D4.1 Orchestrator Prototype

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	D4.1 Orchestrator Prototype
Workpackage	WP4 Resource Orchestration and Operations repositories
Due Date	Abril, 2016
Submission Date	May, 2016
Version	0.1
Status	Final
Editor	José Bonnet (AlticeLabs)
Contributors	José Bonnet (AlticeLabs), Santiago Rodríguez (Optare), Aurora Ramos, Felipe Vicens (ATOS), George Xilouris, Stavros Kolometsos, Christos Sakkas (NCSRD), Stuart Clayman, Alex Galis, Francesco Tusa, Dario Valocchi (UCL), Theodore Zahariadis, Panos Trakadas, Panos Karkazis, Sotiris Karachontzitis (SYN), Thomas Soenen (iMinds), Sharon Mendel-Brin (Nokia), Bruno Vidalenc (Thales), Michael Bredel (NEC), Muhammad Shuaib Siddiqui, Dani Guija (i2CAT), Manuel Peuster, Sevil Dräxler (UPB)
Reviewer(s)	Michael Bredel (NEC), Pedro Aranda (TID)

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)	

Executive Summary:

5G poses a number of challenges to our society, both technical and non-technical, of which the SONATA NFV project intends to answer the extreme need for **flexible programmability of 5G networks**. By pushing for a Digital Society and the general spread of new and disruptive Digital Services, Communication Service Providers foresee significantly lower barriers for third party service provider entrants to the usage of their infrastructure which was once a protected realm in the name of quality of service, business support, etc.

In this emerging context, Communication Service Providers need Service Platforms that are radically different from the ones they currently own or use, highly flexible, extensible and enable authorized usage, modification or even extension to this flexible behavior in order to support new business models and technological trends.

This deliverable describes the prototype version of SONATA Service Platform consisting of a modular and customisable **Orchestrator**.

The platform accepts (network) services and (virtual network) functions described in Yet Another Markup Language (YAML), which is a human-readable format that can be automatically validated by the use of a schema. We have adopted ETSI's way of describing these entities, but we are not at all locked to their current version. Different forms of describing these entities can be adopted in the future, just by changing their validators/interpreters.

We have addressed **flexibility** and **extensibility** by choosing three key technologies:

- a **plugin**-based architecture, supported by a **message-oriented middleware**: the platform will ship with a relevant set of such plugins, allowing an off-the-shelf usage, but it's behavior may be changed if different plugins are provided;
- a **micro-services**-based architecture, which extends the flexibility to parts of the system not needing a message broker (a plugin can be a micro-service, but not necessarily the other way around: micro-services must be kept very simple in terms of the service(s) they provide);
- an **infrastructure abstraction** which supports different implementations of the Virtual Infrastructure Manager (VIM).

To facilitate implementation of the platform, same **agile techniques** are used that are recommended for Developers developing and providing services to the platform -- namely considering **DevOps** as our way to develop and push forward the code that is to reach production along the deployment pipe-line. The complete Service Platform implementation is not yet accomplished, but its main functionality is already available in this prototype version.

Contents

List of Figures

List of Tables

1 Introduction

This deliverable is the first document related to the SONATA's Service Platform development to be published. It documents the early prototype for the Service Platform, after 10 months of work in the project, ready to be integrated with the prototype of the SDK (documented in [8]). The main focus of the first year prototype is to prioritise the needed features for a complete work cycle of the SONATA SP. This deliverable describes the design and implementation of those features.

1.1 The 5G challenges

The set of technologies commonly known as **5G** [16][17][2][1], and specially the **faster** internet supporting new and possibly disruptive services, the need for **faster** service deployment, etc., is having a tremendous impact in the current set of actors of our society. If a 'faster Internet' is the easiest to sell flavour of 5G, it will only happen if the technology and the businesses supporting it change accordingly as well. **Software Defined Networks** (SDN) and **Network Function Virtualisation** (NFV) are such technologies that, together with movements such as **Open-source Software**, not only accelerate the provisioning of communication services but also significantly lower the barrier to entry in those markets.

The SONATA Project addresses (some of) these challenges, in particular the technological ones that will allow a much faster time-to-market of services produced by a much wider number of third party service providers. On the Service Platform side, described in this deliverable, the challenge is addressed by building or adopting/adapting the most adequate tools to this scenario of faster deploying services that are provided by a much wider set of third party providers. These tools are further described below.

1.2 Service Platform Overview

The Service Platform Architecture has been extensively described in [6]. Figure 1.1 is repeated below for easier reading.

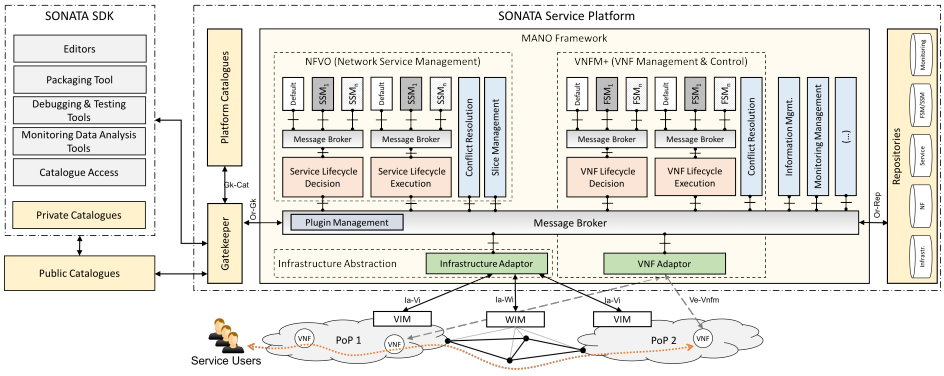

Figure 1.1: Detailed architecture of SONATA service platform

Before going in to implementation details of the first year prototype, it is important to overview the implemented features of different modules of the Orchestrator shown in Figure 1.1.

By significantly widening the number of service providers that will be able to deploy services in the Service Platform, the platform operator exposes itself and the supporting infrastructure to threats like never before. The **Gatekeeper** component has the responsibility to control which of those providers can effectively contribute with new services to the **Catalogue**. However, in the first prototype, the control is performed as the focus was to build a first version, like a Minimum Viable Product (see [20]). More features will be added in the next versions.

The extreme flexibility needed for the Service Platform, led us to consider a **plug-in** model, supported by a **message-based broker**. The platform's features can be extended by providing different **Service Specific Managers** (SSM) and **Function Specific Managers** (FSM) plugins that, by interpreting the defined messages in different ways and generating new ones change the default behaviour of the service platform in a controlled way.

Developing new plugins becomes the standard way of customizing the service platform and its behavior. We support the plugin development task by providing a plugin framework that implements abstract plugins. These plugins can be easily inherited and modified and hide most of the plugin management complexity, like plugin registration procedures. They allow developers to focus on their platform or service specific needs without bothering with platform internal mechanics. Our framework also offers an easy to use messaging API for topic-based synchronous and asynchronous request/reply communication via the message broker.

Monitoring is also a key feature of the first year prototype. There are many products and services allowing monitoring of a solution, both open-sourced and commercial. But monitoring cloud-based services poses challenges that add to the (big) 'size' dimension the existing solutions have. In our case, as explained above, the platform usually doesn't know what it is going to monitor until the service is on-boarded in the Catalogues (or even later, when it is instantiated). This flexibility as to what is to be monitored is not easily achievable, specially since high volumes of data are usually the case, and good analytical and visualisation performance is a requirement.

For the current version, only **OpenStack** was considered as an implementation of the **VIM** concept, but our intention is to make the Service Platform independent of the concrete VIM to be used. This is done by considering the **Infrastructure Abstraction** block, which will abstract other possible implementations for the VIM, or even other instances of the SONATA Service Platform infrastructure.

Last, but not the least, we have adopted the **DevOps** and **Agile** way of developing the Service Platform itself, from the Developer to the Service Platform, and back to the Developer again with monitoring data, as we prescribe the SONATA users to do. All the mechanisms supporting this are still being defined (e.g., deployment for now is made only to the Integration environment, and not to production), but the whole team is evolving pretty fast on this track. Further details on this part of the project are presented in [9].

1.3 Deliverable content at a glance

The content of this deliverable is organised as described next.

The current section is the **Introduction**, where the whole scenario is defined. In section Section 2, **Gatekeeper** is detailed, namely the features that are already supported (future modules/micro-services are just lightly mentioned). Section 3 details the **Orchestrator kernel**, namely the message broker and how anyone can interact with it, both from the perspective of developing plugins for it or somehow simply interact with it. Section 4 details the design work that started for the specific service and function managers plugins. The next section, Section 5, covers **Catalogues,**

Repositories, and Supporting Functionalities of the previously mentioned components. In Section 6, we highlight the aspects of the **Year One Storyboard** that have been implemented. Finally, we draw some **Conclusions** from this first year of work in Section 7.

Appendixes hold the following content. Appendix A, shows the **Schema** used to validate the descriptors of **packages**, **services** and **functions**. Both these Annexes are shared with [8]. Finally, in Appendix D relevant Message Broker messages are listed.

2 Gatekeeper

The SONATA Service Platform Gatekeeper is the component that controls the access to the features of the Service Platform.

We start by presenting a 'bird's eye' view of the main processes and entities interacting with the Gatekeeper and explaining them. Some concepts that have guided us in the design and implementation of this sub-system are also presented. Details about the already implemented features are given later.

As the Gatekeeper was specified and designed, other (sub-)components came to relevance. These components are: the Graphical User Interface (GUI, see Section 2.5) and the Business Support Systems (BSS, see Section 2.6).

2.1 Concepts and Guidelines

This first sub-section presents the high level processes the Gatekeeper will have to support, as well as the involved entities. Figure 2.1 shows theses processes and entities[1].

A summary of these high-level components and processes is given below. These elements and processes influence the features and architecture further detailed in the next pages.

2.1.1 Elements

The picture above has the following elements:

- **End User:** those who may 'buy' a service from the owner of the Service Platform;
- **Developer:** those who develop Network Services and deploy them the Service Platform;
- **Admin:** those users who have the credentials to administrate the Service Platform, by having access to more information (usually configuration) than the common user;
- **User Mng.:** the module that will encapsulate all the authentication and authorisation of the Gatekeeper's users;
- **Package Mng.:** the module that encapsulates the details of managing packages received from the SDK;
- **License Mng.:** the module that will encapsulate all the licensing management of the packages submitted and/or requested from the Service Platform;
- **KPIs:** the module that will encapsulate all the key performance indicators of the Service Platform;
- **GK's GUI:** the module that shows in graphical form the information the Gatekeeper has;
- **GK's API:** the module that makes available the features the Gatekeeper provides;

[1]Note that in this prototype not all of these processes have been implemented.

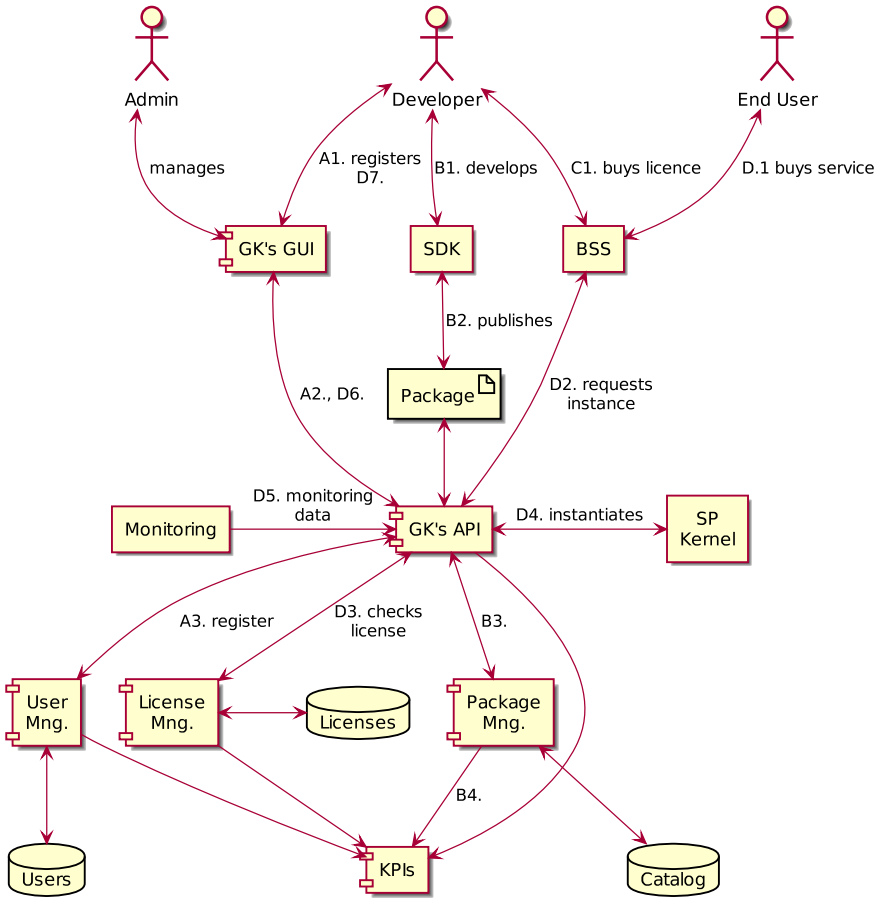

Figure 2.1: High level Gatekeeper processes and entities

- **SDK:** the module that allows the Developer to generate and consume packages to/from the Service Platform;
- **BSS:** the module that is a surrogate for all the 'business' (therefore the name, Business Support System) there can be with the Service Platform;
- **SP Kernel:** the module abstracting all the behaviour of the Service Platform in terms of its kernel;
- **Monitoring:** the module that allows monitoring of the SP;
- **Users:** the users database;
- **Catalog:** the catalogue database;
- **Licenses:** the license database;
- **Package:** the item exchanged between the SDK and the Service Platform.

2.1.2 Processes

The main, high-level processes are the following:

- **A - A developer registers in the Service Platform:** this allows the Service Platform owner to control who accesses its platform;
- **B - A registered developer publishes a new package:** a registered developer may submit packages with services, maybe re-using already available functions;
- **C - A registered developer buys a license to use a function:** some functions may need to be bought in order to be used/included in a service;
- **D - A customer buys a service:** available services maybe bought by customers, thus monetising the eco-system.

2.1.3 Architectural guidelines

This sub-section describes the common guidelines according to which the Gatekeeper Core components have been designed and implemented.

2.1.3.1 REST and micro-services

After a very successful experience in using this form of resource organisation/exposure in T-NOVA [10], we have opted to repeat this architecture flavour in this project.

REST [13] stands for **Representational State Transfer**, an " '[...] architectural style for distributed hypermedia systems [...]' ". By using this architectural style, a system is organised in **resources**, which are univocally represented by a **Uniform Resource Identifier** (URI). These resources may then be manipulated through their representation, in a **stateless** way (the state of the resource may be contained in part of the URI, as 'query parameters', in the body or in the headers of the request). Client and Server systems designed in this way exchange messages on top of the **Hyper-Text Transport Protocol** (HTTP), thus taking advantage of its statelessness and well known possible methods[2]:

[2]There are five more HTTP methods, PATCH, HEAD, OPTIONS, TRACE and CONNECT, but we're not using them.

- **POST** to create a resource;
- **GET** to read a resource;
- **PUT** to update a resource as a whole;
- **DELETE** to delete a resource;

This REST-based architecture supports really well another recent trend (and also followed in T-NOVA [10]), called **micro-services** [14]. A micro-service is an " '[...]independently deployable, small, modular service [...] that runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal [...]' "[15].

2.1.3.2 Feature specification

Features are described in the following **User Story** [3] format:

As a <role>, **I want to** <feature>, **so that** <benefit>

These user stories have been written from the Service Platform's perspective. Therefore the **role** segment is often omitted, and assumed as the **Service Platform user**, unless stated otherwise.

2.1.3.3 Reference architecture

A reference architecture for every micro-service used for the implementation of the Gatekeeper's Core components is shown in Figure 2.2.

The **Northbound Interface** (NBI) is the only entry point of this kind of micro-services, where the micro-services' clients request the services. This NBI then relies on a **Router** component for a first level validation (some parameters, HTTP headers, etc.), which then requires services from the set of **Model** classes found to be the most adequate to model the problem at hand. These model classes may themselves need services provided by other micro-services, which are required through a **Southbound Interface** (SBI). For the simplest interactions (e.g., returning the supported API), a router can implement the logic without a model.

For a concrete example of applying this please refer to Section 2.2.

2.2 Gatekeeper API

In the SONATA's Service Platform the Gatekeeper API plays the role of a *facade* [12] both for the Gatekeeper's modules and of most the the Service Platform, both allowing for a tight control of who can access what and simplifying the access of the systems that want to use the Platform's services.

2.2.1 Features

This sub-section describes the features expected from the Gatekeeper API. For simplification, all features are expressed in just one user story. Additional user stories will be written, designed, implemented, tested and included in farther versions of the Service Platform.

Accepts new packages: **As a** Platform Owner, **I want to** be able to accept and store new packages, **so that** I can have them available in the Service Platform's Catalogue(s). In the current version all developers may submit a package.

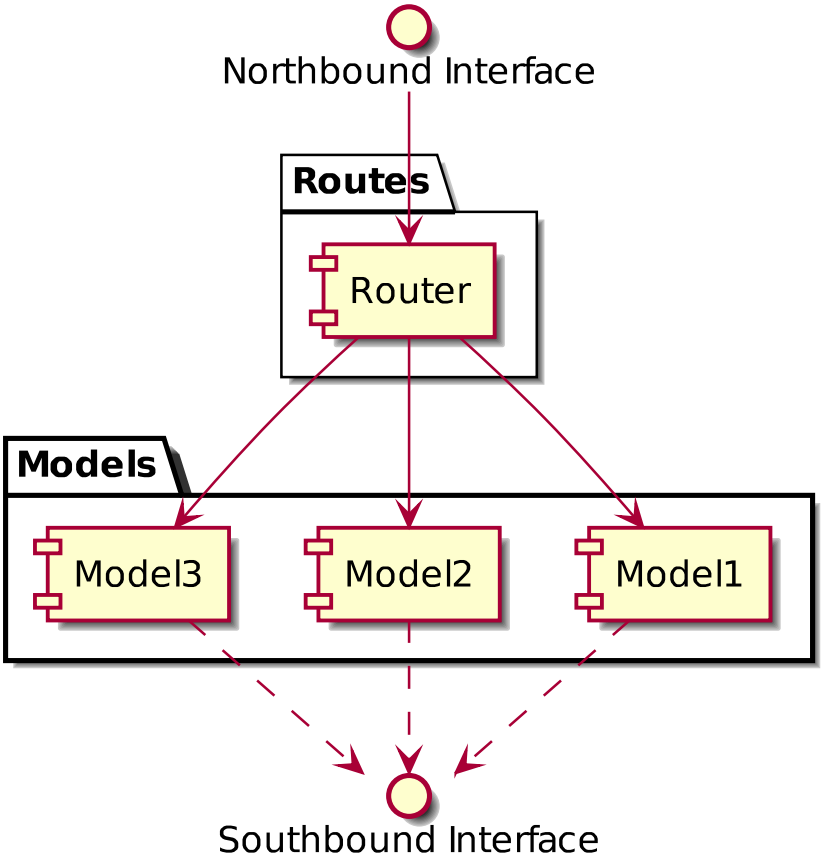

Figure 2.2: The Gatekeeper's reference architecture for micro-services

Provides a list of existing packages: **As a** Platform Owner, **I want to** be able to provide a list of registered packages, **so that** Developers can see and use some of the services or functions those packages describe. In the current version all developers will have access to all packages. In later versions, with the introduction of the **User Management** module, the list of packages will be restricted to the ones the developer retrieving it is allowed (*authorised*) to see.

Provides a package: **As a** Platform Owner, **I want to** provide a package from the Service Platform's Catalogue, **so that** Developers can use it. In the current version all developers can retrieve any package.

Provides a list of services that can be instantiated: **As a** Platform Owner, **I want to** be able to provide a list of services, **so that** the BSS can see them.

Accepts requests for the instantiation of a service: **As a** Platform Owner, **I want to** be able to accept requests for the instantiation of a specific service, **so that** an End User can benefit from that service.

Accepts requests for the status of a previous service instantiation requests: **As a** Platform Owner, **I want to** be able to provide the status of a previously submitted instantiation request, **so that** the BSS can know when the requested instantiation can be given as ready to the End-User.

Provides a list of supported paths in the API: **As a** Platform Owner, **I want to** be able to provide a list of implemented paths to the services I provide, **so that** a Developer can use it.

Provides OpenAPI formatted list of supported paths in the API: **As a** Platform Owner, **I want to** be able to provide a list of implemented paths to the services I provide in OpenAPI format, **so that** a Developer can use it. We have chosen OpenAPI, 'fka' Swagger, to document our APIs. The effort of maintaining several sources of this API or converting between them is still under evaluation.

Provides access to the platform's logs: **As a** Platform Owner, **I want to** be able to provide access to the Platform's logs, **so that** a platform developer or administrator can use it.

2.2.2 API

This sub-section describes the (external) API of Gatekeeper API, which offers the endpoints shown in Table Table 2.1.

Table 2.1: The Gatekeeper's REST API

Endpoint	Method	Description	Returned code(s)
`/`	GET	Receive a list of implemented endpoints like this one	Ok (200)
`/api-doc`	GET	Receive an OpenAPI (HTML)-formatted page of the implemented endpoints	Ok (200)
`/packages`	POST	Submits a new package to the Gatekeeper. The format of the file s described by the `son-schema` defined, in Appendix A. In this version, the package file is discarded, with its relevant data stored in the Catalogues. In later versions the file will be preserved.	Created (201)

Endpoint	Method	Description	Returned code(s)
	GET	Retrieve a list of package data currently registered in the system. If the result of the query in a single package, a package file is returned (like if it was requested as `GET /packages/:uuid`). It supports pagination with the `offset` (default value zero) and `limit` (default value ten) parameters.	Ok (200), Not found (404)
`/packages/:uuid`	GET	Retrieve a package file in the `son-schema` format, who's id is the given `uuid`. In this version the package is rebuilt with all the information existing in the Catalogues. In later versions the original file will be returned.	Ok (200), Not found (404)
`/services`	GET	Retrieve a list of services. Supports queries like `/services?status=Active` to only retrieve services with certain characteristics (in this example only `Active` services will be returned. Supports restrictions on which fields should be returned with `/services?fields=uuid,name,status`. Supports pagination with the `offset` (default value zero) and `limit` (default value ten) parameters.	Ok (200), Not found (404)
`/requests`	POST	Submits a new service instantiation request, with the `service_uuid` in the body of the request. Returns the data created, including the `request uuid` to be used later.	Created (201), Not found (404)
	GET	Retrieves a list of requests. Supports queries like `/requests?service_uuid=:service_uuid` to only retrieve requests with certain characteristics (in this example, requests made for the given `service_uuid`). Supports pagination with the `offset` (default value zero) and `limit` (default value ten) parameters.	Ok (200)
`/requests/:uuid`	GET	Retrievse data associated with the request who's id is `uuid`.	Ok (200), Not found (404)
`/admin/logs`	GET	Retrieves the API log file.	Ok (200)
`/admin/packages/logs`	GET	Retrieves the Package Management micro-service log file.	Ok (200)
`/admin/services/logs`	GET	Retrieves the Service Management micro-service log file.	Ok (200)

2.2.3 Internal Architecture

Figure 2.3 shows the high-level architecture of SONATA's Gatekeeper API. It shows the unique access point to the whole API and how it uses different models to access the other micro-services.

In this concrete example, the '/' and the '/api-doc' do not need any 'model' to perform what's requested [11], the 'Package Management' model accesses the 'Package Management Service' (see Section 2.3) and both the 'Service' and the 'Request' models access the 'Service Management Service' (see Section 2.4).

2.2.4 Message Sequence Charts

This sub-section presents the MSCs where the Gatekeeper API plays a role.

The following message sequence charts were omitted, due to it's simplicity:

- Provides a list of supported paths in the API
- Provides OpenAPI formatted list of supported paths in the API

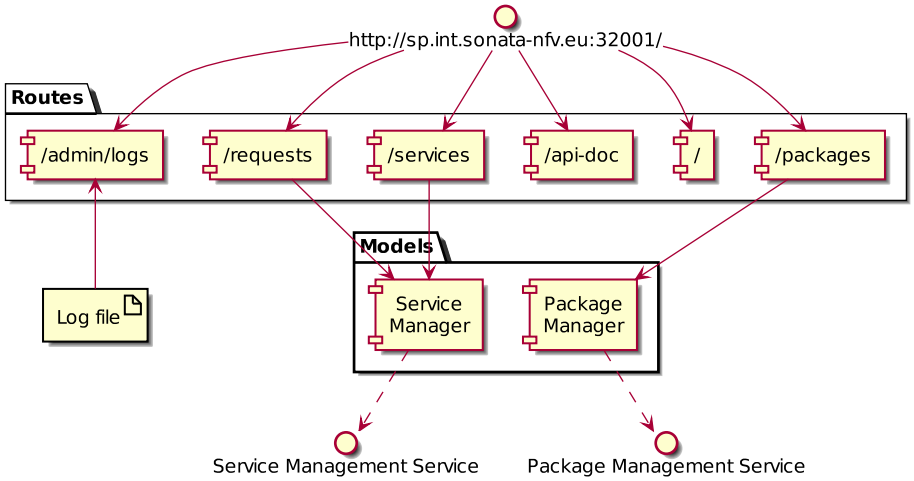

Figure 2.3: Gatekeeper's API high level architecture

Accepts new packages Figure 2.4 shows the sequence diagram for accepting a new package.

Provides a list of existing packages Figure 2.5 shows the sequence diagram for accepting a request for the list of existing packages.

The possibility of returning the package file when the query results in only one package covers the scenario of fetching a package by a trio of fields that uniquely identify it: `vendor`, `name` and `version`.

Provides a package Figure 2.6 shows the sequence diagram for the request for an existing package.

Provides a list of services that can be instantiated Figure 2.7 shows the sequence diagram for listing existing services.

Accepts requests for the instantiation of a service Figure 2.8 shows the sequence diagram for instantiating a specific service.

Accepts requests for the status of a previous service instantiation requests Figure 2.9 shows the sequence diagram for the status of a previous instantiation request.

2.2.5 Technologies used

Table 2.2 lists the technologies used in the implementation of the Gatekeeper API.

Table 2.2: Technologies used in the Gatekeeper's API component

Name	Type	Purpose
addressable	library	interpreter of query parameters in the URL
ci_reporter_rspec	library	to be used by **rubocop**
foreman	library	manager of multiple processes
puma	library	application server

Name	Type	Purpose
rack-parser	library	used to parse HTTP requests
rack-test	library	used in tests
rake	library	dependency manager
rest-client	library	REST client
rspec	framework	tests framework
rspec-its	library	to be used by **rspec**
rubocop	library	for styling checking
rubocop-checkstyle_formatter	library	to be used by **rubocop**
ruby	programming language	programming language
rubyzip	library	to manipulate package files
sinatra	framework	web application framework
sinatra-contrib	library	add-ons for the **sinatra** web-app framework
sinatra-cross_origin	library	add-ons for the **sinatra** web-app framework (cross-origin calls to services)
sinatra-logger	library	add-ons for the **sinatra** web-app framework (logging)
web mock	library	for mocking external services

2.2.6 Tests

This sub-section describes tests designed and implemented for the Gatekeeper's API.

2.2.6.1 Unit tests

We are restricting unit tests of the API to the access of the root path (`/`) and the API documentation in HTML (`/api-doc`)

1. List API methods by accessing the root:
   - **Execution:** `curl localhost:5000/`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of methods available
2. List API methods in HTML by accessing the `/api-doc`:
   - **Execution:** `curl localhost:5000/api-doc`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of methods available in HTML format

Further unit test are planned, namely covering the `offset` and `limit` parameters for paginating extensive lists of results.

2.2.6.2 Module Tests

Integration tests considered are the ones that join together the Gatekeeper's API and the 'Package Management Service' (see Section 2.3) and the 'Service Management Service' (see Section 2.4), with a 'mocked' Catalogues.

1. **Packages:**
   a) submit a valid package:
      - **Set-up:** have a valid package (`sonata-demo.son`, in the current folder)
      - **Execution:** `curl -X POST -F "package=@sonata-demo.son" \`

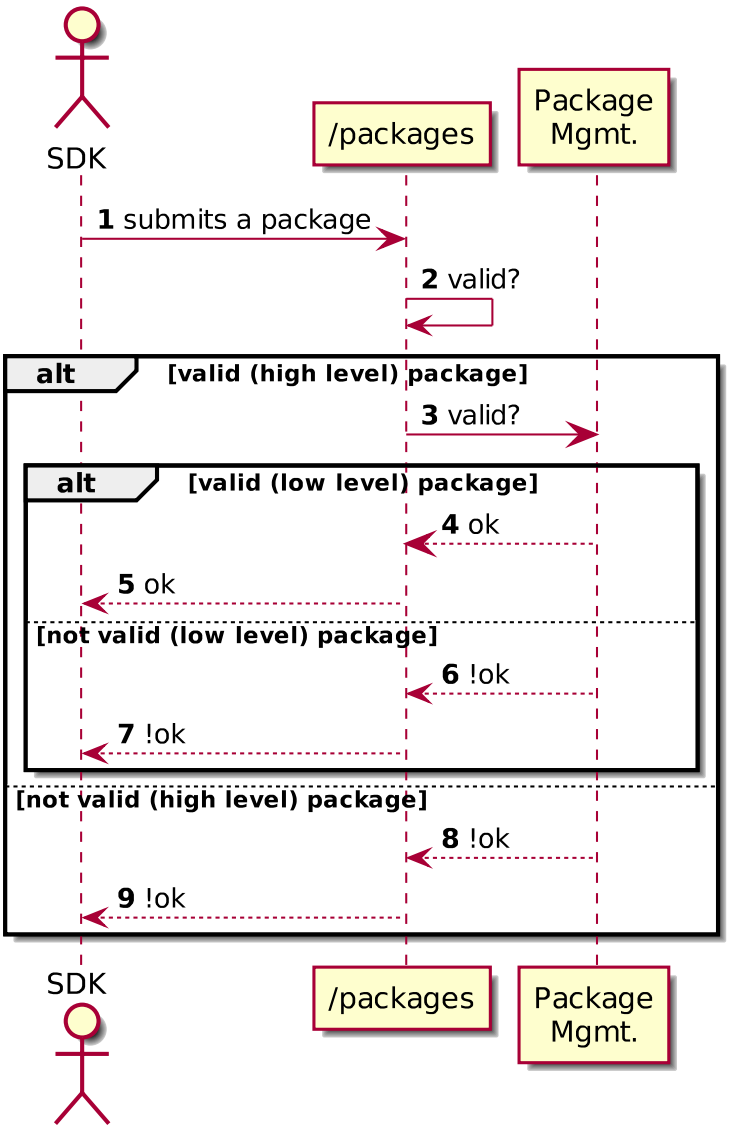

Figure 2.4: The Gatekeeper accepts a new package from the SDK

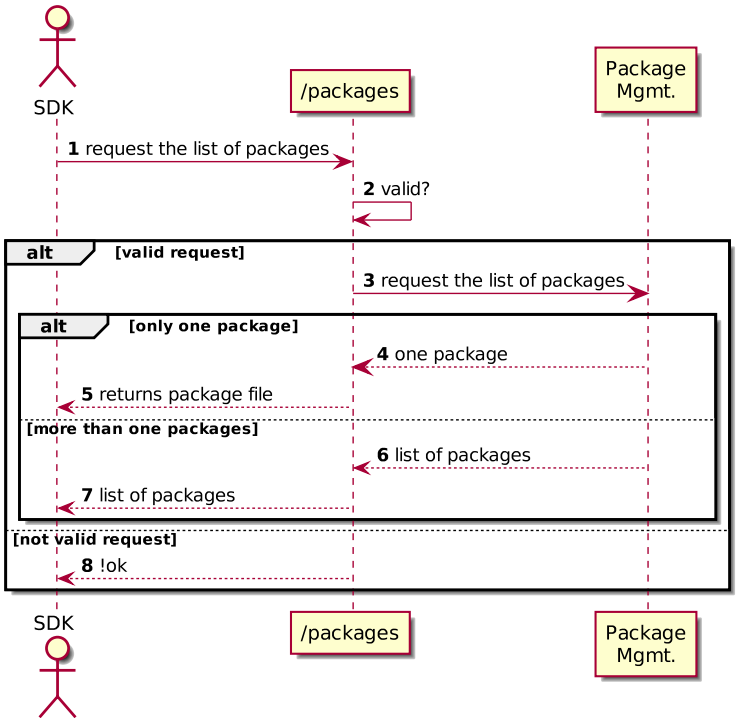

Figure 2.5: The Gatekeeper accepts a request for listing existing packages

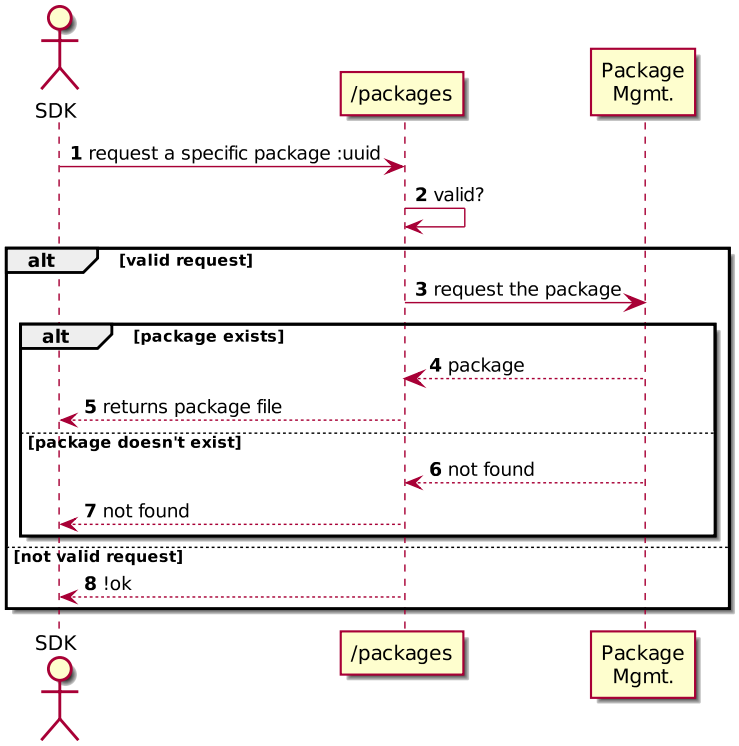

Figure 2.6: The Gatekeeper accepts a request for a specific packages

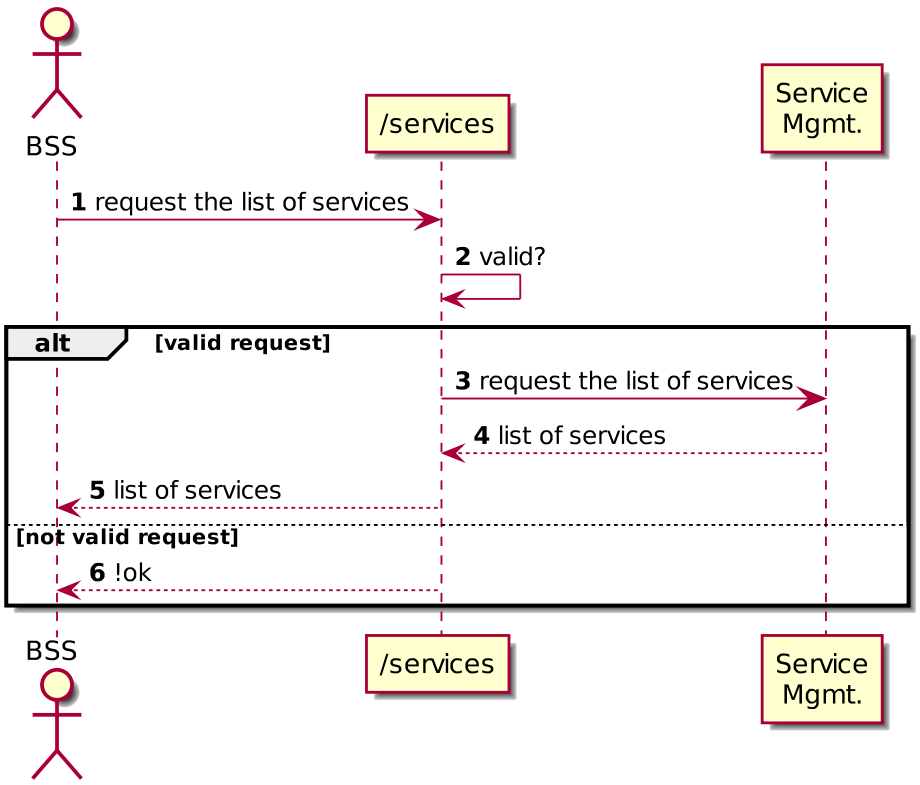

Figure 2.7: The Gatekeeper accepts a request for listing existing services

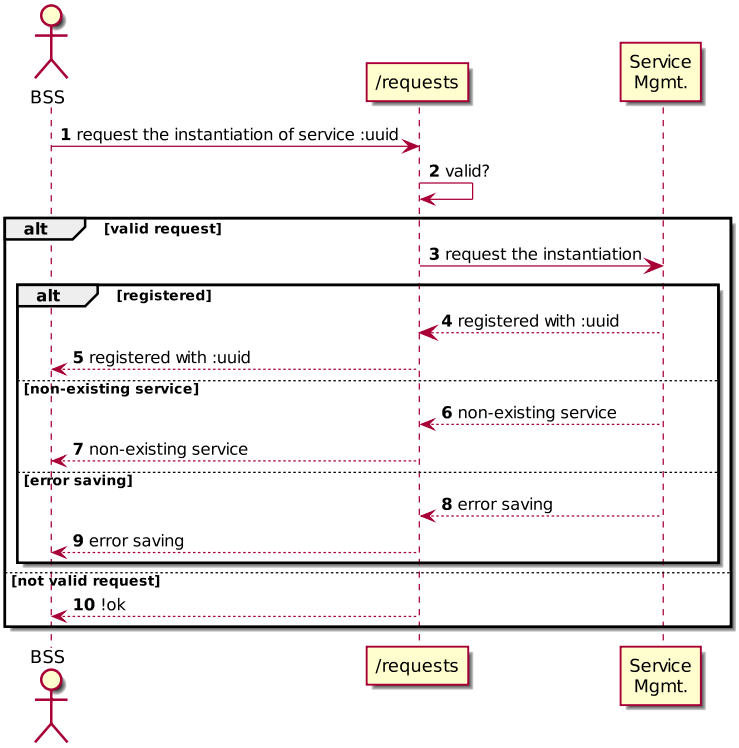

Figure 2.8: The Gatekeeper accepts a request for instantiating a specific service

