POLYTECHNIC OF TURIN

Master Degree Thesis in Computer Engineering

Experimentation of End-to-End Telco Services through a Service Portal Operating on an ONAP Orchestrator

Supervisors
prof. Guido Marchetto
prof. Fulvio Risso

Candidate
Serena Flocco

Academic Year 2019-2020

Summary

The infrastructure used by network operators to deliver services to users at the edge of the network is evolving towards a Cloud-native one. As a consequence, end-to-end services are implemented as chains of PNFs and VNFs and an automated process is required to manage their lifecycle. The Open Network Automation Platform (ONAP) project is one of the available alternatives allowing to manage this kind of services, providing service design, creation, KPI monitoring and closed-loop control capabilities.

In order to help Communication Service Providers (CSPs) to manage the described infrastructure and all the business operations related to it, the TeleManagement Forum (TM Forum) organization has standardized a suite of REST based APIs in its Open API program. In particular, service providers' business operations have been split into different categories, each one referring to a different layer of their infrastructure. The Open API program defines a set of REST API specifications to standardize the functionalities to be implemented at each layer and the communication among them. This thesis work focuses on three layers defined in the TM Forum's standard: Resource, Service and Product. Resources, also called Resource Facing Services (RFS) represent the category of services internal to the service provider, which are needed to support a Service at higher layer, i.e. a Customer Facing Service (CFS). Therefore, a CFS represents the commercial view of a service exposed to customers.

The ONAP platform is placed at Resource layer, which is also the lowest one. In particular, it is responsible for managing resources located in different Cloud Central Offices (Cloud-COs) and communicates with applications at Service layer through its External APIs component, also called ONAP NBI. ONAP External APIs implementation is based on TM Forum's REST specifications and allow applications at Service layer to trigger actions on resources (i.e., creation, deletion or modification) by interacting with ONAP in a standard way. This component has been studied, tested and evaluated. For what concerns the Service layer, Service Order Management (SOM) applications can be placed here. An example is Service Resolver developed by Orange, an open source project which has been tested and improved. Service Resolver exposes a set of REST APIs based on TM Forum's standard. It accepts requests from BSS applications and maps them into requests to be sent to ONAP External APIs component in order to trigger the instantiation process of the resources

needed to support the required service. Finally, BSS applications refer to the Product layer and are responsible for mapping customer requests into requests to be sent to SOM applications. The work performed at this layer regards the implementation of a BSS application called Service Portal, composed by a GUI from which customers can order the desired product and a backend component sending requests to Service Resolver, allowing it to instantiate the service which corresponds to the product purchased by the customer.

The chain of functionalities implemented in the three described layers realises the steps required by service providers to instantiate end-to-end services according to customers requests. In conclusion, the purpose of this thesis is to provide a complete analysis of this process with details on how it has to be implemented at each layer, thanks to the contribution given to Service Resolver project and the development of the Service Portal.

Acknowledgements

Of course, aknowledgments are not my best. And everything I can write here will never be sufficient to show my gratitude to people who have never left me alone during these years, despite the many difficult moments.

Firstly, I would like to thank my parents for always supporting me and being proud of my work. Thanks to my sisters, Michela and Alessia, the hardest thing for me has been to be far from you.

Thank you, Mirko, there are no words to describe the importance of your presence by my side.

Thanks to all my friends, those that have always been there for me. Each one of you knows. Thank you Nicola & Nicola for the great time spent together in Turin and thank you, Dalila, you are one of the few persons able to understand me. You are my second family.

Finally, thanks to professors Fulvio Risso and Guido Marchetto for giving me this opportunity, and thanks to team members who followed me during my thesis work at Telecom Italia. It was an unforgettable experience.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

In recent years, the diffusion of new paradigms such as Software Defined Networking (SDN) and Network Function Virtualization (NFV) has determined a radical change of the way to operate networks. The transformation of telecommunication networks is related to the possibility to virtualize network components and to decouple control and data plane, allowing to reduce costs of ownership and to increase efficiency. As consequence, Communication Service Providers (CSP) aim to exploit the described technologies also in public networks and, therefore, their infrastructures are evolving toward Cloud-native ones. In particular, Central Offices, which nowadays are still mostly hardware-based, will be replaced soon by a cloudified version: the Cloud Central Office (Cloud-CO) [1], whose architecture has been standardized by the Broadband Forum (BBF) organization [2].

This cloudification process is lead by both economical and technical reasons. First of all, the increasingly wide distribution of Cloud Services is related to their flexibility, higher performance and rapid availability. Thus, the adoption of Cloud-COs aims to provide all these advantages, combined with the possibility to satisfy customers' requests in a dynamical way. Moreover, they allow CSPs to build networks which are adaptable, agile, scalable and dynamic, reducing Capital Expenditures (CapEx), migration and Operational Expenditures (OpEx) costs, cutting the mean time of delivery. In other words, this gives network operators the opportunity to run a single network with all varieties of access technologies, and flexibly deploy new value-added services.

In this scenario, end-to-end services delivered to users at the edge of the provider's network can be implemented as chains of Virtual Network Functions (VNF) and Physical Network Functions (PNF) located into different Cloud-COs. In order to manage the lifecycle of these services, an automated process playing the role of a Service Orchestrator is required. In this way, network operators can get rid of the vendor lock-in problem, introducing an additional economical benefit.

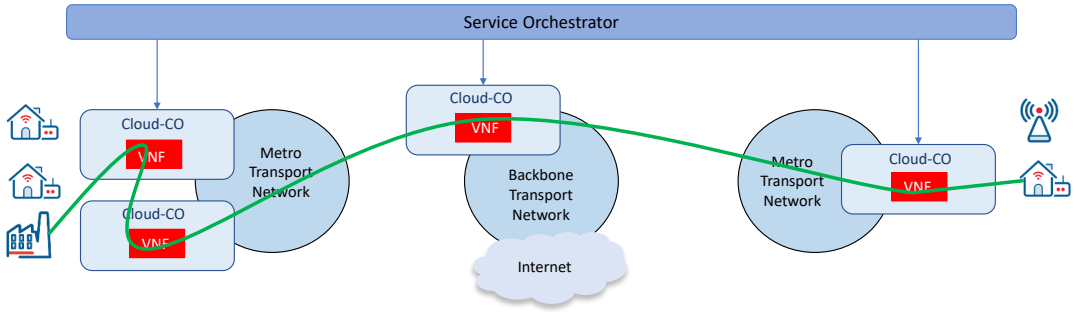


Figure 1.1. Cloud-COs-based infrastructure

The management of the described infrastructure is complex, therefore it has been split into different layers, which are described in the Business Process Framework (eTOM) [3], a hierarchical catalog of the key business processes required to run a service-focused business, which has been standardized by TeleManagement Forum (TM Forum) organization [4]. It is shown in figure 1.2.

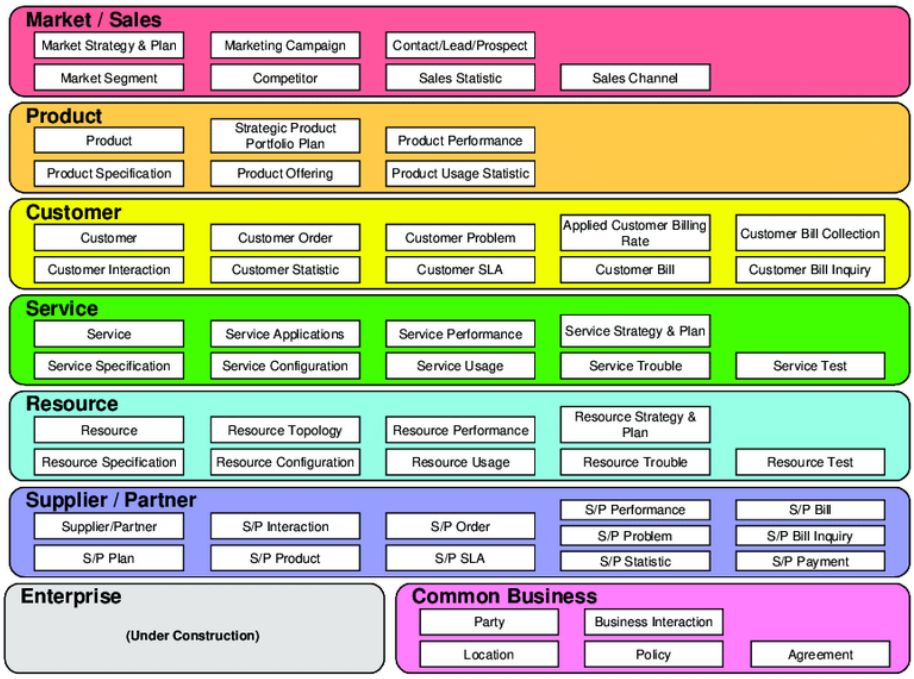


Figure 1.2. Business Process Framework (eTOM)

At Product layer, there are BSS[1] applications from which customers can buy the desired products, for example fiber connectivity service offered by the network provider. Customers' requests then need to be translated in requests to trigger instantiation of the corresponding service. These requests are managed by Service Order Management (SOM) applications at Service layer. Since this kind of services are usually implemented as chains of VNFs or PNFs, SOM applications have to decompose them into the list of resources to be instantiated by the network provider to fulfill customers' requests. These requests have to be sent to a network automation platform placed at Resource layer, able to manage the lifecycle of available resources (e.g. a VNF located in a Cloud-CO). Finally, the communication among applications at different layers is performed through REST APIs standardized by TM Forum.

1.2 Purpose of the thesis

In a context such as the one described in the previous section, telecommunications service providers are planning a new strategy to deliver services to users at the edge of the network, according to the new technologies adopted in a Cloud-based infrastructure and to their business needs. Consequently, they are studying and testing different solutions in order to select the most effective one to be used in the future.

This thesis work has been developed at Telecom Italia and aims to offer a detailed analysis of the functionalities to be implemented at different layers of eTOM and of how the communication among them works, an improvement of a SOM application developed by Orange and the implementation of a BSS application at Product layer.

In particular, regarding the Resource layer, the purpose is to study and understand how the ONAP orchestration platform works, focusing on the standard APIs exposed on its NorthBound Interface[2] towards SOM applications. At Service layer, the target is to exploit the knowledge acquired about ONAP to improve Service Resolver for ONAP, the SOM application developed by Orange, and to study its REST APIs based on TM Forum's standard. Finally, the BSS application to be implemented at Product layer aims to allow customers to trigger service instantiation or deletion through a Graphical User Interface (GUI), thanks to a backend component mapping

[1] BSS stands for *Business Support System* and includes all the components needed by telecommunications service providers to support business operations towards customers.

[2] The Northbound Interface, or simply NBI, represents the application interface with higher-level layers of software.

requests to Service Resolver, following the TMF specifications[3] studied at lower layers.

1.3 Thesis organization

The thesis is organized as follows:

- **Chapter 1 Introduction** describes the context and the motivation of this work.
- **Chapter 2 ONAP** provides an overview about the ONAP platform, focusing on some relevant components for this thesis.
- **Chapter 3 Service Resolver for ONAP** analyses the SOM application developed by Orange and shows the contribution given in order to improve it.
- **Chapter 4 Service Portal: architecture** describes the different components of Service Portal design and architecture, the internal interaction among them and the external one with the Service Resolver.
- **Chapter 5 Service Portal: implementation** focuses on the technical choices performed during Service Portal implementation phase, providing details about the tools, programming languages and frameworks used.
- **Chapter 6 Service Portal: validation** shows how the requirements provided by Telecom Italia have been satisfied and analyses Service Portal performance.
- **Chapter 7 Conclusion and Future Work** sums up the objectives reached thanks to this work and suggests some possible improvements.

[3] TMF specifications refer to the TM Forum documents which describe the standard to be followed in REST APIs implementation at each layer of eTOM. They are numbered and each number corresponds to a specific API (e.g., the TMF641 specification corresponds to Service Ordering Management API) [5].

Chapter 2

ONAP

2.1 Overview

The Open Network Automation Platform (ONAP) is an open source project hosted by the Linux Foundation [6]. It has been created in order to provide service design, creation and orchestration capabilities, with the possibility to handle the full lifecycle of services implemented exploiting the combination of SDN and NFV technologies, which are usually deployed in Cloud-based environments.

The project was born in 2017 with the aim of satisfying the rising need for a common automation platform which allows telecommunication and cloud service providers to deliver network services on demand, improving velocity and simplifying equipment interoperability and integration. In this way, CSPs can reduce costs of hardware maintenance, avoiding the installation of new data center equipment and the upgrade of CPE[1] devices. Services can be created dynamically and deployed in a virtualized environment, so lifecycle operations, such as instantiation, configuration changes and upgrades, are all simply software steps that can be fully automated. As a consequence, the combination of increased hardware utilization and adoption of open source software may results in reduced CapEx, while automation and hardware standardization can decrease OpEx.

The strength of ONAP derives from its massive scale automation capabilities for both physical and virtual network elements, thanks to the offering of a unified framework which allows vendor-agnostic and policy-driven service design, implementation, KPI[2] monitoring and lifecycle management. Service instantiation can be performed dynamically, since ONAP supports closed control loop automation with the ability to react to real-time changes. Service agility, interoperability and simplified

[1] CPE stands for *Customer Premise Equipment* and represents the hardware needed by users at the edge of the network to access providers' communication services.

[2] KPI or *Key Performance Indicator* provides a measurement of the success of an activity it is referred to. In this case, it is referred to perfromance parameters that are significant for a particular service.

integration are achieved thanks to its modularity and layered nature, the support of data modeling languages which allow a rapid service and resource deployment, and the suite of open NorthBound REST APIs provided. These characteristics allow ONAP to support multiple VNF environments by integrating with multiple VIMs[3], VNFMs[4], SDN Controllers and legacy equipment. Therefore, the described approach allows network and cloud operators to optimize their physical and virtual infrastructure for cost and performance; at the same time, ONAP's use of standard models reduces integration and deployment costs of heterogeneous equipment.

All these features and the fact ONAP is open source represent a powerful innovation, able to change how CSPs build and operate their networks, and which, therefore, includes several new architectural concepts.

Furthermore, the platform is in continuous evolution thanks to the developers community responsible for continuously updating software and improving its components and functionalities. The last published version is El Alto, and a new one is released every six months. As part of each release, the ONAP community also defines blueprints for key use cases, such as 5G, BBS (Broadband Service), CCVPN (Cross Domain and Cross Layer Virtual Private Network), Voice over LTE (VoLTE), and vCPE (virtual Customer Premise Equipment). Testing these blueprints with a variety of open source and commercial network elements during the development process provides the ONAP platform developers with real-time feedback on in-progress code, and ensures a trusted framework that can be rapidly adopted by other users of the final release.

In conclusion, ONAP will harmonize the way VNFs and network services are deployed, simplifying the interaction between CSPs and VNF vendors.

2.2 ONAP functional description

ONAP's service design, deployment and operations activities are provided by two major frameworks, the Design-time framework and Run-time one, as shown in figure 2.1.

Each function executed by ONAP is associated to a set of required activities. They can be summed up as follows:

- **Service design**: thanks to a robust design framework, which allows service definition in all aspects, it is possible to model resources and their relationships,

[3]The *Virtual Infrastructure Manager* (VIM) is responsible for controlling and managing the NFV infrastructure (NFVI) compute, storage, and network resources, usually within one operator's infrastructure domain [7].

[4]The VNFM or *VNF Manager* is a key component of NFV-MANO, the standard which allow interoperability of software-defined networking elements that use network function virtualization technology. In particular, the VNFM is responsible for lifecycle management of VNFs [8].

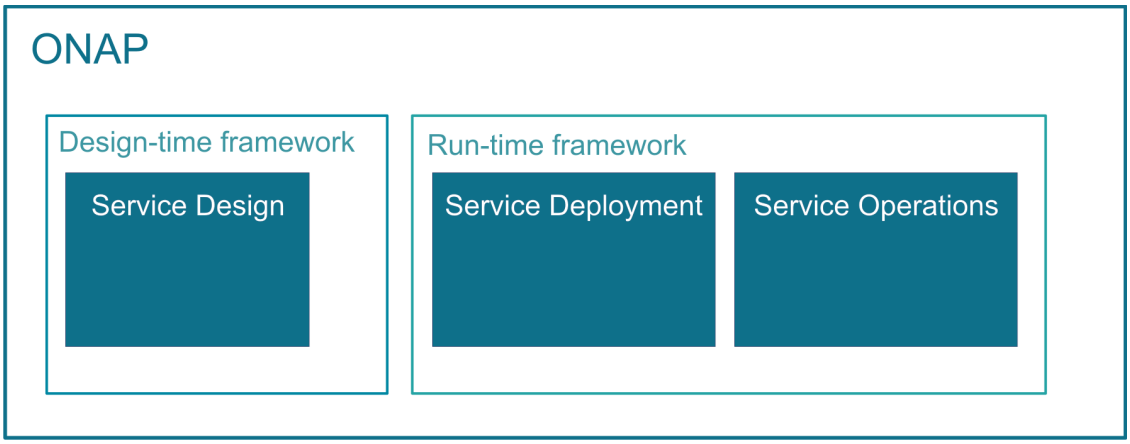


Figure 2.1. ONAP main functions

specify the policy rules controlling service behavior and applications, analytics and closed control loop events needed for a reactive management of the service.

- **Service deployment**: the policy-driven orchestration and control framework provides automated instantiation of the service and performs required actions on services.
- **Service operations**: the analytic framework monitors service behavior during its lifecycle based on the specified KPI parameters, collected data and policies to enable real-time response as required, making possible to deal with situations that require dynamical resource healing or scaling.

Each mentioned function is composed by a sequence of sub-tasks needed to complete it. The following guidelines show the main steps to be executed in a chronological order, presenting ONAP's functional structure:

1. Service design activity includes the following sequence of sub-tasks:
   - Planning VNF onboarding: checking which VNFs will be necessary for the required features
   - Creating resources to compose services
   - Distributing the defined services to ONAP components, by exploiting a publish/subscribe pattern
2. Service orchestration and deployment, which requires:
   - Definition of the necessary VNFs for the service
   - Definition of the steps to orchestrate the service
   - Selection of valid cloud region

- Calling cloud APIs to deploy VNFs
- Application of configuration on VNFs by controllers

3. Service operations, which are:

- Closed Loop design and deployment
- Collecting and evaluating event data

2.3 Architecture

ONAP architecture [9] defines two major areas, design-time and run-time, as described in the previous section. This separation allows for clean delineation between design and operational roles. Moreover, functions for managing ONAP itself are included, since it is a Cloud-native application.

The design-time environment allows to model resources, services, products and to define their management and control functions. This is achieved by using a common set of specifications, usually called “recipes”, and policies for controlling service behavior and process execution, which facilitates reuse of models and improves efficiency.

Components in the run-time framework execute the rules, policies and other models defined in the design and creation environment. The distribution process of models and policies from the design-time environment to ONAP run-time modules, such as the Service Orchestrator (SO), Controllers, Data Collection, Analytics and Events (DCAE) and Active and Available Inventory (A&AI), enable them to know the operations to be performed when a new service instance has to be created, modified or deleted, or when a workflow has to be applied in case service KPI parameters do not respect a given threshold.

Figure 2.2 provides a high-level view of the ONAP architecture and shows its various components.

As a Cloud-native application composed by different services, ONAP requires complex initial deployment and post-deployment management. Deployment methodology needs to be flexible since the platform needs to be highly reliable, scalable, secure and easy to manage. To satisfy all these requirements, ONAP architecture is microservices-based and its components are runned in Docker[5] containers with an optimized image size.

The ONAP Operations Manager (OOM) module is responsible for orchestration, lifecycle management and monitoring activities of all the other platform components, by using Kubernetes container orchestrator[6] to ensure CPU efficiency and resiliency

[5] Docker [10] is an open platform that provides the ability to package and run an application in an isolated environment called a container. More details will be provided in chapter 6.

[6] Kubernetes [11] is an open source platform for managing and orchestrating containerized applications in cloud environment.

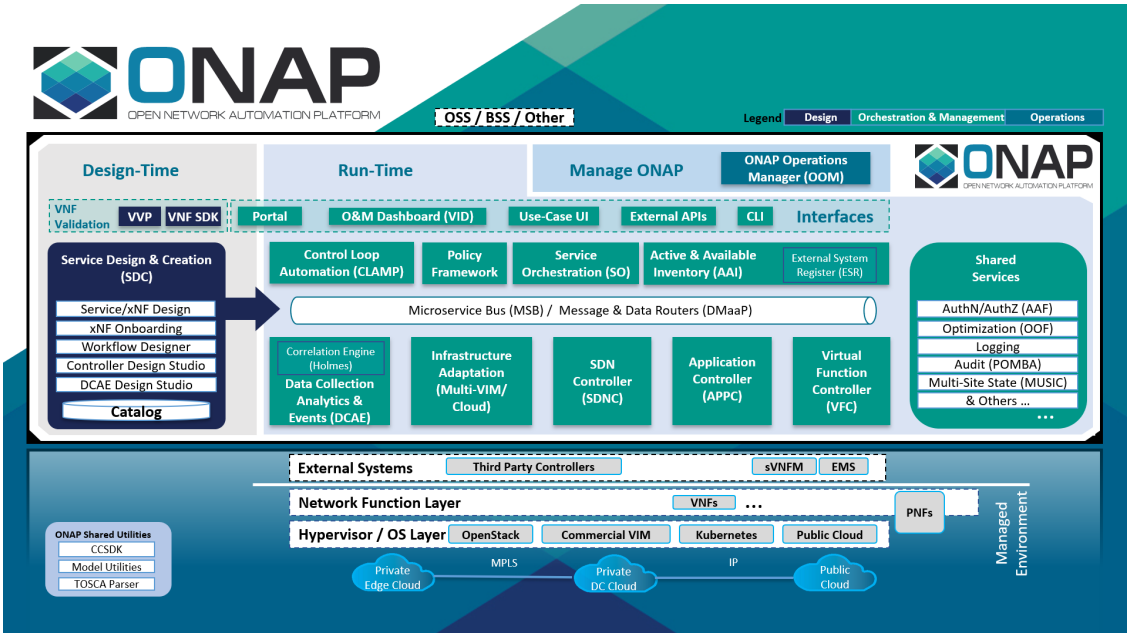


Figure 2.2. ONAP El Alto Architecture

of the managed components. Moreover, it supports a wide variety of Cloud infrastructures to suit operators' individual requirements.

