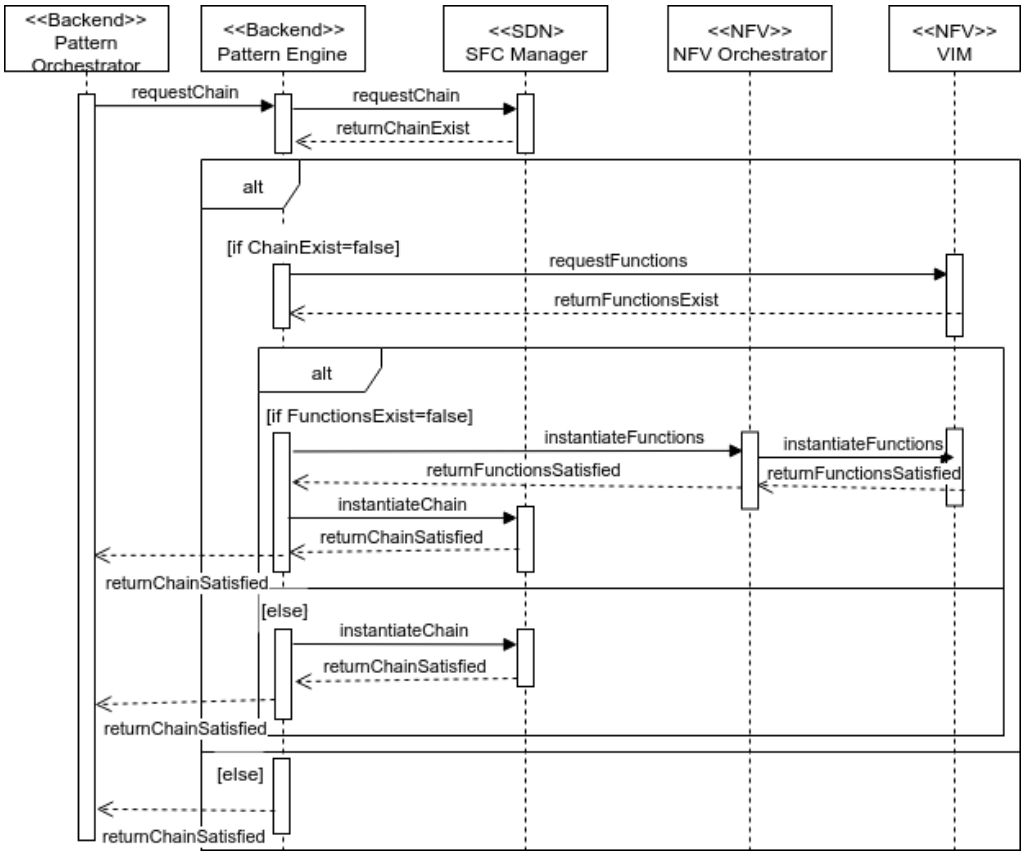


FIGURE 3-13 SEMIOTICS SFC CONTROL FLOW

The integration of the different components that participate in the control flow configuration especially with the pattern engine and the SFC Manager is done through the exposed interfaces of the SFC manager where the pattern engine can send and receive requests. More specifically, the SFC manager exposes REST interfaces able to instantiate the respective Service Functions and Chains by the insertion of respective templates in JSON formats. In addition, the ACL, the classifiers, and the forwarders can be defined based on the respective REST interfaces. In Table 3, the JSON syntax of data which is expected by the SFC manager and the address is provided.

TABLE 3 SFC COMPONENTS AND JSON TEMPLATES

Service Function (SF)	
JSON Syntax (data)	*"service-function": [{"name","ip-mgmt-address", "rest-uri","type", "nsh-aware", "sf-data-plane-locator": [{"name","port","ip","transport", "service-function-forwarder"}] }]*
URL (uri)	*/restconf/config/service-function:service-functions/*
Service Function Forwarder (SFF)	
JSON Syntax (data)	"service-function-forwarder": [{"name","service-node", "service-function-forwarder-ovs:ovs-bridge": {"bridge-name"}, "sf-data-plane-locator": [{"name","port","ip","transport", "service-function-forwarder"}] }], "service-function-dictionary": [{"name", "sff-sf-data-plane-locator": {"sf-dpl-name", "sff-dpl-name" }}]
URL (uri)	*/restconf/config/service-function-forwarder:service-function-forwarders*
Classifier	
JSON Syntax (data)	"service-function-classifier": [{"name","scl-service-function-forwarder": [{"name", "interface"}], "acl":{"name","type"}]
URL (uri)	*/restconf/config/service-function-classifier:service-function-classifiers/*
Service Function Chain	
JSON Syntax (data)	"service-function-chain": [{"name", "symmetric", "sfc-service-function": [{"name", "type"}, {"name", "type"}]
URL (uri)	*/restconf/config/service-function-chain:service-function-chains/*

Regarding the data flow, traffic classification is based on the predefined SFC for providing secure chains to forward the different kind of traffic of this use case (**KPI 5.2**). Through the definition of said chains, each of the traffic types gets routed through a chain of service functions tailored to its intrinsic requirements and characteristics, such as QoS and trust levels, and, by extension, its desired SPDI properties. These services are "stitched" together to create a service chain, with numerous options for adaptations when required (e.g. to adapt to link failures). The flexible traffic steering towards network functions enabled by SFC can also be leveraged to integrate novel, adaptable security services, such as steering suspicious traffic to security appliances. The deployment of these enhanced security concepts is in line with the enhanced protection requirements of certain sensitive application domains, such as critical infrastructures, given that the old paradigm of perimeter defences and trusted internal networks is obsolete, as recent attacks have demonstrated. Considering the latter, another important element in the operation of the above is the SPDI-based management of the various involved components and their compositions, through the Pattern-based framework that is in the core of the SEMIoTICS approach. In addition, the involved components in data flow are the following:

- The **Use case field devices** can contain sensors and actuators.
- The **Open Virtual Switches (OVS)** are programmable switches supporting OpenFlow rules able to interact with the SDN Controller. Two main roles of OVS switches as classifiers (to classify the traffic) and as forwarders (to forward the traffic to the respective VNF). An OVS switch can be Virtual (i.e. as a Virtual or Physical).
- The **Virtual Network Functions (VNFs)** are responsible to manage the traffic and express the service functions as described previously. That may include a firewall, IDS, Load-Balancer, Deep Packet

Inspection (DPI) or a honeypot hosted by the VIM and handled both by the SFC manager and the NFV orchestrator.

- The **Use case application** are responsible to interact with the distributed field layer use case devices.

The procedure depicted in Figure 3-14 presents the traffic classification either from a use case device or an application through a number of different service function (security VNFs) that constitute a chain. The classifier is responsible to identify the type of traffic based on specific predefined ACLs including characteristics such as IP and port, to forward to the respective chain.

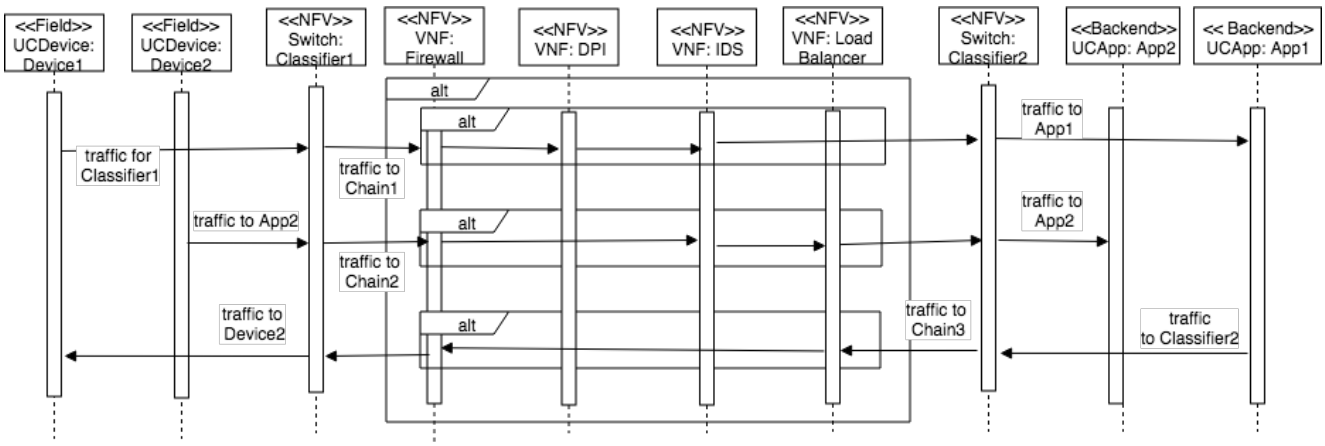


FIGURE 3-14 SEMIOTICS SFC DATA FLOW

There are a number of different methods to use the exposed interfaces and to insert the required SFC configurations in the SFC Manager. Each of these methods is related to application that uses these interfaces. The exposed REST APIs interface is used with a Python function to PUT configurations (i.e., classifiers, forwarders etc.) from the command line enabling semi-dynamic configurations in the SFC Manager as presented in Figure 3-15.

```
def put(host, port, uri, data, debug=False):
    '''Perform a PUT rest operation, using the URL and data provided'''

    url='http://'+host+":"+port+uri

    headers = {'Content-type': 'application/yang.data+json',
               'Accept': 'application/yang.data+json'}
    if debug == True:
        #print "PUT %s" % url
        #print json.dumps(data, indent=4, sort_keys=True)
        r = requests.put(url, data=json.dumps(data), headers=headers, auth=HTTPBasicAuth(USERNAME, PASSWORD))
    if debug == True:
        #print r.text
        r.raise_for_status()
        time.sleep(1)
```

FIGURE 3-15 REST CALLS SFC CONFIGURATION IN PYTHON

On the other hand, JAVA is required to GET or PUT configurations (i.e. chains, functions, ACLs etc.) as required or provided by the pattern rules enabling dynamic configurations in the SFC Manager via REST APIs as expressed in Figure 3-16 and Figure 3-17.

```java
public class GetData {
        public GetData(host, port, uri) {
            URL url = new URL("http://"+host+":"+port+uri);
            String name = USERNAME;
            String password = PASSWORD;
            String authString = name + ":" + password;
            String authStringEnc = Base64.getEncoder().encodeToString(authString.getBytes());
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            conn.setRequestMethod("GET");
            conn.setRequestProperty("Accept", "application/json");
            conn.setRequestProperty("Authorization", "Basic " + authStringEnc);
            br = new BufferedReader(new InputStreamReader(conn.getInputStream()));
            while ((line = br.readLine()) != null) {
                jsonData += line + "\n";
            }
        }
}
```

FIGURE 3-16 GET REST CALL FOR SFC CONFIGURATION IN JAVA

```java
public class PutData {
        public PutData(host, port, uri, jsonData) {
            URL url = new URL("http://"+host+":"+port+uri);
            HttpURLConnection conn = (HttpURLConnection) url.openConnection();
            String name = USERNAME;
            String password = PASSWORD;
            String authString = name + ":" + password;
            String authStringEnc = Base64.getEncoder().encodeToString(authString.getBytes());
            conn.setDoOutput(true);
            conn.setDoInput(true);
            conn.setUseCaches(false);
            conn.setRequestMethod("PUT");
            conn.setRequestProperty("Content-Type", "application/json");
            conn.setRequestProperty("Accept", "application/json");
            conn.setRequestProperty("Authorization", "Basic " + authStringEnc);
            OutputStreamWriter osw = new OutputStreamWriter(conn.getOutputStream());
            conn.connect();
            osw.write(jsonData);
            osw.flush();
            osw.close();
            conn.disconnect();
            System.err.println(conn.getResponseCode());
    }
}
```

FIGURE 3-17 PUT REST CALL FOR SFC CONFIGURATION IN JAVA

The instantiation of the service functions to configure the data flow in the SFC Manager can be given by the insertion of a JSON file such as the one depicted in Figure 3-18. The file is inserted by the use of either the deployed PYTHON function or the JAVA supporting either the semi-dynamic or the dynamic one in relation also with the enabled pattern rule. In the list of the service functions, the firewall, the DPI, the IDS and the Load balance have been defined as the most crucial ones to enable the SPDI properties required by each chain to guarantee. Each VNF has a unique IP address which is required for the configuration and integration with the other functions interacting also with the use case devices and apps. The insertion of the service functions in the SFC manager can be given as follows:

put(controller, port, */restconf/config/service-function:service-functions/*, service-functions, True)

```
{ "service-functions": {
    "service-function": [
        {
            "name": "firewall-1",
            "ip-mgmt-address": firewall,
            "rest-uri": "http://"+firewall+":5000",
            "type": "firewall",
            "nsh-aware": "true",
            "sf-data-plane-locator": [
                {
                    "name": "firewal-1-dpl",
                    "port": 6633,
                    "ip": firewall,
                    "transport": "service-locator:vxlan-gpe",
                    "service-function-forwarder": "SFF1"
                }
            ]
        },
        {
            "name": "dpi-1",
            "ip-mgmt-address": dpi,
            "rest-uri": "http://"+dpi+":5000",
            "type": "dpi",
            "nsh-aware": "true",
            "sf-data-plane-locator": [
                {
                    "name": "dpi-1-dpl",
                    "port": 6633,
                    "ip": dpi,
                    "transport": "service-locator:vxlan-gpe",
                    "service-function-forwarder": "SFF2"
                }
            ]
        },
        {
            "name": "ids-1",
            "ip-mgmt-address": ids,
            "rest-uri": "http://"+ids+":5000",
            "type": "ids",
            "nsh-aware": "true",
            "sf-data-plane-locator": [
                {
                    "name": "ids-1-dpl",
                    "port": 6633,
                    "ip": ids,
                    "transport": "service-locator:vxlan-gpe",
                    "service-function-forwarder": "SFF3"
                }
            ]
        },
        {
            "name": "loadbalancer-1",
            "ip-mgmt-address": loadbalancer,
            "rest-uri": "http://"+loadbalancer+":5000",
            "type": "qos",
            "nsh-aware": "true",
            "sf-data-plane-locator": [
                {
                    "name": "loadbalancer-1-dpl",
                    "port": 6633,
                    "ip": loadbalancer,
                    "transport": "service-locator:vxlan-gpe",
                    "service-function-forwarder": "SFF3"
                }
            ]
        }
    ]
  }
}
```

FIGURE 3-18 SERVICE FUNCTION JSON DATA SFC CONFIGURATION

The instantiation of a sequence of functions can constitute a service chain as can be seen in Figure 3-19. Similar to the insertion of service functions in the SFC manager through the exposed service function REST interface, service chains can be inserted by the use of the PYTHON or JAVA functions:

put(controller, port, "*/restconf/config/service-function-chain:service-function-chains/*", service-function-chain, True)

```
{
    "service-function-chain": [
            {
                "name": "SFC1",
                "symmetric": "true",
                "sfc-service-function": [
                    {
                       "name": "firewall-abstract1",
                       "type": "firewall"
                    },
                    {
                        "name": "dpi-abstract1",
                        "type": "dpi"
                    },
                    {
                        "name": "ids-abstract1",
                        "type": "ids"
                    }
                ]
            }
        ]
}
```

FIGURE 3-19 SERVICE CHAIN JSON DATA SFC CONFIGURATION

Finally, the last step of the software integration for function chaining is based on the instantiation of the SFC when a VNF does not exist or is failed in the VIM (OpenStack). In this case, the pattern engine uses the exposed by the NFV orchestrator (OSM) interface to instantiate a VNF based on the VNF catalog of all usable VNFDs (VNF Descriptors) as described in the D3.2. The role of the pattern engine in this case is to react as the OSS/BSS (Operations Support System and Business Support System) to support service chaining requirements either at design or at runtime.

3.1.6. INTEGRATION OF SEMANTIC BACKEND VALIDATOR WITH OTHER COMPONENTS

The main purpose of the Backend Semantic Validator (BSV) component is to tackle the semantic interoperability issues that arise in the SEMIoTICS framework (see Deliverable D4.4), at the application orchestration layer. In fact, the component is responsible for the mapping between data types to ensure that data flow is possible between smart objects (Things, i.e. Sensor, Actuator). Moreover, semantic transformation methods (Adaptor Nodes) have been developed with the purpose of resolving, if possible, conflicts among the semantic annotations.

The components of the SEMIoTICS architecture that are involved in this process are the BSV which is responsible for semantic validation mechanisms; the Thing Directory component that are the repository of knowledge containing the necessary Thing models; the Recipe Cooker component, which is responsible for cooking (creating) recipes reflecting user requirements on different layers (cloud, edge, network) as well as transforming recipes into understandable rules for each layer and includes the Adaptor Nodes to resolve semantic conflicts. It uses the Thing Directory with all the models required to create these rules. At the field layer, the GW Semantic Mediator (GWSM) component for the semantic mapping between different data

models; the Semantic API Protocol Binding (SAPB) component for binding different protocol and exposing a common semantic API located at the Generic IoT Gateway layer (see Table 4).

TABLE 4 LIST OF COMPONENTS THAT INTERACT WITH THE BSV COMPONENT

Component	Components that will be used/consumed by this component	Layer of component that will be consumed	Description of interactions
Backend Semantic Validator	*Thing Directory*	*Backend*	Searching for the necessary Thing models in Thing Directory component, in order to detect any potential semantic conflicts between the interacting domains
	Recipe Cooker	*Backend*	Connecting with Recipe Cooker to resolve these semantic conflicts using the Adaptor Nodes that configure an Interaction Pattern in accordance with the application's requirements.
	Semantic API & Protocol Binding	*Field*	Transferring the translated request to the Semantic API & Protocol Binding component which is responsible to trigger the GW Semantic Mediator in the filed layer, in order to send the request in an appropriate format to the target Thing (actuator).

The functionality of this component consists of three basic steps:

1. Searching for the necessary Thing models in the Thing Directory component to detect any potential semantic conflicts between the interacting domains.
2. Connecting with Recipe Cooker and Semantic Edge Platform (in the field) to resolve these semantic conflicts using the Adaptor Nodes that configure an Interaction Pattern in accordance with the application's requirements.
3. Transferring the translated request to the Semantic API & Protocol Binding component which is responsible to trigger the GW Semantic Mediator in the filed layer to send the request in an appropriate format to the target Thing (actuator).

The procedure of the semantic interoperability mechanisms between the backend and the field layer is highlighted by a sequence diagram in Figure 3-20.

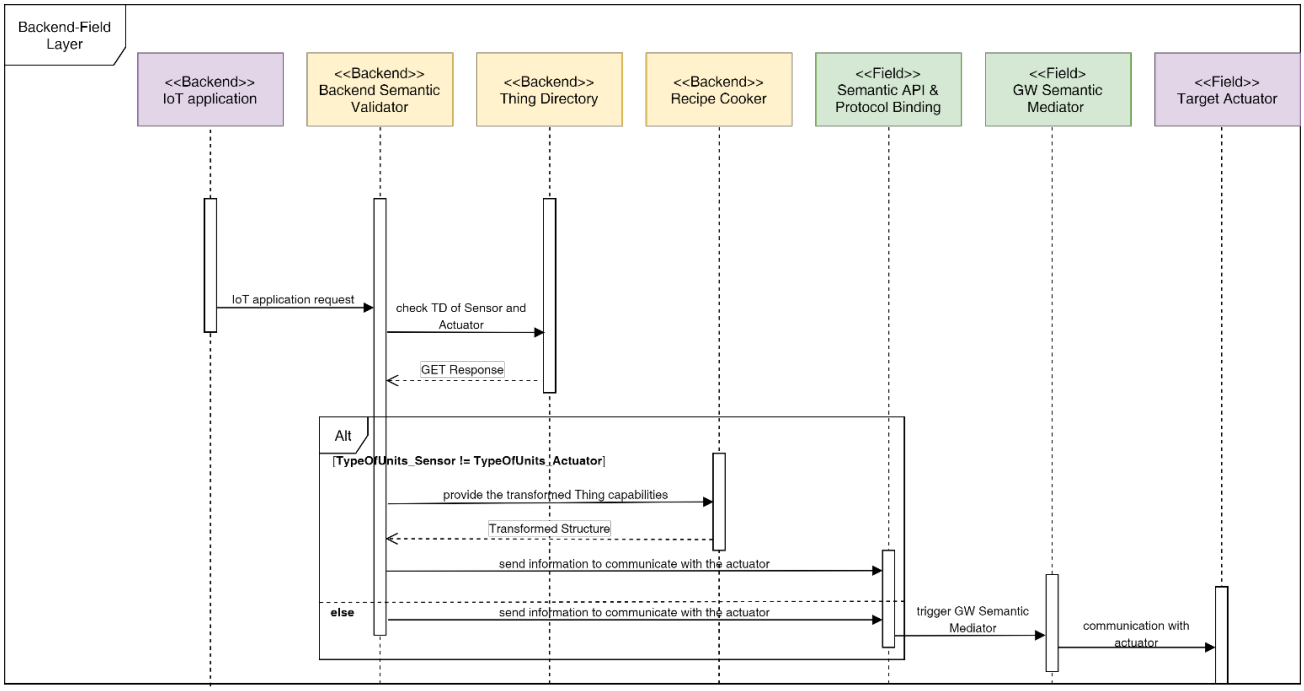


FIGURE 3-20 SEQUENCE DIAGRAM FOR SEMANTIC INTEROPERABILITY MECHANISMS

In Cycle 1, the first and the second step of the functionality of this component have been developed. This implementation includes the interaction of BSV with the Recipe Cooker and Thing Directory at the application orchestration layer. Particularly, based on the component requirements, two main POST service requests have already been developed; the ***validateData*** and the ***validateRecipeFlow*** POST request service for the first and the second step of the above functionality respectively.