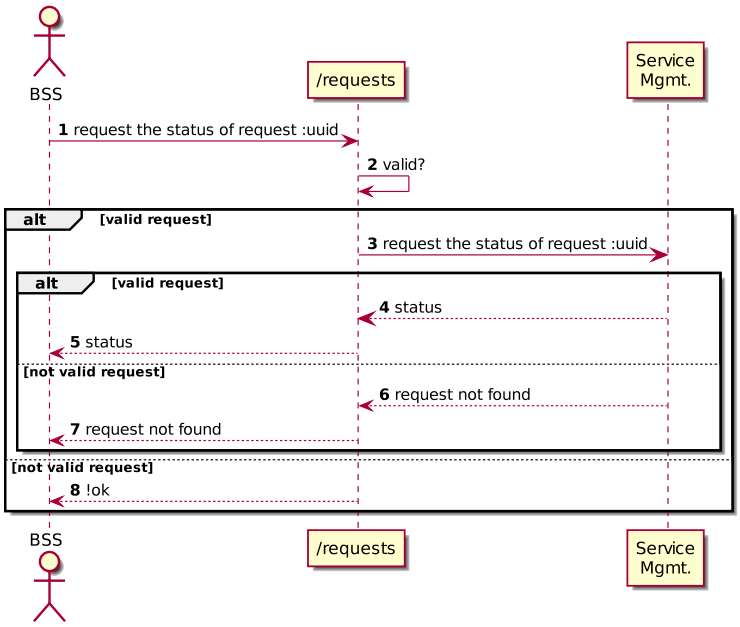

Figure 2.9: The Gatekeeper accepts a request for the status of a previous instantiation request

`localhost:5000/packages`

- **Expected:** HTTP code 201 (Created) is returned, together with the JSON representation of the created package;
- **Clean-up:** remove the package from the current folder;

b) get a specific package by its UUID:

- **Set-up:** have the `:uuid` of a package successfully created in the Catalogue;
- **Execution:** `curl localhost:5000/packages/:uuid`
- **Expected:** HTTP code 200 (OK) is returned, together with the package file;
- **Clean-up:** remove the package from the folder where ir has been dowloaded to;

c) get a specific package by its `:vendor`, `:name` and `:version`:

- **Set-up:** have the `:vendor`, `:name` and `:version` of a package successfully created in the Catalogue;
- **Execution:** `curl 'localhost:5000`

`/packages?vendor=:vendor&name=:name&version=:version'`

- **Expected:** HTTP code 200 (OK) is returned, together with the package file;
- **Clean-up:** remove the package from the folder where ir has been dowloaded to;

d) get a list of packages:

- **Set-up:** have at least two packages successfully created in the Catalogue;
- **Execution:** `curl localhost:5000/packages`
- **Expected:** HTTP code 200 (OK) is returned, together with the list of packages in the Catalogue;
- **Clean-up:** remove the packages from the Catalogue;

2. **Services:**

   a) get a list of services:

   - **Set-up:** have at least two services successfully created in the Catalogue;
   - **Execution:** `curl localhost:5000/services`
   - **Expected:** HTTP code 200 (OK) is returned, together with the list of services in the Catalogue;
   - **Clean-up:** remove the services from the Catalogue;

3. **Requests:**

   a) create an instantiation request:

   - **Set-up:** have a valid service uuid (`service_uuid`) from which an instance is going to be requested;
   - **Execution:** `curl -X POST --data "service_uuid=:service_uuid" \`

   `localhost:5000/requests`

   - **Expected:** HTTP code 201 (Created) is returned, together with the JSON representation of the created request;

   b) get a list of requests:

   - **Set-up:** have at least two requests successfully created;
   - **Execution:** `curl localhost:5000/requests`
   - **Expected:** HTTP code 200 (OK) is returned, together with the list of requests successfully created;
   - **Clean-up:** remove the requests;

   c) get the status of a specific request:

   - **Set-up:** have a valid request uuid (`:uuid`);
   - **Execution:** `curl localhost:5000/requests/:uuid`
   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the request;
   - **Clean-up:** remove the requests;

2.3 Gatekeeper's Package Management

In the SONATA Service Platform the Gatekeeper Package Management plays the role of wrapping and unwrapping data that is related to a package, according to a pre-defined format (see Appendix A).

2.3.1 Features

This sub-section describe the features expected for Gatekeeper Package Management.

Accepts new packages: **As a** Platform Owner, **I want to** be able to accept new packages, **so that** I can validate them and store their content in the Service Platform's Catalogue(s).

Validates a package: **As a** Platform Owner, **I want to** be able to check if the provided package is according to the definition, **so that** only valid packages get into the system. This user story uses the schema defined in Appendix A.

Builds a package file: **As a** Platform Owner, **I want to** be able to build a package file from the package, service and functions already registered in the catalogues, **so that** this package file can be sent to the API

Unbuilds a package file: **As a** Platform Owner, **I want to** be able to unbuild a package file and store the package, service and function data in the catalogues, **so that** the respective data can be saved in the Catalogues

Stores a package in the Catalogue: **As a** Platform Owner, **I want to** be able to save a package descriptor in the catalogues, **so that** the package can later be fetched

Loads a package from the Catalogue: **As a** Platform Owner, **I want to** be able to load a package descriptor from the catalogues, **so that** its descriptor becomes available

Deletes a package from the Catalogue: **As a** Platform Owner, **I want to** be able to delete a package descriptor from the catalogues, **so that** its descriptor is no more available

Stores a service in the Catalogue: **As a** Platform Owner, **I want to** be able to save a service descriptor in the catalogues, **so that** the service can later be fetched

Loads a service from the Catalogue: **I want to** be able to load a service descriptor from the catalogues, **so that** its descriptor becomes available

Deletes a service from the Catalogue: **As a** Platform Owner, **I want to** be able to delete a service descriptor from the catalogues, **so that** its descriptor is no more available

Stores a function in the Catalogue: **As a** Platform Owner, **I want to** be able to save a function descriptor in the catalogues, **so that** the function can later be fetched

Loads a function from the Catalogue: **As a** Platform Owner, **I want to** be able to load a function descriptor from the catalogues, **so that** its descriptor becomes available

Deletes a function from the Catalogue: **As a** Platform Owner, **I want to** be able to delete a function descriptor from the catalogues, **so that** its descriptor is no more available

Provides a list of existing packages: **As a** Platform Owner, **I want to** be able to provide a list of registered packages, **so that** the API can return it to the SDK

Provides a package: **As a** Platform Owner, **I want to** provide a package from the Service Platform's Catalogue, **so that** the API can return it

Provides a log file view: **As a** Platform Owner, **I want to** be able to provide a view to the log file, **so that** it can be used for managing/monitoring/debugging the micro-service

2.3.2 API

This sub-section describes the (external) API of Gatekeeper's Package Management (see Table Table 2.4).

Table 2.3: Package Management micro-service REST API

Endpoint	Method	Description	Returned code(s)
`/packages`	POST	Submits a new package to the Gatekeeper. The format of the file must be conferment with the `son-schema` defined. In this version, the package file is discarded, with its relevant data stored in the Catalogues. In later versions the file will be preserved.	Created (201)
	GET	Retrieve a list of package data currently registered in the system. If the result of the query in a single package, a package file is returned (like if it was requested as `GET /packages/:uuid`). It supports pagination with the `offset` (default value zero) and `limit` (default value ten) parameters.	Ok (200), Not found (404)
`/packages/:uuid`	GET	Retrieve a package `son-schema` formatted file, who's id is `uuid`. In this version the package is rebuilt with all the information existing in the Catalogues. In later versions the original file will be returned.	Ok (200), Not found (404)
`/packages/:uuid/package`	GET	Retrieve a package file, who's id is `uuid`. In this version the package is rebuilt with all the information existing in the Catalogues. In later versions the original file will be returned.	Ok (200), Not found (404)
`/admin/logs`	GET	Retrieve the currently available log file. The implementation will have to be improved to support bigger files (currently, log files are reset for every new pull request, so most of the times they'll have a manageable size.	Ok (200), Not found (404)

2.3.3 Internal Architecture

Figure 2.10 shows the high-level architecture of the Package Management module in the SONATA Gatekeeper.

This micro-service has only two 'routes' (please refer to Section 2.1.3.3), one focused on the access to the Packages and the other to the log file. Models are split by it's core function: loading and saving packages, services and functions (the later might later migrate to the Service Management micro-service (see Section 2.4) and to a possible Function Management micro-service.

2.3.4 Message Sequence Charts

This sub-section describes the MSCs where Gatekeeper's Package Management plays a role.

We have opted to show only the two most relevant MSCs, which cover the main features of this micro-service: to know how to store a package, its service and functions from a package file, and to rebuild a package file from the stored package, service and functions descriptor. All the components are used in only these two MSCs.

In order not to harm readability of the charts, not all error conditions are shown.

2.3.4.1 Accepts new packages

Accepting, validating and storing is one of the core features of the Package Management micro-service. The MSC shown in Figure 2.11 details this features.

This MSC has two details that worth mentioning:

1. a package may have a service, one or more functions or both a service and at least a function;

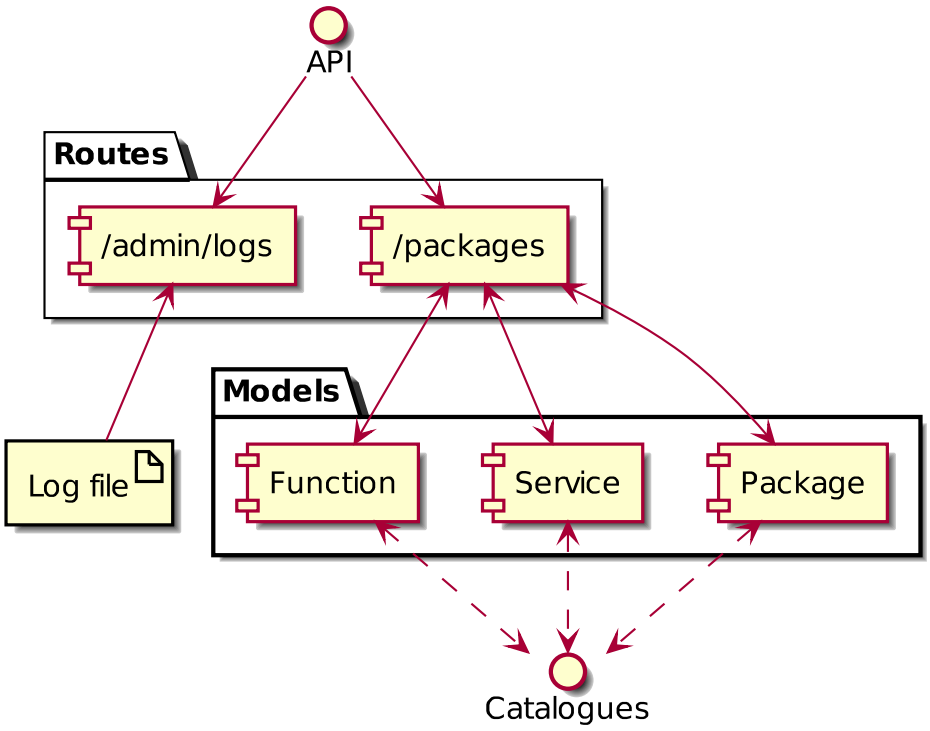

Figure 2.10: Gatekeeper's Package Management high level architecture

2. if for some reason adding one of these components fails, the components successfully added in this context must be deleted.

This last point still deserves further work, since both services and specially functions may already exist in the Catalogues.

2.3.4.2 Provides a package

Providing a package, in the current version, means building the package from scratch, from the data stored in the Packages, Services and Functions Catalogues. The MSC shown in Figure 2.12 details this features.

2.3.5 Technologies used

The technologies used in the implementation of this micro-service are similar to those used for the Gatekeeper's API (please refer to Section 2.2, Table 2.2).

2.3.6 Tests

This sub-section describes tests designed and implemented for the Package Management micro-service.

2.3.6.1 Unit tests

We are restricting unit tests of the Package Management to the access of the log file path (`/admin/logs`, i.e., those not needing any model). The remaining unit tests became very basic, since we are mocking the Catalogues micro-service. We execute these accessing the internal URL (`http://localhost:5100`).

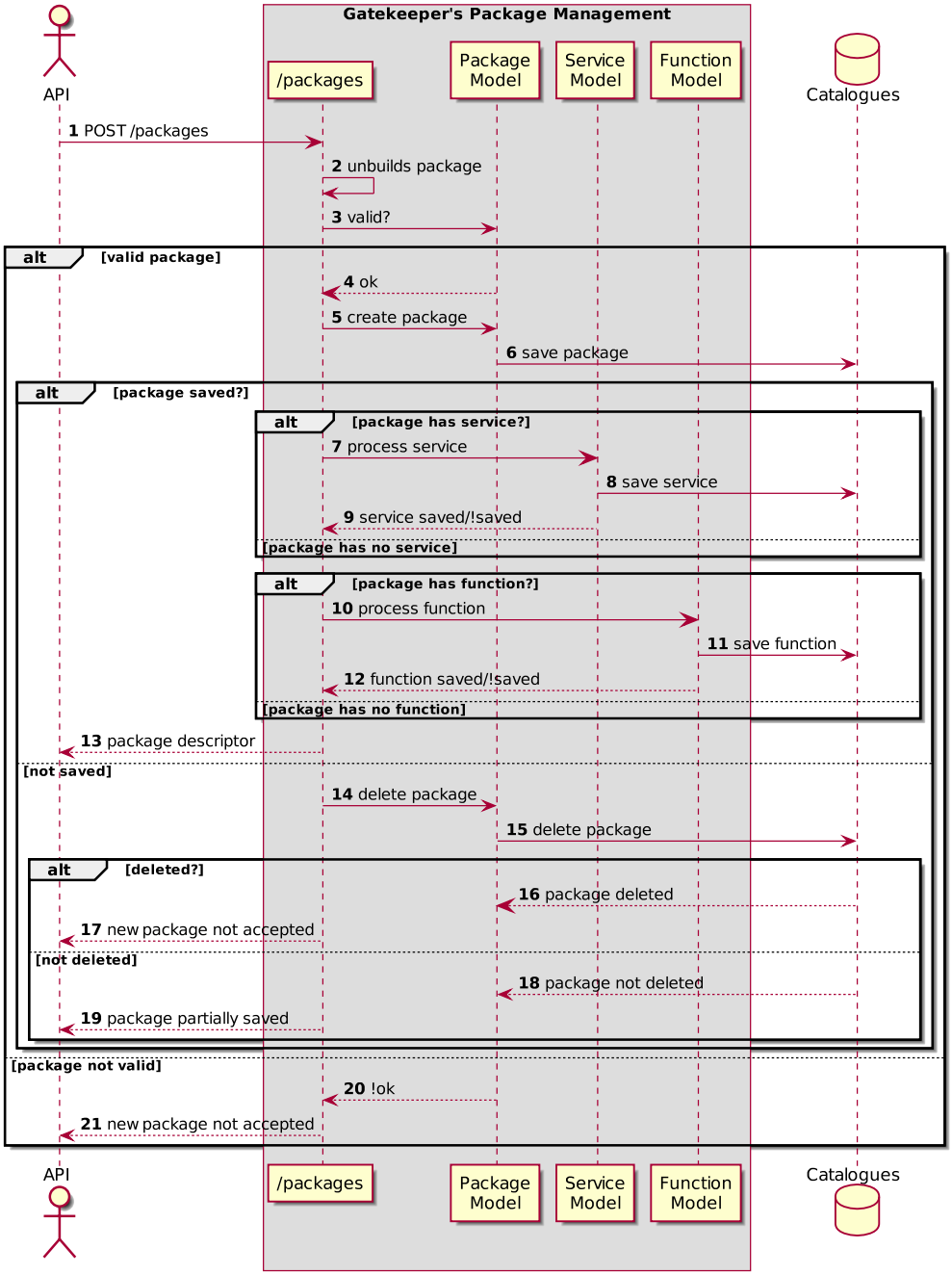

Figure 2.11: The Gatekeeper's Package Management accepts new package from the API

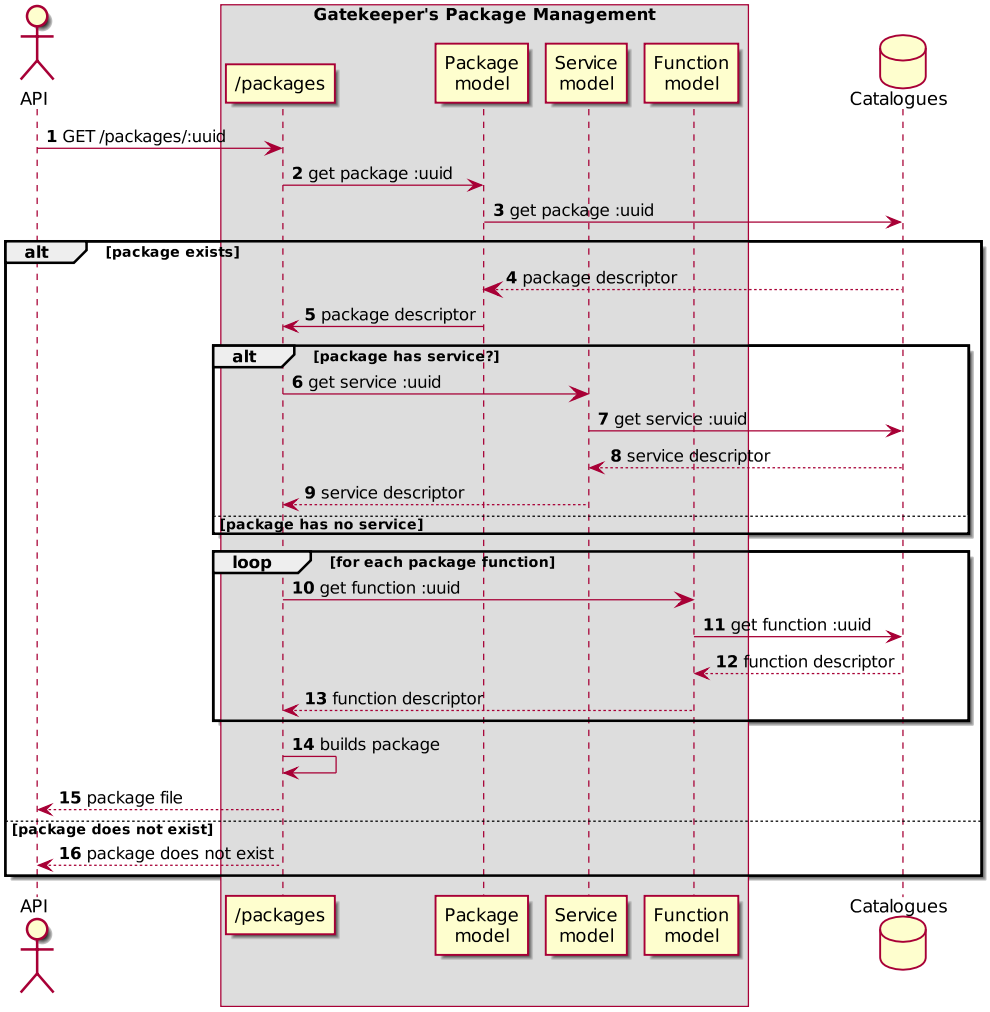

Figure 2.12: The Gatekeeper's Package Management accepts new package from the API

The unit tests of the Package router are the following:

1. Accept a package:
   - **Set-up:** have a valid package (`sonata-demo.son`), in the current folder
   - **Execution:** `curl -X POST -F "package=@sonata-demo.son" localhost:5100/packages`
   - **Expected:** HTTP code 201 (Created) is returned, together with the JSON representation of the created package;
   - **Clean-up:** remove the package from the current folder;
2. List existing packages:
   - **Execution:** `curl localhost:5100/packages`
   - **Expected:** HTTP code 200 (OK), with the list of packages available
3. Get a specific package:
   - **Set-up:** have a valid package `uuid`
   - **Execution:** `curl localhost:5100/packages/:uuid`
   - **Expected:** HTTP code 200 (OK), with the package descriptor
4. Get package file:
   - **Set-up:** have a valid package `uuid` for which a package file has already been generated
   - **Execution:** `curl localhost:5100/packages/:uuid/package`
   - **Expected:** HTTP code 200 (OK), with the package file
   - **Clean-up:** remove the obtained file
5. List logs:
   - **Execution:** `curl localhost:5100/admin/logs`
   - **Expected:** HTTP code 200 (OK), the log content shown (it has to include this same call as its last line)

Further unit tests will be written namely to test the package validation.

2.3.6.2 Module Tests

For the module tests we have considered models of the 'Package Management Service' (see Section 2.3) and the 'Service Management Service' (see Section 2.4), with 'mocked' Catalogues.

1. submit a valid package:
   - **Set-up:** have a valid package (`sonata-demo.son`, in the current folder)
   - **Execution:** `curl -X POST -F "package=@sonata-demo.son" localhost:5100/packages`
   - **Expected:** HTTP code 201 (Created) is returned, together with the JSON representation of the created package;
   - **Clean-up:** remove the package from the current folder;
2. get a specific package by its UUID:

- **Set-up:** have the `:uuid` of a package successfully created in the Catalogue;
- **Execution:** `curl localhost:5100/packages/:uuid`
- **Expected:** HTTP code 200 (OK) is returned, together with the package file;
- **Clean-up:** remove the package from the folder where ir has been dowloaded to;

3. get a specific package by its `:vendor`, `:name` and `:version`:
   - **Set-up:** have the `:vendor`, `:name` and `:version` of a package successfully created in the Catalogue;
   - **Execution:** `curl 'localhost:5100/packages?vendor=:vendor&name=:name&version=:versi`
   - **Expected:** HTTP code 200 (OK) is returned, together with the package file;
   - **Clean-up:** remove the package from the folder where ir has been dowloaded to;
4. get a list of packages:
   - **Set-up:** have at least two packages successfully created in the Catalogue;
   - **Execution:** `curl localhost:5100/packages`
   - **Expected:** HTTP code 200 (OK) is returned, together with the list of packages in the Catalogue;
   - **Clean-up:** remove the packages from the Catalogue;

2.4 Gatekeeper's Service Management

In the SONATA Service Platform, the Gatekeeper Service Management plays the role of dealing with all the aspects related to (network) services.

2.4.1 Features

This sub-section describes the features expected for Gatekeeper Service Management.

Validation of who can play a role in each story will not be implemented in this version. When it is, it will be left to the API level (see Section 2.2).

Provides a list of available services: **As a** Platform Owner, **I want to** be able to provide a list of available services, **so that** they can later be instantiated.

Accepts service instantiation requests: **As a** Platform Owner, **I want to** be able to accept an instantiation request, **so that** I can pass the request to the MANO Framework. Please note that the operation of allocating resources on the VIM is usually time-consuming, and therefore made asynchronously.

Retrieve all the service/functions data: **As a** Platform Owner, **I want to** be able to retrieve from the Catalogues all the needed service and functions descriptors, **so that** they can be passed to the MANO Framework.

Requests an instantiation of a service to the MANO Framework: **As a** Platform Owner, **I want to** be able to request the MANO Framework for a services instantiation, **so that** the MANO Framework can ask the Infrastructure Adapter for the needed resources.

Accepts MANO Framework responses: **As a** Platform Owner, **I want to** be able to accept the answers to the requests for service instantiation, **so that** they can be communicated to the BSS.

Provides the status of the registered requests: **As a** Platform Owner, **I want to** be able to provide the status of the registered requests, **so that** so that the BSS can notify the End User about that status.

Provides a log file view: **As a** Platform Owner, **I want to** be able to provide a view to the log file, **so that** it can be used for managing/monitoring/debugging the micro-service.

2.4.2 API

This sub-section describes the (external) API of Gatekeeper's Package Management (see Table Table 2.4).

Table 2.4: Package Management micro-service REST API

Endpoint	Method	Description	Returned code(s)
`/services`	GET	Provides a list of services currently registered in the Catalogues. Query parameters like `/services?status=active` can be used to restrict services returned. Specific fields can be selected by using the special query parameter `fields`, like in `/services?fields=uuid,name`. Returned results are paginated, by using special query fields like `offset` (default o `0`) and `limit` (default o `10`).	Ok (200)
`/requests`	POST	Submits a new service instantiation request to the Gatekeeper, providing the `service_uuid`	Created (201)
	GET	Retrieve a list of requests currently registered in the system. Query parameters like `/requests?status=ready` can be used to restrict requests returned. Specific fields can be selected by using the special query parameter `fields`, like in `/services?fields=status,updated_at`. Returned results are paginated, by using special query fields like `offset` (default o `0`) and `limit` (default o `10`).	Ok (200), Not found (404)
`/admin/logs`	GET	Retrieve the currently available log file. The implementation will have to be improved to support bigger files (currently, log files are reset for every new pull request, so most of the times they'll have a manageable size).	Ok (200), Not found (404)

2.4.3 Internal Architecture

Figure 2.13 shows the high-level architecture of SONATA's Gatekeeper Service Management.

There are three possible routes, one that provides the list of available services, one for the instantiation and one to access the log file. Models are split by the concept they encapsulate: Services (load from the Catalogues), Requests (stored in and fetched from the database) and MQServer (connect to the RabbitMQ).

2.4.4 Message Sequence Charts

This sub-section describes the MSCs where Gatekeeper Service Management micro-service plays a role.

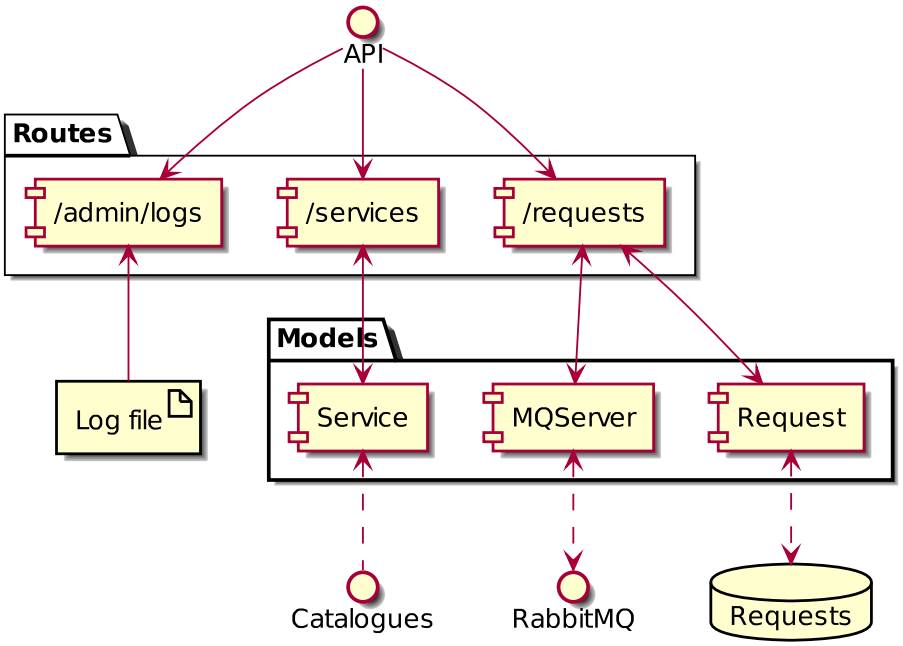

Figure 2.13: Gatekeeper's Service Management high level architecture

2.4.4.1 List of services that can be instantiated

Figure 2.14 shows the MSC of the request of active services (those that can be instantiated by the MANO Framework).

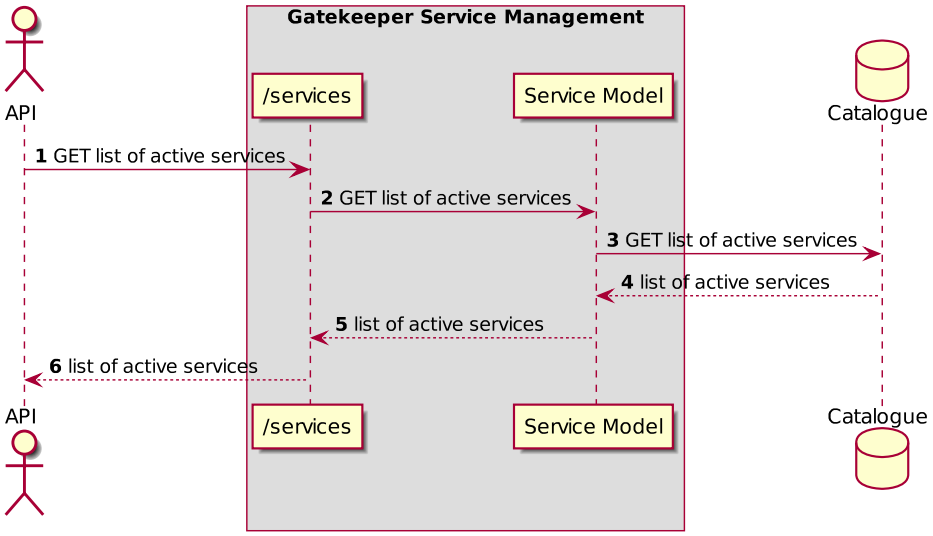

Figure 2.14: Services available for instantiation

The section of only the active ones can be done by using query parameters in the request, such as in `/services?status=active`.

2.4.4.2 Request for the instantiation of a service

The instantiation of a service is a complex operation in the Service Platform. For the Gatekeeper, it involves gathering all data related to the service and submitting them to the MANO Framework. Figure 2.15 shows the MSC of this operation.

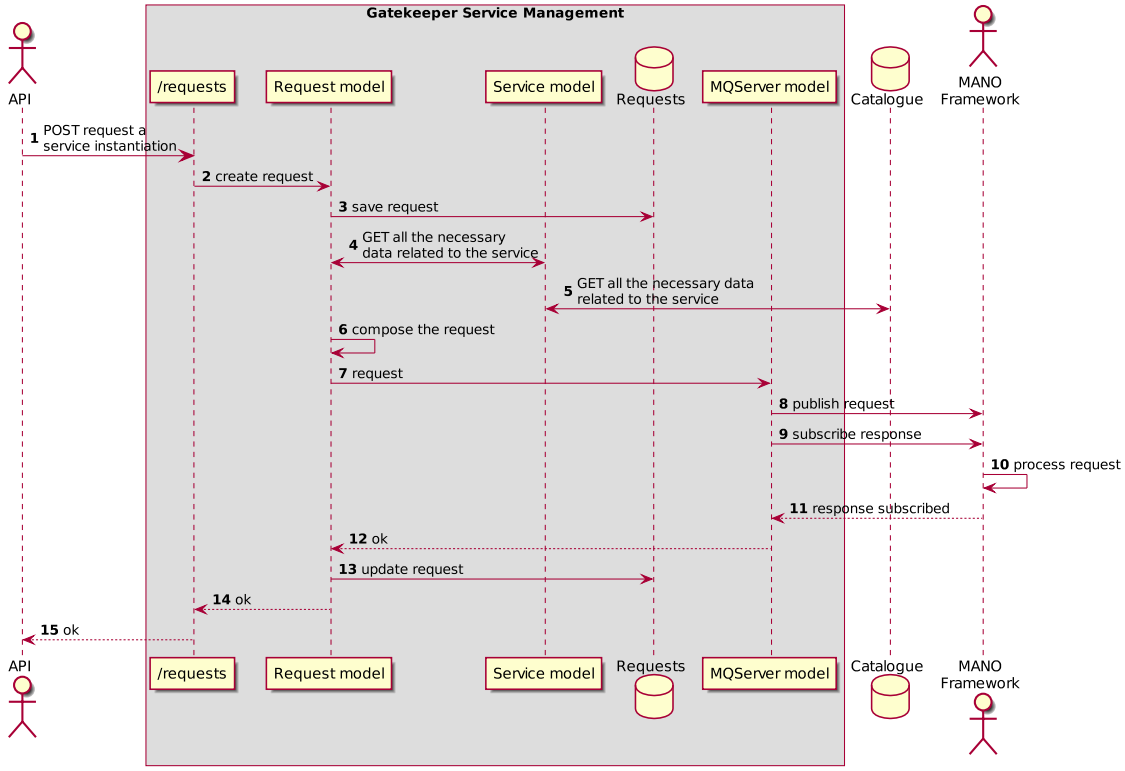

Figure 2.15: Request the instantiation of a service

2.4.4.3 The MANO Framework notifies the Gatekeeper about a service readiness

Instantiating a service in a VIM may take a relatively long time, when compared to a common web interaction. If the platform used a synchronous call, it would probably time-out most of the times. So, the Gatekeeper's request is registered on the MANO Framework side, and then executed and when it finishes, a call is made to the Gatekeeper, notifying the service is ready to be used (or the request resulted in an error). When this notification occurs, the Gatekeeper updates the `status` field in the request. Figure 2.16 shows the MSC of this operation.

