D4.2 Service Platform First Operational Release and Documentation

Project Acronym	SONATA
Project Title	Service Programing and Orchestration for Virtualized Software Networks
Project Number	671517 (co-funded by the European Commission through Horizon 2020)
Instrument	Collaborative Innovation Action
Start Date	01/07/2015
Duration	30 months
Thematic Priority	ICT-14-2014 Advanced 5G Network Infrastructure for the Future Internet

Deliverable	D4.2 Service Platform First Operational Release and Documentation
Workpackage	WP4 Resource Orchestration and Operations repositories
Due Date	November 30th, 2016
Submission Date	December 23rd, 2016
Version	0.1
Status	To be approved by EC
Editor	José Bonnet (AlticeLabs)
Contributors	José Bonnet, Alberto Rocha, Miguel Mesquita (AlticeLabs), Santiago Rodríguez (Optare), Aurora Ramos, Felipe Vicens (ATOS), George Xilouris, Stavros Kolometsos, Christos Sakkas (NCSRD), Dario Valocchi (UCL), Theodore Zahariadis, Panos Trakadas, Panos Karkazis, Sotiris Karachontzitis (SYN), Thomas Soenen (IMEC), Sharon Mendel-Brin (Nokia), Michael Bredel (NEC), Muhammad Shuaib Siddiqui, Dani Guija (i2CAT), Manuel Peuster, Sevil Dräxler, Hadi Razzaghi Kouchaksaraei (UPB)
Reviewer(s)	Zichuan Xu (UCL), Michael Bredel (NEC), Bruno Vidalenc (THALES)

Keywords:

Service Platform, orchestrator, gatekeeper, VIM

Deliverable Type		
R	Document	**X**
DEM	Demonstrator, pilot, prototype	
DEC	Websites, patent filings, videos, etc.	
OTHER		

Dissemination Level		
PU	Public	**X**
CO	Confidential, only for members of the consortium (including the Commission Services)	

Disclaimer:

This document has been produced in the context of the SONATA Project. The research leading to these results has received funding from the European Community's 5G-PPP under grant agreement n° 671517.
All information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.
For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the authors' view.

Executive Summary:

5G is posing significant technical and non-technical challenges to our society. The SONATA NFV project has chosen, among others, the **flexible programmability of 5G networks**, by providing Communication Service Providers a Service Platform that can accommodate their needs in this new and much more challenging (5G) environment. This platform must be **highly flexible**, to be able to be adapted to different markets and segments and support new and unforeseen services, but **secure**, so that only authorised people can change the platform's behaviour in a controlled way. Higher flexibility comes from the capability to **extend** the platform, in a microservice-oriented architecture, and security is achieved by using a **gatekeeper** supporting the authentication and authorisation burden, out of the main service deployment loop. Short times to deploy new or updated services, as well as providing adequate mechanisms and environments for testing these new or updated services also support the platform's needed flexibility.

These mechanisms described in this deliverable, covering the work done from the previous deliverable (D4.1, Orchestrator Prototype) and the first year project review. As Agile Methodologies supporters, we are not designing everything up front, but opted to do it iteratively, together with some implementation, testing and deployment.

After having proved the whole concept in the first year, we are now making the Service Platform more secure, with users having to register themselves and APIs accessing the platform by using HTTPS. We are now able to control every service usage through a licensing mechanism, and collect KPIs on each API usage and performance. Functions can now have their Specific Managers (Services could already have these in the first year) changing the platform's default behaviour in scaling and placement. These Specific Managers can be securely uploaded, having their interactions with the rest of the platform restricted to an adapter component name Executive Plugin.

We are considering kind of Virtual Infrastructure Manager (VIM), based entirely on containers. This option will put to the test our Infrastructure Abstraction layer, which will have to support a different VIM from the ones more similar to OpenStack.

After this second year, developers will be able to ask the Service Platform for near real-time monitoring data about their services and functions. This mechanism is already designed, and is being implemented at the time of writing.

The **DevOps** approach in the development of the Service Platform itself will be enhanced, giving us an edge over the different kinds of problems Developers will face when deploying services and functions using it.

Support for the expansion of the open-source community around the Service Platform implementation will also increase, by contributing to the social network channels the project has, and contributing to other projects of that community.

Contents

List of Figures

List of Tables

1 Introduction

This is the second deliverable related to the SONATA's Service Platform development and it documents the evolution that has happened since our first deliverable, D4.1 [7] and the first year review.

Results of the first ten months of work documented in **D4.1** were crucial for proving several concepts we thought as key: the highly dynamic cycle of **producing a new service** using the SDK, comprised of one or more functions, **describing it** in a standard way (following a schema), **submitting** it to the Service Platform, having it **instantiated** in a real VIM and providing **monitoring** data was a very important result. The next step, already demonstrated in the first year review of the project, was to be able to submit a **Specific Service Manager** that changed the platform's behaviour for that service only: this is another innovation in terms of currently available Service Platforms and Orchestrators. And in parallel with these two steps, was the **DevOps approach** we have chosen to adopt from the beginning in developing the Service Platform itself: we have chosen the best set of tools the open-source community uses, like **GitHub**, **Jenkins**, **Docker**, **Ansible**, etc., and put them working together into an agile approach for developing software. All this released into the **open-source community**, which is something also worth to mention.

In this deliverable, released after seventeen months of work, we describe the features we felt were next in terms of value added to the platform. The list of new features is extensive and is explained throughout the Deliverable. New modules, supporting these new features, will also be implemented as micro-services, thus keeping the flexibility in terms of scalability we have designed from the start. Each module will provide the most adequate kind of interface, either REST-based or message-based.

Some of those features are really innovative (e.g., a container-based VIM, working in a MANO context), therefore deserving some more time of consolidation of the proposed solution. For these features, we present the state-of-the-art of what we have designed so far, and leave the definitive solution to be presented later.

Overall, these features push the SONATA Service Platform even further into a 5G world, where high **operational efficiency**, global **scale** and extreme **flexibility** are at stake. Higher operational efficiency is achieved by NFV and SDN, taking advantage of computational, storage and connection resources virtualisation. High scalability is supported by our architectural choices (already documented in **D2.3** [5]), namely the option for a micro-service based architecture, in which each component can scale separately. Higher flexibility comes as a side effect of a micro-service based architecture, where different components responding to the same interface (which can be a set of messages) can easily be interchanged, thus improving or changing a Service Platform functionality without extensive impacts.

1.1 Content organisation

The content of this deliverable is organised as follows.

The current section is the **Introduction**, where the whole document is summarised. Then, in Section 2, **Service Platform Security**, we describe the overall security policy that is being implemented across the platform. Section 3, **Gatekeeper**, explains the changes we have designed and implemented since **D4.1** [7] in this key component of the Service Platform, including new modules (users, licences and KPIs) and changes in the existing ones (mostly the API, the GUI and the BSS). Section 4, **Catalogues and Repositories**, describe the changes designed and implemented on those components this year and Section 5, **The MANO Framework**, details the evolution that took place on that crucial Service Platform sub-system, namely with the introduction of the **Function** and **Service Lifecycle Managers** and the introduction of **Specific Managers Infrastructure**. Section 6, **Infrastructure Abstraction**, discusses the design changes needed to accommodate an infrastructure as distinct from the common VIM's used as a container-based VIM. **Monitoring**, in Section 7, also evolved a lot since D4.1 [7], namely in providing monitoring data efficiently to the Developer's SDK. To make the Service Platform's modularity more bold, we present all its **Internal Interfaces** in Section 8 and finally we draw some **Conclusions** from these months of work in Section 9.

Annexes hold the list of the **Acronyms** used in Appendix A and the **Glossary** in Appendix B.

2 Security within the Service Platform

The security within the Service Platform includes the authentication and authorisation between microservices' ([12]) APIs, HTTPS for external APIs and the enforcement of security in the Message Broker and databases.

2.1 Authentication and authorisation between microservices APIs

The communication between microservices is secured through **access tokens** that are generated by the security component described in SONATA Architecture D2.3 [5] as AuthC/AuthZ. This module deals with identifying microservices and controlling their access to other microservices within the Service Platform by associating user rights and restrictions with the established identity.

JSON Web Token (JWT, [1]) is an open standard [19] that defines a compact and self-contained way for securely transmitting information between different parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with HMAC algorithm [16]) or a public/private key pair using RSA [40].

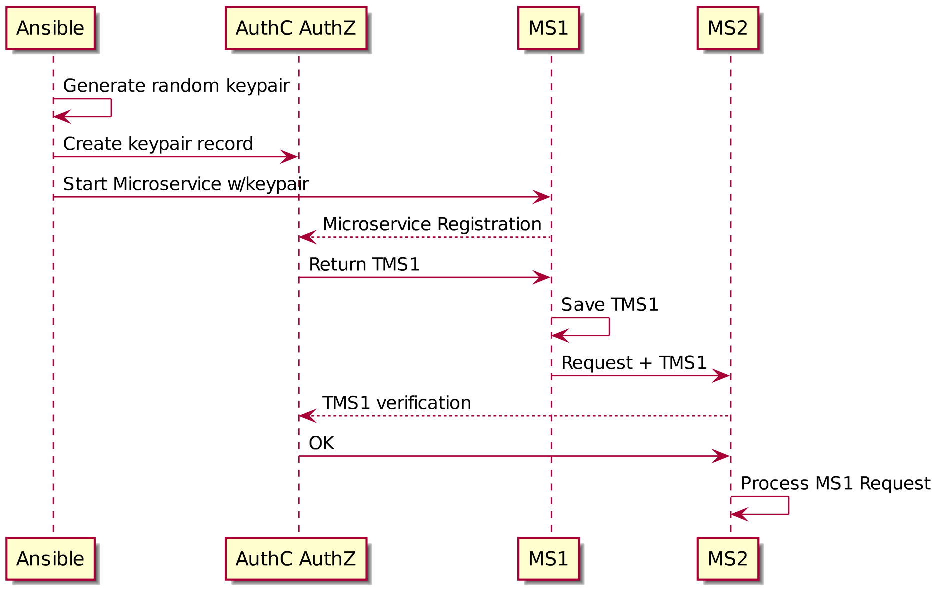


Figure 2.1: Service Platform internal web APIs Authentication and Authorization

In the SONATA Service Platform (SP), the JWT is used in order to authenticate and authorise the SP components. In the installation process, a keypair will be generated and injected to each container (AuthC/AuthZ, MS1, MS2 in Figure 2.1) in the start process. The microservice will use this keypair in the registration and one token will be assigned to it. Once microservices are registered and have their own token, they will be able to communicate with other microservices APIs.

Table 2.1 lists the set of microservices in which we will implement authentication and authorization.

Table 2.1: Web APIs where Authentication and Authorization will be implemented

#	Microservice A	Microservice B
1	GTK-API	BSS GTK-VIM GTK-REC GTK-USR GTK-LIC GTK-KPI GTK-SRV GTK-PKG GTK-FNCT
2	GTK-FNCT	Catalogues
3	GTK-PKG	Catalogues
4	GTK-SRV	Catalogues
5	GTK-REC	Repositories
6	SLM	Repositories Monitoring
7	FLM	Repositories

This impacts both the **installation scripts** and **microservices APIs**.

2.2 HTTPS in external Web Components

Another level of security is in the **external connections**: the **SDK** [6], the **GUI** (see Section 3.5) and the **BSS** (see Section 3.6).

In this case, security will be achieved at the **connectivity** level, using HTTPS and Web Secure Sockets (WSS) [17].

2.2.1 HTTPS versus Web Secure Sockets

Web Sockets ([18]) is a distinct and independent protocol from HTTP, only using the HTTP protocol in its handshake, as an upgrade request on the Web Server. Due to its possible interactive nature it facilitates the exchange of real-time information greatly between peers. Its functionality is achieved by the definition of a standard way to send information from the server to the client without it being requested. Web Sockets thus makes it possible to open an interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply. It's clearly innovative because it provides a full duplex communication over a single TCP connection. Although its specially designed to be used between a web server and a web browser, the protocol can be used between any client and server.

Please note that this section is not an endorsement of one technology over the other. In fact, both are perfectly viable individually, serving to attain different objectives. If we wish to have absolute control on the part of the client, issuing transactional requests, evidently HTTP is an obvious choice. If a more interactive communication, with possible actions from the part of the server, is expected, Web Sockets ([18]) are the only choice.

The common use of the port `80` also provides an easy way of avoiding firewall blocking of information. Web Sockets also provides a secure way of communication with the aim to produce secure stream communications. So insecure communications have the URI `ws` and secure ones would be identified by the URI `wss`.

The protocol handshake aims to establish a Web Socket connection for which we have the following bidirectional messages:

Client request (just like in HTTP, each line ends with \r\n and there must be an extra blank line at the end):

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

Server response:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

After accepting success the exchange of the specific protocol begins. A obvious advantage in using the same port for both protocols (HTTP and WS) and (HTTPS and WSS) to provide the Web Sockets is to use the same mechanism to secure the network: a base64 code is sent, to avoiding replication of the packet by a caching proxy, without providing any type of authentication, privacy or integrity.

During the specific protocol phase, the messages are minimally framed with a small header followed by payload. The messages could be split across several frames in which the end would be signalled by the FIN bit on the protocol. Further extensions will provide multiplexing of frames to avoid starvation from a source with big data transmissions.

2.2.2 Distinct methods for external connections

First we will need to create the files associated with the website of the model we have chosen to provide the microservices. See the sections 'Reverse Proxy for distinct URLS' and 'direct access to distinct ports'.

The advantage of using letsencrypt ([11]) for producing a global security certificate in Ubuntu is its cost: it's free and recognized and trusted as a certificates provider entity. To install letsencrypt and generate the first SSL certificate for a specific URL we need to execute the following commands (it's preferable to achieve root status than to issue sudo before each command -- when addressing security issues most of the commands will need root permissions anyway):

```
$ sudo su
$ cd ~root
$ // clone the letsencrypt git repository.
$ git clone https://github.com/letsencrypt/letsencrypt.git letsencrypt
$ cd letsencrypt
$ // and request the SSL certificate:
$ ./letsencrypt-auto certonly --webroot \
> -w /home/web/sp.sonata-nfv.eu/public_html \
> -d 'sp.sonata-nfv.eu'
```

If you request a certificate for the master domain (1st level domain aka 'sonata-nfv.eu') use `-d` parameter twice. With and without WWW prefix like this:

```
$ ./letsencrypt-auto certonly --webroot \
> -w /home/web/sonata-nfv.eu/public_html \
> -d sonata-nfv.eu -d sp.sonata-nfv.eu
```

Without this, the certificate won't be valid for visitors opening the site with the `www` prefix. You can add as many subdomains as needed.

2.3 Authentication and authorisation within Message Broker

This section describes the authentication and authorisation mechanisms implemented for access control in the SONATA Message Broker.

As described already in [4] and [7] we are using **RabbitMQ** as SONATA's SP **Message Broker**. RabbitMQ instances can be used to carry sensitive application data or affect the stability of an entire system, we need to make sure that our RabbitMQ deployments are secured properly.

When a RabbitMQ client establishes a connection to a server, it specifies a virtual host within which it intends to operate. A first level of access control is enforced at this point, with the server checking whether the user has any permissions to access the virtual hosts, and rejecting the connection attempt otherwise.

2.3.1 RabbitMQ Virtual hosts

Virtual hosts are used to logically separate a broker instance into multiple domains, each one with its own set of exchanges, queues, and bindings. It is really similar to the Virtual Hosts of any Web Server in the enterprise. Clients have to choose one of the Virtual Hosts, since a Client cannot be allowed to connect to another Virtual Host while connected to one Virtual Host.

Resources, i.e. exchanges and queues, are named entities inside a particular virtual host; the same name denotes a different resource in each virtual host. A second level of access control is enforced when certain operations are performed on resources.

RabbitMQ distinguishes between configure, write and read operations on a resource. The configure operations create or destroy resources, or alter their behaviour. The write operations inject messages into a resource. And the read operations retrieve messages from a resource.

2.3.1.1 Access control configuration parameters

Access control configuration parameters are the following:

- Creates a virtual host: `add_vhost {vhost}`
- Deletes a virtual host: `delete_vhost {vhost}`
- Lists virtual hosts: `list_vhosts [vhostinfoitem ...]`
- Sets user permissions: `set_permissions [-p vhost] {user} {conf} {write} {read}`
- Sets user permissions: `clear_permissions [-p vhost] {username}`
- Lists permissions in a virtual host: `list_permissions [-p vhost]`
- Lists user permissions: `list_user_permissions {username}`

2.3.1.2 User Manager configuration parameters

User Manager configuration parameters are the following:

- Adding a User: `add_user {username} {password}`
- Deleting a User: `delete_user {username}`
- Changing a User password: `change_password {username} {newpassword}`
- Delete a Username password: `clear_password {username}`
- Lists users: `list_users`

2.3.1.3 Authentication

This parameter allows you to identify who connects to the message broker.

2.3.1.4 Authorisation

This parameter allows you to determine the set of privileges and permissions for the authenticated user. After a client is successfully authenticated by the message broker, it needs to perform some activities in some virtual hosts. The following types of permissions are configured in the message broker:

- **configure:** This allows a resource to be created, modified, or deleted
- **write:** This allows a resource to be written to
- **read:** This allows a resource to be read from

2.3.2 MANO Framework security workflow

The workflow of creating a virtualhost for each plugin, adding a user for each FSM/SSM, and setting permissions are shown in Figure 2.2

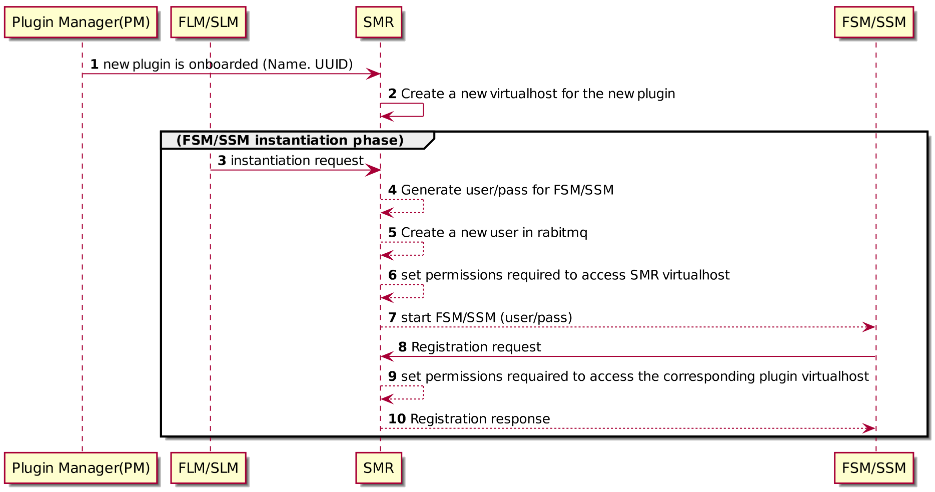

Plugin Manager(PM)
FLM/SLM
SMR
FSM/SSM
1 new plugin is onboarded (Name. UUID)
2 Create a new virtualhost for the new plugin
(FSM/SSM instantiation phase)
3 instantiation request
4 Generate user/pass for FSM/SSM
5 Create a new user in rabitmq
6 set permissions required to access SMR virtualhost
7 start FSM/SSM (user/pass)
8 Registration request
9 set permissions requaired to access the corresponding plugin virtualhost
10 Registration response
Plugin Manager(PM)
FLM/SLM
SMR
FSM/SSM

Figure 2.2: SSM/FSM security

Following two steps are taken for creating a new virtualhost:

- **Step 1:** Plugin manager sends a message to SMR whenever it successfully registers a new plugin. The message contains the new plugin name and UUID.
- **Step 2:** SMR creates a new virtualhost for the new plugin. Note that SMR creates a virtualhost for itself as well. This virtual host will be used for FSM/SSM registration.

FSM/SSM instantiation phase is used for adding a new user and setting permissions which are performed as follows:

- **Step 3:** SMR receives the FSM/SSM instantiation message from FLM/SLM.
- **Step 4:** SMR generates a pair of user/pass for the FSM/SSM.
- **Step 5:** SMR adds a new user to RabbitMQ using the generated user/pass in the previous step.
- **Step 6:** SMR sets permissions for FSM/SSM so that it can use the SMR virtualhost in order to register itself.
- **Step 7:** SMR starts the FSM/SSM container and sends the user/pass to the FSM/SSM through an ENV(environment variable).
- **Step 8:** Using the user/pass received from SMR, FSM/SSM sends a registration request message to SMR through SMR virtualhost.
- **Step 9:** In this step, SMR sets permissions on the FSM/SSM's virtual host, so that communication between the two is possible.
- **Step 10:** Eventually, SMR sends a response message to FSM/SSM.

2.4 Authentication in Databases

Authentication in Databases is a key issue whenever security is addressed. To avoid the definition of trivial and easy to guess or brute force attack passwords, the current trend is to generate them **randomly**. The next step is to pass them **securely** to the relevant containers.

Good practices in securing database users' passwords prescribes that they:

- must never be stored in a database as clear text
- must never be stored in disk at all
- should not be recoverable from the database, i.e., should be generated with a high degree of complexity to be extremely difficult to infer or crack.

2.4.1 Generate unencrypted random password in CLI

To generate unencrypted but random passwords in Linux CLI the following tools are commonly available: `makepasswd`, `mkpasswd`, `pwgen`. Example for `mkpasswd`:

Install `mkpasswd` on CentOS 7:

```
$ sudo yum whatprovides "*/mkpasswd"
$ sudo yum install expect
$ mkpasswd
```

Install mkpasswd on Ubuntu 14.04 or 16.04:

```
$ sudo apt-get install whois
$ mkpasswd
```

You can also store a password in a shell variable:

```
$ PGPWD=$(mkpasswd --method=SHA-512)
$ echo "$PGPWD"
```

More info on **mkpasswd** in [30]. There are other tools to generate random passwords, like: Using `/dev/urandom`

```
$ echo ‘</dev/urandom tr -dc A-Za-z0-9 | head -c8‘
```

Using `openssl` ([33])

```
$ openssl rand -base64 16
```

Generate an MD5 encrypted password for user ’sonata’

```
$ sudo usermod -p ‘makepasswd char=20 crypt-md5‘ sonata
```

2.4.2 Generate encrypted password for Databases users

Passwords are sensitive data. As such, encrypting the user’s password is crucial. Actually, there are several cipher algorithms like DES (1970), 3-DES, RSA (1978), IDEA (1990), PGP (1991), AES, and many more (see [42]). And there are also public tools to execute the ciphering, e.g., cryptographytools.com ([8]).

Straight DES encryption using 56-bit keys is considered not to be enough for modern computing capacity, since a password can be revealed within a day. There is also another drawback for DES: the same password produces the same key (see, e.g., online md5 hash calculator in [14]).

The password encryption method does not solve the problem of storing the password itself: a good security practice is not to store the password at all (not even encrypted), but to store the salted hash of the encrypted password instead.

2.4.2.1 Non-reversible encryption

Cryptographic methods apply an algorithm to the variable length input password and generates a ’hash’ that acts like a digital fingerprint of the original password. This is also called an ’asymmetric’ method: quick to generate but difficult to retrieve. Password-hashing function should protect against dictionary attacks and rainbow tables. Hashing algorithms commonly used are MD5, SHA-family. MD5 and SHA-1 does not introduce complexity enough since you can easily get two passwords with the same hash output. SHA256 uses 256-bit words, 32 bytes. These algorithms are also known as **message digests**: the submitted password at user’ login is kept in memory and its hash is computed. If the computed hash matches the stored hash, then his authentication is accepted.

Hash and Salt

However applying only an hash algorithm doesn't solve the problem of getting the same hash for identical password. By adding a piece of data called 'salt' (aka **nonce**, from *number used once*) along with the password in the hash calculation you can produce an output of unique hashes. Considering that salt is included to assure unique hash and it's not an encryption key, it can be stored in the users database as "username:salt:hash". To avoid hash collision choose a random salt: `/dev/urandom` in Linux systems.

Salt and Hash Stretching

The previous methods addresses the 'non-reversibility', 'no repeated hashes', and 'no hint of password length' requirements. However doesn't solve the scenario of an offline attack when an hacker with heavy computational power gets access to the users database for a long period of time. To slow down an offline attack, a number of iterations can be introduced in the hash calculation without slowing down the real user login. A number of repeated hash algorithms are available (eg, **PBKDF2**, **bcrypt**, **scrypt** or **argon2**) along with the HMAC-SHA-256 hashing algorithm.

2.4.2.2 How to protect SONATA database passwords

Finally lets see how to use non-reversible encryption to protect Sonata database passwords. The purposed method rolls on 2 steps:
1. generate a hash of the password

```
$ mkpasswd --method=SHA-512
Password:
$6$r/MVD5Lbu$...w0rICZV4xa1
```

2. store the hash as Ansible variables (eg. roles/pgsql/vars/main.yml)

```
# vars file for postgresql
db_name: son-sp
db_user: sonata
db_password: "$6$x03p5CK1GE . . . sAoetvgGeT90"
#db_password: "{ { lookup('env', 'PGSQLPWD') } }"
```

A security enforcement can be added by protecting the variables file with Ansible Vault:

```
$ ansible-vault create roles/pgsql/vars/main.yml
$ ansible-vault edit roles/pgsql/vars/main.yml
```

Now, just test and run it:

```
$ ansible-playbook --syntax-check --ask-vault-pass deploy-pgsql.yml
$ ansible-playbook --ask-vault-pass \
> --private-key=~/.ssh/id_ansible deploy-pgsql.yml
```

2.4.2.3 Internal database cryptographic tools

Currently, database engines used in Sonata Service Platform (PostgreSQL, MySQL and MongoDB) have built-in cryptographic tools. So, the developer has an important role on security when creating the data model. **PostgreSQL** provides cryptographic functions in the `pgcrypto` ([35]) module. In **MySQL**, the **PASSWORD()** function can be used, after the client connects, to generate a password hash or by using a password-generating statement (CREATE USER, GRANT, or SET PASSWORD)[10]. **MongoDB** uses a password hashing algorithm based on SCRAM-SHA-1 (PBKDF2 with 10,000 iterations).

2.4.2.4 Secrets backend

When a platform has many database accounts to manage, it could be a solution to store it on a dedicated repository like **Hashicorp Vault** [15] - 'a tool for managing secrets'.

Vault secures, stores, and tightly controls access to tokens, passwords, certificates, API keys, and other secrets in modern computing. Vault handles leasing, key revocation, key rolling, and auditing through a unified API'. Hashicorp provides Vault pre-compiled binaries for **LINUX** [26], for PostgreSQL [37], MySQL [28] and MongoDB [27] to be used as **Secret Backends**.

3 Gatekeeper

This section covers the improvements and new features implemented since D4.1 [7] and the first year review on the **Gatekeeper**.

First we document changes made to the Gatekeeper's API, in order to accommodate the security changes described in Section 2. Then, the new modules (**User Management** in Section 3.2, **Licence Management** in Section 3.3 and **KPIs Management** in Section 3.4) are detailed. Finally we list changes made to two of the Gatekeeper's API consumers, the **GUI** (in Section 3.5) and the **BSS** (in Section 3.6).

3.1 Gatekeeper API

The current section describes the new features of the Gatekeeper's API since [7].

3.1.1 Security related changes

Changes mentioned in Section 2 naturally impacted the Service Platform's API.

Since the Service Platform is now distinguishing its users (see Section 3.2), APIs will have to include some form (also discussed in Section 2) of authenticating and authorising what people and microservices want to do.

Every microservice will also have to proceed with an authentication procedure and other security verifications. For example, when a **Developer** submits a new package, the Gatekeeper will have to check if that developer has an authorisation for **on-boarding** packages.

3.1.2 New modules

In the next version of the Gatekeeper's API we will extend the current API to support the new modules of the Gatekeeper: the User Manager, the Licence Manager and the KPIs Manager.

Figure 3.1 shows these new modules, in yellow.

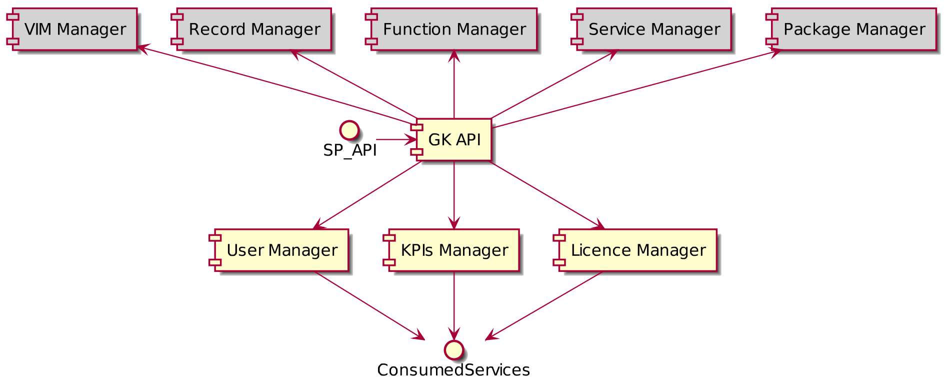


Figure 3.1: New modules to which the Gatekeeper's API has to interact with

These new modules will be detailed in their own section in this deliverable (please see Section 3.2, Section 3.3 and Section 3.4). The new internal interfaces that were made available are described in Section 8.

3.1.2.1 Requirements

This sub-section summarises requirements for three new modules.