- validateData POST service request (see Figure 3-21): it receives a request from the IoT application, in JSON-LD/JSON format. The JSON-LD/JSON Parser is implemented as part of the BSV component, in order to analyze the received input and extract the meaningful information from these set of data. After that, the BSV interacts with the Thing Directory component; this stage consists of two procedures, the TD discovery of the specific Thing and the TD registration for the case that this Thing is not included in the Thing Directory. In the first case, the send GET function is developed that uses HttpURLConnection to send an HTTP GET request to Thing Directory in order to get the search result. For this discovery, SPARQL query can be used to retrieve TDs based on their IDs and should be percent-encoded. Depending on the above result, if the TD of the Thing is not in the Thing Directory, a POST request in Thing Directory was implemented for the registration of the new TD.

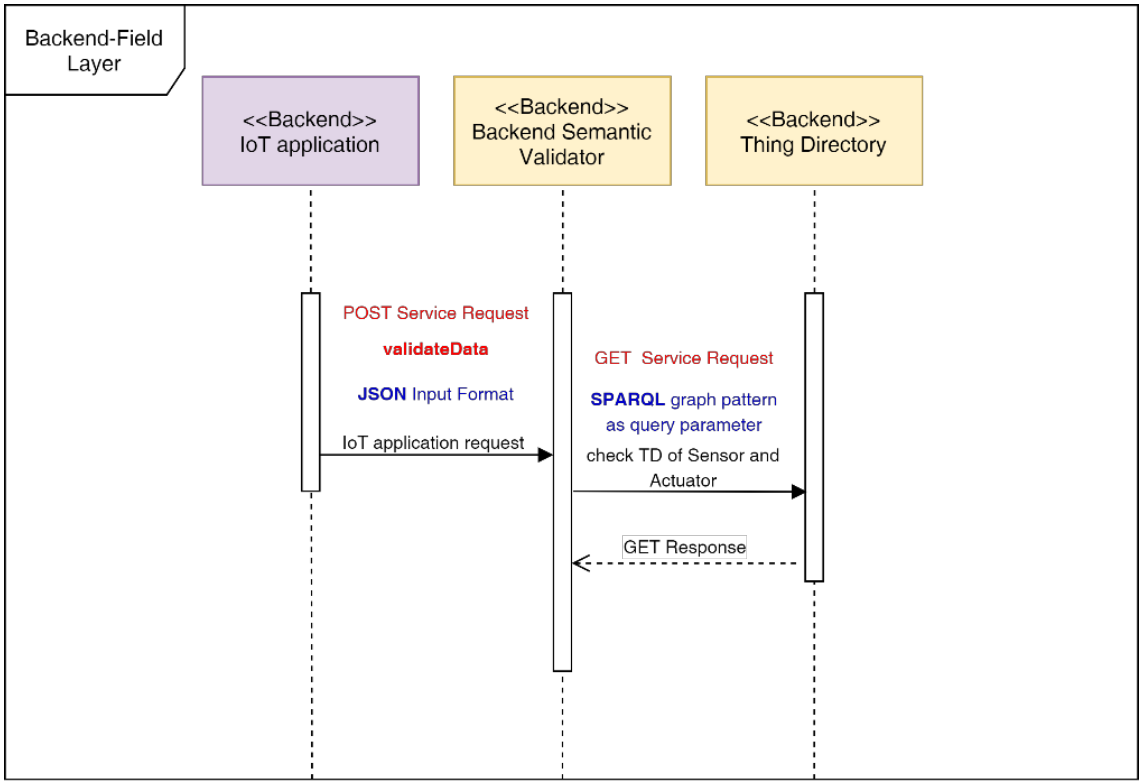


FIGURE 3-21 SEQUENCE DIAGRAM - FIRST POST SERVICE REQUEST BSV

- validateRecipeFlow POST service request (see Figure 3-22): it receives a request from Recipe Cooker in JSON format (the recipe flow). This request aims to trigger BSV to check for any interoperability conflicts between the two Things of the specific recipe. Next, the BSV component connects with the Thing Directory component to ensure that these specific Things have already been registered in order to receive information on their TDs. This is a required step, otherwise, the BSV cannot resolve semantic differences and ensure that data flow is possible between them. The BSV parses the TDs to discover for the semantic interoperability between the connected Things. In this phase, there are two possible cases, the interacting Things used the same data transformation techniques and the interacting Things used the different data transformation techniques. In the second case, the BSV searches in Recipe Cooker for the corresponding Adaptor Node. If the Adaptor Node does not exist, the BSV should develop and add it in the Recipe Cooker. Finally, the BSV sends the response back to Recipe Cooker, using JSON format, with the updated flow, which has a new “wire” with the Adaptor Node between two initial Things (ingredients) of the recipe. The updated flow can be imported and saved by the Recipe Cooker. The advantage of this process is that after resolving the semantically interoperable conflicts between these two specific Things, in any future interaction that will be required for these, the Adapter Node will be added to the corresponding recipe to ensure semantic interoperability.

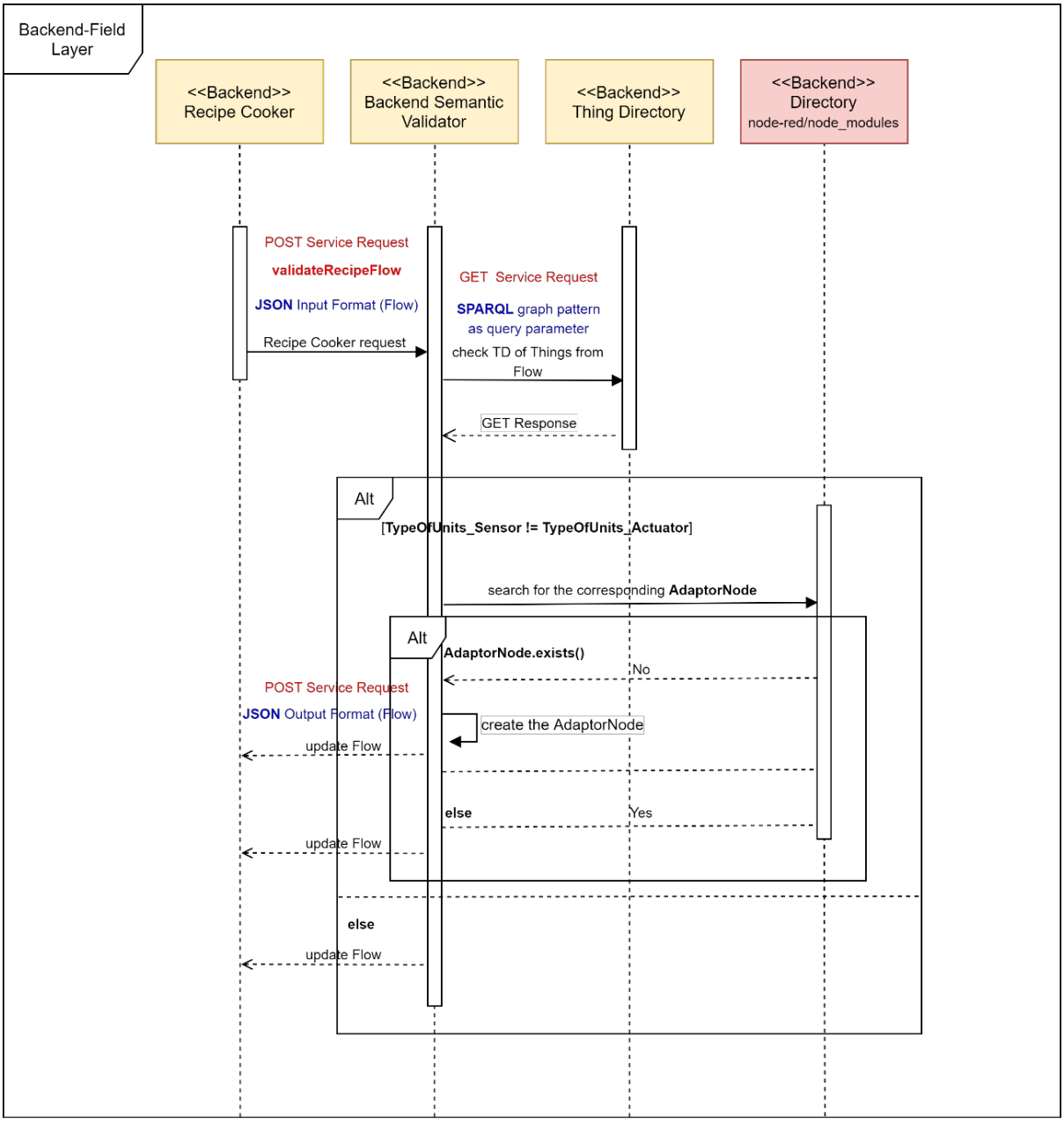


FIGURE 3-22 SEQUENCE DIAGRAM - SECOND POST SERVICE REQUEST BSV

3.2. Integration flows delivered in cycle 2

3.2.1. GUI INTEGRATION WITH SECURITY MANAGER

Every application which deals with sensitive and confidential data should be secured. This can be accomplished in many ways - one way to provide security to web applications is to use OAuth2[7]. It is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service. It works by delegating user authentication to the service that hosts the user account and authorizing third-party applications to access the user account. OAuth2 provides authorization flows for web and desktop applications, and mobile devices. During the second cycle of the development, it was decided to have both GUI's API and web application secured and since Security Manager can be served as OAuth2 provider it was agreed to use it as one.

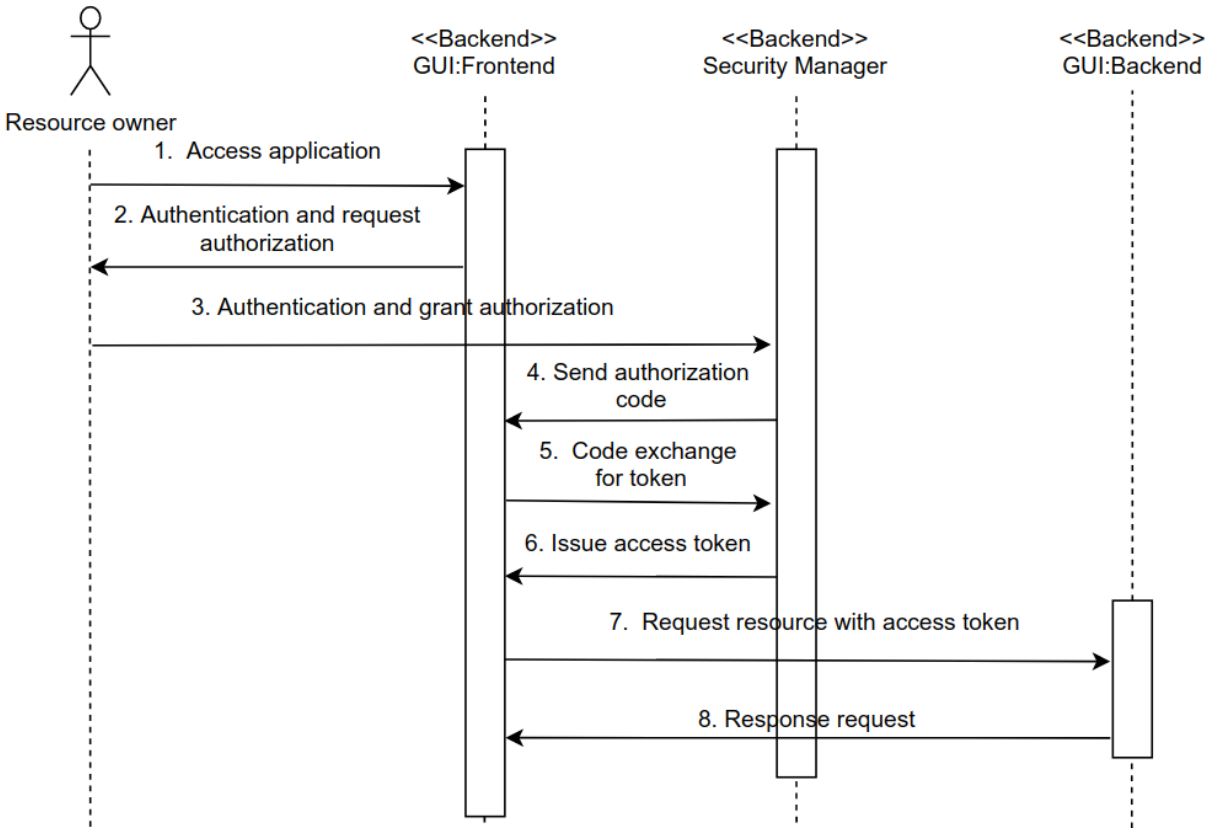


FIGURE 3-23 OAUTH CODE GRANT FLOW

1. Access Application: The user accesses GUI and triggers authentication and authorization.
2. Authentication and Request Authorization: The GUI redirects the user to the Security Manager login page where it prompts the user for their username and password.

[7] https://oauth.net/2/

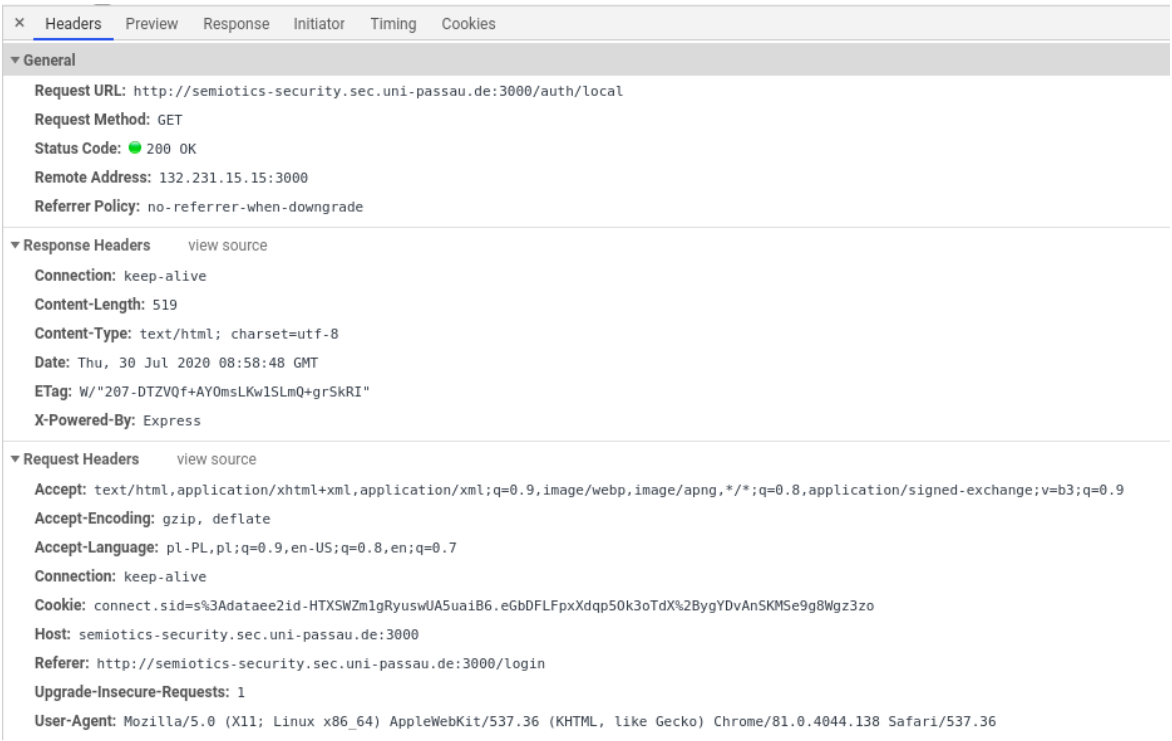


FIGURE 3-24 REDIRECT TO SECURITY MANAGER'S LOGIN FORM

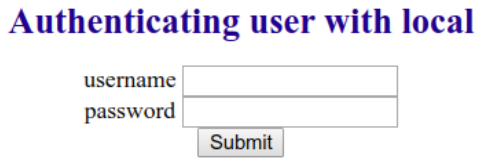


FIGURE 3-25 SECURITY MANAGER'S LOGIN FORM

3. Authentication and Grant Authorization: Security Manager receives the authentication and authorization will be checked: If user's credentials are invalid, the request's response is 302 Found and the user is redirected back to the login page. As soon as credentials are valid, the user is redirected further and the workflow will continue.

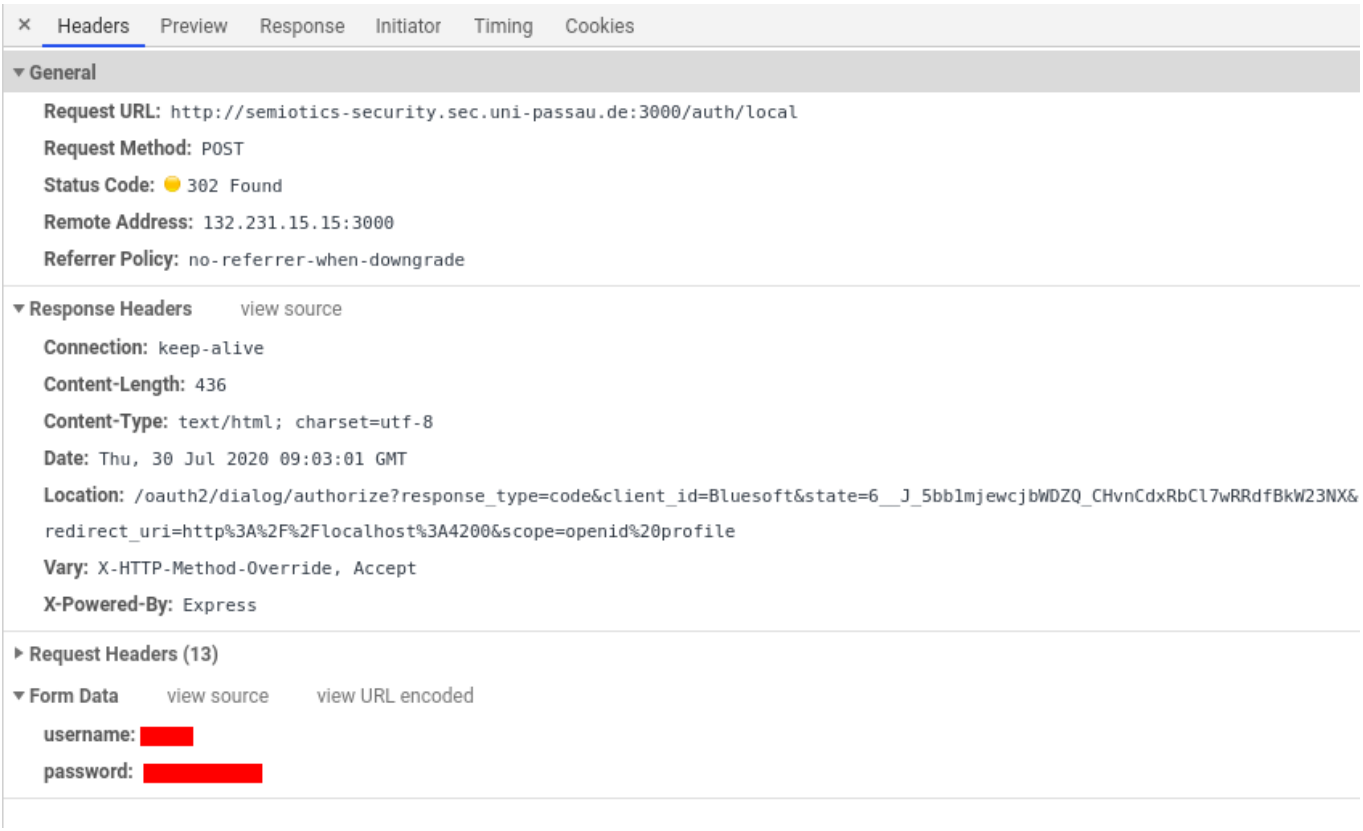


FIGURE 3-26 REDIRECT AFTER SUCCESSFUL LOGIN

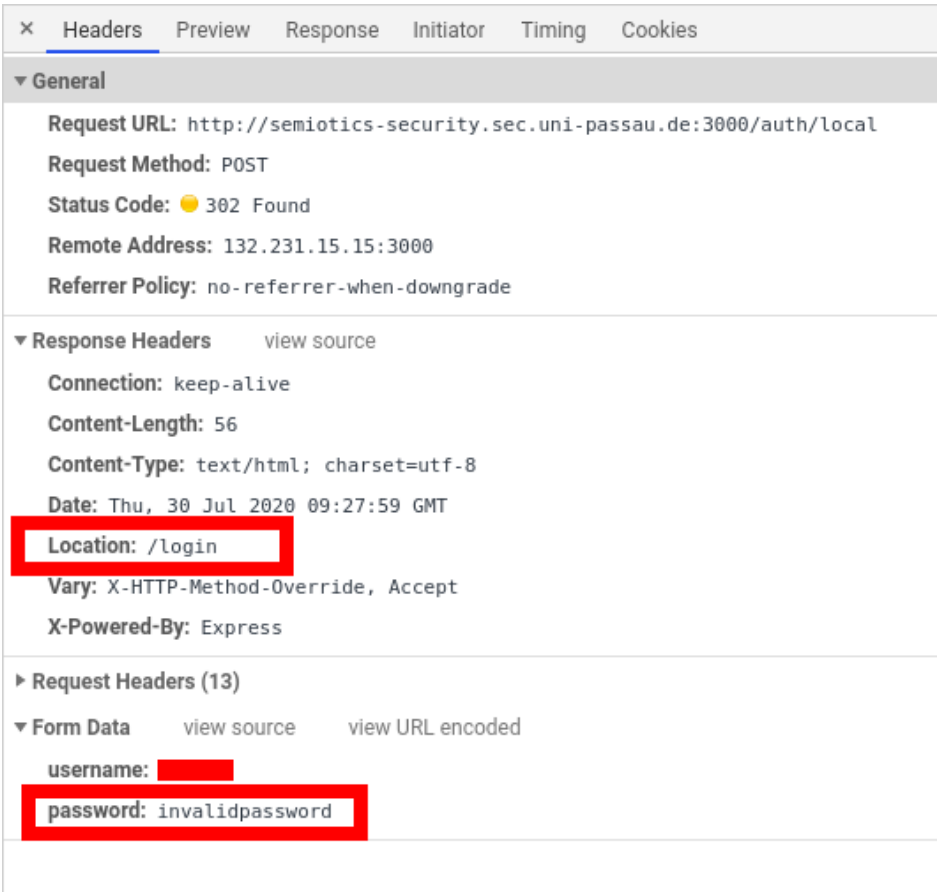


FIGURE 3-27 REDIRECT BACK TO THE LOGIN FORM AFTER UNSUCCESFUL LOGIN

4. Send Authorization Code: After the user authorizes the app, the Security Manager generates an authorization code and sends it back to the GUI as URL parameter.