In addition, ONAP offers a GUI and an API interface. Both design-time and run-time environments are accessed through the Portal Framework, with role-based access for service designers and operations personnel.

ONAP interfaces with other external subsystems. On the NorthBound Interface ONAP talks to OSS[7] and BSS applications. On the SouthBound Interface[8], ONAP communicates with the VIM, NFVI (Network Function Virtualization Infrastructure), and SDN controllers, as well as with the NFV Cloud hosting all the created resources, such as VNFs and PNFs.

Among all ONAP components, five of them are relevant for this thesis work: SDC, SO, A&AI, DMaaP and External APIs.

2.3.1 Service Design and Creation

The Service Design and Creation (SDC) project [12] is a tool for design-time activities such as VNFs onboarding, services and policies creation, workflows definition,

[7]The therm OSS, or *Operational Support System*, refers to all the systems needed by CSPs to manage their networks and to ensure their functionalities.

[8]The SouthBound Interface, also named SBI, refers to the application interface with lower-level layers of software.

data analytic applications onboarding and models distribution to run-time components. Moreover, it provides a role-based distinction of users (e.g.: designer, tester, administrator, governance).

SDC is catalog-driven, which means all the defined models are stored in a catalog, that actually represents a single source of service data. The centralization of this data allows providers to increase time to market for new digital services.

The SDC manages the catalog content and logical assemblies of catalog items to establish rules on VNFs realization in target environments. A virtual assembly of specific catalog items, together with related workflows and configuration data, defines how the deployment, activation, and life-cycle management of VNFs have to be executed.

The models defined in SDC describe asset capabilities and how they have to be managed. Inside the SDC Catalog, two levels of assets are managed:

- **Resource**: it represents a combination of one or more Virtual Function Components (VFCs), along with all the information necessary to instantiate, update, delete, and manage the Resource. A Resource also includes license-related information. There are three kinds of Resource:
  - Infrastructure, composed by the Cloud resources (e.g., Compute and Storage)
  - Network (e.g., a VNF)
  - Application (e.g., a load balancer)
- **Service**: it is an object composed by one or more Resources. Service Designers create Services from Resources, including all of the information needed to instantiate, update, delete, and manage the Service.

There are four major sub-components in SDC:

- The **Catalog**, which is the repository for Resources and Services.
- The **Design Studio** is used to create, modify, and add Resource and Service definitions in the Catalog.
- The **Certification Studio** is used to test new assets in order to certify them.
- The **Distribution Studio** is used to deploy certified assets.

In addition, the SDC exposes a set of REST APIs to the outside ONAP components and SDC Integrated Development Environment is accessible by designers through the ONAP portal. Thanks to its graphical interface and visual tools, users can drag and drop different components to model their service and to inspect the requirements provided by each VNF to the service. SDC offers an interface for distributing

the modeled services, TOSCA[9] artifacts and CSAR[10] files to SDN Controller, A&AI, Service Orchestrator and other ONAP run-time components, using DMaaP notitications.

2.3.2 Service Orchestrator

The Service Orchestrator (SO) component [15] is responsible for performing actions on services and automates the execution of tasks, rules and policies needed to control these services. ONAP SO performs orchestration at high level, thanks to its end-to-end view of the network infrastructure. In order to fulfill the required tasks, it interacts with four controllers, which are the Multi-VIM, the SDN Controller (SDN-C), the Application Controller (APP-C) and the Virtual Function Controller (VF-C). In addition, it communicates with SDC and A&AI to obtain topologies, configurations and resource models. The SO can be invoked by means of APIs calls from BSS applications or manually through VID[11]. SO takes also homing[12] decisions and manages rollbacks operations in case errors occur.

In order to understand the operations performed by SO to instantiate a service, the sequence of steps of instantiation flow [17] will be presented. Figure 2.3 shows an overview of the mentioned flow.

After the design phase and SDC service CSAR distribution, an instantiation request for the desired service can be sent to SO via VID Portal (ONAP GUI), UUI (User Interface), CLI (Command Line Interface) or External APIs module. The user select the service from the available ones that could be instantiated and finds the corresponding identifier. This information is stored in SDC Catalog, also available at VID terminal, where the user can see the list of service models, search the one he is interested in, and ask to deploy it.

ONAP provides three methods to instantiate a service:

- The **A La Carte** method requires the user to build and send operations for each object to be instantiated: service, VNFs and networks. In other words, once the service object has been created, the various VNFs or networks that compose it have not been instantiated yet. To build instantiation requests for all resources to be created, the user needs to collect by himself all necessary parameters and

[9] TOSCA is the acronym of *Topology and Orchestration Specification for Cloud Applications* [13] and is a standard modeling language used to describe the topology of Cloud-based web services, and how they are composed, managed and related among them.

[10] The SDC Service CSAR [14] is a package of artifacts which captures the information associated with a service defined at design time.

[11] The *Virtual Infrastructure Deployment* (VID) [16] enables users to ask for the creation or deletion of services and their components.

[12] Homing is intended as the process of determining the physical or virtual resources in which workloads will be placed.

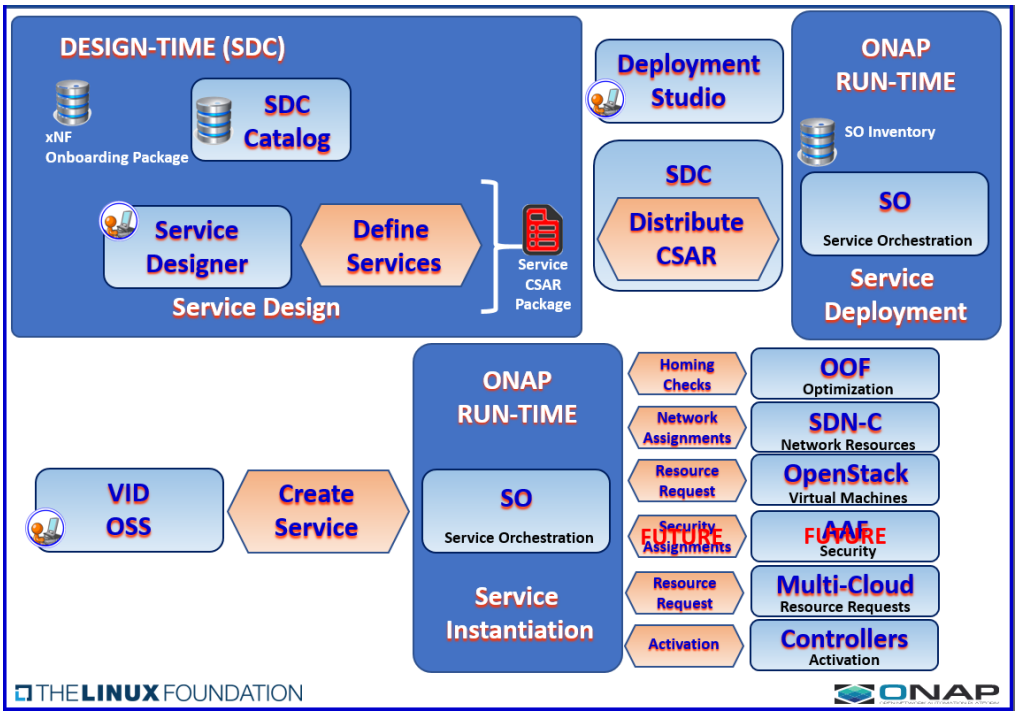


Figure 2.3. ONAP SO Service Instantiation Flow

associated values. These operations can be performed via VID, SO REST API or ONAP External APIs.

- The **Macro** method allows the user to build and send only one request to instantiate all objects composing the service: for this reason it is also known as "one-click instantiation". The SO derives the sequence of actions, called "building blocks", composing the Macro flow associated to the service and executes them in order to instantiate all resources. For this method, only SO REST API can be used.

- The **E2E** (End-to-End) method has the same output than the Macro one, but resources are instantiated one after the other by executing the associated BPMN workflow[13] instead of using building blocks. The BPMN workflow related to each resource has to be specified at design time.

When SO receives an instantiation request, the first step is the creation of a new record in the Active and Available Inventory. SO asks A&AI for creation of service

[13]BPMN, which means *Business Process Model and Notation* [18], is a standard notation used for business process modeling, which provides a graphical representation describing the steps that compose business processes.

record and of records for the resources, such as VNFs and PNFs, associated to the service. After this, it performs other operations such as service homing, Cloud resource requests, network assignments, instantiation, service activation and finally notifies to the user the final outcome of his initial request.

If the user requests to delete a service instance, the SO just performs the opposite process: it checks if service exists in A&AI, asks for Cloud resource deallocation, removes the active instance and all its resources.

2.3.3 Active and Available Inventory

The Active and Available Inventory (also called AAI or A&AI) [19] is the ONAP module storing real-time views of available Services and the relationships among them. It registers active and available Service instances and relevant information about them in an Inventory, keeping all these data up-to-date.

Data stored in AAI is continuously updated in real-time, according to the changes made within the Cloud environment. Since AAI is metadata-driven, new resources and services can be added quickly with Service Design and Creation (SDC) catalog models, by means of the model loader, such that long development cycles can disappear. Moreover, new item types can be added to Inventory through schema configuration files.

ONAP AAI exploits graph data technology to store relationships between Inventory items. This allows to identify chains of dependencies between items. AAI displays relationships between products and the services composing them, but also between services and their resources. It also shows relationships among different VNFs that are chained to realise an end-to-end service.

Data collected by AAI can be used during service delivery, analysis of problems, impact analysis, and many other processes, thanks to providing a reliable snapshot of the current state of all resources managed by the ONAP platform.

2.3.4 Data Movement as a Platform

The Data Movement as a Platform component [20], usually called DMaaP, is one of the two buses by means of which the different ONAP modules can communicate. It is designed to provide low cost data movement services, ensuring high performance. Its role is to allow data transport from any source to any destination, respecting the quality, security, and concurrency requirements needed to fulfill the business and customer requests, and using the appropriate format.

For what concerns DMaaP functional architecture, shown in figue 2.4, three areas can be identified:

- **Data Filtering**, which includes all the operations needed to pre-process data before transport begins. Data pre-processing is executed via data analytics and exploits compression mechanisms in order to reduce the size of data before the next processing steps.

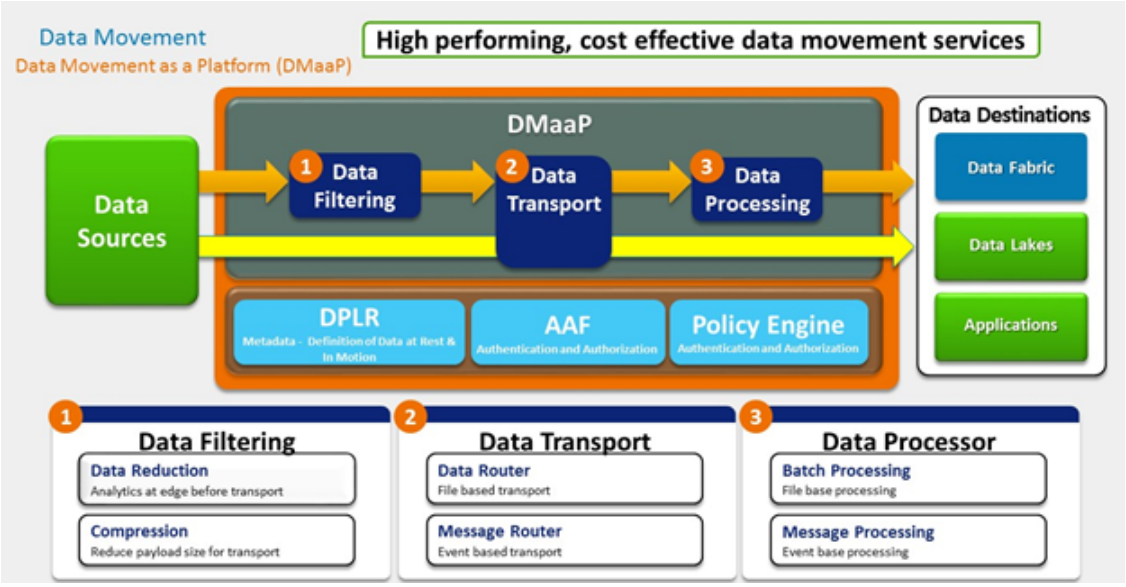


Figure 2.4. ONAP DMaaP functional architecture

- **Data Transport**, responsible for data movement operations, which can be file based or message based. Data can be propagated both inter data centers or intra data centers. During transport, ability to move data from any system to any system with minimal latency represents a fundamental requirement to be satisfied. Moreover, delivery and high availability must be guaranteed.

- **Data Processing**, where data transformation, aggregation, and analysis operations are executed with low latency and high throughput. In addition, it can ensure scalability and fault-tolerance to the traffic across data centers. Both batch and real-time data processing is guaranteed.

Inside DMaaP there are four sub-components. They are:

1. The **Message Router** (or simply MR), a messaging service based on the publish/subscribe pattern, able to guarantee reliability and to perform high-volume messages exchange. It exposes a RESTful API and it is available to clients as a web service. It is initially built over Apache Kafka[14].

2. The **Data Router** (DR), a project which aims to offer a common framework allowing producers to send data to consumers. DR exposes a RESTful interface to access it as a web service, known as the DR Publishing and Delivery API.

[14]Apache Kafka [21] is an event streaming platform, scalable and reliable, initially intended as a simple messaging queue and evolved after it was open sourced by LinkedIn in 2011.

3. The **Data Movement Director** (or DMD), which implements a client to the DMaaP platform and is responsible for publishing and subscribing data.

4. The **Data Bus Controller**, which is the DMaaP Provisioning API.

The Data Movement as a Platform component is used by ONAP modules to perform the distribution process, i.e. the propagation of service models defined in SDC Catalog to the microservices in ONAP run-time environment, with the purpose to make them available to allow service creation, modification, deletion and monitoring.

2.3.5 ONAP External APIs

The External APIs component [22] is also called ONAP NBI, which stands for North-Bound Interface. It brings to ONAP a set of REST APIs that can be used by external systems, such as BSS, to interact with other ONAP components, hiding all the complexity of these components. These APIs are based on TM Forum's REST specifications, that will be described in the next chapter.

Figure 2.5 provides a global view about ONAP External APIs architecture [23], showing its interaction with other ONAP components and the available API operation.

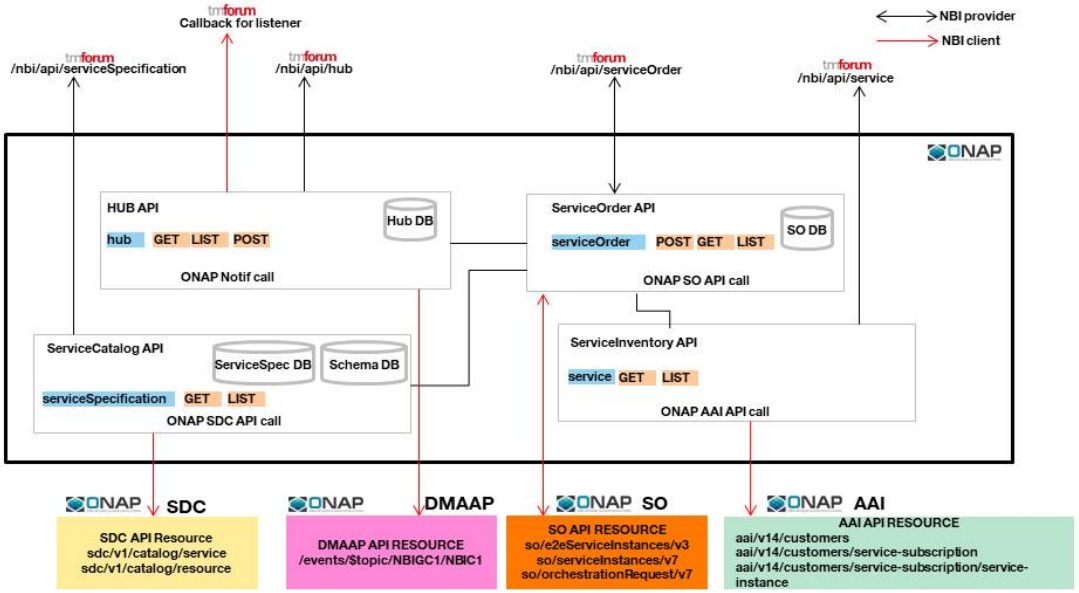


Figure 2.5. ONAP External APIs Architecture

ONAP External APIs provides access to different information stored in ONAP platform components. By means of exposed API methods, BSS applications can get the list of available models from SDC and the information stored in A&AI. Moreover,

BSS can ask for service instantiation, modification or deletion by sending a Service Order request, whose syntax will be described in the next chapter. ONAP NBI will convert this request to ONAP SO request, hiding the sequence of operations performed at lower level. Furthermore, client applications can also receive notifications about the state of their Service Orders thanks to an interface to ONAP DMaaP component. In this way, the user can interact with the ONAP platform in a standard way, without the necessity of technical knowledges about each component and executed processes, making this kind of interaction easier and more intuitive.

Four REST APIs are exposed:

- **ServiceOrder API**, based on the TMF641 specification for Service Orders management. It provides all the methods that allow to retrieve (GET), create (POST), modify (PUT) and delete (DELETE) Service Orders.
- **ServiceCatalog API**, which follows the TMF633 specification (Service Catalog management). This API manages ONAP NBI Service Catalog containing all the service models available in SDC Catalog. It is used by the NBI internally and by the user in order to get the service models list from SDC, without the need to perform direct calls to SDC component.
- **ServiceInventory API**, compliant to the TMF638 specification for Service Inventory management, which provides access to the NBI service Inventory. All the active service instances stored in AAI are copied inside it. Also in this case, the user can retrieve service instances in a transparent way.
- **HUB API**, also based on TMF641, which allows user to subscribe to ONAP NBI notifications in order to check order status and percentage of progress; this mechanism is implemented by subscribing to ONAP DMaaP topics and listening for events on that topics.

In addiction, an API method is available to execute the NBI healthcheck. By means of this request, the user can monitor the status of ONAP NBI and, in particular, if it is connected to SDC, AAI and SO.

The diagram in figure 2.6 shows the workflow executed by the External APIs component when a Service Order request is received.

When ONAP NBI receives a Service Order request, it firstly checks if the order is valid. In particular, if the request body contains the mandatory information and the specified fields contain values compliant to the API specification defined by TM Forum, then order processing can be continued, otherwise an error code is returned.

The second step is order storage. This requires for SO intervention, which has to initialize some attributes values such as order *id* and order *state*, which is usually set to “acknowledged”. Finally, if the order creation ends correctly, a success code is returned.

At this point, the Service Order is decomposed into different blocks. These blocks are called Order Items, which specify the action to be performed on a given service.

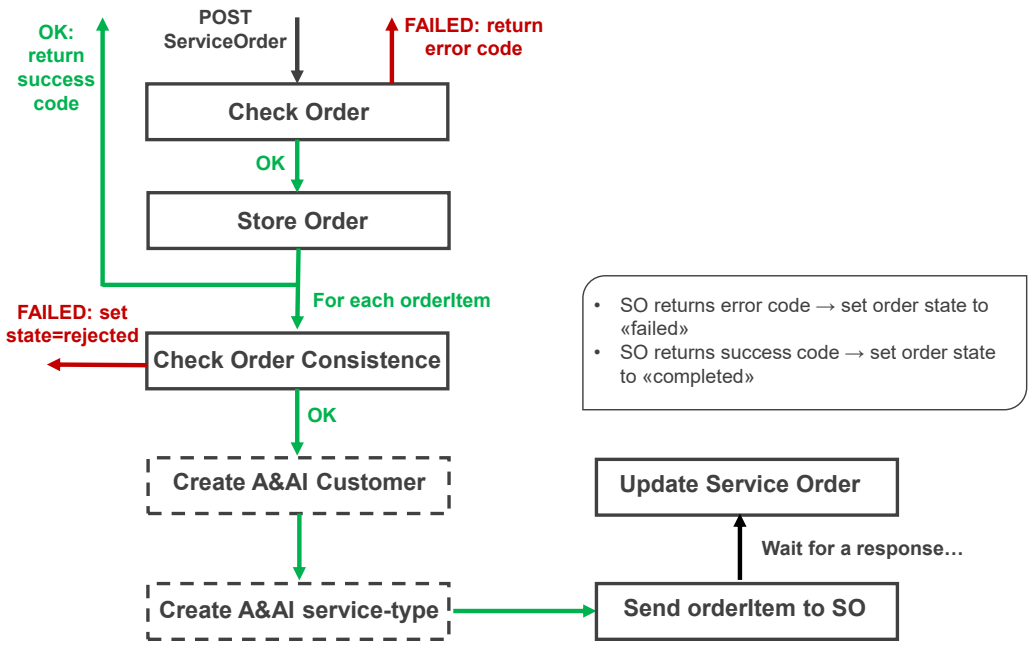


Figure 2.6. ONAP NBI Workflow for Service Order

So, for each order item included in the service order request body, the following sequence of operations is executed:

1. **Check Order Consinstence**, based on the action required on the specified service. In particular, if the action is “add” (i.e., instantiate), the NBI has to check that the service specification exists in the catalog via the ServiceCatalog API, while if it is “delete” it needs to find the service instance related to the provided customer information (if this information is missing, the service instance is searched among those associated to the “generic” customer) in the Inventory by using the ServiceInventory API. In addiction tenant/cloud information is checked. The result of these checks determines if the order item has to be set in “rejected” state and an error code has to be returned or if the processing can go on.

2. **Create A&AI customer** (if not already present), only if action is “add” for all order items.

3. **Create A&AI service type** (if not already present) associated to the customer.

4. **Send order Item to SO**, so the NBI retrieves the information about the service or VNF to be created, deleted or modified, it them to fill the SO request body and sends a POST request via SO API.

5. **Update Service Order**. If the SO returns a success code in the response, then the order item state is set to “completed”, otherwise its value is “rejected”. If all order items have the state set to “completed”, the state of the Service Order is set to the same value, while even if a single order item has been rejected, the Service Order state is changed accordingly.

Service Order management in ONAP NBI supports 2 modes:

- **E2E integration**: NBI calls SO API to apply E2E instantiation method.
- **Service-level only integration**: NBI triggers only SO request at service instance level (not at VNF level and nothing will be created on cloud platform). This corresponds to the A La Carte instantiation method: the service instance is only created in AAI and the objects composing it are not instantiated.