2.4.4.4 The BSS enquires the Gatekeeper about a service readiness

Given the asynchronous nature of the instantiation request (see above), the BSS (the 'actor' that asks for the instantiation of a given service) needs a way to check if the request has been fulfilled. An alternative design to this 'pull'-model would be the request to include a 'call-back' (a 'push'-model) the Gatekeeper would use upon an answer from the MANO Framework. Figure 2.17 shows the MSC of this operation.

In order to increase readability of the charts, not all error conditions are shown.

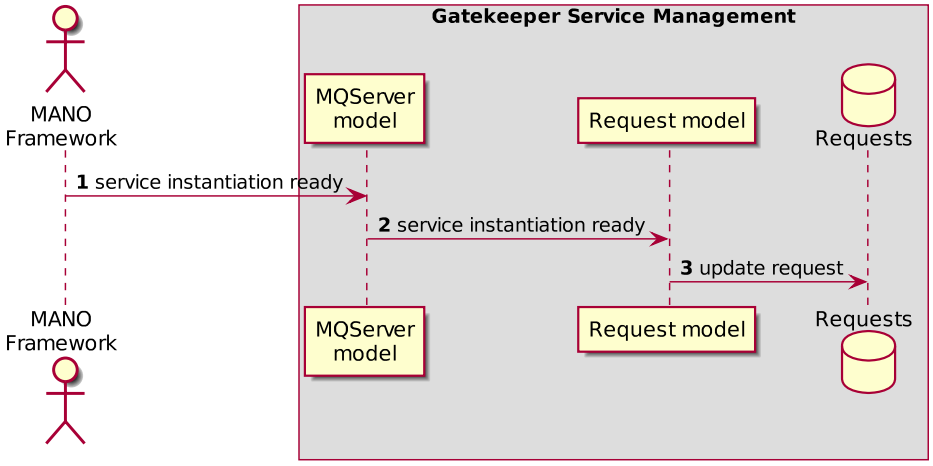

Figure 2.16: The Gatekeeper is notified about a service instantiation

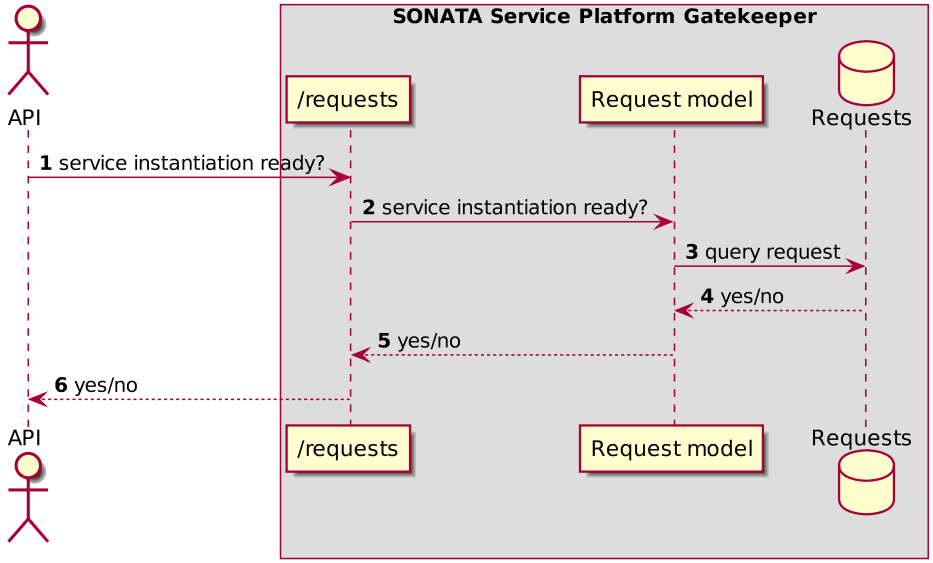

Figure 2.17: The BSS asks the Gatekeeper about a service instantiation

2.4.5 Technologies used

The technologies used in the implementation of this micro-service are similar to those used for the **Gatekeeper API** (please refer to Section 2.2, Table 2.2), plus those described in Table 2.5.

Table 2.5: Technologies used in the Gatekeeper's API component

Name	Type	Purpose
activerecord	library	to map between Object and Relational
bunny	library	to connect to the RabbitMQ Message Queue Server
pg	driver	to connect to the PostgreSQL Relational Database server
sinatra-activerecord	library	complements **sinatra** in using **activerecord**
sinatra-active-model-serializers	library	complements **sinatra** in serialising records into/from JSON

2.4.6 Tests

This sub-section describes tests that have been designed and implemented to test the Gatekeeper's Package Management.

2.4.6.1 Unit tests

The unit tests shown are only for the routes, since unit tests for the models are currently very basic[3].
We execute these tests by mocking the models and accessing the internal URL (`http://localhost:5300`).

Unit tests done to the Service Management routers are the following:

1. Available services:
   - **Execution:** `curl localhost:5300/services`
   - **Expected:** the Service model must be called once
2. Register an instantiation request:
   - **Execution:** `curl -X POST -d "service_uuid=:service_uuid" localhost:5300/requests`
   - **Expected:** the Service model must have been accessed (validate `service_uuid`), the Request model must have been accessed once (to save the request), the MQServer model must have been accessed twice (to publish the request and to subscribe the response)
3. Provide data about a specific request:
   - **Execution:** `curl localhost:5300/requests/:uuid`
   - **Expected:** The Request model must have been accessed once
4. List logs:
   - **Execution:** `curl localhost:5300/admin/logs`
   - **Expected:** the log file must have been accessed

Further tests will be specified and executed, specially taking into consideration all the possible combinations of query parameters (`status=...`, `fields=...` and `offset/limit`).

[3]We are mocking the Catalogues micro-service and the RabbitMQ.

2.4.6.2 Module Tests

Module tests done to the Service Management micro-service were the following:

1. Available services:
   - **Set-up:** have at least two services stored in the (fake) Catalogue
   - **Execution:** `curl localhost:5300/services`
   - **Expected:** 200 (Ok), with the list of available service descriptors
2. Register an instantiation request:
   - **Execution:** `curl -X POST -d "service_uuid=:service_uuid" \`

   `localhost:5300/requests`
   - **Expected:** 201 (Created), with the request descriptor. The Request database must have been accessed once, the Catalogue must have been accessed once for the service descriptor and once for every function descriptor the service depends on, and the MQServer must have been accessed twice, once for the subscription and another for the publishing
3. Provide data about a specific request:
   - **Set-up:** have at least one request stored in the database
   - **Execution:** `curl localhost:5300/requests/:uuid`
   - **Expected:** 200 (OK), with the request descriptor, 404 (Not Found). The Request database must have been accessed once

Further tests will be specified and executed, specially taking into consideration all the possible combinations of query parameters (`status=...`, `fields=...` and `offset/limit`).

2.5 Gatekeeper's GUI

Gatekeeper GUI is an API management and visualization tool that enables SONATA developers to manage their services throughout their whole lifecycle and enables the Service Platform administrator to provision and monitor platform resources easily and securely. In this perspective, the Gatekeeper GUI has been designed, developed and implemented to cover the needs of the two aforementioned groups, namely service developers and platform administrators, in supporting the process of DevOps in SONATA.

2.5.1 Features

This sub-section describes the features implemented in this release of Gatekeeper GUI.

2.5.1.1 Sign in

In order to start using the GUI services offered by the SONATA Service Platform, the registered developer or Service Platform administrator must sign in to the Gatekeeper portal using personal credentials (username and password), as depicted in (Figure 2.18).

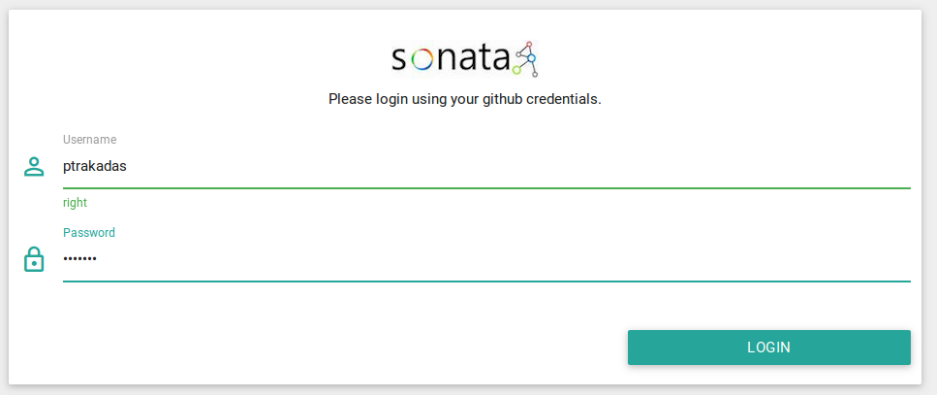

Figure 2.18: Sign In

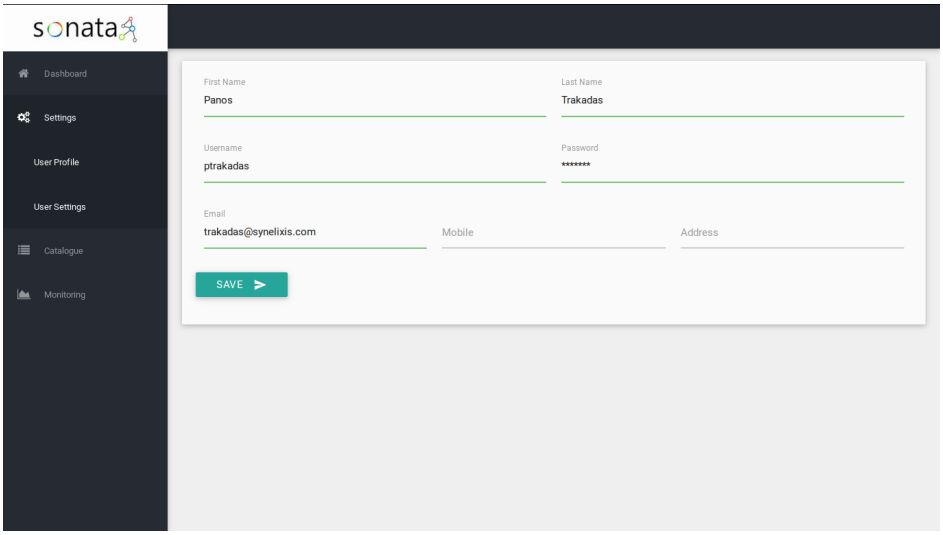

Figure 2.19: User Profile

2.5.1.2 Settings/User Profile

User profile, as depicted in Figure 2.19, allows a registered user to update his personal information, such as email, contact details, etc.

2.5.1.3 Settings/User settings

The developer/administrator is able to modify settings related to his account, e.g. language, time zone, etc. (Figure 2.20)

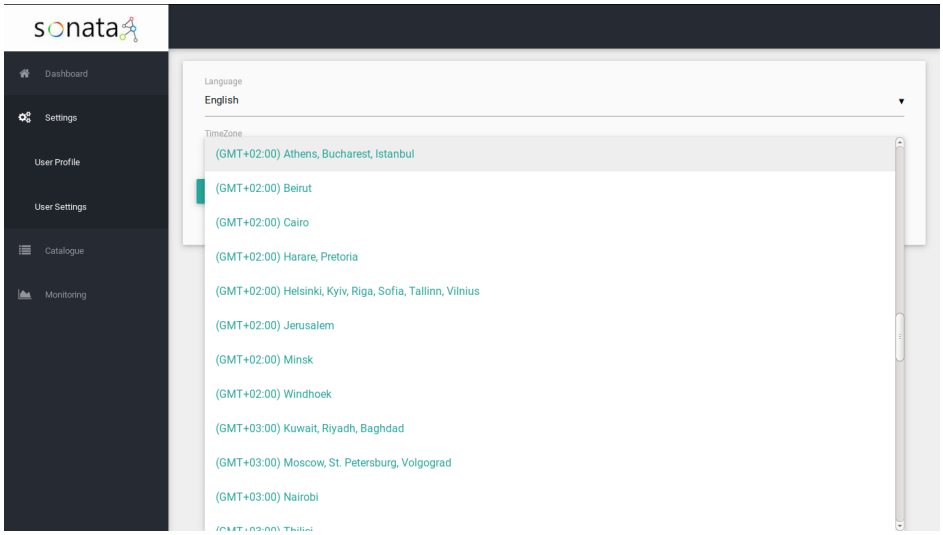

Figure 2.20: User Settings

2.5.1.4 Catalogue/Services

As shown in Figure 2.21, the developer/administrator is able to manage services. In particular, he can check the status of the services, upload a new service description, delete one of the existing ones and get notified for pending tasks. Additionally, details of a particular service can be displayed, as shown in Figure 2.22.

2.5.1.5 Monitoring Services and Functions

As shown in Figure 2.23, the developer is able to check the performance of the functions and/or services instantiated in several nodes, either in the form of Virtual Machines (Figure 2.24) or containers (Figure 2.25).

Figure 2.26 shows how specific monitoring data is displayed for VNFs.

2.5.2 Internal Architecture

This sub-section describes the Internal Architecture of GUI component and its interconnection with other SONATA components (Figure 2.27).

2.5.3 Message Sequence Charts

This sub-section describes the MSCs where the **GUI** plays a role.

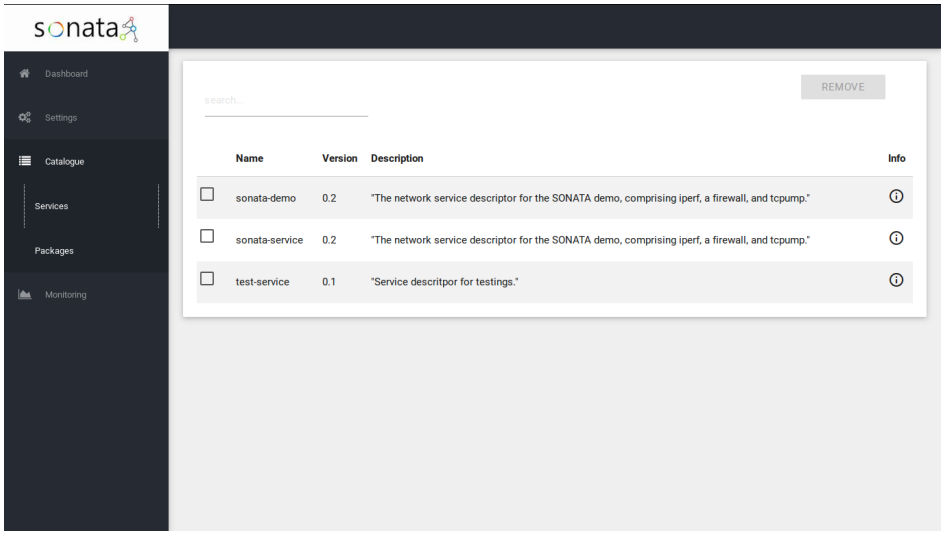

Figure 2.21: List of Services

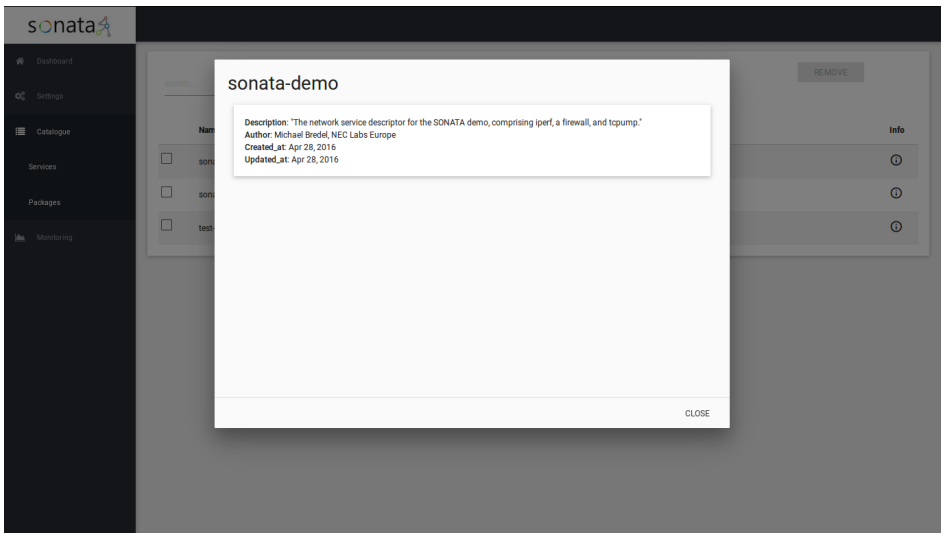

Figure 2.22: Service Details

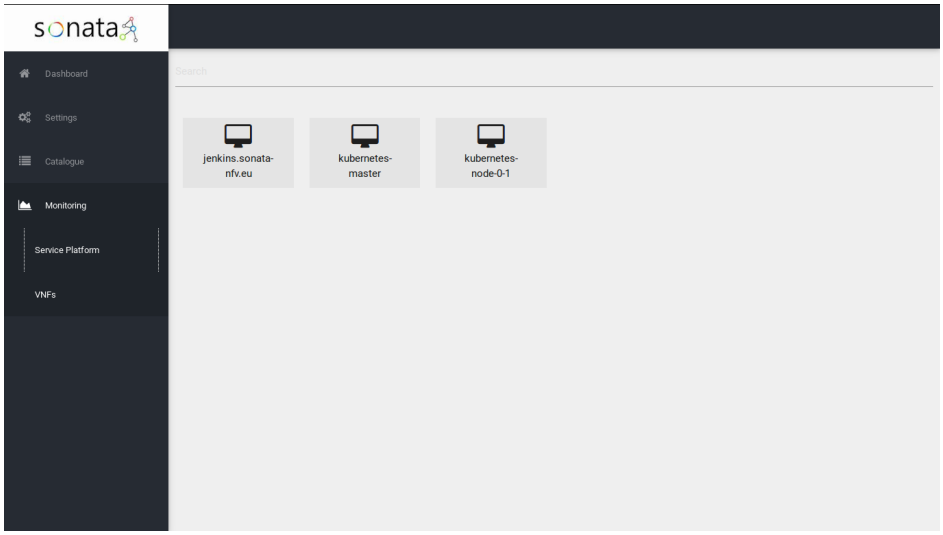

Figure 2.23: Monitoring Nodes

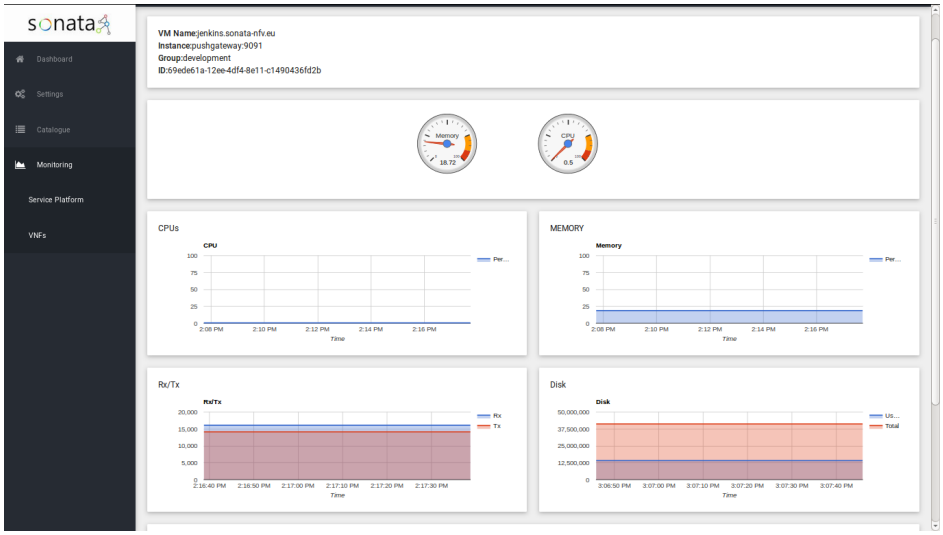

Figure 2.24: Monitoring Virtual Machines (in this case, Memory, CPU, Storage and Network).

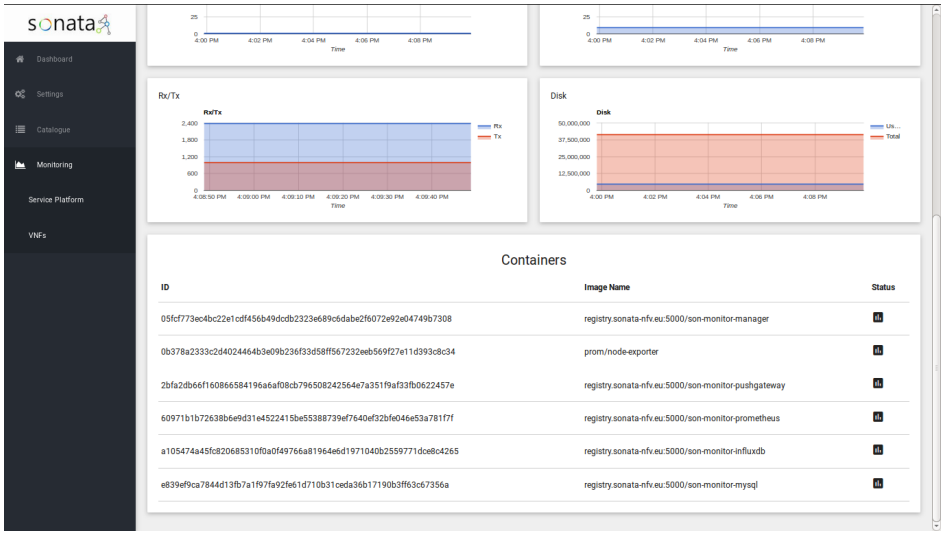

Figure 2.25: Monitoring Containers of a given VM.

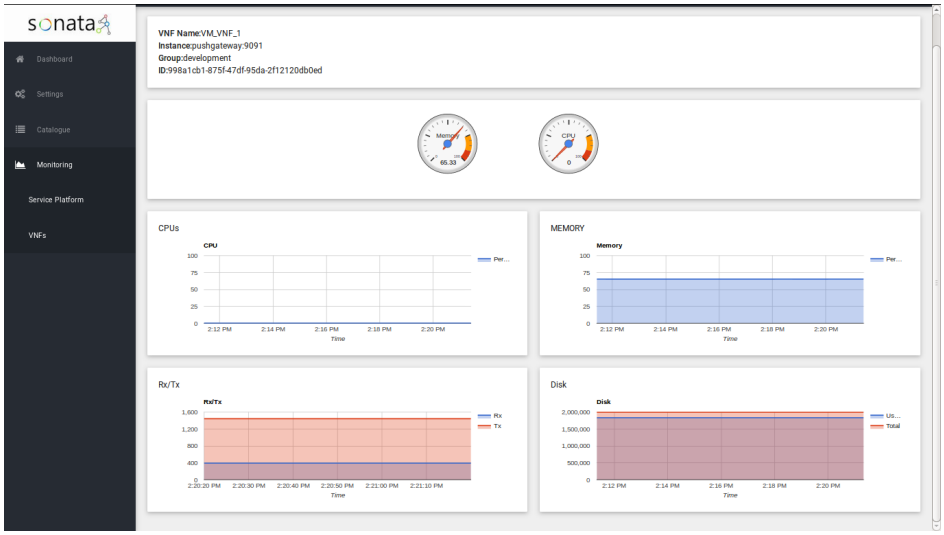

Figure 2.26: VNF Monitoring (in this case, Memory, CPU, Storage and Network).

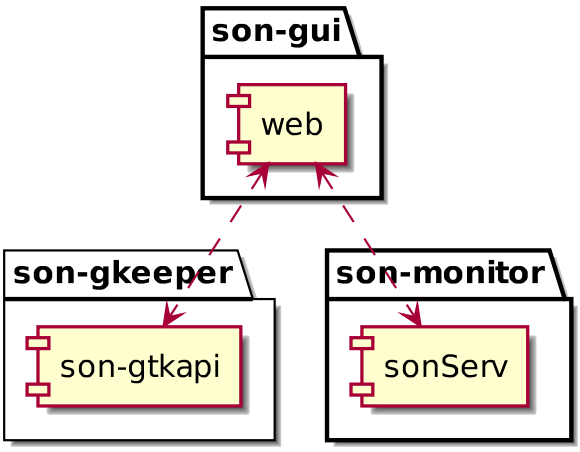

Figure 2.27: Gatekeeper GUI high level diagram

2.5.3.1 Retrieve the list of network services

This is one of the most important views in the GUI, to be used by both developers and Service Platform administrator, depicting the list of network services that can be instantiated. There is also a details view, in case the Developer needs to check details for a particular network service.

2.5.3.2 Retrieve the list of packages

Furthermore, the GUI can display the list of packages available in the Service Platform.

2.5.3.3 Retrieve monitoring data for functions and services

The GUI takes advantage of the API methods provided by the Monitoring Server to visualize monitoring data for instantiated network services and functions that consist of.

2.5.4 Technologies used

The technologies used in the implementation of the Gatekeeper's GUI are listed in Table 2.6.

Table 2.6: Technologies used in the implementation of the Gatekeeper's GUI

Name	Type	Purpose
angular	framework	JavaScript Framework for Web apps
bower	tool	Package manager for the web
grunt	tool	JavaScript Task Runner
node.js	tool	JavaScript Runtime
npm	tool	Package manager for JavaScript
yeoman	tool	Web Scaffolding Tool

2.5.5 Unit Tests

The following unit tests are based on Grunt javascript task runner which is already installed in son-gui container.

1. Check GUI in several Web Browsers:

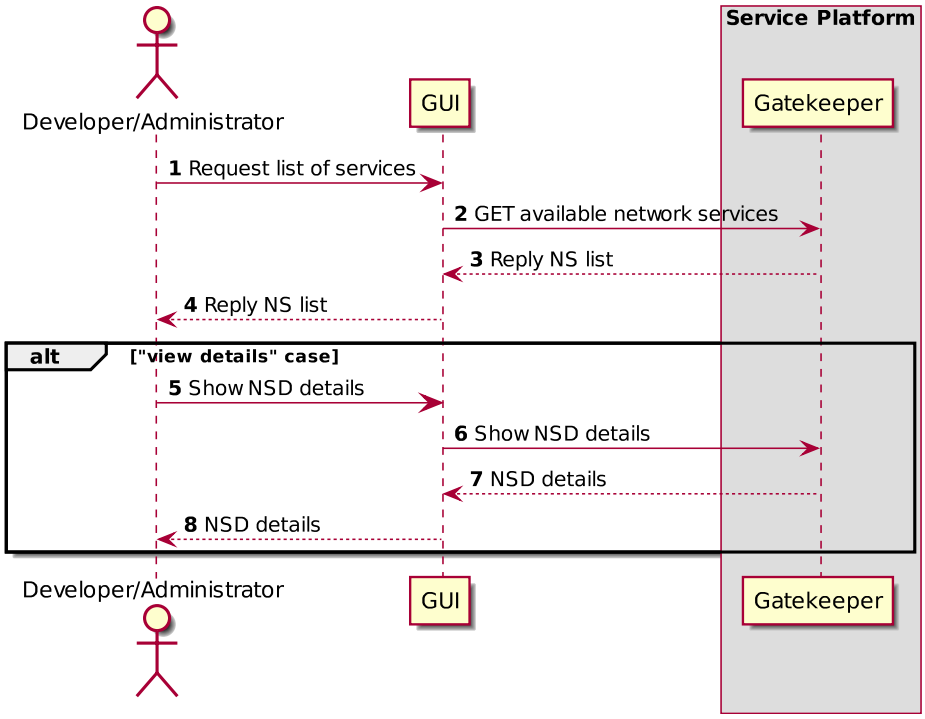

Figure 2.28: Retrieve the list of network services

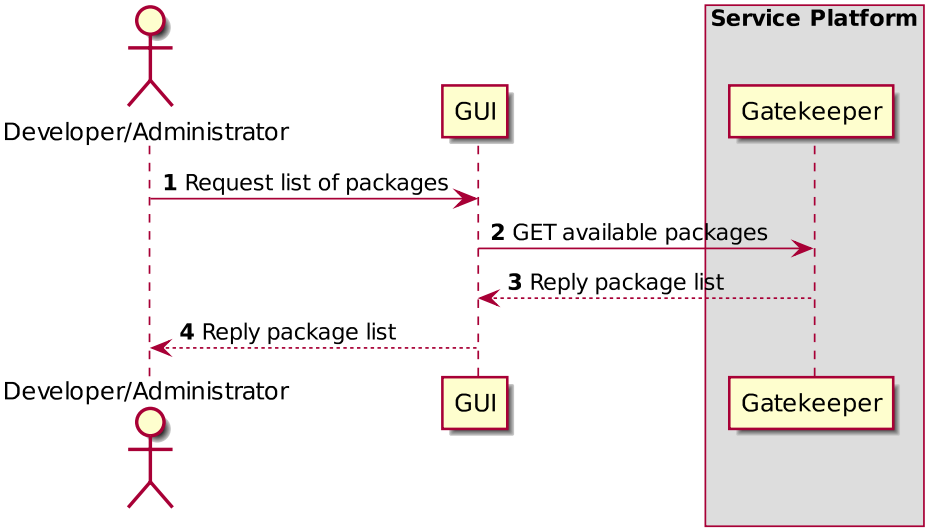

Figure 2.29: Retrieve the list of packages

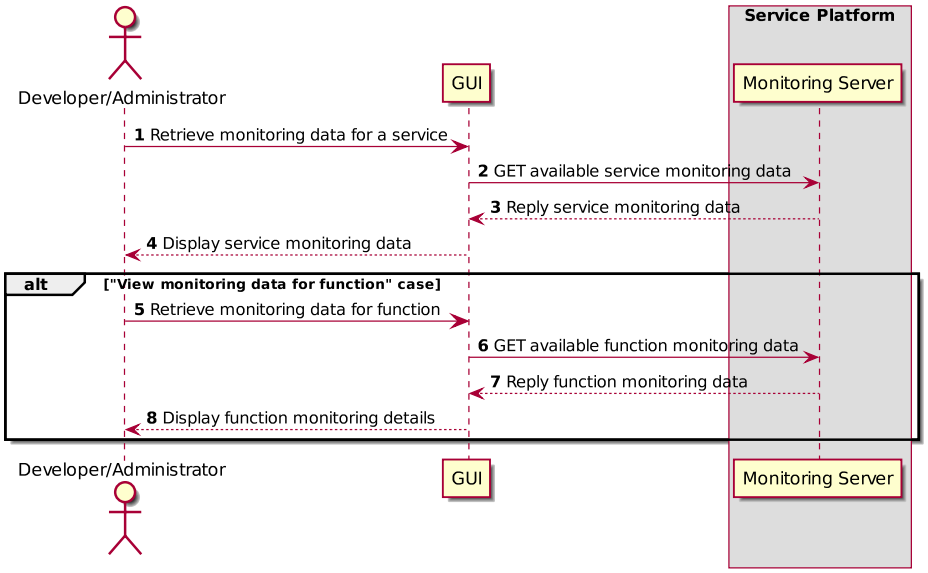

Figure 2.30: Retrieve monitoring data

- **Execution:** `grunt check_browsers`
- **Expected:** Done, without errors

2. Check that all necessary libraries are installed:
   - **Execution:** `grunt check_libs`
   - **Expected:** Done, without errors
3. Check Apache web server is running:
   - **Execution:** `grunt check_server`
   - **Expected:** Done, without errors

2.6 Gatekeeper's Business Support System

In the SONATA Service Platform the BSS is a simple module accessed by End User through a GUI with the following functionalities (further detailed below):

- Retrieve the list of services that can be instantiated;
- Allow the End User to create instances of those services;
- Retrieve the data about those instantiations requests (namely, it's status).

The BSS will access the Service Platform through the gatekeeper using the API specified in Section 2.2.

2.6.1 Features

This sub-section describes the features implemented in this release of Gatekeeper BSS

2.6.1.1 Retrieve the catalogue and create instantiations

The BSS web contains a "Network Services" menu that will show the list of Network Services that can be instantiated (Figure 2.31). From this menu it is possible to:

- show details about a NSD (Figure 2.32)
- instantiate a specific NSD (Figure 2.33)

Figure 2.31: NSD List

2.6.1.2 Retrieve the instantiations requests

After the instantiation is requested it will be possible to check the request status in the "Instantiations Requests" section, which will show:

- The list of Instantiation Orders
- A box to filter by Instantiation ID

2.6.2 Internal Architecture

Figure 2.34 shows the Internal Architecture of BSS component.

2.6.3 Message Sequence Charts

This sub-section describes the MSCs where **BSS** plays a role.

2.6.3.1 BSS Retrieve the catalogue and create instantiations

Figure 2.35 shows how the BSS retrieves the catalogue of valid services that can be instantiated, and how it creates a new service request.

2.6.3.2 BSS Retrieve the instantiations requests

Figure 2.36 shows how the BSS retrieve information about the instantiations requests created.

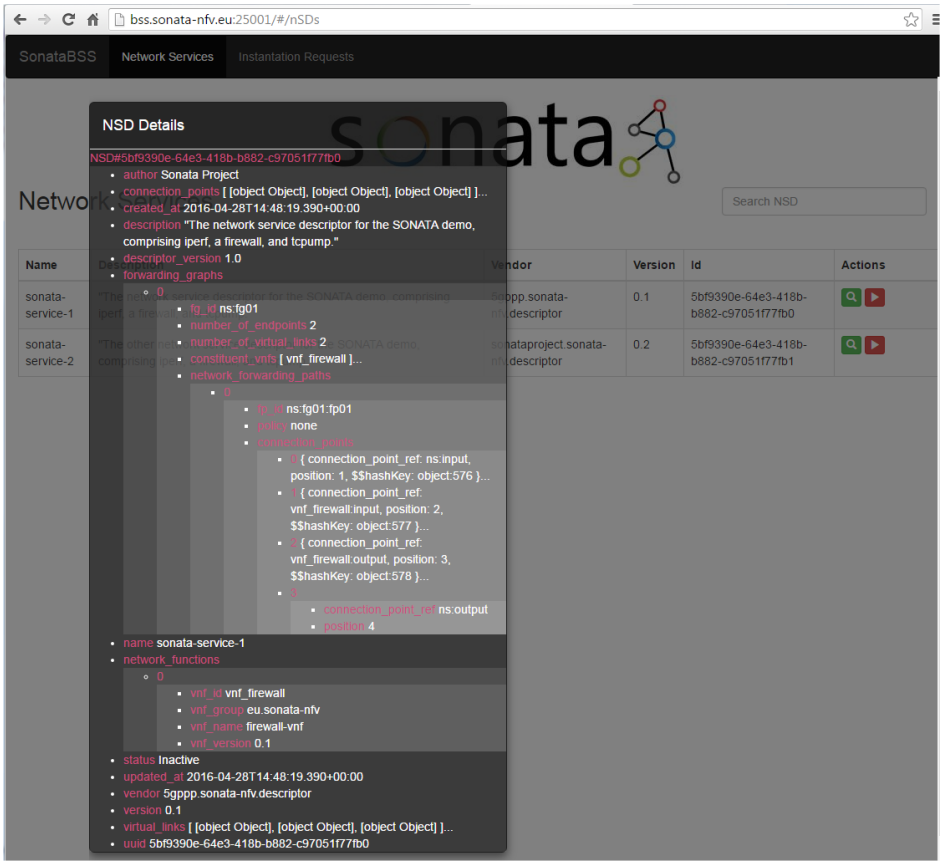

Figure 2.32: View NSD details

Figure 2.33: NSD instantiation

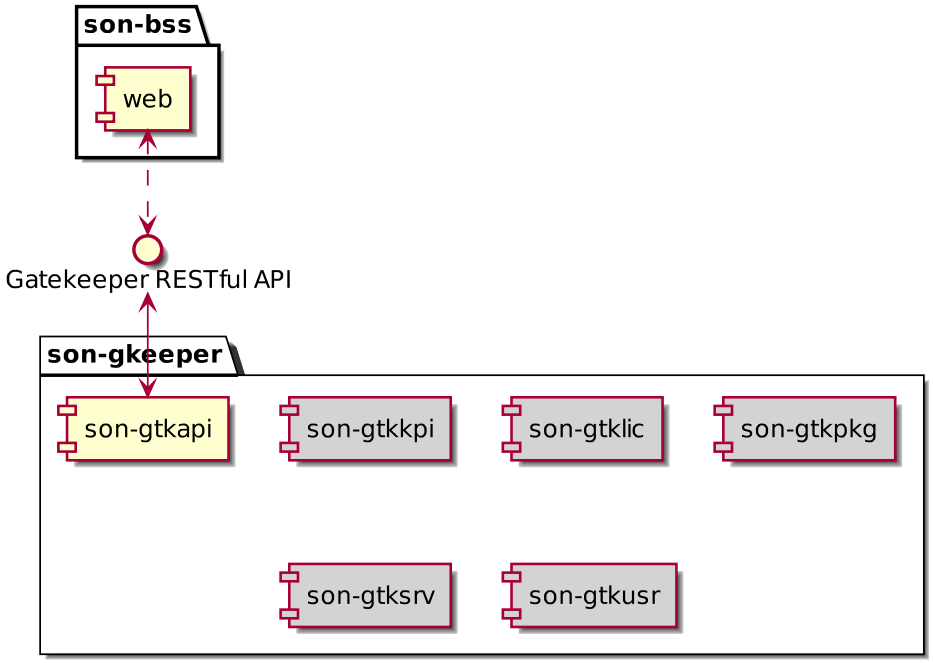

Figure 2.34: BSS Internal Architecture

2.6.4 Technologies used

Technologies used in the implementation of the BSS were the same as the ones used for the Gatekeeper's GUI, outlined in Section 5.3.2.5, to which **protractor**, a test framework for AngularJS applications, has beeb added.