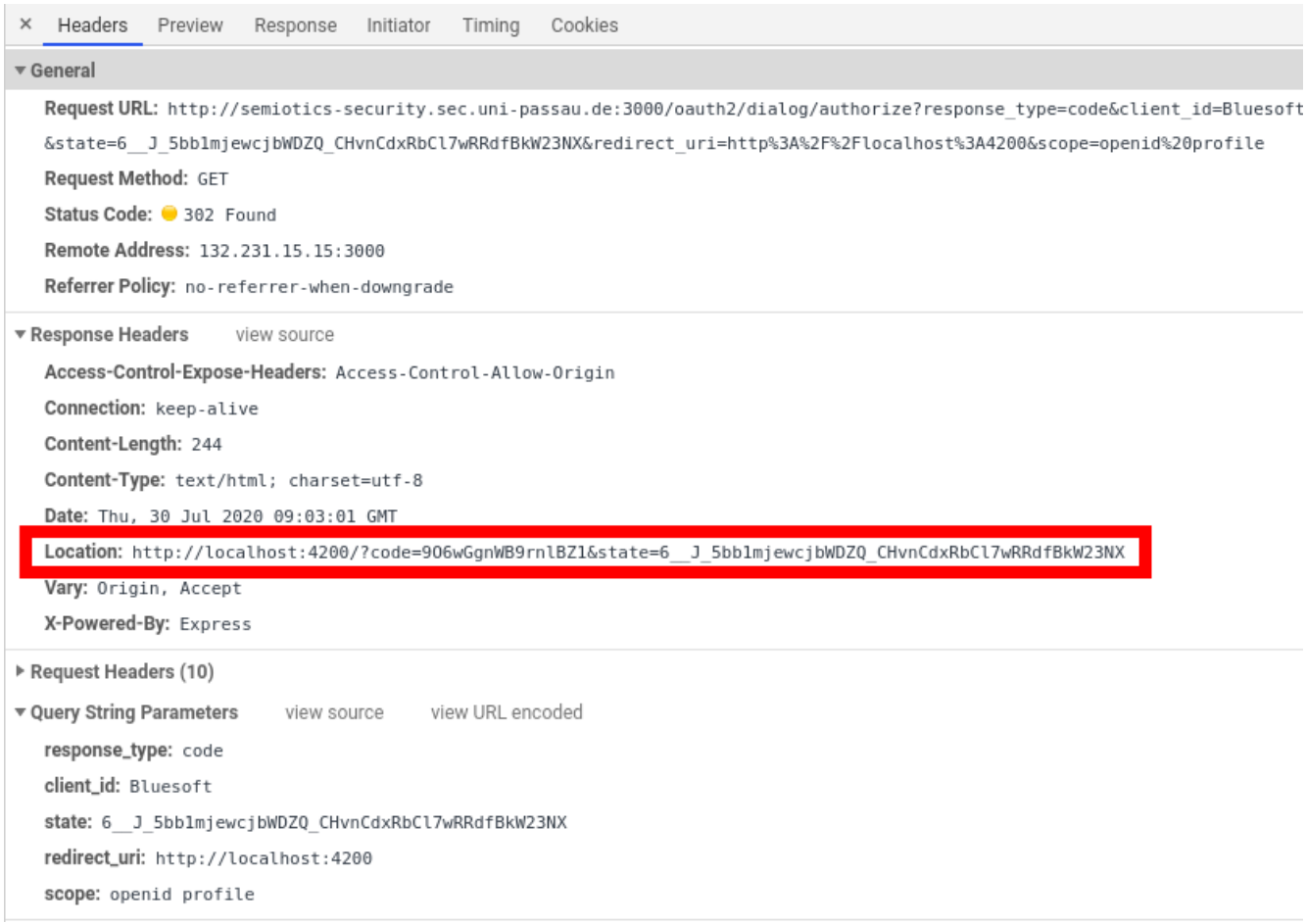


FIGURE 3-28 GENERATING AUTHORIZATION CODE

5. Request Code Exchange for Token: The GUI uses the authorization code to request an access token from Security Manager.

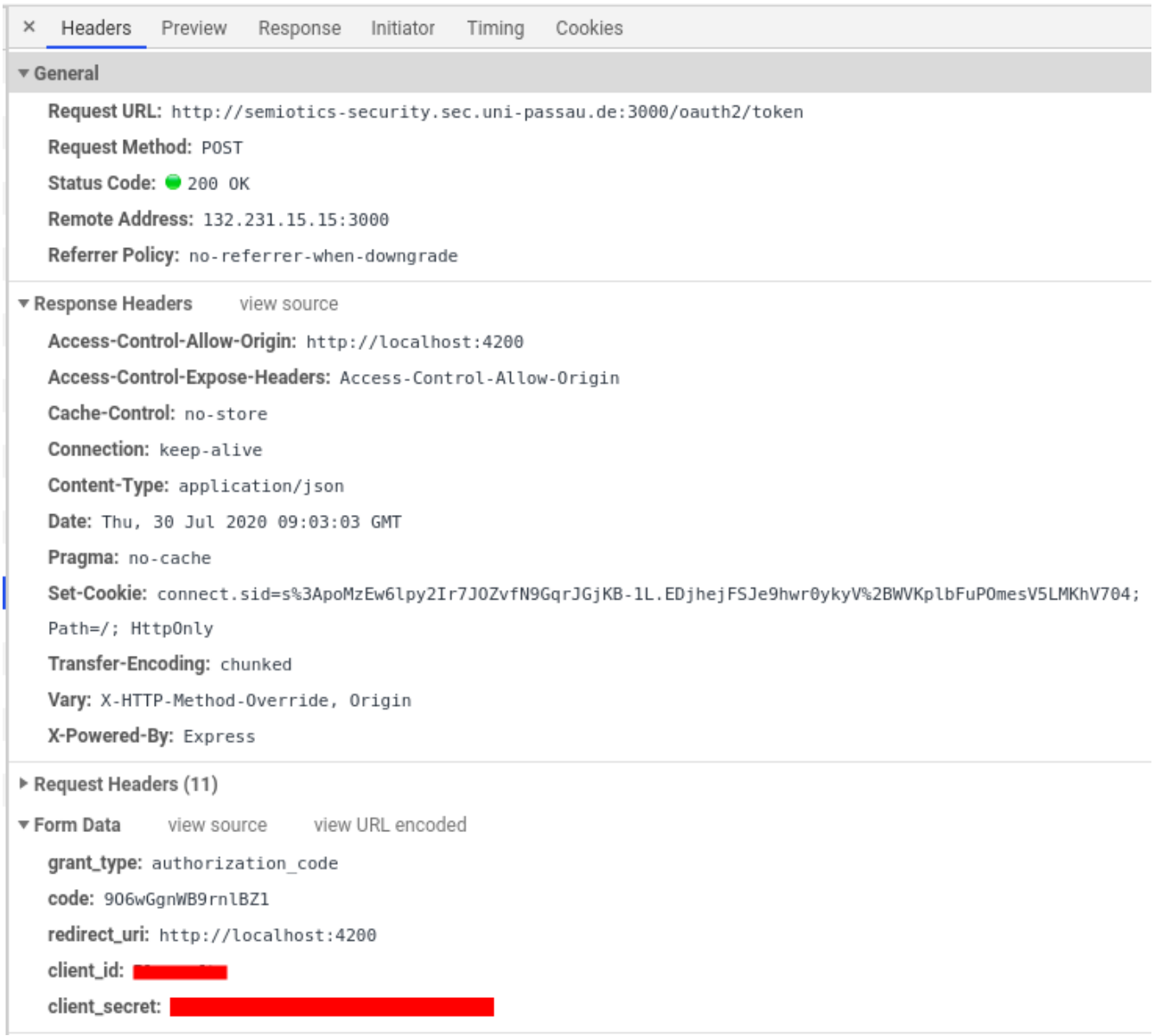


FIGURE 3-29 HTTP REQUEST TO GET AUTHORIZATION TOKEN

6. Issue Access Token: Security Manager validates the authorization code and if valid issues an access token.

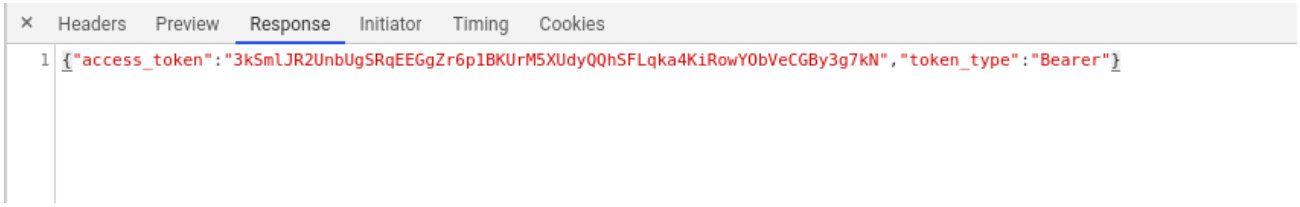


FIGURE 3-30 THE AUTHORIZATION TOKEN

7. Request Resource w/ Access Token: The GUI attempts to access the resource from the resource's server by adding the access token to every HTTP request.

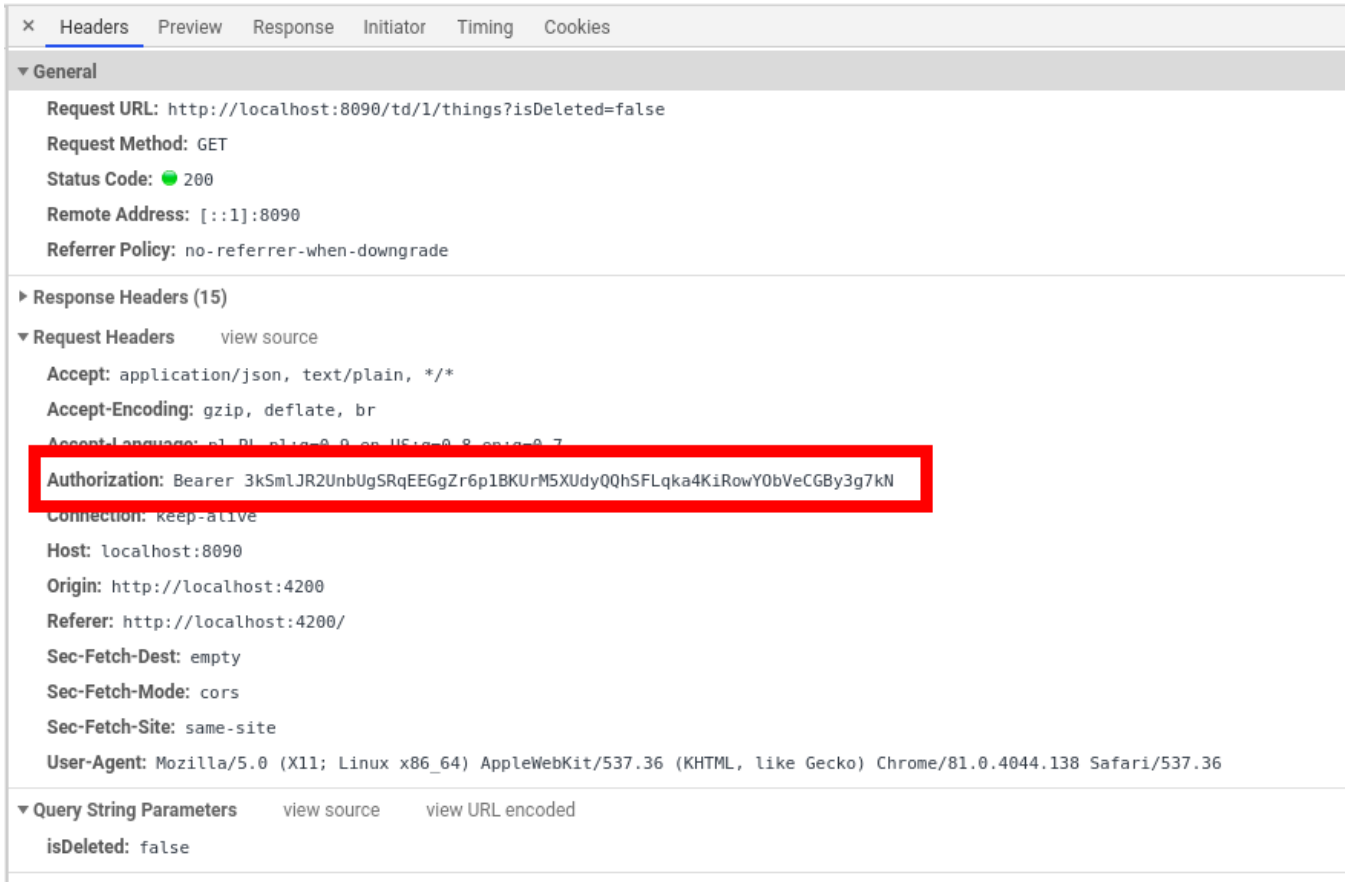


FIGURE 3-31 AN EXAMPLE OF HTTP REQUEST WITH AUTHORIZATION TOKEN

8. Return Resource: To have API secured, an authentication filter was implemented. The filter intercepts every HTTP request destinated for GUI:Backend and checks whether the token is present in the request header; if it is then the token is sent back to Security Manager. By doing so we are able to validate the token and get the information about the resource owner. If the token is valid and hasn't

expired, the response code is 200 and the response body is a JSON which contains all the essential information about the resource owner and token's expiration date (Figure 3-32). On the other hand, if the token had been forged or it had expired by the time request is sent, the response code is 400 meaning that the original request should be denied access to the resource (Figure 3-33). This enforces strict access control for all applications within the GUI with the help of the Security Manager.

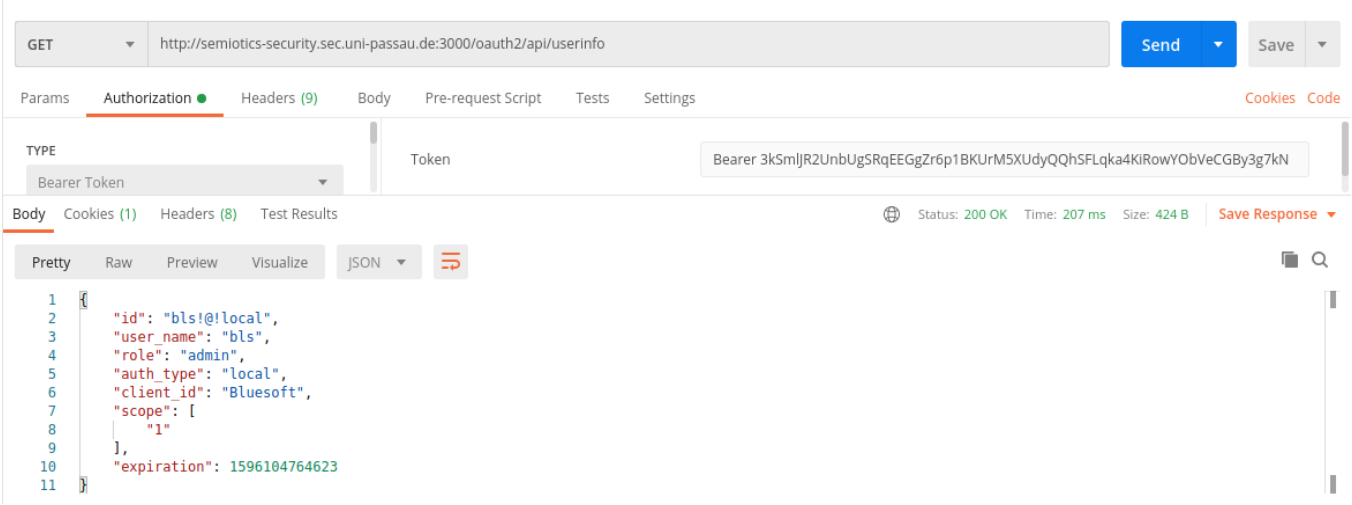


FIGURE 3-32 USER DETAILS RESPONSE

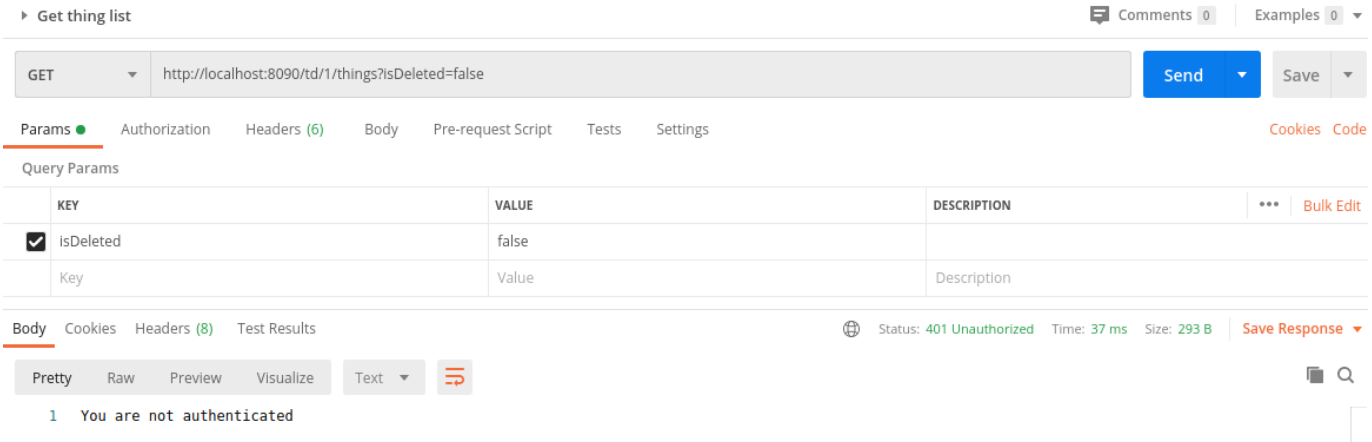


FIGURE 3-33 API RESPONSE DUE TO THE INVALID TOKEN

3.2.2. GUI INTEGRATION WITH MONITORING COMPONENT

In the second cycle, the integration with Monitoring Component has been developed. The Monitoring API is described according to OpenAPI specification and is shown in Figure 3-34

```yaml
swagger: '2.0'
info:
  description: A draft version of the MPD APIs
  version: v0.2
  title: 'SEMIoTICS Monitoring, Prediction and Diagnosis (MPD) APIs'
  contact:
    name: Domenico Presenza
    url: www.eng.it
    email: domenico.presenza@eng.it
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
host: 'localhost:8080'
basePath: /semiotics/api
tags:
  - name: mdpqueries
schemes:
  - http
paths:
  /semiotics/api/mdp/queries:
    put:
      tags:
        - mdpqueries
      summary: >-
        Submit the query specified in the request body. Returns the id given by
        the MPD to the corresponfd monitoring task
      description: >-
        Monitoring component uses events streams generated by signallers to
        match patterns sepcified by the query.
      operationId: run
      parameters:
        - in: body
          name: body
          description: The query used to filter events
          required: true
          schema:
            $ref: '#/definitions/Query'
      responses:
        '200':
          description: successful operation
          schema:
            type: string
  '/semiotics/api/mdp/queries/notifications/{queryId}':
    delete:
      tags:
        - mdpqueries
      summary: Deletes the query identified by the path variable 'queryId'
      description: ''
      operationId: cancel
      parameters:
        - name: queryId
          in: path
          description: Identifier of the query.
          required: true
          type: string
      responses:
        '200':
          description: successful operation
          schema:
            type: boolean
  '/semiotics/api/mdp/queries/{qid}/running':
    get:
      tags:
        - mdpqueries
      summary: Checks whether the query indicated by the path variable 'qid' is running
      description: Returns the number of currently ongoing quries.
      operationId: isRunning
      parameters:
        - name: qid
          in: path
          description: Identifier of the query.
          required: true
          type: string
      responses:
        '200':
          description: successful operation
          schema:
            type: boolean
definitions:
  Duration:
    type: object
    properties:
      seconds:
        type: integer
        format: int64
      nano:
        type: integer
        format: int32
      units:
        type: array
        items:
          $ref: '#/definitions/TemporalUnit'
      negative:
        type: boolean
      zero:
        type: boolean
  EventCondition:
    type: object
  EventSource:
    type: object
    properties:
      type:
        type: string
      source:
        type: string
      api:
        type: string
  EventType:
    type: object
    properties:
      name:
        type: string
      input:
        $ref: '#/definitions/EventSource'
      eventsPattern:
        $ref: '#/definitions/EventsPatternExpression'
  EventsPatternExpression:
    type: object
    properties:
      name:
        type: string
      condition:
        $ref: '#/definitions/EventCondition'
      windowTime:
        $ref: '#/definitions/Duration'
      minTime:
        type: integer
        format: int32
      maxTime:
        type: integer
        format: int32
      optional:
        type: boolean
      greedy:
        type: boolean
      next:
        $ref: '#/definitions/EventsPatternExpression'
      contiguityCondition:
        type: string
      untilCondition:
        $ref: '#/definitions/EventCondition'
  Query:
    type: object
    required:
      - listeners
    properties:
      id:
        type: string
        xml:
          attribute: true
      events:
        type: array
        items:
          $ref: '#/definitions/EventType'
      listeners:
        type: array
        items:
          type: string
      validityPeriod:
        $ref: '#/definitions/ValidityPeriod'
      qos:
        type: object
        additionalProperties:
          type: string
```

FIGURE 3-34 MONITORING API

There are described three endpoints, and the structure of the request body. The reach description of the Monitoring functionalities is provided in deliverable D4.9 on Monitoring, prediction and diagnosis mechanisms.