The choice of the mode is done by NBI depending on information retrieved in SDC. If the service specification in the SDC Catalog is within a category “E2E Service”, NBI will use E2E SO API to trigger E2E instantiation, otherwise only API at service instance level will be used. The Macro instantiation method is not yet supported.

To sum up, ONAP External APIs play a primary role from customer perspective. Everyone can develop a client application to NBI API and interact with ONAP components in a transparent, standardized and simple way.

2.3.6 ONAP NBI testing

After having studied ONAP NBI, it was important to explore and evaluate its functionalities by performing some practical tests. Therefore, some requests was sent in order to verify service instantiation and deletion via ONAP External APIs.

In both cases, a Service Order was sent to ONAP NBI by means of a POST request to the Service Order API. In particular, the request body must be filled correctly, specifying the information required to identify the service to be instantiated or deleted. The method used was the A La Carte one.

For what concerns service instantiation request, the first information to be retrieved is the model of the service to be instantiated from ONAP SDC Catalog. A list of all available service models in ONAP SDC is returned by ONAP NBI when sending a GET request to the Service Catalog API, so this was the first test performed. An example of JSON response body is provided below [24].

```
[
  {
    "id": "string",
    "href": "string",
    "name": "string",
    "description": "string",
```

```
"@type": "string",
"@schemaLocation": "string",
"@baseType": "string",
"invariantUUID": "string",
"toscaModelURL": "string",
"toscaResourceName": "string",
"category": "string",
"subcategory": "string",
"distributionStatus": "DISTRIBUTION_NOT_APPROVED",
"version": "string",
"lifecycleStatus": "NOT_CERTIFIED_CHECKOUT",
"targetServiceSchema": {
  "@type": "string",
  "@schemaLocation": "string"
},
"attachment": [
  {
    "id": "string",
    "name": "string",
    "description": "string",
    "@type": "string",
    "artifactLabel": "string",
    "artifactGroupType": "string",
    "artifactTimeout": "string",
    "artifactChecksum": "string",
    "artifactVersion": "string",
    "generatedFromUUID": "string",
    "url": "string",
    "mimeType": "string"
  }
],
"relatedParty": [
  {
    "id": "string",
    "role": "string",
    "name": "string"
  }
],
"resourceSpecification": [
  {
    "id": "string",
    "version": "string",
```

```
        "name": "string",
        "@type": "string",
        "resourceInstanceName": "string",
        "resourceInvariantUUID": "string",
        "resourceType": "string",
        "modelCustomizationName": "string",
        "modelCustomizationId": "string"
      }
    ],
    "serviceSpecCharacteristic": [
      {
        "name": "string",
        "description": "string",
        "valueType": "string",
        "@type": "string",
        "@schemaLocation": "string",
        "required": true,
        "status": "string",
        "serviceSpecCharacteristicValue": [
          {
            "valueType": "string",
            "isDefault": true,
            "value": "string"
          }
        ]
      }
    ]
  }
]
```

Listing 2.1. Response body returned after a GET request to the ServiceCatalog API

At this point, a model was selected among those in the list and the parameters attached to it were used to fill the Service Order request body. After that, the information about customer (i.e., *relatedParty* field) and *orderItem* were inserted. The customer is not mandatory, but if it is not included, the order will be associated to the "generic" one, that had to be previously defined in ONAP A&AI. In order to ask service instance creation, the *action* specified for that orderItem must be "add". An example of request body is provided below, according to the ServiceOrder API documentation [25].

```
{
  "externalId": "{{externalId}}",
  "priority": "1",
  "description": "{{service}} order for generic customer via Postman",
  "category": "Consumer",
  "requestedStartDate": "2018-04-26T08:33:37.299Z",
  "requestedCompletionDate": "2018-04-26T08:33:37.299Z",
  "relatedParty": [
    {
      "id": "{{customer_name}}",
      "role": "ONAPcustomer",
      "name": "{{customer_name}}"
    }
  ],
  "orderItem": [
    {
      "id": "1",
      "action": "add",
      "service": {
        "name": "{{service_instance_name}}",
        "serviceState": "active",
        "serviceSpecification": {
          "id": "{{auto_service_id}}"
        }
      }
    }
  ]
}
```

Listing 2.2. Request body for POST request to the ServiceOrder API

Finally, the POST request was sent to the Service Order API and a success code was returned in the response, with the id of the created order and its state value equal to “acknowledged” in the payload. After a few seconds, a GET request for the created order was sent to the Service Order API in order to check again the order state, that at this point was set to “completed”. An example of response body provided in ServiceOrder API documentation is shown below.

```
{
  "id": "string",
  "href": "string",
```

```
"externalId": "string",
"priority": "string",
"description": "string",
"category": "string",
"state": "acknowledged",
"orderDate": "2020-03-10T10:28:59.199Z",
"completionDateTime": "2020-03-10T10:28:59.199Z",
"requestedStartDate": "2020-03-10T10:28:59.199Z",
"requestedCompletionDate": "2020-03-10T10:28:59.200Z",
"expectedCompletionDate": "2020-03-10T10:28:59.200Z",
"startDate": "2020-03-10T10:28:59.200Z",
"@baseType": "string",
"@type": "string",
"@schemaLocation": "string",
"relatedParty": [
  {
    "id": "string",
    "href": "string",
    "role": "string",
    "name": "string",
    "@referredType": "string"
  }
],
"orderRelationship": [
  {
    "type": "string",
    "id": "string",
    "href": "string",
    "@referredType": "string"
  }
],
"orderItem": [
  {
    "id": "string",
    "action": "add",
    "state": "acknowledged",
    "percentProgress": "string",
    "@type": "string",
    "@schemaLocation": "string",
    "@baseType": "string",
    "orderItemRelationship": [
      {
```

```
      "type": "reliesOn",
      "id": "string"
    }
  ],
  "service": {
    "id": "string",
    "href": "string",
    "name": "string",
    "serviceState": "string",
    "@type": "string",
    "@schemaLocation": "string",
    "serviceCharacteristic": [
      {
        "name": "string",
        "valueType": "string",
        "value": {
          "serviceCharacteristicValue": "string"
        }
      }
    ],
    "serviceRelationship": [
      {
        "type": "reliesOn"
      }
    ],
    "relatedParty": [
      {
        "id": "string",
        "href": "string",
        "role": "string",
        "name": "string",
        "@referredType": "string"
      }
    ],
    "serviceSpecification": {
      "id": "string",
      "href": "string",
      "name": "string",
      "version": "string",
      "targetServiceSchema": {
        "@type": "string",
        "@schemaLocation": "string"
```

```
          },
          "@type": "string",
          "@schemaLocation": "string",
          "@baseType": "string"
        }
      },
      "orderItemMessage": [
        {
          "code": "string",
          "field": "string",
          "messageInformation": "string",
          "severity": "information",
          "correctionRequired": true
        }
      ]
    }
  ],
  "orderMessage": [
    {
      "code": "string",
      "field": "string",
      "messageInformation": "string",
      "severity": "information",
      "correctionRequired": true
    }
  ]
}
```


Listing 2.3. Response body returned after a POST request to the ServiceOrder API

In order to verify that a new service instance was created, a check was performed in ONAP A&AI. In conclusion, the first part of the test was successfully completed.

The second part of the test is about removing an existing service instance. In order to select the service to be deleted, the list of active service instances was obtained by means of a GET request to the Service Inventory API, that takes it from ONAP A&AI. The structure of response body returned to this request is shown below [26].


```
[
  {
    "id": "string",
    "name": "string",
```

```
    "serviceSpecification": {
      "id": "string",
      "name": "string"
    },
    "relatedParty": {
      "id": "string",
      "role": "string"
    }
  }
]
```

Listing 2.4. Response body returned after a GET request to the ServiceInventory API

Once the target service instance has been retrieved in the list, the information attached to it was included in the request body for the Service Order to be sent. Also in this case the customer was specified, while the action field value in the orderItem was set to “delete”. So, the POST request to the Service Order API was sent and order status was checked until it was set to “completed”. The service instance was consequently deleted also from A&AI.

Finally, orders with wrong or missing information in the payload were sent to the Service Order API and a response with an error code was returned. In particular, when sending a Service Order without the relatedParty field, the order failed because the “generic” customer did not exist in ONAP A&AI. In this case, the reason of the failure was retireved by inspecting the log files of ONAP NBI application.

In conclusion, ONAP External APIs offer a simpler way to trigger actions on services, that is very useful for BSS applictions. On the other hand, it has some limitations:

- The Macro instantiation method is not supported
- If an order fails, details are not provided and in order to understand what went wrong log files of the application have to be inspected
- Few documented use cases with respect to other ONAP components
- Documentation can be enriched for users that are not very familiar to ONAP

Chapter 3

TM Forum and the Open API Program

3.1 Overview

TM Forum stands for TeleManagement Forum. It is a global industry association born to help communication service providers to maximize their business success. In particular, the purpose of the organization is to lead the rapid digital evolution process in which communications service providers and suppliers are involved. This is done by providing an open environment and a practical support to allow them to adapt their business operations and systems to the needs introduced by the transormation affecting them. As a non-profit member organization, TM Forum represents over 850 companies generating $2 trillion in revenue and serving five billion customers across 180 countries.

TM Forum's Open API Program is a global initiative to provide end-to-end connectivity, interoperability and portability across complex services. It offers a suite of standard REST based APIs enabling rapid, repeatable, and flexible integration among operations and management systems and simplifying creation, build and operation of complex innovative services. These Open APIs are technology agnostic, thus can be used to implement any kind of digital service, including those appertaining to categories such as Internet of Things, Smart Health, Big Data, NFV or Next Generation OSS and BSS applications.

Among the several benefits brought by this initiative, there are:

- **Improvement of business and IT agility**: Open APIs enable a modular platform architecture, allowing companies to abstract its complexity and to define a set of capabilities which can be used to cut time to market for new services and the costs to create and operate them.

- **Reduced cost and complexity of operations**: providers need to reduce costs by simplifying operations and creating harmony across different geographic

regions in the world.

- **Enabling Global Connectivity**: currently, the existing wholesale ecosystem results to be inefficient with respect to new NFV and SDN based infrastructures, since it requires to be manually handled. Operators can exploit Open APIs to implement new IT systems able to radically improve and simplify provisioning, to provide flexibility and to ensure scalability of their services.
- **Reduced integration cost, risk and time**: since providers are constantly under pressure to rapidly improve their service efficiency, the costs and risks associated with integration processes need to be reduced. TM Forum Open APIs represent a solution to reduce time and risk of transforming business operations and performing eventual future changes.
- **Providing a global platform for telco-based innovation**: the geographic restrictions affecting most of telecommunications services limits the ability of CSPs to attract software developers. By exposing a set common APIs, cooperation between operators and developers communities can be simplified.

For these reasons, the Open APIs developed by TM Forum represent a powerful toolset that can be used to establish a coherent integration architecture combining management capabilities needed to operate and maintain complex digital services at global scale. Service providers need not only to ensure software performance, huge set of functionalities or effectiveness of their services, but they also have to deal with the full range of business support capabilities required for commercial service operation, such as customer management, product management, accounting, billing, monitoring, infrastructure planning, design and ordering. In this context, the above mentioned APIs offer a solution which has interoperability, integration and standardization as key concepts and which avoids providers to become locked into vendor specific approaches.

Among the TM Forum's standard APIs, three of them are relevant for this thesis work and, therefore, will be descripted in detail.

3.2 TMF641: Service Ordering Management API

The TMF REST API for Service Order Management [27] provides a standardized mechanism for placing a Service Order, a type of order which allows to ask for service instantiation, deletion or modification, used between a customer and a service provider or between a service provider and a partner or vice versa. In particular, this API allows to create, update and retrieve Service Orders, including filtering operations, and manages related notifications.

A Service Order can be referred to two different categories of services: Customer Facing Service (CFS) and to a Resource Facing Service (RFS). Resource Facing Services are services internal to the service provider, which are indirectly part of a

product and are invisible to the customer. They corresponds to the Resource concept defined in eTOM standard, therefore are placed at Resource layer, and exist to support one or more Customer Facing Services. A Customer Facing Service represents the realization of product within an organization's infrastructure, thus it represent the service exposed to customers and which will be delivered to them. It is placed at Service layer according to the eTOM model. In particular, a CFS can be composed by one or more RFS needed to realise it and provides functionalities at the boundary of the provider's infrastructure in a protocol-agnostic way.

From a component perspective, a Service Order should be available from:

- A Service Orchestrator, with the possibility to be referred both to CFS and RFS
- An Infrastructure Manager, with the possibility to be referred only to RFS

The API resource model[1] is represented in figure 3.1.

The *ServiceOrder* resource refers to a type of order which can be used to describe a group of operations on services by means of a list of service order items, with one *orderItem* per service. A service order item references an *action* on an existing or new service, which describes the operation to be done on the service, such as add, change or delete. The service order is sent from the BSS system in charge of the product order management to the SOM system that will manage the request fulfillment.

The *ServiceOrderItem* sub-resource can be associated to only one *ServiceRestriction* sub-resource, which is a data structure that captures the configuration to apply to an existing subscribed service or to a new one, depending on the action requested on the given service. This sub-resource has a relationship with other ones, including:

- The *RelatedParty* sub-resource, which is a reference to a party involved in the order (e.g., final customer).
- *Characteristic*, which describes a given characteristic of an object or entity through a name-value pair.
- *Place*, which defines the place where the products are sold or delivered.

The *ServiceSpecificationRef* relationship points to the service specifications required to realize a Product Specification, while the *ResourceRef* one points to the supporting resources composing that service.

The table in figure 3.2 provides Service Order *state* and service Order Item state fields description.

For what concerns the operations that can be performed on a Service Order, they are:

[1] A resource model is a model describing the resource hierarchy modeled by the API it is referred to.

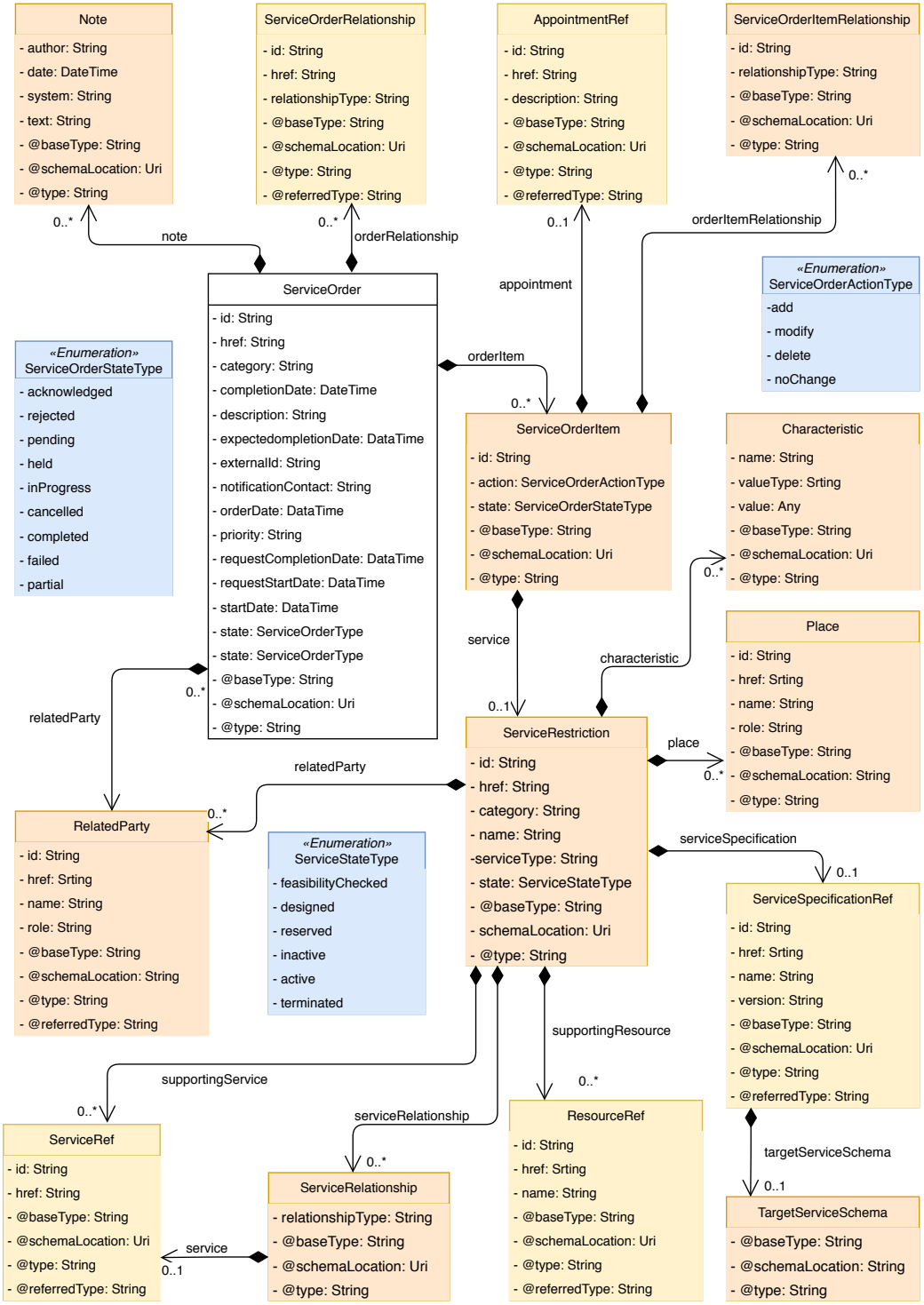


Figure 3.1. TMF641 Resource Model

- Retrieval of a Service Order or a collection of Service Orders depending on filter criteria
- Partial update of a Service Order
- Creation of a Service Order
- Deletion of Service Order (for administration purposes)
- Notification of events on sent Service Orders

According to the API documentation, the corresponding methods are listed in the table in figure 3.3.

Acknowledged	The Acknowledged state is where an order has been received and has passed message and basic business validations.
In Progress	The In Progress state is when service delivery has started.
Cancelled	The Cancelled state is where an In-Flight Order has been successfully cancelled.
Completed	The Completed state is where an order has complete provision and the service is now active.
Rejected	The Rejected state is where:

Figure 3.2. TMF641 state field values

ONAP NBI ServiceOrder API has been implemented following this specification.

3.3 TMF638: Service Inventory Management API

The Service Inventory API [28] provides standardized mechanism for Service Inventory management such as creation, partial or full update and retrieval of representation of services from the Inventory, which contains a list of service instances. It allows also notification of events related to service lifecycle. The API resource model is shown in figure 3.4.

Service is a base class for defining the Service hierarchy. All Services are characterized as either being possibly visible and usable by a Customer or not. This gives rise to the two sub-classes of Service: CustomerFacingService and ResourceFacingService, that has to be specified in the *category* field of the Service resource. Most of sub-resources correspond to the ones described in the TMF641 resource model (e.g., Characteristic, RelatedParty).
The table in figure 3.5 lists the API available methods. The operations that can be executed on service instances stored in the Inventory are:

- Retrieve a service or a collection of services depending on filter criteria

	Method	URI	Description
serviceOrder	GET	/serviceOrder	List or find ServiceOrder objects
	POST	/serviceOrder	Creates a ServiceOrder
	GET	/serviceOrder/{id}	Retrieves a ServiceOrder by ID
	PATCH	/serviceOrder/{id}	Updates partially a ServiceOrder
	DELETE	/serviceOrder/{id}	Deletes a ServiceOrder
Notification listeners (client side)	POST	/listener/serviceOrder-CreateNotification	Client listener for entity ServiceOrderCreate-Notification
	POST	/listener/serviceOrder-AttributeValueChange-Notification	Client listener for entity ServiceOrderAttribute-ValueChangeNotification
	POST	/listener/serviceOrder-StateChangeNotification	Client listener for entity ServiceOrderStateChange-Notification
	POST	/listener/serviceOrder-DeleteNotification	Client listener for entity ServiceOrderDelete-Notification
Events subscription	POST	/hub	Register a listener
	DELETE	/hub/{id}	Unregister a listener

Figure 3.3. TMF641 methods definition

- Partial update of a service
- Create a service (for administration users only)
- Delete a service (for administration users only)
- Notification of events on service

An implementation of this API is provided by the ServiceInventory API of ONAP NBI component.

3.4 TMF633: Service Catalog Management API

The Service Catalog Management API [29] provides a standardized solution for adding Service Specifications to an existing Catalog, which contains a list of service models. This API allows the management of the entire lifecycle of the Service Catalog elements and their consultation during several processes such as ordering process. The API resource model is shown in figure 3.6.