User manager module

This component's requirements are detailed in Section 3.4.1, together with its API (in Section 3.4.5). We will mature the design and implementation of the module and only then make the API available to the rest of the modules. Table 3.1 summarizes component's requirements:

Table 3.1: User Management User Stories

ID	Summary	As a (user)	I want (action)	So that (benefit)	Applicable user role
UMAPI01	Registration	User	to be able to sign up to SONATA Service Platform	I can later publish and update services and check other public services offered	Developer, Customer, Provider, Admin
UMAPI02	Login	Registered User	to be able to sign in to SONATA Service Platform	I can view the status, validation and other info about my services	Developer, Customer, Provider, Admin
UMAPI03	Profile Update	Registered User	to be able to update my profile settings	I can provide the most up-to-date information to the Service Platform	Developer, Customer, Provider, Admin
UMAPI04	Authentication	Registered User	to be able to authenticate my requests to the Service Platform	The Service Platform acknowledges my identity	Developer, Customer, Provider, Admin
UMAPI05	Authorization management	Registered User	to be able to authorize other users of the Service Platform	The Service Platform grants access to my services and functions	Developer, Customer, Provider, Admin

Licence manager module

This component's requirements are detailed in Section 3.4.1.

Table 3.2 summarises the **job stories** [24] for the API to use the **Licence Manager** module.

Table 3.2: Licence Management User Stories

ID	Summary	When (event)	I want (action)	So I can (benefit)	Comments
LMAPI01	Define different types of licences	the platform starts	to define 'public' and 'private' types of licences	have different behaviour for these two different types of licences	
LMAPI02	Create licence	the BSS requests (*)	to create a licence for an end-user and a service (and its functions)	have services and functions validated when instantiated	Besides its type ('public' or 'private'), licences must reference its owner (user), end-user, service and a URL to be called. (*) See Section 3.6.4

ID	Summary	When (event)	I want (action)	So I can (benefit)	Comments
LMAPI03	Validate licence on package downloading	a package download is requested	its services' and functions' licences to be validated	only authorised services and functions are downloaded	Downloading is assumed to have the intention of reuse. The owner of the package can download it whether its services and functions have a 'public' or a 'private' licence
LMAPI04	Validate licence on service instantiation	a service instantiation is requested	the service's and it's functions' licences to be validated	only authorised services and functions are instantiated	Service instantiation is only authorised if ALL licences associated with it are valid
LMAPI05	Cancel licence	the BSS/GUI requests	the service's and it's functions' licences to be cancelled	the service instance cannot be used	

Trivial requirements, such as retrieving a list of licences a given end-user or owner has, are not listed in this table, but will be implemented.

Figure 3.2 shows the context for these requirements.

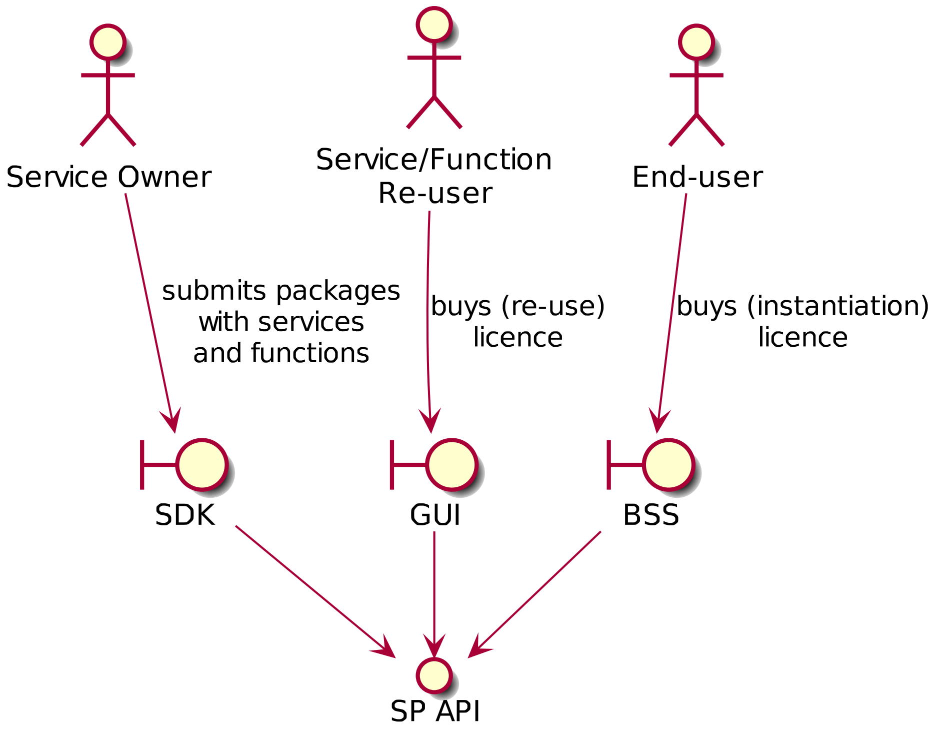


Figure 3.2: Context for Licence Manager

A **Service Owner** develops a service and functions, and on-boards them in the Service Platform, trough the **SDK**.

When that service and/or functions are 'private', any other user that wants to **re-use** the service or functions (**Service/Function Re-user**) has to buy a licence. This can be done by the **Service/Function Re-user** accessing the **GUI** and 'buying' a re-use licence from the **Service Owner**. The download of the licensed package through the **SDK** then will work smoothly.

On the other hand, when an **End-User** wants to instantiate a 'private' service, he/she must access the **BSS** and 'buy' a licence for that service. When the service instantiation is requested in the same **BSS**, the process runs smoothly.

This previous 'buy' step is not needed when the service is marked as 'public'.

Please note that, when the **Service Owner** tries to download his/her own packages, the **Licence Manager** is not even contacted (see the sequence diagram shown in Figure 3.3, with error conditions not shown for the sake of brevity).

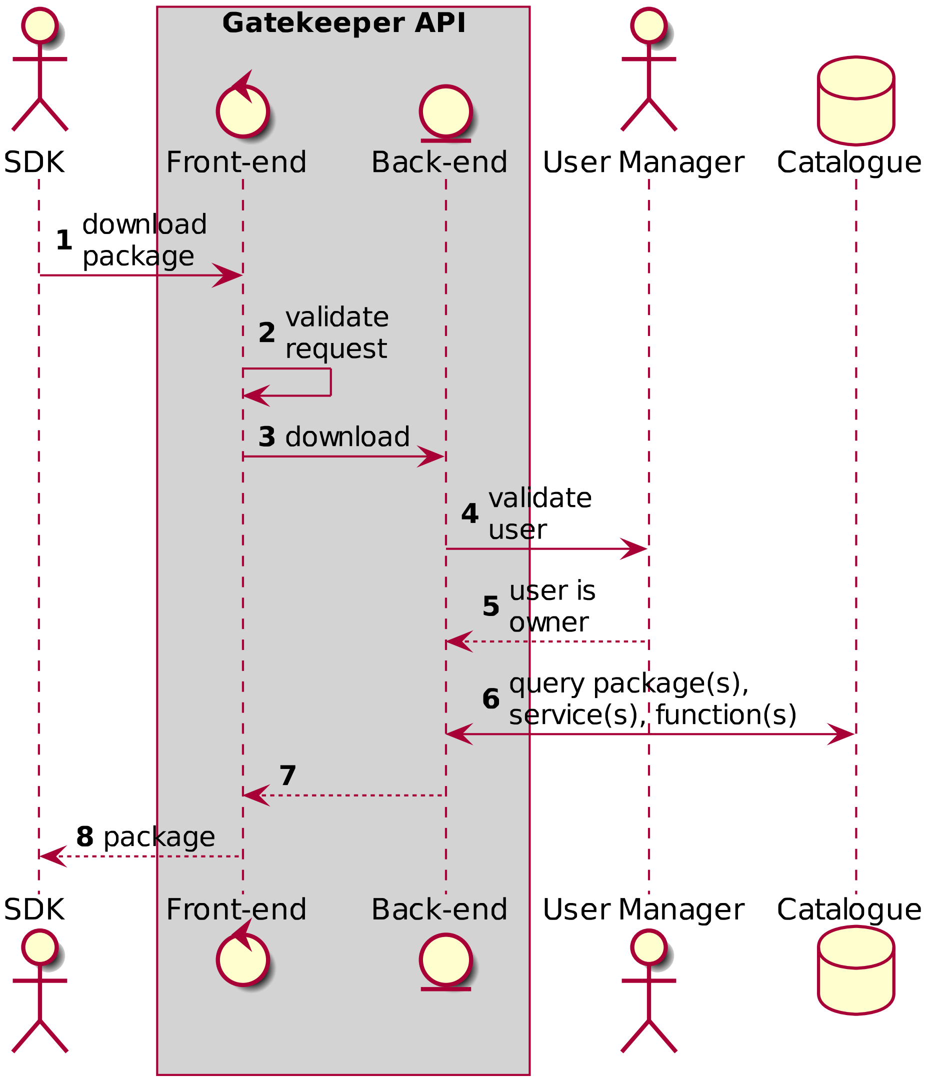


Figure 3.3: Sequence diagram for licence validation when the owner requests a download of his/her own packages

KPIs manager module

This component's requirements and the API are detailed in Section 3.4.1 and in Section 3.4.5, respectively. We need the module's design and implementation to mature a little bit more for that API to be made available to the KPI provider modules -- the Gatekeeper's API and all the others, both existing and new (see Figure 3.1).

3.1.3 Other improvements

Besides the changes mentioned above, we have also improved a number of features:

- **Name-spacing and versioning the API:** the Gatekeeper's API will start to be name-spaced and versioned with `/api/v2`;
- **Package storage/downloading:** packages are now stored side by side with their descriptors, which allows the download of the integral package for re-use, instead of rebuilding the package (and loosing all comments in the process);
- **Linking headers:** responses to requests that might have multiple records will include a `Link` header, which, together with the already supported `limit` (the maximum number of records returning in a response, defaulting to `10`) and `offset` (the number of the page within the result set, defaulting to `0`), allows easier navigation by the Gatekeeper's API through results.

3.2 User management module

This component is responsible of managing and controlling the permissions and authorizations of the platform users, allowing or denying the requested actions.

3.2.1 Requirements

Table 3.3 collects the set of user stories related to the User Management considering the Roles of the SONATA Platform.

Table 3.3: User Management User Stories

ID	Summary	As a (user)	I want (action)	So that (benefit)
1.1	Developer Registration	Developer	to be able to sign up to SONATA Service Platform	I can later publish and update services and check other public services offered.
1.2	Developer Login	Registered Developer	to be able to sign in to SONATA Service Platform	I can view the status of my services (running, suspended, etc), progress of pending service description validation, service package versions, etc.
1.3	Developer Profile Update	Registered Developer	to be able to update my profile settings	I can provide the most up-to-date information to the Service Platform.
1.4	Developer Authentication (while providing a package)	Registered Developer	to be able to authenticate my requests to the Service Platform	the Service Platform knows requests (deploy, monitoring) are mine.
1.5	Developer Authorization management (while providing a package)	Registered Developer	to be able to authorize other developers and users of the Service Platform	they can use my service and functions.
2.1	Customer Registration	Customer	to be able to sign up to SONATA Service Platform	I can later check the public services offered.
2.2	Customer Login	Registered Customer	to be able to sign in to SONATA Service Platform	I can view the status of my services' status (running, suspended, etc).
2.3	Customer Profile Update	Registered Customer	to be able to update my profile settings	I can provide the most up-to-date information to the Service Platform.
2.4	Customer Customer authentication	Registered Customer	to be able to authenticate my requests to the Service Platform	the Service Platform knows requests (instantiate, pause, resume, retire) are mine.
2.5	Customer Customer authorization	Registered Customer	to be able to use my authorized services	services can 'legally' use platform resources.

ID	Summary	As a (user)	I want (action)	So that (benefit)
3.1	Service Provider Registration	Service Provider	to be able to sign up to SONATA Service Platform	I can later monitoring and operate the services offered.
3.2	Service Provider Login	Registered Service Provider	to be able to sign in to SONATA Service Platform	I can view the status of my company/group services (running, suspended, etc).
3.3	Service Provider Profile Update	Registered Service Provider	to be able to update my profile settings	I can provide the most up-to-date information to the Service Platform.
3.4	Service Provider authentication	Registered Service Provider	to be able to authenticate my requests to the Service Platform	the Service Platform knows requests (instantiate, pause, resume, retire) are mine.
3.5	Service Provider authorization management (developers)	Registered Service Provider	to be able to authorize other developers of the Service Platform	they can modify/update the services and functions owned by my company/group.
3.6	Service Provider authorization management (users)	Registered Service Provider	to be able to authorize other users of the Service Platform	they can instantiate and use the services and functions owned by my company/group.
4.1	Platform Admin Login	Registered Sonata Platform Admin	to be able to sign in to SONATA Service Platform	I can later check the deployment process, monitoring, operate and retire the services.
4.2	Platform Admin Authentication	Registered Sonata Platform Admin	to be able to authenticate my requests to the Service Platform	the Service Platform knows requests (instantiate, pause, resume, retire) are mine.
4.3	Platform Admin Authorization management	Registered Sonata Platform Admin	to be able to unauthorized users/developers to the Service Platform	I can revoke permissions due to a continuous malfunction of the services deployed by a developer, or bad use of the services and resources by a developer/user, to preserve the integrity of the platform

3.2.2 User management module implementation

This section defines the User Management module implementation which is responsible of users and services authentication and authorization within the Service Platform. The User Management Module implementation is separated into the following components:

- **Adapter** (son-gtkusr), which is part of the Service Platform Gatekeeper.
- **Keycloak**, the Identity and Access Management open-source tool that acts as authentication and authorization server.
- **Database**, which is currently an optional component and might support additional features later. Core features use Keycloak's internal database.

These components communicate through secured RESTful interfaces as they follow the microservice architecture pattern inside the module. More information about its design and architecture can be found in section 3.3.4 of Deliverable D2.3 [5].

Implementation architecture follows the design in Deliverable D2.3. Figure 3.4 shows the first version for the User Management module architecture where the Adapter component enables an Access REST API and interacts with the Access Management and Identity Provider tool.

3.2.2.1 Authentication and authorisation adapter

The Adapter component provides a tight integration to the underlying Service Platform and the the authentication and authorization server (Keycloak).

Based on the centralized approach and the architecture chosen to implement a security layer on the Service Platform Gatekeeper, managing users/micro-services access requires an adapter or

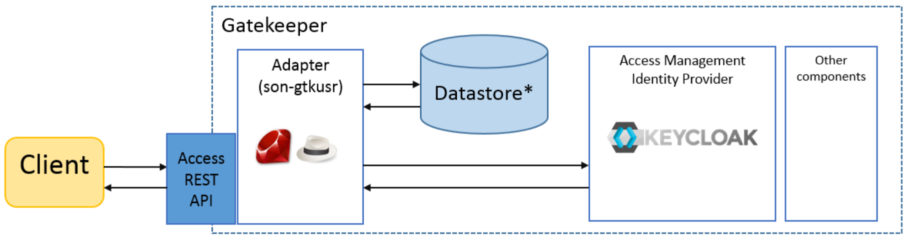


Figure 3.4: User Management module architecture and communication flows

client library to connect the authentication and authorization server and secure the platform. Given that the authentication and authorization server uses OpenID Connect [32] as a protocol for the resource Resource Provider, the Adapter is a required component that will be responsible of securely connect the User Management API on the Gatekeeper with the authorization and authentication server, forward registration, authentication and authorization messages.

In order to secure applications and services, Keycloak (authentication and authorization server) API is made in a way that requires an adapter (client) in each component to secure. To avoid such complexity and overhead of having a client in each SONATA component, the Adapter is the only component that will directly connect to the Keycloak from the SP. It is the main component that adds the authority entity to the platform through the Gatekeeper component using Keycloak API.

New features

The Adapter is a new component that comes into scene in SONATA Year 2 plan. It is under development, and while it is found in an early state, it support new features:

HTTPS The Adapter introduces a security layer adding HTTPS to the exposed APIs

User account registration The Adapter currently support a limited function to register end-users on the authentication and authorization server. This process automatically assigns a default User account with the provided user information. This feature will be improved adding roles and permissions associated to the user.

Centralized user authentication and authorization The exposed RESTful API is currently enabling registration and login features from a single point in the Gatekeeper. End-users of the platform can use the Adapter API to register to the Platform, provided of User accounts and be able to log-in and receive an Access Token that grants the use of services within the platform.

Planned features

There are plans to introduce new features during SONATA Year 2 development. Some of these features are already defined:

Centralized micro-service authentication and authorization The same way that the Adapter enables security features to users, a security layer will be introduced for micro-services within the SP. This feature will allow micro-services to register to the platform when installed, and retrieve an Access Token that will grant a identity and authorization to communicate other micro-services within the platform.

Social login (Git accounts) This feature enables logging-in to the platform using social networks accounts such GitHub, with the ease of not having to register users twice to the platform if an account is linked to a SONATA entity. No code or changes to SP micro-services is required. However it requires connecting Keycloak to the GitHub authorization server.

Adapter technical details

The Adapter is being implemented from scratch using Ruby Programming language and Sinatra Framework (as an internal component of the Gatekeeper). It will communicate with the Keycloak Identity and Access manager for most of authentication and authorization processes.
This component is a special micro-service that is granted of administrator rights over the Keycloak as it will be responsible of orchestrate requests from users and other icro-services.
It can be supported with an optional database to implement functionalities that are not bound to the Keycloak, e.g. Ownership database for access authorization to certain resources.

3.2.2.2 Authentication and authorisation server

From a candidates list of authentication and authorization tools to consider, Keycloak was chosen as open source tool to provide and support authentication and authorization functionalities to the Service Platform. Keycloak is an open source Identity and Access Management solution based on standard protocols and provides support for OpenID Connect (OIDC), OAuth 2.0 [21], and SAML [41] (more details can be found in Keycloak webpage at www.keycloak.org). OIDC has been chosen to secure platform's access and communication. It is responsible of granting secured access to end-users and micro-services. It includes a set of administrative UIs and exposes a RESTful API, which provides the necessary means to create permissions for protected resources and scopes, then associate those permissions with authorization policies or roles, enforcing authorization decisions when managing access to other micro-services or resources.

A resource provider in the SP (as can be the SP Catalogue), requires to rely on a security component that manages some kind of information to decide if an access should be granted to the protected resources. For RESTful-based resource providers, that information is usually obtained from a security token, usually sent as a bearer token on every request to the resource provider. Keycloak is the security component that produces security tokens called Access Token (JWT) based on the information of an user or service account, its roles, permissions, etc.

New features

Keycloak is a new component introduced to the SP that is part of the User Management module. While it offers a long list of features, User Management module is currently enabling required core functionalities of authentication and authorization:

Single-Sign On (SSO) Users authenticate with Keycloak rather than individual applications. This means that your applications don't have to deal with login forms, authenticating users, and storing users. Once logged-in to Keycloak, users don't have to login again to access a different application. This also applied to logout. Keycloak provides single-sign out, which means users only have to logout once to be logged-out of all applications that use Keycloak.

Security standard protocols Standard protocols such OpenID Connect and OAuth 2.0 are already enabled within Keycloak.

Administration console This feature allows SONATA administrators to easily manage Keycloak framework. It currently supports a workspace ('realm' for Keycloak) that connects with the Adapter and provides configuration settings to define Access Tokens parameters.

Planned features

During SONATA Year 2, it is planned to introduce new features to work on the authentication and authorization server:

- Dynamic creation and management of permissions
- Dynamic assignment and management of roles
- Role-based access control (RBAC)
- User-based access control (UBAC)
- Integration with SONATA BSS and/or GUI

Adapter and Keycloak integration

The Adapter (son-gtkusr) uses Keycloak RESTful API to securely manage authentication and authorization processes from the Service Platform.

The Adapter is responsible of performing 3 main functionalities in the User Management module. Figure 3.5 shows communication flows for these functionalities:

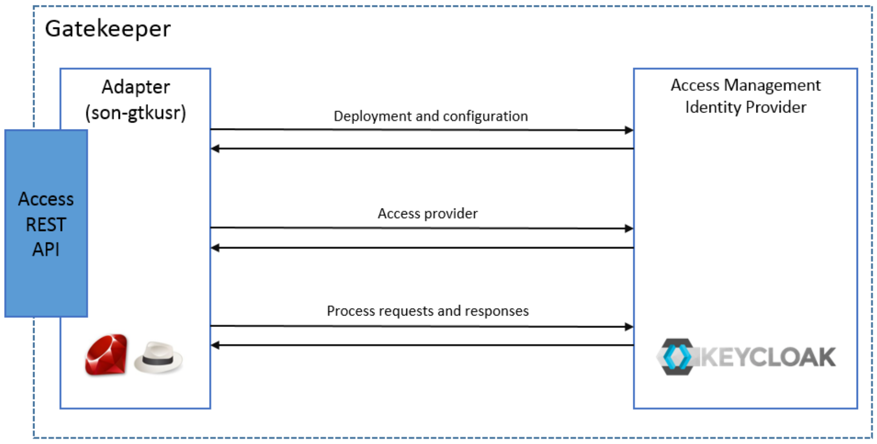

Gatekeeper
Adapter (son-gtkusr)
Access REST API
Access Management Identity Provider
Deployment and configuration
Access provider
Process requests and responses
KEYCLOAK

Figure 3.5: Adapter and Keycloak integration interfaces

Deployment and configuration When the Adapter (`son-gtkusr` as shown in Figure 3.5) is deployed (within the Gatekeeper) it must retrieve a configuration to obtain the endpoints available from the Keycloak and get administrator credentials as a Keycloak client. This is performed in two different processes, the client registration where the Adapter announces itself to the Keycloak, and client authentication, where it requests a special Access Token to establish secure communication to the Keycloak.
In order for an application or service to utilize Keycloak, it has to register a client in Keycloak. This process can be achieved manually by an administrator through the admin console, or can be dynamically done through administration REST endpoints. Clients can also register themselves through the Keycloak client registration service.

The Adapter (considered the client on the Gatekeeper) need a token in order to invoke the Keycloak authentication and authorization services. When the User Management module is deployed, it is assigned a keypair to be used when announcing to the Keycloak. This process is detailed in the section Section 2. To retrieve the Adapter configuration for a Keycloak client, it uses the assigned keypair credentials using HTTP basic authentication. This includes the following header in the request: `Authorization: basic BASE64(client-id + ':' + client-secret)`

The Keycloak returns a special Access Token that is only granted to the Adapter. This token uses JWT standard, and contains necessary information to grant administration rights to the Adapter on the Gatekeeper. Once the Access Token is provided, the Adapter proceeds to retrieve the client configuration accessing to `/realms/<realm>/clients-registrations/install/<client id>` and the OpenID configuration endpoint `/realms/{realm-name}/.well-known/openid-configuration`. It is most important endpoint to know is the well-known configuration endpoint and it lists endpoints and other configuration options relevant to the OpenID Connect implementation in Keycloak.

Keycloak also exposes a number of endpoints to manage user identity and authorization from the Adapter. These endpoints are just accessible by the Adapter, as they exposed only for internal communication between the Adapter and Keycloak.

Access provider The Adapter uses OpenID Connect with an “admin” Client Credentials as the authorisation grant obtained by its own Client ID/Secret keypair. Microservices inside and outside the SP will also be using Client Credentials grants (assigned to Services accounts), while end-users will be using Resource Owner Password Credentials grants (assigned User accounts). The Adapter is responsible of forwarding to the Keycloak all end-users or microservices requests made though a client.

Client registration and login workflow: Figure 3.6 shows the interaction between components in order to register a client and then log-in into the Service Platform to retrieve an Access Token.

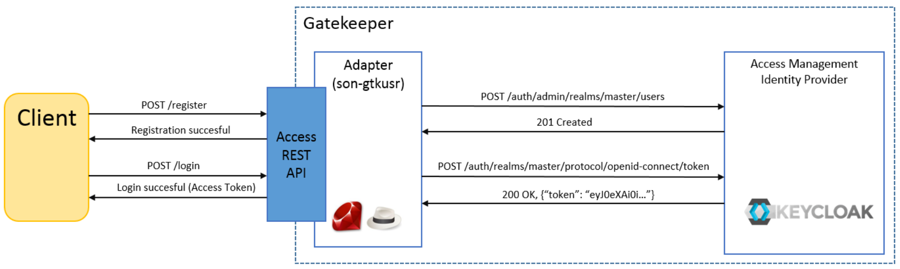

Gatekeeper
Client
POST /register
Registration succesful
POST /login
Login succesful (Access Token)
Access REST API
Adapter (son-gtkusr)
POST /auth/admin/realms/master/users
201 Created
POST /auth/realms/master/protocol/openid-connect/token
200 OK, {“token”: “eyJ0eXAi0i...”}
Access Management Identity Provider
KEYCLOAK

Figure 3.6: User Management module client registration and login workflow

When a end-user or microservice is registered and requests a login to the platform, Keycloak evaluates the log-in credentials forwarded by the Adapter and received from the end-user client or microservice in a two step process each request:

- It first authenticates the identity of the requester.
- It then checks whether authenticated identity is authorized to access.

If the login process is successful, the Adapter is responsible of returning a generated Access Token to the requester client.

Table 3.4 shows the Adapter exposed API endpoints through the Gatekeeper:

Table 3.4: Access Token verification endpoints.

Action	Description	HTTP method	Adapter Path	Keycloak path
Registration	Public endpoint that allows the registration of the end-user to the platform	POST	`/register`	`/auth/admin/realms /master/users`
Login	Public endpoint that allows end-users to obtain tokens by supplying credentials directly	POST	`/login`	`/realms/{realm-name} /protocol/openid-connect/token`
Userinfo	Secured endpoint that returns standard claims about the authenticated user	POST	`/userinfo`	`/realms/{realm-name} /protocol/openid-connect/userinfo`
Log-out	Secured endpoint that Logs out the authenticated user	POST	`/logout`	`/realms/{realm-name} /protocol/openid-connect/logout`

Process requests and responses The Adapter enables a REST API to accept operations from end-users/micro-services. However, this API will expose a public interface to allow registering to the SONATA Platform, and a secure interface for the authenticated/authorized processes as seen in 'Access provider' section.
However, the secured interfaces of the API will work with Access Tokens (considered bearer token type) included in each message header from end-users and micro-services clients. These messages need to be sent through the Gatekeeper API, which will use Adapter interfaces to authenticate and authorize incoming messages and access requests. This means that the Adapter will only be responsible of directly receiving those messages about registering and logging-in to the platform. Communication between micro-services, SP-external requests and SP-internal requests will use Gatekeeper dedicated API. Gatekeeper is then responsible of calling Adapter interfaces to authenticate and authorize these communications and requests. See more detailed information in section 3.3.4 from Deliverable D2.3 [5].

Table 3.5 present endpoints that are exposed by the Adapter to be accessed by the Gatekeeper in order to authenticate and authorize received tokens from any request:

Table 3.5: Access Token verification endpoints.

Action	Description	HTTP method	Adapter Path	Keycloak path
Token Authentication	Secured endpoint that performs authentication of the end-user	POST	`/auth`	`/realms/{realm} /protocols/openid-connect/token/introspect`
Token Authorization	Secured endpoint used to check roles and permissions for the provided identity	POST	`/authorize`	`/realms/{realm-name} /protocol/openid-connect/auth`

An Access Token is validated when the Gatekeeper receives `200 OK` responses from authentication and authorization endpoints. Then, the request can proceed. If the requester Access Token is not valid, the Gatekeeper returns the corresponding code and message.

3.2.3 Authentication and authorization external APIs

The Service Platform Gatekeeper exposes a RESTful API that enables external access to the Platform. However, this external access is designed to primarily be used by the SONATA SDK (see [6]).

On the **SDK** side, `son-access` is a new component introduced in the SONATA Year 2 plan that is responsible for establishing a secure communication between the SDK and the Service Platform. This component has been designed and implemented to provide a new single multipurpose catalogue entity for the whole SONATA ecosystem. The Service Platform Catalogue needs to be accessible from outside the SP through the external API that the SP Gatekeeper offers. Furthermore, son-access component on SDK is responsible of providing end-user credentials to the SP User Management module. This provides a secured connection between SDK end-users and the Service Platform, using their credentials to access the Platform and being able to submit and request stored package files and descriptors from the SP Catalogue. For more information about how external APIs are secured see 'Process requests and responses' section.

3.3 Licence management

The current section describes what SONATA has designed and implemented in order for the Service Platform to be able to fine-grain control which services and functions can be reused and/or instantiated. This is accomplished through a module named **Licence Manager**.

3.3.1 Requirements

Table 3.6 summarises the **job stories** [24] for the **Licence Manager** module.

Table 3.6: Licence Management User Stories

ID	Summary	When (event)	I want (action)	So I can (benefit)	Comments
1	Licence types	a licence is defined	to define its type	implement different behaviours for different types	Includes a notification of 'Create Licence Type' to the KPIs module
2	Licences	a (public or private) licence is created	to store its owner (a user), the service instance it relates to and its type	later validate it	Includes a notification of 'Create Licence' to the KPIs module
3	Private licences	a private licence is created	to store its owner URL	later validate the licence in that URL	The URL should have a REST interface and respond to POST (with the owner and the service instance). If the URL doesn't exist or respond, an error should be returned. Queries to the URL should have a timeout (like 5sec)
4	Validate 'public' licence	a request for the validation of a 'Public' licence is received	'valid' should be returned	re-use or instantiate the service without restrictions	Includes a notification of 'Validation of (public) Licence' to the KPIs module
5	Validate 'private' licence	a request for the validation of a 'Private' licence is received	the validation of the licence should be checked against the provided (external) URL	re-use or instantiation of the service depends on the results of calling that URL	The URL should have a REST interface and respond to GET (with the user and the service instance). If the validation fails, 'not valid' should be returned. Includes a notification of 'Validation of (private) Licence' to the KPIs module.
6	Cancel licence	a request for the cancellation of a licence is received	the licence should be marked as cancelled	return 'not valid' on a validation request	For 'Private' licences, a DELETE request should be made to the external URL. Includes a notification of 'Cancelation of Licence' to the KPIs module.