During this cycle, the graphic user interface for registering query, list of active queries, and list of high-level events have been added to GUI. The next sections contain the description of the above mention integrations.

3.2.2.1. Monitoring query registration with GUI

To GUI frontend has been added functionality of adding the monitoring query. In Figure 3-35 QUERY REGISTRATION VIEWis shown the view for adding a new monitoring query.

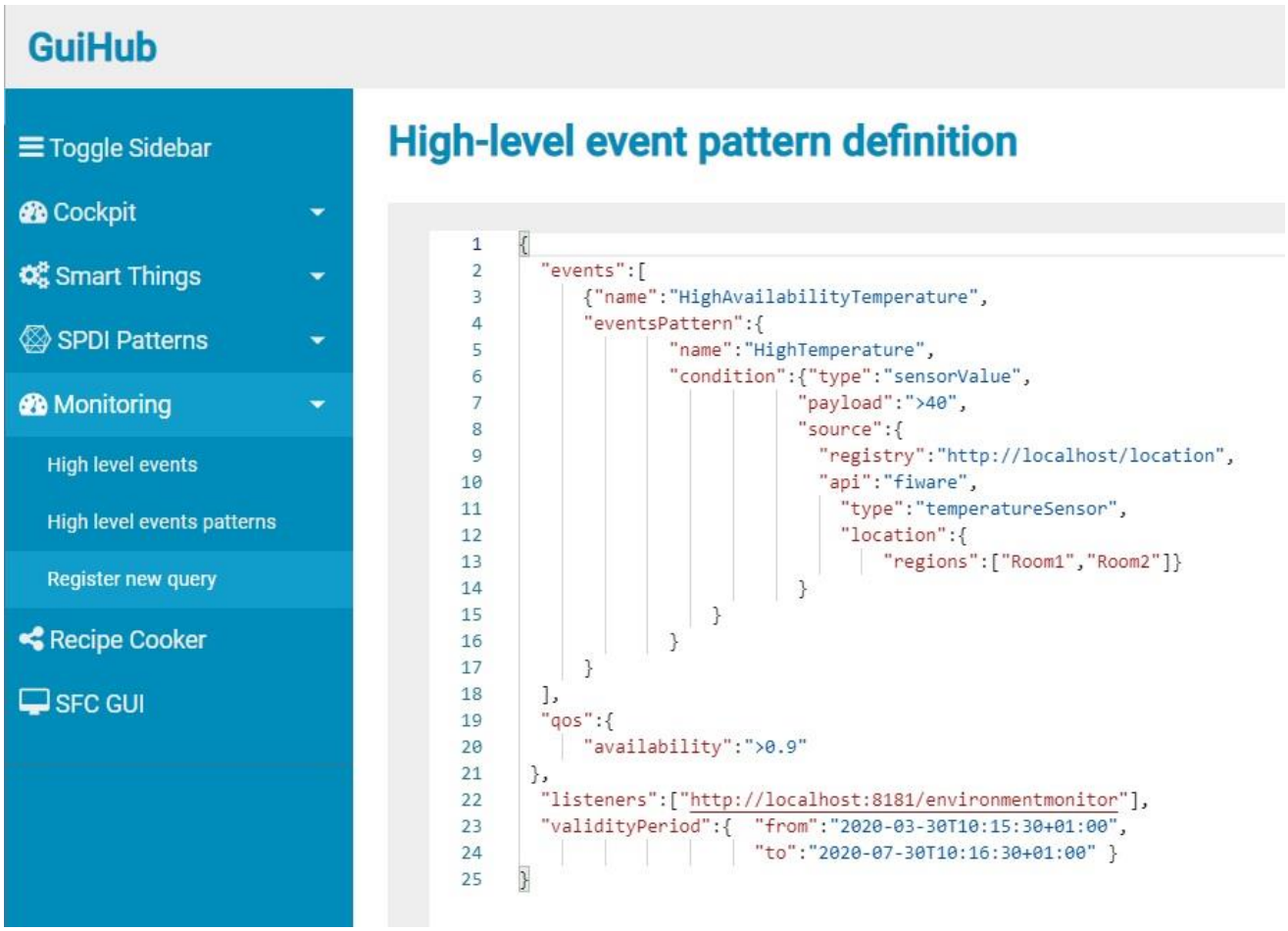


FIGURE 3-35 QUERY REGISTRATION VIEW

The query is sent to the Monitoring Component and if is valid, then is store in GUI Backend DataBase. The sequence diagram (Figure 3-36) shows the registration of a new query. User put the body of a query into from, click the register button, GUI sends the query to the Monitoring if the query is valid GUI save it in its database and display success notification otherwise unsuccessful message.

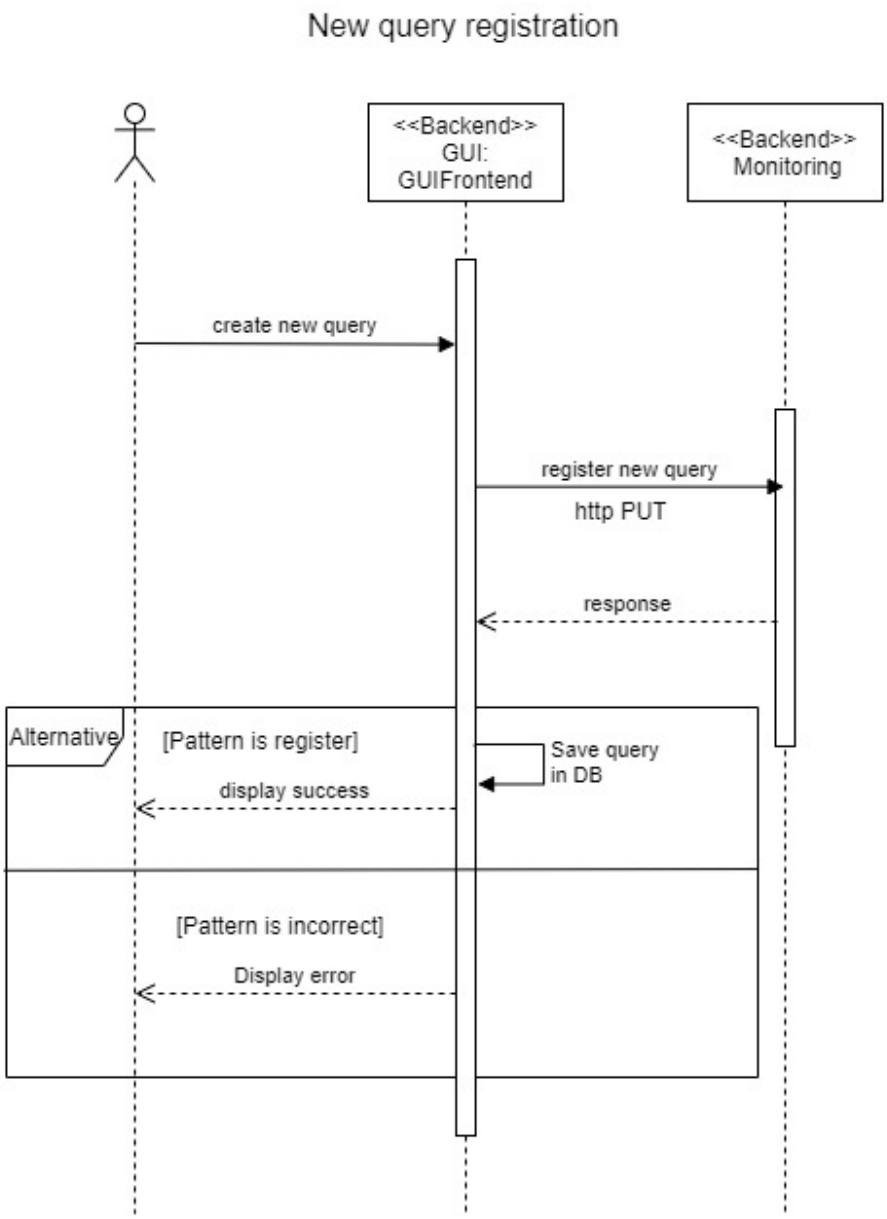


FIGURE 3-36 NEW QUERY REGISTRATION

On Figure 3-37 is shown the body of registration a new query Http request, the id is automatically added by the GUIBackend, the rest of the body is rewritten from the user input form presented in Figure 3-35.

```
1  {   "id":"Q2100",
2      "events":[
3      {"name":"FallEvent",
4          "input":{"fiware":"http://localhost:9090/smartphone"},
5          "eventsPattern":{
6          "name":"SingleEventPattern",
7          "condition":{"type":"IMUEvent"}
8                  }
9          }
10     ],
11     "listeners":["http://localhost:8181/sara/falldetector",
12                  "http://localhost:8080/semiotics/api/mdp/storage/observations"],
13     "validityPeriod":{"from":"2020-03-31T10:15:30+01:00", "to":"2020-08-31T10:16:30+01:00" }
14 }
```

FIGURE 3-37 SIMPLE QUERY HTTP REQUEST BODY

3.2.2.2. View of active monitoring queries

During Cycle 2 the view of high-level events has been added. A new view is shown in Figure 3-38. The table contains six columns which four of them can sort by clicking on the double arrow icon. Also, the search box allows to find the high-level event pattern by its name which is Figure 3-37 on line 4. The green button “Add new Pattern” leads to the registration new query view which is shown in Figure 3-35.

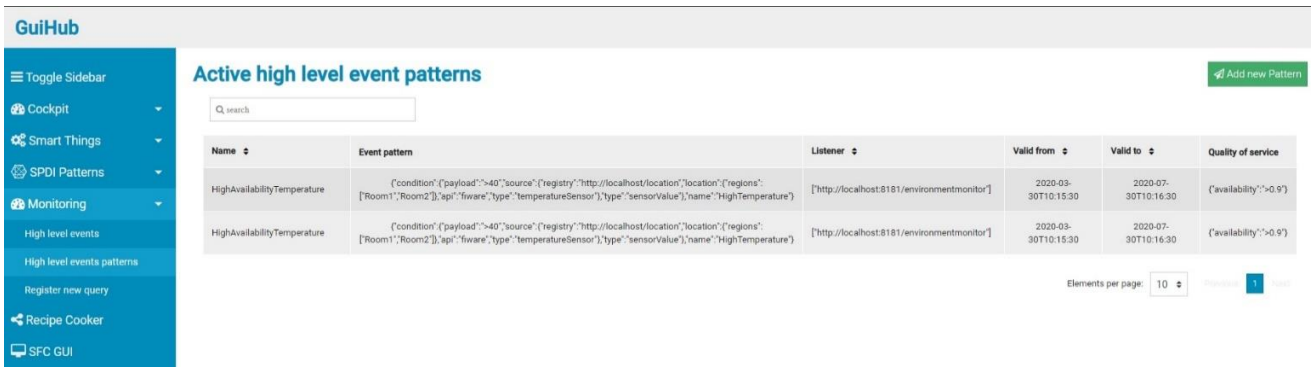


FIGURE 3-38 VIEW OF ACTIVE QUERIES

3.2.2.3. View of high-level events

The new view of the list of high-level events has been added and is shown in Figure 3-39. The table contains five columns in which four of them can be sorted by clicking on the double arrow icon. The search boxes above columns allows to search the high-level event by source, id, or event pattern name. The blue eye icon provides access to the details of base events.

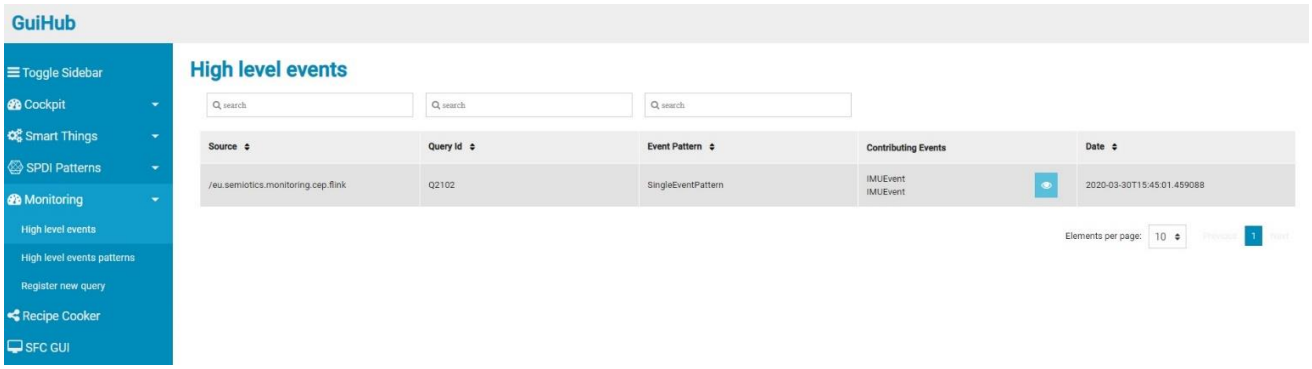


FIGURE 3-39 VIEW OF HIGH-LEVEL EVENTS

In the GUI Backend has been also added the endpoint for receiving the high-level events from Monitoring Component. In Figure 3-40 is shown the sequence diagram of high-level propagation.

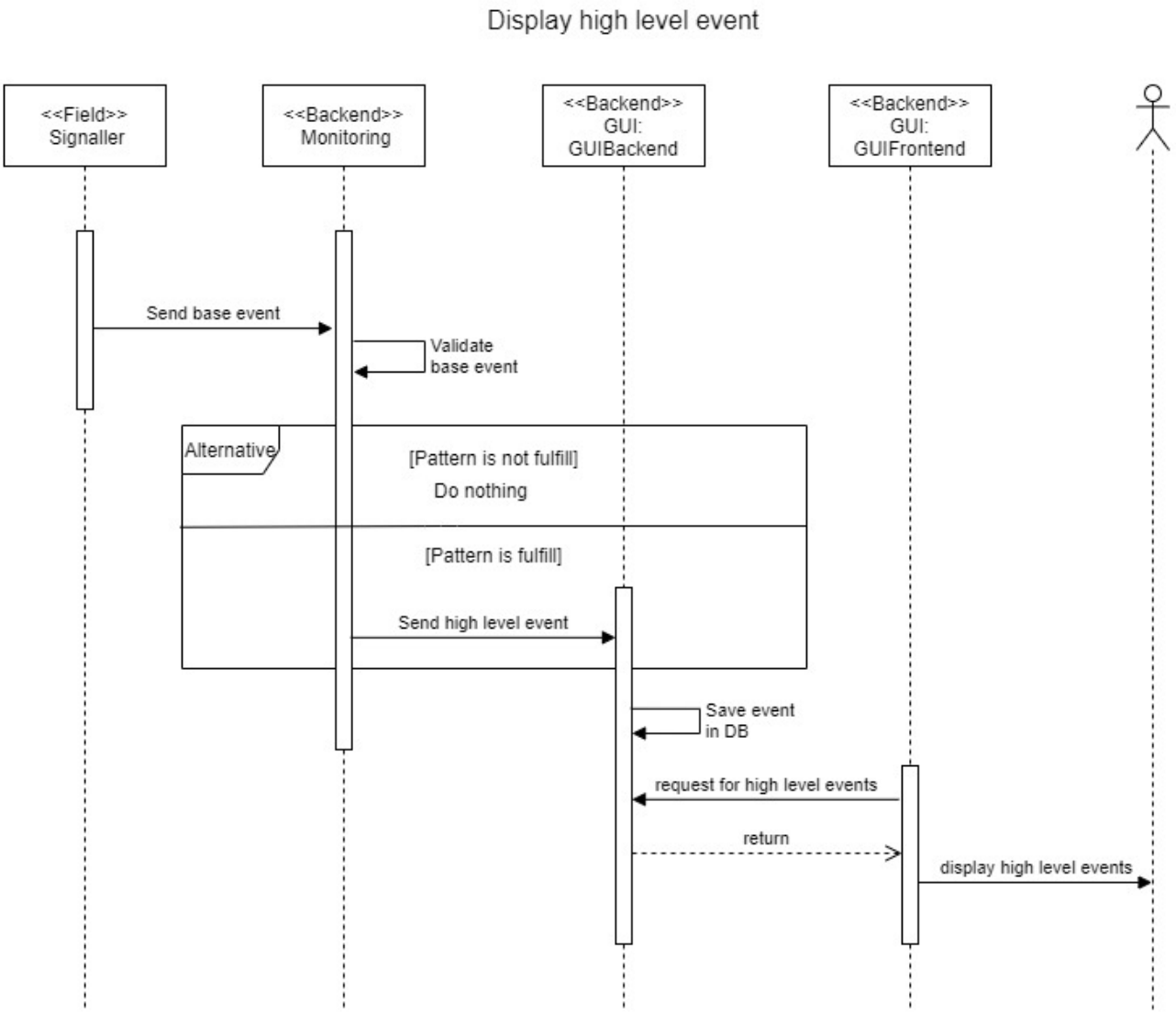


FIGURE 3-40 SEQUENCE DIAGRAM OF HIGH-LEVEL PROPAGATION

The low-level event is also called the base event, the structure of it and high-level event are shown in Figure 3-41. In each high-level event is also a list of base events that triggered it.

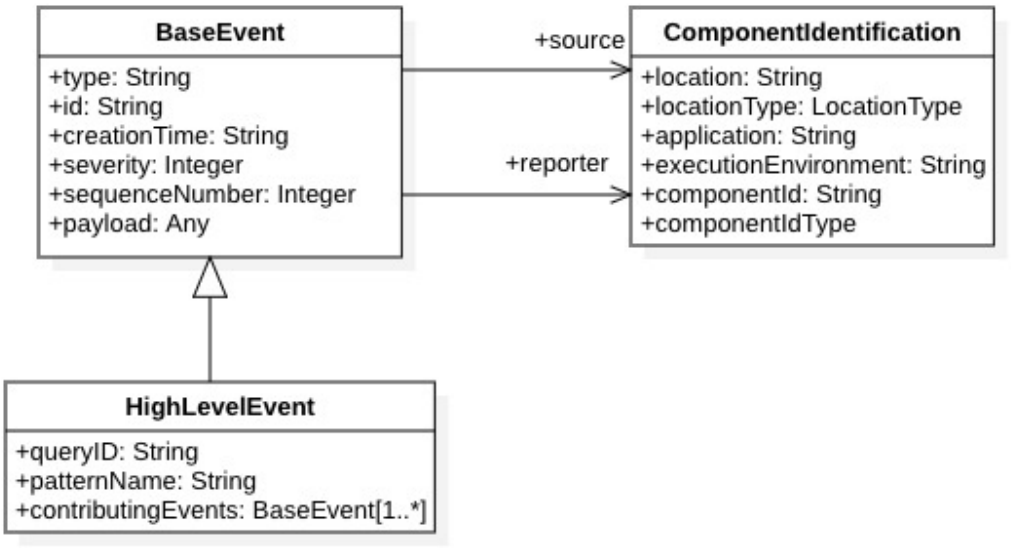


FIGURE 3-41 SEMIOTICS EVENT OBJECT MODEL

3.2.3. BACKEND PATTERN ENGINE INTEGRATION WITH SECURITY MANAGER

Regarding the integration of Backend Pattern Engine (BPE) with Security Manager (SM), it has been focused on BPE to be able to get information from the SM associated with the location of the patient. To that end, the communication between the two components has been established without having to house the two components in the same premises. Case in point, the SM is located in the premises of University of Passau and the BPE is located in FORTH's premises.

The flow of information is one way, meaning that the BPE requests information from the SM and not the other way around. For that reason only the SM had to be publicly available and that was provided under the "semiotics-security.sec.uni-passau.de" domain. From the BPE side, the location of the SM is retrieved from the Environment Variables of the hosting system. Even if the Environment Variables are not set, a fall back location of the SM is embedded in the code of the BPE (Figure 3-42).