2.6.5 Tests

This sub-section describes tests that have been designed and implemented to test the BSS. The tests uses `grunt`, a javascript task runner which is already installed in son-bss container. These tests are executed with the following command:

```
grunt serve:unit_tests --suite=unit
```

2.6.5.1 Network Services

Get Available Services list Requests a list of available services from the SP using the RESTful gatekeeper interface. The test performs an HTTP GET request.

Instantiate New Service Requests the instantiation of a new service using the RESTful gatekeeper interface. The test performs an HTTP POST request obtaining the request id.

2.6.5.2 Instantiation Requests

Get Instantiation Requests list Retrieves a list of requested instantiation orders which may be filtered by request_id, using the RESTful gatekeeper interface. The test performs an HTTP GET request.

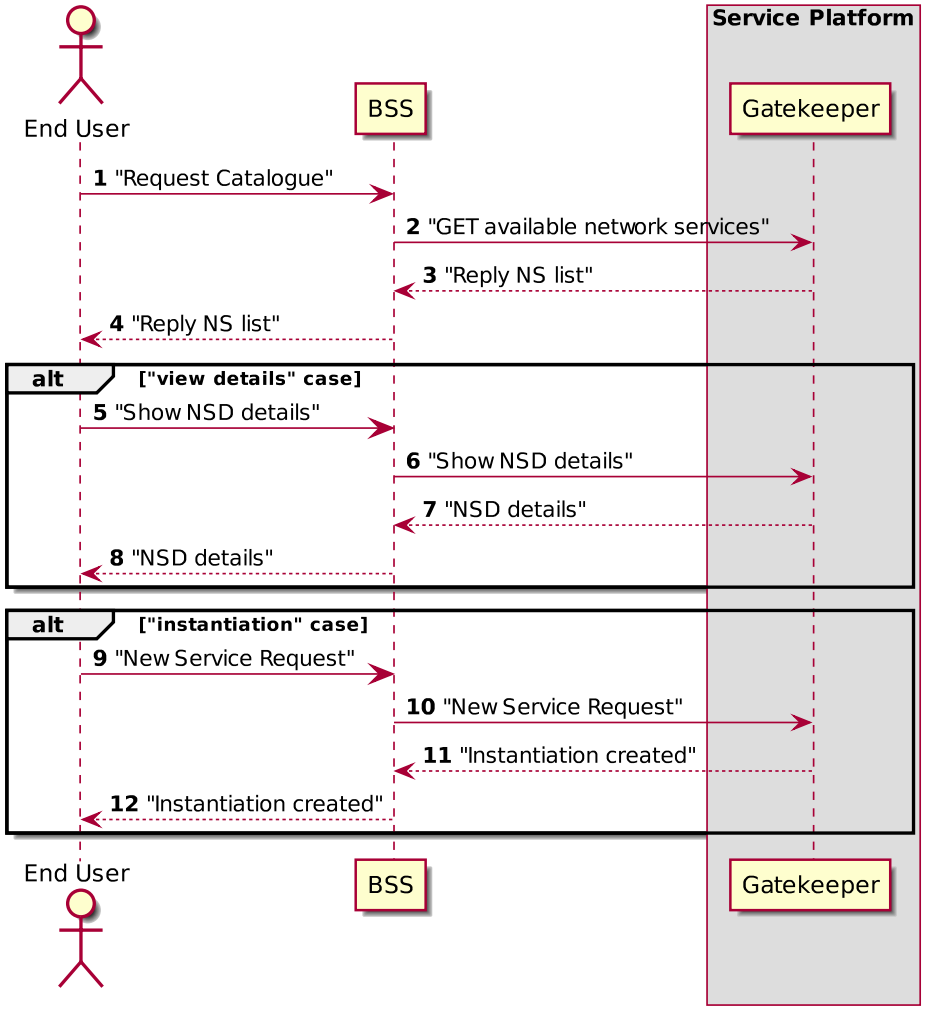

Figure 2.35: BSS retrieves the Catalogue and creates instantiations

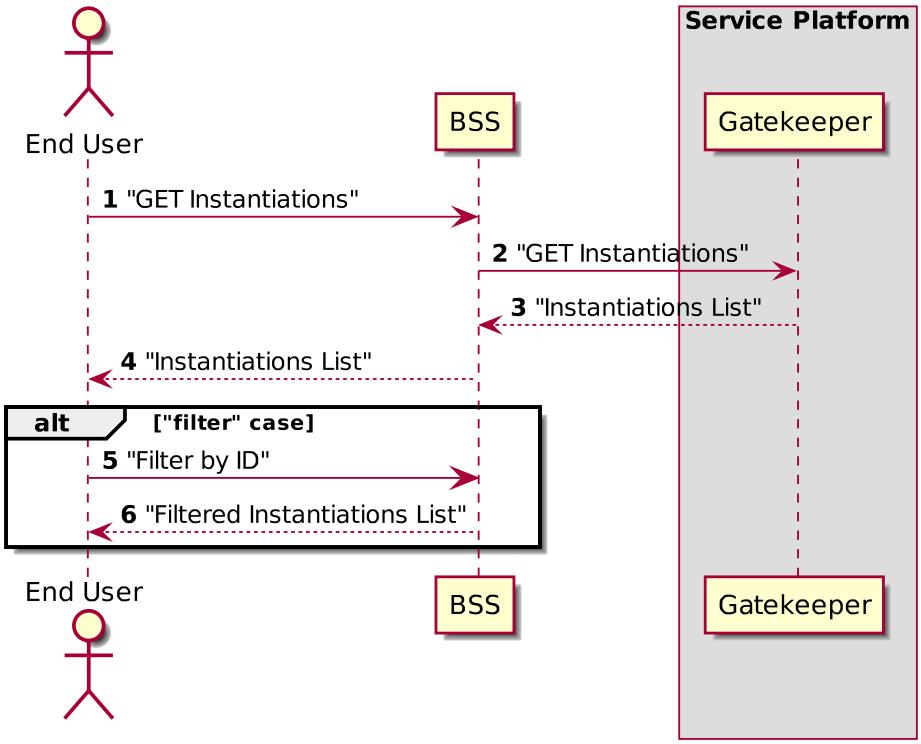

Figure 2.36: BSS retrieves the instantiations requests

2.7 Other Gatekeeper components

Iin the next releases the SONATA Gatekeeper will have additional components. This section presents our current line of thought.

- **User management:** although not one of the core features of the Gatekeeper, will be needed to validate which developers may access the Service Platform, which services and functions will each one of them have access to, etc;
- **License management:** where the different kinds of licenses (the provided service, the needed services and functions, etc.) will be managed;
- **Key Performance Indicators:** where KPIs about the platform will be stored, to be shown in the Gatekeeper's GUI (see Section 2.5);
- **Function Management:** the equivalent to the **Service Management** micro-service (see Section 2.4), this micro-service might have to be implemented to release the former on of some of its current responsibilities.

3 Orchestration Framework

As defined in the Description of Work (DoW) [7], task T4.2 gathers the work related to establishing a framework that contains the core functions to which plug-ins can be added to implement the set of features expected from a Network Service Orchestrator (NSO). As defined in Deliverable 2.2 [6] and presented in Figure 3.3, the heart of the framework is an asynchronous **Message Broker** to which plugins connect. The Basic functionalities of the plugins, related to the communication with the message broker, are encapsulated in the **Base Plugin**. The management of the plugins is performed by the **Plugin Manager**. The following sections provide details on the design and implementation of these core components of the framework.

3.1 Message Broker

The message broker is part of the core components of the MANO Framework provided by the Service Platform.

3.1.1 Features

The message broker is responsible for inter-component communication between plugins who communicate with it asynchronously. This sub-section describes the features expected from the message broker.

3.1.1.1 Topic-based Communication with Publish/Subscribe Pattern

As further explained in D2.2 [6], a topic-based publish/subscribe pattern enables each component to talk to all other components without the need to configure or advertise API endpoints among them. It also makes it easy to introduce additional components that are integrated into the workflow of existing ones without changing their implementation.

The publish/subscribe communication pattern also enables a single message to be directly delivered to multiple receivers, which enables distributing global information in the MANO Framework.

3.1.1.2 Hierarchical Topic Structure

A hierarchical topic structure allows components to have fine-grained control over the information they want to receive.

3.1.1.3 Asynchronous Communication

The communication between the MANO framework components is completely asynchronous to "release components" when handling requests that need time to generate a reply (e.g. resource allocations).

3.1.2 Mano Framework Message Definition

As detailed above, the messages transferred by the message broker have hierarchical structure and are used in an asynchronous manner. There are two patterns of communications:

1. Request / Response
2. Notification

The following subsections present some examples that help describing their format. A full list of the messages used in the current implementation can be found in Appendix D.

3.1.2.1 Request / Response

Example: MANO Plugin Registration Message
The flow of this messaging pattern is shown in Figure 3.1.

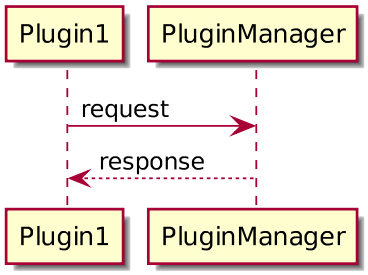

Figure 3.1: MANO Plugin Registration Message

Request

- Request topic: `platform.management.plugin.register`
- Request message properties :
  - `app_id = {plugin-identifier}`
  - `content_type = 'application/json'`
  - `reply_to = {response_topic_name}`
  - `correlation_id = {new UUID to identify this request}`
- Request message header: None (can contain arbitrary key-value pairs)
- Request message body (JSON):

```
{
 "name" : "my_cool_plugin",
 "version" : "v0.1-dev2",
 "description" : "A description of the plugin"
}
```

Response

- Response topic: `platform.management.plugin.register.response`

- Response message properties:
  - `app_id = {plugin-manager-identifier}`
  - `content_type = 'application/json'`
  - `correlation_id = {UUID-from-request-message}`
- Response message header: None
- Request message body (JSON):

```
{
 "status" : "OK",
 "uuid" : "{UUID to identify this plugin registration in following requests}",
 "error": "(e.g. plugin not allowed)"
}
```

3.1.2.2 Notification

Example: MANO Plugin Heartbeat Message The flow of this messaging pattern is shown in Figure 3.2.

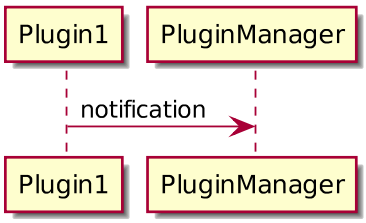

Figure 3.2: MANO Plugin Heartbeat Message

Notification

- Notification topic: `platform.management.plugin.heartbeat`
- Notification message properties:
  - `app_id = {plugin-identifier}`
  - `content_type = 'application/json'`
- Notification message header: **None**
- Notification message body (JSON):

```
{
 "uuid" : "{UUID received from registration call}",
 "state" : "{RUNNING | PAUSED | FAIL | (whatever)}"
}
```

3.1.3 Technologies used

The message broker is based on a RabbitMQ system [21]. This technology was selected based on the requirements and options provided by D2.2 [6].

3.2 Plugin Manager

MANO plugins are the main components of the MANO framework, they are under the control of the platform operator. To keep track of the connected MANO plugins, the *plugin manager* component is used, which is an plugin-like component that communicates with other plugins over the broker, offers a management interface, and interacts with the configuration interface of the message broker. The plugin manager does not provide functionalities related to service management or orchestration tasks and is only responsible for managing MANO plugins as such.

Figure 3.3 shows the high-level architecture of the SONATA MANO framework and how the plugin manager integrates into it. It shows that the plugin manager communicates over the message broker using AMQP just like any other plugin. It also shows a REST-based management interface that is used by a simple command line interface (CLI) tool that can be used by the platform operator.

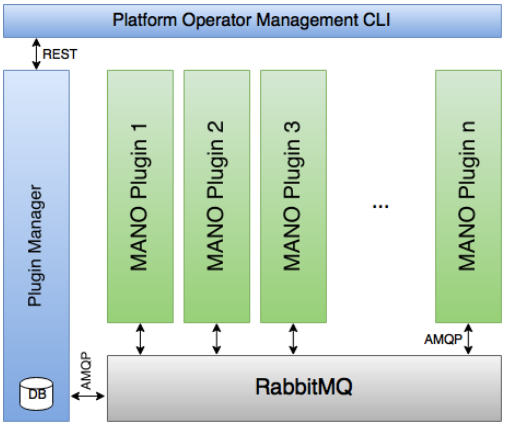

Figure 3.3: High-level view of the SONATA MANO framework with the plugin manager and several connected MANO plugins

3.2.1 Features

The plugin manager implements the functionalities described below.

3.2.1.1 Plugin registration and deregistration

After a plugin is authenticated to the broker, it is allowed to exchange messages with it. Additionally, it has to register itself to the plugin manager so that the system becomes aware of its presence. This registration procedure is shown in Figure 3.5 and uses the standard publish/subscribe mechanism on a dedicated *registration* topic over which each authenticated plugin can publish and the plugin manager subscribes.

3.2.1.2 Bookkeeping

An additional functionality of the plugin manager is maintaining records of active plugins and their current state. Each plugin can query the plugin manager to obtain information about available functionalities in the system. Additionally, the plugin manager broadcasts important system changes to all plugins, e.g., if a new plugin is added to the system.

3.2.1.3 Monitoring

All plugins are executed as independent entities that can fail. The plugin manager is responsible for monitoring the system and check for failed plugins that do not respond anymore. If it detects a failed plugin, it can generate events to inform other components about the issue. This functionality is implemented by a publish/subscribe based heartbeat mechanism.

3.2.1.4 Management

The plugin manager must offer an interface to the platform operator to control and monitor plugins that are connected to the platform. This interface should also allow to manipulate the lifecycle of these plugins. It should, in particular, allow a platform operator to remove plugins from the platform or deactivate (pause) them.

3.2.2 Internal Architecture

The plugin manager consists of several small components shown in Figure 3.4. It is based on our *base plugin framework* described in Section 3.3 and uses the *base plugin's* messaging layer to communicate with the broker. As a result, the plugin manager behave pretty much like a normal MANO plugin but it does not register or deregister to itself.

The package of the plugin manager project contains a central component, called *SonPluginManager* that implements the entire plugin management logic. It receives all registration requests from MANO plugins and creates the corresponding records in the *Model* component. This *Model* component is based on the *mongoengine* library and connects to a MongoDB instance in which all plugin records are persistently stored. This allows the plugin manager to be restarted without loosing information about the platform's system state.

The management interface for the platform operator is implemented as a RESTful interface based on the *Flask* and *flask-restful* library.

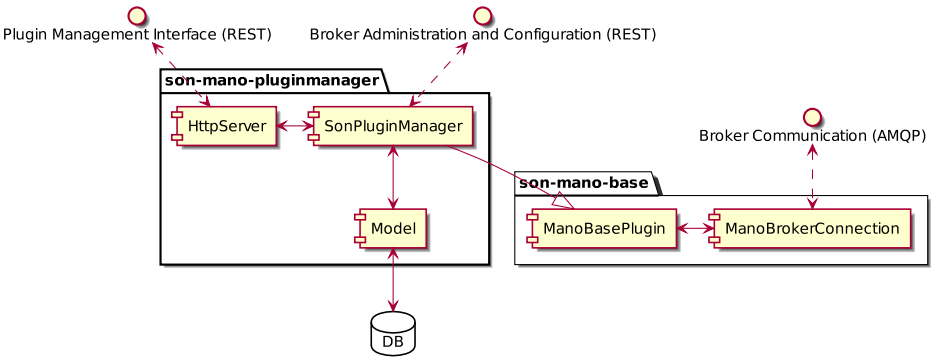

Figure 3.4: Plugin manager componetns and interfaces

3.2.2.1 Data Model

Table 3.1 shows the current data model for a plugin record in the plugin manager's database.

Table 3.1: Plugin table definition

Field name	Field type	Comments
uuid	String	unique

Field name	Field type	Comments
name	String	
version	String	
description	String	
state	String	
registered_at	DateTime	
last_heartbeat	DateTime	

This model will evolve over time to accommodate, e.g., more plugin monitoring data has to be stored. This evolution of the internal data structures is easily supported by the chosen, schema-less database solution *MongoDB*.

3.2.3 Message Sequence Charts

The message sequence chart in Figure 3.5 shows how a plugin is added, registered, and activated with the help of the plugin manager. First, the platform operator configures the new MANO plugin that should be added to the system (1). This configuration will include, e.g., the address and credentials needed to connect to the message broker. After this, the platform operator starts the plugin (2) which initially connects to the message broker using the authentication mechanism of the broker (3)(4).

The plugin then publishes a registration request when it is finally started (5) to the registration topic to which the plugin manger subscribes (6). The plugin manager creates a new record for the plugin and returns a registration response (7) that includes a UUID generated by the plugin manager that is used to identify this MANO plugin instance in other request (8). The plugin is now registered to the system (9). However, it is still in READY state which means that it waits for a signal to start with its operation. This mechanism allows the plugin manager to centrally control the order in which multiple plugins are brought to operation. In the READY state, the new plugin already sends periodic heartbeat messages to the plugin manager to keep the manager informed about its state (10)(11).

The plugin manager can now decide to put the new plugin to operation by sending a lifecycle start event to it (12). This event causes the plugin to change its state and begin with its operation. The plugin continues with sending periodic heartbeat messages to the plugin manager to indicate its health status (13)(14). These message can also be used to collect basic monitoring information inside the MANO framework. In parallel to all these messages does the plugin manager distribute plugin status updates on a dedicated messaging topic whenever an update happens in the system, e.g., a new plugin is added (15). This ensures that all plugins are always aware of the set of active plugins.

Figure 3.6 shows the deregistration procedure of a MANO plugin. This procedure can either be triggered by the plugin itself when it decides do quit or by the platform operator who can request a plugin to be stopped and removed by using the plugin manager's API (2). After the plugin is informed that it will be disconnected from the system, it changes its state to STOPPED and initiates the deregistration procedure by sending the corresponding request to the plugin manager (5) (6). The plugin manger removes the plugin record from its data model and confirms the deregistration request (7) (8). After this, the plugin closes its connection to the broker and terminates. At the same time, the plugin manager broadcasts the information that the set of active plugins has changes to all other plugins via the message broker (11).

3.2.4 API

The plugin manager's REST management interface offers the following endpoints:

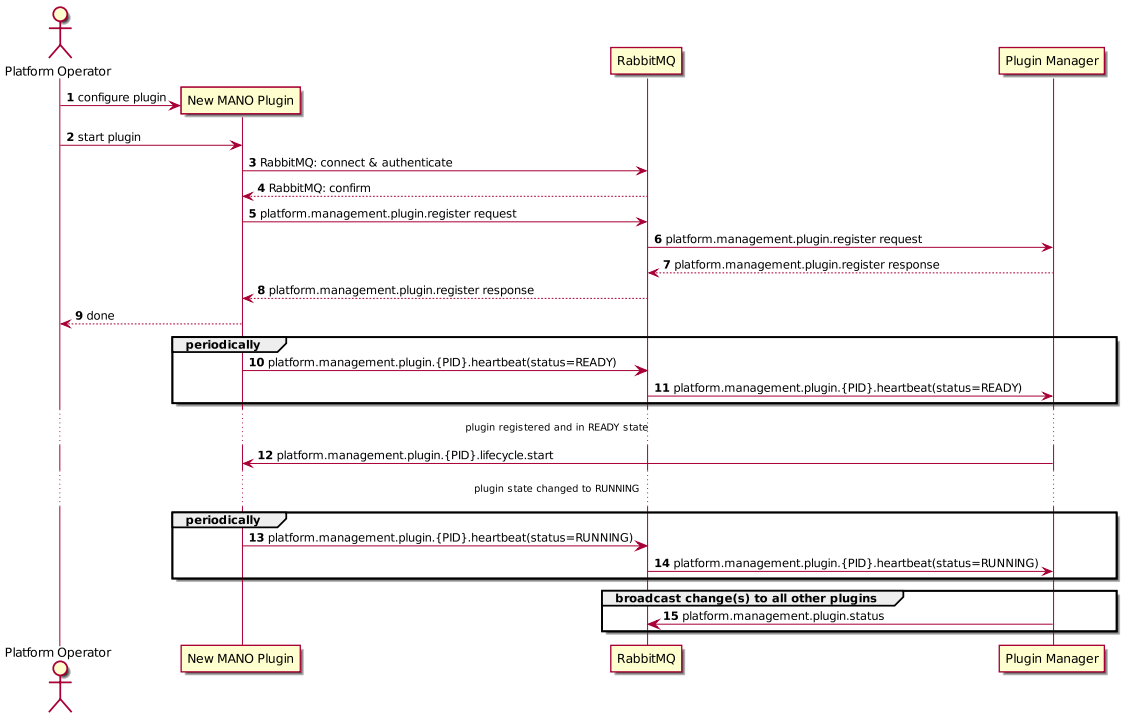

Figure 3.5: MANO plugin registration and activation procedure

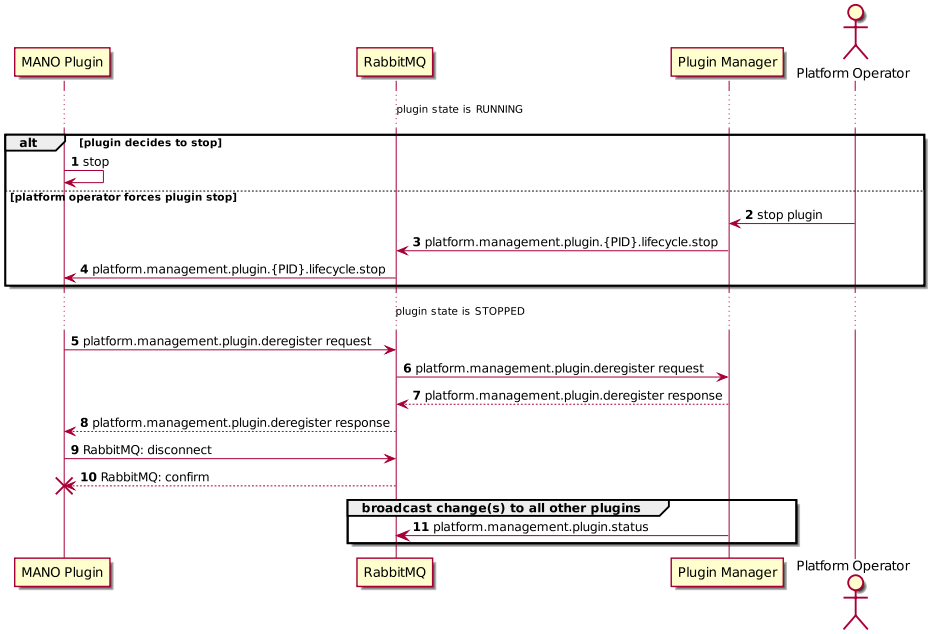

Figure 3.6: MANO plugin de-registration procedure

Table 3.2: REST management API

Endpoint	Method	Description	Returned code(s)
/api/plugins	GET	Receive a list of UUIDs of plugins currently registered in the system.	Ok (200)
/api/plugins/:uuid	GET	Receive status information of the given plugin.	Ok (200), Not found (404)
		Specified plugin not found.	Not found (404)
/api/plugins/:uuid	DELETE	Remove (stop and deregister) a plugin.	Ok (200)
		Specified plugin not found.	Not found (404)
/api/plugins/:uuid/lifecycle	PUT	Change the lifecycle state of a plugin.	Ok (200)
		Specified plugin not found.	Not found (404)
		Target state not available.	Unknown state (500)

3.2.5 User interface

The command line tool for the platform operator offers the following functionality:

- Display a detailed help page.
  - `son-pm-cli -h`
- List UUIDs of all plugins that are currently registered to the platform.
  - `son-pm-cli list`
- Print detailed information about the plugin specified with the given UUID.
  - `son-pm-cli info -u <uuid_of_plugin>`
- Remove (stop and deregister) the plugin specified with the given UUID.
  - `son-pm-cli remove -u <uuid_of_plugin>`
- Change the lifecycle state of the plugin specified with the given UUID.
  - `son-pm-cli lifecycle-pause -u <uuid_of_plugin>`
  - `son-pm-cli lifecycle-start -u <uuid_of_plugin>` (automatically done after registration)

3.2.6 Technologies used

Table 3.3 lists the technologies used in the implementation of the plugin manager.

Table 3.3: Technologies used by the plugin manager component

Name	Type	Purpose
argparse	library	parse command line arguments
flask	library	lightweight webframework for fake gatekeeper REST interface
flask-restful	library	REST extension for flask
MongoDB	external tool	database to store plugin records
mongoengine	library	wrapper to interact with MongoDB
pika	library	AMQP client library for RabbitMQ

Name	Type	Purpose
pytest	library	unittest framework
pytest-runnder	library	support for pytest
Python 3.4	programming language	programming language
RabbitMQ	external tool	message broker for inter plugin communication
requests	library	HTTP client library to interact with REST interfaces

3.2.7 Tests

The plugin manager component implements a set of tests that check its interfaces towards other plugins, e.g., registration procedure, and its management interface towards the platform operator. The test implementation is based on Python's default test module **unittest** and requires a locally running message broker to send messages to the plugin manager. The available test cases are described in the following.

3.2.7.1 PM: Plugin Interface Tests

Registration: Tests the registration procedure by sending a registration request to the PM and validating the registration response.

Initial lifecycle events: Checks that a lifecycle start event is sent to a plugin after it was registered to the PM.

Status update reception: This test subscribes to the plugin status update topic and checks if the PM sends status updates whenever something in the platform was changed.

De-registration: Tests the de-registration procedure by sending a de-registration request and validating its response.

3.2.7.2 PM: Management Interface Tests

Get plugin list Requests a list of registered plugins from the PM using its RESTful management interface. The test performs a HTTP GET request and validates if the received list matches to the state of the platform.

Get plugin info Requests more detailed information about a specific plugin. Validates the response and its contents.

Remove plugin Triggers the remote removal of a previously registered plugin by sending a HTTP DELETE to the management interface. Checks if the plugin was really removed from the platform.

Set plugin lifecycle state Sets the state of a plugin to *PAUSE* by using the lifecycle management interface.

3.3 Base Plugin

The SONATA Base Plugin does not have any NSO functionality, it is skeleton that contains to core functionalities needed in order to connect to the MANO Framework. It is a collection of helper functions and base classes implemented in order to help developers to create new plugins for the SONATA MANO framework. One of its main purposes is to hide the RabbitMQ complexity and the plugin registration procedure from the plugin developer.

3.3.1 Features

The Base Plugin contains two main classes:

1. Abstraction and simplification of the interaction with the messaging broker (please refer to Section 3.1). Therefore it implements different synchronous and asynchronous messaging interfaces on top of RabbitMQ's topic-based publish/subscribe mechanisms.

2. Abstract version of a MANO plugin that already implements basic features, like plugin registration and deregestration as well as heartbeat mechanisms. These abstract plugin can easily be extended by a MANO plugin developer to create a plugin with custom functionality.

3.3.2 Internal Architecture

This section details the internal architecture of the Base Plugin.

3.3.2.1 Interaction with Message Broker

This base interaction with the Message Broker is modeled by two classes: ManoBrokerConnection and ManoBrokerRequestResponseConnection shown in Figure 3.7. The first class contains all mechanics to setup and configure the connection to RabbitMQ. It provides the basic interface to publish/subscribe messages to/from a certain topic. The second class extends the first class and adds some more abstract methods to do asynchronous request/response pattern over the basic publish subscribe system.

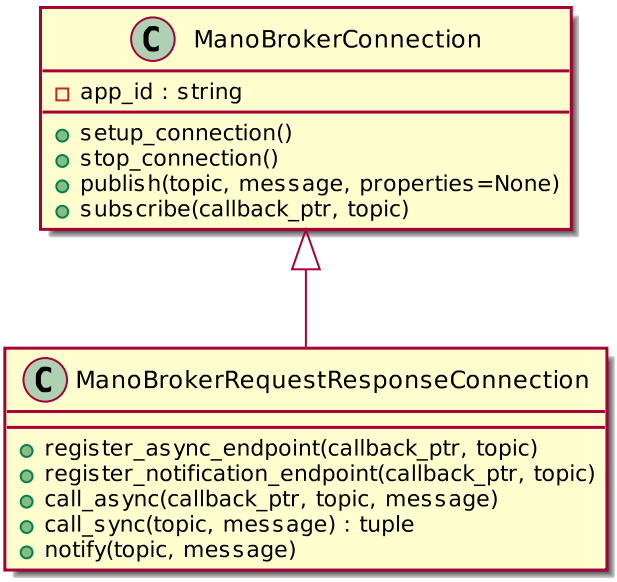

Figure 3.7: Messaging module class overview

ManoBrokerConnection

- `app_id` - Name of the component that interacts with the broker. Typically set to name of the plugin.

- `setup_connection()` - Connect to the RabbitMQ broker. Done automatically if ManoBasePlugin class is used to implement a plugin.

- `stop_connection()` - Disconnect from the RabbitMQ broker.
- `publish(topic, message, properties=None)` - Publish a message to the given topic. Optional: RabbitMQ message properties.
- `subscribe(callback_ptr, topic)` - Subscribe to the given topic and call the callback_ptr functions whenever a message is received.

ManoBrokerRequestResponseConnection

- `register_async_endpoint(callback_ptr, topic)` - Register (subscribe) to a certain topic with an asynchronous endpoint. This means that the callback function is called when a message is received on the given topic. The return value of this function is then automatically encapsulated in a second RabbitMQ message send back as a response on the same topic. The callback function is executed in a new thread.
- `register_notification_endpoint(callback_ptr, topic)` - Basically the same as subscribe. Does not send any response after callback has finished. The callback function is executed in a new thread.
- `call_async(callback_ptr, topic, message)` - Sends the given message to the given topic. Calls the given callback function when the corresponding response is received. This is like an asynchronous RPC call. Attention: Due to the pub/sub semantics, multiple response messages can be received from multiple senders.
- `call_sync(topic, message)` - Sends the given message to the given topic to which an asynchronous endpoint should be registered in another MANO plugin. This is a synchronous call: It blocks until the first result is received and returns the result as a tuple of the format: (ch, method, props, body) where body contains the message body of the response.
- `notify(topic, message)` - Basically the same as publish. Fire and forget messaging.

3.3.2.2 Plugin Base Functionalities

The base functionalities required by any plugin are modeled by a single class shown in Figure 3.8. This class should be extended by new MANO plugins and contains all the needed logic to register and deregister the plugin. It also implements all the needed logic to listen to plugin lifecycle events sent by the plugin manager and offers event methods for them.

- `manoconn` - pointer to a ManoBrokerRequestResponseConnection object that is created in the constructor and hides all the RabbitMQ functionality behind a simple interface that makes the communication with other MANO framework components much simpler.
- `name` - each plugin has a name.
- `state` - each plugins is in a certain state (RUNNING|PAUSED|STOPPED|FAILED).
- `version` - a plugin should have a version.
- `register()` - registers the plugin to the plugin manager. The plugin automatically starts to send periodic heart beat messages (additional thread) after the registration has completed.
- `deregister` - de-registers the plugin in the plugin manager.

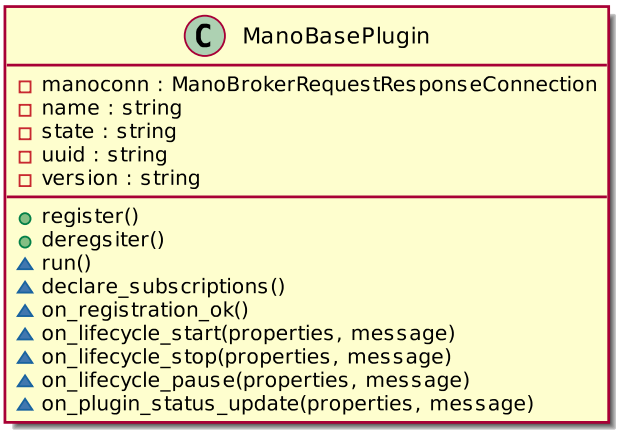

Figure 3.8: Plugin module class overview

- `run()` - Called after the registration procedure has been triggered. Does not contain any code and should be overridden by a custom MANO plugin implementation. As the registration procedure might not have finished when this method is called, the developer can use lifecycle events (on-lifecycle-start or on-registration-ok) to run code *after* the registration procedure.
- `declare_subscriptions()` - called during the initialization phase. Custom topic subscriptions should be placed within this method.
- `on_registration_ok()` - called after registration procedure completed. To be overridden.
- `on_lifecycle_start(properties, message)` - called when lifecycle start event is received from plugin manager. To be overridden. Arguments: RabbitMQ message and properties.
- `on_lifecycle_pause(properties, message)` - called when lifecycle pause event is received from plugin manager. To be overridden. Arguments: RabbitMQ message and properties.
- `on_lifecycle_stop(properties, message)` - called when lifecycle stop event is received from plugin manager. To be overridden. Arguments: RabbitMQ message and properties.
- `on_plugin_status_update(properties, message)` - called when plugin manager broadcasts an update message with the list of plugins registered to the system. To be overridden. Arguments: RabbitMQ message and properties.

3.3.3 Technologies used

Table 3.4 lists the technologies used in the implementation of the MANO framework's base plugin.