All jobs mentioned in Table 3.6 must include a call to the **KPIs Management** module, to register the event occurred.

3.3.2 Module architecture

This module is responsible for handling licensing of the various SONATA services. It has different types of licenses that can be associated to specific services and users. It also allows for multiple license status and thus checking their validity. This module will be part of the **Gatekeeper**, as shown in Figure 3.7 (Note: for the sake of simplicity, only the Licence Manager connection to consumed services is represented).

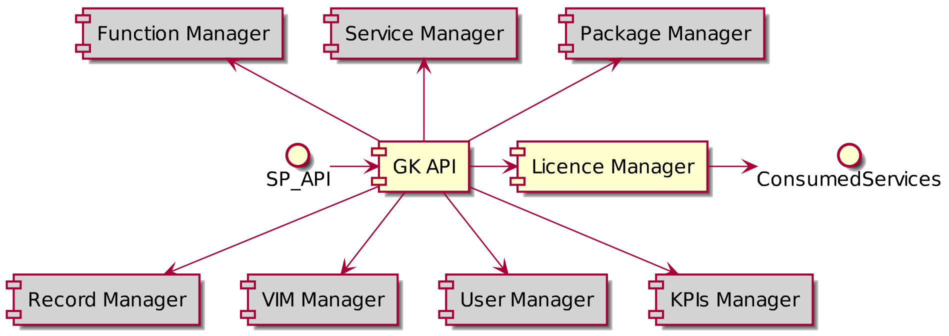


Figure 3.7: The Licence Manager module in the Gatekeeper's architecture

The external access to the features provided by the module is made through the **Gatekeeper's API**.

The module internally follows the generic architecture already mentioned in [7] (see Figure 3.8), with two components, a **front-end** to receive, validate and process requests (in this case coming from the Gatekeeper's API) and a **back-end**, to contact the other components.

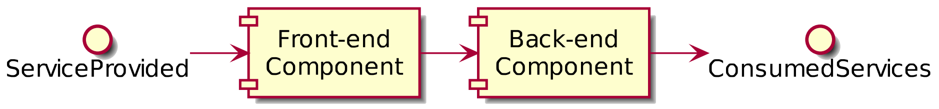


Figure 3.8: The Licence Manager module architecture

This module is implemented using **flask**, which is a micro-framework for the **python** programming language based on **Werkzeug**, **Jinja 2**.

3.3.3 Module interactions

This section briefly describes the interactions the **Gatekeeper API** has with the **Licence Manager**.

First, the designed method of creating the two licence types we are going to consider is explained. Then, the end-to-end interaction of creating a licence for a given service or function instance is described. Next, we describe the three different kinds of licence validation and finally we describe how a licence can be cancelled.

3.3.3.1 Licence type creation

For simplification, we are considering just two types of license: **public**, under which the service or function can be (re-)used ate will, and **private**, for which the developer has to provide a call-back

URL to validate each (re-)use.

We can simplify this by hard-coding the two types of licences.

3.3.3.2 Licence creation

A licence instance is created when an **End-User** (through the BSS -- see Section 3.6.4) buys an instance of a service that has a private licence.

This scenario executes according to the sequence diagram shown in Figure 3.9 (error conditions are not represented for brevity).

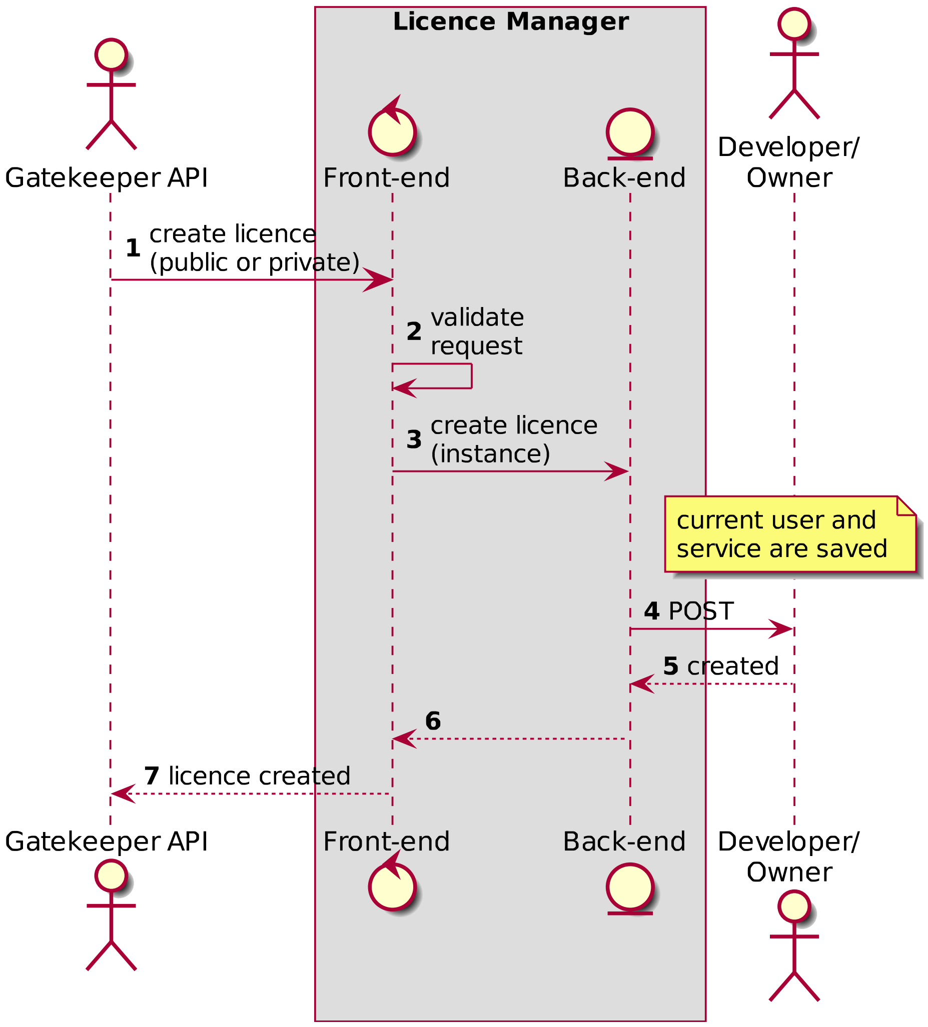


Figure 3.9: Sequence diagram of End-user licence instance creation

The steps in this sequence diagram are the following:

- **Step 1:** the **Gatekeeper API** asks the **Licence Manager** for the creation of a licence;
- **Steps 2** and **3:** the **Licence Manager front-end** validates the request and passes it to the **back-end**;
- **Step 4:** the **Licence Manager back-end** makes a POST to the Developer-provided URL with the **end user** (mandatory) and the **service** (only for instantiation licences);

- **Steps 5** to **7:** are the response from the POST;

3.3.3.3 Licence validation

Public licences do not need to be validated, but private licences do. Private licences are validated when another **Developer** than the owner wants to re-use the service or functions and when an **End-User** wants to instantiate a service.

The validation done when a service or function is downloaded by other Developer than the owner of that service or function follows the sequence diagram shown in Figure 3.10 (error conditions are not shown for the sake of brevity).

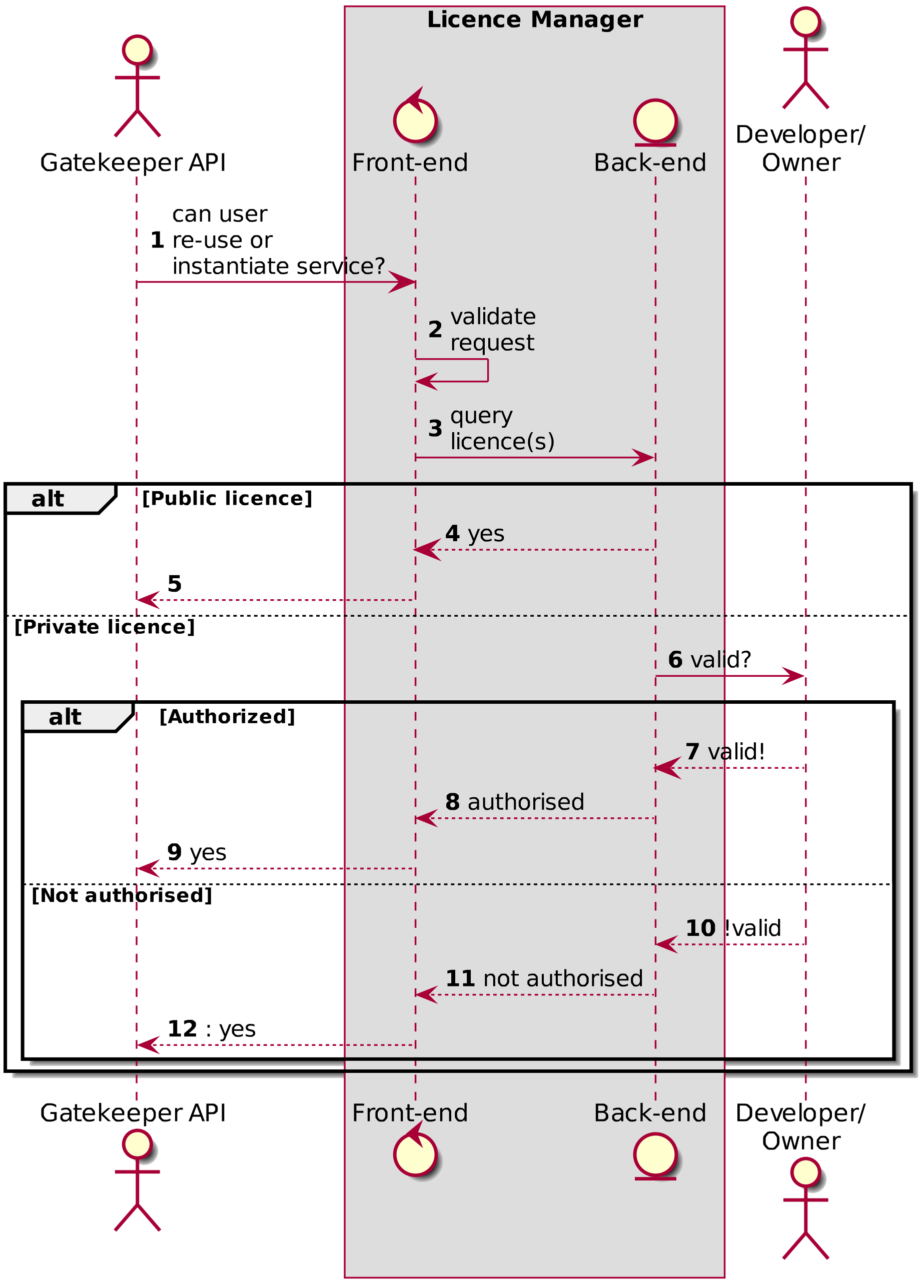


Figure 3.10: Sequence diagram of licence validation

The steps of this sequence diagram mean the following:

- **Step 1:** the **Gatekeeper API** asks the **Licence Manager** if a given user can re-use or instantiate a given service;
- **Steps 2** and **3:** the **Licence Manager's front-end** validates the request and passes it to the **back-end**;
- **Steps 4** and **5:** for 'Public' services, the answer is immediate, and is returned to the **Gatekeeper API**;
- **Step 6:** for 'Private' licences, the Service Owner's URL is called;
- **Steps 7** to **9:** if the user has a licence, a positive answer is returned;
- **Steps 10** to **12:** otherwise, no re-use or instantiation should be allowed.

3.3.3.4 Licence cancellation

Cancelling a licence is a trivial operation, so we are not showing its sequence diagram.

3.3.4 Module API

Table 3.7 shows the proposed API for this module.

Table 3.7: Proposed API for the Licence Management module

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	type	GET	`/types`	• status: active, inactive • type UUID: public, private	query	No
create	type	POST	`/types`	• description: Description of the license. • duration: Duration in days that license is valid for. • status: active, inactive	body	Yes
query	licence	GET	`/licences`	• licence UUID • user UUID • service UUID • type UUID • status: active, inactive	query	No

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	licence	GET	`/licences/<id>`	• licence UUID • user UUID • service UUID • type UUID • status: active, inactive	URL	Yes
create	licence	POST	`/licences`	• description: Description of the license • user UUID • service instance UUID • type UUID • status: active (default), cancelled	body	Yes
update	licence	PUT	`/licences/<id>`	• status=inactive	body	Yes

3.4 KPIs management

This section describes the module designed to manage key metrics and key performance indicators (KPIs) of the gatekeeper, which can provide recommendations for action to the system.

3.4.1 Requirements

Table 3.8 summarises the **job stories** [24] for the **KPI Manager** module. There are two types of entities: the events or information units, the records related with the activities performed in the Gatekeeper (API invocation, service instantiation, user log in, etc), and the key performance indicators (KPIs), the metrics generated from the events.

Table 3.8: KPI Management User Stories

ID	Summary	When (event)	I want (action)	So I can (benefit)
1	Event types	an event is defined	to define its type	manage the information related with gatekeeper entities like API invocations, services and function instantiations, users, etc
2	Event creation	a gatekeeper action is performed	to register the information under a specific event type	group the events by categories (API invocations, services, functions, users, vims, etc)
3	Event update	a not immediate gatekeeper action is completed	to update the information of the specific event	reflect the final status and the last update date (completion date)

ID	Summary	When (event)	I want (action)	So I can (benefit)
4	Create KPI	a new key performance indicator is needed	to define the involved events, the measurement parameter, the count/average/filtering operation	collect the specific performance information about an entity or group of entities
5	Get KPI	it's needed to know an specific KPI value	retrieve the information from the KPI Manager	get the specific information about an entity or group of entities which will provide recommendations about future actions
6	Cancel KPI	a KPI is not representative or obsolete	to disable it	centralize the managing efforts in active and valid KPIs

3.4.2 Module interactions

The KPI Manager will collect all relevant events produced in the Gatekeeper, so every time that an action is requested to the gatekeeper, it will create a new event through the KPI Manager API; when the action is completed (successfully or error) the event status will be updated containing creation and last update dates.

The Figure 3.11 shows the interaction between the Gatekeeper API and the new KPI Manager module.

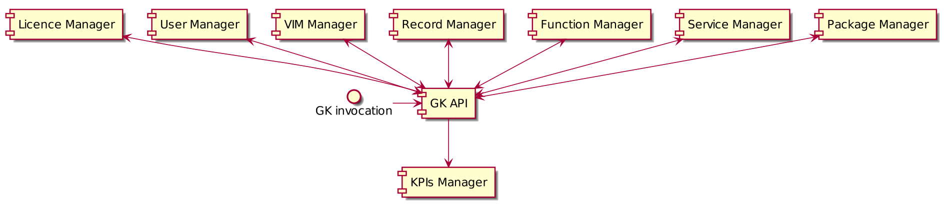


Figure 3.11: Gatekeeper and KPI Manager interaction

3.4.2.1 Events related processes

Figure 3.12 shows the create, update and query event processes that will be performed by the KPI Manager module.

3.4.2.2 Processes related with the KPIs

Figure 3.13 shows the create, query and delete KPI processes that will be performed by the KPI Manager module.

3.4.3 Gatekeeper's KPIs

Table 3.9 shows the set of KPIs that will be exposed by the Gatekeeper module

Table 3.9: Gatekeeper KPIs

Group	KPI	Description
API	Total calls	Total API calls received in the Gatekeeper
API	Top calls	Top 10/20/... of API calls
API	Total wrong calls	Total error responses
API	Top error	Top 10/20/... error types in responses
USER	Total users	Total users registered in the platform
USER	Active users	Total users logged in the platform

Group	KPI	Description
USER	Total developers	Total developers registered in the platform
USER	Total customers	Total customers registered in the platform
USER	Active customers	Number of customers with at least one instance running
USER	Top customer	Top 10/20/... of customers with the highest amount of service instances running
USER	Top developers	Most active developers
USER	Total service providers	Total service providers registered in the platform
USER	Top service providers	Service providers with the highest amount of services deployed
SERVICE	Total Available services	Total service on-boarded in the platform available for instantiation
SERVICE	Total services	Total active service instantiations
SERVICE	Top Services	Top 10/20/... of services instantiated
SERVICE	Instantiation time	Average time for instantiation
SERVICE	Total error instances	Total instances with ERROR status
FUNCTION	Total available functions	Total functions on-boarded in the platform available for instantiation
FUNCTION	Total functions	Total active function instantiations
FUNCTION	Top functions	Top 10/20/... of functions instantiated
VIM	Total VIMs	Total VIMs included in the platform
VIM	Top VIM	Top 10/20/... of vims used

3.4.4 Module architecture

As shown in Figure 3.14 this Gatekeeper module is composed by three components, a **storage** module to receive and collect the information about the events published by the gatekeeper, a **query** module that generates the KPIs performing queries over the stored information, and a **representation** module, a dashboard that shows the KPIs.

To obtain the KPIs is needed to store different information events produced by the Gatekeeper. It is proposed the use of an open source solution who allow the storage, query and visualization of the generated metrics. There are several options that are being considered: Parse Server [34], Piwik [36], Kong [25], Datadog [9], Influxdb [22] + Graphite [13], Kibana + Elasticsearch [23], Prometheus [38], etc.

For example, **Parse Server** is an open source tool that permits the storage of the data in a backend as a service (BaaS) model. Figure 3.15 shows the Parse dashboard, a component who permits the representation of the collected information, allowing filtering and querying data.

3.4.5 Module API

Table 3.10 shows the proposed API for this module.

Table 3.10: Proposed API for the KPI Management module

Action	Entity	Http method	Path	Parameters	Parameter location	Required
create	event	POST	`/events/api`		body	
				`id`		`Yes`
				`error`		`No`
query	event	GET	`/events/api`		URL	
				`id`		`No`

Action	Entity	Http method	Path	Parameters	Parameter location	Required
update	event	PUT	`/events/api`		body	
				`id`		`Yes`
				`parameter to change`		`Yes`
				`new value`		`Yes`
create	event	POST	`/events/users`		body	
				`id`		`Yes`
				`type`		
				`status`		
				`last action`		
query	event	GET	`/events/users`		URL	
				`id`		`No`
update	event	PUT	`/events/users`		body	
				`id`		`Yes`
				`parameter to change`		`Yes`
				`new value`		`Yes`
create	event	POST	`/events/services`		body	
				`id`		`Yes`
				`owner`		`Yes`
				`status`		`Yes`
				`error`		`No`
query	event	GET	`/events/services`		URL	
				`id`		`No`
update	event	PUT	`/events/services`		body	
				`id`		`Yes`
				`parameter to change`		`Yes`
				`new value`		`Yes`
create	event	POST	`/events/functions`		body	
				`id`		`Yes`
				`owner`		`Yes`
				`status`		`Yes`
				`error`		`No`
query	event	GET	`/events/functions`		URL	
				`id`		`No`
update	event	PUT	`/events/functions`		body	
				`id`		`Yes`
				`parameter to change`		`Yes`
				`new value`		`Yes`
create	event	POST	`/events/vims`		body	
				`id`		`Yes`
				`owner`		
				`status`		

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	event	GET	`/events/vims`		URL	
				`id`		`No`
update	event	PUT	`/events/vims`		body	

				`id`
				`parameter to change`
				`new value`
						`Yes`
						`Yes`
						`Yes`

create	KPI	POST	`/kpis/api`		body	

				`id`
				`condition: name, total account, period, error code, etc`
						`Yes`
						`Yes`

query	KPI	GET	`/kpis/apis`		URL	
				`id`		`No`
delete	KPI	DELETE	`/kpis/apis`		URL	
				`id`		`Yes`
create	KPI	POST	`/kpis/users`		body	

				`id`
				`condition: name, total account, period, error code, etc`
						`Yes`
						`Yes`

query	KPI	GET	`/kpis/users`		URL	
				`id`		`No`
delete	KPI	DELETE	`/kpis/users`		URL	
				`id`		`Yes`
create	KPI	POST	`/kpis/services`		body	

				`id`
				`condition: total account, period, error code, etc`
						`Yes`
						`Yes`

query	KPI	GET	`/kpis/services`		URL	
				`id`		`No`
delete	KPI	DELETE	`/kpis/services`		URL	
				`id`		`Yes`
create	KPI	POST	`/kpis/functions`		body	

				`id`
				`condition: name, total account, period, error code, etc`
						`Yes`
						`Yes`

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	KPI	GET	`/kpis/functions`		URL	
				`id`		`No`
delete	KPI	DELETE	`/kpis/functions`		URL	
				`id`		`Yes`
create	KPI	POST	`/kpis/vims`		body	
				`id`		`Yes`
				`condition: name, total account, period, error code, etc`		`Yes`
query	KPI	GET	`/kpis/vims`		URL	
				`id`		`No`
delete	KPI	DELETE	`/kpis/vims`		URL	
				`id`		`Yes`

3.5 Graphical User Interface

This section describes and specifies the additional functionalities of the SONATA Graphical User Interface to be developed during the second year of the project, including those that have already been implemented, but have not been included in Deliverable 4.1 [7].

3.5.1 Extended GUI views

The extended SONATA GUI now has views related to API calls provided by SONATA components, such as Repository, Monitoring Framework, etc. In particular, the following views have already been implemented and integrated to the SONATA GUI.

3.5.1.1 Dashboard view

As shown in Figure 3.16, the Dashboard view presents the high-level information of the nodes comprising the SONATA Service Platform.

However, more information is available in this page (Figure 3.17), such as the resources allocated and their state of performance.

Moreover, at the same page, the information on the number of active VMs (or VNFs) is available (see Figure 3.18), while information with respect to each VM can also be displayed (Figure 3.19).

3.5.1.2 Alerting view

Another extension of GUI view from those described in D4.1 is related to the integration of the Alerting mechanism provided by the Monitoring Manager API. In this view, the user is able to change the refresh rate and also be informed about the rule that has been triggered, the unique id and name of the function (VM) and its state, as shown in Figure 3.20.

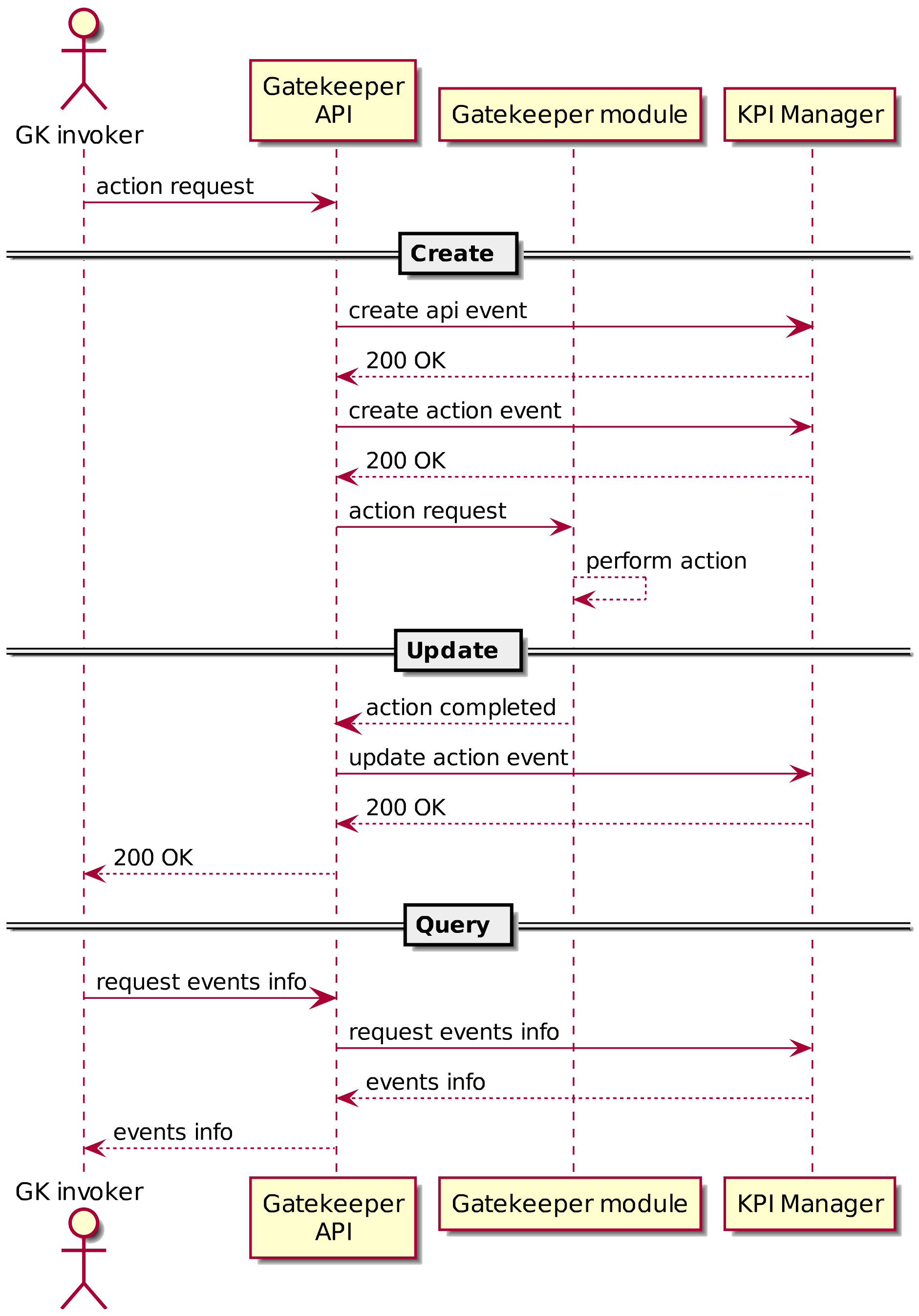


Figure 3.12: Processes related with the Events

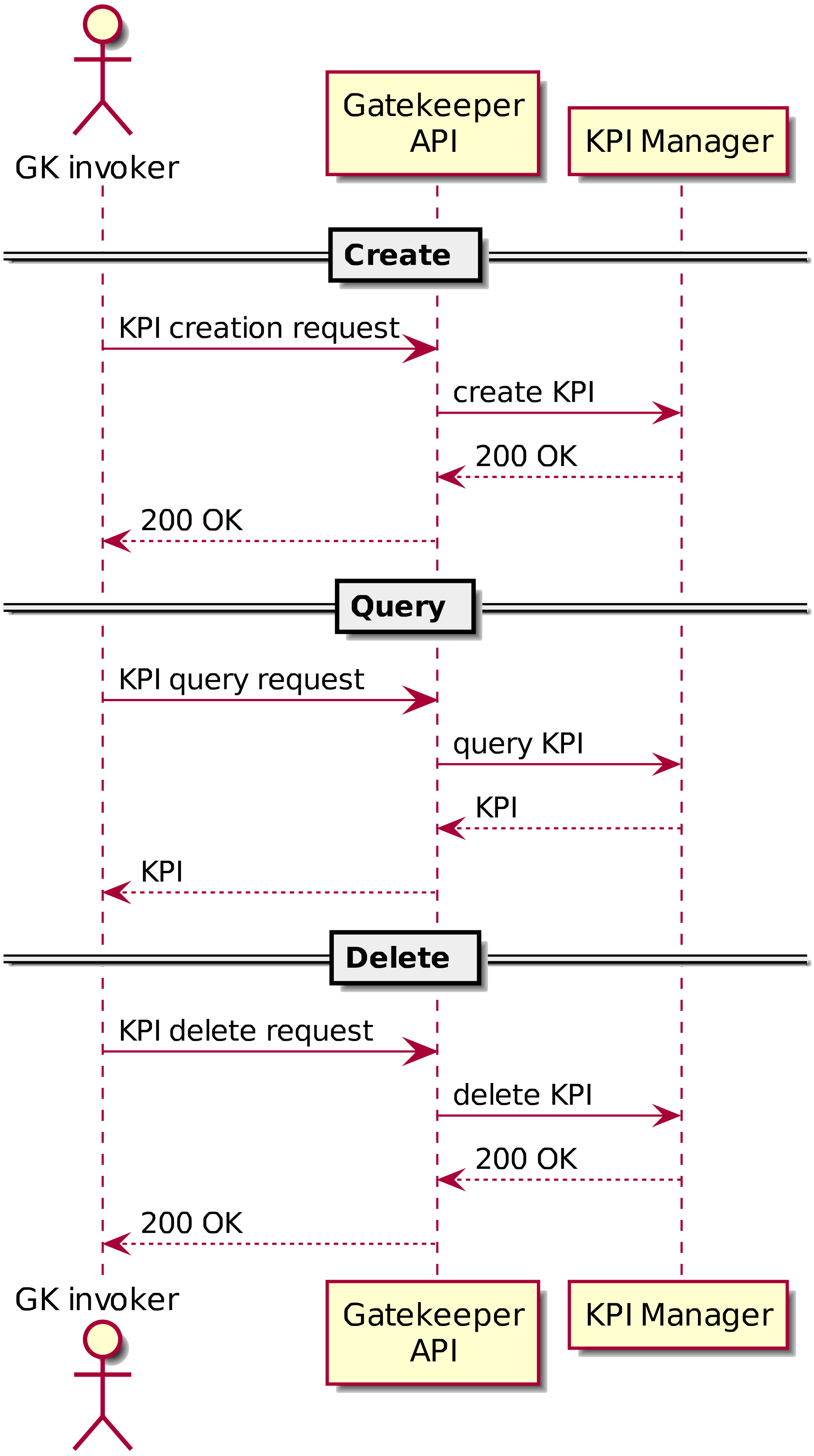


Figure 3.13: Processes related with the KPIs

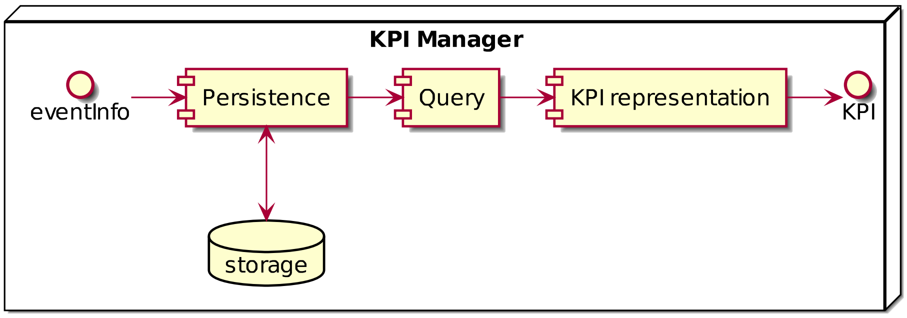
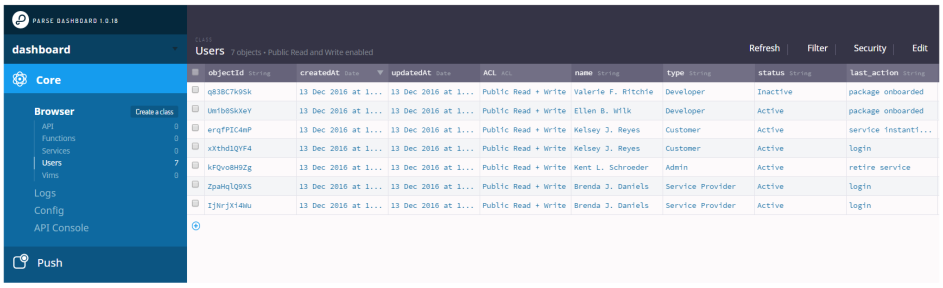