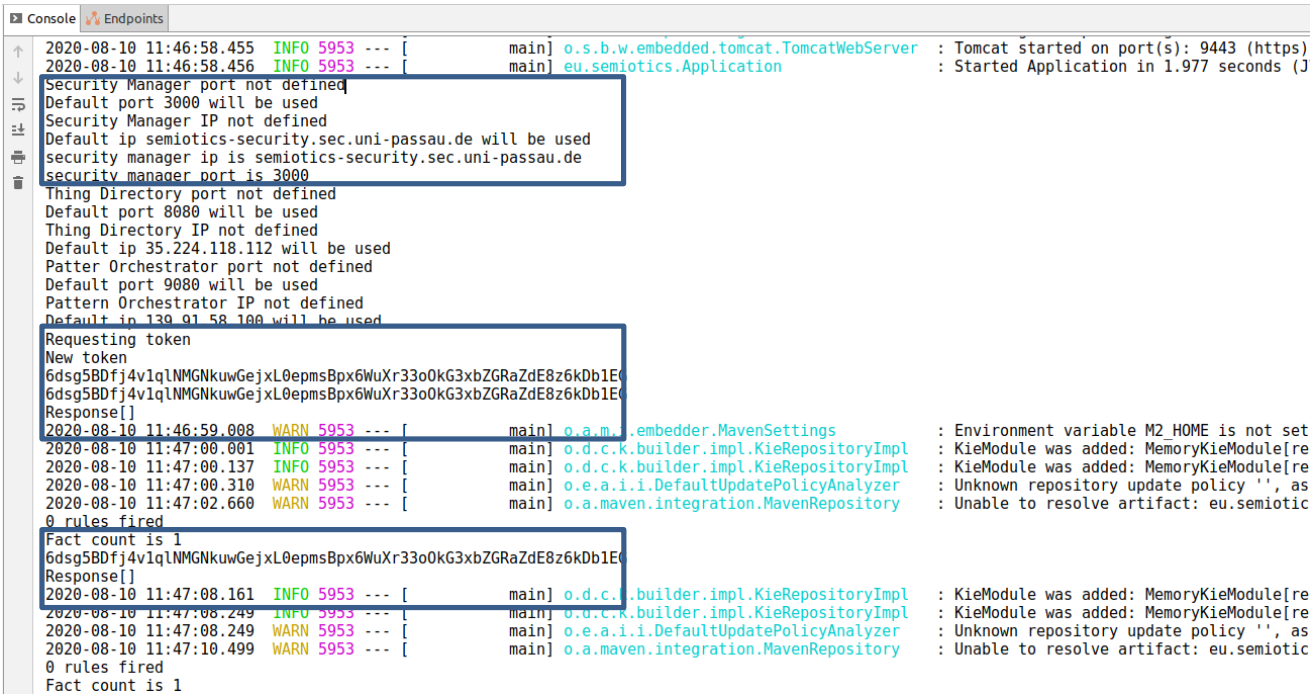


FIGURE 3-42 SECURITY MANAGER IP, PORT AND TOKEN

The BPE requests periodically the list of users that have access to the location of the patient. Under normal circumstances this list is meant to be empty, indicating that no one has access to the location of the patient, hence the privacy of the patient is ensured. Upon receiving the response from the SM, a property indicating the privacy of the patient, is inserted to the Drools Engine as a fact, and a corresponding verification rule is triggered, indicating that the privacy of the patient holds (Figure 3-43). On the contrary when the need arises for the location of the patient to be disclosed to other users, e.g. when the patient falls, then the said list in no longer empty but it includes the users who have access to the patient's location. As soon as the BPE realizes that the location list is no longer empty, the corresponding property for the privacy is updated and inserted to the Drools Engine which will result in failing to trigger the corresponding rule and consequently indicating that the privacy of the patient no longer holds.

In order for the above interaction to be completed, the BPE initially requests a Bearer token from the SM and the said token is used when the BPE requests the list of entities with access to the location of the patient from the SM. The token has limited lifetime, therefore when the token expires the SM responds with the error 401 Unauthorized. Whenever the BPE receives that aforementioned error, it requests a new Bearer token from the SM.

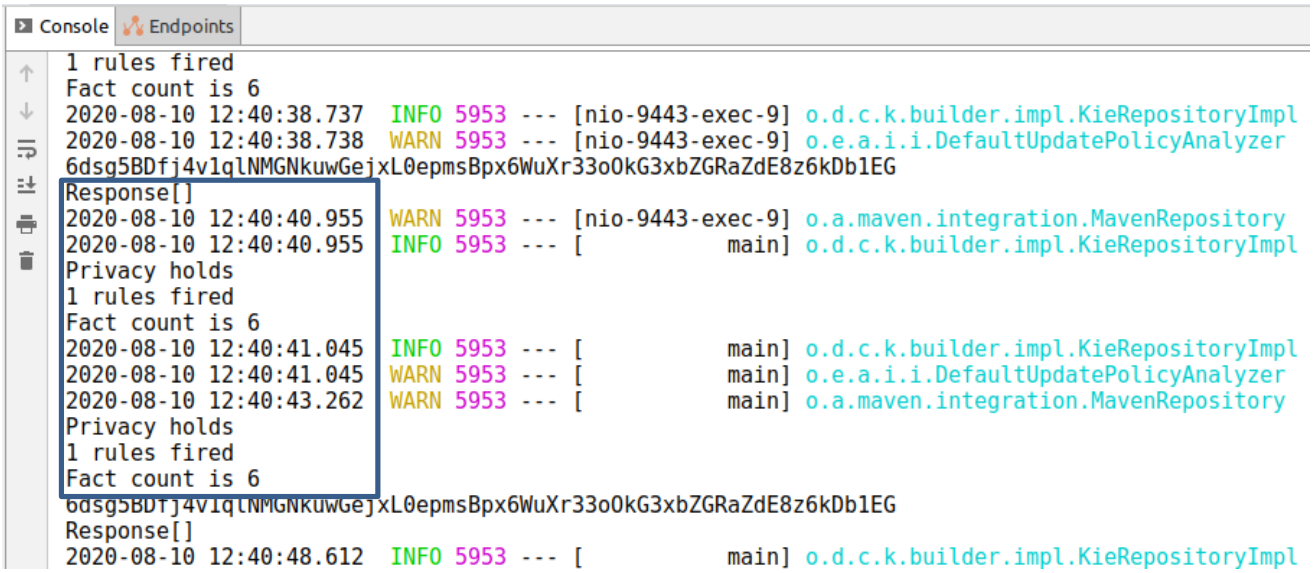


FIGURE 3-43 PRIVACY RULE IS TRIGGERED PERIODICALLY

3.2.4. MONITORING COMPONENT INTEGRATION WITH PATTERN ENGINE

As already mentioned in section 5.3 of D4.9 SEMIoTICS Monitoring, Prediction and Diagnosis Mechanisms (final) the monitoring component is integrated with the Pattern Engine at the SDN/NFV Layer using listeners for nodes and links. The said listeners interact directly with the Pattern Engine in order to provide notifications regarding the topology changes.

The network topology is composed of nodes and links between those nodes. The said nodes may be any network device such as a simple switch, a gateway or a host. Whenever a link failure occurs, the PE at the SDN layer must be informed of this, in the form of Drools Fact in order to reason whether any active SPDI or QoS property has been violated.

The data in SDN controller is tree-based represented and there are interfaces that can monitor/listen changes in the tree. Two data-tree change listeners are implemented for monitoring the node and link changes as identified in the SDN network topology. The interface methods are invoked every time when there is a data change event in the specific path of the tree. The two modification types that we deal with, are “delete” (Figure 3-44) and “write” (Figure 3-45) as defined by the implementation of the said interface in SDN controller. The “delete” represents the removal of a link or a node and the “write” represents the addition of a link or a node accordingly.

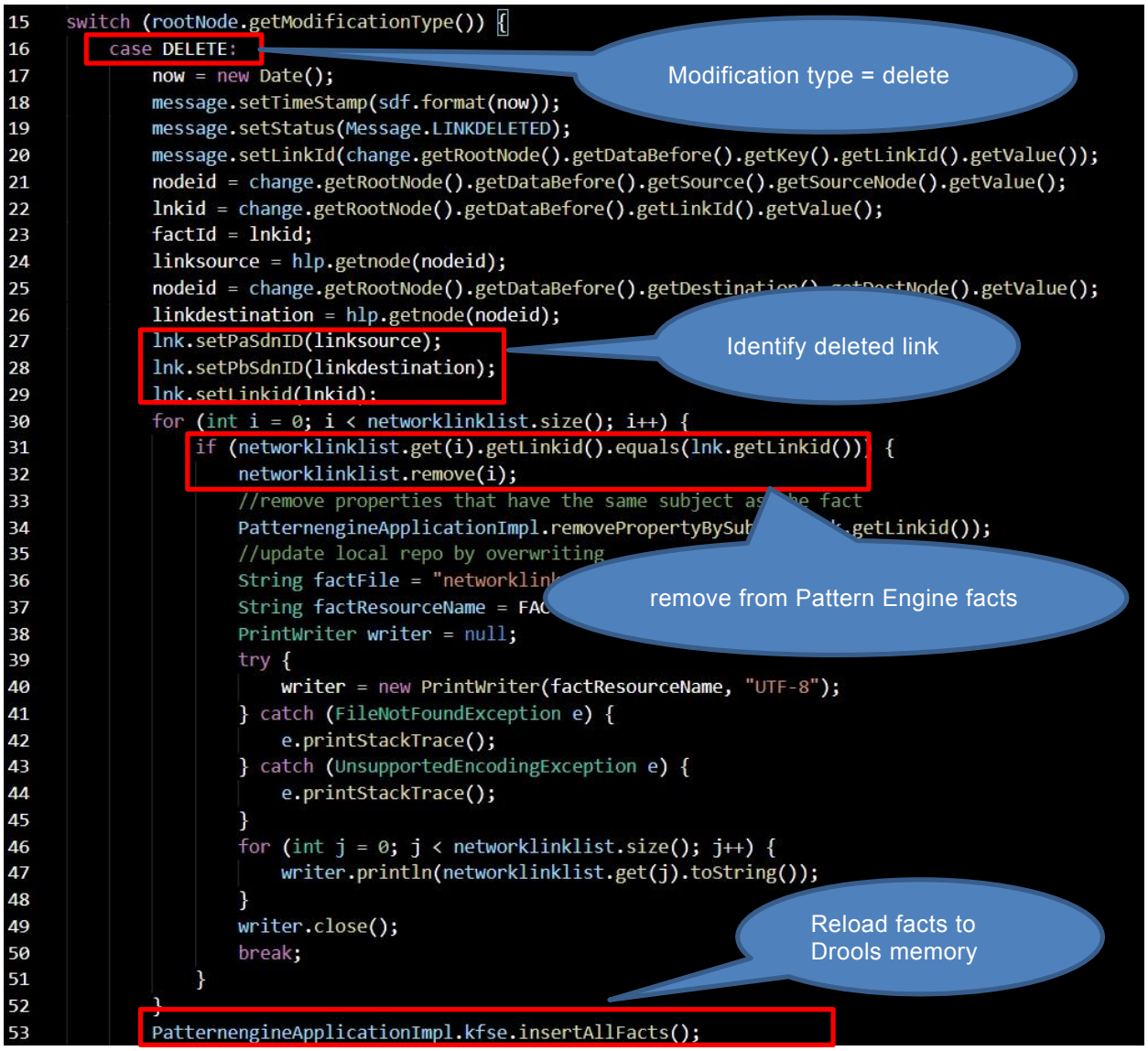
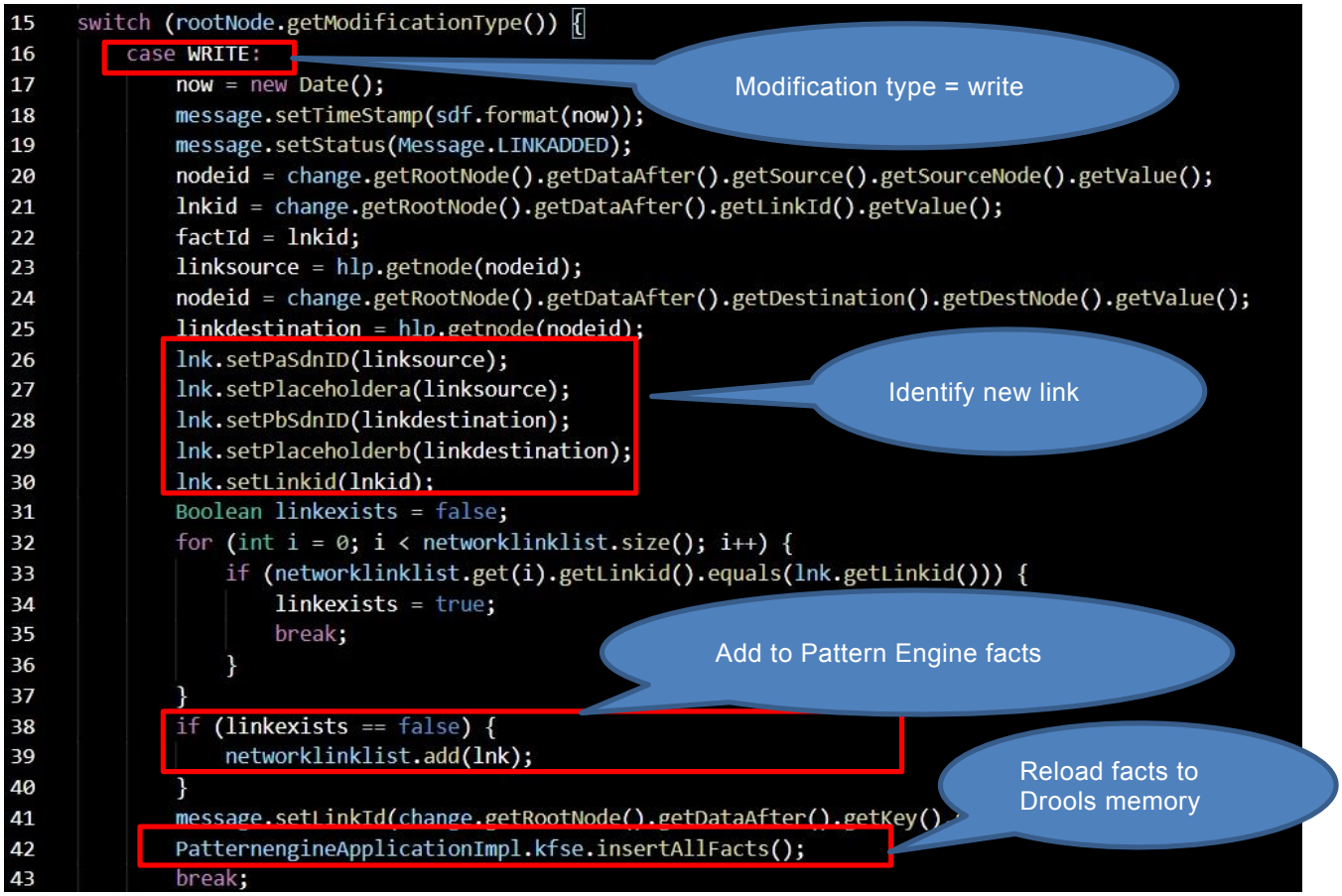


FIGURE 3-44 MODIFICATION TYPE "DELETE"




```
15  switch (rootNode.getModificationType()) {
16      case WRITE:
17          now = new Date();
18          message.setTimeStamp(sdf.format(now));
19          message.setStatus(Message.LINKADDED);
20          nodeid = change.getRootNode().getDataAfter().getSource().getSourceNode().getValue();
21          lnkid = change.getRootNode().getDataAfter().getLinkId().getValue();
22          factId = lnkid;
23          linksource = hlp.getnode(nodeid);
24          nodeid = change.getRootNode().getDataAfter().getDestination().getDestNode().getValue();
25          linkdestination = hlp.getnode(nodeid);
26          lnk.setPaSdnID(linksource);
27          lnk.setPlaceholdera(linksource);
28          lnk.setPbSdnID(linkdestination);
29          lnk.setPlaceholderb(linkdestination);
30          lnk.setLinkid(lnkid);
31          Boolean linkexists = false;
32          for (int i = 0; i < networklinklist.size(); i++) {
33              if (networklinklist.get(i).getLinkid().equals(lnk.getLinkid())) {
34                  linkexists = true;
35                  break;
36              }
37          }
38          if (linkexists == false) {
39              networklinklist.add(lnk);
40          }
41          message.setLinkId(change.getRootNode().getDataAfter().getKey()
42          PatternengineApplicationImpl.kfse.insertAllFacts();
43          break;
```

FIGURE 3-45 MODIFICATION TYPE "WRITE"

3.2.5. RECIPE COOKER INTEGRATION WITH THING DIRECTORY

From the Recipe Cooker, the Thing Directory can be called and incorporated in the definition of recipes (or: application flows). To integrate these two components, we have developed the TD Search node. It is capable of sending search requests to the Thing Directory and receives as a response all Thing Descriptions (TD) that match the search request. Figure 3-46 shows a simple example of an application flow that uses the TD Search node to discover suitable devices. The search is triggered by the 'inject' node (in blue) and the received TD is printed out using the 'debug' node (in green). Similar to printing the TD out using the 'debug' node, it can also be handed over to other kinds of nodes for further processing. E.g., it can be handed over to a UI node that would represent the contents of the TD to a user.

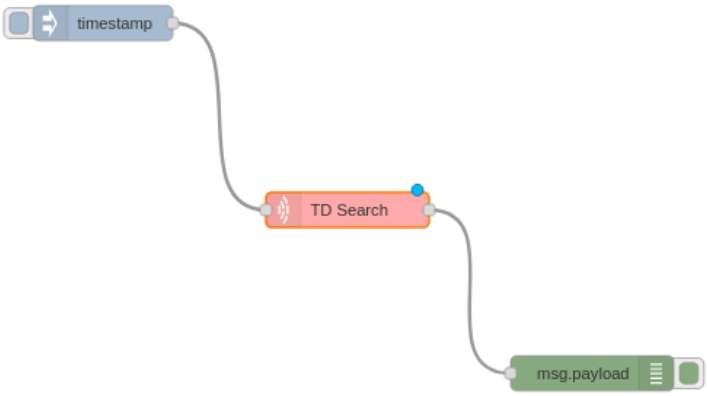


FIGURE 3-46 EXAMPLE APPLICATION FLOW USING THE TD SEARCH NODE.

In order to execute the search queries on the Thing Directory, the TD Search node needs to be configured correctly. This can be done by double clicking the node after which the configuration panel (Figure 3-47) is shown. In the configuration panel, the URL and port of the Thing Directory needs to be specified, the search context needs to be defined (this is the namespace of the TD context to be searched for), and the search type needs to be given, which is mapped to the 'type' field of the properties of all TDs registered in the Thing Directory.

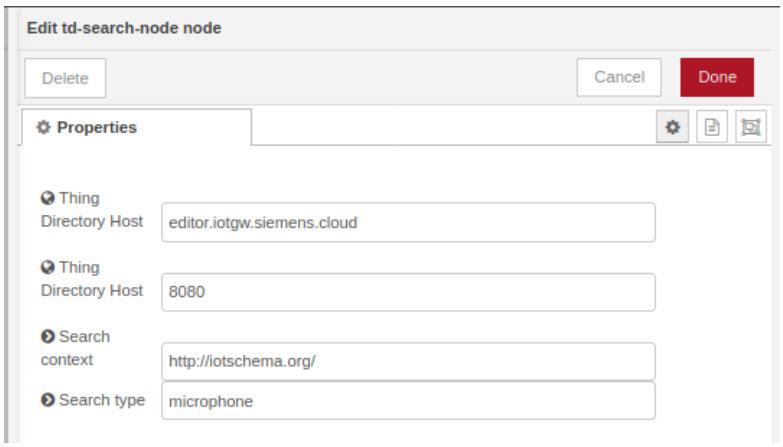


FIGURE 3-47 CONFIGURATION OF THE TD SEARCH NODE.

The search / discovery request sent out to the Thing Directory formulated based on the configuration of the TD Search node is a JSON message. An exemplary request, formulated out of the configuration shown in Figure 3-47 is presented below:

```
{
   "@context": ["http://www.w3.org/ns/td",
               {"iot": "http://iotschema.org/"}],

   "@type": "iot:microphone"
}
```

3.2.6. BACKEND SEMANTIC VALIDATOR INTEGRATION

As was mentioned in sub-section 3.1.6 and in D4.11 “Semantic interoperability mechanisms for IoT (Final)”, BSV provides validation mechanisms to ensure semantic interoperability in SEMIoTICS, interoperability between targeted external IoT platforms and SEMIoTICS (see sub-section 4.1) and semantic adaptation through both the interacting with the Pattern Engine (Backend layer) and using the Adaptor Nodes in Recipe Cooker. The last, was finalized during the cycle 2. Specifically, to enable the interoperability between the flow’s Things, a number of different steps are required, following the corresponding sequence diagram in Figure 3-49. In fact, the Recipe Cooker component, which is responsible for cooking (creating) recipes reflecting user requirements, sends the recipe in Pattern Orchestrator component, which is in charge of the automated configuration, coordination, and management of different patterns and their deployment to express requirements of the flows to guarantee interoperability based on architectural patterns; this step includes the insertion of the interoperability requirement from the Orchestrator to the Pattern Engine to enforce the respective pattern rules (see Figure 3-48). The pattern is expressed in a machine-processable Drool rule format of the said semantic interoperability for any inserted flow. The when part identify the requested placeholders placed in sequence, required to satisfy the semantic interoperability property. If the conditions are met, the rule in then can guarantee that the requested property is satisfied.

```
rule "Sequence Semantic Interoperability Verification"
when
      Placeholder($pA:=placeholderid)
      Property ($pA:=subject, category=="semantic", $prvaluein1:=input_value,
        $prvalueout1:=output_value, satisfied==true)
      Placeholder($pB:=placeholderid)
      Property ($pB:=subject, category=="semantic", $prvaluein2:=input_value,
        $prvalueout2:=out_value, satisfied==true)
      Sequence($sId:=placeholderid, $pA:=placeholdera, $pB:=placeholderb)
        $PR: Property ($sId:=subject, category=="semantic", $prvalueout1==$prvaluein2,
      satisfied==false)
then
      modify($PR){satisfied=true, input_value=$prvaluein1, output_value=$prvalueout2};
end
```

FIGURE 3-48 SEMANTIC INTEROPERABILITY VERIFICATION DROOL RULE

Therefore, this rule used by Pattern Engine to trigger the BSV, which resolves semantic interoperability issues, between any link of Things in the flow recipe. Particularly, the BSV receives a request with the flow id from and the Things’ id for each link. Based on this information, the component begins the procedure to tackle the semantic interoperability issues between these two things from the said flow. For that reason, it sends SPARQL query to Thing Directory to receive the Thing Description of the Things. In the sequel, the final phase of the interoperability adaptation is the following. It involves the harmonization of the semantic model capabilities with the registration of extra Adaptor Nodes in the Recipe Cooker if required.
Namely, there are three possible results, which are highlighted in Figure 3-49. Firstly, the link source and destination are interoperable, so the BSV updates the Pattern Engine 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, the BSV not only sends the TRUE response to pattern engine, but it also updates the flow in the Recipe Cooker. Lastly, when the link source and destination are not interoperable and the BSV does not have the required information to develop the Adaptor Nodes, it sends FALSE response to the Pattern Engine.