Table 3.4: Technologies used by the base plugin component

Name	Type	Purpose
pika	library	AMQP client library for RabbitMQ
pytest	library	unittest framework
pytest-runnder	library	support for pytest
Python 3.4	programming language	programming language
RabbitMQ	external tool	message broker for inter plugin communication

3.3.4 Tests

There are a couple of unit tests implemented for the base plugin and especially its messaging component. The test implementation is based on Python's default test module *unittest* and requires a locally running message broker to send messages between the test components. The available test cases are described in the following.

Broker connection Test the functionality to establish a basic connection to a RabbitMQ message broker instance.

Basic publish/subscribe Uses the basic publish/subscribe methods of the broker connection and checks if messages could be exchanged with them.

Asynchronous request/response Uses the asynchronous request/response API provided by ManoBrokerRequestResponseConnection to send a message to a test echo endpoint. Checks if the response message has the same content as the request message.

Synchronous request/response Uses the synchronous request/response API provided by ManoBrokerRequestResponseConnection to send a message to a test echo endpoint. Checks if the response message has the same content as the request message.

Notification Uses the one-way notification API provided by ManoBrokerRequestResponseConnection to send a message to a test endpoint. Checks if the message arrives at the endpoint.

4 Orchestration Plugins

As defined in the Description of Work [7], task T4.3 coordinates the development of a rich set of plugins that provide the set of features needed for the NSO. As defined in Deliverable 2.2 [6] and presented in Figure 1.1, SONATA's extensible MANO framework is built out of a set of loosely coupled components, called MANO plugins. Each of these plugins implements a limited, well-defined part of the overall management and orchestration functionality. The following sections provide details on the implemented plugins and plugins that have been discussed and designed for future implementation.

4.1 Service Lifecycle Plugin

The Service Lifecycle Plugin, or Service Lifecycle Manager(SLM), is the plugin that handles all decisions concerning the lifecycle of the services. Through the message broker, it receives requests to deploy a service. Based on the Network Service Descriptor (NSD) and the virtual network function descriptors (VNFDs), which are part of the request, the SLM decides where to place it and how to scale the service. The SLM translates the request into a message for the Infrastructure Adaptor, the plugin that is responsible for the actual deployment. When the Infrastructure Adaptor reports back the status and the specifics of the deployment, the SLM uses this data to form a Network Service Record (NSR), and a Virtual Network Function Record (VNFR) for each VNF involved and stores these records in the repositories through a REST API. Based on the information in the descriptors, the SLM informs the Monitoring Manager which metrics concerning the service need to be monitored and which tresholds of these metrics are to be reported to the SLM. When one of these metrics reaches its treshold, the SLM adapts the lifecycle of the service, based on the information in the descriptor.

4.1.1 Features

The Service Lifecycle Manager implements the features described next.

Registration: When activated, the SLM will try to register to the plugin manager. When deactivated, the SLM will deregister from the plugin manager;

Heartbeat: When running and registered to the plugin manager, the SLM sends out a heartbeat to indicate to the plugin manager that it is still running;

Handle service requests: The SLM is the access point for the MANO framework for the service requests. Requests are received from the Gatekeeper through the message broker in the form of a python dictionary. In this dictionary, the network service descriptor can be found under the 'NSD' key, and the virtual network function descriptors in a list under the 'VNFDs' key. In Figure 4.2, this resembles step 1 and 2;

Check availability of resources: Upon receiving a new service request, the SLM checks with the Infrastructure Adaptor whether the required resources to deploy this new service are available. See steps 3-6 in Figure 4.2;

Deploy a service: When the resources needed to deploy the new service are available, the SLM sends a request to the Infrastructure Adaptor to deploy the service. See steps 7-10 in Figure 4.2;

Store deployed services: When the Infrastructure Adaptor replies to the SLM on a service deploy request, the SLM stores the data concerning this deployment in the repositories if the deployment was successful. This data is stored as a network service record and a virtual network function record for each VNF in the service. See step 11 and 12 in Figure 4.2;

Inform Monitoring Manager: When a new service is deployed, the SLM informs the Monitoring Manager which metrics regarding this service it should monitor, what their tresholds are and on which broker topic it should send a message once a treshold has been reached. The SLM deducts this information from the descriptors that are part of the service request. See step 13 and 14 in Figure 4.2;

Act on monitoring feedback: When the SLM gets feedback from the Monitoring Manager on a treshold being reached, the SLM deducts from the descriptors which action to take upon this event. Possible actions are recalculating the scaling, the placement, terminating the service, etc.. See step 15 and 16 in Figure 4.2;

Internal Architecture: The SLM is a plugin based on the *son-mano-base* library, which was described earlier. It also uses the *son-mano-base* messaging layer to communicate with the broker. The central component of the SLM is the *ServiceLifecycleManager*, shown in Figure 4.1 with all its interfaces. For communication with the repositories and outgoing information to the Monitoring Manager, it uses a REST API, for other communication (Gatekeeper, SSMS, Infrastructure Adaptor, incoming Monitoring Manager information, ...) it uses the message broker connection.

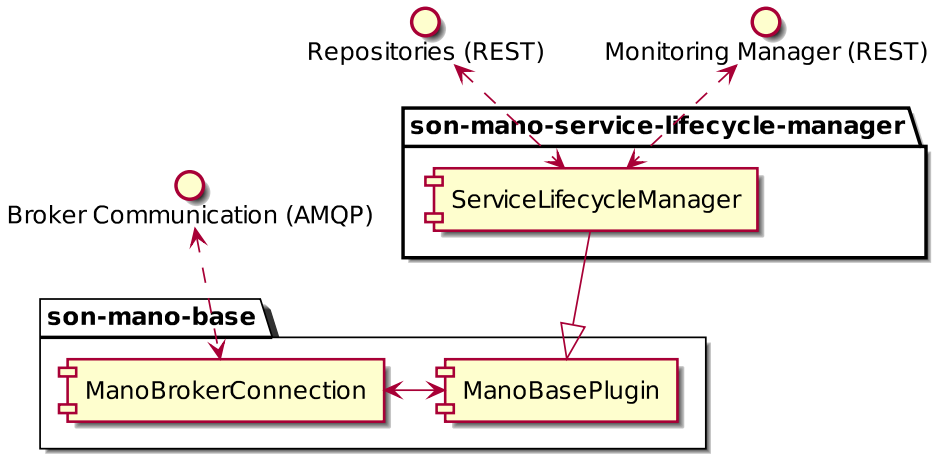

Figure 4.1: Service Lifecycle Manager and interfaces

4.1.2 Message Sequence Charts

The Message Sequence Chart in Figure 4.2 shows the order of events and the types of communication involved in deploying a new service (from the SLM point of view). When communication is done through the message broker, the specific topics are listed. For the Message Sequence involving the

registration of a plugin and its hartbeat, see the section on the plugin manager. For additional information on specific steps, see the Features section.

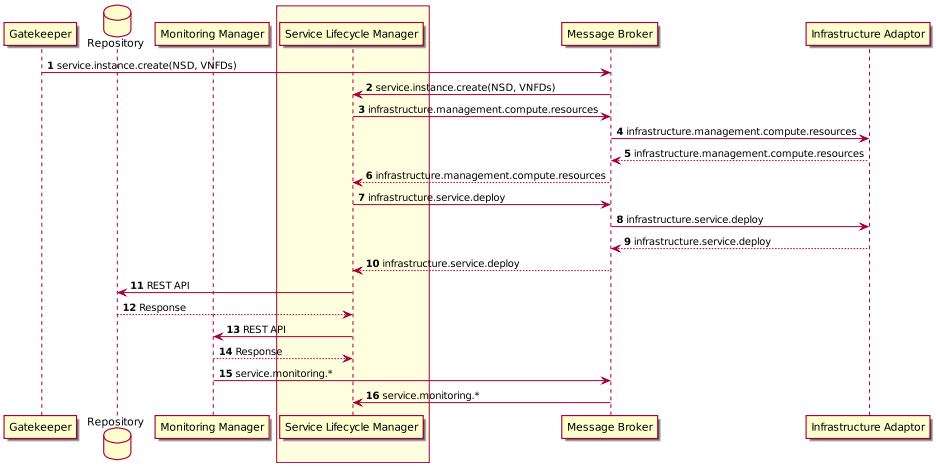

Figure 4.2: Deploying a service

4.1.3 API

Communication with the Repositories or the Monitoring Manager through the REST interface is always initiated by the SLM. For the API of this communication, see their respective sections. Requesting the SLM to deploy a new service is done by sending a message (on the RabbitMQ broker) on the service.instance.create topic with the following characteristics:

Table 4.1: SLM service request API

Properties	Body

`app_id = {GK}`
`content_type = 'application/yaml'`
`reply_to = service.instance.create`
`correlation_id = {UUID to identify this request}`
	`NSD:`
	`  descriptor_version:`
	`  ...`
	`VNFD1:`
	`  descriptor_version:`
	`  ...`
	`VNFD2:`
	`  descriptor_version:`
	`  ...`

The responses from the SLM to a service request are messages posted on the instance.service.create topic with the following characteristics:

Table 4.2: SLM service response API

Timing	Properties	Body

Directly after the request is received	`app_id = {SLM}`
	`content_type = 'application/yaml'`
	`correlation_id = {UUID request message}`
		`status: "INSTANTIATING"|"ERROR"`
		`error: "any error message"|null`
		`timestamp: {timestamp}`

Timing	Properties	Body

After service is instantiated or when an error occurs	`app_id = {SLM}`
	`content_type = 'application/yaml'`
	`correlation_id = {UUID request message}`
		`uuid: "{uuid new service instance}"`
		`status: "READY"|"ERROR"`
		`error: "any error message"|null`
		`timestamp: {timestamp}`
		`NSR:`
		`  {network service record}`
		`VNFR1:`
		`  {VNF record}`
		`VNFR2:`
		`  ...`

4.1.4 User interface

The SLM has no user interface.

4.1.5 Technologies used

Table 4.3 lists the technologies used in the implementation of the service lifecycle manager..

Table 4.3: Technologies used by the service lifecycle manager component

Name	Type	Purpose
pika	library	AMQP client library for RabbitMQ
pytest	library	unittest framework
pytest-runnder	library	support for pytest
Python 3.4	programming language	programming language
RabbitMQ	external tool	message broker for inter plugin communication
requests	library	HTTP client library to interact with REST interfaces
PyYAML	library	library to convert data between yaml files and python objects

4.1.6 Tests

The SLM component implements a set of tests to check its behaviour when interacting with other components. The test implementation is based on Python's default test module unittest and requires a locally running message broker and plugin manager. The available test cases are described below.

4.1.6.1 SLM: Plugin Interface Tests

Registration Tests whether the SLM registers to the plugin manager when activated.

Heartbeat Tests whether the SLM sends out a heartbeat message when running correctly.

4.1.6.2 SLM: Deploying a Service Tests

Response to a New Service Request Checks the reaction of the SLM when it receives a new service request from the gatekeeper. The SLM should request resource availability from the infrastructure adaptor.

Response to a Resource availability message Checks the reaction of the SLM when it receives resource availability information from the infrastructure adaptor. The SLM should make a service deployment request to the infrastructure adaptor if enough resources are available.

Response to a Deployed Service Checks the reaction of the SLM when it receives word from the infrastructure adaptor that a new service has been deployed. The SLM should store the correct records in the repositories.

4.2 Future Plugins

As detailed in D2.2 [6] SONATA's SP intends to include several default plugins that will be an integral part of it and will provide essential NSO functionalists. The implementation of plugins started with the core functionality of a network service orchestrator -- the Service Lifecycle Manager that is detailed in Section 4.1. Design work has started on more core plugins. The following sections provide details on those plugins.

4.2.1 Service Specific Manager Plugin

The concept of SONATA's Service Specif Manager (SSM) Plugin was widely detailed in D2.2 [6]. A SSM is responsible for managing one or more services belonging to exactly one platform customer. If no custom event handlers are needed, the service could rely on the default SSMs that will be part of the SONATA platform.

4.2.1.1 Features

This section details the user stories further defined for the SSM.

Service placement calculation

Calculates the placement for the service, without dealing with the required scaling operations (i.e., it does not modify the service structure). Service scaling needs to be done before, if necessary.

- Input:
  - (Modified) topology/resource information
  - NSD
  - Service instance record
- Functionality:
  - Calculate the desired placement for the service without making any changes to the service graph
- Output:
  - Mapping of each VNF in the NSD to a network node and mapping of the corresponding paths among them to network links.

Service Scaling

Creating and updating the service graph. It should be called for initial service graph calculation as well as scaling out/in operations initiated by an FSM to check for further scaling requirements. It does not deal with scaling up/down operations. The updated service graph needs to be passed to a placement SSM.

- Input:

  - NSD
  - Service instance record
  - Trigger message specifying what needs to be scaled out/in (e.g., as calculated by an FSM)

- Functionality:
  - Calculate the initial service graph based on the NSD
  - Check feasibility of the scaling operation -

- Output:
  - Scaling instructions to corresponding FSMs
  - Updated service instance record

Service Scaling and Placement Calculation

Creating and updating the service graph (should be called for initial service graph calculation as well as scale out/in operations initiated by an FSM). It also is responsible for calculating the placement for the created/updated service graph.

- Input:
  - NSD
  - Service instance record
  - Trigger message specifying what needs to be scaled out/in (e.g., as calculated by an FSM)

- Functionality:
  - Calculate the desired placement for the service without making any changes to the service graph
  - Calculate the initial service graph based on the NSD
  - Check feasibility of the scaling operation
  - Calculate further scaling requirements as a result of scaling out/in the requested VNFs

- Output:
  - Mapping of each VNF in the NSD to a network node and mapping of the corresponding paths among them to network links (format?)
  - Scaling instructions to corresponding FSMs
  - Updated service instance record

Service Lifecycle Management

Responsible for leading the order of lifecycle events for the service, e.g.:

- Starting the SSMs belonging to the service
- Triggering the placement
- Triggering manual scaling for the service

Different services might have different preferences for the order of these operations, which can be expressed by lifecycle management SSMs.

- Input:
  - NSD
  - Trigger message including the current state of the service lifecycle
- Functionality:
  - Lead the service through its lifecycle. It should be the first SSM that is instantiated from the service package. Starts with instantiating the rest of required SSMs for the service. Based on the current state of the service lifecycle, it publishes to the right topics to trigger the next lifecycle event.
- Output:
  - Trigger messages for lifecycle events

Service Monitoring

This SSM is plugged into service monitoring executive plugin and exposes "monitoring" capability. It is responsible for collecting and operating on service-related monitoring data.

- Input:
  - Monitoring data
- Functionality:
  - Collect monitoring data related to the service and process it based on pre-defined rules
- Output:
  - Trigger messages for scaling operations
  - Feedback to service developer regarding the service

4.2.1.2 SSM Development

Each SSM can subscribe/publish to the message bus provided by the corresponding executive plugin. The information required by the SSMs and the communication of the SSMs with the rest of the MANO framework is done through the executive plugin. To keep the actual implementation of SSM functionalists as flexible as possible, we need to keep the communication between the SSM and executive plugin simple. Ideally, the executive plugin calls the SSM with required information, the SSM does its calculations and returns the results, which can be a success announcement including the calculated results or a failure announcement including a description of the problem that has occurred. An example of the planned role of a SSM can be found in Figure 4.2

4.2.2 Function Specific Manager Plugin

The concept of SONATA's Function Specif Manager (FSM) Plugin was widely detailed in D2.2 [6]. A FSM is responsible for managing lifecycle events of one or more network functions belonging to exactly one platform customer. If no custom event handlers are needed, the VNF could also rely on the default FSM that will be part of the SONATA platform.

4.2.2.1 Features

This section details the features defined for the FSM.

VNF Deployment: Handling the deployment of the VNF. This is relevant when there is a specific order of deployment of the different VNF components, or if special steps between components is required;

VNF Scaling: Handle the scaling of VNF. This is relevant specially for a customized way of scaling a VNF vertically, by adding extra resources such as CPU and memory. Another use case is when special steps are needed during the scaling process;

VNF Shutdown:

Handling the Shutdown of the VNF. This is relevant when there is a specific order of shutdown of the different VNF components, or if special steps between components is required.

4.2.2.2 FSM development

Each FSM can subscribe/publish to the message bus provided by the corresponding executive plugin. The information required by the SSMs and the communication of the SSMs with the rest of the MANO framework is done through the executive plugin. To keep the actual implementation of SSM functionalists as flexible as possible, we need to keep the communication between the SSM and executive plugin simple. Ideally, the executive plugin calls the SSM with required information, the SSM does its calculations and returns the results, which can be a success announcement including the calculated results or a failure announcement including a description of the problem that has occurred.

5 Catalogues, Repositories, and Supporting Functionalities

This section describes the components that provide supporting functionalities to the Orchestrator prototype, such as Catalogues, Repositories, and Infrastructure Abstraction. For each, we provide details about their features, interfaces, workflows and implementation.

5.1 Service Platform (SP) Catalogues

The SONATA Service Platform Catalogues provides management and storage of Package Descriptors (PD), Network Service Descriptors (NSD), VNF Descriptors (VNFD), Service/Function Specific Management Descriptors (SSMDs/FSMDs), and network-service packages. In fact, the SP Catalogues are collections of PD catalogue, NSD Catalogue, VNFD catalogue and SSMD/FMSD catalogue, as shown in Figure 5.1. Only the Gatekeeper populates the SP Catalogues with valid descriptors submitted to it. The stored descriptors are used to instantiate new services or to provide developer community access to these descriptors. The subsequent subsections describe the features, APIs, internal architecture, functional workflows, used technologies and unit tests for the SP Catalogues.

5.1.1 Features

The SP Catalogues have the following features:

Storage: The SP Catalogues are responsible for storing the PDs, NSDs, VNFDs, and SSMDs/FSMDs in the database. These descriptors are required by other SONATA components for development, selection, and instantiation of network services. It is worth mentioning that the SP Catalogues also provide storage and retrieval of package file, i.e., the ZIP file.

Retrieval: The SP Catalogues enable the retrieval of stored PDs, NSDs, VNFDs, and SSMDs/FSMDs via a REST API. It allows both a single or bulk retrieval of descriptors. The details of the REST API is given in Table 5.1.

Update: The SP Catalogues allow updating PDs, NSDs, VNFDs, and SSMDs/FSMDs via the REST API. The details of the REST API is given in Table 5.1.

Status: The SP Catalogues allow checking or setting the status of a PD, NSD, VNFD, and SSMD/FSMD via the REST API. The status reflects either the descriptor is active or inactive for service instantiation. The details of the REST API is given in Table 5.1.

Delete: The REST API of SP Catalogues allow deleting a particular PD, NSD, VNFD, and SSMD/FSMD as given in Table 5.1.

5.1.2 API

This sub-section the RESTful API of the SP Catalogues. This API is described in Table 5.1.

Table 5.1: SP Catalogues REST management API

Uri	Method	Description	Returned code(s)
/catalogues/network-services	GET, POST	List all the available NSDs or store a NSD in the SP Catalogues.	Ok (200), Not Found (404), Unsupported Media Type (415)
/catalogues/network-services	GET, PUT, DELETE	List, update, or delete a NSD using the naming trio, i.e., name, vendor, & version.	Ok (200), Unsupported Media Type (415)
/catalogues/network-services?status=:status	GET	List all NSDs with status *active* or *inactive*.	Ok (200)
/catalogues/network-services?version=last	GET	List only the last version for all NSDs.	Ok (200)
/catalogues/network-services/:id	GET, PUT, DELETE	List, update, or delete a NSD using the UUID.	Ok (200), Not Found (404), Unsupported Media Type (415)
/catalogues/network-services/:id?status=:new_status	PUT	Set status of a NSD using the UUID.	Ok (200), Not Found (404), Unsupported Media Type (415)
/catalogues/vnfs	GET, POST	List all the available VNFDs or store a NSD in the SP Catalogues.	Ok (200), Not Found (404), Unsupported Media Type (415)
/catalogues/vnfs	GET, PUT, DELETE	List, update, or delete a VNFD using the naming trio, i.e., name, vendor, & version.	Ok (200), Unsupported Media Type (415)
/catalogues/vnfs?status=:status	GET	List all VNFDs with a given `status`.	Ok (200)
/catalogues/vnfs?version=last	GET	List only the last version for all VNFDs.	Ok (200)
/catalogues/vnfs/:id	GET, PUT, DELETE	List, update, or delete a VNFD using the UUID.	Ok (200), Not Found (404), Unsupported Media Type (415)
/catalogues/vnfs/:id?status=:new_status	PUT	Set status of a VNFD using the UUID.	Ok (200), Not Found (404), Unsupported Media Type (415)
/catalogues/packages	GET, POST	List all the available package descriptors or store a package in the SP Catalogues.	Ok (200), Not Found (404), Unsupported Media Type (415)
/catalogues/packages	GET, PUT, DELETE	List, update, or delete a package descriptor using the naming triplet, i.e., name, vendor, & version -- see Appendix A.	Ok (200), Unsupported Media Type (415)
/catalogues/packages?status=:status	GET	List all packages with status *active* or *inactive*.	Ok (200)
/catalogues/packages?version=last	GET	List only the last version for all packages.	Ok (200)
/catalogues/packages/:id	GET, PUT, DELETE	List, update, or delete a package descriptor using the UUID.	Ok (200), Not Found (404), Unsupported Media Type (415)
/catalogues/packages/:id?status=:new_status	PUT	Set status of a package using the UUID.	Ok (200), Not Found (404), Unsupported Media Type (415)

Uri	Method	Description	Returned code(s)
/catalogues/zip-packages/:id	GET	List a particular network service package (ZIP file) using the UUID.	Ok (200), Not Found (404)
/catalogues/zip-packages	POST	Store a new network service package (ZIP file).	Ok (200), Unsupported Media Type (415)

5.1.3 Internal Architecture

The SP Catalogues are collections of NSD, VNFD, and SSMD/FSMD catalogues, as shown in Figure 5.1. The SP catalogues not only serve as the storage point for all the NSDs, VNFDS, SSMDs/FSMDs, etc. in the SONATA SP but also enable access to these descriptors for other developers. In terms of implementation, a mongoDB acts as the main storage and each catalgoue, i.e., NSD, VNFD, and SSMD/FSMD maps to a separate collection within the database. Hence, the schema of each collection in a database is adaptable to the structure of the descriptor stored. A REST API, as front-end to the database, enables access for Gatekeeper to retrieve or store descriptors in the SP Catalogues. The SP catalogue accepts NSDs, VNFDs, SSMDs/FSMDs, etc. in both YAML and JSON format.

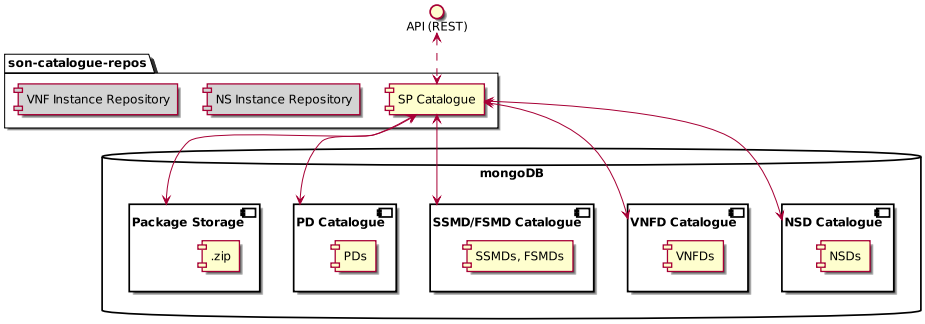

Figure 5.1: SP Catalogues components and interfaces

5.1.4 Functional Workflows

The functional workflows with the SP Catalogues are described in the following sub-sections.

5.1.4.1 Descriptor Storage in SP Catalogues

One of the main functionalities of the SP Catalogues is the storage of NSDs, VNFDs, PDs, and SSMDs/FSMDs. The workflow illustrating how an NSD, after its development in the SDK, is stored and published in the SP Catalogues is shown in Figure 5.2. The storage of an NSD amounts to its publication as after storage in the SP Catalogues, the NSD is also available to other developer with appropriate credentials. The storage request is received at the Gatekeeper and it is only after a successful validation, as shown in Step 3, the Gatekeeper performs a POST operation to store it in the SP Catalogues. A similar workflow can be envisioned for storing VNFDs, PDs, SSMDs/FSMDs, and network service package files.

5.1.4.2 Network Service Display and Instantiation

The workflow shown in Figure 5.3 illustrates how SP Catalogues are involved in displaying the available network services at the SP and in the service instantiation process. Figure 5.3 is an

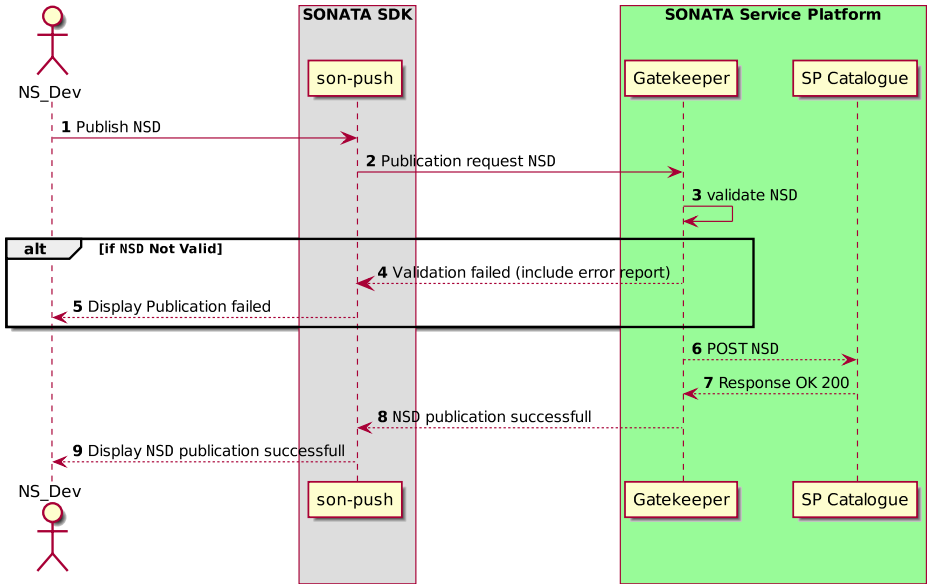

Figure 5.2: Descriptor and Package Storage in SP Catalogues.

extended version of Figure 2.35. Steps 1 - 6 illustrate the process of an End User retrieving the list of all available network services in a particular SONATA SP using the BSS. These steps are similar for listing the available VNFs and network service packages. Steps 7 - 12, show the information retrieval for a particular network service. The high-level message exchange for a network service instantiation request is shown by Step 13 -18. As mentioned earlier and shown in Steps 3, 4, 9, 10, 15, and 16, the Gatekeeper is the only entity retrieving information from the SP Catalgoues. The interactions in Steps 3, 4, 9, 10, 15, and 16 are carried out by using the REST API, as described earlier.

5.1.5 Technologies Used

Table 5.2 lists the technologies used in the implementation of the SP Catalogues.

Table 5.2: Technologies used in the SP Catalogues API

Name	Type	Purpose
addressable	library	Interpreter of query parameters in the URL
ci_reporter_rspec	library	Connects CI::Reporter to RSpec
json	library	JSON implementation as a Ruby extension in C.
json_schema	library	A JSON Schema V4 and Hyperschema V4 parser and validator.
thin	library	Application server
rack-test	library	Rack::Test is a small, simple testing API for Rack apps
rake	library	Dependency manager
rest-client	library	REST client
rspec	framework	Tests framework
rspec-mocks	library	To be used by **rspec**
rspec-its	library	To be used by **rspec**
rubocop	library	For styling checking
rubocop-checkstyle_formatter	library	To be used by **rubocop**
ruby	programming language	Programming language

Name	Type	Purpose
mongoid	framework	ODM (Object Document Mapper) Framework for MongoDB
mongoid-pagination	library	A simple pagination module for Mongoid
mongoid-grid_fs	library	Mongoid/Moped implementation of the MongoDB GridFS specification
sinatra	framework	Web application framework
sinatra-contrib	library	Add-ons for the **sinatra** web-app framework
web mock	library	For mocking external services
yard	tool	Documentation generation tool for the Ruby programming language

5.1.6 Unit Tests

The SP Catalogues module interacts with the Gatekeeper module only. The following tests are implemented to ensure that the Gatekeeper is able to retrieve, store, and update descriptors in the SP Catalogues. In order to carry out the unit tests, an external mongoDB is required. In this subsection, unit tests for NSD are defined, however, similar test are performed for PDs, VNFDs, and packages. The existence of PD, NSD, VNFD, and package files is prerequisite to the following tests.

5.1.6.1 Retrieval Test

Tests the retrieval of a signle NSD, VNFD or a package as well as bulk retrieval of NSD, VNFD or packages by performing a GET using the API defined above.

1. List API methods by accessing the root:
   - **Execution:** `curl -X GET localhost:4002/catalogues/network-services`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of methods available
2. Retrive an NSD with naming trio:
   - **Execution:** `curl -X GET localhost:4002 \`

   `/catalogues/network-services?name=eu.sonata-nfv.service-descriptor\`

   `&vendor=sonata-demo&version=0.2`
   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the NSD limited by 10 per page
3. Retrive a NSD with UUID:
   - **Execution:** `curl -X GET localhost:4002 \`

   `/records/catalogues/network-services/32adeb1e-d981-16ec-dc44-e288e80067a1`
   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the NSD with UUID 32adeb1e-d981-16ec-dc44-e288e80067a1
4. Retrive all active NSDs:
   - **Execution:** `curl -X GET \`

   `localhost:4002/records/catalogues/network-services?status=active`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of all active NSDs in the SP Catalogues

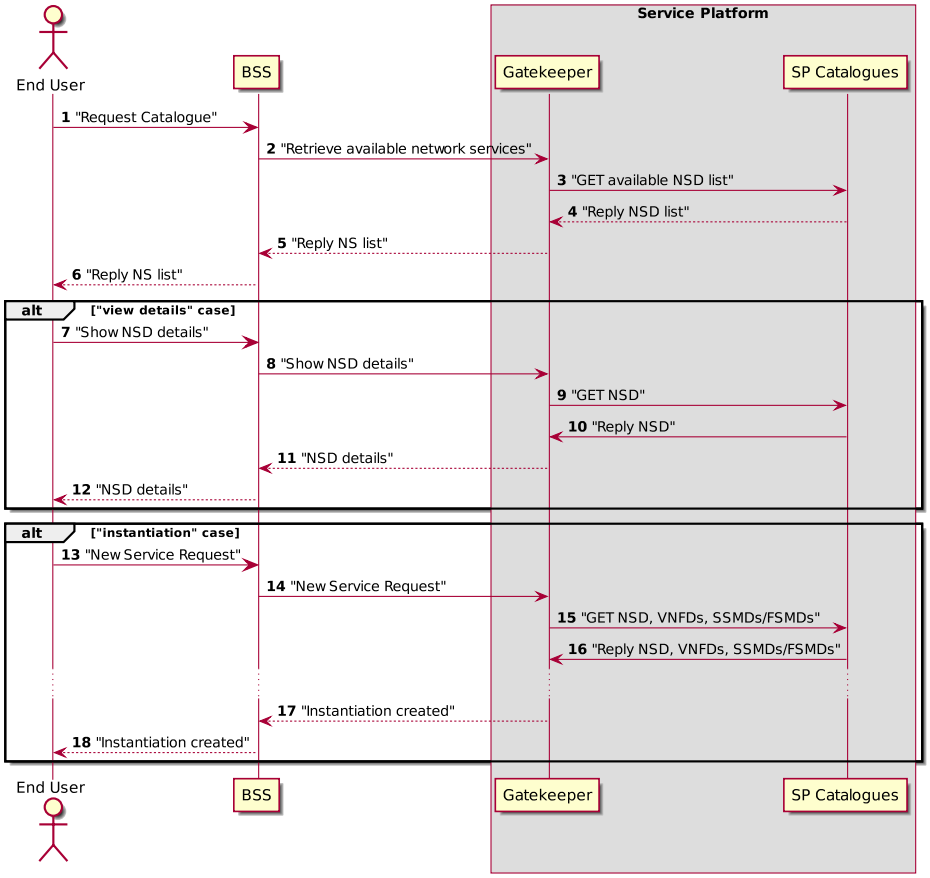

Figure 5.3: Role of SP Catalogues in service display and instantiations.

5.1.6.2 Store Test

Tests the storage of a single NSD, VNFD or a package by performing a POST using the API defined above.

1. Submit a new nsd:
   - **Set-up:** have a valid nsd in a json format (`nsr-example.json`, in the folder `./spec/fixtures`)
   - **Execution:** `curl -X POST -d @nsd-example.json \`

   `localhost:4002/catalogues/network-services --header "Content-Type:application/json"`
   - **Expected:** HTTP code 200 (OK) is returned

5.1.6.3 Update Test

Tests updating an NSD, VNFD, or a package by performing a PUT as defined in the API above. It also tests updating the status of an NSD, VNFD, or a package.

1. Update a NSD with UUID:
   - **Execution:** `curl -X PUT localhost:4002 \`

   `/catalogues/network-services/32adeb1e-d981-16ec-dc44-e288e80067a1`
   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the NSD with a new UUID
2. Update status of an NSD with UUID:
   - **Execution:** `curl -X PUT localhost:4002 \`

   `/catalogues/network-services/32adeb1e-d981-16ec-dc44-e288e80067a1?status=inactive`
   - **Expected:** HTTP code 200 (OK) is returned,

5.1.6.4 Delete Test

Tests the removal of an NSD, VNFD or a package by performing a DELETE operation as defined in the API.

1. Delete an NSD with UUID given:
   - **Execution:** `curl -X DELETE localhost:4002 \`

   `/catalogues/network-services/32adeb1e-d981-16ec-dc44-e288e80067a1`
   - **Expected:** HTTP code 200 (OK) is returned

5.2 Service Platform Repositories

The Service Platform (SP) Repositories were designed to hold the information about services, functions, monitoring and resources, provided by the Infrastructure Abstraction (see Section 5.3) for every instantiation.

5.2.1 Message Sequence Charts

This section contains the MScs that show the order of events and the types of communication involved in the instantiation process and the update of a network service.

5.2.1.1 Instantiation process

Figure 5.4 illustrates how the Service Platform Repositories are involved in the service instantiation process. The process starts when the Gatekeeper (GK) triggers a new service creation request to the Service Lifecycle Managemet (SLM). The SLM checks the resources availability and deploys the VNFs that compose the Network Service (NS), saving each VNF instance record in the VNF Repository. When the SLM finishes the deployment process it updates the NS instance in the NS Repository. Before notifying that the instantiation is done to the GK, the SLM subscribes the monitoring parameters in the Monitoring Repository.

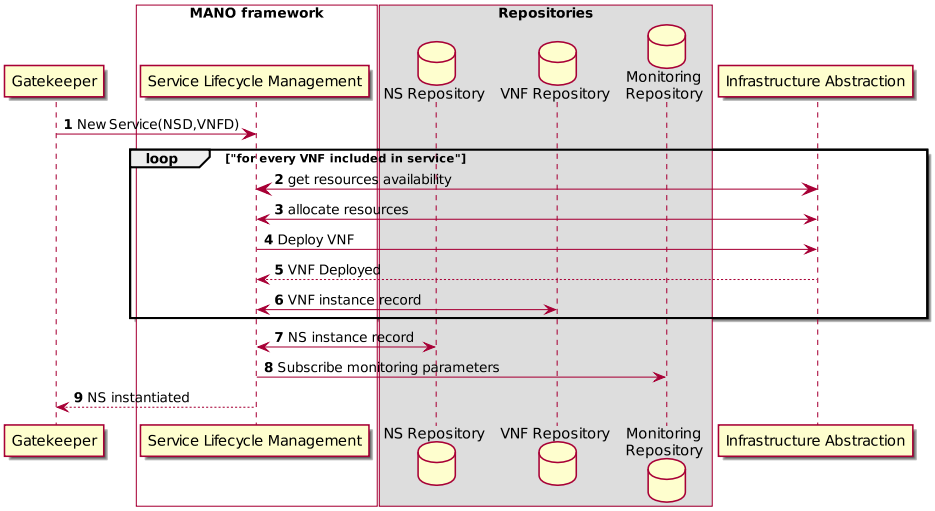

Figure 5.4: Network Service provisioning from the repositories point of view

5.2.1.2 Updating a network service

Figure 5.5 illustrates how the Service Platform Repositories are involved in the service updating process.