The *ServiceCatalog* resource is the root entity for Service Catalog management. A Service Catalog is a group of Service Specifications made available through service candidates that an organization provides to the consumers. A Service Specification

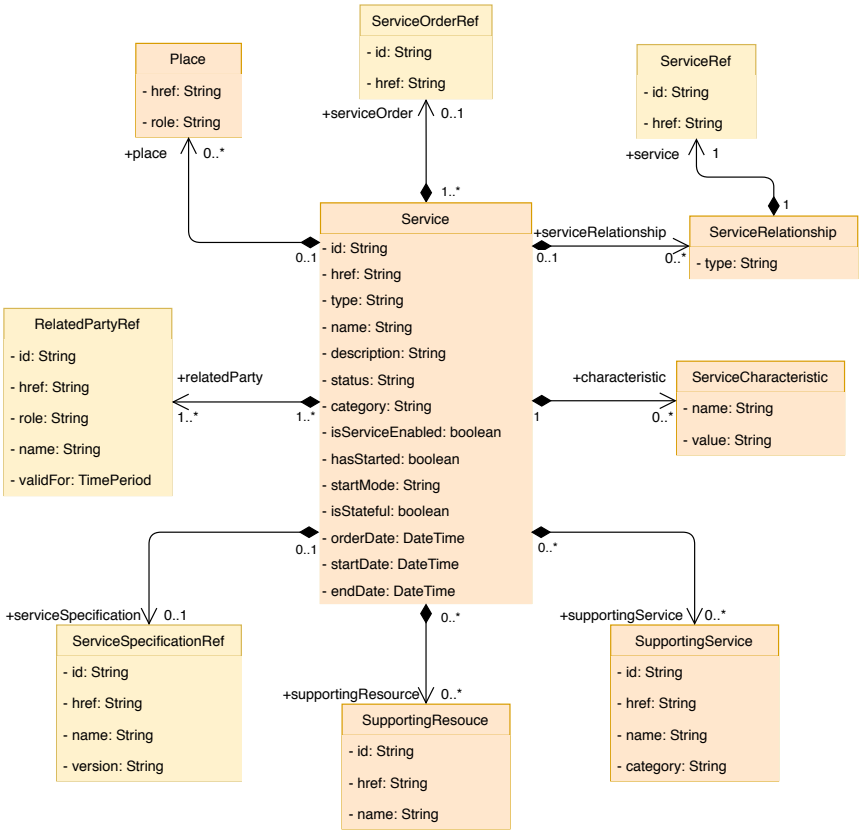


Figure 3.4. TMF638 Resource Model

represents a service model defined following the TMF633 standard. A Service Catalog typically includes name, description and time period that is valid for. It will have a list of *ServiceCandidate* catalog items. A ServiceCandidate is an entity that makes a *ServiceSpecification* available to a catalog. A ServiceCandidate and its associated ServiceSpecification may be “published” (i.e., made visible) in any number of ServiceCatalogs, or in none.
Service Catalog API allows to perform the following operations on resources:

- Retrieve an entity or a collection of entities depending on filter criteria
- Partial update of an entity
- Create an entity
- Delete an entity (for administration purposes)
- Manage notification of events

	Method	URI	Description
service	GET	/service	List or find Service objects
	POST	/service	Creates a Service
	GET	/service/{id}	Retrieves a Service by ID
	PATCH	/service/{id}	Updates partially a Service
	DELETE	/service/{id}	Deletes a Service
Notification listeners (client side)	POST	/listener/serviceCreate-Notification	Client listener for entity ServiceCreateNotification
	POST	/listener/serviceAttribute-ValueChangeNotification	Client listener for entity ServiceAttributeValue-ChangeNotification
	POST	/listener/serviceState-ChangeNotification	Client listener for entity ServiceStateChange-Notification
	POST	/listener/serviceBatch-Notification	Client listener for entity ServiceBatchNotification
	POST	/listener/serviceDelete-Notification	Client listener for entity ServiceDeleteNotification
Events subscription	POST	/hub	Register a listener
	DELETE	/hub/{id}	Unregister a listener

Figure 3.5. TMF638 methods definition

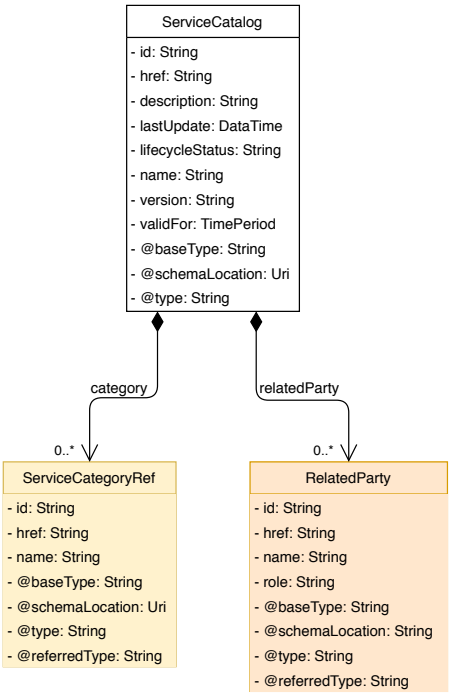


Figure 3.6. TMF633 Resource Model

A subset of API methods, the more interesting ones for this thesis work, are described in table in figure 3.7.
The ServiceCatalog API implemented in ONAP NBI module follows this specification.

	Method	URI	Description
serviceCatalog	GET	/serviceCatalog	List or find ServiceCatalog objects
	POST	/serviceCatalog	Creates a ServiceCatalog
	GET	/serviceCatalog/{id}	Retrieves a ServiceCatalog by ID
	PATCH	/serviceCatalog/{id}	Updates partially a ServiceCatalog
	DELETE	/serviceCatalog/{id}	Deletes a ServiceCatalog
serviceCategory	GET	/serviceCategory	List or find ServiceCategory objects
	POST	/serviceCategory	Creates a ServiceCategory
	GET	/serviceCategory/{id}	Retrieves a ServiceCategory by ID
	PATCH	/serviceCategory/{id}	Updates partially a ServiceCategory
	DELETE	/serviceCategory/{id}	Deletes a ServiceCategory
serviceCandidate	GET	/serviceCandidate	List or find ServiceCandidate objects
	POST	/serviceCandidate	Creates a ServiceCandidate
	GET	/serviceCandidate/{id}	Retrieves a ServiceCandidate by ID
	PATCH	/serviceCandidate/{id}	Updates partially a ServiceCandidate
	DELETE	/serviceCandidate/{id}	Deletes a ServiceCandidate
serviceSpecification	GET	/serviceSpecification	List or find ServiceSpecification objects
	POST	/serviceSpecification	Creates a ServiceSpecification
	GET	/serviceSpecification/{id}	Retrieves a ServiceSpecification by ID
	PATCH	/serviceSpecification/{id}	Updates partially a ServiceSpecification
	DELETE	/serviceSpecification/{id}	Deletes a ServiceSpecification
Notification listeners (client side)	POST	/listener/service-SpecificationCreate-Notification	Client listener for entity ServiceSpecificationCreate-Notification
	POST	/listener/service-SpecificationChange-Notification	Client listener for entity ServiceSpecification-ChangeNotification
	POST	/listener/service-SpecificationDelete-Notification	Client listener for entity ServiceSpecification-DeleteNotification
	POST	/listener/service-CandidateCreate-	Client listener for entity ServiceCandidateCreate-

		Notification	Notification
	POST	/listener/service-CandidateChange-Notification	Client listener for entity ServiceCandidate-ChangeNotification
	POST	/listener/service-CandidateDelete-Notification	Client listener for entity ServiceCandidate-DeleteNotification
	POST	/listener/serviceCategory-CreateNotification	Client listener for entity ServiceCategoryCreate-Notification
	POST	/listener/service-CategoryChange-Notification	Client listener for entity ServiceCategory-ChangeNotification
	POST	/listener/serviceCategory-DeleteNotification	Client listener for entity ServiceCategory-DeleteNotification
	POST	/listener/serviceCatalog-CreateNotification	Client listener for entity ServiceCatalogCreate-Notification
	POST	/listener/serviceCatalog-ChangeNotification	Client listener for entity ServiceCatalog-ChangeNotification
	POST	/listener/serviceCatalog-BatchNotification	Client listener for entity ServiceCatalogBatch-Notification
	POST	/listener/serviceCatalog-DeleteNotification	Client listener for entity ServiceCatalogDelete-Notification
Events subscription	POST	/hub	Register a listener
	DELETE	/hub/{id}	Unregister a listener

Figure 3.7. TMF633 methods definition

Chapter 4

Service Resolver for ONAP

4.1 Overview

Service Resolver [30] is a Service Order Management (SOM) application, according to TM Forum terminology, developed by Orange. It processes *CFS Orders*, that are Service Orders for Customer Facing Services, containing *orderItems* with "add" or "delete" action for the CFS they are referred to. Its code is open source and, as a prototype, it is usable only for demo purposes, not for production.

Service Resolver is placed at intermediate level between BSS applications and ONAP External APIs module, therefore at eTOM Service layer. When a CFS Order is received from a BSS system, it is decomposed into the corresponding set of RFS needed to create it and a Service Order request is sent to ONAP External APIs for each identified RFS. For this reason, Service Resolver addresses a need that ONAP is not able to satisfy: the ability to select autonomously the RFS to be instantiated for a CFS with characteristics specified by the customer. For example, to provide a Firewall service, we need a Firewall software solution described by a RFS model or specification. There can be multiple firewall software solutions from various vendors or open source communities. How the best solution for the required service can be selected? ONAP is not able to take decision on resources needed to create a particular service, because a service can be composed by different combinations of resources and ONAP has not any criterion to choose the best one satisfying customer needs.

Service Resolver has very interesting characteristics which can be resumed as follows:

- It is **generic**: it allows to order any kind of service, for example virtual Firewalls, Internet Access, Network Slice services or monitoring services
- It is **model-driven**: processing operations are not performed with respect to the specific service, but they are simply based on a service definitions, which are standard for every kind of service

- It is **adaptative** to the context : the “solution” needed to satisfy client request is elaborated onlywhen a new order is received, by applying some rules in a dynamical way
- Its implementation is **based on standard APIs**, in particular the TMF641, TMF633 and TMF638
- It is light: it is deployed on a few small Docker containers.
- It can be runned in **simulation mode** (without ONAP) to allow Service Designer to test service definitions before performing a real deployment

Service Resolver architecture and functionalities will be described in the next section.

4.2 Architecture

Service Resolver application has been entirely developed in Python, using the Connexion[1] framework, and it exposes six REST APIs:

- RFS Specification API
- CFS Specification API
- Service (CFS and RFS) Instance API
- CFS Order API
- Instantiation Decision API
- Instantiation Rules Inventory API, which is a “work in progress” feature, not yet used.

The RFS Specification API is used to insert new RFS Specifications (i.e., models) in the Service Resolver RFS Catalog and to get the list of available ones. In the same way, the CFS Specification API provides access to the CFS Catalog in Service Resolver database, such that CFS Specifications can be added or retireved. A CFS Specification is a set of data that describes the various characteristics and “solution” that can potentially be used to instantiate the service. In other words, a CFS can be defined thorugh a model which specifies the set of RFS that can compose it, a list of characteristics that the CFS can have (i.e., that the customer can require) and the rules allowing to select the correct RFS to be instantiated to satisfy each defined characteristic. Those APIs are based on TMF633 standard.

[1] Connexion [31] is a Python framework that allows to handle HTTP requests defined using OpenAPI specification [32].

The Service Instance API manages Service Resolver Service Inventory, which contains all created instances of both CFS and RFS, according to the TMF638 specification.

The CFS Order API is based on the TMF641 specification and it is responsible for CFS Order Management. In particular, it receives CFS Orders and translates them in the corresponding set of RFS Orders to be sent to ONAP External APIs in order to instantiate or delete the CFS. Moreover, it is possible to perform GET requests in order to retrieve the list of created orders or only a particular one if the *id* is specified as parameter in the URL.

When a CFS Order for a new CFS instance creation is received, a decision about the set of RFS to be instantiated to fulfill the request has to be taken. The Instantiation decision API is responsible for doing exactly that: it evaluates the set of RFS composing the CFS to be instantiated and returns a solution containing the list of selected RFS to the CFS Order API.

Figure 4.1 provides a component view of Service Resolver and shows high level interactions among the different micro-services that compose it.

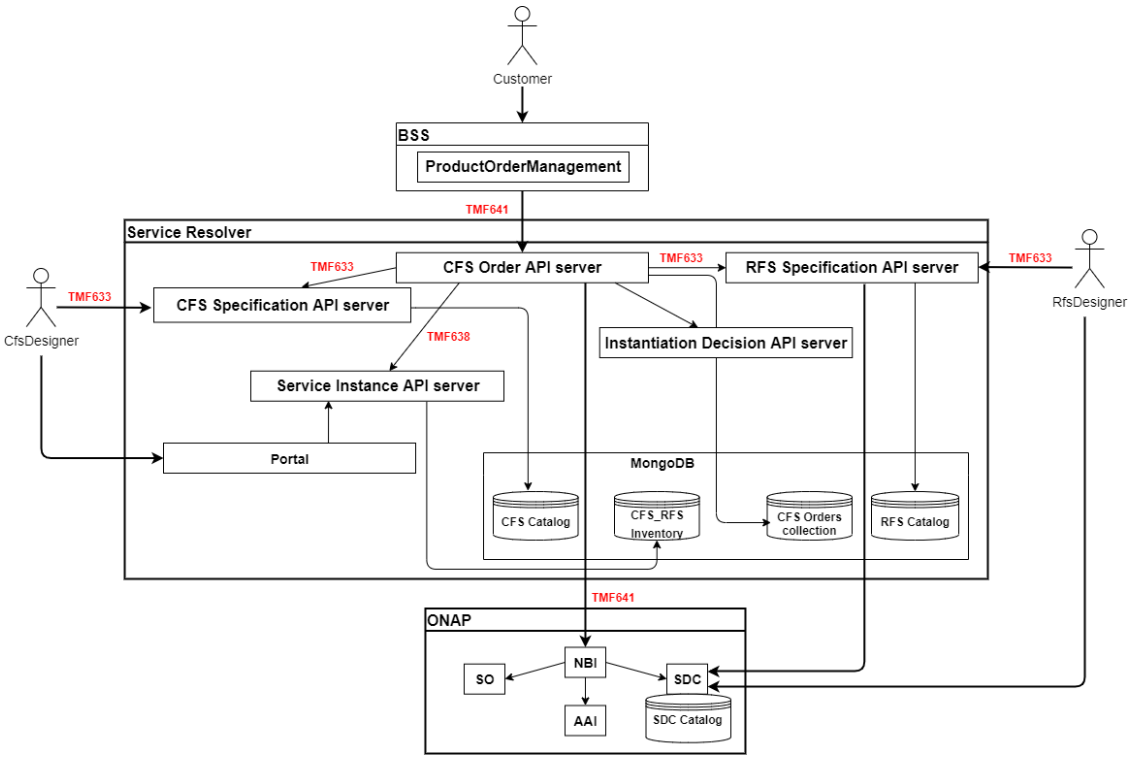


Figure 4.1. Service Resolver component view

The diagram includes three actors: the customer which sends the CFS Order through a BSS application, the RFS designer and the CFS designer. In particular, the RFS designer is responsible for designing services at Resource level, so he defines

them in ONAP SDC Catalog. Once those models have been distributed to ONAP run-time components, the RFS designer can insert them into Service Resolver RFS Catalog and the CFS designer can use them to compose the CFS models that will be made available for instantiation requests.

As shown in figure 4.1, the API servers composing Service Resolver interact among them in order to perform the required processing operations. This kind of interactions will be detailed in the section 4.4.

In addition, a "demo portal" is also part of the application in order to allow CFS designers to test if service instance composition is correct after perfoming some CFS Orders. Figure 4.2 provides an example of the relationships displayed by the portal. Finally, a common MongoDB database[2] is used to store all information.

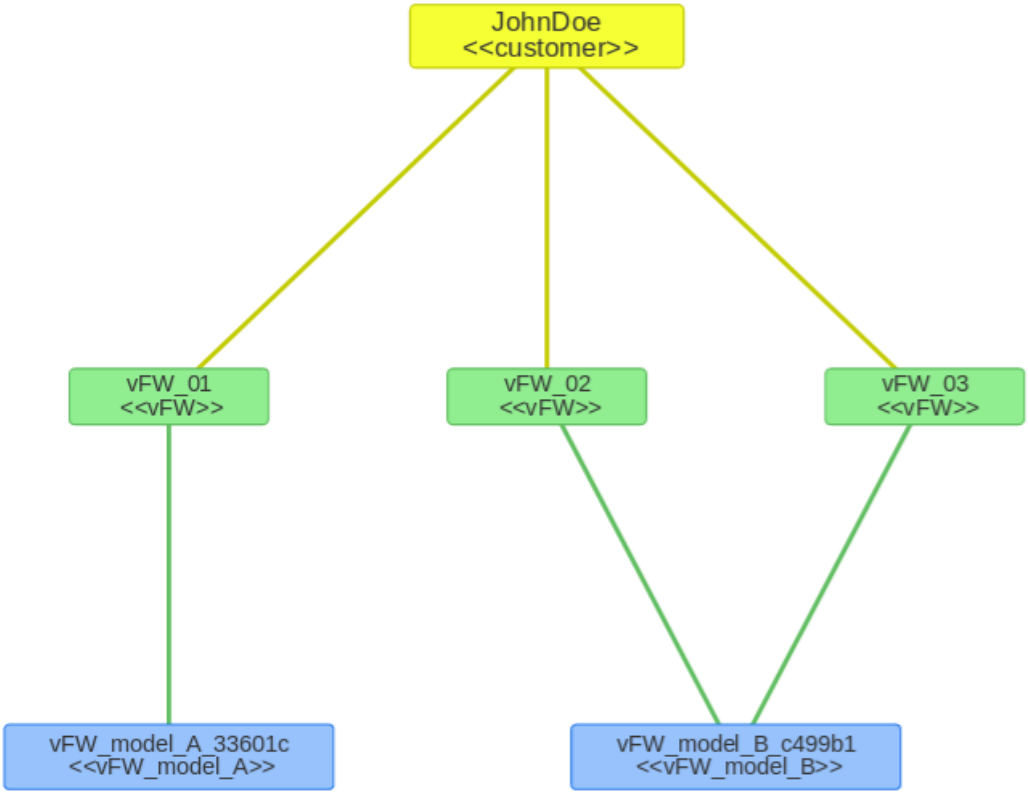


Figure 4.2. Service Resolver demo portal

The next section explains more about the REST APIs exposed by Service Resolver.

[2]MongoDB is a NoSQL document-based database, used to provide data persistence for distributed applications and particularly suitable for Cloud-based environments [33].

4.3 Service Resolver REST APIs

4.3.1 RFS Specification API

This API provides methods to manage the RFS Catalog, a collection of RFS Specification stored in Service Resolver database. A RFS Specification, or RFS model, is a set of information that describe a service usually defined ONAP SDC. The RFS model data structure will then be send via a POST message to `http://api_rfs_spec_server:6001/serviceResolver/serviceCatalogManagement/api/v1/rfsSpecification`, the URL to reach the RFS specification API server. The value of "api_rfs_spec_server" as well as all server names inside the URL must be set to "localhost" if Service Resolver is tested locally or to the IP address of the machine on which it is installed. The available request methods are:

- POST, used to insert a new RFS model
- GET, which allows to retrieve the list of RFS Specifications from the RFS Catalog or a specific model by *id*
- PUT, used to update an existing RFS model
- DELETE, which deletes a given RFS Specification by *id*

If using Service Resolver in simulation mode, you need to provide fake data on used ONAP platform but it will not be used. You do not need to provide any *id* for the new RFS Specification since it will be generated by Service Resolver itself. An example of request body is shown below.

```
{
  "name": "vFW_model_A",
  "supportingResource": [
    {"name": "onap_1",
    "id": "001",
    "role": [
          {
          "description": "NFV-Orchestrator",
          "id": "001",
          "name": "NFV-Orchestrator"
          }
        ]
    }
  ]
}
```

Listing 4.1. POST RFS Specification request body example (simulation mode)

If using Service Resolver connected with some ONAP platforms, it is necessary to add the RFS Specification *id*, the ONAP platform that will perform the instantiation and the Cloud platform where RFS will be instantiated. The RFS Specification id value have to match with a service model UUID[3] from ONAP SDC Catalog, otherwise Service Resolver will not be able to get TOSCA file for the specified model. The response will contain the full RFS definition with all informations coming from ONAP. An example of request body is below.

```
{
  "name": "vFW_model_A",
  "id": "{{auto_vFW_model_A_spec_id}}",
  "supportingResource": [
    {"name": "onap_1",
    "id": "001",
    "role": [
          {
          "description": "NFV-Orchestrator",
          "id": "001",
          "name": "NFV-Orchestrator"
          }
        ]
    }
  ],
    "cloud": {
          "lcpCloudRegionId": "{{onap_cloud_region_id}}",
          "tenantId": "{{tenant_id}}",
          "cloudOwner": "{{cloud_owner_name}}"
    }
}
}
```

Listing 4.2. POST RFS Specification request body example (with connection to ONAP)

4.3.2 CFS specification API

This API exposes the same methods than the previous one, but in this case CFS Specifications in the CFS Catalog are managed. A CFS Specification, or CFS model, is a set of information that describe a service usually defined from customer point

[3]UUID stands for *Universally Unique Identifier* and it is intended as an alphanumeric string which uniquely identifies a service model inside the SDC Catalog.

of view. The CFS model data structure will then be send via a POST message to Service Resolver using the CFS Specification API. The URL to reach it locally or not is `http://api_service_spec_server:6004/serviceResolver/serviceCatalogManagement/api/v1/serviceSpecification`. This data structure contains:

- The *name* and a *description* of the CFS it is referred to
- A list of key-value pairs coming from TMF633 standard
- A *serviceSpecCharacteristic* array describing the characteristics of the CFS
- A *serviceSpecRelationship* array with all the RFS models that can compose the CFS

For each element in the serviceSpecRelationship array, there is:

- The *name* and the *id* of the RFS model, which must match with existing RFS Specification from RFS Catalog and consequently also from ONAP SDC Catalog
- A *Relationship type*
- A *description*
- An *instantiationPriority* with a value that will allow Service Resolver to know how sequencing the instantiation of RFS
- An *instantiationDecisionRules* array, which contains the rule to be applied to select the list of RFS to be instantiated

An example of request body for creating a new CFS Specification is shown below.

```
{
    "description": "vFW use-case",
    "name": "vFW",
    "version": "1.0",
    "category": "cfs",
    "href": "",
    "lastUpdate": "",
    "lifecycleStatus": "",
    "@type": "",
    "@baseType": "",
    "validFor": {
      "startDateTime": "",
     "endDateTme": ""
    },
    "serviceSpecCharacteristic": [
```

```
        {
            "description": "",
            "name": "featureLevel",
            "valueType": "string",
            "possible_values": ["simple", "rich"],
            "mandatory_characteristic": true
        }
    ],
    "serviceSpecRelationship": [
        {
            "id": "{{auto_vFW_model_A_spec_id}}",
            "name": "vFW_model_A",
            "relationshipType": "cfs_to_rfs",
            "instantiationPriority": 1,
            "description": "that rfs is instantiated by default",
            "instantiationDecisionRules": [
            {
                "ruleEvaluationPriority": 1,
                "rule_name": "param_in_cfs_order_rule",
                "rule_params": {
                    "cfs_param_name": "featureLevel",
                    "operator": "equal",
                    "target_value": "simple"
                },
                "rule_results": [{
                        "rule_response": "no_param",
                        "decision": "instantiate",
                        "is_definitive_decision": true
                    },
                    {
                        "rule_response": "true",
                        "decision": "instantiate",
                        "is_definitive_decision": true
                    },
                    {
                        "rule_response": "false",
                        "decision": "do_not_instantiate",
                        "is_definitive_decision": true
                    }
                ]
            }
            ]
```

```
},
{
    "id": "{{auto_vFW_model_B_spec_id}}",
    "name": "vFW_model_B",
    "relationshipType": "cfs_to_rfs",
    "instantiationPriority": 1,
    "description": "instantiated only if featureLevel is rich",
    "instantiationDecisionRules": [
    {
        "ruleEvaluationPriority": 1,
        "rule_name": "param_in_cfs_order_rule",
        "rule_params": {
            "cfs_param_name": "featureLevel",
            "operator": "equal",
            "target_value": "rich"
        },
        "rule_results": [{
                "rule_response": "no_param",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            },
            {
                "rule_response": "true",
                "decision": "instantiate",
                "is_definitive_decision": false,
                "next_instantiationDecisionRules": 2
            },
            {
                "rule_response": "false",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            }
        ]
    },
        {
            "ruleEvaluationPriority": 2,
            "rule_name": "rfs_instance_capacity_rule",
            "rule_params": {
                "spec_name": "vFW_model_B",
                "set_id_key": "all",
                "capacity": 2
            },
```

```
                    "rule_results": [{
                            "rule_response": "true",
                            "decision": "do_not_instantiate",
                            "is_definitive_decision": true
                        },
                        {
                            "rule_response": "false",
                            "decision": "instantiate",
                            "is_definitive_decision": true
                        }
                    ]
                }

            ]
        }
    ]
}
```

Listing 4.3. POST CFS Specification request body example

The instantiation decision rules are specified by CFS designer in order to provide a criterion to select the desired combination of RFS to be instantiated for the CFS ordered by the customer, based on the characteristics he requires for the CFS. In particular, Service Resolver proposes three rules:

- **param_in_cfs_order_rule**: the RFS will be instantiated if a parameter is specified in the order with a specific value
- **rfs_instance_in_a_set_rule**: the RFS will be instantiated if a RFS model is not already instantiated in a "set". In this context, a set can be intended as the list of service instances attached to a customer or the list of all instances in the Service Resolver Inventory
- **rfs_instance_capacity_rule**: the RFS will be instantiated if a RFS model is already instantiated and has no longer capacity available to support other services.