Figure 3.14: The KPI Manager module architecture



Users 7 objects • Public Read and Write enabled

objectId String	createdAt Date	updatedAt Date	ACL ACL	name String	type String	status String	last_action String
q83BC7k9Sk	13 Dec 2016 at 1...	13 Dec 2016 at 1...	Public Read + Write	Valerie F. Ritchie	Developer	Inactive	package onboarded
Umib0SkXeY	13 Dec 2016 at 1...	13 Dec 2016 at 1...	Public Read + Write	Ellen B. Wilk	Developer	Active	package onboarded
erqfPIC4mP	13 Dec 2016 at 1...	13 Dec 2016 at 1...	Public Read + Write	Kelsey J. Reyes	Customer	Active	service instanti...
xXthd1QYF4	13 Dec 2016 at 1...	13 Dec 2016 at 1...	Public Read + Write	Kelsey J. Reyes	Customer	Active	login
kFQvo8H9Zg	13 Dec 2016 at 1...	13 Dec 2016 at 1...	Public Read + Write	Kent L. Schroeder	Admin	Active	retire service
ZpaHqlQ9XS	13 Dec 2016 at 1...	13 Dec 2016 at 1...	Public Read + Write	Brenda J. Daniels	Service Provider	Active	login
IjNrjXi4Wu	13 Dec 2016 at 1...	13 Dec 2016 at 1...	Public Read + Write	Brenda J. Daniels	Service Provider	Active	login

Figure 3.15: Parse Dashboard Component

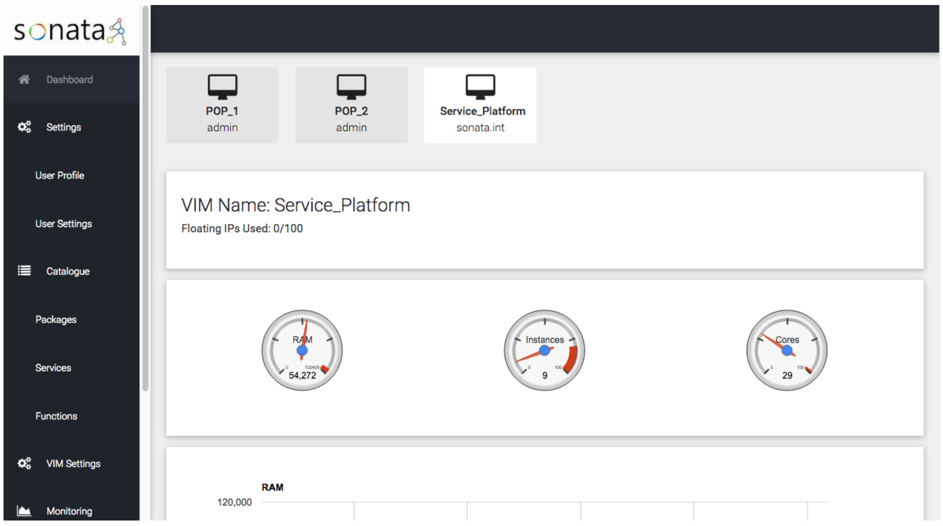


Figure 3.16: SONATA SP high-level overview

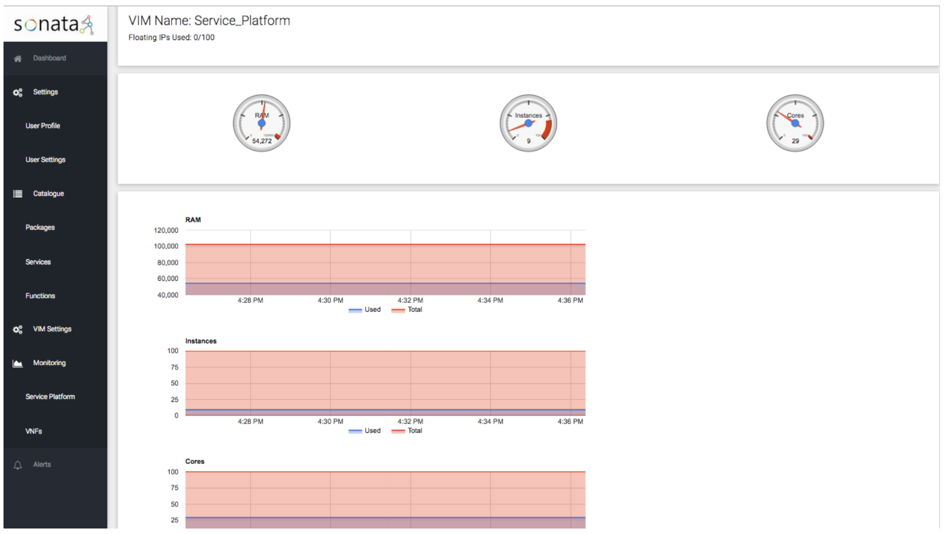
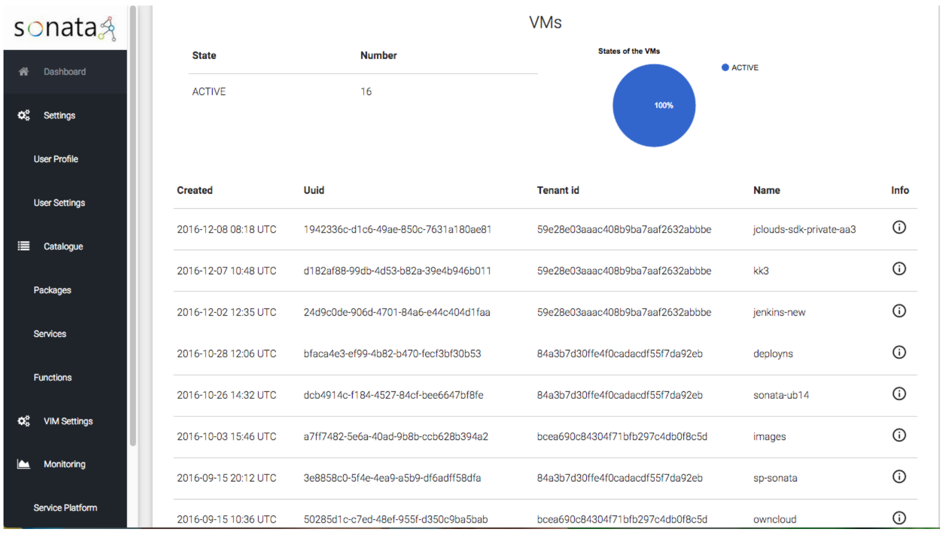


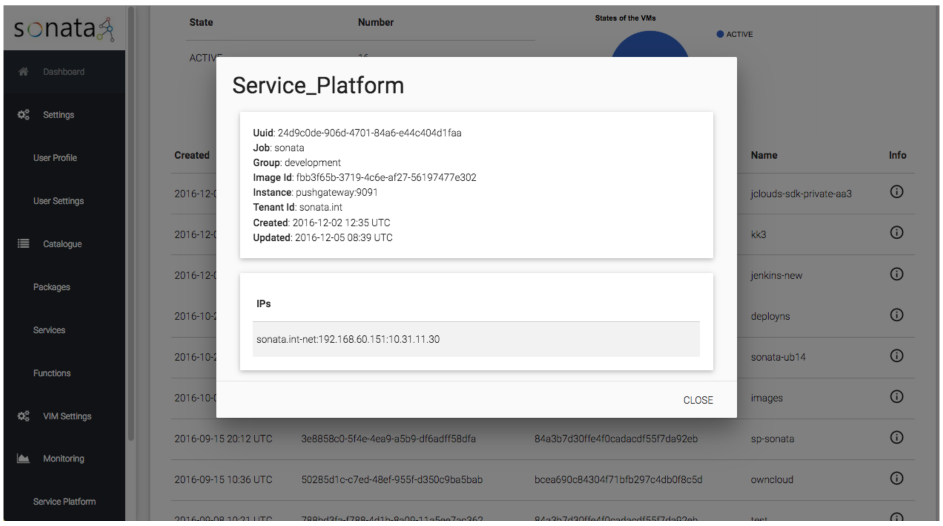
Figure 3.17: SONATA SP resources allocated



State	Number
ACTIVE	16

Created	Uuid	Tenant id	Name	Info
2016-12-08 08:18 UTC	1942336c-d1c6-49ae-850c-7631a180ae81	59e28e03aaac408b9ba7aaf2632abbbe	jclouds-sdk-private-aa3	ⓘ
2016-12-07 10:48 UTC	d182af88-99db-4d53-b82a-39e4b946b011	59e28e03aaac408b9ba7aaf2632abbbe	kk3	ⓘ
2016-12-02 12:35 UTC	24d9c0de-906d-4701-84a6-e44c404d1faa	59e28e03aaac408b9ba7aaf2632abbbe	jenkins-new	ⓘ
2016-10-28 12:06 UTC	bfaca4e3-ef99-4b82-b470-fecf3bf30b53	84a3b7d30ffe4f0cadacdf55f7da92eb	deployns	ⓘ
2016-10-26 14:32 UTC	dcb4914c-f184-4527-84cf-bee6647bf8fe	84a3b7d30ffe4f0cadacdf55f7da92eb	sonata-ub14	ⓘ
2016-10-03 15:46 UTC	a7ff7482-5e6a-40ad-9b8b-ccb628b394a2	bcea690c84304f71bfb297c4db0f8c5d	images	ⓘ
2016-09-15 20:12 UTC	3e8858c0-5f4e-4ea9-a5b9-df6adff58dfa	84a3b7d30ffe4f0cadacdf55f7da92eb	sp-sonata	ⓘ
2016-09-15 10:36 UTC	50285d1c-c7ed-48ef-955f-d350c9ba5bab	bcea690c84304f71bfb297c4db0f8c5d	owncloud	ⓘ

Figure 3.18: SONATA SP VMs/Containers information



Service_Platform

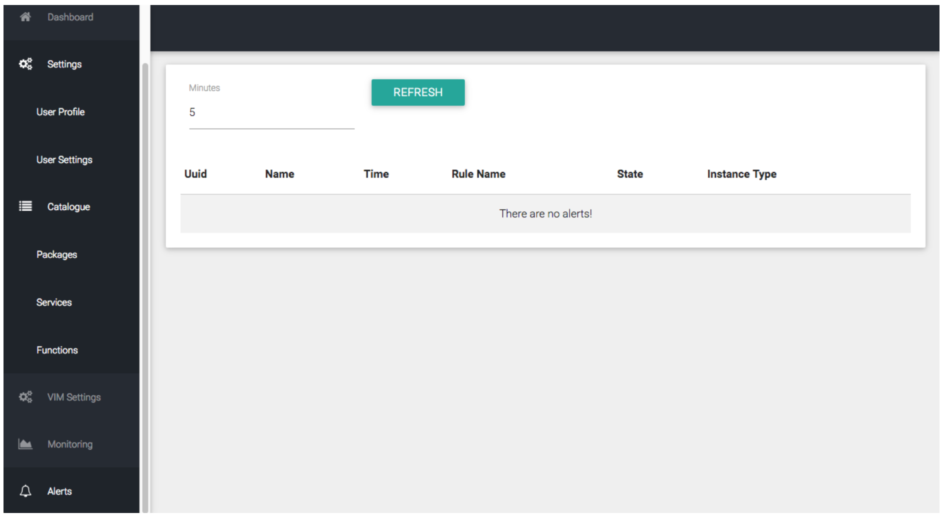
Uuid: 24d9c0de-906d-4701-84a6-e44c404d1faa
Job: sonata
Group: development
Image Id: fbb3f65b-3719-4c6e-af27-56197477e302
Instance: pushgateway:9091
Tenant Id: sonata.int
Created: 2016-12-02 12:35 UTC
Updated: 2016-12-05 08:39 UTC

IPs

sonata.int-net:192.168.60.151:10.31.11.30

Created			Name	Info
2016-09-15 20:12 UTC	3e8858c0-5f4e-4ea9-a5b9-df6adff58dfa	84a3b7d30ffe4f0cadacdf55f7da92eb	sp-sonata	ⓘ
2016-09-15 10:36 UTC	50285d1c-c7ed-48ef-955f-d350c9ba5bab	bcea690c84304f71bfb297c4db0f8c5d	owncloud	ⓘ

Figure 3.19: SONATA SP VM/Container extended info



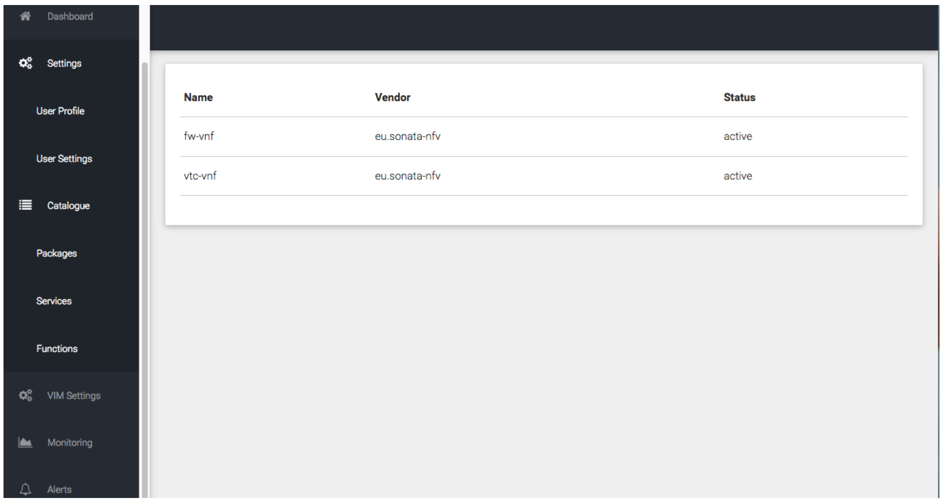
Minutes
5

Uuid	Name	Time	Rule Name	State	Instance Type
There are no alerts!					

Figure 3.20: Alerting view

3.5.1.3 Functions view

Finally, once the API call of the SONATA Repository component became available, it has been integrated to the SONATA SP GUI, providing information on the VNFs comprising a network service, such as the example shown in Figure 3.21.



Name	Vendor	Status
fw-vnf	eu.sonata-nfv	active
vtc-vnf	eu.sonata-nfv	active

Figure 3.21: Functions view

3.5.2 Integration with AuthN/AuthZ mechanism

Until now, the authentication mechanism supported by the SONATA GK was based on the GitHub API. However, a stronger security perspective has been decided to be followed during the second year and thus GUI component will comply with the requirements set by the SONATA Service Platform as a whole.

3.5.3 Improve user friendliness

During the second year, the SONATA GUI will support dynamic modifications on the views. For example, the user will be able to change the time duration of the displayed data, and add/remove charts or displayed metrics.

Moreover, the GUI views will be extended whenever new API calls become available from the SONATA components.

3.6 Business Support Systems

This section shows the improvements and upgrades to be included in the BSS module related with the new functionalities to develop in the Gatekeeper. These new improvements are described in the following sub-sections.

3.6.1 Https

The Service Platform will start having a secured API, so the BSS will modify all the Gatekeeper's API invocations to use their HTTPS equivalents.

3.6.2 Pagination links

With the inclusion of link headers [39] by the gatekeeper, the API invoked by the BSS can return a set of ready-made links so the BSS won't have to construct the pagination links. Here is an example of a Link header, grabbed from GitHub's documentation:

```
Link:
<https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
<https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```

In addition to next and last, another interesting field is the total number of available results. The Gatekeeper API can use a custom HTTP header like `X-Total-Count` to send this field.

BSS will adapt its pagination method to include the link header.

3.6.3 User management

Figure 3.22 describes the user management that will be developed in the BSS module.

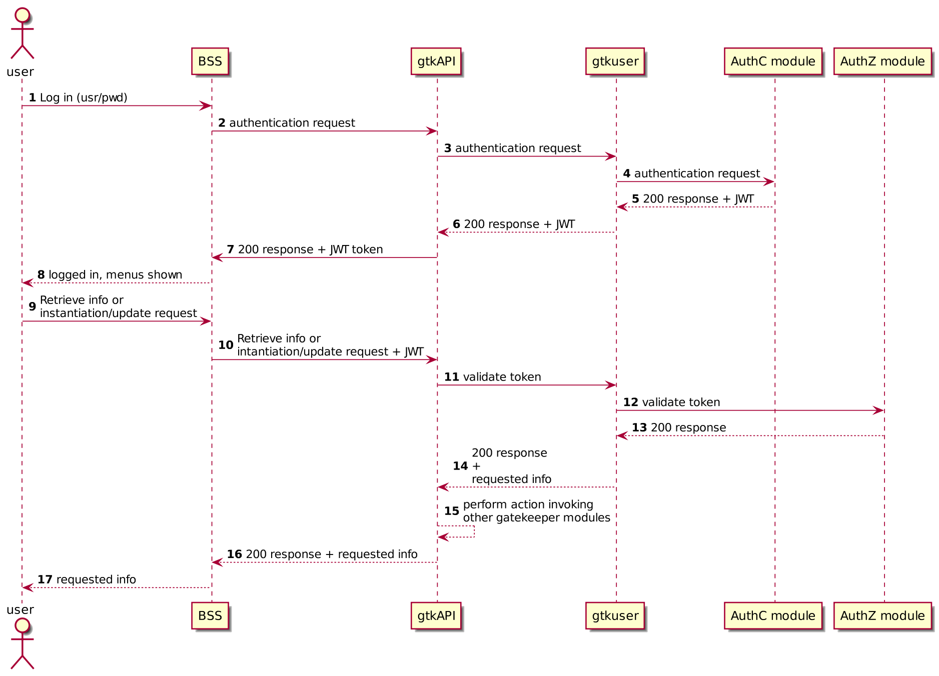


Figure 3.22: BSS User Management

3.6.4 License management

The BSS module will update the service instantiation logic to include the license management performed in the gatekeeper. This logic is affected in two different points:

- The New service license request: the BSS user requests a private service license, which allows him/her to instantiate that service;
- The Private Service Instantiation request: depending on the needed license by private services, the instantiation process can fails with "no license" case.

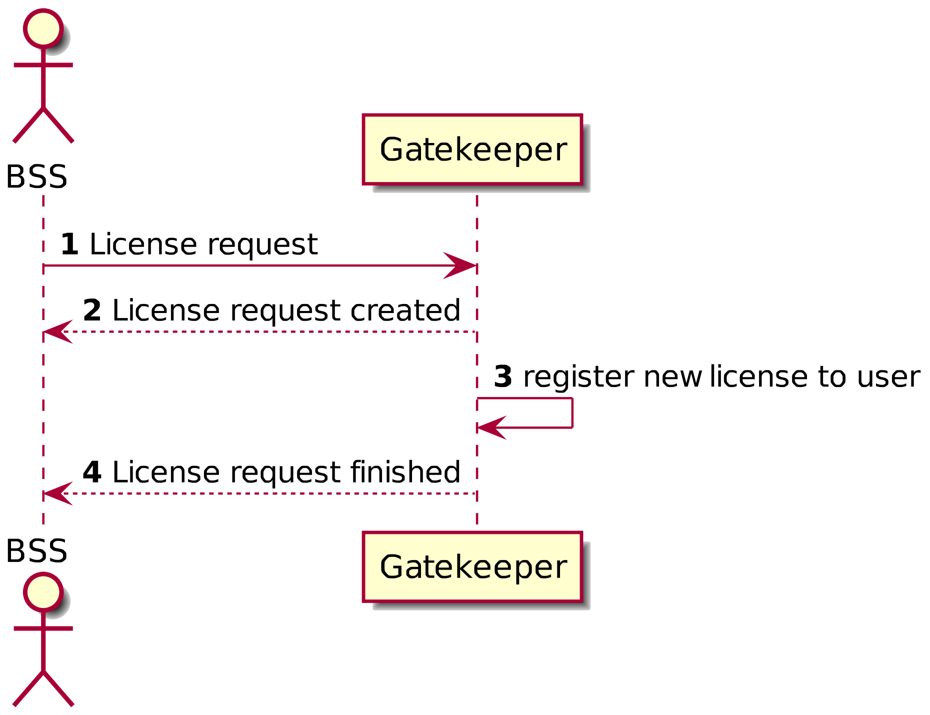

BSS
Gatekeeper
1 License request
2 License request created
3 register new license to user
4 License request finished
BSS
Gatekeeper

Figure 3.23: Sequence diagram of licence management for a service license request

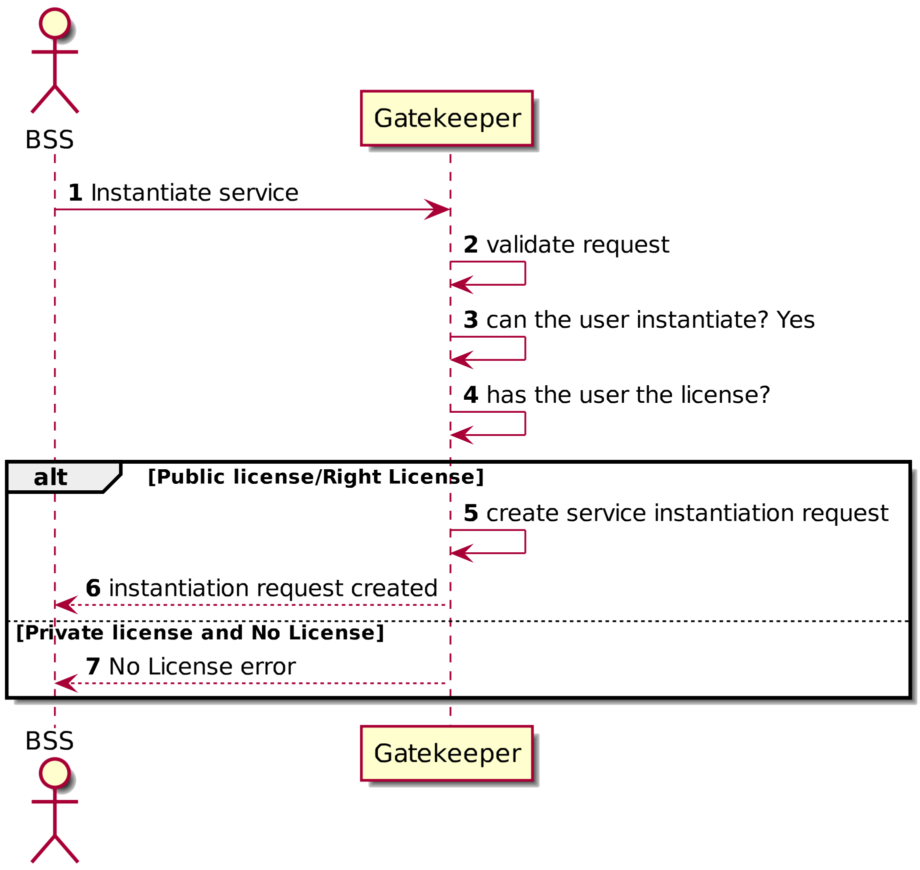

BSS
Gatekeeper
1 Instantiate service
2 validate request
3 can the user instantiate? Yes
4 has the user the license?
alt [Public license/Right License]
5 create service instantiation request
6 instantiation request created
[Private license and No License]
7 No License error
BSS
Gatekeeper

Figure 3.24: Sequence diagram of licence management for a service instantiation

Figure 3.23 shows the steps that will be followed to request a new service license.
Figure 3.24 shows the steps that will be followed to request a new service instantiation.

4 Catalogues and repositories

Catalogues and Repositories share their implementation in a single component (`son-catalogue-repos`) in the Service Platform architecture, which includes RESTful APIs and the databases. For more detailed information about the Catalogues and Repositories, see section 5.1 from Deliverable D4.1 [7].

4.1 Catalogues

For SONATA Year 2 plan on the Service Platform, there are some features under development for the Service Platform (SP) Catalogues and other features that will arrive later. This component is now responsible for the storage for NSDs, VNFDs, PDs and SONATA packages (files called son-packages). Access to the SP Catalogues is only available through the SP Gatekeeper component.

4.1.1 New features

This section shows new features recently added or that are under development right now.

4.1.1.1 Meta-data and data levels

The catalogues are now introducing a new meta-data level for each stored descriptor. Before this feature, when a descriptor was stored in the catalogue, meta-data was added to the descriptor in the same level as the descriptor data. This mixed fields such `created_at` and `id` with the rest of descriptor data. This feature now separates meta-data from data into two different levels. The new structure now includes next fields:

```
  {
id: '...meta-data...',
md5: '...meta-data...',
status: '...meta-data...',
created_at: '...meta-data...',
updated_at: '...meta-data...',
signature: '...meta-data...',
descriptor_data: {...data...}
}
```

The `descriptor_data` field, replaced by descriptor type name 'NSD', 'VNFD' or 'PD', exclusively contains descriptor data. Figure 4.1 summarizes this new dual-level structure for each catalogue document entry.

4.1.1.2 SONATA Packages storage

A new SP Catalogues feature supports storing small size binary files in the MongoDB database. SONATA packages a.k.a son-packages contain all bundled files received by the Gatekeeper, such descriptors and the meta information. The API for son-packages currently supports GET, POST and DELETE methods using son-packages UUIDs as arguments.

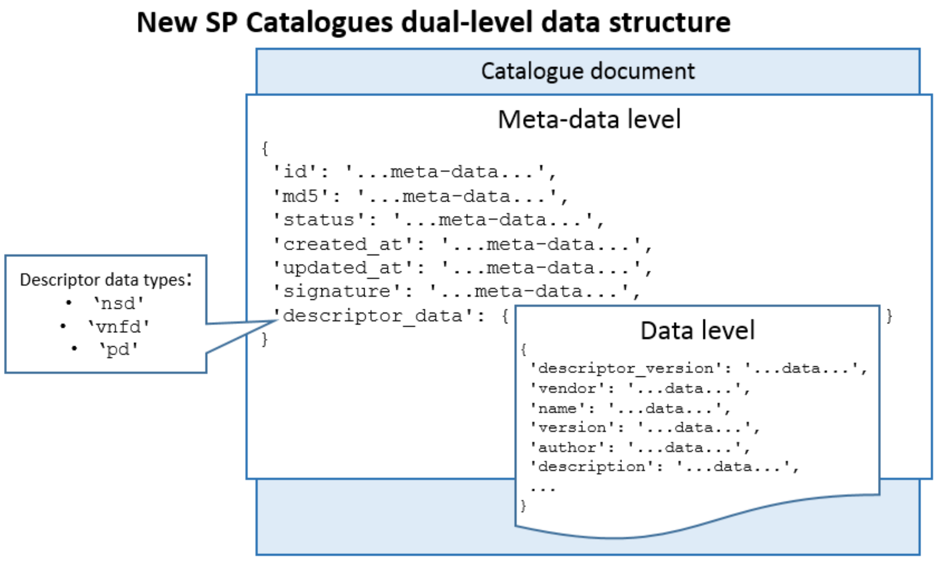

New SP Catalogues dual-level data structure
Catalogue document
Meta-data level
{
'id': '...meta-data...',
'md5': '...meta-data...',
'status': '...meta-data...',
'created_at': '...meta-data...',
'updated_at': '...meta-data...',
'signature': '...meta-data...',
'descriptor_data': {
}
}
Descriptor data types:
• 'nsd'
• 'vnfd'
• 'pd'
Data level
{
'descriptor_version': '...data...',
'vendor': '...data...',
'name': '...data...',
'version': '...data...',
'author': '...data...',
'description': '...data...',
...
}

Figure 4.1: New catalogues dual level data structure

4.1.2 Planned features

There are some new features that are planned to be introduced in the catalogues during the SONATA Year 2 phase.

- **HTTPS:** Add a security layer to the APIs;
- **Descriptor dependencies mapping:** Create a mapping for descriptors that bind them when dependencies are found between NSDs, VNFDs and PDs. This mapping can be helpful to add consistency on the SP Catalogues;
- **DELETE method updates:** Currently the DELETE does not process any dependency check and directly removes a descriptor from the catalogues. This method is planned to be improved in order to check dependencies before doing any change on a descriptor stored in the catalogues. When all checks are passed, then a descriptor can be safely removed from the catalogues.

4.1.3 Authentication and authorisation

The communication between SONATA Service Platform microservices and SP Catalogues will be authenticated in order to increase the security within Service Platform.

As the other platform components, a JWT, which grants secure access, will be generated in the User Management module, and validated by the SP Catalogues API. This process is detailed in Section 2. Microservices affected with this implementation are the **Gatekeeper** and the **Catalogues**.

4.2 Repositories

This section describes and specifies the additional functionalities of the SONATA Repositories to be developed during the second year of the project.

4.2.1 Authentication and authorisation

The communication between SONATA Service Platform microservices and Repositories will be authenticated in order to increase the security within Service Platform

A JWT will be generated in the Identity Management and validated by the repository API. The process is detailed in the section Section 2. The microservices affected with this implementation are the **Service Lifecycle Management** (SLM), the **Function Lifecycle Management** (FLM), the **Gatekeeper** (its **Records** API) and the **Repositories**.

5 MANO Framework

This section covers the improvements implemented on the **MANO Framework** and the further planned work. The enchantments include the addition of the Function Lifecycle Manager (see Section 5.1) and the Service and Function Specific Managers (see Section 5.2) managed by the Specific Managers Registry (see Section 5.3).

5.1 FLM and SLM

In this section, we describe the new features for the Service Lifecycle Manager (SLM) and Function Lifecycle Manager (FLM) that will be introduced during the second year of the SONATA project. They can be divided in to three categories

1. Updating the existing workflows to the new SP architecture.
2. Adding new workflows that will allow the SP to pause, resume and terminate a service.
3. Converting the SLM into a task manager.
4. Workflow Engine Based S/FLM.

Each of the above categories is further discussed in this section, as well as the multiple interfaces between different components. The exact APIs for all these interfaces can be found in Section 8.

5.1.1 Updating the existing workflows to the new SP architecture and APIs

The MANO framework will be extended with a new plugin the Function Lifecycle Manager (FLM). The FLM will be responsible for managing the lifecycle of Virtual Network Functions (VNF). Managing the lifecycle of a VNF includes starting/pausing/migrating/terminating it, configuring it according to the information made available in its descriptor or according to instructions from a Function Specific Manager (FSM), and addressing monitoring information that should trigger an update of the VNF. The functional split between the Service Lifecycle Manager (SLM) and the FLM is the level on which they operate. The SLM works on the level of the service where the FLM works on the level of the VNF, unaware of any information that concerns the service.

First, this new plugin should be included in the existing MANO framework workflows. Figure 5.1 contains a Message Sequence Chart (MSC) that describes how the MANO Framework will deploy a network service in which the FLM is included. The service deploy request ends with the SLM instructing the Monitoring Manager to start monitoring and the Repositories to store the records of the running instances. These interactions have not changed and can be found in deliverable D4.1 (see [7]). Nevertheless, some of the steps in Figure 5.1 deserve some explanation

- **Steps 2** and **3:** The SLM requests a view of the available resources. This information should include the available PoPs, usage rate and availability in these PoPs, how these PoPs are connected, and any other information that is required for placing the service;

- **Step 4:** The SLM calculates how the service will be mapped on the infrastructure, based on the resource information it received. This task can be delegated to a Service Specific Manager (SSM);
- **Steps 5** and **6:** The SLM informs the Infrastructure Adaptor (IA) which PoPs will be used by the service;
- **Step 7:** The SLM informs instructs the FLM to deploy a new FLM;
- **Step 8:** The FLM interacts with a Function Specific Manager (FSM) to customize the configuration of the VNF that will be deployed;
- **Steps 9** and **10:** The FLM instructs the IA to deploy the VNF, on a specific PoP;
- **Steps 12** and **13:** When all the VNFs are deployed, the SLM informs the IA how the chain them together;
- **Steps 14** and **15:** The SLM instruct the IA to configure the WAN, after which traffics starts flowing trough the service and it can be considered as running;

In **Step 4** and **Step 8**, the SLM/FLM can interact with a SSM/FSM to perform the task. A detailed explanation on how this works can be found in Section 5.2.

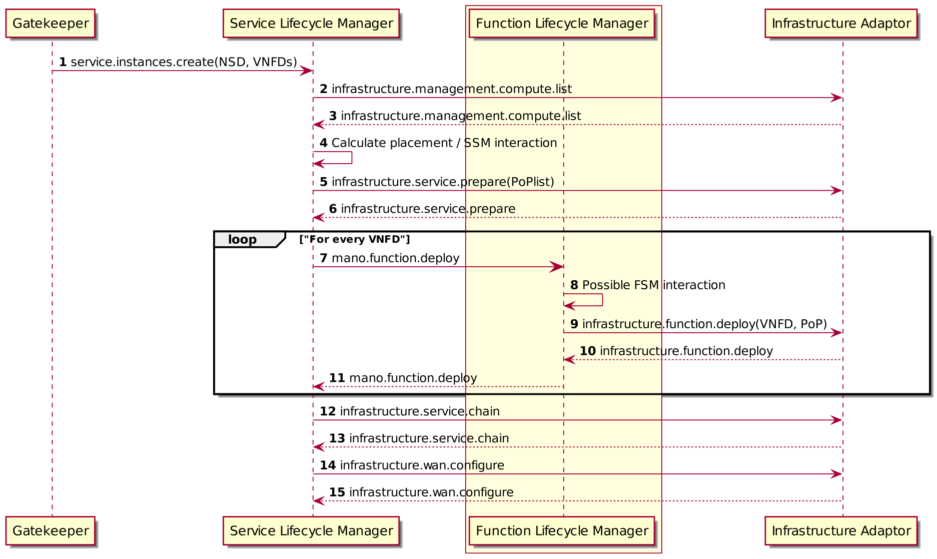

Gatekeeper
Service Lifecycle Manager
Function Lifecycle Manager
Infrastructure Adaptor
1 service.instances.create(NSD, VNFDs)
2 infrastructure.management.compute.list
3 infrastructure.management.compute.list
4 Calculate placement / SSM interaction
5 infrastructure.service.prepare(PoPlist)
6 infrastructure.service.prepare
loop ["For every VNFD"]
7 mano.function.deploy
8 Possible FSM interaction
9 infrastructure.function.deploy(VNFD, PoP)
10 infrastructure.function.deploy
11 mano.function.deploy
12 infrastructure.service.chain
13 infrastructure.service.chain
14 infrastructure.wan.configure
15 infrastructure.wan.configure
Gatekeeper
Service Lifecycle Manager
Function Lifecycle Manager
Infrastructure Adaptor

Figure 5.1: Deploying a service

5.1.2 Introducing new workflows to the SP and MANO framework

Next to updating the already existing workflows, we plan to add new workflows to the MANO framework. This allows us to support a wider variety of requests we might receive from the customer (through the Gatekeeper) or from the infrastructure (through the Monitoring Manager). In this section, we address the four workflows that will be added during the second year of SONATA: pausing, resuming, terminating and updating a service.

5.1.2.1 Pausing a service

A customer might request to pause a running service. Figure 5.2 describes how the SP components will satisfy such a request. The SLM first requests the IA to "deconfigure" the WAN, after which no more traffic is flowing through the service. In a next step, the SLM requests the IA to pause different VNFs in the service. As no more traffic is flowing through these VNFs, the pausing can be performed safely.

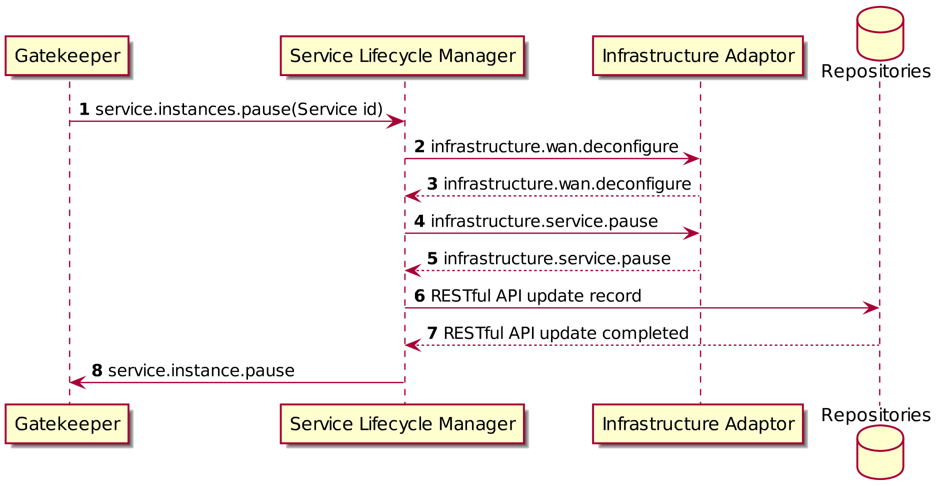

Gatekeeper
Service Lifecycle Manager
Infrastructure Adaptor
Repositories
1 service.instances.pause(Service id)
2 infrastructure.wan.deconfigure
3 infrastructure.wan.deconfigure
4 infrastructure.service.pause
5 infrastructure.service.pause
6 RESTful API update record
7 RESTful API update completed
8 service.instance.pause

Figure 5.2: Pausing a service

5.1.2.2 Resuming a service

A customer might request to resume a previously paused service. Figure 5.3 describes how the SP components will satisfy such a request. The SLM first requests the IA to restart the different VNFs in the service. This can be done safely, as no traffic will be flowing through them until the WAN is configured. In a next step, the SLM requests the IA to reconfigure the WAN, after which traffic starts flowing through the VNFs and the service can be considered as running.

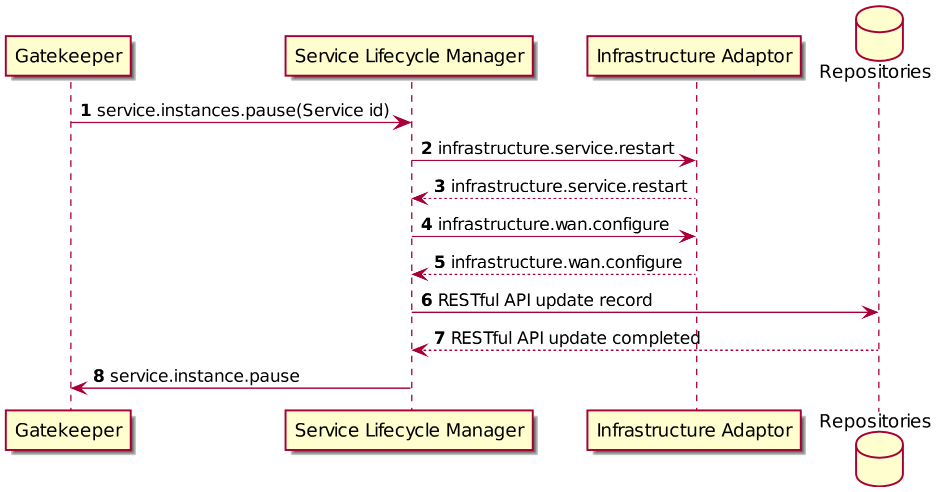

Gatekeeper
Service Lifecycle Manager
Infrastructure Adaptor
Repositories
1 service.instances.pause(Service id)
2 infrastructure.service.restart
3 infrastructure.service.restart
4 infrastructure.wan.configure
5 infrastructure.wan.configure
6 RESTful API update record
7 RESTful API update completed
8 service.instance.pause

Figure 5.3: Resume a service

5.1.2.3 Terminating a service

A customer might request to terminate a running service. Figure 5.4 describes how the SP components will satisfy such a request. The SLM first gives all the instructions as if it were pausing the service. Once the service is paused, and no more traffic is flowing through it, it instructs the IA to terminate all the VNFs in the service.

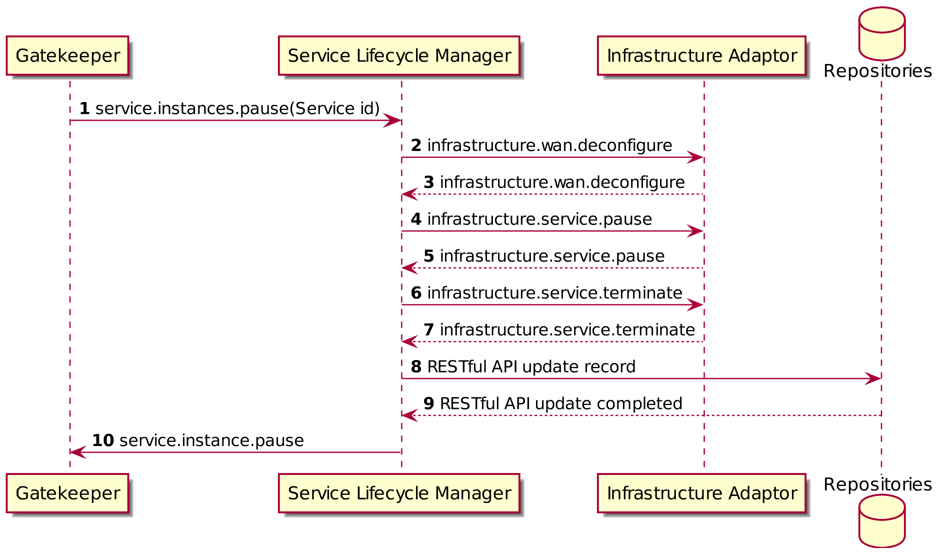

Gatekeeper
Service Lifecycle Manager
Infrastructure Adaptor
Repositories
1 service.instances.pause(Service id)
2 infrastructure.wan.deconfigure
3 infrastructure.wan.deconfigure
4 infrastructure.service.pause
5 infrastructure.service.pause
6 infrastructure.service.terminate
7 infrastructure.service.terminate
8 RESTful API update record
9 RESTful API update completed
10 service.instance.pause
Gatekeeper
Service Lifecycle Manager
Infrastructure Adaptor
Repositories

Figure 5.4: Terminate a service

5.1.2.4 Updating a service

It should be possible to update a service. During the first year review of SONATA, we demonstrated how an SSM of a running service can be updated. During the second year of SONATA, we will identify other customer update processes (e.g. replacing one VNF in the service by a newer version) that are required to support the different needs of telecom operator customers, and design and implement a workflow for them.

Updating a service can also be the result of a monitoring alarm, if the SLM decides that this is the appropriate response to the monitoring alarm. For example, a monitoring trigger that indicates a shortage of resources for a VNF should be answered with a workflow that either provides more resources to the VNF, or migrates it to a different PoP. Therefore, during the second year of SONATA, we will also design and implement those SLM workflows that respond to the possible monitoring alarms.

5.1.3 Converting the SLM into a task manager

The SLM will function as a task manager and will be developed completely by the SONATA consortium. All the individual tasks that the SLM needs to support will be implemented, and the different workflows as detailed earlier in this section will be obtained by chaining such tasks together. To add flexibility to the MANO framework, these tasks will be overwritable by SSMs. For example, when a monitoring trigger indicates a shortage of resources for a VNF, a SSM might overwrite the default reaction of the SLM. Instead of instructing the FLM to request more resources for this VNF, it might force the SLM to first contact a placement SSM to check if a relocation of this VNF might be possible at a low cost. Depending on the cost of relocating the VNF, it can instruct the SLM to consider a relocation or to instruct the addition of resources.

5.1.4 Workflow Engine Based S/FLM

To showcase the plugable aspect of the MANO framework, a different version of either the SLM or the FLM will be developed during the second year of SONATA. This version will be based on an open sourced workflow engine - Mistral (see [3]). This engine will be wrapped as a SONATA plugin. This will enable defining specific lifecycle operations using mistral workflows, enabling services to have unique management without developing a complete S/FSM.

5.2 Specific Managers Infrastructure

As mentioned in previous deliverables and also earlier in this deliverable, Function- and Service-Specific Managers (FSMs and SSMs) are management programs included in the service package that are used exclusively for managing their corresponding functions or services, e.g., for calculating service owner's desired placement or scaling. While the internal structure and design of the specific managers are not limited or defined, we define clear interfaces for the communication between FSMs/SSMs and the rest of the service platform.

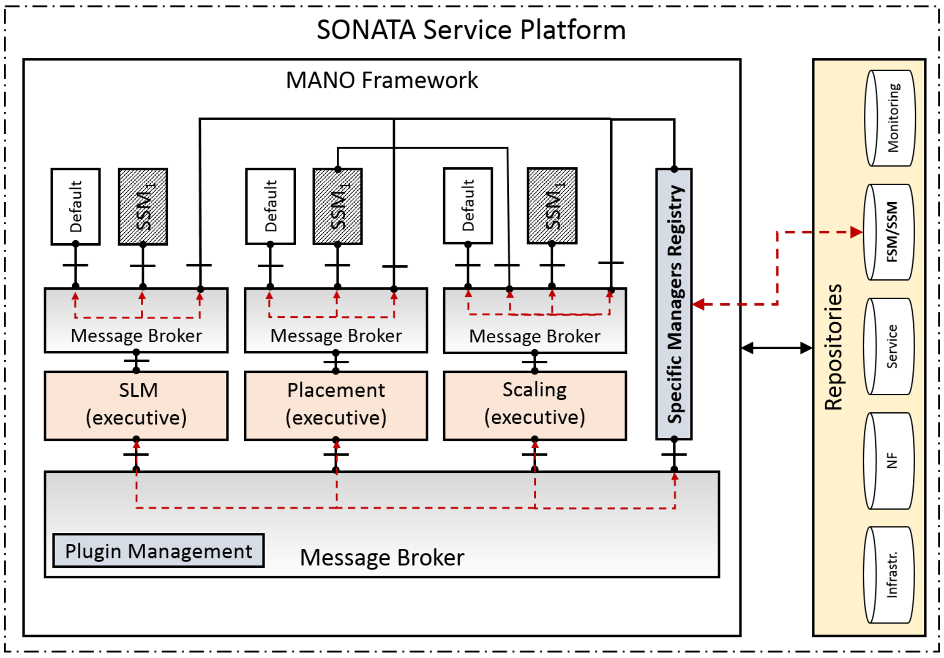


Figure 5.5: Specific managers infrastructure in SONATA MANO framework

As shown in Figure 5.5, FSMs and SSMs are connected to executive plugins, which are customizable MANO plugins responsible for specific tasks within the MANO framework, e.g., lifecycle management, placement, scaling, etc. Executive plugins are managed by the Plugin Management component similar to other MANO plugins.

The communication between FSMs/SSMs and their corresponding executive plugin(s) takes place through dedicated message brokers, isolated from the main message broker in the MANO framework. Depending on the functionality, each executive plugin exposes certain interfaces and developers can design FSMs/SSMs that can exchange information with the executive plugin using this pre-defined interface. The FSM/SSM requests the information it needs from the executive plugin. The executive plugin obtains the required information from the rest of the service platform, transforms it to the right level of abstraction for the target FSM/SSM (e.g., by removing sensitive information not related to the service) and provides it to the FSM/SSM. Similarly, the results produced by the FSM/SSM are checked and validated by the corresponding executive plugin. The decisions and requirements provided by the specific managers are considered for operation if they are not conflicting with service platform's global policies or requirements of other services and functions.

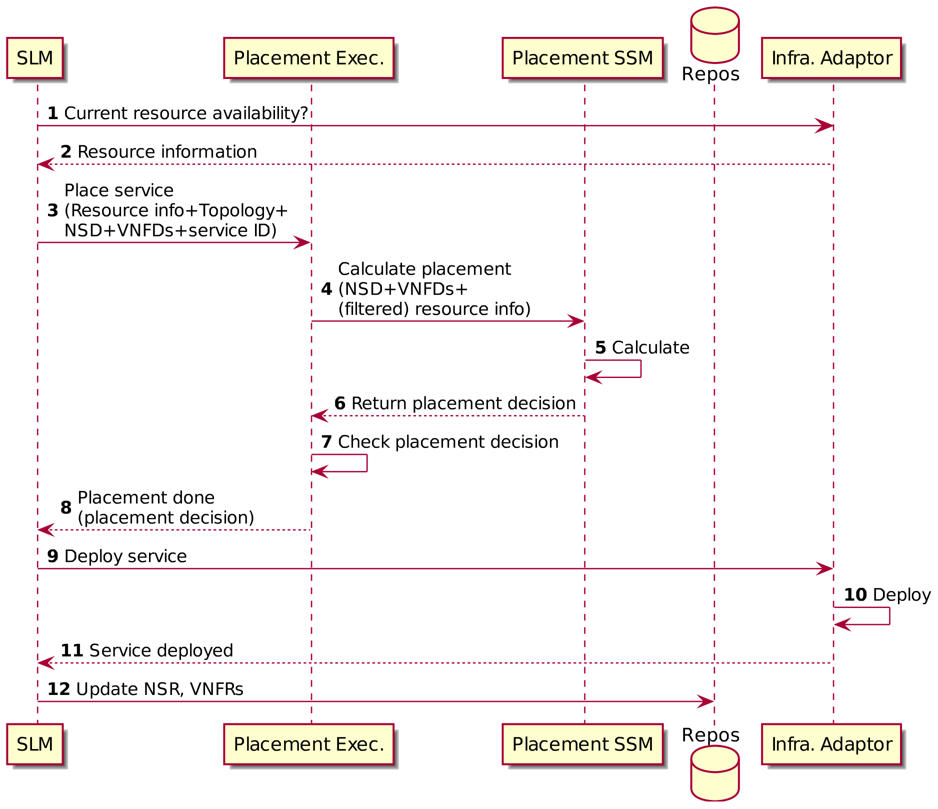

SLM
Placement Exec.
Placement SSM
Repos
Infra. Adaptor
1 Current resource availability?
2 Resource information
3 Place service (Resource info+Topology+ NSD+VNFDs+service ID)
4 Calculate placement (NSD+VNFDs+ (filtered) resource info)
5 Calculate
6 Return placement decision
7 Check placement decision
8 Placement done (placement decision)
9 Deploy service
10 Deploy
11 Service deployed
12 Update NSR, VNFRs

Figure 5.6: An example including service placement plugin

Figure 5.6 illustrates where the service placement functionality within the MANO framework can be customized. I.e., an executive plugin with pre-defined interfaces can accept SSMs that calculate the desired placement for services. This figure shows the workflow performed by the Service Lifecycle Manager (SLM), the placement executive plugin, placement SSM of a service, and the infrastructure adaptor for deploying a service with a specific placement preference.

Figure 5.7 extends this setup, by including a function-specific scaling manager. In this figure, the SLM receives a monitoring alert regarding a pre-defined threshold that has been reached and informs the scaling executive plugin. The alert is propagated over the dedicated message broker, together with the information required for taking a scaling decision. Such a function-specific scaling decision can result in including additional instances of the function in the service structure. For this reason, the placement for the service needs to be recalculated in order to maintain the optimal state for the network and service.

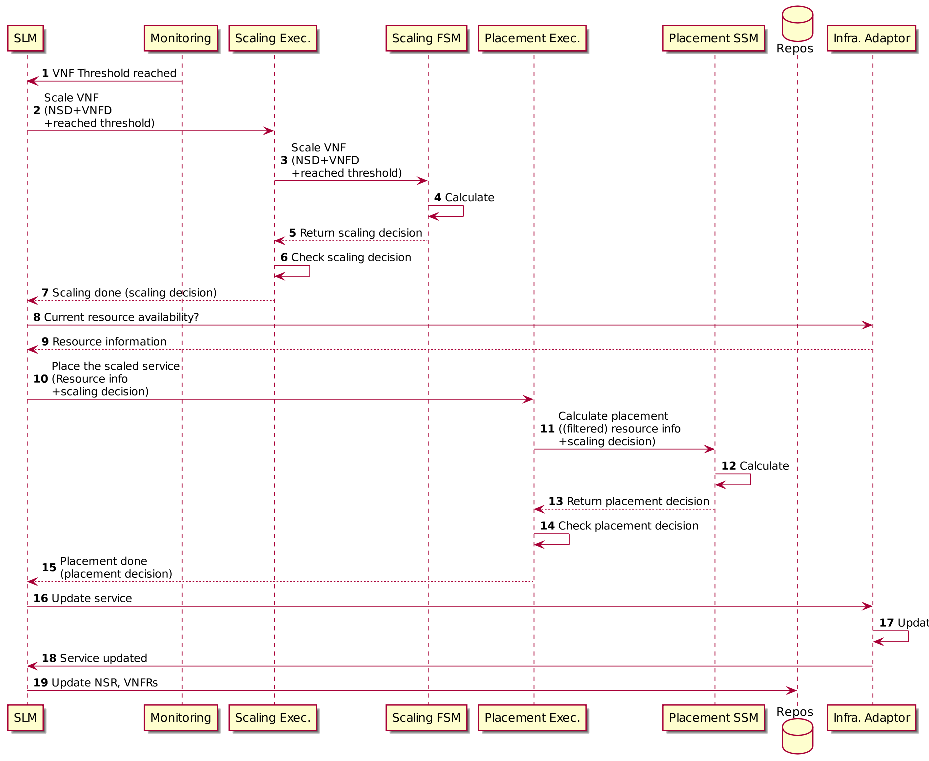

SLM
Monitoring
Scaling Exec.
Scaling FSM
Placement Exec.
Placement SSM
Repos
Infra. Adaptor
1 VNF Threshold reached
Scale VNF 2 (NSD+VNFD +reached threshold)
Scale VNF 3 (NSD+VNFD +reached threshold)
4 Calculate
5 Return scaling decision
6 Check scaling decision
7 Scaling done (scaling decision)
8 Current resource availability?
9 Resource information
Place the scaled service 10 (Resource info +scaling decision)
Calculate placement 11 ((filtered) resource info +scaling decision)
12 Calculate
13 Return placement decision
14 Check placement decision
15 Placement done (placement decision)
16 Update service
17 Update
18 Service updated
19 Update NSR, VNFRs

Figure 5.7: An example including function scaling and service placement plugins

5.3 Specific Managers Registry

Specific Manager Registry (SMR) is a MANO framework plugin that is responsible for FSM/SSMs lifecycle management including onboarding, instantiation, registration, updating, and termination. As shown in Figure 5.5, it interacts with other MANO framework plugins through the message broker, e.g., to obtain SSM onboarding request from SLM. It also employs the SSM/FSM repository to store a record of FSM/SSM (FSMR/SSMR).

Note that the initial SMR was presented as part of SONATA's first year review but was not part of of deliverable D4.1 ([7]), so this section documents its functionalities.

5.3.1 SMR features

This section describes functionalities that SMR provides in order to manage SSM/FSM lifecycle.

5.3.1.1 SSM/FSM on-boarding

SSM/FSM Onboarding function downloads SSM/FSM images from the docker registry that stores the SSM/FSM images. It retrieves SSM and FSM image URIs from Network Service Descriptor (NSD) and Virtual Network Function Descriptor (VNFD), respectively. SMR obtains NSDs from SLM and VNFDs from Function Lifecycle Management (FLM) through onboarding message requests. The SSM and FSM onboarding workflows are shown in Figure 5.8 and Figure 5.9, respectively.

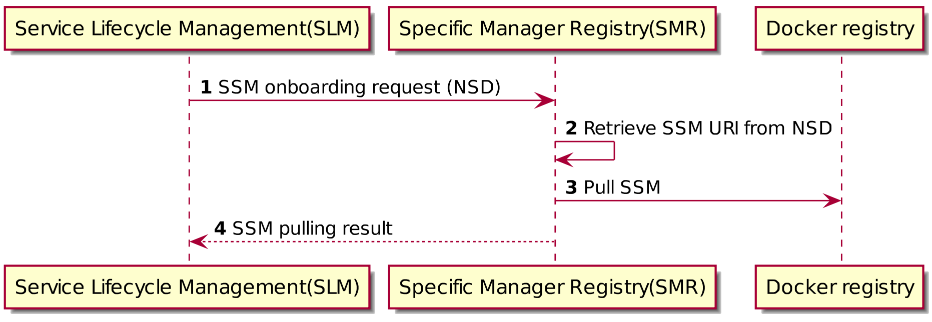

Service Lifecycle Management(SLM)
Specific Manager Registry(SMR)
Docker registry
1 SSM onboarding request (NSD)
2 Retrieve SSM URI from NSD
3 Pull SSM
4 SSM pulling result

Figure 5.8: Sequence diagram for SSM onboarding

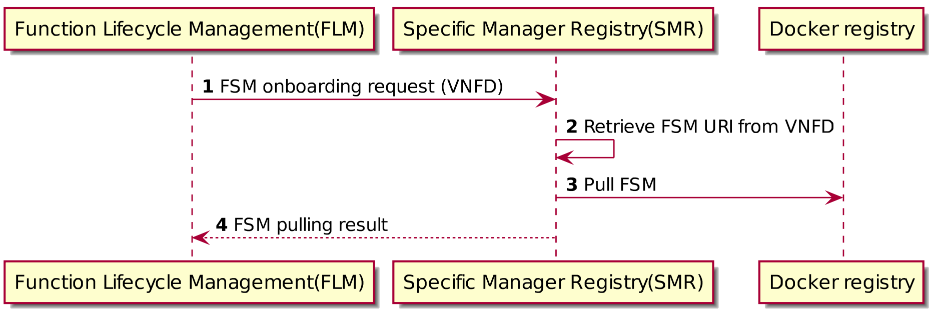

Function Lifecycle Management(FLM)
Specific Manager Registry(SMR)
Docker registry
1 FSM onboarding request (VNFD)
2 Retrieve FSM URI from VNFD
3 Pull FSM
4 FSM pulling result

Figure 5.9: Sequence diagram for FSM onboarding

5.3.1.2 SSM/FSM instantiation and registration

SMR instantiation function is responsible for starting SSM/FSM containers. SLM triggers this function for SSMs by sending an instantiation request message to SMR which contains the UUID of the service that the SSM belongs to. For FSMs, the triggering message comes from FLM which contains the UUID of the corresponding network function. This function is also responsible for SSM/FSM registration. Once SSM/FSM has been instantiated, it sends a registration request to SMR. Then, SMR generates a UUID for the SSM/FSM and stores a record of it in the SSM/FSM repository. Figure 5.10 and Figure 5.11 show the sequence diagram of SSM and FSM instantiation/registration, respectively.

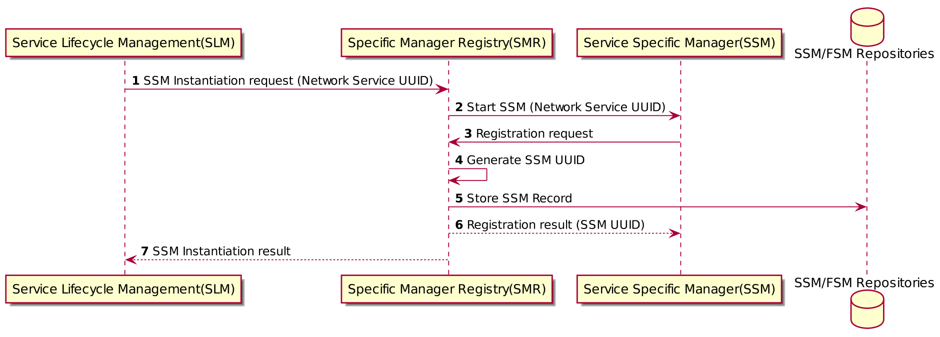

Service Lifecycle Management(SLM)
Specific Manager Registry(SMR)
Service Specific Manager(SSM)
SSM/FSM Repositories
1 SSM Instantiation request (Network Service UUID)
2 Start SSM (Network Service UUID)
3 Registration request
4 Generate SSM UUID
5 Store SSM Record
6 Registration result (SSM UUID)
7 SSM Instantiation result

Figure 5.10: SSM instantiation and registration sequence diagram

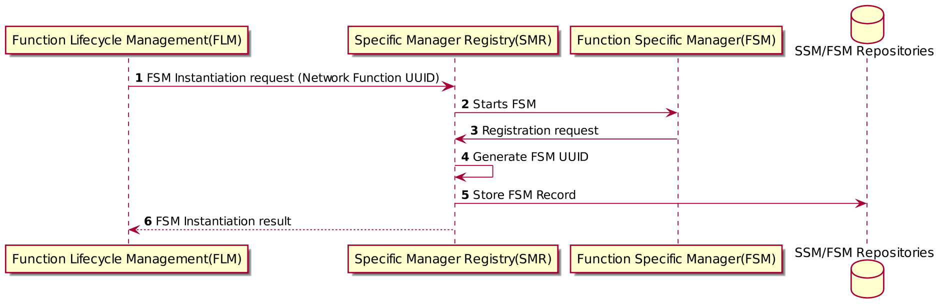

Function Lifecycle Management(FLM)
Specific Manager Registry(SMR)
Function Specific Manager(FSM)
SSM/FSM Repositories
1 FSM Instantiation request (Network Function UUID)
2 Starts FSM
3 Registration request
4 Generate FSM UUID
5 Store FSM Record
6 FSM Instantiation result

Figure 5.11: FSM instantiation and registration sequence diagram

5.3.1.3 SSM/FSM updating and termination

This function updates the running FSM/SSMs. SLM/FLM can trigger this function by sending a request message to SMR containing the NSD/VNFD that includes the URI of the new SSM/FSM image and the UUID of the SSM/FSM that is targeted for updating). This function, first, deploys the new SSM/FSM and then terminates the old one. The sequence diagram of updating SSM and FSM are shown in Figure 5.12 Figure 5.13, respectively.

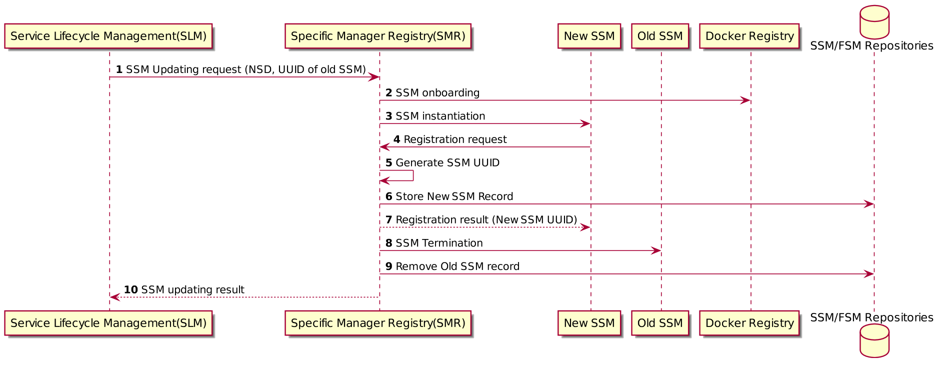


Figure 5.12: SSM updating and termination sequence diagram

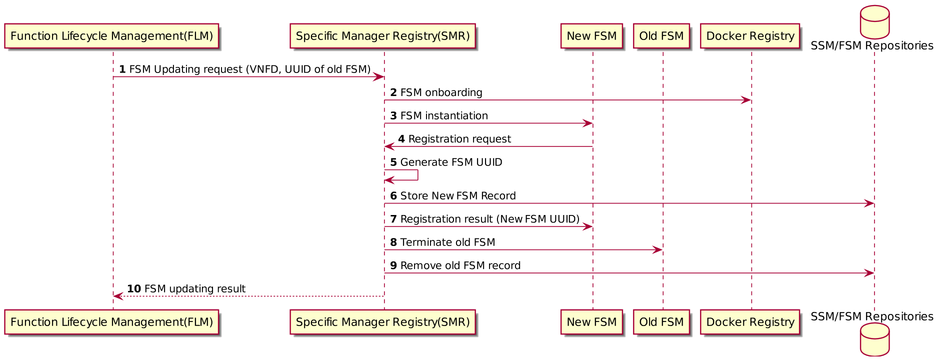


Figure 5.13: FSM updating and termination sequence diagram

6 Infrastructure Abstraction

In this section we describe the advances in the design and implementation of the Infrastructure Abstraction layer. The first subsection describes the **wrapper mechanism**, that has been used to provide abstract access by the MANO framework to VIM resources. The second subsection documents the update to the list of APIs offered by the **Infrastructure Abstraction** (IA) and the changes in the OpenStack wrapper, which are needed to meet the requirements of the new SP architecture. Finally the last subsection gives some details and plans for the extension of the Infrastructure Abstraction layer to support container-based VIM, such as Kubernetes.

6.1 Infrastructure Abstraction interfaces

The IA layer offers a technology independent view of the resource to the MANO framework. In order to achieve this objective, the IA is internally organised in so-called wrappers. These wrappers hide states, configurations, functions and implementations which are specific for each technology (OpenStack, OpenVIM, vCloud, OpenDaylight, VTN, etc...). Wrappers are divided into four categories:

- **Compute Wrapper**: Wraps a VIM to deploy virtual machines, create virtual networks, virtual routers and virtual ports.
- **Network Wrapper**: Wraps a VIM to deploy rules and rule sets to enforce Service Function Chaining or other network policies.
- **Storage Wrapper**: Wraps a VIM to store and retrieve VDU images.
- **WIM Wrapper**: Wraps a WIM to configure a WAN or a portion of a WAN to enforce intra-PoP connectivity for services.

Each of this wrapper is designed to be an interface, offering specific functions which are combined to implement the IA API. Compute, Network and Storage wrappers are considered as the three parts of a NFVI-PoP, so to able to store Virtual Machines (VMs) images, deploy these VMs and configure their internal networking. For this reason, for each Compute wrapper registered to the IA, there must be a relevant Storage wrapper to store the images needed for the service deployment, and a relevant Network wrapper to enforce network configuration on the instances of these images.

As an example, let us take into consideration the “infrastructure.service.deploy” call described in [7]. This needs several internal steps to be completed. In the first place, the VIM-Adaptor sub-module receives the call and must prepare the environment in the selected PoP(s) in order to host the service deployment. This step includes retrieving the appropriate storage wrappers from its internal repository, and pre-load (if needed) the VM images used by the service in the relevant VIM image repository. After this step, the relevant compute wrapper is retrieved from the IA repository, and the NSD that comes with the “infrastructure.service.deploy” request can be passed to it for the translation and deployment process. The actual deployment phase happens asynchronously under the hood of the compute wrapper, which notifies the VIM-adaptor when the process is completed. Once the service has been successfully deployed by the compute wrapper, the relevant network wrapper is called to enforce network configuration based on the Service Forwarding Graph (SFG) contained in the NSD.