The process starts when the gatekeeper receives a service update request from the SDK (1). The gatekeeper retrieves the information about the Network Service from the NS Repository (2) and retrieves the new NSD/VNFD from the Catalogue (3). It then triggers the Service Update to the Service Lifecycle Management (4). The SLM starts the update process and deploys the new VNFs saving one VNF Instance record per each (5, 6, 7, 8). When the SLM finish the deployment process it updates the NSR in the NS Instance Repository (9) and notifies the Gatekeeper about the successful NS update (10). The last step is the notification to the SDK from Gatekeeper that the update is done (11).

5.2.2 NS Instance Repository

In the SONATA Service Platform the Network Service Instances Repository stores data from the Network Services Instances which are running in the Platform. The Network Service Record will

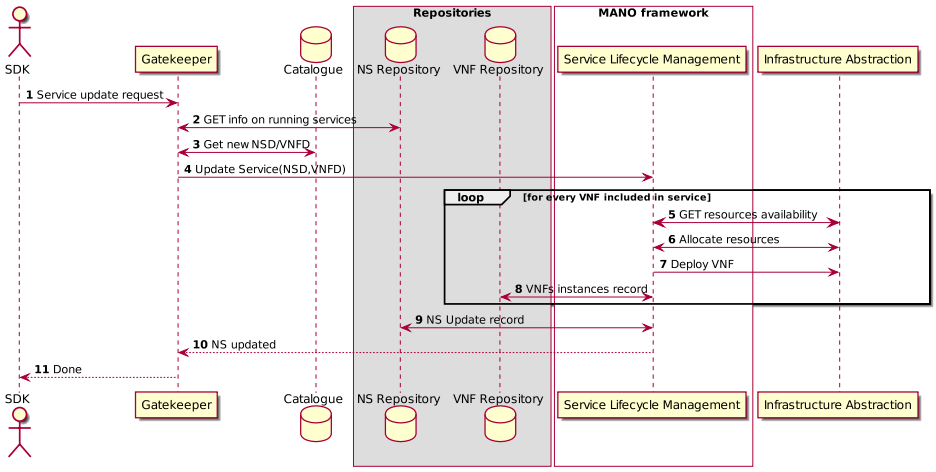

Figure 5.5: Update a Network Service.

be stored with a unique universal identifier UUID, together with all the fields resulting from the instantiation process.

5.2.2.1 Features

The features of this Network Service Repository are the following:

Create an NS Record: The NS Instance Repository is responsible for storing Network Service Records, after a request from the SLM, to be retrive later.

Retrive an NS Record: The stored NSR is retrieved from the NS Instance Repository.

Update an NS Record: The Service Lifecycle Management can update the Network Service Record stored in the NS Instance Repository.

Delete an NS Record: The Service Lifecycle Management can delete the Network Service Record stored in the NS Instance Repository.

5.2.2.2 API

Table 5.3 shows the RESTful API of the NS Instance Repository.

Table 5.3: NS Instance 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)
		If any NS instance exists, it returns an error	Not Found (404)
/records/nsr/ns-instances/:uuid	GET	List specific NSR instance information in JSON format	Ok (200)
		If the NS instance doesn't exist it returns an error	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)
		If the JSON have schema mismatch then it returns an error.	Unprocessable Entity (422)
		If the UUID exists then it returns an error.	Conflict (409)
		If the content type isn't a JSON it returns an error	Unsupported Media Type (415)
		If the payload is a bad formated JSON it return an error	Bad Request (400)
/records/nsr/nsr-instances/:uuid	PUT	Update a NSR instance. The content type have to be a JSON. It returns the updated NSR.	Ok (200)
		If the UUID doesn't exist it returns an error.	Not Found (404)
		If the content type isn't a JSON it returns an error	Unsupported Media Type (415)
		If the new UUID exists then it returns an error.	Conflict (409)
		If the JSON have schema mismatch then it returns an error.	Unprocessable Entity (422)
/records/nsr/ns-instances/:uuid	DELETE	Delete a NSR instance. It returns the NSR	Ok (200)
		If the UUID doesn't exists then it returns an error.	Not Found (404)

5.2.2.3 Internal Architecture

The Internal Architecture of NS Instance Repository is described in Figure 5.6. It consists of a service access to the MongoDB database.

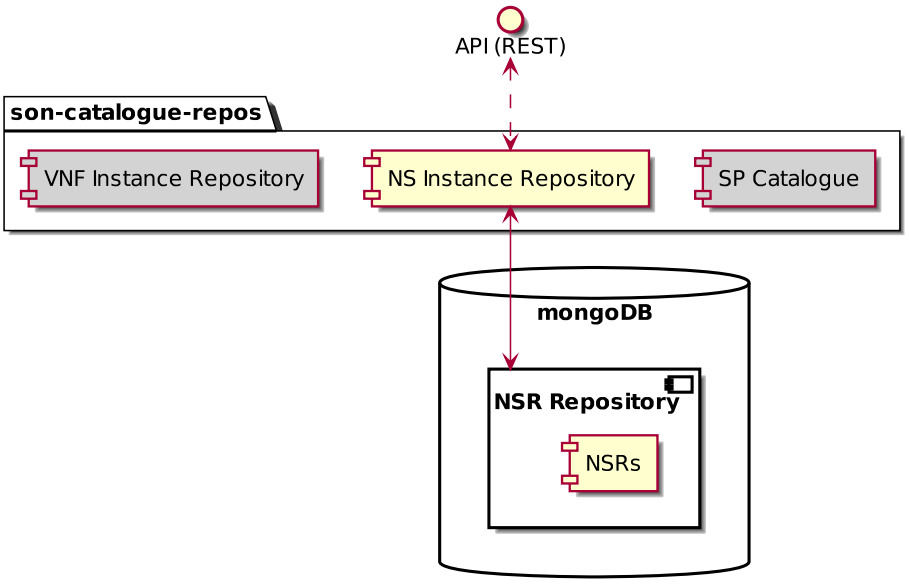

Figure 5.6: NS instance repository components and interfaces.

5.2.2.4 Message Sequence Charts

The message sequence chart in Figure 5.4 shows how the NS Instance Repository acts in the instantiation of a network service.

5.2.2.5 Technologies used

The implementation of the NS Instance Repository used the same technologies as the **Catalogues** (see Section 5.1, Table 5.2).

5.2.2.6 Tests

This sub-section describes tests that have been designed and implemented to test the NS Instance Repository API.

Unit tests

We are performing the unit test using an external mongoDB to make it simple.

1. List API methods by accessing the root:
   - **Execution:** `curl localhost:4002/records/nsr`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of methods available
2. Submit a new NSR with correct parameters:
   - **Set-up:** have a valid NSR in a JSON format (`nsr-example.json`, in folder `./spec/fixtures`)
   - **Execution:** `curl -X POST -d @nsr-example.json \`

   `localhost:4002/records/nsr/ns-instances --header "Content-Type:application/json"`

   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the created NSR;
3. Submit duplicated NSR:
   - **Set-up:** have a valid NSR in a JSON format (`nsr-example.json`, in folder `./spec/fixtures`)
   - **Execution:** `curl -X POST -d @nsr-example.json \`

   `localhost:4002/records/nsr/ns-instances --header "Content-Type:application/json"`

   - **Expected:** HTTP code 409 (Conflict) is returned;
4. Submit a new NSR with incorrect parameters:
   - **Set-up:** have a not valid NSR in a JSON format (`nsr-example-with-errors.json`, in the folder `./spec/fixtures`)
   - **Execution:** `curl -X POST -d @nsr-example-with-errors.json`

   `localhost:4002/records/nsr/ns-instances --header "Content-Type:application/json"`

   - **Expected:**

   HTTP code 422 (Unprocessable Entity) is returned;
5. Retrive an NSR without UUID given:
   - **Execution:** `curl localhost:4002/records/nsr/ns-instances`
   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the NSRs found, limited by 10 per page

6. Retrive an NSR with UUID given:

   - **Execution:** `curl localhost:4002/records/nsr/ns-instances/`

   `32adeb1e-d981-16ec-dc44-e288e80067a1`

   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the NSR with UUID 32adeb1e-d981-16ec-dc44-e288e80067a1

7. Delete an NSR with UUID given:

   - **Execution:** `curl -X DELETE`

   `localhost:4002/records/nsr/ns-instances/32adeb1e-d981-16ec-dc44-e288e80067a1`

   - **Expected:** HTTP code 200 (OK) is returned

Further unit test are planned, namely covering the `offset` and `limit` parameters for paginating extensive lists of results and the update of an NSR.

5.2.3 VNF Instance Repository

In the SONATA Service Platform, the VNF Instance Repository plays the role of a data container of VNF instances. These records are created to index the virtualized resources allocated to each VNF instance and include sufficient information to allow future changes to the deployed VNF through the virtualized network functions lifecycle management.

5.2.3.1 Features

The VNF Instance Repository implements the following functionalities:

Retrieving VNF Instances: The repository contains the list of VNF instances deployed in the service platform, information that can be retrieved, including all the data of the instance.

Creating New VNF Instances: One feature of the VNF repository is the ability to create new network function instances in the service platform.

Updating VNF Instances: Another important functionality is updating the existing instances information, especially relevant in the lifecycle management.

Deleting VNF Instances: Another use case is the deletion of VNF instances.

5.2.3.2 API

Table 5.4 shows the RESTful API of the VNF Repository.

Table 5.4: VNF Instance 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)
/records/vnfr/vnf-instances?output=YAML	GET	List all VNF instances in YAML format	Ok (200), Error Establishing a Database Connection (500)

Uri	Method	Purpose	Returned code(s)
/records/vnfr/vnf-instances/:id	GET	List specific VNF instance information in JSON format	Ok (200), Not Found (404)
/records/vnfr/vnf-instances/: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/:id	PUT	Update a VNF instance	Ok (200), Parsing error (400), Duplicated ID (400), Unsupported Media Type (415)
/records/vnfr/vnf-instances/:id	DELETE	Delete a VNF instance	Ok (200), Not Found (404)

5.2.3.3 Internal Architecture

The internal architecture of the VNF Instance Repository is shown in Figure 5.7.

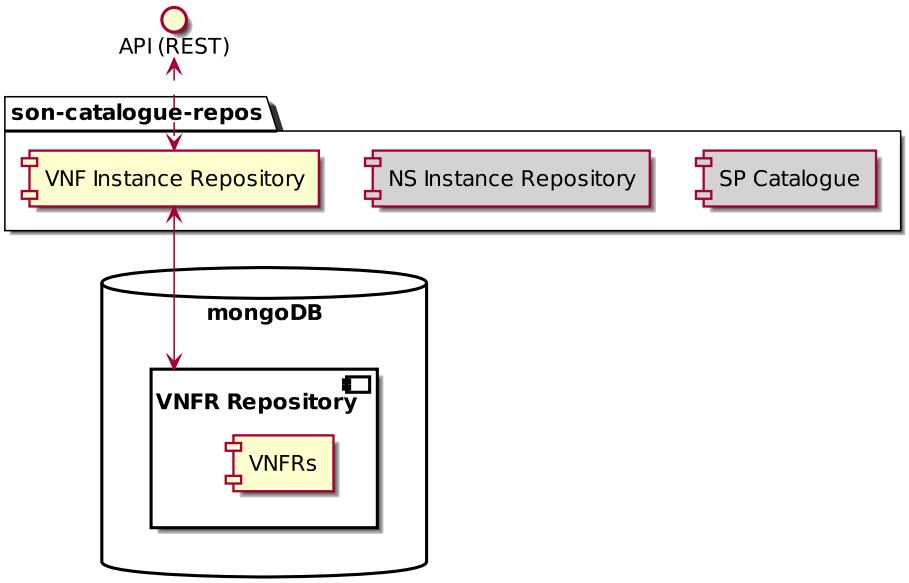

Figure 5.7: VNF Instance Repository components and interfaces.

5.2.3.4 Message Sequence Charts

The message sequence chart (see Section 5.2, Figure 5.4) shows how the VNF Instance Repository acts in the instantiation of a network service.

5.2.3.5 Technologies used

Technologies used in the implementation of the VNF Instance Repository were the same as the ones used in the implementation of the SP Catalogues (see Section 5.1, Table 5.2).

5.2.3.6 Tests

This sub-section describes tests that have been designed and implemented to test the VNF Instance Repository API.

Unit tests

We are performing the unit test using an external mongoDB to make it simple.

1. List API methods by accessing the root:
   - **Execution:** `curl localhost:4002/records/vnfr`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of methods available
2. Submit a new VNFR with correct parameters:
   - **Set-up:** have a valid VNFR in a JSON format (`vnfr-example.json`, in the folder `./spec/fixtures`)
   - **Execution:** `curl -X POST -d @vnfr-example.json localhost:4002/records/vnfr/vnf-instances --header "Content-Type:application/json"`
   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the created vnfr;
3. Submit duplicated VNFR:
   - **Set-up:** have a valid VNFR in a JSON format (`vnfr-example.json`, in the folder `./spec/fixtures`)
   - **Execution:** `curl -X POST -d @vnfr-example.json localhost:4002/records/vnfr/vnf-instances --header "Content-Type:application/json"`
   - **Expected:** HTTP code 409 (Conflict) is returned;
4. Submit a new VNFR with not all the mandatory parameters:
   - **Set-up:** have an invalid VNFR in a JSON format (`vnfr-example-with-errors.json`, in the folder `./spec/fixtures`)
   - **Execution:** `curl -X POST -d @vnfr-example-with-errors.json localhost:4002/records/vnfr/vnf-instances --header "Content-Type:application/json"`
   - **Expected:** HTTP code 422 (Unprocessable Entity) is returned;
5. Retrive a VNFR without UUID given:
   - **Execution:** `curl localhost:4002/records/vnfr/vnf-instances`
   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the VNFR limited by 10 per page
6. Retrive a VNFR with UUID given:
   - **Execution:** `curl localhost:4002/records/vnfr/vnf-instances/62b21f06-154e-0893-3e95-0123c541a54f`
   - **Expected:** HTTP code 200 (OK) is returned, together with the JSON representation of the VNFR with UUID 62b21f06-154e-0893-3e95-0123c541a54f

7. Delete a VNFR with UUID given:

   - **Execution:** `curl -X DELETE`

   `localhost:4002/records/vnfr/vnf-instances/62b21f06-154e-0893-3e95-0123c541a54f`

   - **Expected:** HTTP code 200 (OK) is returned

Further unit tests are planned, namely covering the `offset` and `limit` parameters for paginating extensive lists of results and the update of a VNFR.

5.2.4 Monitoring Repository

SONATA monitoring repository aims to collect and process monitoring data from the VNFs that compose NSs and provide related performance information. In this sense, developer has the ability to choose from predefined metrics (RAM usage, CPU usage, hard disk utilization, etc) or develop his own monitoring probe in order to capture service-specific behaviour. Moreover, the developer is able to define rules based on a composition of metrics from one or more VNFs in order to receive notifications in real time.

5.2.4.1 Features

The Monitoring Repository implements the following features.

Manage alerting rules: The Monitoring Repository allows the management of the alerting rules per service instantiated in the SONATA Service Platform.

Manage notification types: The Monitoring Repository is responsible for managing the type of notification selected by the user per service instantiation.

Manage function monitoring: The Monitoring Repository provides methods for managing the monitoring of each particular function comprising a network service.

Manage service monitoring: The Monitoring Repository provides methods for managing the monitoring of each network service instantiated by a developer.

5.2.4.2 API

This sub-section describes the RESTful API of Monitoring Repository, which offers the endpoints shown in Table 5.5.

Table 5.5: Monitoring Repository REST API

Endpoint	Method	Description	Returned code(s)
`/alerts/rules`	GET, POST	Retrieve/insert details about rules triggering alerts on instantiated network services.	OK (200), Created (201), Not found (404)
`/alerts/rule/:pk`	GET, PUT, PATCH, DELETE	Retrieve, insert, update, delete details about rules triggering alerts on a particular network service.	OK (200), Created (201), Not found (404)
`/functions`	GET, POST	Retrieve/insert details regarding the list of instantiated functions.	OK (200), Created (201), Not found (404)
`/function/:pk`	GET, PUT, PATCH, DELETE	Retrieve, insert, update, delete details on a particular function.	OK (200), Created (201), Not found (404)

Endpoint	Method	Description	Returned code(s)
`/metrics`	GET, POST	Retrieve/insert details in the list of monitoring metrics.	OK (200), Created (201), Not found (404)
`/metric/:pk`	GET, PUT, PATCH, DELETE	Retrieve, insert, update, delete details on a particular monitoring metric.	OK (200), Created (201), Not found (404)
`/services/user/:userID`	GET	Retrieve the list of services instantiated by a unique user ID.	OK (200), Created (201), Not found (404)
`/service/:pk`	GET, POST, PUT, PATCH, DELETE	Retrieve, insert, update, delete details on a particular service.	OK (200), Created (201), Not found (404)
`/notification/types`	GET, POST	Retrieve/insert types of user notifications (email, SMS, pub/sub queue).	OK (200), Created (201), Not found (404)
`/notification/type/:pk`	GET, PUT, PATCH, DELETE	Retrieve, insert, update, delete details on a particular notification type.	OK (200), Created (201), Not found (404)
`/prometheus/metrics/list`	GET	Retrieve the list of available metrics.	OK (200), Not found (404)
`/prometheus/metrics/data`	POST	Insert metrics data on Prometheus server.	OK (200), Created (201), Not found (404)

5.2.4.3 Internal Architecture

The SONATA Monitoring Repository collects and processes monitoring data from the VNFs and NSs. The developer has the ability to activate predefined metrics (RAM and CPU usage, hard disk usage, etc.) or develop and implement his own monitoring metric in order to capture service-specific behaviour. Moreover, the developer can define rules based on metrics gathered from one or more VNFs in order to receive notifications in real time. In general, the developer will be able to subscribe to a message queue or he can get the alert notifications by email and/or SMS on his smartphone. Most importantly, monitoring data and alerts are accessible through a RESTful API or directly accessing the Gatekeeper GUI. The internal architecture of Monitoring System, including Monitoring Repository, is depicted in Figure 5.8.

5.2.4.4 Message Sequence Charts

This section depicts Service Platform Monitoring System interactions in Message Sequence Charts.

During initialization/instantiation phase

During the instantiation process of a service (along with the instantiation of its virtualized network functions), the Function Lifecycle Management passes monitoring information to the Monitoring Manager, including the metrics and alerts that are initiated after the service start-up.

During normal operation

Notifications are sent to the developer through the SONATA Message Broker whenever an alert is activated (monitoring values beyond a given threshold).

Retrieve Monitoring data

After receiving an alert, the developer or/and Service Platform administrator can access real-time monitoring or historical data regarding a network service or a function through the Gatekeeper

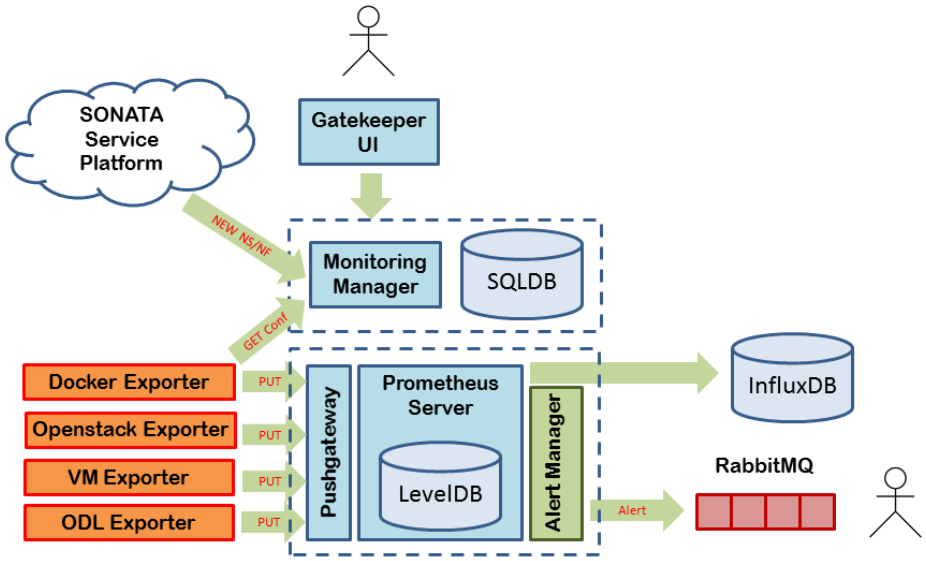

Figure 5.8: SONATA Monitoring Repository.

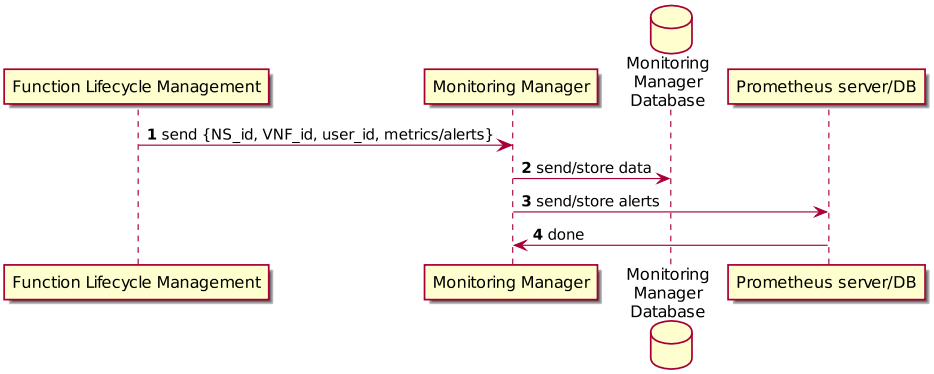

Figure 5.9: During initialization/instantiation phase.

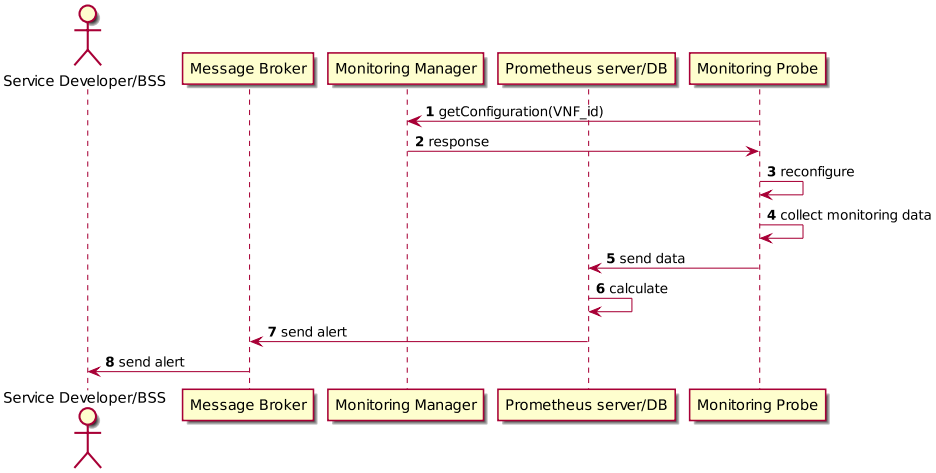

Figure 5.10: Notifications sent during normal operation.

GUI.

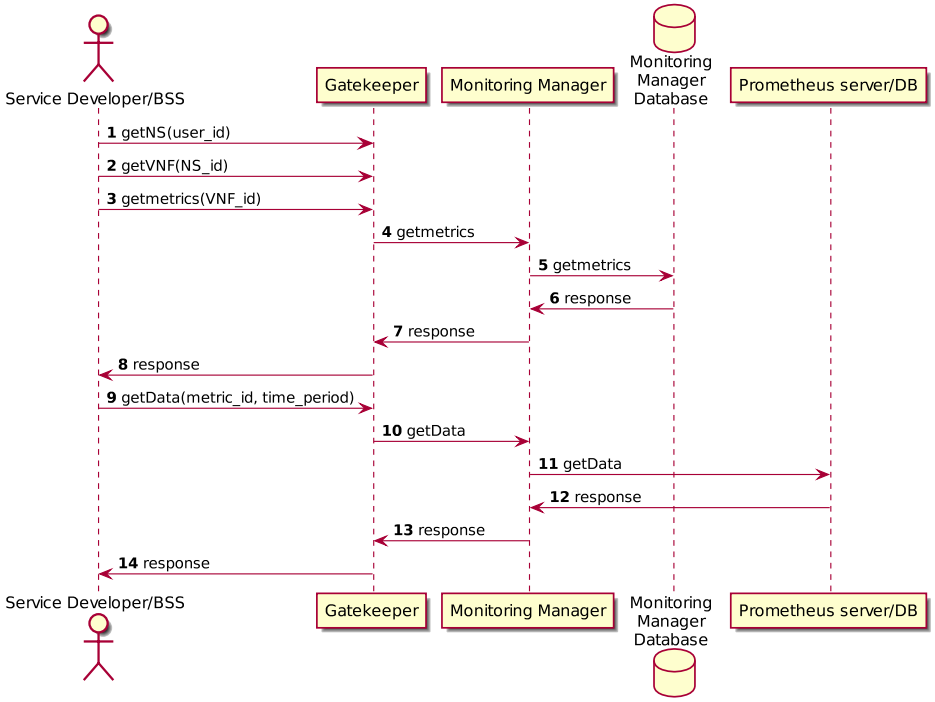

Figure 5.11: Retrieve Monitoring data

Update Monitoring configuration file

The SONATA monitoring solution we have developed provides the ability to the developer to modify the monitoring parameters through a configuration file. This way, the developer can add a new metric, change the monitoring frequency of an existing metric or even include a new alerting rule.

5.2.4.5 Technologies used

Technologies used in the implementation of the Monitoring Repository are listed in Table 5.6.

Table 5.6: Technologies used in the implementation of the Monitoring Repository.

Name	Type	Purpose
Prometheus	monitoring framework	Open source Monitoring Framework
InfluxDB	time-series database	Platform for storing and collecting time-series data
Django REST framework	toolkit	Toolkit for building Web APIs
MySQL	relational database	Relational database for storing additional relational data

5.2.4.6 Unit tests

The following unit tests check that all components of the monitoring service are running and communicating properly with each other.

1. List metrics available in Pushgateway :

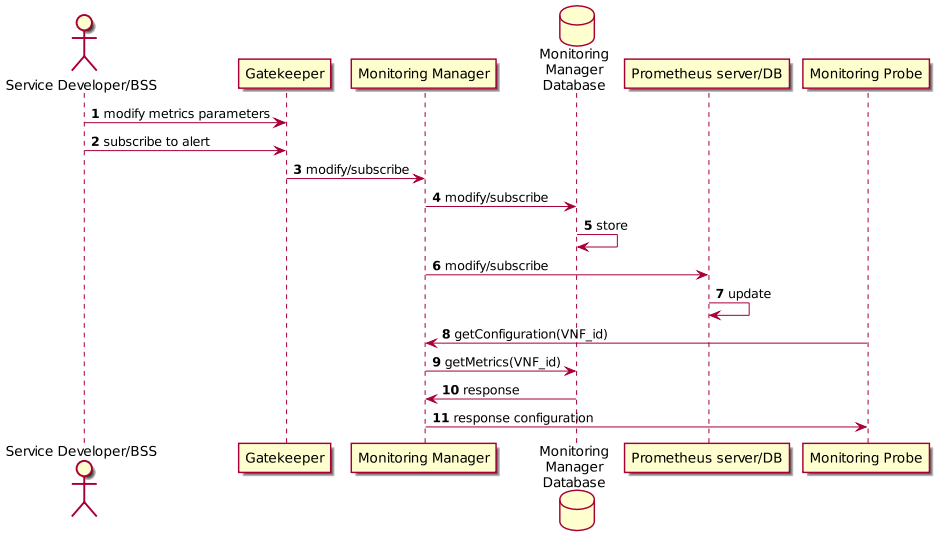

Figure 5.12: Update Monitoring configuration file

- **Execution:** `curl localhost:9091/metrics`
- **Expected:** HTTP code 200 (OK) is returned, with the list of metrics available in Pushgateway

2. List Network Services:
   - **Execution:** `curl localhost:8000/api/v1/services`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of Network Services which are monitored
3. List VNFs:
   - **Execution:** `curl localhost:8000/api/v1/functions`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of VNFs which are monitored
4. List metrics available in Prometheus:
   - **Execution:** `curl localhost:/api/v1/prometheus/metrics/list`
   - **Expected:** HTTP code 200 (OK) is returned, with the list of metrics available in Prometheus server

5.3 Infrastructure Abstraction

In the SONATA Service Platform the Infrastructure Abstraction plays the role of an abstraction layer between the MANO framework and the underlying (virtualised) infrastructure.

The Infrastructure Abstraction allows the orchestrator's entities to interact with the infrastructure, regardless of the specific technology used to manage it. It exposes interfaces to manage service and VNF instances, retrieve monitoring information about the infrastructure status, reserve resources for services deployment.

It is composed of two main modules, the Virtual Infrastructure Manager Adaptor (VIM Adaptor) and the WAN infrastructure Manager Adaptor (WAN Adaptor). The VIM Adaptor is responsible

for exposing an interface to interact with one or more VIMs, managing computational, network or storage resource in one or more Points of Presence to the Sonata MANO framework . The WIM Adaptor allows the service platform to manage network resources connecting different NFVI-PoPs in a vendor agnostic fashion, in order to provide connectivity to the deployed services.

5.3.1 VIM Adaptor

The VIM Adaptor is one of the components of the Infrastructure Abstraction layer. It enables the interaction between the Service Platform components and one or more VIM. Using specific wrapper for different VIM implementation, the VIM Adaptor exports a common interface to the Service platform to manage computational and storage resource, deploy and manage services, provide necessary instance information for the monitoring and management facilities of the service platform.

5.3.1.1 Functions

The VIM Adaptor implements the following functions:

VIM Abstraction: The interface exposed to the MANO framework entities is technology independant. It implement methods and API to deploy services, retrieve the VIM(s) status and manage resources, indipendently from the technology used to manage the virtual infrastructure.

Resource Discovery: The VIM Adaptor exposes information from the VIM in an abstract form to the MANO framework, collecting and fetching information on resource availability and infrastructure status through the message bus API.

5.3.1.2 Features

The VIM Adaptor implements the following features:

Register/Deregister VIM: The *VIM Adaptor* allows the *Service Platform Operator* to register or deregister a VIM to be used by the MANO framework to deploy services.

List registered VIMs: The *VIM Adaptor* allows the *Service Platform Operator* to list the available VIMs, along with information about their status and their resource availability.

Retrieve Compute Resource Availability: The *VIM Infrastructure Adaptor* allows the *Service Platform Orchestrator* to know if the underlying infrastructure is able to host a service, given its resource requirements.

Deploy New Service Instance: The *VIM Adaptor* allows the *Service Platform Orchestrator* to deploy a new service instance, given the relevant Network Service Descriptor and the list of the relevan VNF Descriptors.

Delete Existing Service Instance: The *VIM Adaptor* allows the *Service Platform Orchestrator* to remove and existing service instance, given the relevant Network Service Record.

5.3.1.3 API

Table 5.7 shows the API currently offered by the VIM Adaptor through the Message bus.

Table 5.7: VIM Adaptor pub/sub API.

Name	Description	Topic
Register VIM	Registers a new VIM to the platform. Requires the VIM vendor, endpoint and credentials. Returns success or error code, and an UUID to identify the new VIM.	infrastructure.management.compute.add
De-Register VIM	Remove a VIM from the platform. Requires the VIM type and UUID. Returns success or error code, and an UUID to identify the new VIM.	infrastructure.management.compute.remove
List VIMs	Returns the available VIMs along with relevant information.	infrastructure.management.compute.list
Check Resources	Check if a VIM can handle the deployment of a service, given the VIM UUID and the service resource requirements. Returns success or error code.	infrastructure.management.compute.resource
Deploy Service	Trigger the deployment of a new service, given a SONATA NSD and VNFDs in YAML. Returns instance information for the deployed service.	infrastructure.service.deploy

5.3.1.4 Internal Architecture

This sub-section describes the Internal Architecture of the VIM Adaptor. The module is conceptually divided in two parts. The upper part of the VIM Adaptor realizes the interface toward the service platform, exposing the API described above through the message bus. The use of queues and of the mux/demux system enables the asynchronous handling of multiple API call. The upper part of the system is shown in Figure 5.13.

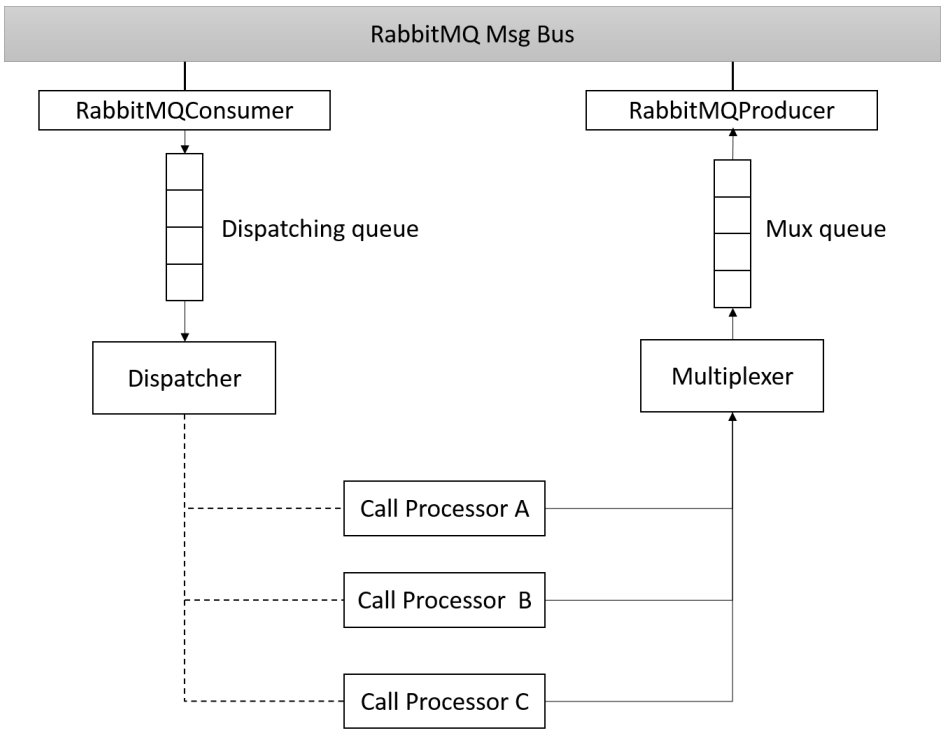

Figure 5.13: Software Architecture of the upper part of the VIM Infrastructure Adaptor

Call processors, which implement procedure for the different API calls, act as a conceptual border

between the two parts of the VIM adaptor. The lower part of the VIM adaptor is responsible for the actual interaction with the VIMs. Its architecture is sketched in Figure 5.14. Call processors can access the available VIMs using the Wrapper bay. It allows the retrieval and use of VIM Wrappers. The VIM wrappers expose an interface toward call processors, enabling service deployment and instance management, hiding the vendor specific procedures and clients.