Using param_in_cfs_order_rule, the following three informations have to be provided: *cfs_param_name*, *operator* and *target_value*. The operator attribute can have one value among equal, greater, less or not_equal. An example of how this rule can be specified is provided below.

```
"rule_params": {
        "cfs_param_name": "featureLevel",
```

```
        "operator": "equal",
        "target_value": "rich"
    }
```

Listing 4.4. param_in_cfs_order_rule example

Using rfs_instance_in_a_set_rule, the information which has to be specified includes the *spec_name*, which must be the name of a RFS Specification, and *set_id_key*, which can have a value corresponding to "customer" or "all". "Customer" value means that Service Resolver will search about RFS instances in the Service Inventory database but only those attached to the customer name provided in the order, while "all" value means that Service Resolver will search about all RFS instances in Service Inventory. An example of the described rule is provided below.

```
"rule_params": {
        "spec_name": "vFW_model_A",
        "set_id_key": "all"
    }
```

Listing 4.5. rfs_instance_in_a_set_rule example

Using rfs_instance_capacity_rule, the following three information are needed: *spec_name*, *set_id_key* and *capacity*. The spec_name and set_id_key fields have the same meaning than in the previous rule, while the capacity one value must be the maximum number of services that can support the RFS, measured as the length of the *serviceRelationship* field in the RFS instance stored in Service Resolver Inventory. An example is shown in the following JSON.

```
"rule_params": {
        "spec_name": "vFW_model_B",
        "set_id_key": "all",
        "capacity": 2
    }
```

Listing 4.6. rfs_instance_capacity_rule example

Those rules can be chained inside a single CFS Specification using the following fields:

- *ruleEvaluationPriority*, needed to sequence the rule evluations for a RFS
- *is_definitive_decision* to prevent from trying to evaluate a next rule
- *next_instantiationDecisionRules* to force rule sequencing

4.3.3 Service Instance API

This API is used to manage the Service Inventory collection in Service Resolver database. It contains both CFS and RFS instances, which are related among them. The base URL to reach the API is `http://api_service_server:6003/serviceResolver/serviceInventoryManagement/api/v1/`.
The methods provided are:

- POST, in order to insert a new service instance
- PUT, which allows to update an existing instance
- GET, used to retrieve a specific instance by *id* or the list of stored service instances
- DELETE, which deletes an existing instance

Among the listed methods, only the GET one should be called by the user, while the other ones are used by the API servers of Service Resolver in order to perform all the needed operations when a Service Order request is received, or can be used for administration purposes.
An example of response body to a GET request is shown below.


```
[
    {
      "@baseType": "Service",
      "@schemaLocation": "",
      "@type": "Service",
      "category": "cfs",
      "href": "/service/c51a00fd-7ab1-42d5-9796-bb3870a50abe",
      "id": "c51a00fd-7ab1-42d5-9796-bb3870a50abe",
      "name": "vFW",
      "place": [
       {
        "id": "",
        "name": "",
        "role": ""
       }
      ],
      "relatedParty": [
       {
        "href": "",
        "id": "JohnDoe",
        "name": "JohnDoe",
        "role": "customer"
```

```
        }
       ],
       "serviceCharacteristic": [
        {
         "name": "featureLevel",
         "value": "rich"
        }
       ],
       "serviceOrder": [
        {
         "action": "add",
         "href": "/cfsOrder/5382e453-f48e-4eee-9ac2-d0aa5d8e5657",
         "id": "5382e453-f48e-4eee-9ac2-d0aa5d8e5657",
         "orderDate": "2019-11-13T09:24:56.448Z"
        }
       ],
       "serviceRelationship": [],
       "serviceSpecification": {
         "href": "/cfsSpecification/08820c2c-a35d-420b-8f6e-93e5c290a1c4
            ",
         "id": "08820c2c-a35d-420b-8f6e-93e5c290a1c4",
         "name": "vFW",
         "version": "1"
        },
        "serviceState": "active",
        "serviceType": "",
        "supportingService": []
    }
]
```


Listing 4.7. Response body example after a GET request to Service instance API

In particular, the instance id of RFS is provided by ONAP, while the id of CFS is generated by Service Resolver. Moreover, the relationships among CFS and RFS composing them are displayed for each instance.

4.3.4 CFS Order API

As described before, this API allows to create and consult CFS Orders. According to the TMF641 specification, a Service Order is a message containing a set of information about an order to add or delete an *orderItem*, which usually describes a customer service. The API implements the following operations:

- POST, for CFS Order creation

- GET, which provides access to the list of orders stored in Service Resolver database
- DELETE, used to delete a CFS Order by *id*
- PUT, to update an existing order by *id*

The CFS Order data structure will then be send via a POST message to Service Resolver through this API, reachable at URL `http://api_service_order_server:6002/serviceResolver/serviceOrderManagement/api/v1/cfsOrder`. This data structure will describe the orderItem, the related customer (if any), the service parameter values and the action to be performed on the service, which can be add or delete.

In case of “add” action, it is necessary to indicate the CFS model to be used to create the CFS instance, while in case of “delete” action, it is necessary to indicate the id of the CFS instance to be deleted. An example of request body for adding a new CFS instance is provided below.

```
{
 "category": "vFW service",
 "description": "service order for vFW customer service",
 "externalId": "BSS_order_01",
  "note": {
   "author": "Orange",
   "date": "May 2019",
   "text": ""
 },
 "notificationContact": "",
 "orderItem": [
   {
     "action": "add",
     "id": "1",
     "service": {
       "name": "vFW_01",
       "place": [
         {
           "id": "",
           "name": "",
           "role": ""
         }
       ],
       "relatedParty": [
         {
           "id": "JohnDoe",
```

```
            "name": "JohnDoe",
            "role": "customer",
            "href": ""
          }
        ],
        "serviceCharacteristic": [
          {
            "name": "featureLevel",
            "value": "simple"
          }
        ],
        "serviceSpecification": {
          "id": "{{auto_vFW_cfs_spec_id}}",
          "name": "vFW",
          "version": "1"
        },
        "serviceState": "active",
        "serviceType": ""
      }
    }
  ],
  "priority": "1",
  "relatedParty": [
    {
      "id": "JohnDoe",
      "name": "JohnDoe",
      "role": "customer",
      "href": ""
    }
  ]
}
```

Listing 4.8. POST CFS Order request body example (add)

Instead, the following JSON is an example of request body for a delete action.

```
{
  "category": "vFW service",
  "description": "service order for vFW customer service",
  "externalId": "Customer_01",
   "note": {
    "author": "Orange",
```

```
    "date": "May 2019",
    "text": ""
  },
  "notificationContact": "",
  "orderItem": [
    {
      "action": "delete",
      "id": "1",
      "service": {
        "name": "vFW_01",
        "id": "{{auto_vFW_cfs_01_id}}",
        "place": [
          {
            "id": "",
            "name": "",
            "role": ""
          }
        ],
        "relatedParty": [
          {
            "id": "JohnDoe",
            "name": "JohnDoe",
            "role": "customer",
            "href": ""
          }
        ]
       }
    }
  ],
  "priority": "1",
  "relatedParty": [
    {
      "id": "JohnDoe",
      "name": "JohnDoe",
      "role": "customer",
      "href": ""
    }
  ]
}
```

Listing 4.9. POST CFS Order request body example (delete)

In these examples, CFS Orders are respectively about creating and removing a

virtual Firewall service for customer “John Doe”, as specified in the *relatedParty* field.

4.3.5 Instantiation Decision API

This API is used to get decision about the RFS to be instantiated to create a CSF with given characteristics. It provides the same methods than the previous ones, but in this case only POST requests are performed by the CFS Order API when it receives an order for CFS creation (i.e., the specified action is “add”).

In particular, the Instantiation Decision API server analyses the rules described in the CFS Specification stored in Service Resolver CFS Catalog and determines if they are satisfied by each candidate RFS to be included in the solution. Therefore, for each RFS associated to the CFS model, if the RFS provides the characteristics needed to create the required CFS, then it will be included in the list of RFS to be instantiated, otherwise it will be excluded from this list.

4.4 CFS Order workflow

This section provides details about the workflows executed by Service Resolver to process RFS and CFS Specifications and CFS Order creation, focusing on the interactions among the different API servers.

In order to send a CFS Order to Service Resolver, it is necessary to provide the definitions of the CFS which can be instantiated and of the RFS which can be used to create them. Therefore, CFS Specifications and RFS Specifications have to be inserted in Service Resolver database as prerequisite. In particular, the following two steps have to be executed by service designers:

1. Insertion of available RFS Specifications in Service Resolover RFS Catalog
2. Definition of the CFS Specifications describing the possible CFS which customers can request

The first operation regards the insertion of RFS models in Service Resolver RFS Catalog via the RFS Specification API. Figure 4.3 shows the UML sequence diagram detailing the actions performed when a POST request is sent to this API.

Before sending a POST request to insert a new RFS Specification in Service Resolver RFS Catalog, the service designer has to get the list of available models from ONAP SDC Catalog and look for those he is interested in. Once he knows the *id* of these models, he can fill correctly the request body and send the request to Service Resolver. At this point, the RFS Specification API server is responsible for retrieving service CSAR file from ONAP SDC, creating a new entry in the RFS Catalog and populating it with the information returned by ONAP SDC and External APIs components.

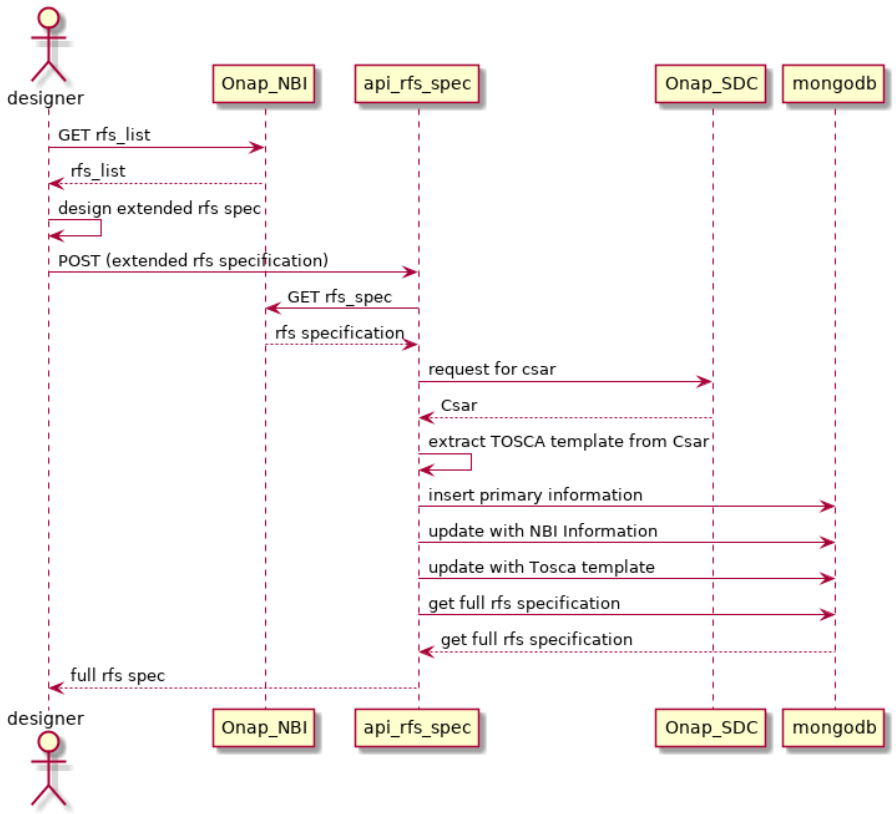


Figure 4.3. Workflow for RFS Specification creation

The second step to be executed before sending CFS Orders to Service Resolver is about creating CFS models in Service Resolver CFS Catalog. The UML sequence diagram in figure 4.4 shows the actions performed by the CFS Specification API server when a POST request is received.

In particular, this micro-service interacts with the RFS Specification API server in order to verify if the list of RFS included in the request body are valid (i.e., they exist in Service Resolver RFS Catalog), create the new CFS model in database and generate the TOSCA description for it.

At this point, once Service Resolver database has been populated, client applications can send CFS Orders to ask for executing the desired actions on available services.

For what concerns CFS Order creation via CFS Order API, a distinction between the add action and the delete one is needed.

In the first case, as shown in the sequence diagram in figure 4.5, the steps performed by Service Resolver are the following ones:

- *orderItem* validation by checking if the CFS it is referred to exists in CFS Catalog, therefore a GET request to the CFS Specification API is sent

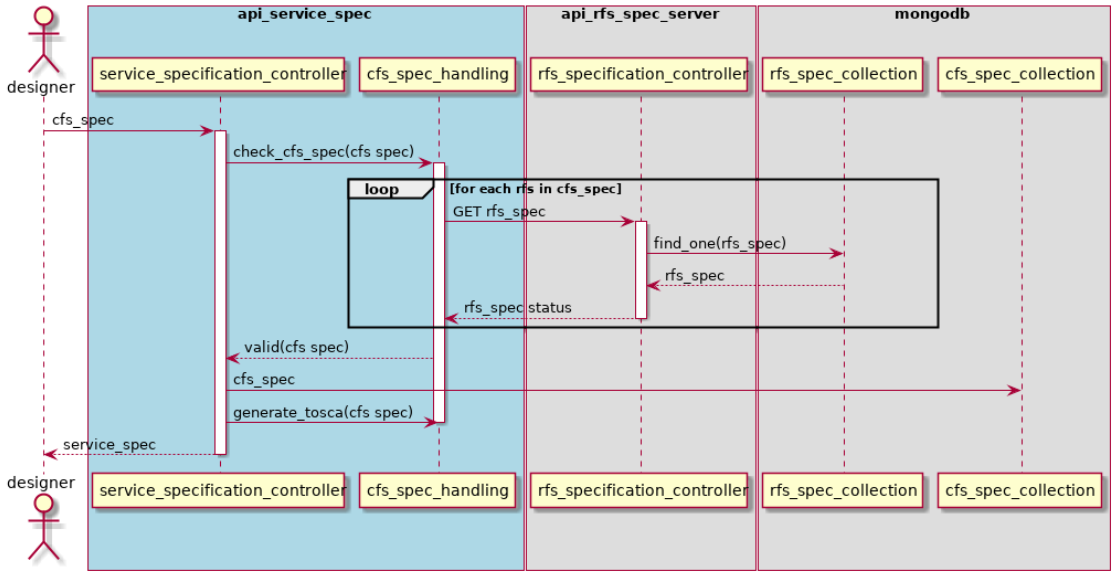


Figure 4.4. Workflow for CFS Specification creation

- CFS instance creation by sending a POST request to the Service Instance API
- For each RFS included in the CFS definition:
  - A validation is performed by sending a GET request to the RFS Specification API, allowing to check RFS existence in Service Resolver RFS Catalog
  - A POST request is sent to the Instantiation Decision API in order to know if that RFS has to be instantiated or not
- When the solution is chosen, for each RFS included in it:
  - The corresponding RFS Order is filled and sent to ONAP External APIs
  - In case of success, the new RFS instance is stored in Service Resolver Inventory
- The CFS instance is updated with the information returned by ONAP External APIs, containing details about the instantiated RFS

The second case is about the deletion of an existing CFS instance and the associated workflow is shown in figure 4.6. Firstly, a GET request to the Service Instance API is performed to retrieve the CFS instance to be removed, then the RFS instance associated to it are analysed. In particular, CFS Order API server checks if each RFS is associated only to that CFS and:

- If yes, it sends an RFS Order to ONAP NBI in order to delete the instance from ONAP and from Service Resolver Inventory

- Otherwise, the RFS is not deleted but only the corresponding relationship with the CFS instance is removed

In both cases, the CFS instance is updated via the Service Instance API and, finally, it is deleted from Service Resolver database.

To sum up, the workflow associated with the main operations performed by Service Resolver and how its components interact among each other have been explained, therefore next section will focus on Service Resolver testing activities performed during this thesis work.

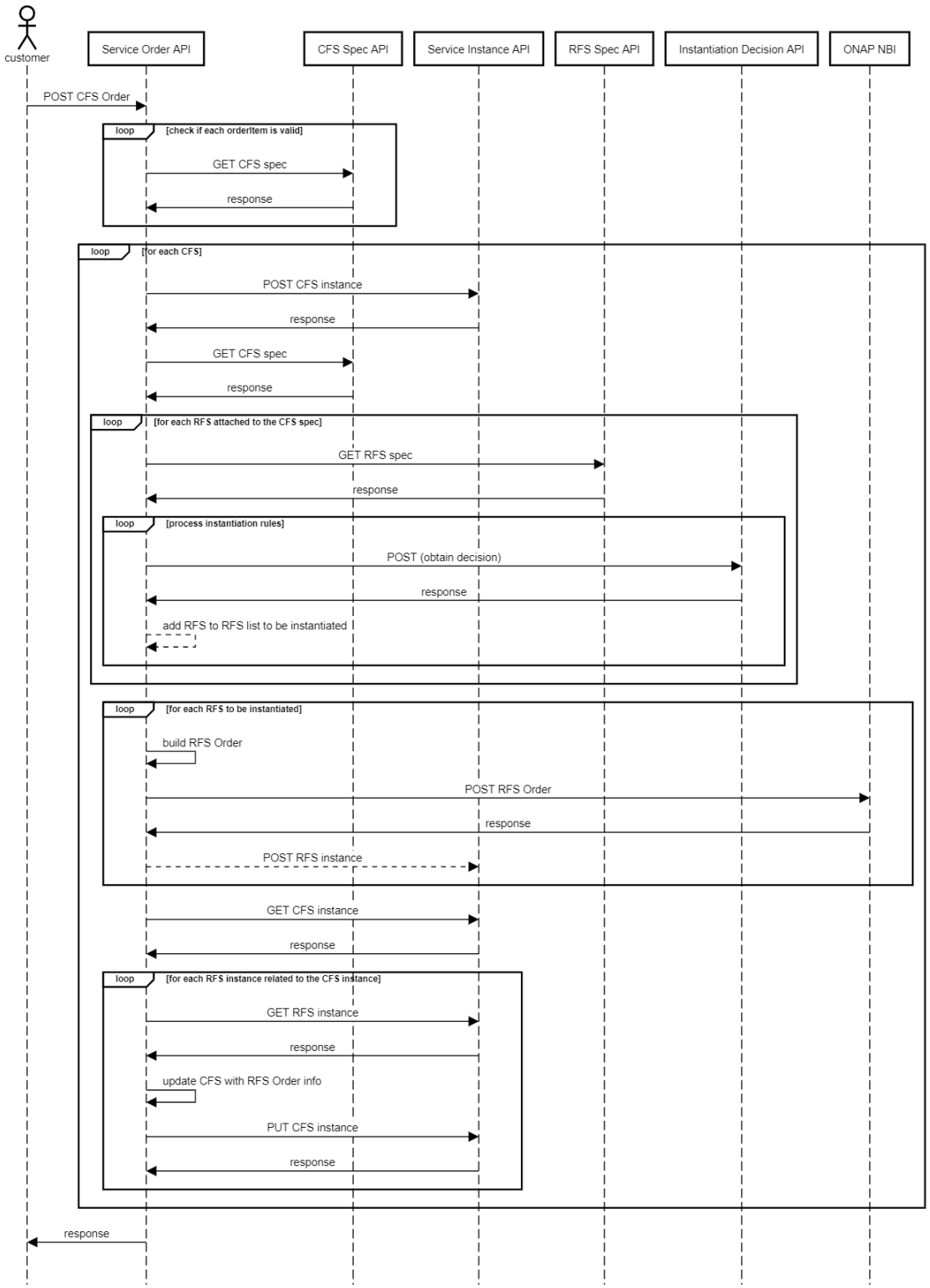


Figure 4.5. Workflow for CFS Order request (action=add)