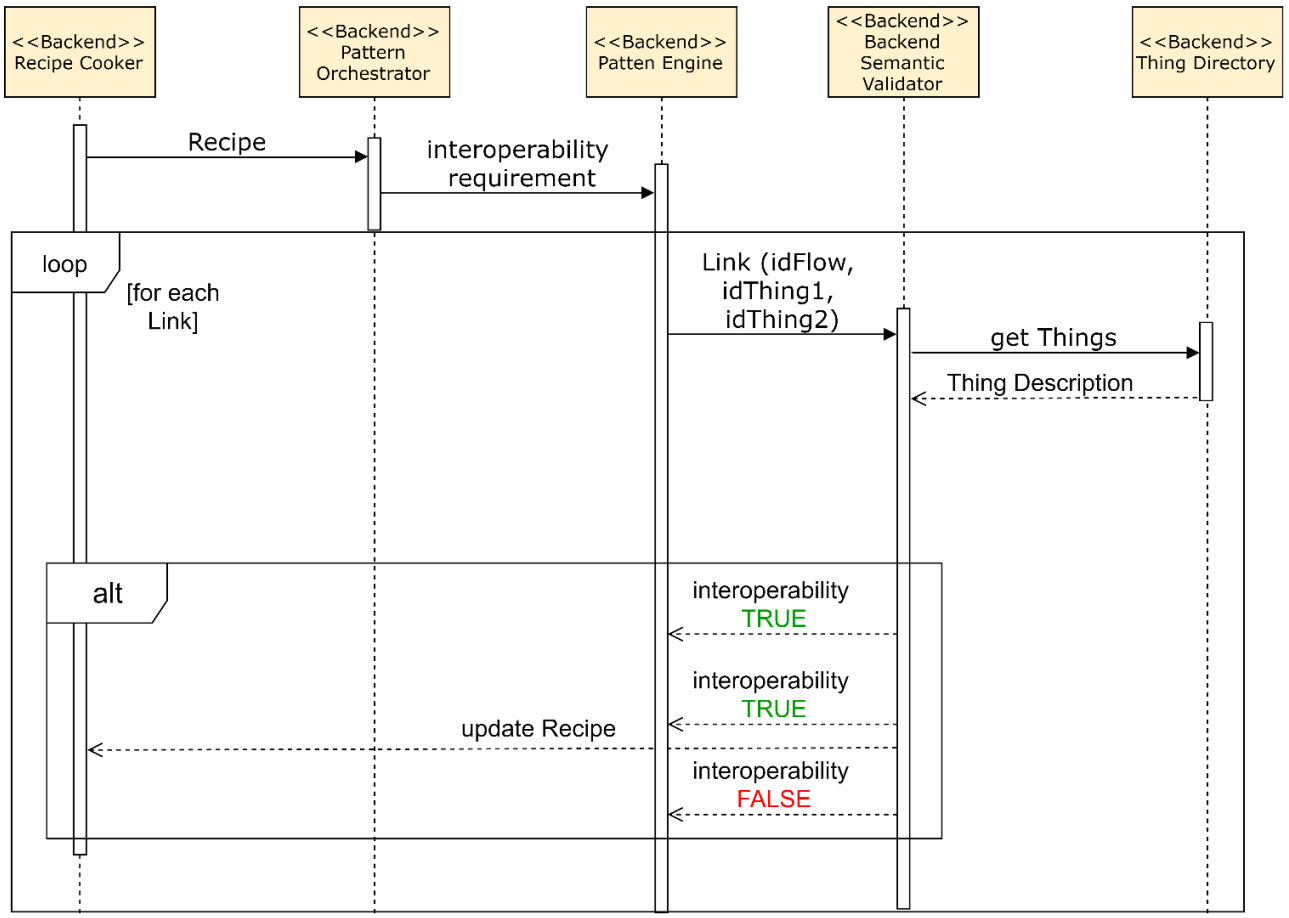


FIGURE 3-49 SEQUENCE DIAGRAM FOR SEMANTIC INTEROPERABILITY ADAPTATION MECHANISMS

The actual evaluation of said integration, the workflow status before and after the semantic validation, is illustrated in Figure 3-50. The Adaptor Nodes are introduced at runtime with the corresponding functionality. In a large-scale scenario, the semantic interoperability is examined for any link/wire between the components of the flow and a combination of different Adaptor Nodes can be used parallel to ensure the interoperability of the system.

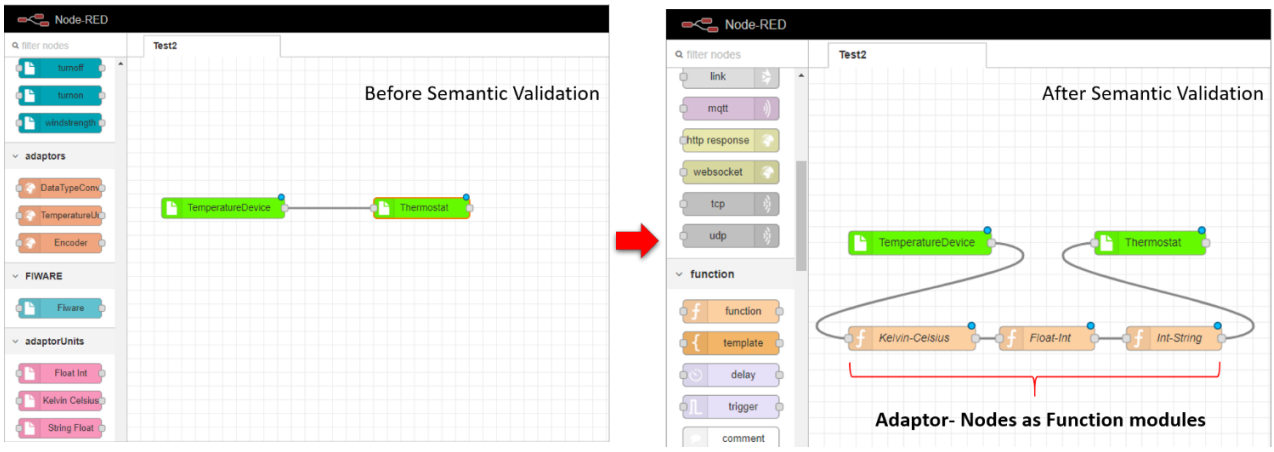


FIGURE 3-50 SEMANTIC VALIDATION/ADAPTATION MECHANISMS

Based on the above, the previous list of the components that interact with BSV component, from cycle 1 (see Table 4), can be updated as following:

TABLE 5 UPDATED LIST OF COMPONENTS THAT INTERACT WITH THE BSV COMPONENT (CYCLE 2)

Component	Components that will be used/consumed by this component	Layer of component that will be consumed	Description of interactions
Backend Semantic Validator	*Thing Directory*	*Backend*	Searching for the necessary Thing models in Thing Directory component, in order to detect any potential semantic conflicts between the interacting domains
	Recipe Cooker	*Backend*	Connecting with Recipe Cooker to resolve these semantic conflicts using the Adaptor Nodes that configure an Interaction Pattern in accordance with the application's requirements.
	Semantic API & Protocol Binding	*Field*	This interaction was not implemented, because it was out of the requirements of the final implementation of the SEMIoTICS UCs scenarios (see D4.11)
	FIWARE Broker & GEs	*Backend*	Searching for the necessary Thing models in external IoT platforms that use other semantic schemas different from iot.schema (e.g. schema.lab.fiware.org)
	Pattern Engine	*Backend*	Pattern Engine runs the pattern rule regards interoperability property and triggers the BSV

4. INTEROPERABILITY WITH EXTERNAL IOT PLATFORMS

No new integrations with external IoT platforms have been delivered as part of Cycle 2 of task T5.2. Nevertheless the general approach and the existing integrations are presented in the subsections that follow.

4.1. General Approach

This section presents the general approach and the interaction of SEMIoTICS components in order to enable the interoperability between targeted external IoT platforms (i.e., FIWARE, AREAS, and MindSphere) with SEMIoTICS framework. The following motivating example with FIWARE is used for the description and analysis of the development of the proposed approach.
The components of the SEMIoTICS architecture (see Figure 4-1) 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.

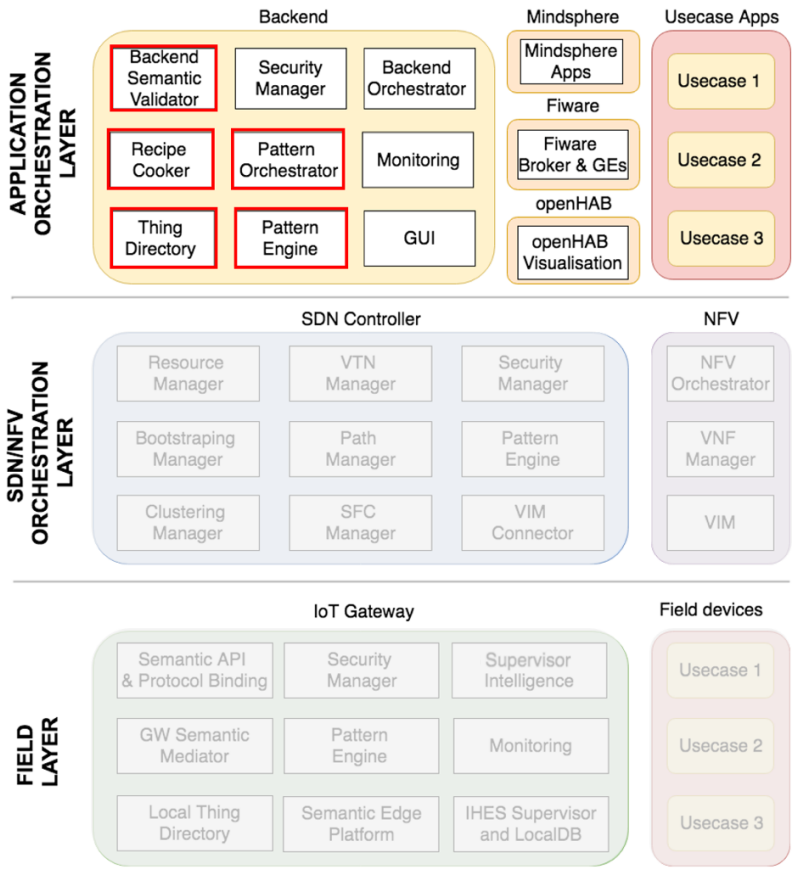


FIGURE 4-1 SEMIOTICS ARCHITECTURE – INTEROPERABILITY WITH EXTERNAL IOT PLATFORMS COMPONENTS

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 (Figure 4-2). The main 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 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 every link, Pattern Engine (Backend) triggers the BSV.

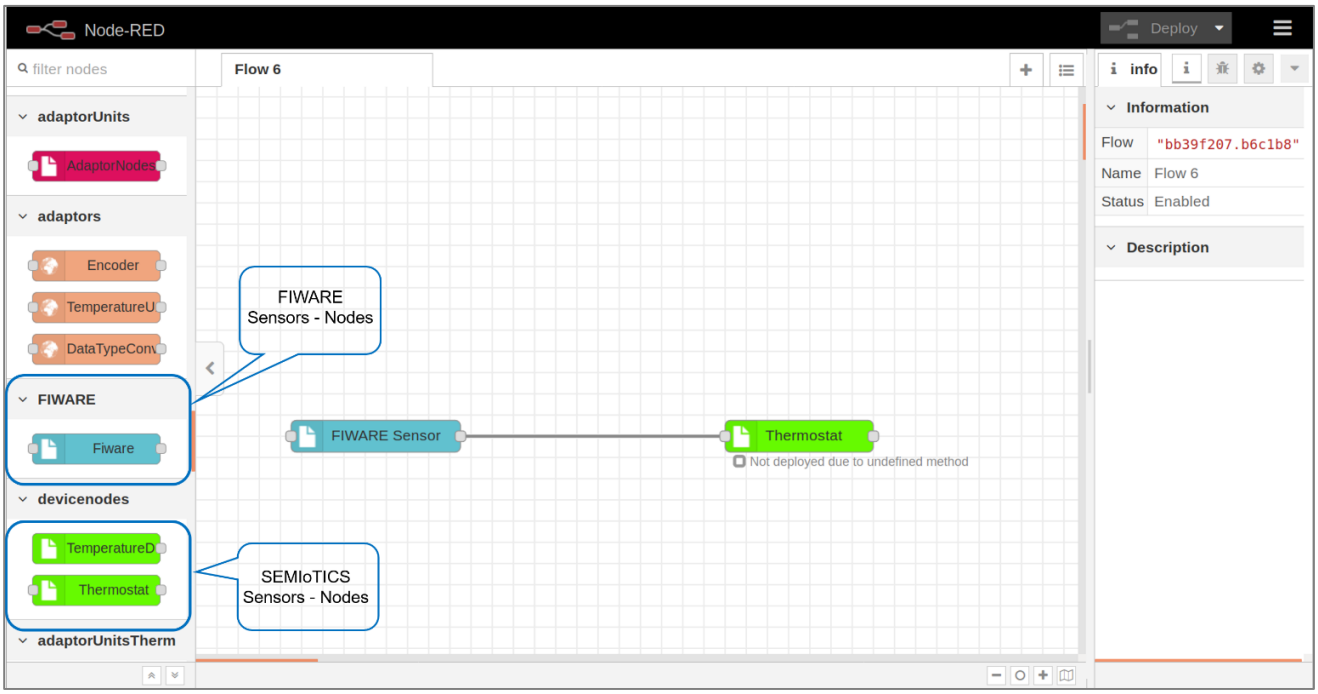


FIGURE 4-2 RECIPE INTERACTION EXAMPLE FIWARE – SEMIOTICS BEFORE SEMANTIC VALIDATION

Following this, the BSV begins the procedure to tackle the semantic interoperability issues between these two Things. Firstly, the semantic description for each Thing is required, for that reason, it sends two requests:

- *getThings* request to Thing Directory in order to receive the Thing Description of SEMIoTICS Thermostat and
- *getElements* request to the FIWARE platform to receive the Element Description of FIWARE Sensor.

Based on this information, the BSV is able to 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. First: the link source and destination are interoperable, so the BSV replies to the Pattern Engine (Backend) with the TRUE response. Second: the link source and destination are not interoperable and the BSV can add Adaptor Nodes in order to guarantee 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 (Figure 4-3). Third: 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.

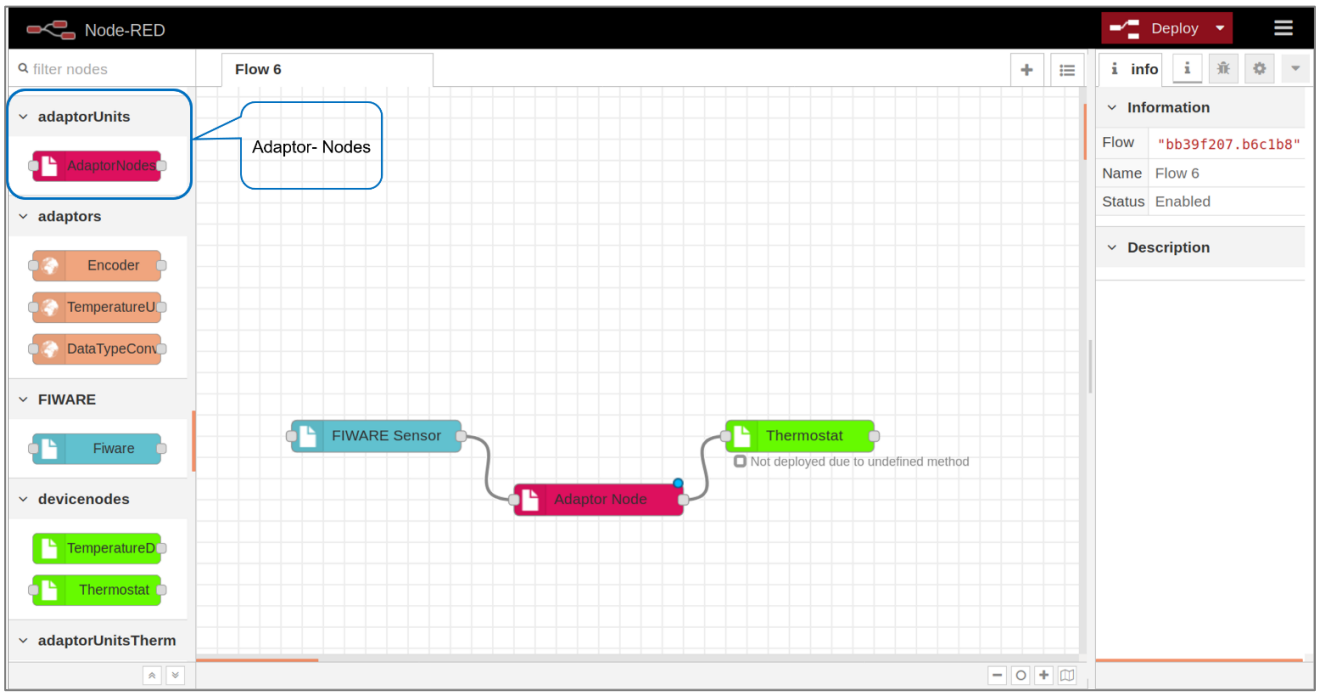


FIGURE 4-3 RECIPE INTERACTION EXAMPLE FIWARE – SEMIOTICS AFTER SEMANTIC VALIDATION

The above approach of the semantic interoperability mechanisms between SEMIoTICS external IoT platforms is highlighted by a sequence diagram, in Figure 4-4. It should be clarified that the term 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 link and other network components among them.

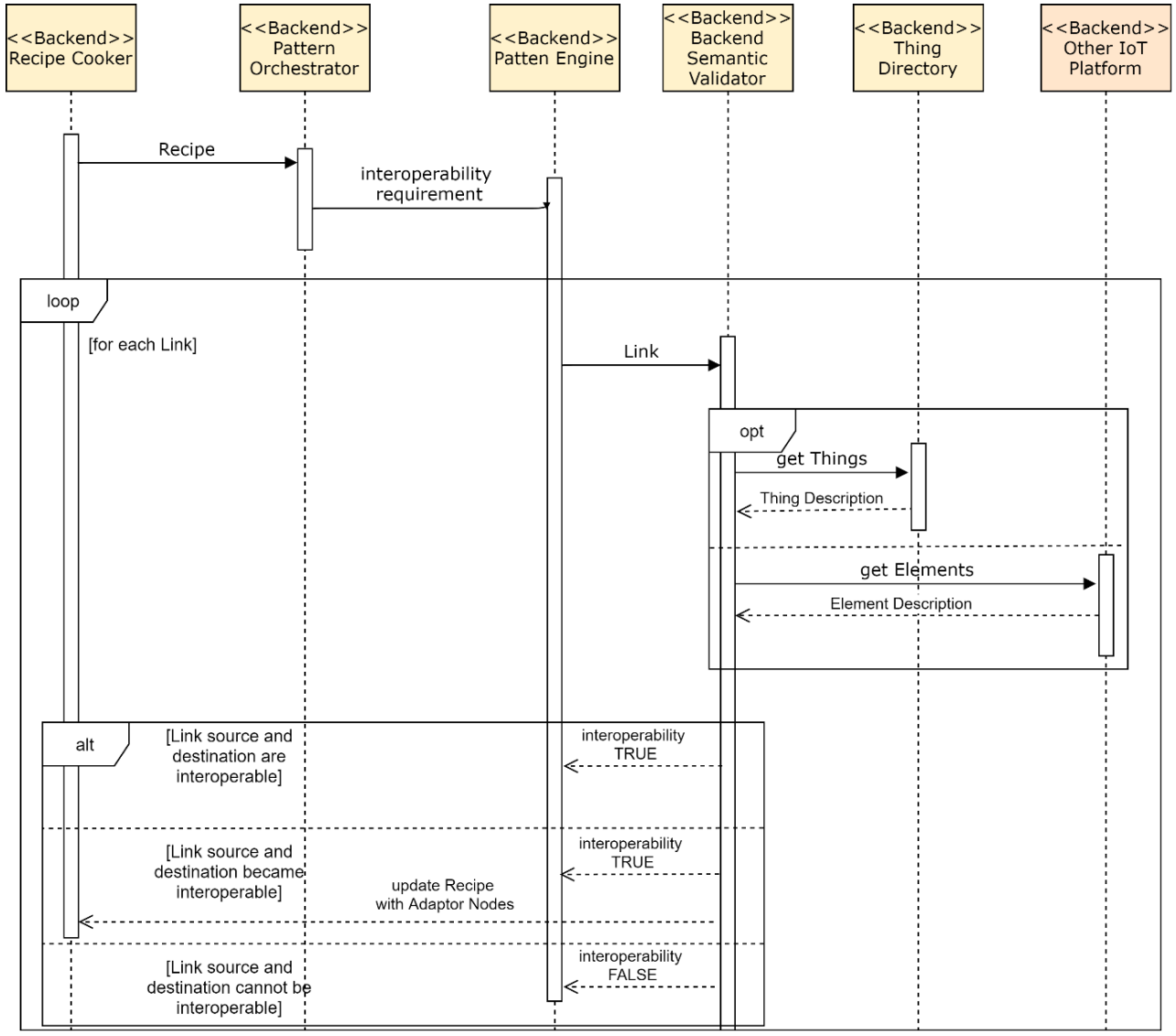


FIGURE 4-4 SEQUENCE DIAGRAM OF INTEROPERABILITY WITH EXTERNAL IOT PLATFORMS

4.2. Integration with FIWARE

4.2.1. METHODOLOGY OF FIWARE COMPONENT VERIFICATION

The verification process has been divided into three steps. As a first step, the GEs which are not related to SEMIoTICS framework were eliminated as not useful. Moreover, since SEMIoTICS project's ambition is to deliver the solution with high impact and possibilities of further exploitation, it was decided that FIWARE components that are deprecated or no longer supported, will not be used in the project. Finally, components which would not compile properly, without errors would not be used either. As a result, the final choice of 9 General Enablers was deeply investigated for possible use in SEMIoTICS framework:

- PEP Proxy – Wilma
- Authorization PDP – Authzforce
- Identity Management – Keyrock
- Publish/Subscribe Context Broker – Orion Context Broker

- IoT Agent
- IoT Discovery
- NetIC
- Data Visualization – Knowage
- Object Storage

The analysis was taking into consideration technology used for development, what interface is offered by GE, what are specific implementation requirements and how such GE may potentially affect other components of SEMIoTICS framework.