After these steps, the service instance is ready to be used, so the WAN can be configured to route the relevant traffic to/through them. Therefore, the WIM-Adaptor sub-module is called to deal with the configuration of the underlying WIM(s). Also in this case the configuration is kept hidden by a relevant WIM wrapper, that receives the configuration parameters for the WAN, and enforces it with the specific technology it is wrapping.

All these components are implemented in Java, as documented in [7]. The communication between the different wrappers and the IA core is implemented through an observer design patterns, and not through method return values. In this design pattern, the wrapper always represents the observed resource while the IA core is the observer. Notifications carry YAML formatted strings which represent the outcome of the function call. These YAML updates are processed by the IA core to be translated to the format required by the IA-SLM and IA-FLM interfaces Section 8.

This mechanism allows the northbound IA API to remain unchanged when the IA is extended to support new VIMs. In fact, in order to implement such an extension, one needs to focus on the design and implementation of a model to translate from the SONATA NSD and VNFD into a FSM or a protocol specific for the wrapper VIM, or translate to a descriptor format supported by the VIM, if any. An example for this design is document in[7] for the OpenStack compute VIM. In Section 6.3 we draft the initial steps of the same process for the container based VIM Kubernetes.

6.2 New Infrastructure Abstraction functionalities

In the first implementation of the IA, we assumed that, once a NFVI-PoP has been selected by the MANO framework, it is used to deploy the complete network service, that is all its constituent VNFs are deployed on the same Compute VIM through the same Compute Wrapper. To fulfil this task, the Y1 Compute Wrapper interface exposed the deployService function. This function requires the complete NSD and the list of all VNFDs to be executed, and it returns a set of information that will be used by the MANO framework to generate the NSR and VNFRs. In order to allow a service to be deployed over multiple NFVI-PoP, and also to increase the number of lifecycle control functions that the MANO framework can carry out through the IA, the deployment process has been refactored splitting it in several smaller task. The IA offers new API calls through the Message Bus that can be used to:

- prepare a list of VIMs to host VNFs for a specific service instance
- deploy a VNF for a specific service instance on a specific VIM
- pause all the functions instances belonging to a specific service instance
- remove a specific function instance
- configure SFC and network policies within all NFVI-PoP involved in a service deployment
- deconfigure SFC and network policies within all NFVI-PoP involved in a service deployment
- configure the WAN between the NFVI-PoP involved in a service deployment
- deconfigure the WAN between the NFVI-PoP involved in a service deployment

These APIs are documented in detail in Section 8.11.0.1 and Section 8.12. Here we give an example of how this API calls are mapped by the IA core to specific calls to the relevant Wrapper interfaces and are used by the MANO framework to manage functions and services lifecycle. Before going into these details, we give an overview on the wrappers implemented in the first IA prototype and on their evolution.

6.2.1 OVS Networking Wrapper

Service Function Chaining is a fundamental operation at the very bottom of the NFV concept. Nonetheless and maybe for this reason, there is no standard solution to SFC in the most common VIM or cloud manager. Project **Service Insertion And Chaining** of OpenStack Neutron [29] and IETF NSH protocol draft [20] are among the various examples of how SFC could be implemented. Since none of this solution is at a mature stage or provides a mature implementation yet, for our first prototype we implemented our own custom SFC agents. It is based on OpenVSwitch, in particular on the ovs-ofctl, which allows to set and manage open flow rules. This agent, running on the Neutron controller of an OpenStack instance, can re-write flow rules so to re-direct specific flows, identified by a pair of IP addresses, through an ordered list of ports, identified by their MAC address, i.e. input and output ports of the VNFCs running inside that OpenStack instance. The IA provides a relevant network wrapper that is able to interact with this SFC agent, providing the ordered list of MAC addresses ports which compose the chain and the flow identifier. The ordered set of MAC addresses is created by the wrapper, which parses the forwarding paths in the forwarding graph, mapping each edge of the graph to the pair of MAC addresses associated with the connection point that the edge unite, using the information contained in the provided NSR and VNFR.

6.2.2 VTN WIM Wrapper

OpenDaylight [31] is an open source SDN Platform that delivers inter-operable, programmable networks. One of it's components is VTN Manager, that manages the virtual network at VTN (Virtual Tenant Network) level. VTN is able to set traffic flow rules, that allow or prohibit communication, as well as redirect packets that meet a particular condition. It exposes its functionality through a northbound interface, implemented through a REST API, that allows controlling rules table and managing virtual network resources. In the scope of SONATA, we implemented a prototype of a WIM, used for managing the WAN between NFVI-PoPs, which is based on OpenDaylight VTN manager. Through VTN in fact, virtual bridges can be created upon physical SDN switches, allowing the forwarding scheme of each flow to be defined programmatically. By this means, the IA can control and configure the WAN to redirect traffic belonging to specific service instance through the needed NFVI-PoP, using the relevant WIM wrapper. The wrapper communicates with the VTN WIM via REST API. The routing complexity is delegated by the IA to the WIM itself, since all the WAN topology and routing information are outside the IA scope. Therefore, the VTN WIM wrapper must only forge a request for the VTN WIM containing the details of the traffic flow to be diverted, and the ordered list of the NFVI-PoP to be traversed. Anyway, managing complex topology in such an abstract way brings interesting challenges. For example one must provide identifier for each PoP which are both part of the IA data model (i.e. IA generated UUID) and understandable by the WIM, which could be managed by a third actor with respect to the SONATA SP operator. Moreover, optimising routing on complex topology in a dynamic and programmatic way is well-known theoretical problem. For these reasons, our first implementation of the VTN WIM assumes a very simple WAN composed by a single switch. The implementation of a WIM managing a more complex, multi-PoP topology is part of the future development plans.

6.2.3 OpenStack Heat Wrapper

The new functionalities exposed by the IA to the MANO framework, namely the Multi-PoP deployment and the advanced function lifecycle management, impose a re-design of the translation model used by the OpenStack wrapper to translate from the SONATA descriptors to the OpenStack data model, which has been presented in [7]. In fact, in the new service deployment, the FLM take care of the deployment of each VNF separately, and so each VNFD is sent separately to the IA. Therefore the VIM adaptor must be able to carry out the deployment of each function independently from the others. To do this, we changed the way our **Network Service** (NS) and VNFs models are translated to Heat template, and also slightly modified the protocol of service and function deployment. As documented in Section 5.1.1, the IA must leave control for the deployment and lifecycle management of VNFs to the MANO framework, also allowing the VNFs of a NS to be deployed in different NFVI-PoP. In order to do this, the translation process of a VNFD must be independent from other VNFDs and from the NSD. In the model proposed in [7], this was not possible since networks, sub-networks, routers and ports are created and identified using a complete view of the Network Service, that is the NSD and all the constituent VNFDs. Therefore, our new model radically changes the way internal connectivity is provided to virtual machines and the control the way Heat stacks are used to represent VNFs and NS. More in details, in our previous model an Heat stack was used to represent a complete NS, with all its constituent VNFs. Now a stack running on an OpenStack VIM only represents the subset of VNFs of a NS which have been placed on that specific VIM. Moreover, since a NS virtual link can span more than one PoP, it cannot be directly mapped 1-to-1 to a Neutron Virtual Router, so intra-VNF connectivity is completely delegated to Networking Wrapper, leaving to the Compute Wrapper in general, and OpenStack Heat in particular, the task to create basic layer 2 and layer 3 connectivity where SFC or other network policies could be enforced later on by a Networking VIM. Table Table 6.1 resumes the translation model used in our first prototype and the mapping foreseen for the second year prototype of the IA.

Table 6.1: SONATA to OpenStack translation model revised

SONATA element	Abstraction model element	New Heat model element
VNFC	Virtual Machine	Virtual Machine
VNFC Connection point	Virtual Machine Port	Virtual Machine Port
VNF Virtual Link	Layer 3 network	L2 forwarding rule/L3 routing rule
VNF Connection Point	Router Port	No mapping
NS Virtual Link	Router	L2 forwarding rule/L3 routing rule
NS Connection Point	Router Port	No mapping

6.2.4 Multi PoP deployment example

To better understand how the combination of these wrapper and models can be used to ensure the foreseen functionalities we will navigate through an example. Figure 6.1 shows the initial state of a NFVI composed by two PoP, both using their instance of OpenStack heat as a compute VIM (VIM1 and VIM2 in the figure), a Glance instance as a Storage VIM and an instance of our OVS SFC agent as Network VIM. (for simplicity, we omit Glance and the compute VIM operation from the picture). The SLM receives a request to instantiate a service composed by three VNFs:

- VNF1 is composed by two VNFC

- VNF2 and VNF3 are composed both by a single VNFC.
- Traffic is flowing from a server to users, but after the NS deployment the traffic coming from the server needs to be processed in order by VNF1 VNF2 and VNF3 before reaching users.
- The SONATA placement executive decided that VNF1 and VNF2 should be deployed on VIM1 and VNF3 should be deployed on VIM2.

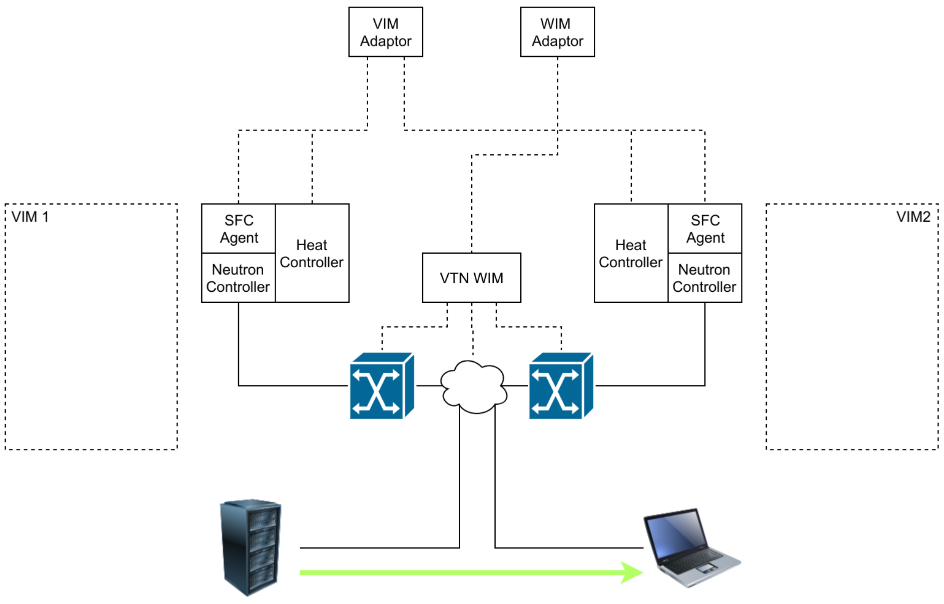


Figure 6.1: Initial state of two NFVI-PoP, the relevant VIMs and the IA entities

As depicted in Figure 5.1, the SLM sends a first request to the IA to prepare the selected VIMs to host the deployment. The VIM adaptor will receive this request and call the compute wrappers of the two OpenStack VIMs in order to create the virtual networks and sub-networks to connect the VNFs that will be deployed in the future. Each VIM creates a Data network, which will be used to exchange user data between VNFs and VNFCs, and a management network, that will be used for management and monitoring traffic. To do this, the OpenStack Wrapper will create a stack with just this two networks, and will connect the management network to the external gateway of the OpenStack tenant, in order to ensure external connectivity to the virtual machines through the management network. This is depicted in Figure 6.2, in the top-left sub-figure, where bold dotted lines between the IA and the VIMs represent the interface through which a given operation is carried out.

After this process is completed the SLM triggers the FLM to handle the deployment of each VNF separately. In turn, The FLM sends a request for VNF1 to be deployed in VIM 1 to the IA. the VIM adaptor receives this request and calls VIM1 compute wrapper to deploy the function. VIM1 compute wrapper translates the VNFD, connecting each VNFC connection point to the data network or to the management network depending on the connection point type specified in the descriptor (internal->data; external/public->management). Then, it retrieves the stack template from the Heat controller and updates it with the new resources. After this process is completed, the deployment will look as in Figure 6.2 top-right. The same procedure is repeated for VNF2 and VNF3, as depicted in Figure 6.2 bottom-left and bottom-right. Before the actual instantiation, the VIM Adaptor could also call the storage wrapper of each PoP in order to load the needed VDU

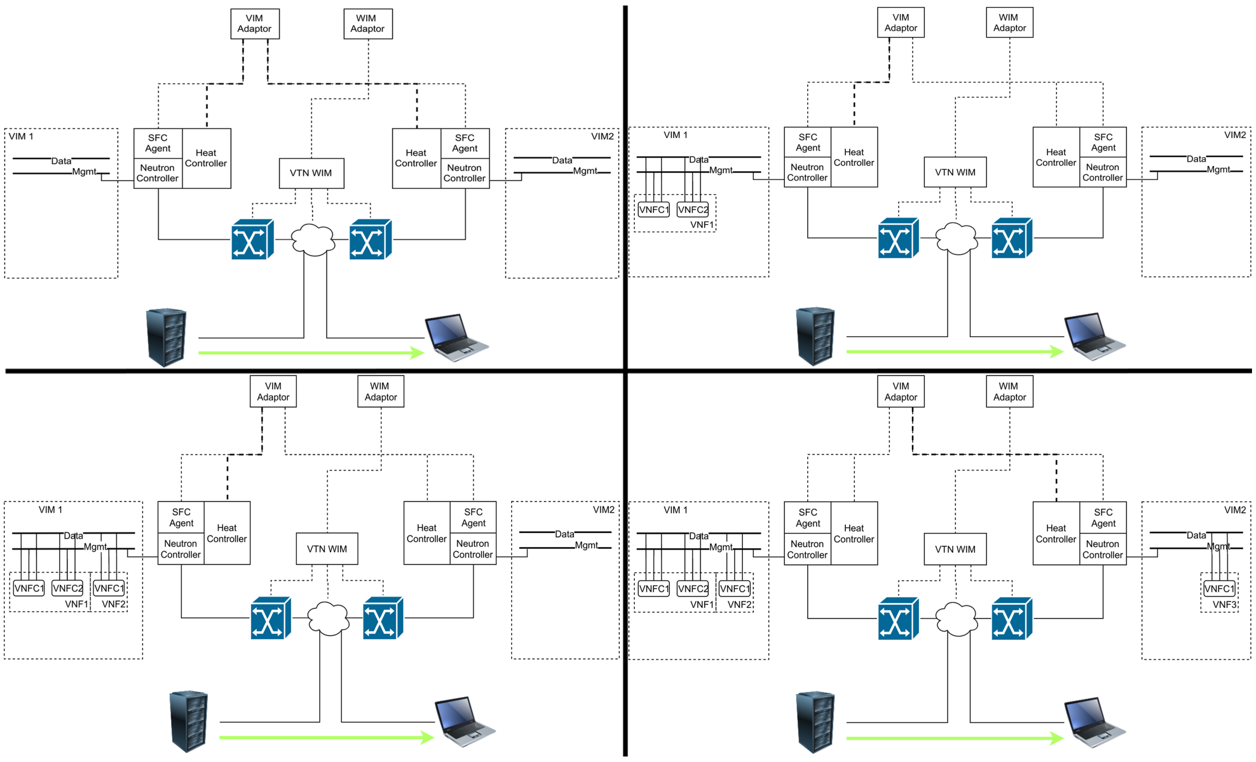


Figure 6.2: Different status of a NS deployemnt

images into the image repository of the VIM for the future deployment. We omitted this step from the pictures for simplicity.

After all the VNFs have been deployed, the control goes back to the SLM (see Figure 5.1, step 11), which issues another request to the IA to configure the VNF chain(s). Also in this case, the VIM adaptor receives the request and calls the network VIM wrappers of PoP1 and PoP2 to serve it. Both wrappers will receive the NS forwarding graphs, together with the NSR and the VNFRs of the deployed service instance, so to be able to compute the subset of the service graph each Network VIM is responsible for, and configure it through the OVS SFC agent they wrap. Figure 6.3 shows the status of the system after this procedure is completed. The red arrows represent forwarding/routing rules configured in the Neutron controlled through the SFC agent.

Finally, the SLM calls for the WAN to be configure, so that user traffic could be processed by the new NS instance. This time, the WIM adaptor receives the request and calls the relevant WIM wrapper, passing it the ordered list of the involved PoP and the identifier of the traffic flow to be redirected. Figure 6.4. The VTN WIM will set up the WAN so to redirect traffic that was previously flowing from the server to the users, through the involved NFVI-PoP (Green numbered arrows in Figure 6.4).

6.2.5 Service Lifecycle Status management

With the proposed model explained in the previous sub-section, we also provide to the MANO framework the level of control needed to change the status of a service/function lifecycle. When the MANO framework wants to put a service instance in a paused state, see Section 5.1.2.1, it can leverage the new IA functions, by sending two separate calls to deconfigure the WAN, so that there's no connectivity disruption between the server and the user (steps 2 and 3 in figure Figure 5.2), and to pause the service instance (steps 4 and 5 in figure Figure 5.2). The first call will be fetched by

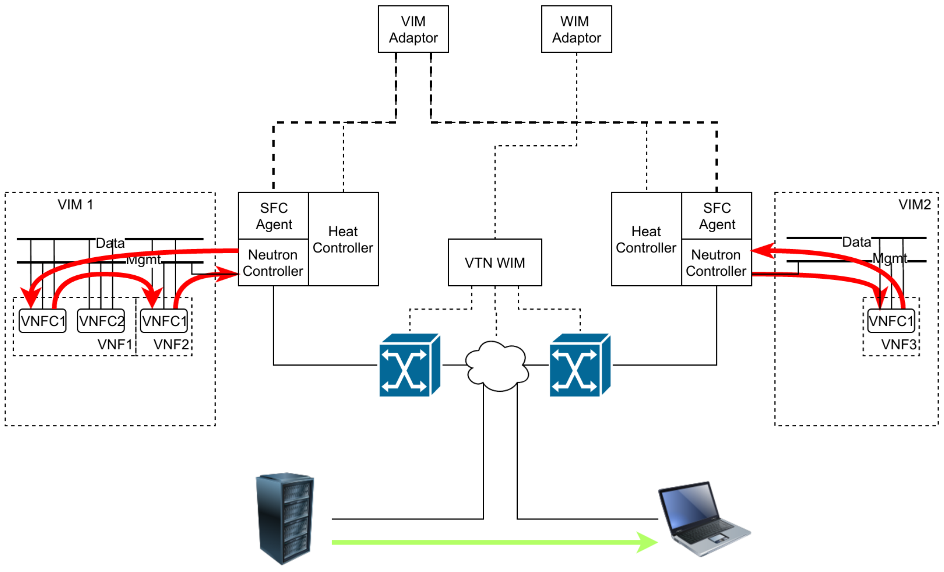


Figure 6.3: Service status after SFC configuration

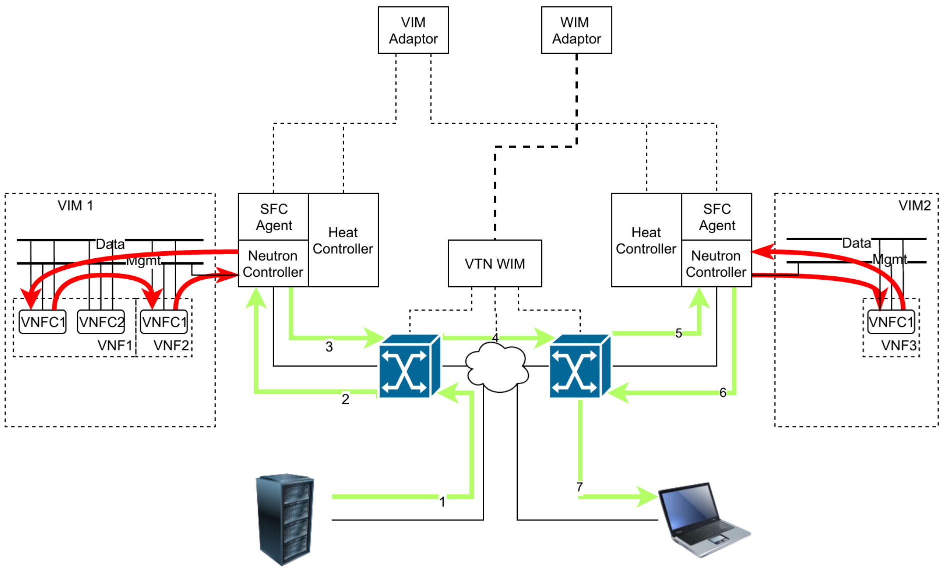


Figure 6.4: Service status after WIM configuration

the WIM adaptor to the relevant WIM wrapper and mapped into a request to the WIM to disable the WAN forwarding rules set for the given service instance. The second call will be dispatched by the VIM adaptor to the compute Wrapper for VIM1 and VIM2 requesting to pause the portion of service running on them. Consequently, each OpenStack Wrapper will use the Heat controller to pause the Heat stack where the VNFs are running. It is worth mentioning that there is no need to de-configure the SFC rules in the Neutron controller. The status after this calls is depicted in Figure 6.5.

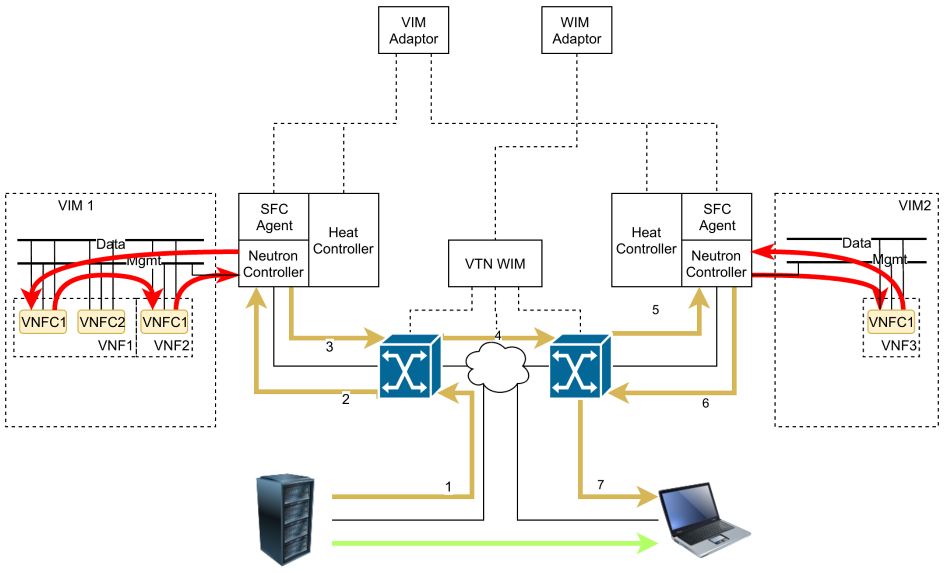


Figure 6.5: Service status after a pause procedure. Yellow elements are paused/deactivated.

Resuming a service (See Section 5.1.2.2 and Figure 5.3) follows the inverse procedure. First, each OpenStack wrapper is called in turn by the VIM adaptor to resume the execution of the relevant VNFs. Then, the WIM adaptor requests the WIM to reactivate the WAN forwarding rules set for the given service instance.

6.3 Kubernetes Wrapper

This section compiles the details of the exploratory ideas which will support Kubernetes VIM [2] and its REST API client to manage the Kubernetes orchestrator.

6.3.1 Kubernetes REST API Client

In order to interact with the Kubernetes endpoint, the Adaptor will wraps the Java client library. This library offers the API to authenticate to Kubernetes and perform several operations. The wrapped client library currently supports the following operations:

- Create a service provided by YAML file definition
- Retrieve the status of the service
- Delete the service

6.3.2 Kubernetes API object creation and translation model

Since a Kubernetes API object is needed to deploy a service, the Kubernetes Wrapper will implement a translation from the SONATA Service Descriptor and VNF Descriptor, detailed in [7], to Kubernetes API object. To achieve this translation, the VIM Abstraction layer resorts to an intermediate mapping, which aims at representing the abstract service definition in the SONATA descriptors in a more deployment-oriented way.

Table 6.2 describes the translation model to be implemented for SONATA in order to use Kubernetes as a VIM.

Table 6.2: SONATA to Kubernetes experimental translation model

SONATA element	Abstraction model element
VNFC	Container
VNFC Connection point	Container (IP, Port)
VNF	Kubernetes POD
VNF Virtual Link	L2 forwarding rule/L3 routing rule
NS Virtual Link	L2 forwarding rule/L3 routing rule

6.3.2.1 Kubernetes schema model

The model shown in Figure 6.6 is the first approach to Kubernetes uses as VIM in an NFV environment. In general, each container is a microservice, but in our model they represent VNFCs. A combination of one or more container represents a VNF. This combination of containers is called POD in the Kubernetes model. A Kubernetes POD is co-located in a physical infrastructure and share the same resources, such as network, memory and storage as the node. Each POD gets a dedicated IP address that is shared by all the containers which compose it. Each container which runs within the same pod gets the same *host name*. In this way they can be addressed as a unit. Going one step further, in order to build a NS, we need to combine VNFs. According to our general wrapper module, we can achieve this objective by combining a group of PODs and connecting them through a set of networking rules inside a virtual network, internal to the NFVI-PoP. Networking between PODs is a feature which is normally not considered in the Kubernetes environment, since PODs are considered to be self-contained application, one independent from each other. But since SFC is a crucial component of 5G Networks, inter-POD networking is a fundamental aspect we are going to investigate in the future.

6.3.3 Impact of container based VIM on the IA northbound API

The purpose of the Infrastructure Abstraction layer is exactly to offer a stable API to the MANO framework, leaving the complexity of the evolving plethora of resource managers under the hood. But the introduction of support for container-based managers in an NFV service platform is an advance that has a big impact on the whole service lifecycle, from development to operation. For this reason, the IA layer data model needs to be extended to cope with NFVI-PoPs in which it could deploy VNFs which are composed by VDU based on virtual machines, and with NFVI-PoP in which only container-based VNFs could be deployed. Complexity could be increased if we assume that a Network Service could be composed both of container-based VNFs and hypervisor-based VNFs.

In this sense, the improvement we described above in terms VNF lifecycle management and multi-PoP deployment, are a key aspect to allow the MANO framework to take care of the deployment of each VNF in an independent way, and with the needed degree of freedom with respect to the

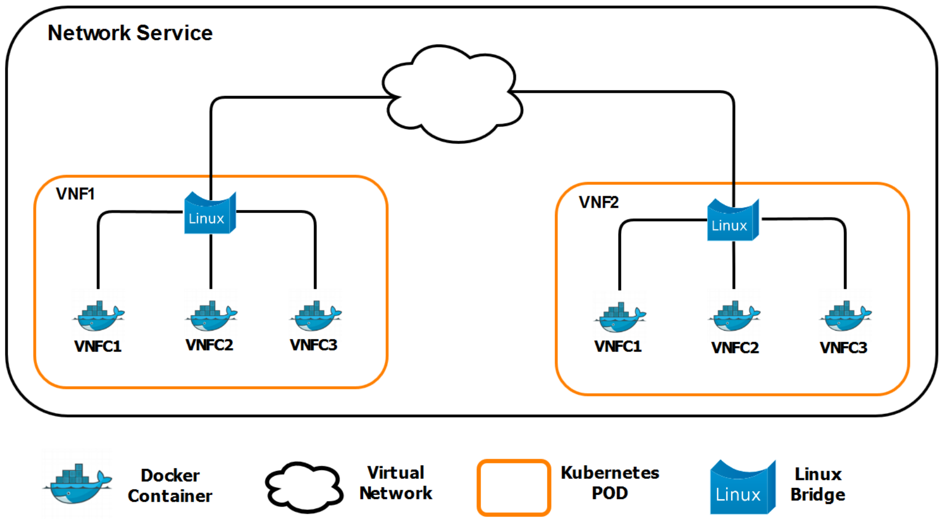


Figure 6.6: Kubernetes Schema Model

NFVI-PoP selection. Both the MANO framework and the IA need to be refined to include in their interface information regarding the specific technology needed by a VNF (MANO->IA) and the specific technology offered by a PoP (IA->MANO). With these interface refinements, the MANO placement executive, or an SSM deputed to placement, could decompose the placement problem in sub-problems, dealing separately with the problem of placing container-based VNFs on the the container-based NFVI, and hypervisor-based VNFs on the hypervisor-based NFVI. The outcome of this process could be delivered by the SLM/FLM to the IA with very minimal changes to the present API, since the deployment of each VNF will be independent from the others, and the technology specific procedures to deploy a container VNF or a virtual machine VNF will be hidden by the relevant VIM wrappers.

7 Monitoring Framework

This section presents the enhancements that are the outcome of several ongoing activities on other SONATA components and use cases. In particular, monitoring framework enhancements have been decided in order to support use cases requirements, integration with other SONATA components, resolve scalability and reliability issues of the Service Platform monitoring manager, support multi-PoP monitoring and alerting functionality, etc.

The enhancements discussed in this section follows the DevOps approach that has been adopted by the project and are part of the SONATA development roadmap of the second year of the project.

7.1 Streaming monitoring data to the SDK

One of the major enhancements included in the development cycle of the second year of the project is the development of a mechanism that allows the service developer to collect (to the SONATA SDK) streaming data from VNFs that have been deployed in the Service Platform. This will be highly beneficial to the developers, as they would be able to monitor the performance of a new service in real environment (in contradiction to the emulator in the SDK), be it the integration or the operational environment as well as in real-time, closing the cycle between Development and Operations (DevOps).

From the requirements elicitation viewpoint, this streaming monitoring mechanism must provide an interface to the developers that would allow to specify the metrics to be streamed back to the SDK and the duration per VNF.

From a technical perspective, it has been decided that the most convenient solution would include the adoption of websockets to accommodate the streaming of monitoring data to the developer.

In order to support this mechanism, there is a need for the definition of two new API calls through the Monitoring Manager:

- `/prometheus/metrics/stream/function/{funcID}/{duration}`: this call will define the beginning of streaming data of all the metrics collected by a particular function ID for a defined period of time;
- `/prometheus/metrics/stream/function/{funcID}/metric/{metricID}/{duration}`: this call will define the beginning of streaming data of a specific metrics collected by a particular function ID for a defined period of time.

Prior to these calls, the developer must be aware of the VNF ID, the metrics collected per VNF, the VNFs comprising his deployed Network Services and other related information and this information is already provided by the existing Monitoring Manager API framework.

There are two already developed and implemented API calls related to this activity, as described below:

- `/functions/service/{srvID}`: this GET call returns details regarding the list of functions composing a network service;
- `/metrics/function/{funcID}`: this GET call returns the metrics collected per VNF.