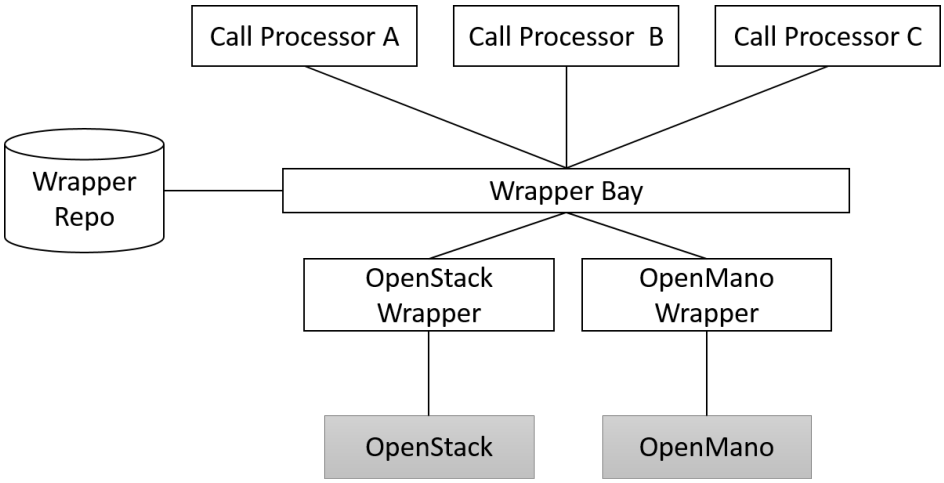

Figure 5.14: Software Architecture of the lower part of the VIM Infrastructure Adaptor

Wrapper information for each registered VIM is stored inside a local respository vith a simple data-model:

- uuid: String(unique)
- vimEndpoint: URL
- vimType: String
- wrapperType: String
- AuthUser: String
- AuthPass: String
- AuthKey: String

The relevant API calls allow the operator to register or remove VIMs from this repository. Each wrapper implements internally the needed procedure to serve the API calls using the VIM it wraps, such as translate the SONATA service and VNF descriptors in the VIM-specific description language, interact with the relevand endpoint for the deployment, manage the instances.

Although the design is completely vendor agnostic, the first prototype of the VIM Adaptor allows registering and interacting with OpenStack [19], assuming the availability of an Heat endpoint to deploy service in a orchestrated fashion [4]. Future works will extend the set of available VIMs, introducing more wrappers.

OpenStack Heat Wrapper

This section describes the details of the first year prototype, which is based on the OpenStack VIM [19] and its cloud orchestrator service Heat [4]. Heat uses its own description language, HOT, to

define application stacks. A stack is composed by a set of resources like virtual machines, networks, routers, ports, etc., that have to be created and interconnected, according to the NS and VNFs descriptors.

OpenStack Heat Client

In order to interact with the Heat endopint, the Adaptor wraps the official OpenStack python client library in a safe usage manner. This library offers the API to authenticate to the Keyston identity manager and to perform several stack operations. The wrapped client library currently supports the following operations:

- Create a stack from an HOT template provided as a YAML string
- Retrieve the status of a stack
- Delete a stack

HOT template and translation models

Since the HOT description language is needed to deploy a stack with Heat, the OpenStack Wrapper implements a translation from the Sonata Service Descriptor and VNF Descriptor, detailed in Appendix A, to the OpenStack HOT template. 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. Although further investigation is needed to assess the generality of this model, we stress that it could be used to translate the SONATA descriptors to other stack description languages with minimal refinement. According to the SONATA descriptors, the basic elements of a Network Service, depicted in Figure 5.15, can be resumed as follows:

- *Network Service*;
- *VNF*: Virtual Network Function. A constituent of a Network Service;
- *VNFC*: Virtual Network Function Component. A constituent of a VNF;
- *Network Service connection point*: an interface of the network service toward the outer world. Green circles in Figure 5.15;
- *VNF Connection point*: an interface of the network function toward the network service scope. Blue circles in Figure 5.15;
- *VNFC Connection point*: and interface of the VNFC toward the VNF scope. Yellow circles in Figure 5.15;
- *Network Service virtual link*: A link connecting two VNF connection points or a VNF connection point and a NS connection point. Blue lines in Figure 5.15;
- *VNF virtual link*: A link connecting two VNFC connection points or a VNFC connection point and a VNF connection point. Red lines in Figure 5.15.

As a simplification we assume the existence of a one-to-one mapping between VNFC and Virtual Deployment Unit (VDU) for the first VIM Adaptor prototype. This mapping represents the virtual resources needed to deploy and run a specific VNFC. With this simplification, the VIM Adaptor module itself operates a further one-to-one mapping from the components described above to a set of constituents which are more deployment-oriented. This mapping is in Table 5.8.

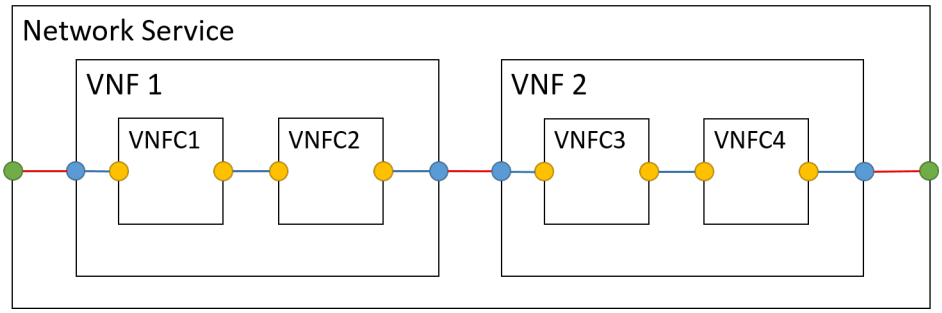

Figure 5.15: A conceptual representation of a Network Service according to the SONATA service descriptor.

Table 5.8: Model mapping.

SONATA NSD element	Abstraction model element
VNFC	VDU -> Virtual Machine
VNFC Connection point	Virtual Machine Port
VNF Virtual Link	Layer 3 network
VNF Connection Point	Router Port
NS Virtual Link	Router
NS Connection Point	Router Port

Following this mapping, the example shown in Figure 5.15 is translated into the model illustrated in Figure 5.16.

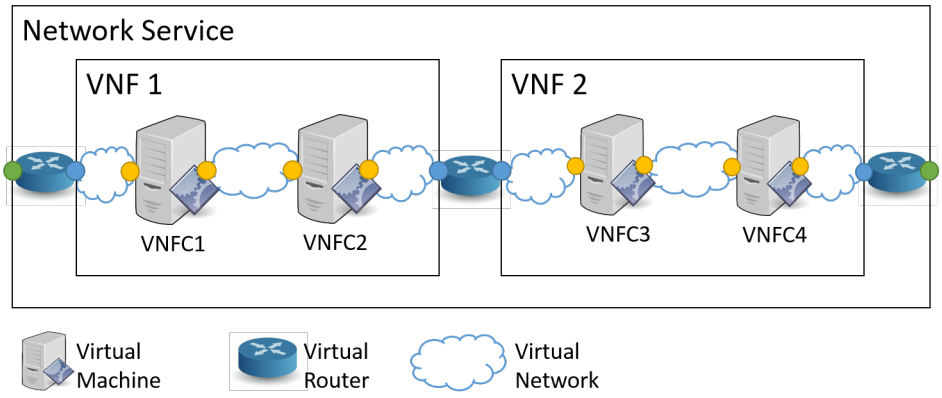

Figure 5.16: A conceptual representation of a Network Service after the translation operated by the VIM Adaptor.

This model can be easily serialized following the HOT template syntax, in order to be deployed in OpenStack using Heat. Neutron Router, Network and Subnetwork [5] can be used to represent networks and virtual routers in the Deployment Model, and Nova servers are the trivial translation for virtual machines.

Regarding resource translation, the VNFD template allows a precise definition of the resources requirement of each VDU (or VM). Nevertheless, Openstack does not allow choosing each resource parameter in this way. Openstack requires using the so called *flavor*. A flavor is a static combination of 3 hardware parameters defining CPU, RAM, and disk storage. In order to be selected during a deployment, it must be defined and created in advance in the VIM scope. Given this constraint, two strategies can be envisioned:

- On the fly creation of a new *flavour* for each VM deployment, with the exact values defined in the vnfd resource_requirements section. This option requires the tenant to be able to create a flavor, that is not a common behaviour, since the flavor creation is usually restricted to infrastructure administrators. Moreover, this will imply an explosion of flavors, that totally breaks the rationale behind the flavor concept in Openstack.

```
resource requirements:
    cpu:
      vcpus: 1
    memory:
      size: 2
      size_unit: "GB"
    storage:
      size: 10
      size_unit: "GB"
```

Figure 5.17 shows an example of a list of flavor that can be retrieved with the Nova API using the python Nova client.

```
$ nova flavor-list
+-----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
| ID  | Name      | Memory_MB | Disk | Ephemeral | Swap | VCPUs | RXTX_Factor | Is_Public |
+-----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
| 1   | m1.tiny   | 512       | 1    | 0         |      | 1     | 1.0         | True      |
| 2   | m1.small  | 2048      | 20   | 0         |      | 1     | 1.0         | True      |
| 3   | m1.medium | 4096      | 40   | 0         |      | 2     | 1.0         | True      |
| 4   | m1.large  | 8192      | 80   | 0         |      | 4     | 1.0         | True      |
| 5   | m1.xlarge | 16384     | 160  | 0         |      | 8     | 1.0         | True      |
+-----+-----------+-----------+------+-----------+------+-------+-------------+-----------+
```

Figure 5.17: Response for the API call flavour-list.

With this flavor configuration, the best corresponding flavor is the *small* flavor. Therefore, the generated HOT section will be the following:

```
properties:
    flavor: "m1.small"
```

5.3.1.5 Message Sequence Charts

The MSC in Figure 5.18 shows the interactions between the Adaptor sub-components which are triggered when the service platform requests for a new VIM to be added. The call is asynchronous. A basic check is made for the input parameters by the specific Call Processor in step 1 of Figure 5.18. Following the specification in the call payload, a new Wrapper is created and configured (Steps 3 and 4). When a new wrapper is created, it executes a basic checks of the credential provided, trying to authenticate to the given endpoint. Upon success, the wrapper updates its internal statuses with the relevant information on the VIM, and is added to the Wrapper Registry (Steps 5,6 and 7). Once the VIM is successfully registered, the call processor is notified, and a response with the VIM UUID is sent to the SP.

The MSC in Figure 5.19 shows the interaction between the Adaptor sub-components needed to deploy a new service instance. The service can retrieve a list of the available VIMs from the VIM adaptor, in order to perform service placement (Steps 1, 2 and 3). Then, the platform provides the needed descriptor in YAML format along with the chosen VIM uuid. The descriptors are deserialised by the relevant call processor (Steps 4 and 5). The Call Processor asks the Wrapper Bay for the pointer to a VIM wrapper able to handle the deployment. When the Wrapper has been retrieved, the call processor sends it the deserialised descriptors, triggers the deployment and waits for completion (Steps 6, 7, 8 e 9). The VIM Wrapper uses the translation model described above to generate an intermediate deploy model from the descriptors, and subsequently serialize this model with the VIM specific template format (Steps 10 and 11). When the deployment

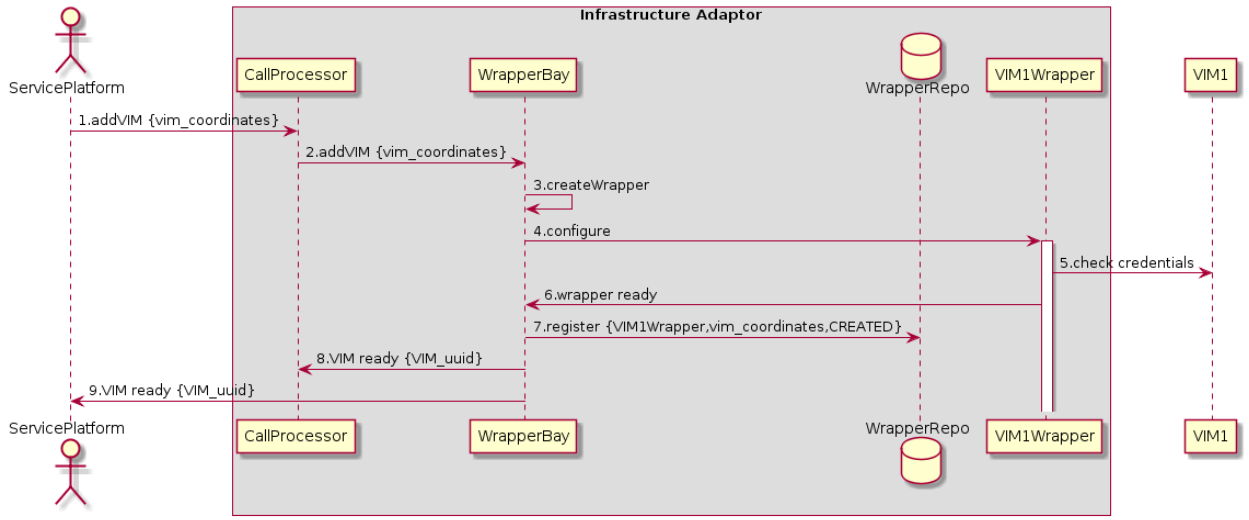

Figure 5.18: Register VIM internal interactions.

template is serialized, the wrapper uses the VIM specific client to trigger the deployment of the service. Depending on the VIM servers specification the client can retrieve some initial information from the response, such as IDs of the deployment and of virtual machines. Using this info, the wrappers enters a loop where it continously retrieves the deployment status, until success or errors are identified (Steps 12, 13 and 14). Finally, the deployment status is reported back to service platform with the relevant information (Step 15 and 16).

5.3.1.6 Technologies used

The VIM Adaptor is developed in Java and Python. External libraries and tools are listed in Table 5.9.

Table 5.9: Libraries

Name	Purpose
Maven	Build manager
JUnit	Unit testing
Jackson Data Format	Serialization/Deserialization
RabbitMQ java client	Pub/Sub system
Openstack python client	Communicate with Openstack
Python library argparse	Argument parser
Python library json	Encode/decode JSON for Python

5.3.1.7 Unit tests

The unit tests developed for the VIM Adaptor are listed below. They rely on the existence of a Mock VIM wrapper to emulate the behaviour of the OpenStack Wrapper from the call processors perspective, and also of a mock message bus, which will produce messages directed to the VIM adaptor, and mock the service platform behaviour when receiving messages from the VIM Adaptor.

1. Boot Adaptor
   - **Set-up:** Mock message bus and MANO plugin manager.
   - **Execution:** Instantiante and start the VIM Adaptor.

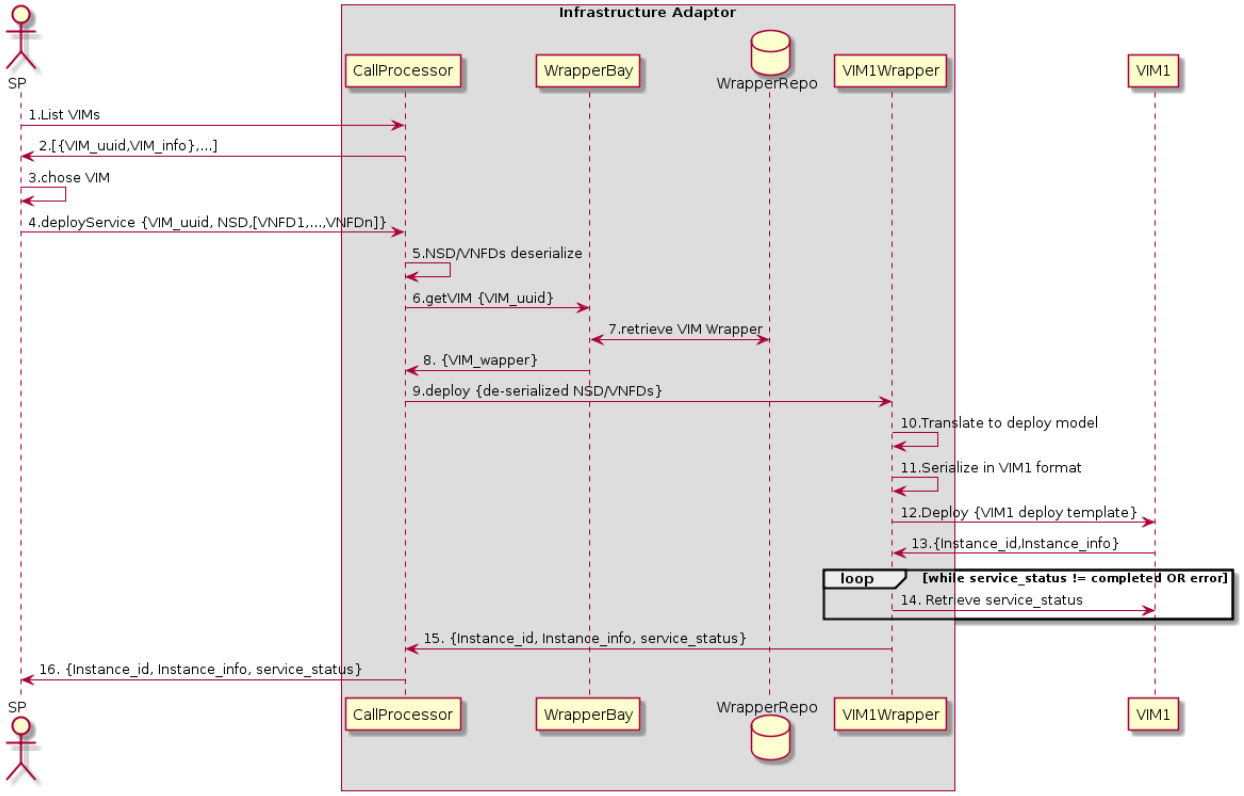

Figure 5.19: Deploy Service internal interactions

- **Expected:** VIM Adaptor registers to the mock MANO plugin manager, receives its UUID and sends 4 heartbeat.
- **Clean-up:** deregister and stop adaptor.

2. Register mock VIM
   - **Set-up:** running adaptor, have the IP, configuration and credentials of the VIM.
   - **Execution:** use the mock message bus to send the relevant message to the VIM adaptor.
   - **Expected:** JSON message send to the mock message bus, containing the VIM UUID.
   - **Clean-up:** remove the VIM from the VIM registry.
3. De-register mock VIM
   - **Set-up:** running adaptor, have the UUID of the mock VIM to de-register and the Mock VIM registered.
   - **Execution:** use the mock message bus to send the relevant message to the VIM adaptor.
   - **Expected:** JSON message with the VIM description.
4. List mock VIMs
   - **Set-up:** running adaptor.
   - **Execution:** use the mock message bus and the relevant API call to register three VIMs, saving their UUIDs in a list. Use the List Vim API call.

- **Expected:** a YAML message with the list of registered VIMs, which must match the list of the saved UUIDs.

5. NSD and VNFD de-serialization
   - **Set-up:** have files with the example SONATA NSD and VNFDs.
   - **Execution:** use the VIM Adaptor package do generate a deserialized object representing the network service and its network functions.
   - **Expected:** The Network Service and network function objects with relevant non-null object members.
6. Check resource mock VIM
   - **Set-up:** running adaptor, have the UUID of the mock VIM and a set of resource constrains.
   - **Execution:** use the mock message bus to send the relevant message to the VIM adaptor.
   - **Expected:** a response is sent to the mock message bus with the code "OK" in a JSON payload.
7. Deploy service mock VIM
   - **Set-up:** running adaptor, have the UUID of the mock VIM and the example SONATA NSD and VNFDs.
   - **Execution:** Use the mock message bus to send the relevant message to the VIM adaptor.
   - **Expected:** a stripped version of a NSR and VNFRs based on the content of the NSD and VNFDs.
8. HOT template generation
   - **Set-up:** running adaptor, have the UUID of the OpenStack VIM and the example SONATA NSD and VNFDs.
   - **Execution:** Use an utility method of the Adaptor to de-serialize the NSD and VNFDs, and get the relevan HOT template.
   - **Expected:** a correctly serialized HOT template (no check on the template validity).
9. Heat client service deploy and deletion
   - **Set-up:** running adaptor, have the UUID of the OpenStack VIM, a simple example HOT template, a running OpenStack environment with Heat.
   - **Execution:** Create an instance of the Heat client and use it to instantiate an Heat stack with the example HOT template and remove it.
   - **Expected:** a clean stack deployment, retrieve the stack info and a clean stack deletion.
10. Heat client deletion failure, non valid stack UUID
    - **Set-up:** a running OpenStack environment with Heat.
    - **Execution:** use the client to remove a non existing stack.
    - **Expected:** a null value is returned.
11. Heat client status retrival failure, non valid stack UUID

- **Set-up:** a running OpenStack environment with Heat.
- **Execution:** use the client to retreve a non existing stack.
- **Expected:** a null value is returned.

5.3.2 WIM Adaptor

WAN Infrastracture Manager (WIM) provides connectivity services between the NFVI-POPs and connectivity to Physical Network Functions. This functional block may be added as a specialized VIM, in particular for new greenfield virtualised deployments. Alternatively, a WIM can also be an existing component of the **OSS/BSS functional block**. The main functionalities offered by WIM are:

- Compute a path based on quality assurance factors such as jitter, RTT, delay and bandwidth calendaring.
- Establish connectivity over the physical network (e.g. set of MPLS tunnels).
- Provide a northbound interface to the higher layers, e.g. NFVO, to provide connectivity services between NFVI-PoPs or to physical network functions.
- Invoke the underlying NFVI network southbound interfaces, whether they are Network Controllers or Network Functions, to construct the service within the domain.

The role of WIM is important both in the multi-PoP scenarios, i.e multiple NFVI-PoPs interconnected via WAN links (can be over multiple WANs), and in single NFVI-PoP environments (central datacenter for NFV based services), i.e. traffic managment from the edge towards the NFVI-PoP. End-to-end connectivity between VMs hosted on different NFVI-PoPs requires the provision of network resources in three domains:

- internal networks of each NFVI-PoP (enforced by the local VIM instance)
- gateway routers at the edge of each NFVI-PoP
- WAN network between them, multiple WANs may exist. Attention needs to be drawn at the different trust or organisational boundaries that are expected to exist between the NFVI-PoP and the WAN domains and may affect the level of information disclosure for the network substrate.

An example network topology to understand the positioning of the WIM in the general view of ETSI NFV as described in [22] is shown in Figure 5.20.

For this example, multiple connectivity services are requested by the Resource Orchestration component of the NFVO:

- Virtual network between gateway in NFVI-PoP 1 and VNF1.
- Virtual network between gateway in NFVI-PoP 2 and VNF2.
- Virtual network between gateway in NFVI-PoP 1 and gateway in NFVI-PoP 2.
- Physical connectivity service between the physical Endpoint 1 and a gateway in NFVI-PoP 1.

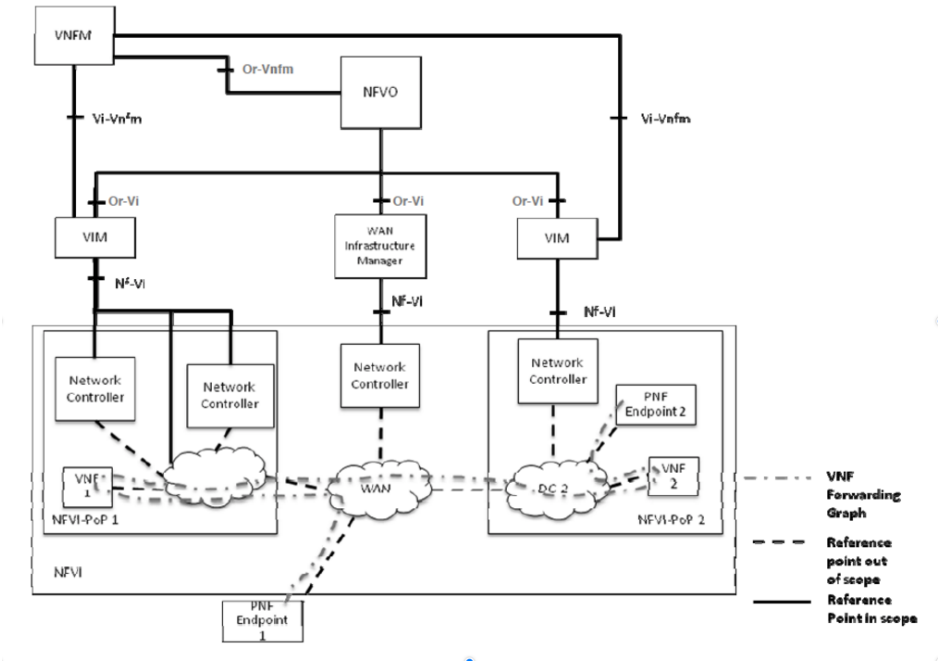

Figure 5.20: Example Network topology including the WIM

- Physical connectivity service between the physical Endpoint 2 and a gateway in NFVI-PoP 2.

This requires each NFVI-PoP VIM to establish a virtual network within its NFVI-PoP, the WIM to establish a virtual network between the NFVI-PoP 1 gateway and the NFVI-PoP 2 gateway, the WIM to establish connectivity between

- the PNF endpoint1 and NFVI-PoP 1 gateway, and the NFVI-PoP 2 VIM to establish connectivity between PNF
- endpoint 2 and the NFVI-PoP 2 gateway.

In the context of the above section, WIM Adaptor role in SONATA follows the functionalities offered by the VIM adaptor, solely focusing on the abstraction of network towards the MANO framework. As such the main functions, implemented by the WIM adaptor are:

WIM Abstraction: Exposes an interface to the MANO framework that is technology agnostic to the underlaying southbound interfaces or functionalities. The API exposes methods for network service deployment, WIM status and resource management. Depending on the underlaying supported functionalities, additionally network isolation and spliting features may be supported.

Resource descovery and monitoring: The WIM Adaptor exposes collected information on resource availability and infrastructure status through the message bus API.

5.3.2.1 Features

WIM features are detailed next.

Register/Deregister WIM The WIM adaptors allows to register WIMs into the MANO (SONATA SP) framework in order to be able to provision services over the network resources they are managing.

Retrieve WIM capabilities This feature allows to retrieve a list of the WIM capabilities

Retrieve Network topology This feature allows the SONATA SP to retrieve connectivity and topology information of the substrate network.

Retrieve Traffic Matrix This feature allows the signalling of the configured traffic matrix (network resources) for the substrate network.

Provision tenant network This feature allows the provision of a new tenant network slice over the substrate networking infrastructure.

Modify tenant network This feature allows the modification of an existing tenant network slice.

Delete tenant network This features allows the deletion of an existing tenant network slice.

Monitor tenant network This feature allows the monitoring of the resource usage of an existing tenant network slice.

5.3.2.2 API

This sub-section describes the (external) API of WIM infrastructure Adaptor.

Table 5.10: WIM Infrastructure Adaptor REST management API

Uri	Method	Purpose	Responce Content	Returned Code(s)
/infrastructure/wim/add	POST	Add a WIM to infrastructure	Return UUID of new WIM	Created (201), Failed(400), Already Exists (409)
/infrastructure/wim/remove/wim_id	DELETE	Remove WIM from in infrastructure		OK (200), Failed(400), Not found (404)
/infrastructure/wim/wim_id	GET	Remove WIM from in infrastructure	Retrieves json list with WIM capabilities	OK (200), Failed(400), Not found (404)
/infrastructure/wim/wim_id/net_topo	GET	Retrieve connectivity and topology	Returns json list of topology	OK (200), Failed(400), Not found (404)
/infrastructure/wim/wim_id/traffic_matrix	GET	Retrieve traffic matrix	Returns json list of traffic matrix	OK (200), Failed(400), Not found (404)
/infrastructure/wim/wim_id/tenant_net	POST	Create new tenant network slice over network	Return UUID of new tenant	Created (201), Failed(400), Not found (404)
/infrastructure/wim/wim_id/tenant_net/tenant_id	PUT	Modify specific tenant network slice		OK (200), No Content (204), Failed(400), Not found (404)

Uri	Method	Purpose	Responce Content	Returned Code(s)
/infrastructure/wim/wim_id/tenant_net/tenant_id	DELETE	Delete specific tenant network slice		OK (200),Failed(400), Not found (404)
/infrastructure/wim/wim_id/monitor/metrics	GET	Return all metrics available	Return json with metrics of WIM	OK (200),Failed(400), Not found (404)
/infrastructure/wim/monitor/metrics/tenant_id	GET	Return all metrics available for specific tenant	Return json with metrics of tenant	OK (200),Failed(400), Not found (404)
/infrastructure/wim/monitor/metrics/tenant_id/ metric_id/latest_rows	GET	Monitor specific tenants metric	Return json with metric data	OK (200),Failed(400), Not found (404)

5.3.2.3 Internal Architecture

The WIM adaptor is required to support multiple technologies usually providing different levels of abstraction and a variety of supported features. Under all those circumstances the WIM should be able to adapt the NFVO request to the closed possible mapping for each technology. This section provides an overlook on the possible cases and attempes a priliminary specification of the component internal architecture. Currently there are two main deployment cases for the WIM Adaptor, subject of the actual WAN technology and management:

Case 1: Programmatic Network Control (i.e. SDN based) This case is considered as the primary target for the WIM adaptor, aligned with the view of ETSI NFV presented in the NFV-MAN document [22]. The substrate networking infrastructure is controlled by Network Controllers that support open and programmatic interfaces (i.e. SDN), thus the WIM interfaces with those Network Controllers in order to provide multi-tenancy, programmability and isolation summarised as Network-as-a-Service (NaaS).

Case 2: Non-programmatic Network Control (legacy Network Management) This case is considered where actual programmatic interfaces are not available and where casual Network Management Systems are used for the resource management and operation of the WAN infrastructures. In this case it is expected that the service requests will be signalled through the OSS/BSS system and not directly through the WIM. This case is therefore considered out of scope for the SONATA's WIM adaptor component.

There can be seen two distinct cases:

VNFs accross multiple NFVI-PoP locations with pre-provisioned end-to-end links. This is the case virtualisation is supported via network overlays. End-to-end connectivity chain is crossing transparently one or more network domains and trust/organizational boundaries, built upon a pre-allocated static WAN connectivity service (e.g. E-Line, E-LAN, IP/MPLS VPN). This scenario is the most simple, hence less dynamic. Even if the WAN overlay network is not pre-provisioned, assuming the existence of a dynamic **Network-as-a-Service** (NaaS) oriented NMS, the actions to be taken will be the provision of the trunk between the PoPs.

VNFs accross multiple NFVI-PoP locations with an SDN based NaaS capable WIM. In this case the SDN-based control in the WAN is leveraged in order to build a richer (in features) and more dynamic network environment that can support the NaaS model and provision WAN connectivity on demand. The scenario implementation is more complex and requires coordi-

nated control of NFVI and WAN resources. This case is the one that will be supported and implemented by SONATA MANO framework.

According to [23] this last case can also be divided into three more scenarios. As the discussion of these scenarios is outside the scope of this document, we summarise them below and we advise the reader to check [23] for details. In summary, three relevant scenarios can be seen:

- **NFVI-PoP and WAN in a common trust domain**: in this scenario WIM implements the application-control interface and uses directly the services exposed by the northbound interface of the WAN SDN controller. The WAN SDN controller, administered by the WAN operator, is in charge of managing the underlying physical resources.
- **NFVI-PoP and WAN in different trust domains - client access to virtual WAN resources**: this scenario extends the previous by defining a 2-level hierarchy of SDN controllers - C (Client, administratively part of the endpoints' NFVI-PoP trust domain) and P (Provider, administratively part of the WAN). To support multiple client networks, the WAN P controller provides a unique set of abstracted resources to each client
- **NFVI-PoP and WAN in different trust domains - client access to physical host resources**: in this scenario the WIM requests network resources through an SDN controller, which interacts directly with the network nodes. This means that the client SDN controller is being provided direct access to control a subset of the WAN network resources.

Other more complex scenarios with different levels of SDN hierachies and support for recursiveness in the the SDN architecture are considered out of SONATA scope. In this view, the WIM Infrastructure Adaptor needs to identify the type of WIM and the level of abstraction in order to efficiently support and map service requests with the appropriate wrapper. For example the ability of a WIM to interact directly with the network elements might provide more refined control on the provisioning and management of the requested network resources. The identification and classification of WIM capabilities is signaled/configured during the registration process.

The figure below presents a preliminary architecture of the internal components of the WIM adaptor.

Northbound-wise, the WIM adaptor interfaces with the RabbitMQ Message Bus. The scheme followed is similar to Figure 5.13. Southbound-wise, the WIM adaptor interfaces with the WIM(s). A specific wrapper is used depending WIM implementation and supported interfaces. Assuming SDN-based WAN, possible candidates for WIM may be ONOS, OpenContrail or OpenDayLight (ODL) using Virtual Tenant Network (VTN) [18] for the NaaS provision. At this stage a simplified WIM exposing ODL capabilities for the creation, deletion and modification of the tenant network slices is needed.

5.3.2.4 Message Sequence Charts

Figure 5.22 shows the interactions taking place between the WIM adaptor and other compoments of the SONATA SP.

5.3.2.5 Technologies used

This sub-section describes the technologies used in the implementation of WIM Infrastructure Adaptor

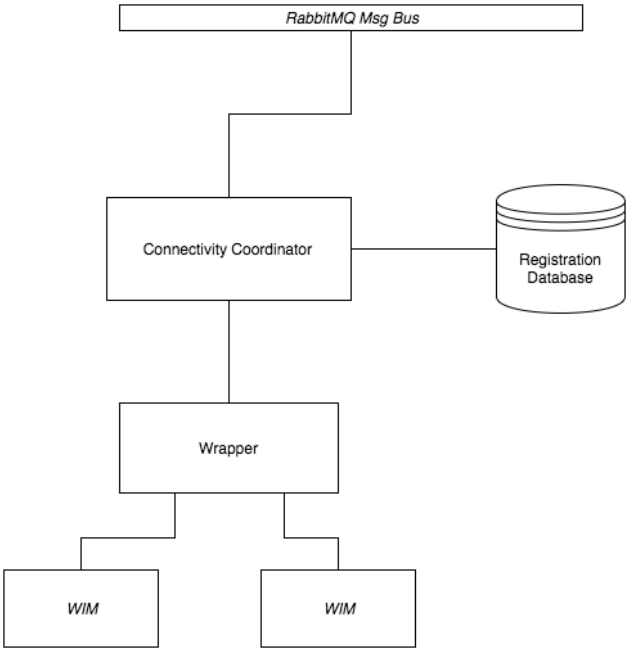

Figure 5.21: Software Architecture of the upper part of the WIM Infrastructure Adaptor.