4.2.2. EVALUATION PROCESS WITH SELECTED GENERAL ENABLERS

In this section, the results of the investigation are presented. According to functionalities or used technology, GEs were grouped into four categories. The first group consists of security-related components (PEP Proxy Wilma, Identity Management – Keyrock, and Policy Manager AuthzForce). In the second group, there are components using NGSI data format (IoT Agent, IoT Discovery, Context Broker). The third group contains only one FIWARE component which is related to SDN and NFV – NetIC. In the fourth category, General Enablers which are related to Database (Data Visualization – Knowage and Data management system Object Storage) have been included.

4.2.3. GROUP 1: SECURITY-RELATED GES

In this group are PEP Proxy Wilma, Identity Management – Keyrock and Policy Manager AuthzForce.

4.2.3.1. Functionality summary

PEP Proxy WILMA

Privacy in FIWARE can be assured through the usage of the PEP Proxy WILMA. In order to provide fully functional security and privacy component, it needs to be combined with other security components such as Keyrock and AuthzForce. WILMA ensures that only permitted users will be able to access the Generic Enablers or REST services. As WILMA is a backend component with no frontend interface, one must use the Identity Management GE web interface for user and application management and roles or permissions configurations. For a request validation, PEP Proxy interacts with the Identity Management and Authorization PDP GE by verifying appropriate parameters depending on the defined security level[8].

Identity manager Keyrock

Using Keyrock in a conjunction with other security components such as PEP Proxy and Authzforce allows adding OAuth2-based authentication and authorization security to services and applications.
One of the main functionalities of this Generic Enabler is to enable developers to add identity management (authentication and authorization) to their applications based on FIWARE identity. This is enabled by use of the OAuth2 protocol[9]. The FIWARE Keyrock Generic Enabler set up all common features of an identity management system so that other components are able to use standard authentication mechanisms to accept or reject requests based on industry-standard protocols[10].

AuthzForce

The Generic Enabler AuthzForce provides a multi-tenant RESTful API for policy administration points as well as for policy decision points. The API follows the REST architecture style and complies with XACML v3.0. This GE plays the role of a Policy Decision Point (PDP)[11]. AuthzForce helps to externalize the authorization logic and take advantage of flexible and standard-compliant Attribute-Based Access Control features. The main

[8] https://fiware-pep-proxy.readthedocs.io/en/latest/
[9] https://fiware-idm.readthedocs.io/en/latest/
[10] https://documenter.getpostman.com/view/513743/RWMLLRui?version=latest
[11] https://fimac.m-iti.org/6d.php

feature is the authorization policy decision evaluation. It evaluates authorization decisions based on XACML policies and attributes related to a given access request. The configuration of the XACML policies to be evaluated by the GE happens at the authorization policy administration point (PAP). The GE also provides some extensibility points e.g. for attribute providers aka PIPs (Policy Information Points). This makes it possible to plug custom attribute provides into the PDP engine to allow it to retrieve attributes from other attribute sources (e.g. remote service) than the input XACML Request during evaluation[12].

4.2.3.2. Feasibility study

PEP Proxy WILMA

WILMA is capable of providing access to GEs or REST services only for FIWARE users what is a significant limitation in the context of the SEMoTICS project where numerous types of users will need to be granted access.

Identity manager Keyrock

Keyrock GE is limited to the smooth cooperation only with the FIWARE GEs environment while Security Manager incorporated into SEMIoTICS architecture can handle all the functionalities offered by the Keyrock generic enabler and more. Security Manager brings OAuth2-based authentication directly out of the box. Another aspect that speaks for the Security Manager is that the Security Manager is also compatible with IoT devices which clearly fits better to the SEMIoTICS IoT concept. The entity storage module of the Security Manager currently supports LevelDB and MongoDB as storage providers for storing the entities. Due to the way it was designed, it is also very easy to extend it to other storage concepts.

AuthzForce

The PDP and PAP in the Security Manager of the SEMIoTICS architecture support also the same structure as introduced by AuthzForce. With the REST Entity API there is also a simple module to enforce proper policies. Moreover, the attribute-based encryption for the Security manager is currently under development within the project that will further increase the security aspect compared to the capabilities of AuthzForce.

4.2.3.3. Feasibility study outcomes

A combination of all of the abovementioned analysis outcomes brought the consortium to the decision that integration with GE from security group does not bring any value-added as the components involved in the architecture, namely Security Manager is a more flexible solution, is not limited to support only FIWARE components, provides wider capabilities and guarantees a higher level of security in the platform.

4.2.4. GROUP 2: NGSI-BASED COMPONENTS

IoT Agent, IoT Discovery and Orion Context Broker belong to this group because they require the NGSI-LD data model. They are responsible for communication, and information acquisition of IoT devices in FIWARE.

4.2.4.1. Functionality summary

Orion Context Broker

As it is mentioned in the official website[13] the Orion Context Broker is an implementation of the Publish/Subscribe Context Broker GE, providing the NGSI9 and NGSI10 interfaces. Using these interfaces, clients can perform several operations:

- register context producer applications, e.g. a temperature sensor within a room
- update context information, e.g. send updates of temperature

[12] https://authzforce-ce-fiware.readthedocs.io/en/latest/
[13] https://catalogue-server.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker

- get a notification when context information changes take place (e.g. the temperature has changed) or receive the value with a given frequency (e.g. to get the temperature value every minute)
- query context information. The Orion Context Broker stores context information updated from applications, so queries are resolved based on that information.

To work properly and store basic data, the Context Broker requires persistent storage, such as MongoDB, which is recommended for this solution.

IoT Discovery

Within IoT Discovery[14] Generic Enabler, two software components are offered: the NGSI-9 server, as well as the Sense2Web platform. The NGSI-9 server provides a repository for the storage of NGSI data and allows conformant clients to register context information about sensors and things and discover context information using ID, attribute, attribute domain, and entity type. Such clients may include the other FIWARE GEs as well, in particular, the Data Handling GE, the Device Management GE for registration, and the IoT Broker for discovery.
The Sense2Web software component is a platform which offers a semantic repository for IoT providers to register and manage semantic descriptions (in RDF/OWL) about their "things", whether they will be sensor/actuator devices, virtual computational elements (e.g. data aggregators) or virtual representations of any physical entity. On the other hand, it enables clients to discover these registered IoT elements by retrieving descriptions in RDF. It supports a probabilistic search mechanism that provides recommended and ranked search results for queries that don't provide exact matching property values. Further, it supports semantic querying via SPARQL and an association mechanism that associates things and sensors based on their shared attribute (e.g. temperature) and spatial proximity, which can then be queried via SPARQL.

IoT Agent

IoT Agent is a Generic Enabler (GE) in FIWARE Reference Architecture[15]. It is a component that allows a group of devices to send their data to and be managed from a Context Broker using their own native protocols. The Context Broker management of the entire lifecycle of context information including updates, queries, registrations, and subscriptions. IoT Agents are not only responsible for the communication aspect but are also concerned with the security issues of the FIWARE platform (authentication and authorization) and provide other common services to the device programmer.
Each IoT Agent provides a north-bound interface, which is used for Context Broker interactions and all interactions beneath this port occur using the native protocol of the attached devices. Essentially, this concept enables a standard interface to all IoT interactions north from an IoT Agent (no matter which (proprietary) protocol is used by the attached device. The standard interface used for this purpose is NGSI-LD[16].

4.2.4.2. Feasibility study

Context Broker

Many of the Orion Context Broker functionalities could be potentially used in SEMIoTICS project, hence this component has been investigated in detail. The first attempt at testing took place in December 2018 with negative results[17]. Basic functionalities did not work according to the documentation provided by the authors and moreover support from FIWARE was not able to solve the issue[18]. After nine months there a second attempt to examine the component has taken place. The new version of the software has solved the issue and

[14] https://fiware-iot-discovery-ngsi9.readthedocs.io
[15] https://fiware-tutorials.readthedocs.io/en/latest/iot-agent/index.html
[16] https://www.etsi.org/deliver/etsi_gs/CIM/001_099/009/01.01.01_60/gs_CIM009v010101p.pdf
[17] https://github.com/telefonicaid/fiware-orion/issues/3374
[18] https://stackoverflow.com/questions/53710837/conflict-error-when-obtaining-attributes-in-fiware-orion-context-broker

the component was working properly. Thus, the Context Broker may have been subjected to further analysis of its use in the project.
One of the difficulties in using this component is another format of thing description. Context Broker uses simple JSON and in SEMIoTICS project, JSON-LD is used. FIWARE is recently switching to NSGI-LD specification to enhance relationships between entities, but currently, it is up to the logic of the application (in this case SEMIoTICS platform) to navigate between entity relationships. It means that an additional component for mappings between these two formats is required. The main differences are shown in the diagrams below.

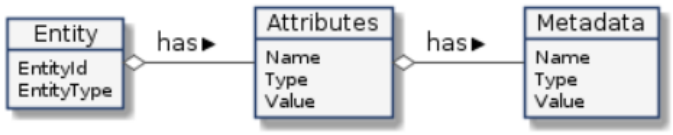


FIGURE 4-5 NGSI V2 DATA MODEL

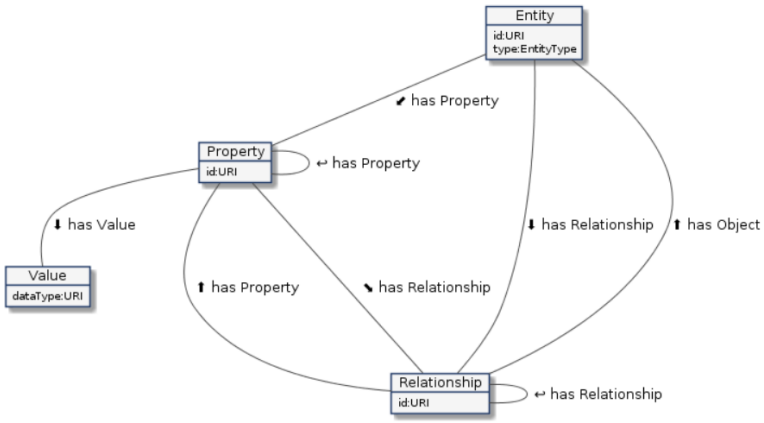


FIGURE 4-6 NGSI LD DATA MODEL AFTER MAPPING SIMPLE JSON

Furthermore, some functionalities of the Context Broker are covered by components already existing in SEMIoTICS. Thing Directory component enables a client to register new things to platforms, quickly search the repository using SPARQL filter or even delete devices. To update or read context information from brownfield devices, the Semantic API&Protocol Binding component can be used while other devices do not require any additional components. Moreover, Sematic API&Protocol Binding offers simple handling of all actions for the device.
However, Context Broker component can be used for monitoring the SEMIoTICS platform. Sending notifications or context information changes is a functionality that does not exist in a project yet. Users could define special queries or expressions to be notified only if selected property or group of properties has changed. What is more, the Context Broker provides collecting data from devices with a set frequency and store in the database, so it can be useful for historic data visualization.

IoT Discovery

For the SEMIoTICS system, this software component is not usable as the NGSI-9 format does not play a role in the component interactions defined in the architectural setup. In the SEMIoTICS systems, the discovery and metadata exchange for things goes beyond the generic RDF/OWL usage. Instead, it is proposed to use the W3C Web of Things format of “Thing Descriptions”[19], which can be represented in RDF but represents a semantically narrow format specifically designed for expressing thing metadata. Similarly, there is a dedicated

[19] https://w3c.github.io/wot-thing-description/

discovery component, the Thing Directory[20], that has been defined and implemented in the context of the W3C Web of Things suite of recommendations. This component has been selected for the SEMIoTICS architecture to cover the functionality of thing discovery, as its interface finely tuned to best support this specific purpose. In comparison, the Sense2Web component offers an all-purpose interface, with functionalities that go beyond the project's needs and hence would bloat the complexity of interactions.

IoT Agent

Only a few IoT Agents already exist. For example, for bridging HTTP/MQTT messaging (with a JSON and UltraLight2.0 payload) and NGSI, as well as for bridging Lightweight M2M and LoRaWAN with NGSI. IoT Agents for other protocols can be developed. Semantic IoT Gateway in SEMIoTICS architecture is responsible for the functionality of IoT Agents. The component provides a standardized semantic-based interface for the integration of brownfield devices, as well as for the integration of any other IoT devices.

4.2.4.3. Feasibility study outcomes

In SEMIoTICS - a W3C standard "Web of Things" (WoT) is promoted, according to which the device interface is described with, so-called, "Thing Description"21 (TD). An implementation of a WoT API, including protocol mappings (binding), also exist for this standard22. The model of TD is based on the concept of Interaction Patterns, as constructs that enable interactions with a thing (device). TD distinguishes Properties, Events, and Actions. The model further specifies security and other kinds of metadata. There has been a big contribution of the Consortium members and SEMIoTICS project itself to the creation of W3C standard. In the proposal to SEMIoTICS, it was declared to promote the W3C WoT standard so the project can not incorporate this group of general enablers into the SEMIoTICS platform. However, the Context Broker provides notification functionalities that could be used by SEMIoTICS. For this purpose, it is considered to develop a bridge towards NGSIv2 (which was defined by FIWARE and is provided by the Context Broker). Further verification is currently conducted to validate the use of Context Broker notification functionality as a part of SEMIoTICS platform.

4.2.5. GROUP 3: SDN AND NFV - RELATED COMPONENTS

In this group, only NetIC General Enabler is put. This is only one component in the FIWARE platform that provides functionalities in the network layer.

4.2.5.1. Feasibility study

The FIWARE Network Information and Control (NetIC) Generic Enabler is intended to provide abstract access to heterogeneous open networking devices. It exposes network status information and it enables a certain level of programmability within the network (depending on the type of network and the applicable control interface). This programmability may also enable network virtualization, i.e., the abstraction of the physical network resources as well as their control by a virtual network provider. Potential users of NetIC interfaces include network service providers or other components of FIWARE, such as cloud hosting. Network operators, virtual network operators, and service providers may access (within the constraints defined by their contracts with the open network infrastructure owners) the open networks to both retrieve information and statistics (e.g. about network utilization) and also to set control policies and optimally exploit the network capabilities.

4.2.5.2. Feasibility study outcomes

In SEMIoTICS, the functions of NetIC are covered by tools of the SEMIoTICS SDN Controller and the Network Function Virtualization component which are the core technologies to be developed within the SEMIoTICS project. Hence, using NetIC would fully double the functionalities already covered by SDN/NFV layer of SEMIoTCIS architecture.

4.2.6. GROUP 4: DATABASE RELATED COMPONENTS

[20] https://github.com/thingweb/thingweb-directory
[21] https://w3c.github.io/wot-thing-description/
[22] https://github.com/eclipse/thingweb.node-wot

In this group, two components that are related to the databases KNOWAGE and Object Storage are included. The first one is stand-alone tools for analyzing and visualizing big data sets. Object storage is a tool for database management.

4.2.6.1. Functionality summary

Knowage

Knowage is a powerful and complex tool for data set analysis and visualization. It can run analysis on data available from numerous online and offline DB, java classes from application, files or web apps through their API. Knowage allows creating various types of visualizations starting from simple tables, through many types of graphs ending on interactive maps. It offers many business analytics tools like periodic reporting, business predictions, and interactive cockpit. Knowage has many built-in pre-configured functions like sorting, grouping and other statistic functions. Using those included functionalities requires only a few configuration steps from the user, like pointing which column in the table is an attribute a which is a measurement.

Object Storage

Object Storage is one of the Generic Enablers within FIWARE. It is used for redundant and scalable data storage using clusters of standardized servers to store petabytes of accessible data. It is a long-term storage system for large amounts of static data that can be retrieved and updated. Object Storage of OpenStack that the GE of FIWARE is completely based on, as mentioned in the FIWARE wiki[23].

4.2.6.2. Feasibility study

Knowage

Knowage provides a REST API with an endpoint for many functionalities which can be used for faster and more robust integration with SEMIoTICS components. This approach covers the expectation for components in the SEMIoTICS platform. None of the already developed components provide such wide capabilities. Take into consideration the above-mentioned features provided by KNOWAGE we are eager to integrate this open-source software into SEMIoTICS platform.

Object Storage

Object Storage uses a distributed architecture with no central point of control, providing greater scalability, redundancy, and permanence. Objects are written to multiple hardware devices and can be files, databases or other datasets that need to be archived. Objects are stored in named locations known as containers. Containers and objects can have metadata associated with them, providing details of what the data represents. Similar to files in a traditional file system - objects in an object store belong to a certain user (account). This GE is ideal for cost effective, scale-out storage. It provides a fully distributed, API-accessible storage platform that can be integrated directly into applications or used for backup, archiving, and data retention.

4.2.6.3. Feasibility study outcomes

The use of the Knowage capabilities is planned to be leveraged in GUI component. Delivery of a dashboard visualizing the data from IoT devices is planned. Leveraging Knowage allows GUI user to benefit from the wide range of widgets available in the Knowage cockpit component. This powerful tool is to be used to present data collected from field devices. Object Storage is to be responsible for the management of the database. It coordinates the usage of disc space, creating backups, archiving and data retention.

4.2.7. CONCLUSION

SEMIoTICS platform is to be integrated with two General Enablers offered by FIWARE framework. They are advanced and robust components. This integration will improve the capabilities of the SEMIoTICS framework. Additional benefits from this integration process are the propagation of open-source FIWARE components along with IoT enthusiasts and professionals in the IoT sector and enhance interest in FIWARE General

[23] https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Object_Storage_Open_RESTful_API_Specification

Enablers as components that can be incorporated in many future projects as robust, safe and easy to use components.
Development of bridge between SEMIoTICS Monitoring Component and Context Broker may allow bringing some of the features provided in the FIWARE platform to SEMIoTICS users and vice versa.

4.3 Integration with CloE-IoT

The CloE - IoT platform aims to simplify the integration of highly distributed, complex and robust IoT solutions exploiting computational resources both in the cloud and at the edge. CloE-IoT is developed by ENG to support its IoT projects and products. Starting from 2020 the CloE-IoT platform is part of the Digital Enabler ecosystem[24].
The CloE - IoT platform offers APIs to access a set of functionalities specifically targeting common IoT requirements (connectivity, device management, security, data storage, etc.) allowing developers to focus on their domain-specific requirements (Figure 4-7).

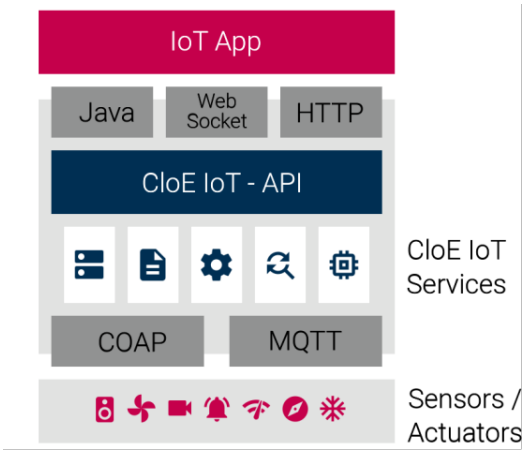


FIGURE 4-7 CLOE-IOT SOFTWARE LAYERS

The CloE-IoT platform supports applications with time- and safety-critical requirements by allowing application logic to be deployed on resource-constrained edge gateways (e.g. smartphones, vehicles, mobile robots): with CloE-IoT platform functionalities available locally even in case of failure of communication with CloE-IoT cloud nodes(Figure 4-8).