The sequence of the actions in establishing a streaming connection between the SDK (developer) and Service Platform (Monitoring Framework) is illustrated below in the Figure 7.1. Upon a request from the SDK, a web socket is opened on the SP (via the GK) where metric values can be pushed into. The SP is in control of the data transfer and it can choose to expose monitored data as soon as it is available. The SDK connects to this web socket to automatically receive the (filtered) monitored data.

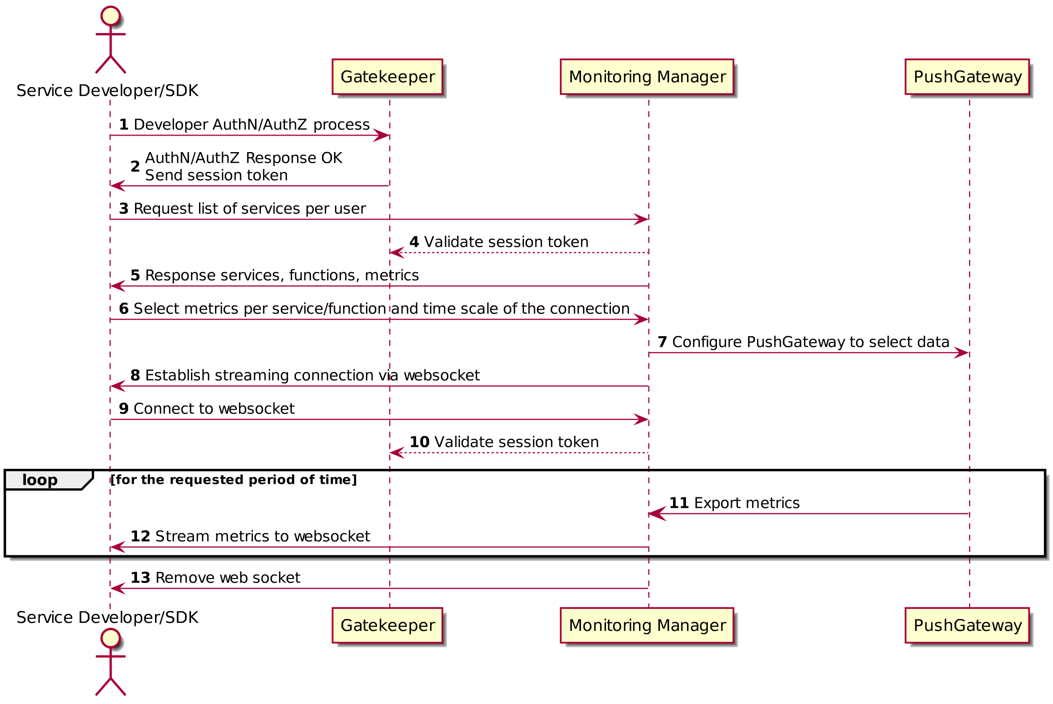

Service Developer/SDK
Gatekeeper
Monitoring Manager
PushGateway
1 Developer AuthN/AuthZ process
2 AuthN/AuthZ Response OK
Send session token
3 Request list of services per user
4 Validate session token
5 Response services, functions, metrics
6 Select metrics per service/function and time scale of the connection
7 Configure PushGateway to select data
8 Establish streaming connection via websocket
9 Connect to websocket
10 Validate session token
loop
[for the requested period of time]
11 Export metrics
12 Stream metrics to websocket
13 Remove web socket
Service Developer/SDK
Gatekeeper
Monitoring Manager
PushGateway

Figure 7.1: Retrieval of Monitoring Data

7.2 Support of user management functionality

During the first year and the testing and evaluation of the implemented features, it became clear that Monitoring Manager must comply with the user management principles of the SONATA project and offer more preferences to the users.

This requirement has led to the definition of a set of API calls related to user management and information. In particular:

`/users` This call provides the ability to the privileged users (such as the Service Platform administrators) to retrieve the list of authorized users of the SONATA Service Platform.

`/user/:pk` This call provides detailed information with respect to a specific user (primary key). There are many fields defined per user, including first and last name, SONATA Service Platform user ID, date of user creation, etc.

7.3 Enhancements related to scalability and reliability of the monitoring framework

This category includes extensions to the API to be implemented in the Monitoring Manager to support new functionality related to requirements imposed by the use cases and the extensibility of SONATA to multiple VIMs and PoPs.

One of the enhancements is related to the ability provided to the developer to configure the monitoring settings of his deployed services. The settings include the frequency that the monitoring data are collected, the rule/alert thresholds modifications, etc.

`/prometheus/configure` This API call will retrieve and or insert configuration parameters related to the monitoring frequency, rules/alerts, notification settings, etc to the Prometheus server.

Another issue is related to the large flow of monitoring data from multiple PoPs to the Monitoring Server and its respective database that might affect the Service Platform performance. In this respect, it has been decided that one monitoring server (Prometheus server) will be deployed per PoP and one (high-level) monitoring server instance will reside in the Service Platform whose database will store a subset of the data (data related to alerts) collected by the monitoring servers per PoP. By adopting this distributed architecture, monitoring framework will gain in terms of scalability and reliability.

Apart from the architectural decision concerning the distributed approach of monitoring framework components to address scalability and reliability issues, a number of API calls must also be defined, as explained below:

`/services` This API will return the list of services deployed in the Service Platform and among other information, it must include fields to support the multi-PoP environment of SONATA (PoP ID where each VNF is deployed, uuid of each VNF, etc).

`/pops` This API will return the list of PoPs connected to the SONATA Service Platform.

`/pop/:pk` This API will return detailed information for a specific PoP connected to the SONATA Service Platform.

7.4 API extensions

The following table Table 7.1 describes the RESTful API extensions of Monitoring Manager, in addition to those presented in Deliverable 4.1.

Table 7.1: Monitoring Manager REST API

Endpoint	Method	Description	Returned code(s)
`/users`	GET, POST	Retrieve/insert details about SONATA Service Platform users.	OK (200), Created (201), Not found (404)
`/user/:pk`	GET, PUT, PATCH, DELETE	Retrieve, insert, update, delete details related to a registered user.	OK (200), Created (201), Not found (404)
`/functions/service/{srvID}`	GET	Retrieve details regarding the list of functions composing a network service.	OK (200), Not found (404)

Endpoint	Method	Description	Returned code(s)
`/metrics/function/{funcID}`	GET	Retrieve details on metrics monitored in a function.	OK (200), Not found (404)
`/services`	GET, POST	Retrieve/insert details in the list of deployed services.	OK (200), Created (201), Not found (404)
`/pop`	GET, POST	Retrieve/insert details on functions and services deployed on PoPs.	OK (200), Created (201), Not found (404)
`/prometheus/configure`	GET, POST	Retrieve/insert configuration parameters related to the monitoring frequency, rules/alerts, notification settings, etc.	OK (200), Not found (404)
`/prometheus/metrics/stream/function/{funcID}/{duration}`	POST	Send the list of parameters to initiate streaming data (all monitored metrics) through websocket.	OK (200), Not found (404)
`/prometheus/metrics/stream/function/{funcID}/metric/{metricID}/{duration}`	POST	Send the list of parameters to initiate streaming data (one monitored metric) through websocket.	OK (200), Created (201), Not found (404)

8 Internal Interfaces

This section describes the internal interfaces of the **Service Platform**, making its modularity evident.

As already presented in **D2.3** [5], Figure 8.1 summarises these interfaces, which are detailed along the remaining of this section.

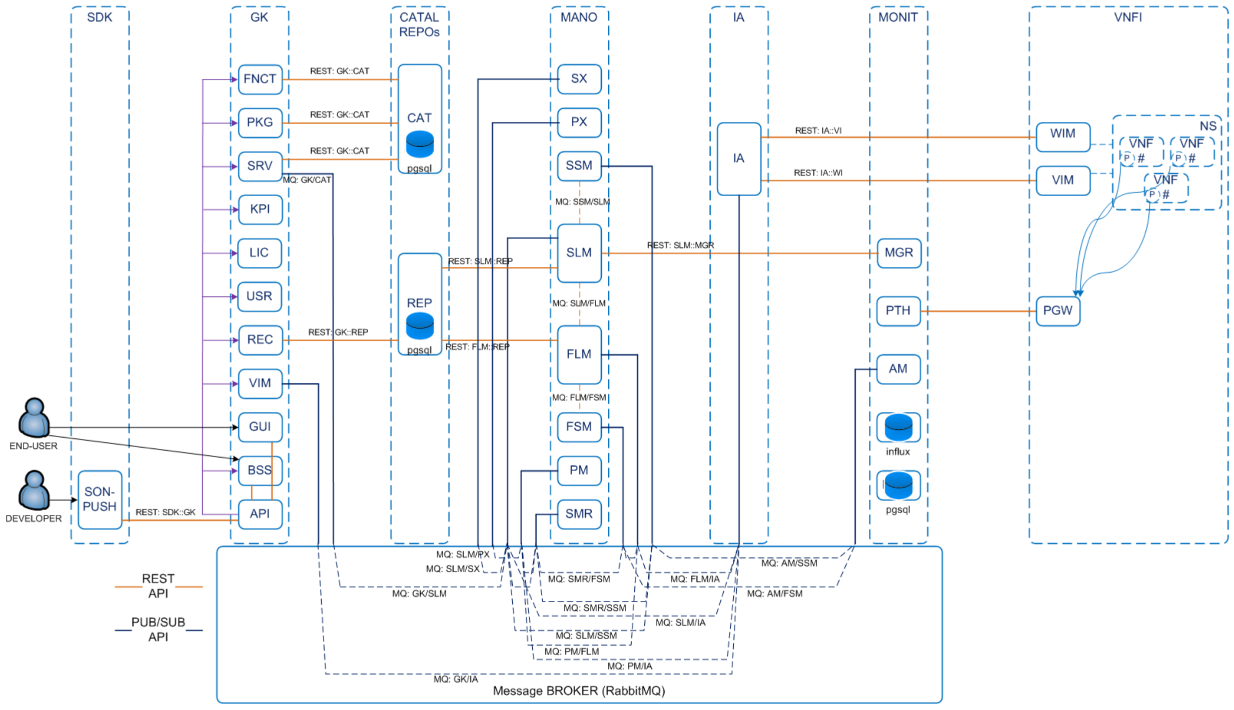


Figure 8.1: SP Component Interfaces

8.1 Graphical User Interface - Gatekeeper Interface

Table 8.1 shows the request Interface between the Graphical User Interface (GUI) and the Gatekeeper (GK). Table 8.2 shows the response Interface between the GUI and the GK.

Table 8.1: Request Interface between the GUI and the GK.

Action	Entity	Method	Path	Parameters	Parameter location	Required
query	user-role	GET	`/roles`	• role ID • name: Role's name	query	No

Action	Entity	Method	Path	Parameters	Parameter location	Required
create	user-role	POST	`/roles`	• name: Role's name	body	Yes
update	user-role	PUT	`/roles/<id>`	• name: Role's name	body	Yes
delete	user-role	DELETE	`/roles/<id>`			Yes
query	user	GET	`/users`	• user UUID • name • email • mobile	query	No
query	user	GET	`/users`	• user UUID	URL	Yes
create	user	POST	`/users`	• name • email • mobile	body	Yes
update	user	PUT	`/users/<id>`	• name • email • mobile	body	Yes

Table 8.2: Response Interface between the GUI and the GK.

Action	Entity	Http method	Path	Responses
query	service	GET	/services	• 200: List of services that meet search conditions retrieved • 404: No services with specified parameters were found
query	instance	GET	/records/services	• 200: List of service instances that meet search conditions retrieved • 400: No Records with specified parameters were found

Action	Entity	Http method	Path	Responses
query	request	GET	/requests	• 200: List of requests that meet search conditions retrieved • 400: No Requests with specified parameters were found
instantiate	service	POST	/requests	• 201: Request was created • 400: No service id was specified or no request was created
update	instance	PUT	/records/services	• 201: Update request created • 400: No Request was created or no valid instance UUID specified • 404: No service found or not nsd_id/latest_nsd_id specified

8.2 Business Support System - Gatekeeper Interface

Table 8.3 shows the request Interface between the Business Support System (BSS) and the Gatekeeper (GK). Table 8.4 shows the response Interface between the BSS and the GK.

Table 8.3: Request Interface between the BSS and the GK.

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	service	GET	/services	• status: the status of the services to be retrieved • vendor: the vendor of the service retrieved • name: the name of the service retrieved • version: the version of the service retrieved • offset: offset the list of retrieved results by this amount. Default is Zero. • limit: number of services to return. Default is 5, max is 100.	query	No

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	instance	GET	/records/services	• owner_id: the UUID of owner of the service • offset: offset the list of retrieved results by this amount. Default is Zero. • limit: number of services to return. Default is 5, max is 100.	query	No
query	request	GET	/requests	• service_id: the UUID of the service of which all requests are done • vendor: the vendor of the request retrieved • name: the name of the request retrieved • version: the version of the request retrieved • offset: offset the list of retrieved results by this amount. Default is Zero. • limit: number of services to return. Default is 5, max is 100.	query	No
instantiate	service	POST	/requests	• service_uuid: the UUID of the service from which an instance is required	body	Yes
update	instance	PUT	/records/services	• nsr_id: the UUID of the instance	query	Yes
update	instance	PUT	/records/services	• nsd_id: the UUID of the original service • latest_nsd_id: the UUID of the most updated service	body	Yes

Table 8.4: Response Interface between the BSS and the GK.

Action	Entity	Http method	Path	Responses
query	service	GET	/services	• 200: List of services that meet search conditions retrieved • 404: No services with specified parameters were found

Action	Entity	Http method	Path	Responses
query	instance	GET	/records/services	• 200: List of service instances that meet search conditions retrieved • 400: No Records with specified parameters were found
query	request	GET	/requests	• 200: List of requests that meet search conditions retrieved • 400: No Requests with specified parameters were found
instantiate	service	POST	/requests	• 201: Request was created • 400: No service id was specified or no request was created
update	instance	PUT	/records/services	• 201: Update request created • 400: No Request was created or no valid instance UUID specified • 404: No service found or not nsd_id/latest_nsd_id specified

8.3 Software Development Kit - Gatekeeper Interface

Table 8.5 shows the request Interface between the SDK and the GK. Table 8.6 shows the response Interface between the SDK and the GK. This interface will evolve to a more complete one, as features like User Management, Licence Management and KPIs become available on the Service Platform's side.

Table 8.5: Request Interface between the SDK and the GK.

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	package	GET	/packages	• status: the status of the package to be retrieved • vendor: the vendor of the package retrieved • name: the name of the package retrieved • version: the version of the package retrieved • offset: offset the list of retrieved results by this amount. Default is Zero. • limit: number of services to return. Default is 5, max is 100.	query	No
create	package	POST	/packages	• A .son file	form parameter	yes

Table 8.6: Response Interface between the SDK and the GK.

Action	Entity	Http method	Path	Responses
query	package	GET	/packages	• 200: List of services that meet search conditions retrieved • 404: No services with specified parameters were found
create	package	POST	/packages	• 201: Package was created • 400: No package was created

8.4 Gatekeeper - User Management Interface

Table 8.7 shows the Interface between the Gatekeeper and the User management module.

Table 8.7: Access Token verification endpoints.

Action	HTTP method	Path	Response	Requires token
Registration	POST	`/sessions/register`	• `200 OK` • `40X Unauthorized/Forbidden`	none
Log-in	POST	`/sessions/login`	• `200 OK` • `40X Unauthorized/Forbidden`	none
Log-out	POST	`/sessions/logout`	• `200 OK` • `40X Unauthorized/Forbidden`	bearer token
Token Authentication	POST	`/sessions/auth`	• `200 OK` • `40X Unauthorized/Forbidden`	bearer token
Token Authorization	POST	`/sessions/authorize`	• `200 OK` • `40X Unauthorized/Forbidden`	bearer token
Userinfo (OAuth 2.0 Claims)	POST	`/users/<username>/userinfo`	• `200 OK` • `40X Unauthorized/Forbidden`	bearer token
User profile update	POST	`/users/<username>/profile`	• `200 OK` • `40X Unauthorized/Forbidden`	bearer token
User authorization management	POST	`/users/<username>/authorizations`	• `200 OK` • `40X Unauthorized/Forbidden`	bearer token

8.5 Gatekeeper - Catalogue Interface

Table 8.8 shows the request Interface between the GK and the Catalogue (CAT). Table 8.9 shows the response Interface between the GK and the CAT. The DELETE is available on the Catalogue side of this API, although it has not yet been made available to external systems. This is due to the implications of deleting a descriptor, which still has to be designed (namely, what should happen when there are instances of a service which descriptor is requested to be deleted).

Table 8.8: Request Interface between the GK and the CAT.

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	NSD	GET	/network-services	• status: the status of the services to be retrieved • vendor: the vendor of the service retrieved • name: the name of the service retrieved • version: the version of the service retrieved • offset: offset the list of retrieved results by this amount. Default is Zero. • limit: number of services to return. Default is 5, max is 100.	query	No
query	VNFD	GET	/vnfs	• status: the status of the services to be retrieved • vendor: the vendor of the service retrieved • name: the name of the service retrieved • version: the version of the service retrieved • offset: offset the list of retrieved results by this amount. Default is Zero. • limit: number of services to return. Default is 5, max is 100.	query	No
query	PD	GET	/packages	• status: the status of the services to be retrieved • vendor: the vendor of the service retrieved • name: the name of the service retrieved • version: the version of the service retrieved • offset: offset the list of retrieved results by this amount. Default is Zero. • limit: number of services to return. Default is 5, max is 100.	query	No

Action	Entity	Http method	Path	Parameters	Parameter location	Required
submit	NSD	POST	/network-services	• None	body	Yes
submit	VNFD	POST	/vnfs	• None	body	Yes
submit	PD	POST	/packages	• None	body	Yes
update	NSD	PUT	/network-services	• UUID: the unique id of the original service	query	Yes
update	NSD	PUT	/network-services	• status: the new status for the updated service	body	No
update	VNFD	PUT	/vnfs	• UUID: the unique id of the original function	query	Yes
update	VNFD	PUT	/vnfs	• status: the new status for the updated function	body	No
update	PD	PUT	/packages	• UUID: the unique id of the original package	query	Yes
update	PD	PUT	/packages	• status: the new status for the updated package	body	No
remove	NSD	DELETE	/network-services	• UUID: the unique id of the original service	query	Yes
remove	VNFD	DELETE	/vnfs	• UUID: the unique id of the original function	query	Yes
remove	PD	DELETE	/package	• UUID: the unique id of the original package	query	Yes

Table 8.9: Response Interface between the GK and the CAT.

Action	Entity	Http method	Path	Responses
query	NSD	GET	/network-services	• 200: List of services that meet search conditions retrieved • 404: No services with specified parameters were found
query	VNFD	GET	/vnfs	• 200: List of functions that meet search conditions retrieved • 404: No functions with specified parameters were found
query	PD	GET	/packages	• 200: List of packages that meet search conditions retrieved • 404: No packages with specified parameters were found
submit	NSD	POST	/network-services	• 201: Descriptor was created • 400: No descriptor was created
submit	VNFD	POST	/vnfs	• 201: Descriptor was created • 400: No descriptor was created
submit	PD	POST	/packages	• 201: Descriptor was created • 400: No descriptor was created
submit	NSD	PUT	/network-services	• 200: Descriptor updated • 400: No update was created or not valid descriptor UUID specified
submit	VNFD	PUT	/vnfs	• 200: Descriptor updated • 400: No update was created or not valid descriptor UUID specified
submit	PD	PUT	/packages	• 200: Descriptor updated • 400: No update was created or not valid descriptor UUID specified

Action	Entity	Http method	Path	Responses
submit	NSD	DELETE	/network-services	• 200: Descriptor disabled • 400: No remove was committed or not valid descriptor UUID specified
submit	VNFD	DELETE	/vnfs	• 200: Descriptor disabled • 400: No remove was committed or not valid descriptor UUID specified
submit	PD	DELETE	/packages	• 200: Descriptor disabled • 400: No remove was committed or not valid descriptor UUID specified

8.6 Service Platform - Repositories Interface

In this section, the interfaces between different micro services in the SP and the Repositories (REP) are described. Table 8.10 describes how service records (NSR) can be stored and managed, Table 8.11 describes the same for the VNF records (VNFR). Each record is identified by a unique UUID (nsr_id/vnfr_id). The GK, the SLM and the FLM are the SP micro services that are using this interface.

Table 8.10: NSR Repository REST API

Uri	Method	Description	Returned code(s)
/records/nsr	GET	REST API Structure and Capability Discovery for /records/nsr/	• OK (200)
/records/nsr/ns-instances	GET	List all NSR instances in JSON format. It supports pagination with the offset (default value zero) and limit (default value ten) parameters.	• OK (200) • Not Found (404)
/records/nsr/ns-instances/:nsr_id	GET	List specific NSR instance information in JSON format	• OK (200) • Not Found (404)

Uri	Method	Description	Returned code(s)
/records/nsr/ns-instances	POST	Submit a new NSR instance. The content type have to be a JSON. It returns the NSR, also in JSON.	• OK (200) • Unprocessable Entity (422) • Conflict (409) • Unsupported Media Type (415) • Bad Request (400)
/records/nsr/nsr-instances/:nsr_id	PUT	Update a NSR instance. The content type have to be a JSON. It returns the updated NSR.	• OK (200) • Not Found (404) • Unsupported Media Type (415) • Conflict (409) • Unprocessable Entity (422)
/records/nsr/ns-instances/:nsr_id	DELETE	Delete a NSR instance. It returns the NSR	• OK (200) • Not Found (404)

Table 8.11: VNFR Repository REST management API

Uri	Method	Purpose	Returned code(s)
/records/vnfr/	GET	REST API Structure and Capability Discovery for /records/vnfr/	• OK (200)
/records/vnfr/vnf-instances	GET	List all VNF instances in JSON format	• OK (200) • Error Establishing a Database Connection (500)

Uri	Method	Purpose	Returned code(s)
/records/vnfr/vnf-instances?output=YAML	GET	List all VNF instances in YAML format	• OK (200) • Error Establishing a Database Connection (500)
/records/vnfr/vnf-instances/:vnfr_id	GET	List specific VNF instance information in JSON format	• OK (200) • Not Found (404)
/records/vnfr/vnf-instances/:vnfr_id?output=YAML	GET	List specific VNF instance information in YAML format	• OK (200) • Not Found (404)
/records/vnfr/vnf-instances	POST	Create a new VNF instance	• OK (200) • Parsing error (400) • Duplicated ID (400) • Unsupported Media Type (415)
/records/vnfr/vnf-instances/:vnfr_id	PUT	Update a VNF instance	• OK (200) • Parsing error (400) • Duplicated ID (400) • Unsupported Media Type (415)
/records/vnfr/vnf-instances/:vnfr_id	DELETE	Delete a VNF instance	• OK (200) • Not Found (404)

8.7 Service Lifecycle Manager - Monitoring Manager Interface

Table 8.12 shows the request Interface between the SLM and the Monitoring Manager (MON). Table 8.13 shows the response Interface between the SLM and the MON.

Table 8.12: Request Interface between the SLM and the MON.

Action	Entity	Http method	Path	Parameters	Parameter location	Required
query	user	GET	/api/v1/users	• first_name: user first name • last_name: user last name • email: user email address • sonata_userid: user sonata id • created: user entrance date	query	No
query	service	GET	/api/v1/services/user/{user_id}	• sonata_srv_id: sonata service id • name: sonata service name • description: service description • user: user description • host_id: host id (optional) • pop_id: pop id where host is deployed • created: service creation date	query	No
query	functions	GET	/api/v1/functions/service/{service_id}	• sonata_func_id: sonata function id • name: function name • description: function description • service: service description • host_id: host (vm/docker) id where function is deployed • pop_id: pop id in where host is deployed • created: function creation date	query	No
query	metrics	GET	/api/v1/metrics/function/{function_id}	• name: metric name • description: metric description • threshold: metric threshold • interval: metric time interval • command: metric command • function: function id • created: creation date of the metric	query	No

Action	Entity	Http method	Path	Parameters	Parameter location	Required
instantiate	service	POST	/api/v1/service/new	• A .son file	from parameter	No
instantiate	users	POST	/api/v1/users	• first_name: User's first name • last_name: user last name • email: user email address • sonata_userid: sonata user id	body	No

Table 8.13: Response Interface between the SLM and the MON.

Action	Entity	Http method	Path	Response
query	user	GET	/api/v1/users	200: List of users received
query	service	GET	/api/v1/services/user/{user_id}	200: List of services retrieved
query	functions	GET	/api/v1/functions/service/{service_id}	200: List of functions retrieved
query	metrics	GET	/api/v1/metrics/function/{function_id}	200: List of metrics retrieved
instantiate	service	POST	/api/v1/service/new	201: New Service updated
instantiate	users	POST	/api/v1/users	201: New user created

8.8 Ia-Vi Interface

Table 8.16 shows the subset of the OpenStack API that are part used in the interface between the Infrastructure Abstraction and the VIM. It includes both API of Nova and Heat, the OpenStack sub-modules responsible for compute resources deployment and orchestration.

Table 8.14: IA-VI interface for OpenStack VIM

Action	Entity	URL	Method	Parameters	Returned code(s)
Create	Stack	{heat_url}/v1/{tenant_id}/stacks	POST	• heat_url: the address and port of the Heat server • tenant_id: the id of the OpenStack tenant requesting this instantiation • heat template: the descriptor of the stack to instantiate	• CREATED (201) • BAD REQUEST (400) • UN- AUTHORIZED (401) • CONFLICT (409)
Update	Stack	{heat_url}/v1/{tenant_id}/stacks/ {stack_name}/{stack_id}	PUT	• heat_url: the address and port of the Heat server • tenant_id: the id of the OpenStack tenant requesting this update • stack_name: the OpenStack name of the stack • stack_id: the OpenStack id of the stack • heat template: the updated descriptor for the stack	• ACCEPTED (202) • BAD REQUEST (400) • UN- AUTHORIZED (401) • NOT FOUND (404) • SERVER INTERNAL ERROR (500)

Action	Entity	URL	Method	Parameters	Returned code(s)
Query	Stack	{heat_url}/v1/{tenant_id}/stacks/ {stack_name}/{stack_id}	GET	• heat_url: the address and port of the Heat server • tenant_id: the id of the OpenStack tenant issuing the query • stack_name: the OpenStack name of the stack • stack_id: the OpenStack id of the stack	• OK (200) • BAD REQUEST (400) • UN- AUTHORIZED (401) • NOT FOUND (404) • INTERNAL SERVER ERROR (500)
Delete	Stack	{heat_url}/v1/{tenant_id}/stacks/ {stack_name}/{stack_id}	DELETE	• heat_url: the address and port of the Heat server • tenant_id: the id of the OpenStack tenant requesting this deletion • stack_name: the OpenStack name of the stack • stack_id: the OpenStack id of the stack	• NO CONTENT (204) • BAD REQUEST (400) • UN- AUTHORIZED (401) • NOT FOUND (404) • INTERNAL SERVER ERROR (500)
Query	Flavors	{nova_url}/flavors	GET	• nova_url: the address and port of the Nova server • tenant_id: the id of the OpenStack tenant	• NO CONTENT (200) • UN- AUTHORIZED (401) • FORBIDDEN (403)

Table 8.15 shows the details on the interface between the Infrastructure Abstraction layer and Ovs-Sfc, a custom agent for SFC enforcement based on Open vSwitch, that has been used alongside OpenStack as a proof of concept for inter-PoP network resource orchestration.

Table 8.15: IA-VI interface for OVS based SFC agent VIM

Action	Parameters	Description	Returned code(s)