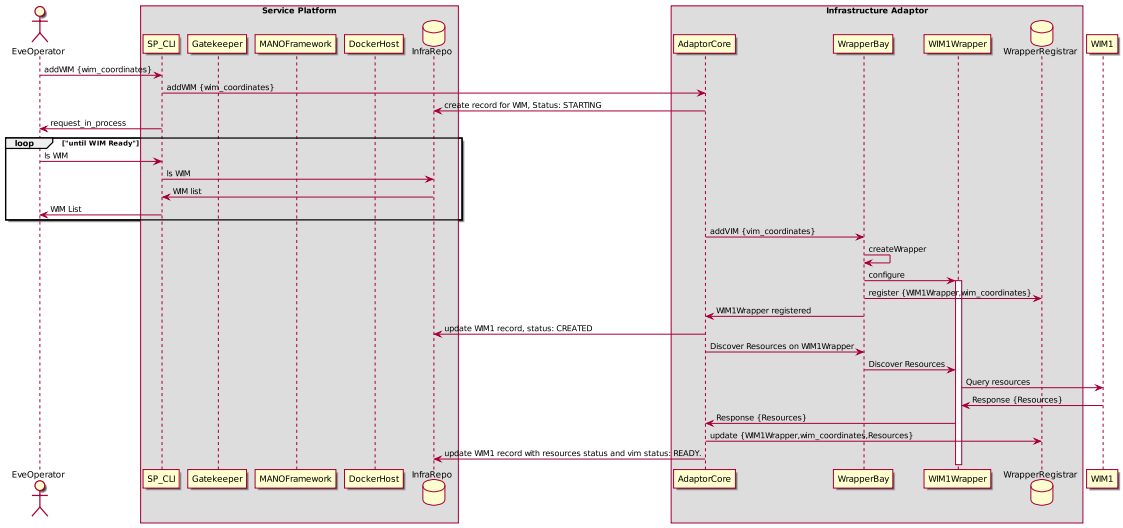

Figure 5.22: WIM Adaptor sequence diagram

Table 5.11: Technologies used by WIM Infrastructure adaptor

Name	Type	Purpose
Python	Programming language	Programming language
flask	framework	lightweight webframework for REST API
flask-restful	library	REST extension for flask
json	library	JSON implementation
mySQL	relational database	A robust and stable relational database

5.3.2.6 Tests

This sub-section describes the tests for WIM Infrastructure adaptor.

Unit tests

We are restricting unit tests of the Package Management to the access of the log file path (`/admin/logs`). Unit tests shown are only for the routes, since unit tests for the models are currently very basic (we're mocking the Catalogues micro-service). We execute these tests by mocking the models and accessing the internal URL (`http://localhost:5100`).

The unit tests done to the Package router are the following:

1. Register WIM
   - **Set-up:** have the IP of the WIM to be contacted.
   - **Execution:** `curl localhost:8080/infrastructure/wim/add`.
   - **Expected:** HTTP code 201 (Registered) is returned, together with the JSON response with the WIM UUID.
   - **Clean-up:** remove the WIM from the WIM registry.
2. De-register WIM
   - **Set-up:** have the IP and UUID of the WIM to be contacted.
   - **Execution:** `curl localhost:8080/infrastructure/wim/remove/wim_id`.
   - **Expected:** HTTP code 200 (Deleted) is returned, together.
3. Retrieve Network Topology
   - **Set-up:** have the IP and UUID of the WIM to be contacted.
   - **Execution:** `curl localhost:8080/infrastructure/wim/wim_id/net_topo`.
   - **Expected:** HTTP code 200 (Retrieved) is returned, together with the JSON response with the network topology.
4. Retrieve Traffic Matrix
   - **Set-up:** have the IP and UUID of the WIM to be contacted.
   - **Execution:** `curl localhost:8080/infrastructure/wim/wim_id/traffic_matrix`.
   - **Expected:** HTTP code 200 (Retrieved) is returned, together with the JSON response with the traffic matrix data.
5. Provision tenant network

- **Set-up:** NSD/VLD/VNFFGD snippet in JSON having connectivity information for the tenant network.
- **Execution:** `curl localhost:8080/infrastructure/wim/wim_id/tenant_net`.
- **Expected:** HTTP code 201 (Created) is returned, together with the JSON response with the network properties.

6. Deletion of a tenant network
   - **Set-up:** NSD/VLD/VNFFGD snippet with changes in JSON having connectivity information for the tenant network.
   - **Execution:** `curl localhost:8080/infrastructure/wim/wim_id/tenant_net/tenant_id`.
   - **Expected:** HTTP code 200 (Deleted) is returned.
7. Modification of tenant network
   - **Set-up:** NSD/VLD/VNFFGD snippet with changes in JSON having connectivity information for the tenant network.
   - **Execution:** `curl localhost:8080/infrastructure/wim/wim_id/tenant_net/tenant_id`.
   - **Expected:** HTTP code 200 (Modified) is returned.
8. Monitoring of tenant network
   - **Set-up:** UUID of the tenant network to be monitored as well as WIM id to be interrogated.
   - **Execution:** `curl localhost:8080 \`

     `/infrastructure/wim/monitor/metrics/tenant_id/metric_id/latest_rows`.
   - **Expected:** HTTP code 200 (monitor) is returned, together with the JSON response with the WIM properties.

Further unit tests will be written namely to test the component validation.

6 Phase 1 storyboard

This Annex documents the storyboard of the first year of the SONATA project. The goal of this script is to provide a reference framework for both the SONATA Service Platform as well as for the Software Development Kit on how the combined installation, development and deployment process is expected to work for the first phase of the project.The idea behind the storyboard is to highlight features of interest. Its main goal is to ensure consistency among SP and SDK, and to provide a guideline and roadmap for the implementation and demonstration process planned at this stage of the project.

6.1 Implanted Story Steps M10

1. Initial state:
   a) A number of servers are available.
   b) Some run the service plattform.
   c) Others will run the actual services and act as *two* VIMs. They will be running OpenStack.
2. Step: Initiate the service platform
   a) Operator Eve has heard great things about SONATA. She also has a couple of machines and two VIMs available and decides to give it a try.
   b) Introduce the VIMs to the SP, Operator Eve types:
      i. `son-platform add VIM IP-of-VIM-Access ssh-key`
3. Step: Get/create a service
   a) Eve talks to Developer Joe about SONATA. Joe loves it and decides to write a service. For that he creates a workspace in the SONATA SDK and starts a new project using the `son-package` and `son-project` from the SDK.
      i. `son-workspace --init --project prj1`
   b) (optional) Functionality to be included from the catalogue. The catalogue supports storage of NS and VNF descriptors in JSON and YAML format, which can be retrieved via a REST API. Validation of the schema (using `son-schema` validation is executed at this time:
      i. firewalls/iptables, iperf, ...
      ii. Three-tier web application, load balancer, nginx, django as application server, postgresql as database. E.g., in docker containers. Load balancing via DNS? Or nginx directly?
   c) Joe specifies this service using the SONATA schema of the network service and creates a package. At the time of packaging the project, the schema validators of `son-schema` are also executed.

i. `son-package --project prj1`

d) (Optional) Joe wants to use the SONATA emulator to locally test/verify the new service using `son-emu`. In the emulator, basic service chaining between NFs is supported using VLANs.

e) Joe uploads this package to the SP gatekeeper (having obtained a key for it from Eve,'using `son-push`.

i. `son-push gatekeeperId --upload_package packagename`

f) The service is then available to be deployed on demand by the BSS

i. `son-push gatekeeperId --deploy_package packagename`

g) Service triggers VIM to start it up.

h) User of the service, Adam does curls, and look, it works

i. `curl www.newservice.demo`

4. Step: Monitor the running service and the performance of the SP

a) Joe monitors the performance of the service by `son-monitor`. The tool already supports monitoring of basic node characteristics such as CPU, memory and storage, as well as monitoring network-related aspects such as bandwidth and delay.

b) (Optional)Joe can analyze the monitoring information using `son-analyze` to scrutinize the running service.

c) Eve monitors the performance of the SP through Gatekeeper GUI.

6.2 Future Story Steps to be Included in First Prototype

1. Step: Demonstrate ability to introduce a service which is managed by a SSMs

a) Joe writes in addition to the service a simple scaling SSM that will trigger creation of addition web servers

b) Service and its SSM are deployed (see same steps as above)

c) The SSM realizes overload situation

d) The SSM triggers installation of a new web server and reconfigures the chosen load balancing system.

1. Step: Demonstrate ability to introduce a service that its components are managed by FSMs

a) Joe writes in addition to the service a simple FSM that manages the lifecycle of the some of its network functions in s very specific manner

b) Service and its FSM are deployed (see same steps as above)

c) The FSM manages the lifecycle events as it should

6.3 Structure

Figure 6.1 shows a very high level representation of the main components of SONATA's SP architecture that enter the **Year One Storyboard**.

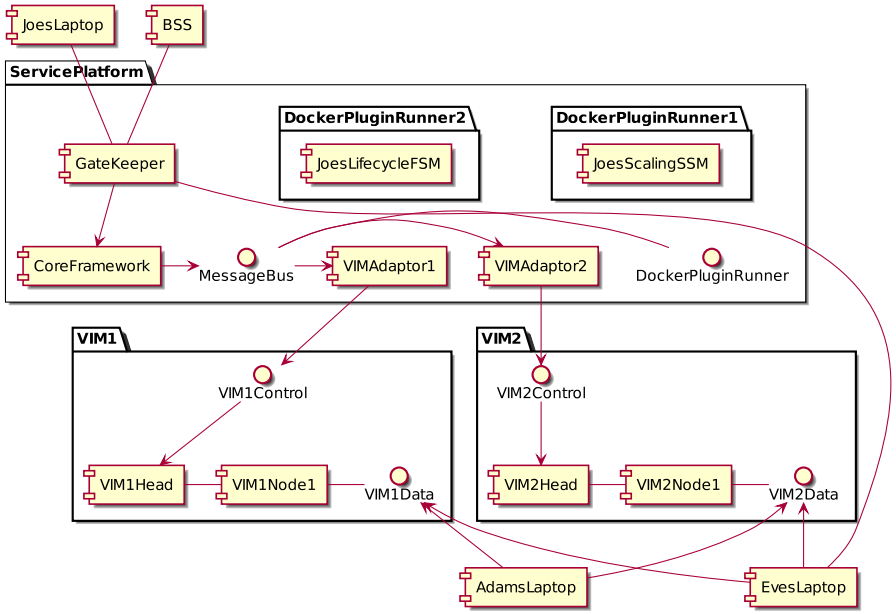

Figure 6.1: Overall structure of SONATA's main components

6.4 Message Sequence Charts

To help understand better the different interactions between all the above mentioned entities, we present in this section, the main message sequence chart.

Figure 6.2 shows an example of how a service could be installed.

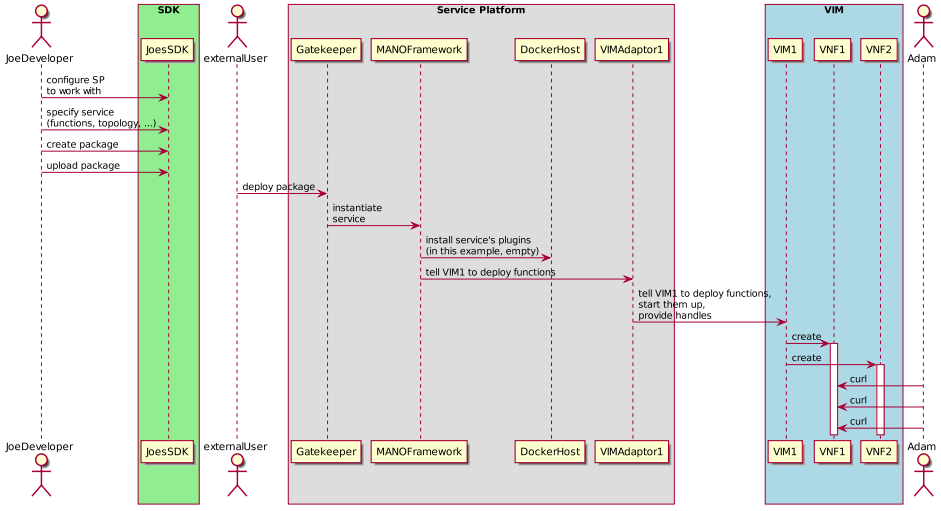

Figure 6.2: Install first service

7 Conclusion

In this deliverable we have described the current status of the development the SONATA Service Platform, as planned for this first milestone.

We have documented the design of the first version of a Service Platform that supports the flexibility and extensibility needed on a 5G perspective for our society. We have made it **modular** by using **plug-ins** and **micro-services**, and **flexible** by using a **message oriented middleware** such as **RabbitMQ**. These two characteristics (modularity and flexibility) are key aspects of adaptable systems such as the Service Platform that we need to implement.

We have reused existing (and open-sourced) products, enhancing them to fit SONATA's more demanding use cases. These catalogues and repositories are in line with the most up-to-date and rapidly evolving relevant standards, which allows the project to play a significant role in those communities.

By considering **Service Specific Managers** (SSMs) and **Function Specific Managers** (FSMs) as some of those **plug-ins**, we effectively give the Service Platform owner the ability to adapt platform's behaviour to unexpected scenarios, or even, for some selected and pre-authorised developers, to provide service or function specific managers (e.g., lifecycle, deployment at a specific location, scaling, etc.). We provide a set of default SSMs/FSMs that allows the Service Platform to work off-the-shelf, but the mechanisms to support other SSMs and FSMs are already in place in the first prototype.

We have opted for the current *de facto* standard for infrastructure virtualisation, **OpenStack**, but the flexibility of the platform will allow us to also use other infrastructure virtualisation implementation, or even another instance of the SONATA Service Platform.

We are building the Service Platform in the same manner we advocate Developers should develop services submitted to it: in an **agile** way, progressively, and not in a 'big-bang' integration at the end of the project. This is only possible when very **efficient** and **high quality** processes, such as **continuous integration** and **continuous deployment** are in place (these aspects of the process will be detailed in another deliverable of the project). For cloud providers wanting to play a significant role under the tough challenges of 5G, a DevOps approach is a must.

We have based our implementation on **open-source** products and technologies, like the before mentioned **RabbitMQ** and **OpenStack**, **Docker** containers, etc., taking advantage of the high quality of these products and technologies, and embracing a movement of which we also want to be a part of.

And, last but not the least, future work will include:

- **New features:** Of these, being able to control who can access what and a robust instantiation process are the most important ones, but we still have a long list of them, like still missing default SSMs/FSMs, (re-)use of public services or functions catalogues, etc.;
- **Support other VIM implementations:** like the Service Platform itself, it would allow different platform owners to include their instances in a 'federation' of service platforms, onto which services could be deployed according to optimisation criteria previously unknown;
- **Improve test coverage:** although the exact value of test coverage metric *per se* is not

relevant, tests defined above (a huge step already!) are not yet all automated. For a true **DevOps** project, this can not be;

- **Improve automation:** we are currently just using the **Integration** environment, but we need to automate our processes into such a level that we can have a new environment ready in just a few minutes.

A Details of son-schema

The SONATA schemata (son-schema) are used to describe and specify the various descriptors used in SONATA. The schemata that act as ground truth for the whole SONATA project. They are based on the JSON schema specification and can be used to verify formal correctness of descriptors, as a basis for databases, and as a blueprint for code implementations. In the following we describe the schemata for the VNFD, the NSD, and the package descriptor. Further information can be found at [8].

A.1 VNF Descriptor Schema

A VNF Descriptor (VNFD) is a deployment template which describes a VNF in terms of deployment and operational behaviour requirements. The VNFD also contains connectivity, interface and KPIs requirements that can be used by NFV-MANO functional blocks to establish appropriate Virtual Links within the NFVI between VNFC instances, or between a VNF instance and the endpoint interface to other Network Functions. Our function descriptor schema specifies the content of a VNFD. It is based on the T-NOVA [10] flavor of the ETSI VNFD that can be found at [22]. It is, however, adapted and extended to meet the SONATA specific needs.

A.1.1 Sections of the Function Descriptor

Below we discuss the various sections of a network function descriptor. The general descriptor section contains some of the mandatory fields that have to be present in each and every function descriptor. All other sections might be optional.

General Descriptor Section

At the root level, we first have the mandatory fields, that describe and identify the virtual network function in a unique way.

- **descriptor_version** identifies the version of the function descriptor schema that is used to describe the network function.
- **$schema** (optional) provides a link to the schema that is used to describe the network function and can be used to validate the VNF descriptor file. This is related to the original JSON schema specification.

Moreover, the VNF signature, i.e the vendor, the name, and the version, is of great importance as it identifies the VNF uniquely.

- **vendor** will identify the VNF uniquely across all VNF vendors. It should at least be comprised of the reverse domain name that is under your control. Moreover, it might have as many sub-groups as needed. For example: eu.sonata-nfv.nec.
- **name** is the name of the VNF without its version. It can be created with any name written in lower letters or alphanumeric symbols.

- **version** names the version of the VNF descriptor. Any typical version with numbers and dots, such as 1.0, 1.1, and 1.0.1 is allowed here. The VNF version must be increased with any new (changed) instance of the network function descriptor. Please note: The whole network function is composed of the descriptor and other artifacts, like virtual machine images. Thus, the network function may change, even if the description remains constant, just because another artifact changes. This might or might not be reflected in the version of the package descriptor.

The general descriptor section also contains some optional components as outlined below.

- **author** (optional) describes the author of the network function descriptor.
- **description** (optional) provides an arbitrary description of the VNF.

Virtual Deployment Units Section

The virtual deployment unit section contains all the information regarding the VDUs, such as virtual machines and containers, that constitute the virtual network functions. The section is mandatory and starts with:

- **virtual_deployment_units** contains all the virtual deployment units (VDUs) that are handled by the network function.

This section has to have at least one item with the following information:

- **id** represents a unique identifier within the scope of the VNF descriptor.
- **vm_image** (optional) specifies a reference to the virtual machine image (or container) that is used for the virtual network function. The image location can be a local file, a file within a package, a remove location, that might be accessed via HTTP, or a reference within the SONATA service platform.
- **vm_image_format** (optional) specifies the image format, such as raw, vmdk, iso, and docker.
- **vm_image_md5** (optional) represent an MD5 hash of the virtual machine image. It is highly recommended to provide an MD5 hash, not only to verify the image, but to also make versioning of the whole virtual network function easier.
- **resource_requirements** details the resources required by the VDU even further.
- **connection_points** (optional) names the connection points offered by the VDU. The connection points can be used to interconnect various VDUs or to connect the VDU to an VNF connection point and to the outside world.
- **monitoring_parameters** (optional) names the monitoring parameters that are collected for this specific VDU and used, e.g. to trigger scaling operations.
- **scale_in_out** (optional) specifies the minimum and maximum number if VDU instances.

Connection Points Section

- **connection_points** (optional)

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **id** A VNF-unique id of a connection point which can be used for references.
- **type** The type of connection point, such as a virtual port, a virtual NIC address, a physical port, a physical NIC address, or the endpoint of a VPN tunnel.
- **virtual_link_reference** (optional) (deprecated) A reference to a virtual link, i.e. the virtual_links:id.

Virtual Links Section

- **virtual_links** (optional) A VNF internal virtual link interconnects at least two connection points.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **id** A VNF-unique id of the virtual link which can be used for references.
- **connectivity_type** The connectivity type, such as point-to-point, point-to-multipoint, and multipoint-to-multipoint.
- **connection_points_reference** The references to the connection points connected to this virtual link.

VNF Lifecycle Events Section

- **lifecycle_events** (optional) An array that contains VNF workflows for specific lifecycle events such as start, stop, scale_out, update, etc.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **vnf_container** The VNF container that is associated with the lifecycle event.
- **events** The actual event such as start, stop, scale_out, update, etc.

Deployment Flavours Section

- **deployment_flavour** (optional) The flavours of the VNF that can be deployed.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **flavour_key** A VNF-unique id of the deployment flavour which can be used for references.
- **vdu_reference** A reference to the VDU, vdu:id.

Monitoring Rules

- **monitoring_rules** (optional) The rules used for monitoring.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **name** The name of the monitoring rule.
- **description** (optional) An arbitrary description of this monitoring rule.
- **duration** The duration the condition has to be met before an event is fired.
- **duration_unit** (optional) The unit of the duration, such as seconds, minutes, and hours.
- **condition** The condition, i.e. a Boolean expression, that must be met to fire the event.
- **notification** A list of notifications that are fired when the condition is met.

A.2 NS Descriptor Schema

A Network Service Descriptor (NSD) is a deployment template for a Network Service referencing all other descriptors which describe components that are part of that Network Service. Our network service descriptor schema specifies the content of a NSD. It is based on the T-NOVA [2] flavor of the ETSI NSD that can be found at [3]. It is, however, adapted and extended to meet the SONATA specific needs.

A.2.1 Sections of the Service Descriptor

Below we discuss the various section of a network service descriptor. The general descriptor section contains some of the mandatory fields that have to be present in each and every service descriptor. All other sections might be optional.

General Descriptor Section

At the root level, we first have the mandatory fields, that describe and identify the virtual network service in a unique way.

- **descriptor_version** identifies the version of the service descriptor schema that is used to describe the network service.
- **$schema** (optional) provides a link to the schema that is used to describe the network service and can be used to validate the VNF descriptor file. This is related to the original JSON schema specification.

Moreover, the service signature, i.e the vendor, the name, and the version, is of great importance as it identifies the network service uniquely.

- **vendor** will identify the NS uniquely across all NS vendors. It should at least be comprised of the reverse domain name that is under your control. Moreover, it might have as many sub-groups as needed. For example: eu.sonata-nfv.nec.
- **name** is the name of the NS without its version. It can be created with any name written in lower letters or alphanumeric characters.

- **version** names the version of the NS descriptor. Any typical version with numbers and dots, such as 1.0, 1.1, and 1.0.1 is allows here. The NS version must be increased with any new (changed) instance of the network function descriptor. Please note: The whole network service is composed of the descriptor and other artifacts, like VNFs. Thus, the network service may change, even if the description remains constant, just because another artifact changes. This might or might not be reflected in the version of the package descriptor.

The general descriptor section also contains some optional components as outlined below.

- **author** (optional) describes the author of the network service descriptor.
- **description** (optional) provides an arbitrary description of the network service.

Network Functions Section

The network functions section contains all the information regarding the VNFs that constitute the virtual network functions. The section is mandatory and starts with:

- **network_functions** contains all the VNFs that are handled by the network service.

This section has to have at least one item with the following information:

- **vnf_id** represents a unique identifier within the scope of the NSD.
- **vnf_vendor** as part of the primary key, the vendor parameter identifies the VNFD.
- **vnf_name** as part of the primary key, the name parameter identifies the VNFD.
- **vnf_version** as part of the primary key, the version parameter identifies the VNFD.
- **description** (optional) a human-readable description of the VNF.

Connection Points Section

- **connection_points** (optional) The connection points of the overall NS, that connects the NS to the external world.

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **id** An NSD-unique id of a connection point which can be used for references.
- **type** The type of connection point, such as a virtual port, a virtual NIC address, a physical port, a physical NIC address, or the endpoint of a VPN tunnel.
- **virtual_link_reference** (optional) (deprecated) A reference to a virtual link, i.e. the virtual_links:id.

Virtual Links Section

- **virtual_links** (optional) A NS internal virtual link interconnects at least two connection points.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **id** A NS-unique id of the virtual link which can be used for references.
- **connectivity_type** The connectivity type, such as point-to-point, point-to-multipoint, and multipoint-to-multipoint.
- **connection_points_reference** The references to the connection points connected to this virtual link.

Forwarding Graph Section

- **forwarding_graph** The forwarding graph describes the traffic steering through the network service. A network service might have more than one forwarding graph.

This section has to have some of the following information:

- **id** The NS-unique id of the forwarding graph.
- **number_of_endpoints** The number of endpoints of a graph.
- **number_of_virtual_links** The number of virtual links in a graph.
- **constituent_virtual**_links References to the virtual links that constitute the forwarding graph.
- **constituent_vnfs** References to the VNFs that constitute the forwarding graph.
- **network_forwarding_paths** The path, i.e. a concatenation of virtual links and VNFs, of the forwarding graph.

VNF Lifecycle Events Section

- **lifecycle_events** (optional) An array that contains VNF workflows for specific lifecycle events such as start, stop, Scale_out, update, etc.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **start** The start event, executed whenever the network service starts.
- **stop** The stop event, executed when the network service stops.
- **scale_out** The scale-out event, when the network service is scaled out.
- **scale_in** The scale-in event, when the network service is scaled in.

Auto-Scale Policy Section

- **auto_scale_policy** (optional) The auto-scale policy connects monitoring event with actions that are executed when some given criteria are met.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **criteria** The criteria that have to be met to execute the given action.
- **action** A list of actions that are execute when the criteria is met.

Monitoring Parameters Section

- **monitoring_parameters** (optional) The parameters used for monitoring.

While the parent section is optional, once it is specified it has to have at least some of the following information:

- **description** A human-readable description of the monitoring parameter.
- **metric** The metric to measure. The metric has to be supported by the service platform.
- **unit** The unit in which the metric is measured.

A.3 Package Descriptor Schema

The package descriptor file specifies the content, artifacts, and dependencies of a SONATA package. The corresponding schema file specifies the structure of the package descriptor. It makes sure the relevant information is provided to parse the package in a meaningful way. It can be used to validate the package descriptor file.

A.3.1 Sections of the Package Descriptor

Below we discuss the various section of a package descriptor. The general descriptor section contains some of the mandatory fields that have to be present in each and every package descriptor. All other sections are optional.

General Descriptor Section

On the root level, the general descriptor section contains the mandatory fields required in the package descriptor.

- **descriptor_version** identifies the version of the package descriptor schema that is used to specify the file structure.
- **$schema** (optional) provides a link to the schema that is used to specify the file structure and can be used to validate the package descriptor file. This is related to the original JSON schema specification.

Moreover, the package signature, i.e. the package_group, the package_name, and the package_version, is of great importance, as it identifies the package uniquely.

Best practices for creating the signature can be derived from the Java Maven naming conventions for groupId, artifactId, and version. To this end, the package_group, the package_name, and the package_version should be named as follows:

- **package_group** will identify the package uniquely across all packages. It should at least be comprised of the reverse domain name that is under your control. Moreover, it might have as many sub-groups as needed. For example: eu.sonata-nfv.nec.
- **package_name** is the name of the package without its version. It can be created with any name written in lower letters or alphanumeric characters.
- **package_version** names the version of the package. Any typical version with numbers and dots, such as 1.0, 1.1, and 1.0.1 is allows here. The package version must be increased with any new (changed) instance of the service.

The general descriptor section also contains some optional components as outlined below.

- **package_maintainer** (optional) describes the maintainer of the package, like John Doe, NEC.
- **package_description** (optional) provides an arbitrary description of the package.
- **package_md5** (optional) provides an MD5 hash over the package content, i.e. all files contained in the package EXCEPT the package descriptor, i.e. /META-INF/MANIFEST.MF, as this file contains this hash.
- **package_signature** (optional) provides a cryptographical signature over the package content, i.e. all files contained in the package EXCEPT the package descriptor, i.e. /META-INF/MANIFEST.MF. Thus, a package customer can verify the integrity and the origin of the package.
- **entry_service_template** (optional) specifies THE service template of this package. In general, the package can contain more than one network service descriptor as dependencies. In order to identify the descriptor that describes the service of this package, it has to be named here.
- **sealed** (optional) is a Boolean value that states whether this package is self-contained, i.e. it already contains all its relevant artifacts (true), or it has external dependencies that may have to be provided from somewhere else. Default is false.

Package Content Section

The package content section contains all the artifacts that are contained and shipped by the package. The section is optional and starts with:

- **package_content** (optional) holds an array of artifacts contained in the package

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **name** specifies the path to the resource in the package.
- **content_type** specifies the type of content, like application/vnd.sonata.service_template
- **md5** (optional) specifies an MD5 hash of the resource.
- **sealed** (optional) overrides the default sealed value specified in the general descriptor section on a per-artifact basis.

Package Resolver Section

The package resolver sections contain information about catalogues and repositories needed to resolve the dependencies specified in this package descriptor. This information might be used in addition to the default catalogues and repositories configured already on the service platform (or the SDK). The section is optional and starts with:

- **package_resolvers** (optional) holds an array of catalogues used to resolve dependencies and download additional packages.

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **name specifies** the path to the catalogue.
- **credentials** (optional) provides the credentials that might be needed to access the catalogue.

Package Dependencies Section

In the package dependencies section, one can specify additional packages this package depends up on. The packages are automatically downloaded from the various catalogues provided either by default from the service platform or as configured in the package resolver section. The section is optional and starts with:

- **package_dependencies** (optional) holds an array of packages this packages depends up on.

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **name** specifies the name of the package, similar to the name of this package.
- **group** specifies the name of the group, similar to the group name of this package.
- **version** specifies the version or version ranges of the package that is needed. For example one can specify the exact version like, 1.0.1-beta, but also ranges such as >= 1.0, = 1.0 && < 2.0, 1.1 || 1.2, etc.
- **credentials** (optional) provides the credentials that might be needed to use this package.
- **verification_key** provides the public key of the package maintainer to verify the package.

Artifact Dependencies Section

- **artifact_dependencies** (optional)

While the parent section is optional, once it is specified it has to have at least one item with the following information:

- **name specifies** the name of the artifact to download.
- **url** specifies the URL where to download the package from. Moreover, there needs to be a protocol handler that is able to download the artifact using the given protocol, like HTTP. For now, we only support HTTP and HTTPS. Thus, the URL has to start with one of these protocols.
- **credentials** (optional) provides the credentials that might be needed to download the artifact.

B 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

FSM Function-Specific Manager

FSMD Function-Specific Manager Descriptor

GUI Graphical User Interface

IaaS Infrastructure as a Service

IDE Integrated Development Environment

IoT Internet of Things

JMS Java Messaging System

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

OSS Operations Support System

PDPackage DescriptorPSA 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

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

C 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.

D Message Topics

As further explained in the main parts of this document, the communication between the different components of SONATA's service platform is based on asynchronous messaging system (see Section 3). As explained in D2.2 (see [6]) and seen in Figure 1.1 there are two types of message brokers:

- Main message broker used for the communication between the executive plugins
- Small dedicated message brokers used for the communication between the executive plugins and the FSMs/SSMs

The tables below list the messages used in the implementation of the MANO framework.

D.1 Events between MANO plugins

D.1.1 Topic category: platform.*

Table D.1 lists the messages on the main message broker used between plugins for platform related operations

- {PID} = plugin id: UUID identifying a plugin after its registration to the system. A new PID is created each time a plugin is registered so that we can have multiple instances of the same plugin assigned to the system.

Table D.1: Platform Messages on the Main Broker

Topics	Description
platform.management.plugin.register	Messages to register new plugins to the system
platform.management.plugin.deregister	Messages to remove a plugin from the system
platform.management.plugin.status	Messages broadcasted by plugin manager. Informs other plugins about plugin changes (e.g. new plugin connects to the system)
platform.management.plugin.{PID}.heartbeat	Heartbeat messages generated by active plugins
platform.management.plugin.{PID}.lifecycle.start	Message to start a plugin
platform.management.plugin.{PID}.lifecycle.pause	Message to pause a plugin
platform.management.plugin.{PID}.lifecycle.stop	Message to stop a plugin

D.1.2 Topic category: service.*

Table D.2 lists the messages on the main message broker used between plugins for network service related operations

- {SIID} = service instance id: UUID identifying a particular service instance (a running service)

Table D.2: Network Service Messages on the Main Broker

Topics	Description
service.inventory.onboard	Messages for the service on-boarding procedure (i.e. triggered by GK)
service.inventory.catalog.*	CRUD operations for service catalog access
service.inventory.repository.*	CRUD operations for service repository access
service.instance.create	Create a service instance
service.instance.{SIID}.lifecycle.start	Messages for the operations needed for managing the start of a NS
service.instance.{SIID}.lifecycle.stop	Messages for the operations needed for managing the stop of a NS

D.1.3 Topic category: infrastructure.*

Table D.3 lists the messages on the main message broker used between plugins for infrastructure related operations

Table D.3: Infrastructure Messages on the Main Broker

Topics	Description
infrastructure.management.compute.add	Registers a new VIM to the platform. Requires the VIM vendor, endpoint and credentials. Returns success or error code, and an UUID to identify the new VIM.
infrastructure.management.compute.remove	Remove a VIM from the platform. Requires the VIM type and UUID. Returns success or error code, and an UUID to identify the new VIM.
infrastructure.management.compute.list	Returns the available VIMs along with relevant information
infrastructure.management.compute.resourceAvailability	Check if the VIM can handle the deployment of a service, given the VIM uuid and the service resource constraints. Returns success or error code
infrastructure.service.deploy	Trigger the deployment of a new service, given a SONATA NSD and VNFDs in YAML. Returns instance information for the deployed service

D.2 Events Between Executive Plugins and FSMs/SSMs

D.2.1 Messages for FSM/SSM registration and lifecycle management

Table D.4 lists the messages on the small dedicated message broker used for FSMs/SSMs lifecycle management

- {SIID} = service instance ID: UUID identifying a particular service instance (a running service)
- {executivePlugin}: e.g., a placement/scaling, monitoring, or lifecycle management executive plugin

Table D.4: FSM/SSM Lifecycle Management Messages on Small Broker

Topics	Description
ssm.management.{executivePlugin}.{SIID}.register	Messages to register new SSMs to the {executivePlugin}

E Bibliography

[1] 5g ppp. 5g-ppp brochure. Website, 2016. Online at https://5g-ppp.eu/wp-content/uploads/2016/02/BROCHURE_5PPP_BAT2_PL.pdf.

[2] 5g ppp. 5g-vision-brochure. Website, 2016. Online at https://5g-ppp.eu/wp-content/uploads/2015/02/5G-Vision-Brochure-v1.pdf.

[3] Mike Cohn. Advantages of the as a user, i want user story template. Website, April 2008. Online at https://www.mountaingoatsoftware.com/blog/advantages-of-the-as-a-user-i-want-user-story-template.

[4] The OpenStack Community. Openstack heat project. Website, May 2016. Online at https://wiki.openstack.org/wiki/heat/.

[5] The OpenStack Community. Openstack neutron project. Website, May 2016. Online at https://wiki.openstack.org/wiki/neutron/.

[6] SONATA Consortium. Sonata deliverable 2.2: Architecture design.

[7] SONATA consortium. H2020-ict-2014-2 - proposal submission forms. Website, November 2014.

[8] SONATA consortium. D3.1: Basic sdk prototype. Website, May 2016.

[9] SONATA consortium. D5.2: Integrated lab based sonata platform. Website, June 2016.

[10] T-NOVA consortium. D3.1: Orchestrator interfaces. Website, September 2015. Online at http://www.t-nova.eu/wp-content/uploads/2016/03/TNOVA_D3.1_Orchestrator_Interfaces_v1.0.pdf.

[11] Inc. Cunningham Cunningham. Model-view-controller. Website, 2004. Online at http://c2.com/cgi/wiki?ModelViewController.

[12] Ralph Johnson Richard Helm Erich Gamma, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. October 1994.

[13] Roy Fielding. Architectural styles and the design of network-based software architectures. Website, 2000. Online at http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

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

[15] Tom Huston. What is microservices architecture? Website, 2014. Online at https://smartbear.com/learn/api-design/what-are-microservices/.

[16] ITU-T. Imt2020 5g draft recommendations. Website, 2015. Online at http://www.itu.int/md/T13-SG13-151130-TD-PLEN-0208/en.

[17] NGMN. Ngmn 5g white paper. Website. Online at https://www.ngmn.org/uploads/media/NGMN_5G_White_Paper_V1_0.pdf.

[18] The OpenDaylight Project. Virtual tenant network (vtn) project. Website, 2016. Online at https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):Main.

[19] The OpenStack Project. OpenStack: The Open Source Cloud Operating System. Website, July 2012. Online at http://www.openstack.org/.

[20] Eric Ries. *The Lean Startup*. 2009. Online at http://theleanstartup.com.

[21] Pivotal Software. Rabbitmq. Website. Online at https://www.rabbitmq.com/.

[22] ETSI NFV WG. Network functions virtualisation (nfv); management and orchestration. Website, 12 2014. Online at http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf.

[23] ETSI NFV WG. Etsi gs nfv-eve 005 v1.1.1 (2015-12), network functions virtualisation (nfv); ecosystem; report on sdn usage in nfv architectural framework. Website, 12 2015. Online at http://www.etsi.org/deliver/etsi_gs/NFV-EVE/001_099/005/01.01.01_60/gs_nfv-eve005v010101p.pdf.