[24] https://www.eng.it/en/our-platforms-solutions/digital-enabler

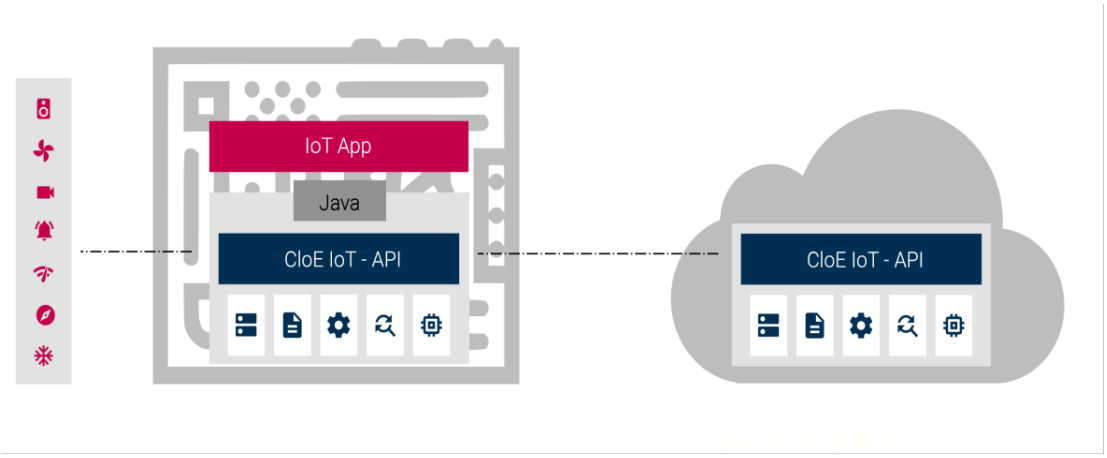


FIGURE 4-8 CLOE-IOT GATEWAY HOSTING APPLICATION LOGIC

The CloE-IoT platform supports applications that need to manage the trade-off between different requirements (e.g. reliability, power consumption, latency, fault-tolerance) by allowing both application logic and platform features to be distributed over a cluster of CloE-IoT enabled gateways (Figure 4-8 CloE-IoT gateway hosting application logic)

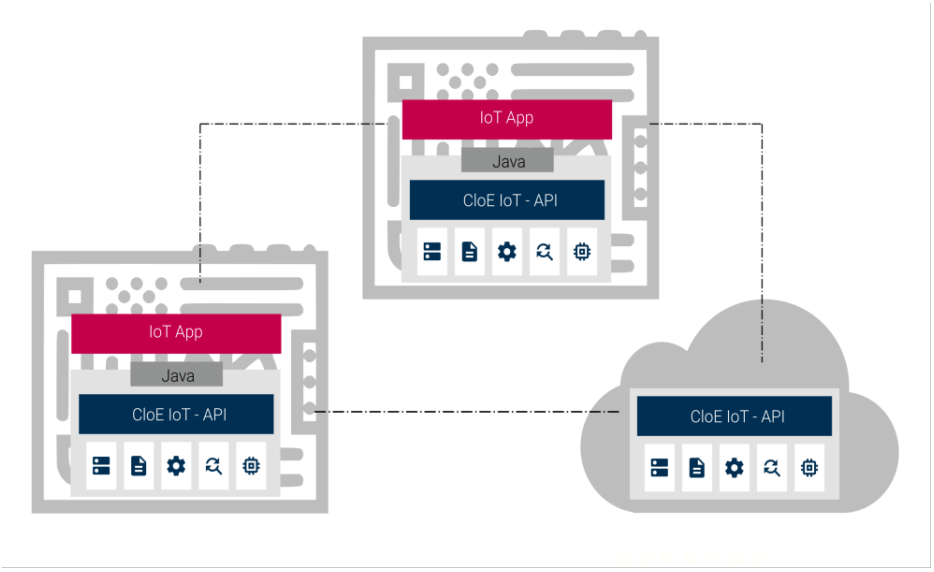


FIGURE 4-9 CLOE-IOT DISTRIBUTED APPLICATION

For what the integration with the SEMIoTICS framework is concerned, the most relevant one is the Client API, the Model API, and the Event API:

- The Client API allows an application to discover the IoT devices registered in an instance of the CloE - IoT platform. IoT devices register itself into a CloE - IoT node using the LwM2M protocol.

- The Model API allows an application to retrieve the resources exposed by registered devices via their object model (i.e. a data structure wherein each element represents a resource, or a group of resources, belonging to a device).

- The Event API allows applications to be periodically notified about the state of the resources hosted by the IoT registered devices. Notifications are pushed towards applications using the WebSocket protocol.
- The integration with the SEMIoTICS framework is achieved by developing a new agent bridging the CIoE-IoT services exposing the above-mentioned APIs with the corresponding components of the SEMIoTICS framework. In particular:

- the Device Manager (exposing the Client API) is to be extended to connect to the SEMIoTICS Thing Directory.

- the Model Provider (exposing the Model API) is to be extended to be able to retrieve Thing Descriptions from the SEMIoTICS Semantic Mediator.

- The Event Manager (exposing the Event API) is to be extended to support the WoT standard and hence to manage events raised by the devices discovered via the SEMIoTICS Thing Directory.

At the same time, the development of a specific signaller (see SEMIoTICS Deliverable D4.2 - “SEMIoTICS Monitoring, Prediction and Diagnosis Mechanisms (first draft)”) is to make the CIoE-IoT platform observable by the SEMIoTICS Monitoring Component. In particular, this signaler will make it possible for the SEMIoTICS Monitoring Component to observe events generated by the CIoE-IoT Event Manager via the FIWARE NGSI v2 interface.

4.3. Integration with MindSphere

SEMIoTICS IoT Gateway is a component that will be integrated with MindSphere, which is the IoT operating system from Siemens[25]. The gateway, among others, provides a mechanism to semantically annotate bootstrapped devices (if a semantic description for them does not exist). The same semantic description of devices can be used for creating digital representation in MindSphere. This procedure is supposed to take place during the onboarding process of a device or an automation system.

MindSphere provides its own information model that is called the Asset Data Model[26]. The model distinguishes notions of Asset, Aspect, and Datapoint. An Asset is a digital representation of a machine or an automation system with one or multiple automation units (e.g. PLC) connected to MindSphere. Aspects are data modeling mechanisms for Assets. Aspects are grouping related data points based on their logical association. Datapoints are points that provide certain functionality, thereby providing and/or consuming data. Examples of the datapoints are electric "power", "current", "voltage" etc.

[25] https://siemens.mindsphere.io/en
[26] https://documentation.mindsphere.io/resources/pdf/asset-manager-en.pdf

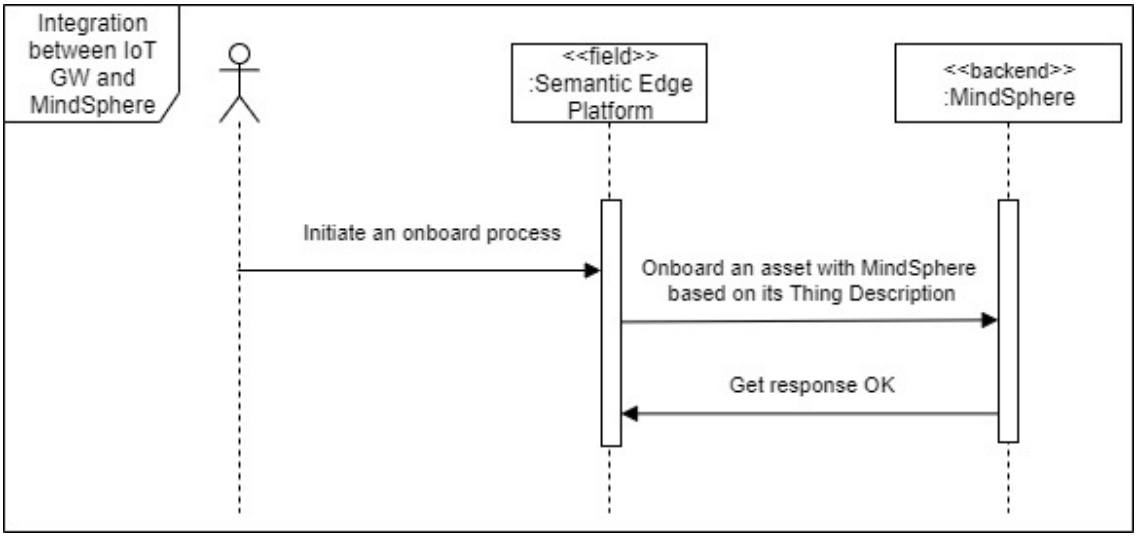


FIGURE 4-10 SEQUENCE DIAGRAM – INTEGRATION OF IOT GATEWAY AND MINDSPERE

Figure 4-10 shows a simplified sequence diagram related to the onboarding process of a device that has already been bootstrapped and for which a Thing Description already has been created, see Section 3.1.2. A user may initiate the onboarding process via Semantic Edge Platform as soon as a Thing Description (TD) has been created for a device. Following the semantics provided in TD, SEMIoTICS IoT Gateway (via Semantic Edge Platform) will interact with MindSphere API in order to automatically create an Asset Data Model. In this way, it will be ensured that the semantics created at the Edge level (by the gateway) is used at the Cloud level too. This approach, proposed by activities in Tasks 3.3, easies creation maintenance of applications since both Cloud- and Edge applications will be based on the same semantic model.

4.4. Integration with OpenHAB

Use Case 3 of SEMIoTICS leverages OpenHAB 2 for sensor value visualization via charts. OpenHAB is written in Java and uses Apache Karaf to create an Open Services Gateway initiative (OSGi) runtime environment. Jetty is used as the HTTP server, which implements the Dashboard and Management GUI and also hosts the OpenHAB REST API. OpenHAB is extended through “add-ons” that handle the interaction with external sensors, data storage backends and chart libraries for sensor value visualization. Furthermore, OpenHAB supports a scripting language to implement automation and “if-this-then-that” scenarios. For example, automation scripts will allow us to combine measurements from multiple sensors, and generate alerts if certain sensor values exceed the specifications.
As previously mentioned, in order to interact with sensors and actuators over the network, a RESTful service is offered by OpenHAB, that gives access to Things, Channels and Items.

- **Things** are entities that can be physically added to a system. They may provide more than one function (for example, a Z-Wave multi-sensor may provide a motion detector and also measure room temperature). Things do not have to be physical devices; they can also represent a web service or any other manageable source of information and functionality. From a user perspective, they are relevant for the setup and configuration process, but not for the operation. Things can have configuration properties, which can be optional or mandatory. Such properties can be basic information like an IP address, an access token for a web service or a device-specific configuration that alters its behavior. Things expose their capabilities through Channels.
- **Channels** represent the different functions the Thing provides. Where the Thing is the physical entity or source of information, the Channel is a concrete function from this Thing. A physical light bulb might have a color temperature Channel and a color Channel, both providing functionality of the one light bulb Thing to the system. For sources of information, the Thing might be the local weather with

information from a web service with different Channels like temperature, pressure and humidity. Channels are linked to Items, where such links are the glue between the virtual and the physical layer. Once such a link is established, a Thing reacts to events sent for an item that is linked to one of its Channels. Likewise, it actively sends out events for Items linked to its Channels. Whether an installation takes advantage of a particular capability reflected by a Channel depends on whether it has been configured to do so. When you configure your system, you do not necessarily have to use every capability offered by a Thing. You can find out what Channels are available for a Thing by looking at the documentation of the Thing's Binding.

- **Bindings** can be thought of as software adapters, making Things available to the system. They are add-ons that provide a way to link Items to physical devices. They also abstract away the specific communications requirements of that device so that it may be treated more generically by the framework.
- **Items** represent capabilities that can be used by applications, either in user interfaces or in automation logic. Items have a State which may store sensor values and they may receive commands (e.g., for actuation purposes).

After successfully deploying the Data Collection system and correctly configuring the Bindings, Channels, and Things, third party clients simply need to send HTTP GET requests to interact with OpenHab, e.g., sending sensor values for visualization via its charting system. The developed interface is presented in Figure 4-11.

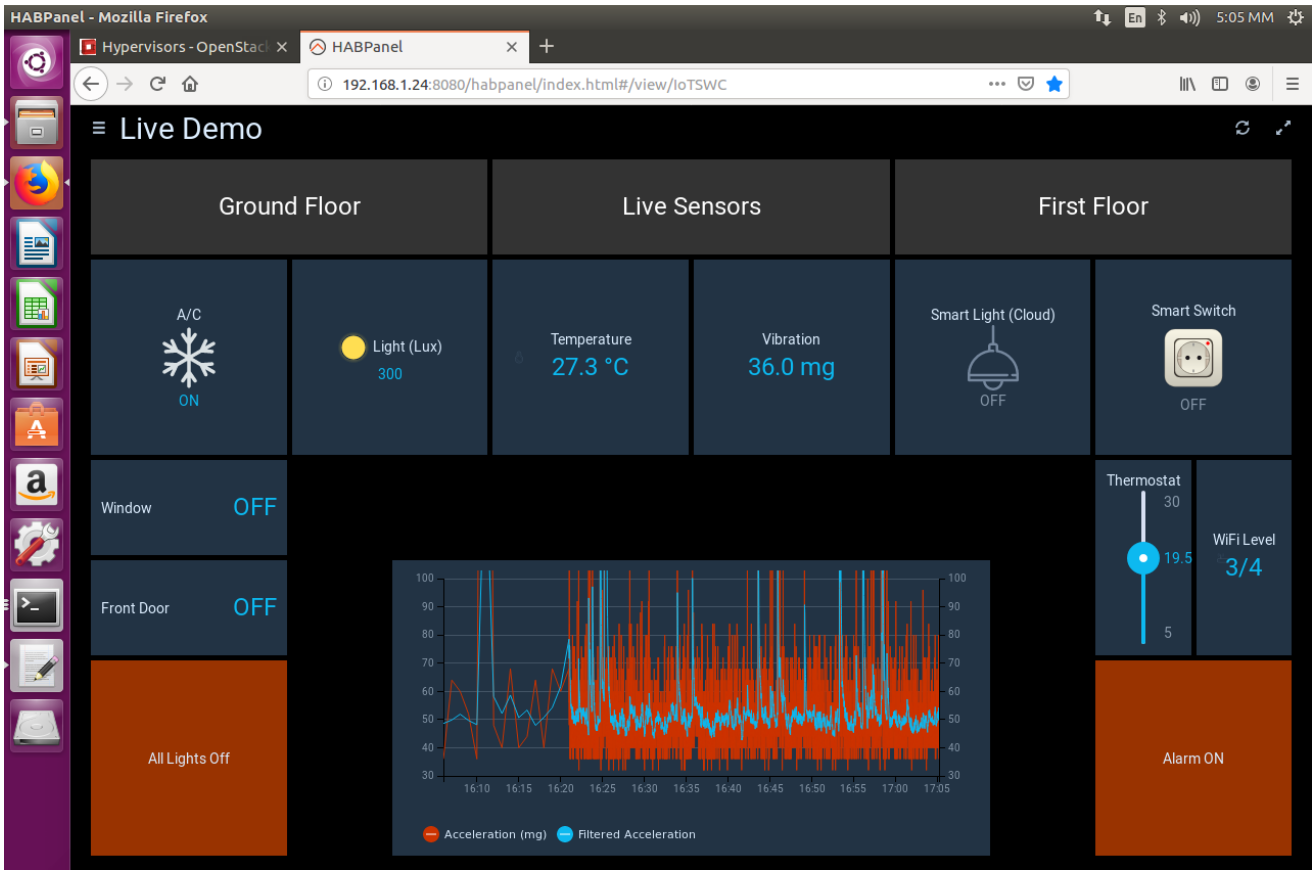


FIGURE 4-11 THE OPENHAB GRAPHICAL USER INTERFACE

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)

Table 6 presents the task objectives and appropriate sections addressing those.

TABLE 6 TASK 5.2 OBJECTIVES

T5.2 Objectives	D5.2 Sections
• Integration, and delivery of the SEMIoTICS framework will all the components developed by WP3 and WP4	3
• Interoperability with targeted external IoT enabling platforms (i.e., FIWARE, CIoE-IoT and MindSphere)	4
• Continuous integration and delivery processes with deployed development supporting tools and tools for automated platform scaling	2

The KPIs and their respective SEMIoTICS objectives that are related to Task T5.2 are described in Table 7.

TABLE 7 KPIS AND OBJECTIVES

Objective		KPI-ID	Description	Related task
1	SPDI Patterns	KPI-1.1	Number of SPDI Patterns	T4.1
1	SPDI Patterns	KPI-1.2	Pattern Language	T4.1
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
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-6.2	Leveraging upon FIWARE assets	T5.3
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
7	Promote the Adoption of EU Technology of EU Technology Offerings Internationally	KPI-7.1	Provision of the SEMIoTICS framework and building blocks	T5.2

5.2. SEMIoTICS implementation requirements

The relevant SEMIoTICS requirements that are indirectly covered by the presented software integration of SEMIoTICS components are summarized in Table 8. It is important to note that all the mentioned requirements, are tracked and described in detail within relevant tasks assigned within the matrix represented in Deliverable

SEMIoTICS high level architecture (final). The requirements which are use case specific are in details covered within the Tasks T5.4, T5.5 and T5.6, respectively.

TABLE 8 REQUIREMENTS´ CORRELATION

Requirements (D5.3)	Description	Related task	Status
R.GP.4	Detection of events requiring a QoS change and triggering network reconfiguration need by SPDI pattern	T3.1, T3.4, T3.5, T4.1, T4.2, T5.4, T5.5	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)	T3.1, T3.4, T3.5, T5.4	Delivered
R.BC.18	The backend layer must feature SPDI pattern reasoning embedded intelligence capabilities	T3.5, T4.1	Delivered
R.NL.10	Interfaces among the MANO and the VIM must ensure seamless interoperability among different entities of the Backend Cloud	T3.1, T3.2, T3.5	Delivered
R .NL.12	The network layer must feature SPDI pattern reasoning local embedded intelligence capabilities	T3.4, T3.5, T4.1	Delivered
R.FD.9	Field devices MUST be able to communicate with the IIoT Gateway / other architectural components.	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.)	T3.1, T4.1, T5.5	Delivered
R.S.4	All components from gateway, via SDN Controller, to cloud platforms and their users MUST authenticate mutually.	T3.2, T3.4, T4.1, T4.5, T5.5	Delivered
R.S.17	There MUST be an interface between the network controller and the network administrators for the designation of the applications' permissions.	T4.1	Delivered
R.S.18	All network functions SHALL be mapped to application permissions	T4.1	Delivered
R.UC1.1	Automatic establishment of networking setup MUST be performed to establish end-to-end connectivity between different stakeholders	T3.1, T3.3, T3.4, T4.1, T5.4	Delivered
R.UC1.2	Automatic establishment of computing environment MUST be performed in IIoT Gateway for the minimum operation of the IIoT devices through 5G network controller based on SDN/NFV	T5.4	Delivered
R.UC1.8	Semantic and robust bootstrapping/registration of IIoT sensors and actuators with IIoT gateway MUST be supported.	T3.3, T4.4, T4.5, T5.4	Delivered

6. CONCLUSION

This deliverable detailed the work performed in WP5 related to the first and second cycle of components' integrations. In order to structure and organize the work properly within task 5.2, sequence flow diagrams for common functionalities were created as well as diagrams for each of the three use cases. From the diagrams, a complete list of interactions between SEMIoTICS components was extracted. Such a list has allowed the component owners to identify the necessary interactions, and to verify whether any APIs and component functionalities were missing.

The integration work carried within Task 5.2, took place in a number of parallel workstreams.

The first stream focused on one of the core framework capabilities which is enabling SPDI pattern distribution to different layers of SEMIoTICS framework as well as their definitions and visualization. The integrations of Pattern Engines and the Pattern Orchestrator components consumed significant effort during cycle 1. Integration between Pattern Engine and Pattern Orchestrators itself was obviously a part of the work delivered. Moreover, the integration of the Recipe Cooker and Pattern Orchestrator was carried out, in order to give a possibility for defining SPDI properties within the Recipe Cooker, consequently translated to specific SPDI pattern requirements, with a GUI capable of visualising the status of the patterns in different flows modelled in the Recipe Cooker. Furthermore, integration of the Pattern Engines and Orchestrator with the SDN/NFV toolbox allows for providing security guarantees through the traffic forwarding via different network security functions. Field devices bootstrapping was also covered, as one of the core flows required for any other flow to take place in the process. Thanks to the GUI component it is currently possible to fully interact with Thing Directory, connect and pull data from WoT compliant devices.

A number of integrations were also carried out in the 2nd cycle. The capacity of the GUI was expanded. Key integration, GUI - Security Manager will protect against unauthorized access to data. Before using the GUI functionality, the user is asked to provide his login and password. On this basis his identity is identified, and his rights are verified. In addition, the integration with Monitoring was carried out. This makes it possible to define high-level event patterns from the user interface. Integration with Security Manager made also for Backend Pattern Engine. It is used in the Use Case 2, which ensures that patient location data is only provided to authorized users, and that if it is provided to an authorised entity due to an exceptional case, e.g. medical emergency, a rule allows to indicate the loss of privacy during monitoring. Moreover, as part of cycle 2, the integrations of Monitoring Component with Pattern Engine, Recipe Cooker with the Thing Directory were made and the integration for Backend Semantic Validator was prepared.

The second part of the work was focused on semantic interoperability with the SEMIoTICS framework as well as with IoT frameworks external to SEMIoTICS. The integration of the Backend Semantic Validator with other components has been done, in order to enable semantic interoperability both internally (within SEMIoTICS) as well as with other IoT platforms. All of the platforms used by different use cases were covered: CLOE-IOT, MindSphere, and OpenHab. Additionally, a feasibility study around FIWARE GEs was performed. The results of the FIWARE feasibility study allowed the consortium to identify specific GEs which can be leveraged by the project and ones that need to be discarded (due to different reasons, such as ceased support for the component, not supported core standards, etc.).

Deliverable D5.7 is the second output of Task 5.2. This document focussing documenting all missing integrations according to the diagrams prepared in previous version. Any changes in sequence flow or architecture have been be taken into consideration, if such occur during development. Next step, the framework integration will be deployed and evaluated within the testbed deployment and testing described and delivered in Task 5.3.