add	{"instance_id":String, "in_segment":String, "out_segment":String, "port_list": [{"port":String, "order":int}]}	add a chain rule identified by "instance_id" for the traffic described by "in_segment"/"out_segment" connecting the ordered list of port provided in "port_list"	• SUCCESS • ERROR, {error_message}
delete	{"instance_id": String}	delete the chain rule identified by "instance_id"	• SUCCESS • ERROR, {error_message}

8.9 Ia-Wi Interface

Table 8.16 shows the details of the interface between the Ia-Wi interface, in the specific case of a WIM based on the Virtual Tenant Network [WikiBibliography#VTN] project of OpenDaylight.

Table 8.16: IA-Wi interface for VTN-based WIM

Action	Parameters	Description	Returned code(s)
configureWim	instance_id, in_segment, out_segment, [PoP]	configure the WAN to route traffic identified by in_segment/out_segment through the provided ordered list of PoP, for the service identified by instance_id	• SUCCESS • ERROR, {error_message}
deconfigureWim	instanceId	deconfigure the WAN for the service identified by instance_id	• SUCCESS • ERROR, {error_message}
updateWimConfiguration	instance_id, in_segment, out_segment, [PoP]	update the WAN configuration to route traffic identified by in_segment/out_segment through the provided ordered list of PoP, for the service identified by instance_id	• SUCCESS • ERROR, {error_message}

8.10 Gatekeeper - Service Lifecycle Manager Interface

Table 8.17 shows the interface between the GK and the SLM. For all message based publish/subscribe interfaces, the message properties are set as follows. All request messages need the following properties: 1) correlation_id that contains a UUID, 2) reply_to that contains the topic on which the response should be published, 3) app_id that contains the sender and 4) content_type that describes the format of the payload. All response messages need the following properties: 1) correlation_id that contains a UUID, 2) app_id that contains the sender and 3) content_type that describes the format of the payload. All notification messages need the following properties: 1) app_id that contains the sender and 2) content_type that describes the format of the payload.

The response to a service create request is split into two parts. First, a direct response is given to indicate whether the instantiation of the service has began or not. A notification is sent when the service is operational. The SLM offers 4 options to the GK: 1) create a service, 2) pause a running service, 3) restart a paused service and 4) terminate a service.

Table 8.17: Request Interface between the GK and the SLM.

Type	Topic	Sender	Receiver	Body
Request	service.instances.create	GK	SLM	• NSD • VNFD
Response	service.instances.create	SLM	GK	• status • error • timestamp
Notification	service.instances.create	SLM	GK	• the instance id of the service • status • error • timestamp
Request	service.instances.update	GK	SLM	• updated NSD • updated VNFD
Response	service.instances.update	SLM	GK	• status • error • timestamp
request	service.instance.pause	GK	SLM	• service_id
response	service.instance.pause	SLM	GK	• status • error • timestamp

Type	Topic	Sender	Receiver	Body
request	service.instance.restart	GK	SLM	• service_id
response	service.instance.restart	SLM	GK	• status • error • timestamp
request	service.instance.terminate	GK	SLM	• service_id
response	service.instance.terminate	SLM	GK	• status • error • timestamp

8.11 Service Lifecycle Manager - Function Lifecycle Manager

Table 8.18 describes the interface between the SLM and the FLM. This interface is a MANO framework internal interface that is used for the deployment of a new service. The SLM breaks down the service request into individual VNF deployment requests, which are handled by the FLM.

Table 8.18: Request Interface between the SLM and the FLM.

Type	Topic	Sender	Receiver	Body
request	mano.function.deploy	SLM	FLM	• the instance id of the VNF • VNFD • the calculated placement of the VNF
response	mano.function.deploy	FLM	SLM	• status • error • VNFR_id

8.11.0.1 Function Lifecycle Manager - Infrastructure Adaptor

Table 8.19 shows the interface between the FLM and the IA. The FLM requests specific VNF deployment from the IA.

Table 8.19: Request Interface between the FLM and the IA.

Type	Topic	Sender	Receiver	Body
request	infrastructure.function.deploy	FLM	IA	• the instance id of the VNF • VNFD • the pop on which the VNF must be placed
response	infrastructure.function.deploy	IA	FLM	• status • error • info intended for the VNFR, that is build by the FLM

8.12 Service Lifecycle Manager - Infrastructure Adaptor Interface

Table 8.20 shows the interface between the SLM and the IA. The SLM manages service related request from the IA, such as creating the service graph, instructing to pause/restart/terminate a service, configuring the WIM and collecting topology information for placement purposes.

Table 8.20: Request Interface between the SLM and the IA.

Type	Topic	Sender	Receiver	Body
request	infrastructure.management.compute.list	SLM	IA	• None
response	infrastructure.management.compute.list	IA	SLM	• the PoP topology, with free/total available resource information
request	infrastructure.service.prepare	SLM	IA	• list of PoPs that will be used by the service
response	infrastructure.service.prepare	IA	SLM	• status • error
request	infrastructure.service.chain	SLM	IA	• The service graph that indicates how the VNFs must be chained • VNF instance ids

Type	Topic	Sender	Receiver	Body
response	infarstructure.service.chain	IA	SLM	• status • error • info for the NSR, that is built by the SLM
request	infrastructure.wan.configure	SLM	IA	• the instance id of the service
response	infrastructure.wan.configure	IA	SLM	• status • error
request	infrastructure.wan.deconfigure	SLM	IA	• the instance id of the service
response	infrastructure.wan.deconfigure	IA	SLM	• status • error
request	infrastructure.service.restart	SLM	IA	• the instance id of the service
response	infrastructure.service.restart	IA	SLM	• status • error
request	infrastructure.service.pause	SLM	IA	• the instance id of the service
response	infrastructure.service.pause	IA	SLM	• status • error
request	infrastructure.service.terminate	SLM	IA	• the instance id of the service
response	infrastructure.service.terminate	IA	SLM	• status • error

8.13 Interfaces relevant for Function-/Service-Specific Managers

Table 8.21 shows the interfaces between SLM and executive plugins responsible for SSMs, as well as FLMs and executive plugins responsible for FSMs and Table 8.22 describes the interfaces between SSMs/FSMs and their corresponding executive plugins.

Table 8.21: Interface between SLM/FLM and executive plugins.

Type	Topic	Sender	Receiver	Body
request	placement.executive.request	SLM	Placement Executive Plugin	• UUID of the service instance • NSD • VNFD • Network topology • Network resources
response	placement.executive.request	Placement Executive Plugin	SLM	• Placement decision, including mapping VNFs to PoPs and mapping virtual links to inter-PoP paths
request	scaling.executive.request	FLM	Scaling Executive Plugin	• UUID of the VNF instance • NSD • VNFD • Scaling trigger
response	scaling.executive.request	Scaling Executive Plugin	FLM	• Scaling decision, including modified resource demands of the VNF and new instances of the VNF to be inserted to the service graph

Table 8.22: Interface between executive plugins and SSMs/FSMs.

Type	Topic	Sender	Receiver	Body
request	placement.ssm.{service_uuid}	Placement Executive Plugin	Placement SSM	• NSD • VNFD • Filtered network topology • Filtered network resources
response	placement.ssm.{service_uuid}	Placement SSM	Placement Executive Plugin	• Placement decision, including mapping VNFs to PoPs and mapping virtual links to inter-PoP paths
request	scaling.fsm.{vnf_uuid}	Scaling Executive Plugin	Scaling FSM	• NSD • VNFD • Filtered scaling trigger
response	scaling.fsm.{vnf_uuid}	Scaling FSM	Scaling Executive Plugin	• Scaling decision, including modified resource demands of the VNF and new instances of the VNF to be inserted to the service graph

Table 8.23 shows the interfaces between FLM/SLM and the SMR and Table 8.24 describes the interfaces between FSMs/SSMs and the SMR.

Table 8.23: Interface between SLM/FLM and SMR.

Type	Topic	Sender	Receiver	Body
request	specific.manager.registry.ssm.on-board	SLM	SMR	• NSD
response	specific.manager.registry.ssm.on-board	SMR	SLM	• status • error
request	specific.manager.registry.fsm.on-board	FLM	SMR	• VNFD

Type	Topic	Sender	Receiver	Body
response	specific.manager.registry.fsm.on-board	SMR	FLM	• status • error
request	specific.manager.registry.ssm.instantiate	SLM	SMR	• NSD • NSR
response	specific.manager.registry.ssm.instantiate	SMR	SLM	• status • error
request	specific.manager.registry.fsm.instantiate	FLM	SMR	• VNFD • VNFR
response	specific.manager.registry.fsm.instantiate	SMR	FLM	• status • error
request	specific.manager.registry.ssm.update	SLM	SMR	• NSD • NSR • VNFR
response	specific.manager.registry.ssm.update	SMR	SLM	• status • error
request	specific.manager.registry.fsm.update	FLM	SMR	• VNFR
response	specific.manager.registry.fsm.update	SMR	FLM	• status • error

Table 8.24: Interface between SSMs/FSMs and SMR.

Type	Topic	Sender	Receiver	Body
request	specific.manager.registry.ssm.registration	SSM	SMR	• name • version • description

Type	Topic	Sender	Receiver	Body
response	specific.manager.registry.ssm.registration	SMR	SSM	• status • name • version • description • UUID • error
request	specific.manager.registry.fsm.registration	FSM	SMR	• name • version • description
response	specific.manager.registry.fsm.registration	SMR	FSM	• status • name • version • description • UUID • error

8.14 Plugin Manager - Plugin Interface

Table 8.25 shows the interface between the Plugin Manager (PM) and the Plugin (P).

Table 8.25: Request Interface between the SDK and the GK.

Type	Topic	Sender	Receiver	Body
request	platform.management.plugin.register	plugin	PM	• name • version • description
response	platform.management.plugin.register	PM	plugin	• status • uuid • error
request	platform.management.plugin.deregister	plugin	PM	• uuid

response	platform.management.plugin.deregister	PM	plugin	• status
notification	platform.management.plugin.status	PM	plugin(s)	• timestamp • plugin_dict
notification	platform.management.plugin .{plugin_uuid}.heartbeat	plugin	PM	• uuid • state
notification	platform.management.plugin .{plugin_uuid}.lifecycle.start	PM	plugin	• null
notification	platform.management.plugin .{plugin_uuid}.lifecycle.pause	PM	plugin	• null
notification	platform.management.plugin .{plugin_uuid}.lifecycle.resume	PM	plugin	• null
notification	platform.management.plugin .{plugin_uuid}.lifecycle.stop	PM	plugin	• null

9 Conclusions

This section lists the conclusions on this deliverable, which documents the work done in SONATA's Work Package 4, **Resource Orchestration and Operations repositories** since the previous deliverable (D4.1: Orchestrator Prototype [7]) and the first year review of the project.

According to what we had planned for the second year, we are addressing the **Service Platform's security** across several levels in this second year. These levels are the access to the external APIs of the platform, the authorisation for every microservice that implements some platform features, the storage of passwords internally, etc.

The main changes on the **Gatekeeper**, the entry point of the Service Platform, was the increase in the security level adopted, by the usage of HTTPS in all the external APIs and the need to authenticate every platform user and authorise every microservice provided as part of the platform. Along this path, the platform has also gained a Licence Management service and a KPIs Management service, which will allow the monetisation of the APIs made available and the usage measurement of the provided services.

In the platform's **Catalogues and Repositories** we have started storing both the package files and its descriptors, with meta-data such as its creation date, signature, etc., and data, i.e., its descriptors, separately. Merging both in the first year's implementation was the most simple solution at the time, but the concept had to evolve from there naturally, to better accommodate new features, such as package signing.

The **MANO Framework**, the core of SONATA's Service Platform, started to be able to support Function Specific Managers, together with the already implemented Service Specific Managers, with their secure usage guaranteed by the Executive Plugins connecting them to the other components to/from which they provide/require services. The existing Service Lifecycle Manager and the newly introduced Function Lifecycle Manager (FLM), will be (re-)implemented as workflow (i.e., task execution) engines, increasing their flexibility. We will have an alternative implementation of the FLM, based on the OpenStack Mistral project.

The platform's **Infrastructure Abstraction** layer is being put to test since the project decided to adopt a container-based VIM such as Kubernetes, in parallel with OpenStack, the more traditional VM-based we have opted for in the first year. This is another innovation we are pursuing, and thus only the first results are documented in this deliverable. Further results will appear in the next deliverable of this work package.

Monitoring the kind of Service Platform we are designing and implementing poses a different set of challenges that traditional and barely existing products and services in this area already answer, namely at the flexibility needed to define new and distinct monitoring parameters for each on-boarded function and service. This greater flexibility is introduced in this second year version of the platform, together with an innovative way of securely providing the service or function developer, for a short time, monitoring data about one of the instances of his/her service or function. This feature allows the automation of such kind of requests, thus increasing the operational efficiency of the platform.

To make evident the highly modular design of the Service Platform, each one of its **Internal Interfaces** is described and documented. Most of these interfaces are implemented over HTTP(S), with one exception: passing monitoring data from the Service Platform to the SDK, which uses the more adequate Web Sockets. Some of the interfaces using HTTP(S) are synchronous and others asynchronous, reflecting the nature of the problem they solve. All these options are according to the most recent trends and technologies in the open-source sector, thus lowering the barrier to adopt from our community.

A Abbreviations

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

CM Configuration Management

CRUD Create, Read, Update, Delete

DSL Domain-Specific Language

ETSI European Telecommunications Standards Institute

FLM Function Lifecycle Manager

FSM Function-Specific Manager

FSMD Function-Specific Manager Descriptor

GitHub Git repository hosting service

GUI Graphical User Interface

IA Infrastructure Adaptor

IaaS Infrastructure as a Service

IDE Integrated Development Environment

IoT Internet of Things

JMS Java Messaging System

JWT JSON Web Token

KPI Key Performance Indicator

MANO Management and Orchestration

NF Network Function

NFV Network Function Virtualization

NFVI-PoP Network Function Virtualisation Points of Presence

NFVO Network Function Virtualization Orchestrator

NFVRG Network Function Virtualization Research Group

NS Network Service

NSD	Network Service Descriptor
NSO	Network Service Orchestrator
OASIS	Organization for the Advancement of Structured Information Standards
OIDC	OpenID Connect
OSS	Operations Support System
PD	Package Descriptor
PSA	Personal Security Applications
REST	Representational State Transfer
RPC	Remote Procedure Call
SDK	Software Development Kit
SDN	Software-Defined Networking or Software-Defined Network
SLA	Service Level Agreement
SLM	Service Lifecycle Manager
SNMP	Simple Network Management Protocol
SP	Service Platform
SSM	Service-Specific Manager
SSMD	Service-Specific Manager Descriptor
VDU	Virtual Deployment Unit
VIM	Virtual Infrastructure Manager
VLD	Virtual Link Descriptor
VM	Virtual Machine
VN	Virtual Network
VNF	Virtual Network Function
VNFD	Virtual Network Function Descriptor
VNFFGD	VNF Forwarding Graph Descriptor
VNFM	Virtual Network Function Manager
WAN	Wide Area Network
WIM	Wide area network Infrastructure Manager

B Glossary

DevOps A term popularized since a series of conferences emphasizing a higher degree of communication between **Dev**elopers and **Op**eration**s**, those who deploy the developed applications.

Function-Specific Manager A function-specific manager is a small management program implemented by a service developer with the help of SONATA's SDK. It is executed by the SONATA service platform to manage individual network functions based on inputs, say monitoring data, specific to the network function it belongs to.

Gatekeeper In general, gatekeeping is the process through which information is filtered for dissemination, whether for publication, broadcasting, the Internet, or some other mode of communication. In SONATA, the gatekeeper is the central point of authentication and authorization of users and (external) Services.

Management and Orchestration (MANO) In the ETSI NFV framework ETSI-NFV-MANO, MANO is the global entity responsible for management and orchestration of NFV lifecycle.

Message Broker A message broker, or message bus, is an intermediary program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. Message brokers are elements in telecommunication networks where software applications communicate by exchanging formally-defined messages. Message brokers are a building block of Message oriented middleware.

Network Function The atomic entity of execution anything in the context of a service. Cannot be further subdivided. Runs as a single executing entity, such as a single process and a single virtual machine. Treated as atomic from the point of view of the orchestration framework.

Network Function Virtualization (NFV) The principle of separating network functions from the hardware they run on by using virtual hardware abstraction.

Network Function Virtualization Infrastructure Point of Presence (NFVI PoP) Any combination of virtualized compute, storage and network resources.

Network Function Virtualization Infrastructure (NFVI) Collection of NFVI PoPs under one orchestrator.

Network Service A network service is a composition of network functions.

Network Service Descriptor A manifest file that describes a network service. Usually, it consists of the description of the network functions in the server, the links between the functions, a service graph, and service specifications, like SLAs.

Resource Orchestrator (RO) Entity responsible for domain wide global orchestration of network services and software resource reservations in terms of network functions over the physical or virtual resources the RO owns. The domain an RO oversees may consist of slices of other domains.

Service-Specific Manager (SSM) A service-specific manager is a small management program implemented by a service developer with the help of SONATA's SDK. It is executed by the SONATA service platform to manage individual services based on inputs, say monitoring data, specific to the service it belongs to.

Service Level Agreement (SLA) A service-level agreement is a part of a standardized service contract where a service is formally defined.

Service Platform One of the key contributions of SONATA. Realizes management functionality to deploy, provision, manage, scale, and place service on the infrastructure. a service developer/operator can use SONATA's SDK to deploy a service on a selected service platform.

Slice A provider-created subset of virtual networking and compute resources, created from physical or virtual resources available to the (slice) provider.

Software Development Kit (SDK) A set of tools and utilities which help developers to create, monitor, manage, optimize network services. A key component of the SONATA system.

Virtualised Infrastructure Manager (VIM) provides computing and networking capabilities and deploys virtual machines.

Virtual Network Function (VNF) One or more virtual machines running different software and processes on top of industry-standard high-volume servers, switches and storage, or cloud computing infrastructure, and capable of implementing network functions traditionally implemented via custom hardware appliances and middleboxes (e.g. router, NAT, firewall, load balancer, etc.).

Virtualized Network Function Forwarding Graph (VNF FG) An ordered list of VNFs creating a service chain.

C Bibliography

[1] Auth0. Get started with json web tokens. Website, December 2016. Online at https://auth0.com/learn/json-web-tokens/.

[2] Kubernetes Community. Kubernetes. Website, November 2016. Online at http://kubernetes.io/.

[3] The OpenStack Community. Openstack mistral project. Website, Dec 2016. Online at https://wiki.openstack.org/wiki/Mistral.

[4] SONATA consortium. D2.2 architecture design. Website, December 2015. Online at http://www.sonata-nfv.eu/content/d22-architecture-design-0.

[5] SONATA consortium. D2.3 updated requirements and architecture design. Website, December 2016. Online at http://www.sonata-nfv.eu/.

[6] SONATA consortium. D3.2 sdk operational release and documentation. Website, December 2016. Online at http://www.sonata-nfv.eu/.

[7] SONATA consortium. D4.1: Orchestrator prototype. Website, May 2016. Online at http://www.sonata-nfv.eu/content/d41-orchestrator-prototype.

[8] Website, December 2016. Online at http://www.cryptographytools.com/index.jsf.

[9] Online at https://www.datadoghq.com/.

[10] Online at https://dev.mysql.com/doc/refman/5.5/en/encryption-functions.html.

[11] Linux Foundation. Documentation let's encrypt. Website, July 2016. Online at https://letsencrypt.org/docs/.

[12] Martin Fowler. Microservices. Website, 2014. Online at http://martinfowler.com/articles/microservices.html.

[13] Online at https://graphiteapp.org/.

[14] Online at http://md5-hash-online.waraxe.us/.

[15] Online at https://www.vaultproject.io/.

[16] IETF. Hmac: Keyed-hashing for message authentication. IETF, 1997. Online at https://tools.ietf.org/html/rfc2104.

[17] IETF. Web secure sockets. IETF, 2011. Online at https://tools.ietf.org/html/rfc6455.

[18] IETF. Web sockets. IETF, 2011. Online at https://tools.ietf.org/html/rfc6455.

[19] IETF. Json web token (jwt). IETF, 2015. Online at https://tools.ietf.org/html/rfc7519.

[20] IETF. Network service header. IETF, 2016. Online at https://datatracker.ietf.org/doc/draft-ietf-sfc-nsh/.

[21] Internet Engineering Task Force (IETF). The oauth 2.0 authorization framework. Website, October 2012. Online at https://tools.ietf.org/html/rfc6749#section-4.4.

[22] Online at https://www.influxdata.com/.

[23] Online at https://www.elastic.co/.

[24] Alan Klement. Replacing the user story with the job story, 2016. Online at http://jtbd.info/replacing-the-user-story-with-the-job-story-af7cdee10c27#.goyz3rc6a.

[25] Online at https://getkong.org/.

[26] Online at https://releases.hashicorp.com/vault/0.6.3/vault_0.6.3_linux_amd64.zip.

[27] Online at https://www.vaultproject.io/docs/secrets/mongodb/index.html.

[28] Online at https://www.vaultproject.io/docs/secrets/mysql/.

[29] OpenStack Neutron. Neutron serviceinsertionandchaining. website, 2016. Online at https://wiki.openstack.org/wiki/Neutron/ServiceInsertionAndChaining.

[30] nixCraft. generating random passwords. Website, 2013. Online at https://www.cyberciti.biz/faq/generating-random-password/.

[31] Online at https://wiki.opendaylight.org.

[32] Online at http://openid.net/connect/.

[33] Openssl. Website, 1999-2016. Online at https://www.openssl.org/.

[34] Online at https://parse.com/.

[35] Online at https://www.postgresql.org/docs/8.3/static/pgcrypto.html.

[36] Online at https://piwik.org/.

[37] Online at https://www.vaultproject.io/docs/secrets/postgresql/.

[38] Online at https://prometheus.io/.

[39] Online at https://tools.ietf.org/html/rfc5988#page-6.

[40] A. Shamir R.L. Rivest and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. MIT, 1977. Online at http://people.csail.mit.edu/rivest/Rsapaper.pdf.

[41] Online at http://saml.xml.org.

[42] SANS. "history of encryption". Website, December 2016. Online at https://www.sans.org/reading-room/whitepapers/vpns/history-encryption-730.