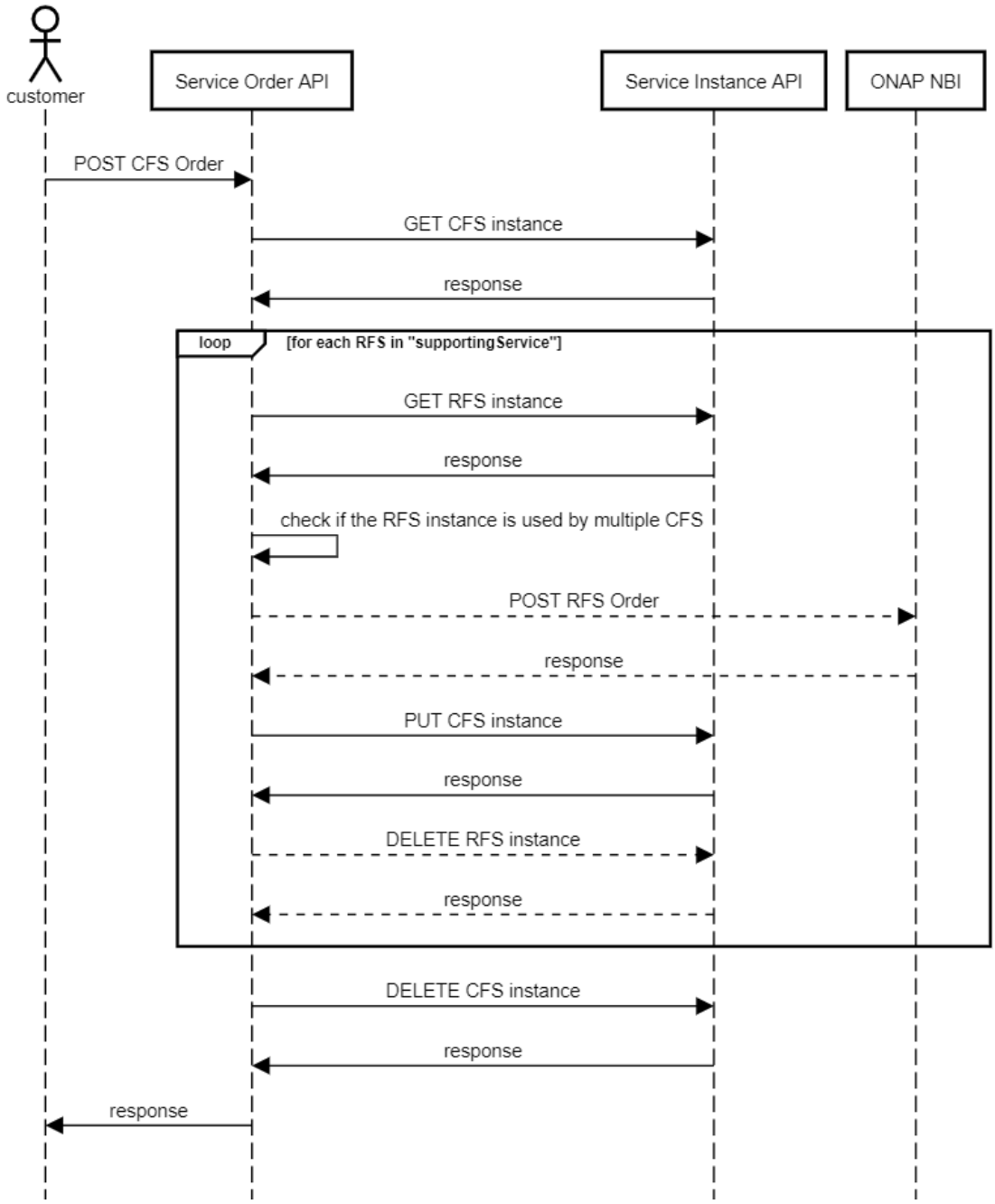


Figure 4.6. Workflow for CFS Order request (action=delete)

4.5 Service Resolver APIs testing and improvement

In order to test Service Resolver APIs and their functionalities, all the software has been installed on a virtual machine. The guide provided by Orange [34] explains all the steps to configure and run the application and provides a list of required tools to be installed as prerequisite. When Service Resolver has to be used in simulation mode, no additional configuration is needed, while when connecting it to an ONAP platform, the user has to modify the machine host file by specifying the name-IP address mapping of the ONAP services (e.g., ONAP SDC, A&AI and NBI). The host file of a Linux virtual machine is located in the *etc* system folder. Without adding these entries to the file, Service Resolver components are not able to communicate with the ONAP ones and the application returns a connection error, caused by failure of DNS resolution of names specified in URLs.

After the installation phase, all the steps to send a CFS Order have been executed, both for add and delete actions. As described in the previous section, these are:

- RFS Specifications insertion
- CFS Specifications definition
- CFS Order creation

For the first two, some POST requests to the RFS and CFS Specification APIs were sent. The same data structures described in the previous section have been used to fill request bodies. Moreover, for CFS model definition, different combination of instantiation rules and RFS have been tested successfully. An example is shown in below.

```
{
    "description": "vFW use-case",
    "name": "vFW",
    "version": "1.0",
    "category": "cfs",
    "href": "",
    "lastUpdate": "",
    "lifecycleStatus": "",
    "@type": "",
    "@baseType": "",
    "validFor": {
        "startDateTime": "",
        "endDateTme": ""
    },
    "serviceSpecCharacteristic": [
      {
```

```
      "description": "",
      "name": "featureLevel",
      "valueType": "string",
      "mandatory_characteristic": true
    }
  ],
  "serviceSpecRelationship": [
    {
      "id": "f307915a-29ff-4ee9-9aa0-92d83a85ccb6",
      "name": "vFW_model_A",
      "relationshipType": "cfs_to_rfs",
      "instantiationPriority": 3,
      "description": "that rfs is instantiated only if featureLevel
         is simple or not indicated",
      "instantiationDecisionRules": [
        {
          "ruleEvaluationPriority": 1,
          "rule_name": "param_in_cfs_order_rule",
          "rule_params": {
          "cfs_param_name": "featureLevel",
          "operator": "equal",
          "target_value": "simple"
        },
          "rule_results": [{
              "rule_response": "no_param",
              "decision": "instantiate",
              "is_definitive_decision": true
          },
          {
              "rule_response": "true",
              "decision": "instantiate",
              "is_definitive_decision": true
          },
          {
              "rule_response": "false",
              "decision": "do_not_instantiate",
              "is_definitive_decision": true
          }
          ]
        }
      ]
  },
```

```
{
    "id": "13968aed-5013-4284-a2dc-9ef124d67f77",
    "name": "vLB_model_A",
    "relationshipType": "cfs_to_rfs",
    "instantiationPriority": 2,
    "description": "that rfs is instantiated only if featureLevel
       is simple or not indicated",
    "instantiationDecisionRules": [
      {
        "ruleEvaluationPriority": 1,
        "rule_name": "param_in_cfs_order_rule",
        "rule_params": {
            "cfs_param_name": "featureLevel",
            "operator": "equal",
            "target_value": "simple"
        },
        "rule_results": [{
            "rule_response": "no_param",
            "decision": "instantiate",
            "is_definitive_decision": true
        },
        {
            "rule_response": "true",
            "decision": "instantiate",
            "is_definitive_decision": true
        },
        {
            "rule_response": "false",
            "decision": "do_not_instantiate",
            "is_definitive_decision": true
        }
      ]
    }
  ]
},
{
    "id": "d909d48d-db9a-47f5-abc8-b6b5326a8dfb",
    "name": "vPG_model_A",
    "relationshipType": "cfs_to_rfs",
    "instantiationPriority": 1,
    "description": "that rfs is instantiated only if featureLevel
       is simple or not indicated",
```

```
        "instantiationDecisionRules": [
        {
            "ruleEvaluationPriority": 1,
            "rule_name": "param_in_cfs_order_rule",
            "rule_params": {
                "cfs_param_name": "featureLevel",
                "operator": "equal",
                "target_value": "simple"
            },
            "rule_results": [{
                "rule_response": "no_param",
                "decision": "instantiate",
                "is_definitive_decision": true
            },
            {
                "rule_response": "true",
                "decision": "instantiate",
                "is_definitive_decision": true
            },
            {
                "rule_response": "false",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            }
          ]
        }
      ]
    },
    {
        "id": "6e143726-3987-49d5-b841-58dc795a23d8",
        "name": "vFW_model_B",
        "relationshipType": "cfs_to_rfs",
        "instantiationPriority": 3,
        "description": "that rfs is instantiated only if featureLevel
           is rich and if there is no longer capacity on existing ",
        "instantiationDecisionRules": [
        {
            "ruleEvaluationPriority": 1,
            "rule_name": "param_in_cfs_order_rule",
            "rule_params": {
                "cfs_param_name": "featureLevel",
                "operator": "equal",
```

```
                "target_value": "rich"
            },
            "rule_results": [{
                "rule_response": "no_param",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            },
            {
            "rule_response": "true",
                "decision": "instantiate",
                "is_definitive_decision": false,
                "next_instantiationDecisionRules": 2
            },
            {
                "rule_response": "false",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            }
            ]
        },
        {
            "ruleEvaluationPriority": 2,
            "rule_name": "rfs_instance_capacity_rule",
            "rule_params": {
                "spec_name": "vFW_model_B",
                "set_id_key": "all",
                "capacity": 2
            },
            "rule_results": [{
                "rule_response": "true",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            },
            {
                "rule_response": "false",
                "decision": "instantiate",
                "is_definitive_decision": true
            }
          ]
        }
      ]
    },
```

```
{
    "id": "1e28785d-d797-4b00-9365-79e43d107800",
    "name": "vLB_model_B",
    "relationshipType": "cfs_to_rfs",
    "instantiationPriority": 2,
    "description": "that rfs is instantiated only if featureLevel
       is rich and if there is no longer capacity on existing ",
    "instantiationDecisionRules": [
    {
        "ruleEvaluationPriority": 1,
        "rule_name": "param_in_cfs_order_rule",
        "rule_params": {
            "cfs_param_name": "featureLevel",
            "operator": "equal",
            "target_value": "rich"
        },
        "rule_results": [{
            "rule_response": "no_param",
            "decision": "do_not_instantiate",
            "is_definitive_decision": true
        },
        {
            "rule_response": "true",
            "decision": "instantiate",
            "is_definitive_decision": false,
            "next_instantiationDecisionRules": 2
        },
        {
            "rule_response": "false",
            "decision": "do_not_instantiate",
            "is_definitive_decision": true
        }
      ]
    },
    {
        "ruleEvaluationPriority": 2,
        "rule_name": "rfs_instance_capacity_rule",
        "rule_params": {
            "spec_name": "vLB_model_B",
            "set_id_key": "all",
            "capacity": 2
        },
```

```
            "rule_results": [{
                "rule_response": "true",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            },
            {
                "rule_response": "false",
                "decision": "instantiate",
                "is_definitive_decision": true
            }
          ]
        }
      ]
    },
    {
        "id": "0ba1e964-f0b7-4061-a6c2-1b59e37eaecd",
        "name": "vPG_model_B",
        "relationshipType": "cfs_to_rfs",
        "instantiationPriority": 1,
        "description": "that rfs is instantiated only if featureLevel
           is rich and if there is no longer capacity on existing ",
        "instantiationDecisionRules": [
        {
            "ruleEvaluationPriority": 1,
            "rule_name": "param_in_cfs_order_rule",
            "rule_params": {
                "cfs_param_name": "featureLevel",
                "operator": "equal",
                "target_value": "rich"
            },
            "rule_results": [{
                "rule_response": "no_param",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            },
            {
            "rule_response": "true",
                "decision": "instantiate",
                "is_definitive_decision": false,
                "next_instantiationDecisionRules": 2
            },
            {
```

```
                "rule_response": "false",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            }
          ]
        },
        {
            "ruleEvaluationPriority": 2,
            "rule_name": "rfs_instance_capacity_rule",
            "rule_params": {
                "spec_name": "vPG_model_B",
                "set_id_key": "all",
                "capacity": 2
            },
            "rule_results": [{
                "rule_response": "true",
                "decision": "do_not_instantiate",
                "is_definitive_decision": true
            },
            {
                "rule_response": "false",
                "decision": "instantiate",
                "is_definitive_decision": true
            }
          ]
        }
      ]
    }
  ]
}
```

Listing 4.10. Example of CFS Specifications used in tests

Once the models have been created in Service Resolver database, a CFS Order can be sent to add a new instance in service Inventory. The first test was made in simulation mode, so without connecting the Service Resolver to a real ONAP platform. In this case, order processing was successfully completed, but there was an error when inserting a new RFS model in Service Resolver. This was due to the need to specify the ONAP platform in the request body even if Service Resolver is working in simulation mode, but the guide provided by Orange did not specified any instruction about that. Therefore, the problem was notified to Orange that updated the documentation.

For what concerns tests performed on Service Resolver connected to ONAP, some

issues were found. Firstly, there were some coding errors that caused the failure of the CFS Order sent. In particular, a “list assignment index out of range” and a “KeyError” were fixed in the Python code responsible for building the request body of RFS Order to be sent to ONAP External APIs. After that, another issue causing CFS Order failure was notified to Orange. This error occurred when sending RFS Orders to ONAP NBI. In more detail, when Service Resolver sends RFS Orders to the ONAP platform, it has to perform the following actions:

- Send RFS Order to ONAP External APIs
- Wait for a response
- If the response notifies success, store the returned RFS instance in service Inventory
- Otherwise, notify failure

After that, as shown in the previous section, Service Resolver updates the created CFS instance with the information included in the RFS returned by ONAP NBI component. The above mentioned problem was not related to ONAP NBI response, which notified success, but each created RFS instance was stored in Service Resolver Inventory with an *id* value different than the one returned in ONAP NBI response body. As a consequence, when a GET request to the Service Instance API was performed in order to retrieve the new RFS instance and use it to update the CFS one, the CFS Order API used the id provided by ONAP NBI and not the one actually stored in the Inventory. At this point, the application returned an error and communicated CFS Order failure to the user. Therefore, the issue was reported to Orange and fixed.

The second part of testing activity focuses on CFS Orders to delete existing CFS instances. The same data structure described in the previous section was used, both in simulation mode and with the connection to ONAP. Initially, no errors were found, but all RFS Orders sent from Service Resolver to ONAP External APIs failed. In order to understand the reason of the failure, an inspection of the log files of ONAP NBI application was needed. The error found in these log files was the following one: ONAP NBI module did not receive the information about the customer (which has to be included in the *relatedParty* field) for the RFS Order to be sent to ONAP Service Orchestrator. Consequently, the RFS instance to be removed was searched among the ones associated to the “generic” customer in ONAP A&AI, but as the “generic” customer was not created before, this request failed and an error was returned causing the RFS Order state field to be set to “failed”. At this point, the problem was notified to Orange and the cause was discovered: Service Resolver uses predefined templates to build the payload of RFS Order requests, but when sending requests to delete RFS instances, the *relatedParty* field and the *serviceSpecification* one, containing information about the RFS model used to create the corresponding CFS instance, were not filled by the application. Therefore, a correction of the code responsible for this operation was made.

All the problems found during the testing phase were fixed and CFS Order processing works properly now. Service Resolver development is not completed yet, so its code is continuously updated and new features are periodically added. At the moment, Orange is working on “Macro” instantiation method that does not follow the TMF641 specification, and a new use case related to 5G network slicing has recently been included in the public repository.

In conclusion, Service Resolver is a “work in progress” application, which needs to be continuously updated, tested and fixed if necessary. A possible feature to be added in the future can be the support of the E2E service instantiation method via ONAP External APIs, but actually its development has not been planned yet. Moreover, the ONAP community will need to include Service Resolver functionalities in the platform sooner or later, but also this aspect has not been defined yet. Nevertheless, Service Resolver realization confirms that service providers are more and more adopting standardized approaches, with a particular interest on customer needs.

Chapter 5

Service Portal: architecture

5.1 Overview

Service Portal is a BSS application which allows users to order Customer Facing Services exposed as products, therefore it is placed at Product layer of eTOM. In particular, through this portal, customers can see the list of available services and perform orders for the desired ones, generating a CFS Order with action="add". These requests are sent to Service Resolver, which will process them and return a response. All purchased products are displayed in another list with the possibility to send a delete request if the user wants to disable the service, which corresponds to a CFS Order with action="delete". Moreover, it is possible to check all performed orders, both for creation and deletion, and the details about their state (i.e., "aknowledged", "completed" or "failed").

The list of products available to customers can be configured by a user with administration privileges, i.e., a Telecom Italia company member responsible for portal management.

In the next sections, Service Portal architecture and functionalities are described in more detail.

5.2 Overall design

Service Portal architecture design is based on a set of functional requirements to be satisfied. The previous section provides a general overview about these functionalities, therefore a deeper analysis is needed.

From a functional point of view, three different categories can be distinguished:

- Client side, including all the functionalities available through the GUI
- Server side, containing the application logic and an API layer
- Data persistence, for data storage

In more detail, client side functionalities includes all the operations that a user should be able to perform through the GUI, from ordering a product to configure products which can be purchased, providing role-based registration and login. Moreover, dynamical management of graphical components required for web pages rendering and for reacting to user actions is performed at this layer.

For what concerns server side, the application should be able to collect user requests, process them and generate the corresponding requests to be sent to Service Resolver. Other functionalities such as users management and authentication, possibility to modify data stored in database or to configure GUI should also be provided.

5.3 Architecture

Based on the functional requirements described before, Service Portal is composed of three modules: backend, frontend and database. Figure 5.1 shows the architecture and how the mentioned components interact.

Firstly, the backend part manages all server side operations, so it has to serve client requests by interacting with database and Service Resolver. The frontend part acts as a client to the backend one and manages all actions performed by the user on browser.

All the components can be runned in standalone Docker containers, giving the application a micro-service based structure. Inter-container communication is managed thorugh a user-defined network and communication between frontend and backend is performed via proxy.

In particular, the Docker network is a virtual network linking all the containers among them, such that they can communicate to each other by using container IP address or container name, thanks to automatic DNS resolution performed for user-defined bridge networks.

For what concerns the proxy, it is responsible for redirecting requests performed by the browser to backend real IP address and port.

Next, a focus on each one of the described components is provided, explaining its architecture and design choices.

5.3.1 Backend

The backend exposes a REST based API towards the frontend and perform requests to Service Resolver APIs. In particular, it exploits CFS Order API and CFS Specification API to send CFS Order and to get CFS models to fill them, respectively.

The main operations performed by the backend component are the following ones:

- Products management: creation, retrieval, modification and deletion of products stored in database

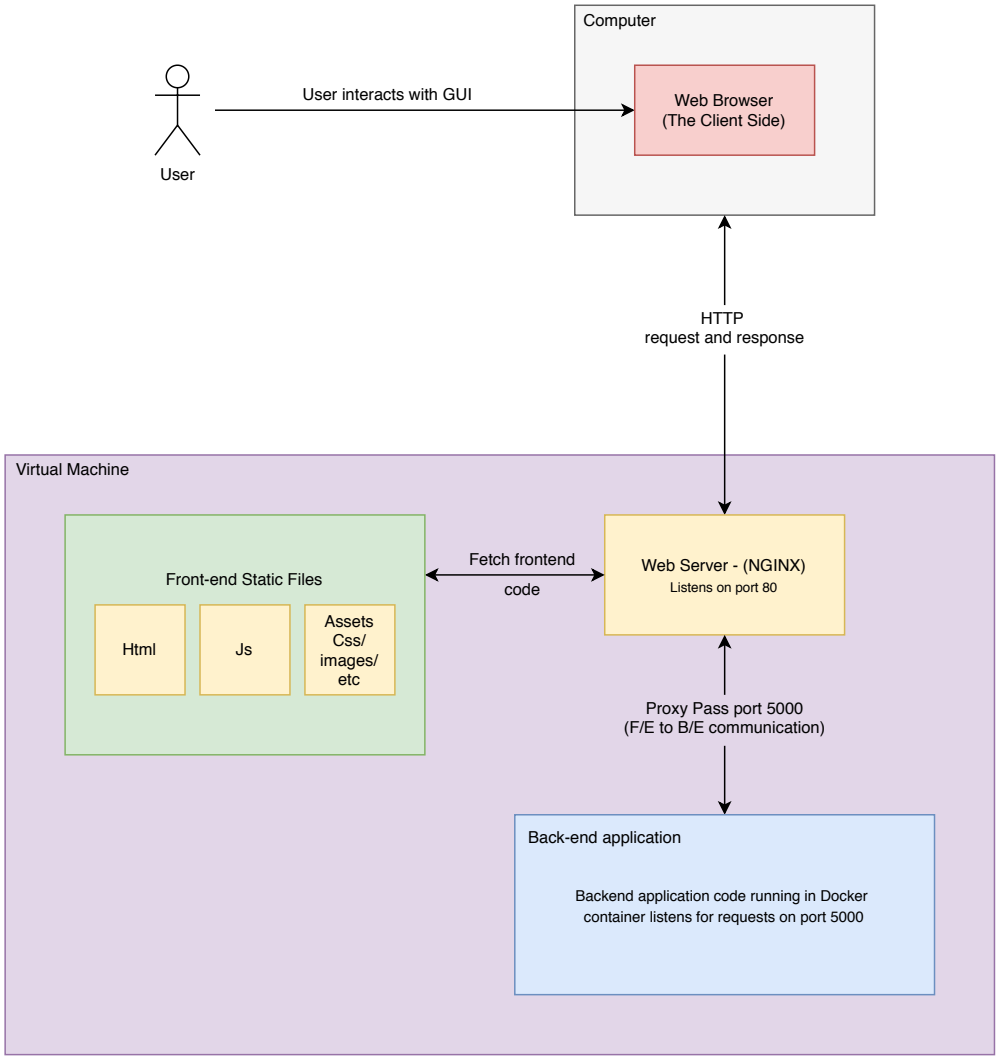


Figure 5.1. Service Portal architecture

- Order management: creation, retrieval and deletion of orders according to customer's requests
- Models retrieval: read the list of CFS Specifications stored in Service Resolver CFS Catalog
- Inventory management: retrieval and deletion of purchased products, corresponding to CFS active instances stored in Service Resolver Inventory
- User management: creation, retrieval and deletion of users
- User authentication: login and logout operations

A key point of backend design is it is model-driven. This means that operations should not depend on a particular product to be displayed to customers or on the particular CFS it is related to. This also requires a mapping between product models and CFS models, which is model-driven too.

In order to discuss the presented features, Service Portal REST API and mapping operations are described.

REST based API

Backend REST API provides an interface to allow client applications to execute any operation required by the user. Among the provided methods, there are a set of them which can be used only by administrators.

Firstly, the API methods to perform product management are presented. Figure 5.2 shows their design.

Resource	Method	Request body	Status code	Resp. body	Meaning
/products	GET		200	Products list	OK
	POST	Product	201	Product	Created
			400		Bad Request
			401		Unauthorized
/products/pages	GET		200	Number of products pages and total products	OK
/products/pages/{page}	GET		200	Products list	OK
/products/{name}	GET		200	Product	OK
			404		Not Found
	PUT	Product	204		No Content
			400		Bad Request
			401		Unauthorized
			404		Not Found
	DELETE		204		No Content
			400		Bad Request
			401		Unauthorized
			404		Not Found

Figure 5.2. API methods for products management

The GET methods allow to access the list of available products or a specific product in the Catalog. Since all users can perform an order on every product, these methods do not require administration privileges. In addition, it is possible to ask only for a specific products page if the retrieval of the complete list should be avoided for performance improvement. Then, a POST method is also provided for product creation, an operation which can be executed only by an administrator. Also the PUT and DELETE methods are reserved for admin profiles and are needed to modify or delete existing products.

A product in the Catalog is characterized by the following information:

- A *name* field, which is a string identifying uniquely each product
- A *description*, containing details about product characteristics
- A *price* field
- A *category* to classify it
- The *name* and the *id* of the CFS model the product is referred to
- The list of *service characteristics* defined in CFS Specification that are needed to realize that product

The CFS Specification id is not provided because Service Portal retrieve from Service Resolver by sending a GET request to the CFS Specification API and looking for the name value specified in the CFS model name field. An example of body for a POST request is shown below.

```
{
    "name": "FW gold",
    "description": "Richest Firewall model with medium protection level
       ",
    "price": "40€",
    "category": "company",
    "CFSmodelName": "vFW",
    "serviceCharacteristic": [
        {
            "name": "featureLevel",
            "value": "rich"
        },
        {
            "name": "protectionLevel",
            "value": "medium"
        }
    ]
}
```

Listing 5.1. Request body example for POST /products

In order to retrieve and configure possible categories for products, a GET, a POST and a DELETE methods are provided, as shown in figure 5.3. These methods are only for administration purposes. A category is represented as a name value pair, which therefore identifies it.

For what concerns order management, Service Portal offers the possibility to GET all orders, also divided by page number, or a single order by id. Orders can

Resource	Method	Request body	Status code	Resp. body	Meaning
/categories	GET		200	Categories list	OK
	POST	Category	201	Category	Created
			400		Bad Request
			401		Unauthorized
/categories/{name}	DELETE		204		No Content
			400		Bad Request
			401		Unauthorized
			404		Not Found

Figure 5.3. API methods for categories management

be filtered by providing the customer associated with them as query parameter in URL. In particular, each customer can access to his orders, but can not see orders performed by other customers: this requires administration privileges. Then, a POST method allows to create a new order for a specific product, which is then mapped to the corresponding CFS Order to be sent to Service Resolver, both for the add and the delete actions. Administrators have also the possibility to DELETE a given order. All the mentioned methods are listed in figure 5.4.

Resource	Method	Request body	Status code	Resp. body	Meaning
/orders	GET		200	Orders list	OK
			401		Unauthorized
	POST	Order	201	Order	Created
			400		Bad Request
			401		Unauthorized
/orders/pages	GET		200	Number of orders pages and total orders	OK
			401		Unauthorized
/orders/pages/{page}	GET		200	Orders list	OK
			401		Unauthorized
/orders/{name}	GET		200	Order	OK
			401		Unauthorized
			404		Not Found
	DELETE		204		No Content
			400		Bad Request
			401		Unauthorized
			404		Not Found

Figure 5.4. API methods for orders management

An order object is described by the following information:

- An *id* field, a string which uniquely identifies it

- The *customer* which performed the order
- The *name* of the ordered product
- The *action* to be performed, therefore add or delete
- In case the action is delete, the *id* and *name* values of the CFS instance to be removed from Service Resolver Inventory have to be specified
- The *status* information, which is filled based on Service Resolver response after having completed order processing

An example of body for a POST request to send a new order is shown below.

```
{
    "id": "string",
    "customer": "TIM",
    "product_name": "FW gold",
    "action": "add",
    "CFSinstanceId": "",
    "CFSinstanceName": "",
    "status": ""
}
```

Listing 5.2. Request body example for POST /orders

Figure 5.5 is the UML sequence diagram which shows in details all the operations performed by Service Portal to process an incoming oder.

In particular, the workflow executed to fulfil an order request can be resumed as follows:

1. **Check payload**: when a new order is received, the first action is to check it is valid, otherwise a 400 error code is returned in the response
2. **Check customer information**: this information is needed to fill the relatedParty field for the CFS Order request body to be sent to Service Resolver
3. **Check order action**: CFS Order request body is filled based on this parameter, in particular:
   (a) If action is “add”, service characteristics defined in the product have to be specified, therefore the product has to be retrieved from database
   (b) If action is “delete”, CFS instance information has to be included
4. **Wait for Service Resolver response**

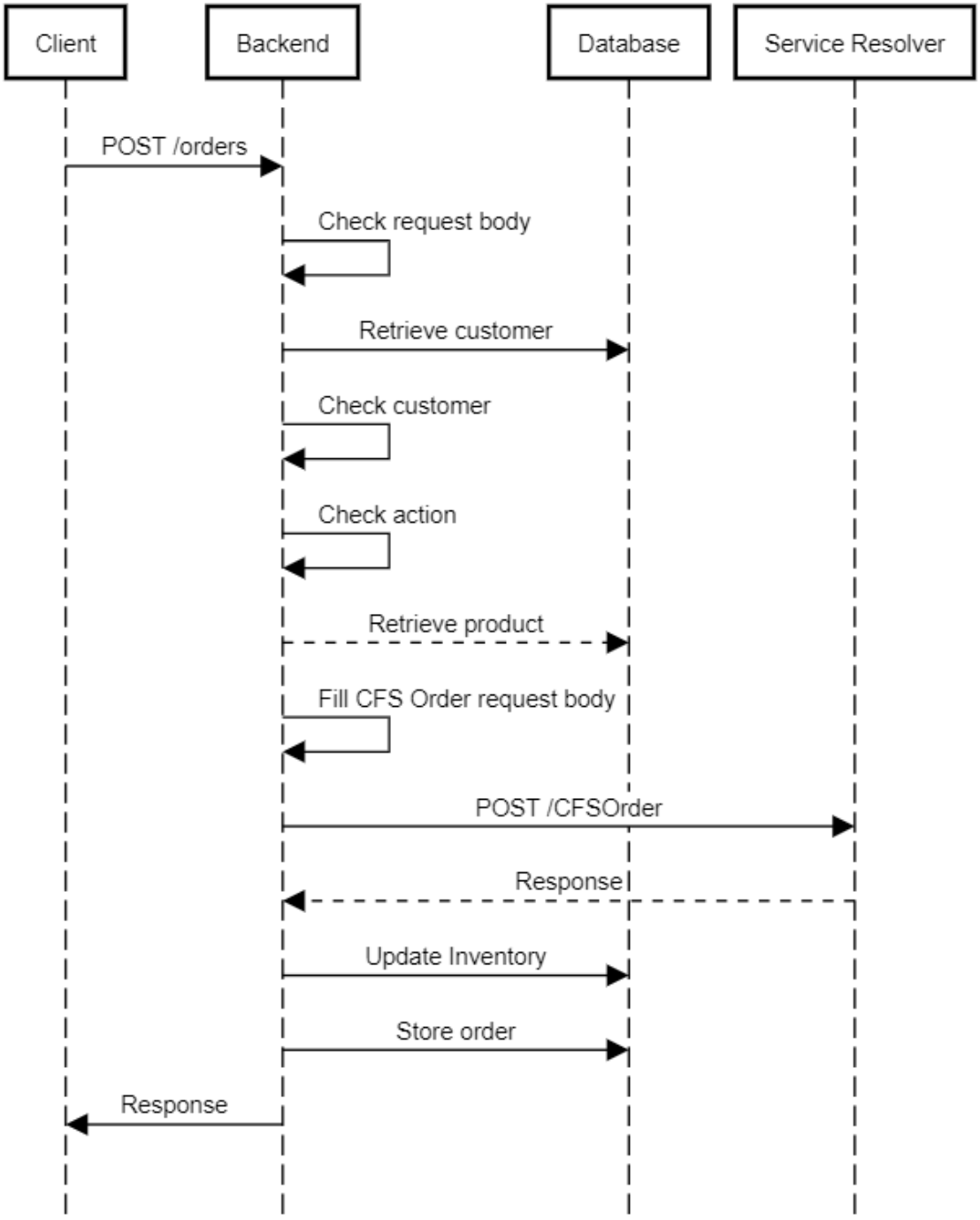


Figure 5.5. Workflow for order creation

5. **Store order and update inventory data** according to the received response
6. **Return response** indicating success or failure

For what concerns CFS models retrieval from Service Resolver CFS Catalog the API methods allowing this operation are shown in figure 5.6. The response body is the list of available models in case of a GET request for `/models` resource, while when performing a GET request specifying the path `/models/{name}/characteristics`, the list of service characteristics associated to the CFS model with name corresponding to the one specified as path parameter is returned.

Resource	Method	Request body	Status code	Resp. body	Meaning
/models	GET		200	CFS models list	OK
/models/{name}/characteristics	GET		200	Service Characteristics	OK

Figure 5.6. API methods for CFS models retrieval

All purchased products, corresponding to the active CFS instances, are stored in a Product Inventory. The management of Service Portal Inventory can be performed through the method listed in figure 5.7. In particular, the GET method provides access to the entire Inventory or to a given page of products stored inside it, and the result can be filtered by customer, specified as query parameter. Also in this case, a customer can get only the list of his purchased products. Administrators can also DELETE a product from the Inventory.

Resource	Method	Request body	Status code	Resp. body	Meaning
/inventory	GET		200	Inventory items list	OK
			401		Unauthorized
/inventory/pages	GET		200	Number of inventory pages and total inventory items	OK
			401		Unauthorized
/inventory/pages/{page}	GET		200	Inventory items list	OK
			401		Unauthorized
/inventory/{id}	DELETE		204		No Content
			400		Bad Request
			401		Unauthorized
			404		Not Found

Figure 5.7. API methods for Inventory management

Finally, there are a set of API methods which allow to manage users and authentication. Regarding users management, the available methods are shown in figure

5.8.

Resource	Method	Request body	Status code	Resp. body	Meaning
/users	GET		200	Users list	OK
			401		Unauthorized
	POST	User	201	User	Created
			400		Bad Request
/users/{username}	GET		200	User	OK
			401		Unauthorized
			404		Not Found
	DELETE		204		No Content
			400		Bad Request
			401		Unauthorized
			404		Not Found

Figure 5.8. API methods for users management

The POST method is used to create a new user, while the GET and the DELETE ones are provided for administration purposes. The information required to create a new user are the following ones:

- *Name* and *surname* of the user
- *Company* name
- *Email*
- *Password*
- *Admin* flag

In case the admin flag is set to true, the application checks if the user can be created with administration privileges. This is established by checking user company name and email, which have to be referred to Telecom Italia. This mechanism is a design choice to easy manage role-based access with respect to the provided functional requirements.

User authentication procedure can be performed through the methods shown in figure 5.9, which allow to execute login and logout actions.

In addiction, the Swagger [1] documentation of the whole API has been generated by exploiting JSON schema definition of API resources.

[1] Swagger [35] is a suite of tools for API development, from design and documentation, to test and deployment.

Resource	Method	Request body	Status code	Resp. body	Meaning
/login	POST	Credentials	200	Access token	OK
			401		Invalid credentials
/logout	POST		200	Ok message	OK

Figure 5.9. API methods for users authentication

Model-driven mapping

As explained in previous sections, Service Portal backend works following a model-driven approach. In particular, it is applied both at product and order levels for the following reasons:

- Product management is model-driven because product characteristics can be defined at creation time and in the same way for any kind of products, without influencing how the application works
- Ordering procedure satisfies this principle thanks to the fact it is performed by mapping products characteristics to the corresponding CFS model to be instantiated or to CFS instance to be deleted, without depending on the particular product or CFS Specification to be used

In more detail, product and order models are defined as JSON templates and are consumed by Service Portal API to execute configuration and mapping operations. All data referred to these models are then stored in an appropriate table inside a database, which contains the set of information needed to describe and use the created models when a certain operation is required by user.

5.3.2 Frontend

The frontend part of Service Portal is responsible for rendering graphical components in the GUI, managing the interaction with the user and acting as a client to the backend side.

GUI design is based on the different functionalities to be provided to users. In particular, web pages structure allows to make a clear distinction among the different operations the user can perform, thanks to the fact it is simple and intuitive. The application is composed by the following pages:

- An home page, describing Service Portal and its main features
- A login page, containing a simple login form with an email and password input fields

- A page for user registration
- User personal area, including three pages for simple customers: a Catalog page showing all products which can be ordered, an Inventory page listing the purchased products and from which an active product can be removed and an Orders page which allows to get details about performed orders. A screenshot of the mentioned pages is provided in figure 5.10.
- Three additional pages for administrators to allow product management: a page for product insertion through a form, a page for product modification and one for product deletion. They are shown in figure 5.11.

Some additional comments about pages for product management are needed. In particular, the administrator can insert products by simply filling the corresponding form fields describing it and selecting product category and CFS model name from a dropdown menu. Product modification is similar, but the product to be modified can be chosen from a dedicated menu which displays all the existing products in the Catalog. Thus, when the admin selects the product he wants to update, all the other form fields are automatically filled and he can simply change the information as desired. Finally, product deletion is very simple: the desired product can be chosen and removed clicking on the corresponding button.

Moreover, layout and graphical components have been designed in order to keep uniform all application pages. In particular, page navigation is available from links displayed in the navigation bar at the top of page, while an additional side bar is shown only to administrator profiles and allows navigation through product management pages.

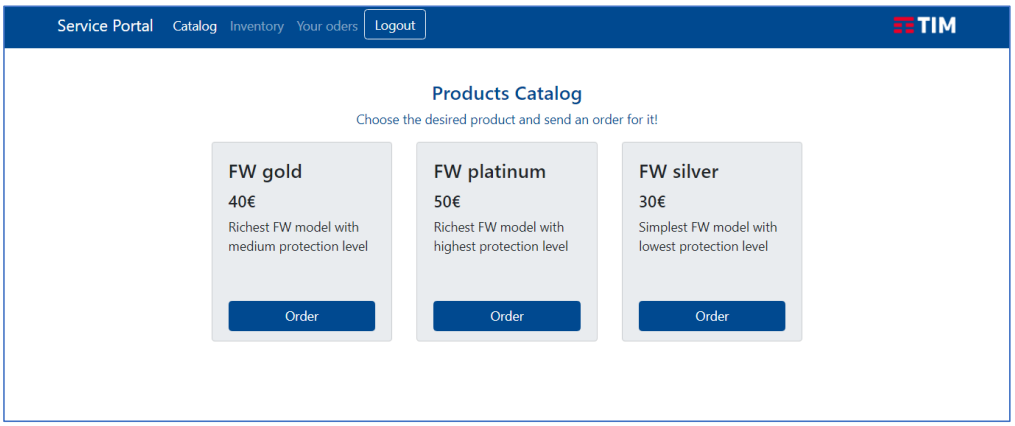
5.3.3 Database

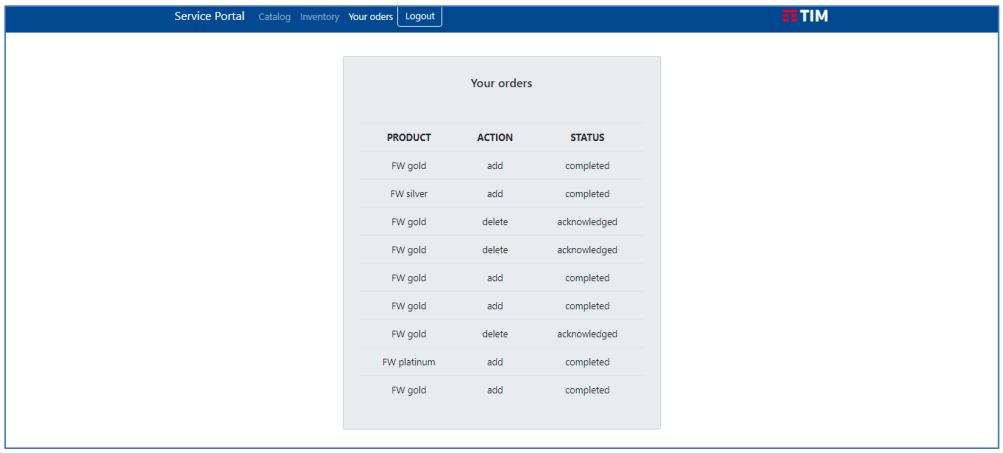
Database architecture is defined based on the structure of data to be stored. The following sets of data are managed:

- Product models, i.e. Product Catalog
- Orders
- List of purchased products, which corresponds to Product Inventory
- Users
- Blacklisted authentication tokens in order to prevent users to perform login using an expired token

Each one of the mentioned categories of information is stored in a different table inside the database. Table attributes corresponds to the ones defined in JSON bodies produced and consumed by backend API methods. For products an additional column

field has been added in order to store CFS model id once it has been retrieved from Service Resolver. In this way, the backend can get and manipulate data without further processing. This represents a very important point for application design, since database structure has a strong influence on performances.





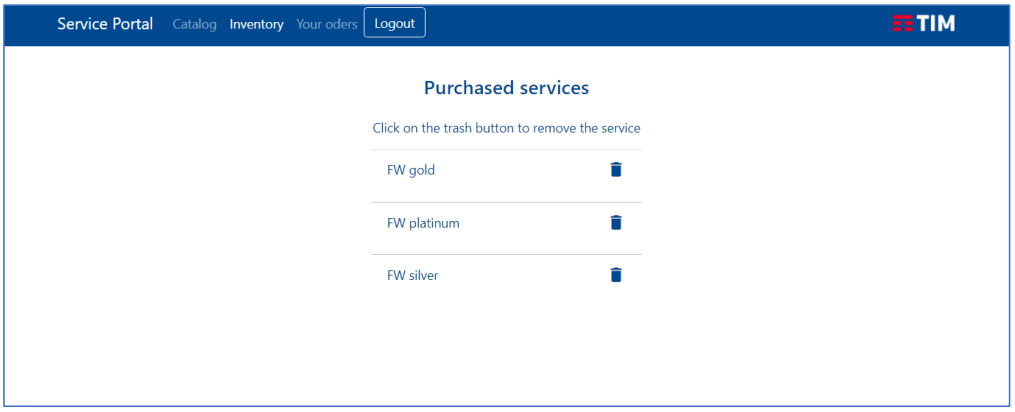


Figure 5.10. Service Portal customer personal area

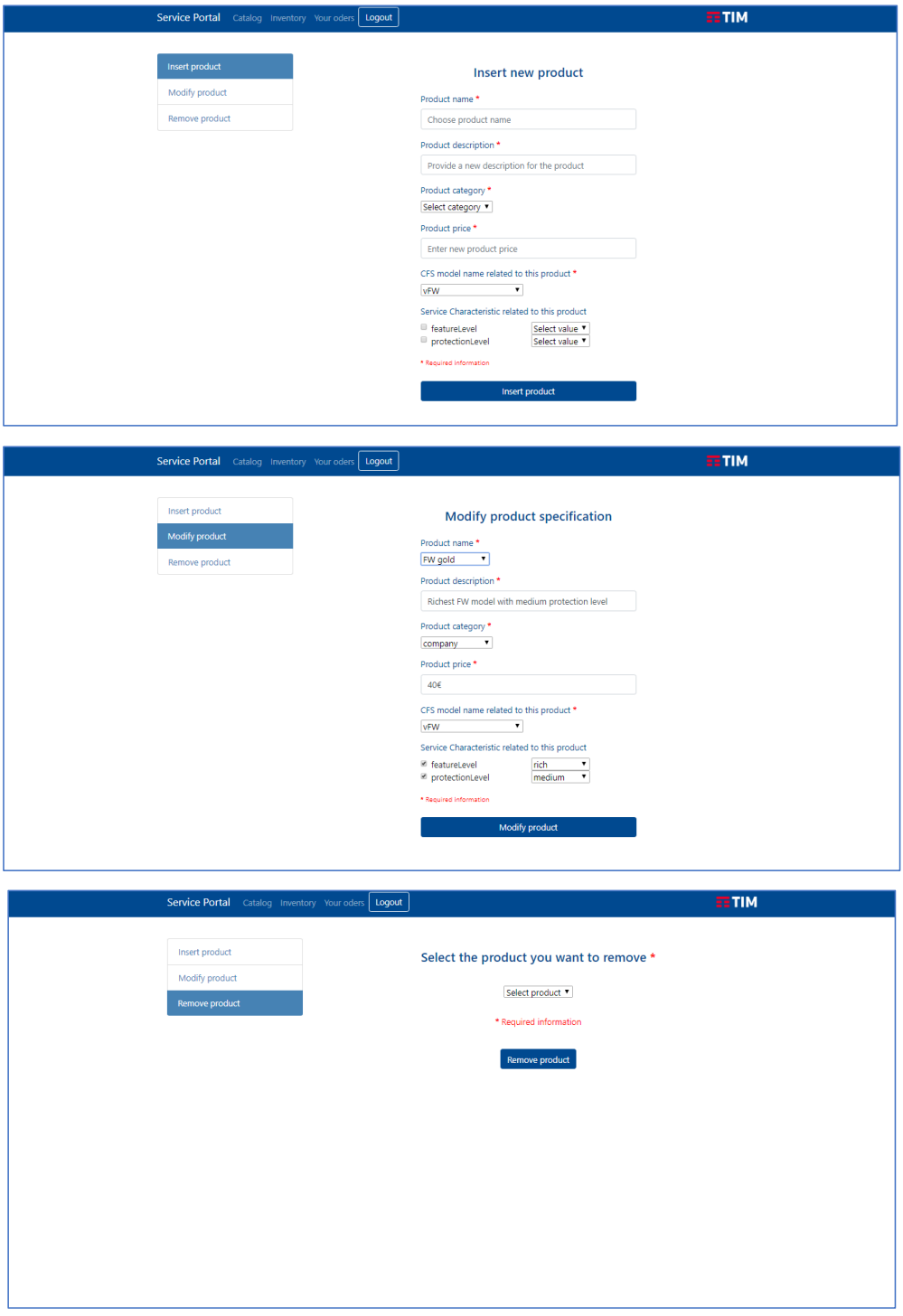


Figure 5.11. Service Portal administrator personal area

Chapter 6

Service Portal: implementation

The starting point for Service Portal implementation was performing technical choices for tools to be used. Once these choices have been made, code structure was defined. After that, implementation phase has been completed and finally the application was configured and tested.

6.1 Technical choices

From a technical point of view, it is necessary not only to choose tools able to satisfy functional requirements described in the previous chapter, but also to allow a clean and fast development and to make the application effective and robust.

Among all the available alternatives, an evaluation has been performed according to the features to be implemented in each component of the application. Therefore, the technologies chosen to build frontend, backend and database and the motivation behind these choices are presented in this section. After that, a focus on the configuration of the whole application is also provided.

6.1.1 MySQL database

The first step regards database implementation, on which backend implementation is also based. According to the described design, a relational database was implemented using MySQL database management system [36]. This choice is due to the following reasons:

- Different kinds of relationship can be identified among tables and data stored inside them
- Tables have a reduced horizontal dimension because data are characterized by a limited number of attributes
- Data availability, security and reliability

- ACID[1] properties are guaranteed while executing transactions
- Ensure scalability when large amount of data has to be managed
- Fast management and high performance
- It is open source, so it is a flexible solution which simplifies maintenance, debugging and upgrade operations

In conclusion, database is composed by five small-sized tables and its implementation was simple and fast.

6.1.2 Python Flask

The programming language selected to develop Service Portal backend was Python and a framework named Flask [37] was used to implement it.

Flask is a lightweight WSGI[2] web application framework. It is designed to easily implement web services, giving developers the possibility to get started quickly. Moreover, it offers the ability to scale up to complex applications, and because of its features it has become one of the most popular Python web application frameworks. Flask suggests how to develop a web application, but it does not enforce any dependency and does not impose any particular project layout. It is up to the developer to choose the tools and libraries he wants to use among the large set of extensions offered by the community. Thanks to simplicity of adding new extensions to a project, adding new functionalities is also very easy.

Regarding this work, the choice of using Flask is due to the following reasons:

- It offers all the tools needed to implement the different functionalities the backend is required to provide. An example is the possibility to exploit an extension to implement token-based user authentication.
- It is easy to learn, thus the time needed to implement backend component was reduced
- It is well documented and easy to debug
- It allows to write clean code, with the possibility to extend or modify it very easily in case this will be necessary in future

[1] ACID is the acronym of Atomicity, Consistency, Isolation and Durability, which are the properties to be guaranteed by transaction operations executed on stored data.

[2] The therm WSGI means *Web Server Gateway Interface* [38], which is a communication protocol defining an implementation-agnostic interface between Python web servers and web applications or frameworks, created to allow the development of portable web services.

- Implementing a REST API and mapping all resources and methods to be executed when a certain request is received is very simple thanks to the `@app.route()` decorator. An example of its use is provided below.

```
@app.route("/api/v1/users/", methods=['GET', 'POST', 'PUT'])
def users():
    ...
```

Listing 6.1. Example of usage of Flask @app.route decorator

Further examples of code and used libraries are provided in the next section.

6.1.3 React and Bootstrap

The framework chosen for frontend implementation is React [39], which is a JavaScript library for building user interfaces and whose popularity has increased rapidly during last years. In this context, React has been use to create HTML pages and components and to manage user interaction with them.

React offers several advantages to frontend developers. First of all, it simplifies component design and writing. In fact, since it is component-based, complex user interfaces can be implemented starting from single component definitions and then they can be composed as needed. Moreover, React uses JSX, which stands for JavaScript XML [40]. In particular, it is an optional syntax extension to JavaScript which allows to write HTML elements in JavaScript and place them in the DOM. The power of React is exactly related to this: it recognizes that rendering logic is strictly coupled with UI logic. Events handling, state changes over time and how data should be processed in order to be displayed are all operations related among them. An example of the JSX syntax used to define React components is shown below.

```
var name = 'Josh Perez';
var element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Listing 6.2. Example of JSX React syntax

An additional benefit of using a component-based approach in UI development is the possibility to write reusable code. Consequently, application growth and maintenance is simplified and code can be kept clean: this improves productivity and makes

application logic easier to manipulate.

React provides also the capability to ensure optimized and faster rendering. This is related to the concept of “state”, which allows the application to effectively manage data to be displayed. In more detail, React components can be characterized by state and “props”, which are both properties which can be passed as HTML attributes. The difference between them is that while props are immutable, state can change at any time. Therefore, components rendering and updates are performed only when a state variable changes. An example of how to use props and state is provided below.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, {this.props.name}!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Listing 6.3. Example of usage of React state and props

Regarding Service Portal frontend implementation, React has been chosen not only thanks to the features above described, but also for the following reasons:

- It is easy to learn, therefore frontend development was fast
- It allows to exploits other libraries and frameworks, in particular the Bootstrap CSS framework [41] to build HTML pages
- It is well-documented
- It is continuously improved by its developers community

Therefore, actually frontend was developed by combining two frameworks: React and Bootstrap. In this way, rendering, events management, sending requests to backend API and all the required JavaScript code was developed using React, while HTML pages and CSS stylesheets was defined using Bootstrap. In particular, Bootstrap allows to create professional web pages and supports major of all browsers, it is light, customizable and reduces time to design HTML components.

6.2 Code's structure

According to Service Portal architecture, code is split in two main parts: backend and frontend. Project organization is displayed in figure 6.1. Then, the structure of both components is described in more detail.

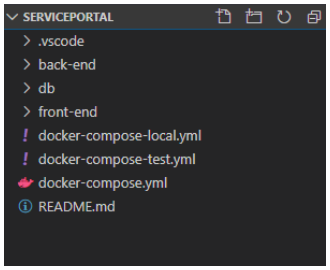


Figure 6.1. Service Portal code's organization

6.2.1 Backend code

Backend code organization is shown in figure 6.2. The *app.py* file contains all API resources and methods definitions and the code implementing them. Moreover, it includes the definition of all configuration parameters and the `main` method implementation.

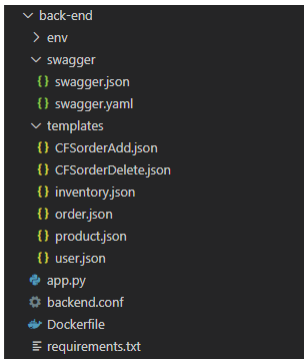


Figure 6.2. Service Portal backend code's organization

An example of code implementing the API method to get product categories is shown below.

```
@app.route('/categories', methods=['GET'])
@jwt_required
def getCategories():
```

```
connection = mysql.connector.connect(user=db_username,password=
    db_password,host=db_host,port=3306, database=db_database)
cursor = connection.cursor(buffered=True)
query = ("SELECT * FROM categories")
data = ""
try:
    cursor.execute(query, data)
    response = []
    if cursor.rowcount > 0:
        result = cursor.fetchall()
        for row in result:
            category = {'name': str(row[1])}
            response.append(category)
    connection.commit()
    cursor.close()
    connection.close()
except mysql.connector.Error as err:
    return make_response(jsonify({'MySQLerror': str(err)}), 500)
return make_response(jsonify(response), 200)
```

Listing 6.4. GET /categories implementation

After the `@app.route` decorator explained in the previous section, there is another one: `@jwt_required`. This decorator is imported from the `flask_jwt_extended` extension for Flask [42], which provides methods to provide a user authentication mechanism based on JSON web tokens[3]. In more detail, when a user performs login, the backend generates a random string, the token, and returns it to the client application. Therefore, client has to include the token in the HTTP "Authorization" header every time it sends a request which requires user to be logged in. The token expires after fifteen minutes, which is the default value, and in this case or when the user performs logout it is inserted in a blacklist to prevent user accesses with an outdated token. The method which checks if a token is valid or has been blacklisted is shown below and it is called whenever the `@jwt_required` decorator is associated to an API method.

```
@jwt.token_in_blacklist_loader
def check_if_token_in_blacklist(decrypted_token):
    jti = decrypted_token['jti']
```

[3] *JSON Web Tokens* are an open, industry standard RFC 7519 [43] method for representing claims securely between two parties.

```
connection = mysql.connector.connect(user=db_username,password=
    db_password,host=db_host,port=3306, database=db_database)
cursor = connection.cursor(buffered=True)
query = """SELECT * FROM blacklist WHERE jti = %s"""
data = (jti,)
resp = False
try:
    cursor.execute(query, data)
    if cursor.rowcount > 0:
        resp = True
    connection.commit()
    cursor.close()
    connection.close()
except mysql.connector.Error as err:
    return make_response(jsonify({'MySQLerror': str(err)}), 500)
return resp
```

Listing 6.5. Token validity check implementation

Another useful extension used in backend implementation is the Python `mysql_connector` one [44], which allows to query database using the SQL syntax and to simply manage transactions.

The *templates* folder contains a set of JSON files used to build the API response bodies and the request bodies to send CFS Orders to Service Resolver. The use of templates avoid useless information processing and is offered by the `render_template` Python extension.

Finally, the *swagger* folder contains the JSON and YAML[4] files providing backend API Swagger documentation.

The backend configuration is detailed in the next section.

6.2.2 Frontend code

Frontend code is organized as shown in figure 6.3. The *src* folder contains all JavaScript code, while all the static components are placed in the *public* folder. Inside the *src* folder there are all the files defining the React main components, along with two directories named *components* and *assets*. The first one contains the subcomponents to be rendered inside the higher level ones, while static assets such as images are placed in the *assets* directory.

[4]YAML stands for *YAML Ain't Markup Language* [45] and it is a human friendly data serialization standard for all programming languages.

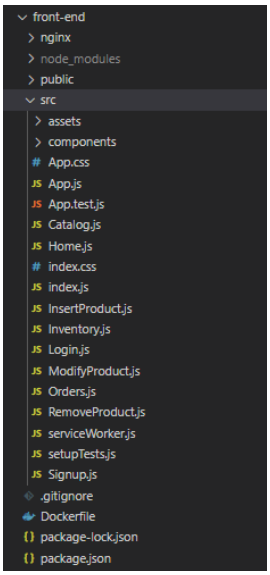


Figure 6.3. Service Portal frontend code's organization

Outside the *src* folder, the *package-lock.json* file lists all npm[5] dependencies to be installed in order to make the application working.

The React application entry point is the *index.js* file. In particular, it renders the App component imported from the *App.js* file inside the HTML root element defined in the *index.html* file, located in the *public* folder.

The *App.js* file specifies the main components to be rendered based on user actions. When the application starts, the default component rendered in the browser is the Home one, defined in *Home.js* file. The list of available products, Inventory items and performed orders are rendered respectively in *Catalog.js*, *Inventory.js* and *Orders.js* files. In the same way, the *InsertProduct.js*, *ModifyProduct.js* and *RemoveProduct.js* defines the views to insert, modify and remove a product. All these components are imported in the *App.js* file, which renders them dynamically, exploiting the `react-router-dom` extension, which is the most popular routing library for React. In particular, page navigation in a React applications can be implemented by using a Router. This allows to define the path to reach each page of the application, the rules to be applied to establish if a user is allowed to access a given page and the eventual redirect action to be performed to block unauthorized access. The following example shows the usage of React Router inside the render method in *App.js* file.

[5]The therm npm stands for *Node Package Manager* [46] and it is a Software Registry containing a huge number of JavaScript libraries, allowing developers to simply download and install it from command line.

```
<Route path="/insertproduct" render={() => (
    this.state.admin === "true"
    ? <InsertProduct loggedin={this.state.loggedin} token={this.state.
        token} handleLogout={this.handleLogout} />
    : <Redirect to="/"/>
)} />
```

Listing 6.6. Example of React Router usage

As described before, components have been implemented by using the `react-bootstrap` extension [47]. This library allows to write HTML pages using HTML elements provided by the Bootstrap framework. An example is the "NavigationBar" component defined in the *NavigationBar.js* file in the components folder. Its code is shown below.

```
function NavigationBar(props) {
    return(
        <Navbar variant="dark" expand="lg" fixed="top" style={navstyle
            }>
            <Container>
                <Navbar.Brand href="">Service Portal</Navbar.Brand>
                <NavBar loggedin={props.loggedin} handleLogout={props.
                    handleLogout}/>
                <Nav className="ml-auto" as="ul">
                    <Nav.Item as="li">
                        <img src={logo} alt=""/>
                    </Nav.Item>
                </Nav>
            </Container>
        </Navbar>
    );
}
```

Listing 6.7. Example of React component implemented using Bootstrap

Finally, the JavaScript `fetch` function [48] has been used to perform asynchronous HTTP requests to Service Portal backend.

6.3 Configuration

Service Portal components have been configured to be runned as Docker containers, allowing its deployment in virtual environments managed inside a Cloud-based

infrastructure. Therefore, the application has been implemented in a Cloud-native perspective, with the possibility to be installed in Multi-Cloud Platforms, since all major cloud computing providers, including Amazon Web Services (AWS) and Google Compute Platform (GCP), have added Docker's support. Moreover, Docker is also supported by other Cloud providers like Microsoft Azure and OpenStack.

Docker containers are started by means of a single command to be typed on terminal, by using Docker Compose [49], which is detailed in the next section. Additional configurations regards NGINX[6] and communication among the backend and Service Resolver.

Therefore, the above mentioned technologies and the motivation they have been used for Service Portal implementation are presented in more detail.

6.3.1 Docker and Docker Compose

A key benefit of using Docker is the isolation. Docker ensures that each container has its own resources that are isolated from other containers. This means that Service Portal components can run independently without affecting the execution of the other ones. In addition, this prevents components to exceed resources usage with respect to the ones individually assigned to them, avoiding performance degradation.

Therefore, a Dockerfile is provided for backend, frontend and database. A Dockerfile is a text document containing all the commands to be executed on the command line to create the Docker container and to install and run an application inside it. The Dockerfile of the backend component is shown below.

```
FROM python:3.7.1

ENV PYTHONDONTWRITEBYTECODE 1
ENV FLASK_APP "app.py"
ENV FLASK_ENV "development"
ENV FLASK_DEBUG True

RUN mkdir /app
WORKDIR /app

RUN pip install --upgrade pip

# We copy just the requirements.txt first to leverage Docker cache
COPY requirements.txt /app/requirements.txt

RUN pip install -r requirements.txt
```

[6] NGINX [50] is a web server for application deployment. More details are provided in section 6.3.2.

```
ADD . /app

EXPOSE 5000

CMD flask run --host=0.0.0.0
```

Listing 6.8. Backend Dockerfile

In particular, the FROM directive specifies the image to be download to set up the container. In particular, a Docker image is a file used to execute code in a Docker container. In this case, the container is created using an image with Python version 3.7.1 installed on it. Then, the ENV directive allows to set environment variables for the container, while the RUN one specifies commands to be launched from terminal. The EXPOSE directive is used to make a given port reachable from outside the container, in this case the specified port number is 5000. Finally, the CMD argument defines how the application installed inside the container has to be runned.

All Docker containers composing Service Portal are built and started by typing only two commands on terminal, using Docker Compose. The Compose tool of Docker allows to define and run multi-container Docker applications by simply writing a YAML file to configure application's services. The Docker Compose file used to configure Service Portal is shown below.

```
version: '3'
services:
  app:
    build: ./back-end/
    ports:
      - "5000:5000"
    links:
      - db
  db:
    build: ./db/
    ports:
      - "3306:3306"
    environment:
      MYSQL_DATABASE: 'serviceportal'
      # So you don't have to use root, but you can if you like
      MYSQL_USER: 'user'
      # You can use whatever password you like
      MYSQL_PASSWORD: 'password'
      # Password for root access
```

```
      MYSQL_ROOT_PASSWORD: 'password'
    expose:
      # Opens port 3306 on the container
      - '3306'
  frontend:
    build: ./front-end/
    ports:
      - "80:80"
    links:
      - app
```

Listing 6.9. Service Portal docker-compose.yml

In this case, under "services" all the components are defined and a name is assigned to them. Then, for each component the location of Dockerfile is specified and through the "links" option network configuration to allow inter-container communication is assigned.

6.3.2 NGINX

Frontend component has been deployed on NGINX web server, running inside a dedicated Docker container. NGINX is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server and it is used to serve static and index files, to perform autoindexing and to manipulate cache. Moreover it provides accelerated reverse proxying with caching, load balancing and fault tolerance capabilities.

Regarding Service Portal, NGINX usage is due to its high performance in serving static file and to the possibility to exploit its proxy capabilities to hide frontend React application and to allow communication between backend and frontend. All the required settings to implement that have been written in the *nginx.conf* file inside the frontend folder of the project. In particular, the server listens on port 80 at address 0.0.0.0, which is bound to any address. Then, requests from browser to frontend are redirected to the React application running on port 3000, while requests from frontend to backend are redirected to port 5000 at the IP address of the machine on which Service Portal is installed. This setting is specified by means of the `proxy_pass` directive [51].

6.3.3 Communication between Service Portal and Service Resolver

In order to communicate with Service Resolver, Service Portal backend has to know the IP address of the machine on which Service Resolver is running. It is also possible to install both applications on the same machine and in this case a different Docker Compose file must be used, in which network configuration allows all Service Portal containers and Service Resolver containers to communicate sharing the

same network by using container names.

In both cases, the URLs to reach Service Resolver CFS Order API and CFS Specification API are written in the *backend.conf* file, which can be edited as needed. Thus, in the first case the machine IP address has to be used, while in the second case it is sufficient to specify Service Resolver container names.

Chapter 7

Service Portal: validation

This chapter sums up Service Portal main requirements and the solutions proposed to satisfy them. Moreover, application performance is analysed based on significant parameters, measuring the quality of user experience, and corresponding values got during testing activity.

7.1 Functional requirements

Functional requirements refer to the list of functionalities to be implemented in Service Portal.

REQUIREMENT	PROPOSED SOLUTION
Declarative UI	React allows to define HTML elements following a declarative approach, so that they can autonomously react to state changes of the components themselves.
Cloud Native	The application is executed in Docker containers.
Pagination of data	Backend API offers dedicated methods to get single pages of data from database, according to the subset of data to be displayed.
Model-driven implementation	Adoption of a single model to define all products, such that the application is independent from the particular type of product, and of Service Resolver models in order to perform mapping operations when CFS Orders has to be sent.
Easy to configure and start	The application can be runned through a single Docker Compose command and works properly with Service Resolver installed on the same machine or on a different machine, thanks to the possibility to provide the desired configuration as explained in the previous chapter.
Possibility to configure products from GUI	Possible thanks to role-based management of users.
Possibility to modify data stored in database	Some data can be modified from GUI, but there are also additional API methods which allow to modify data thorugh clients like Postman, without the need to query directly the database.
User authentication	Provided by a mechanism based on JSON web tokens.

Figure 7.1. Functional requirements

Table in figure 7.1 describes them and the corresponding proposed solutions. All

mentioned requirements have already been discussed in previous chapters, but some points need to be underlined. In particular, backend API offers additional methods with respect to the ones called by the React application. Thanks to them, administrators have the possibility to perform not only products management, but also to access, modify or delete data about users, orders, Inventory and products categories. This feature makes Service Portal a complete application from all points of view, since it has been designed to satisfy all user needs as better as possible and it is well documented.

7.2 Non functional requirements

Non functional requirements are related to how Service Portal functionalities have been implemented. They are listed in figure 7.2.

REQUIREMENT	PROPOSED SOLUTION
Robustness of transactions, simple data management and possibility to retrieve and update data quickly	Adoption of a relational database.
Effectiveness in rendering graphical components	React updates the DOM tree only when needed, according to components state changes.
Implementation of a robust backend service	Management of possible errors and their description provided in API responses following REST design practices
Possibility to load quickly application static contents from the browser	Frontend application deployed on NGINX web server.
Well-documented API	Swagger documentation provides a detailed description of backend API.
Communication between frontend and backend modules independent from the particular machine on which Service Portal is installed	Usage of NGINX proxy capabilities with the possibility to specify backend IP address and port to which all client requests have to be redirected.

Figure 7.2. Non functional requirements

This proves that Service Portal has been implemented following some key principles. In particular, backend Flask application is robust and it has been developed following REST best practices. On the other side, frontend React application provides an optimized rendering of HTML pages, combined with NGINX web server high performance in serving static files. Finally, the MySQL database guarantees reliability and data availability and optimizes time to retrieve and manipulate data.

7.3 Performance analysis

In this section, some tests performed in order to analyse Service Portal performance are presented. These tests aim to analyse user experience when executing actions through the GUI and regard the most critical operations in therms of time. Since ordering products requires Service Portal to communicate with Service Resolver, the most expensive operations are the purchase of a new service and the removal of an active one.

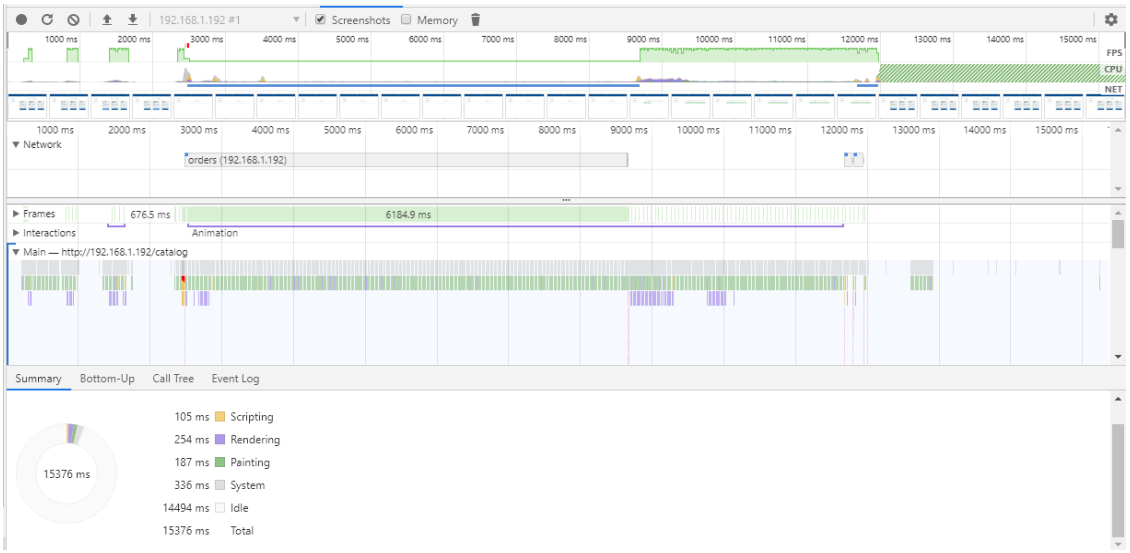


Figure 7.3. Performance profile for product order (add)

For what concerns product ordering corresponding to a new service instance creation, user actions have been recorded for about 15 seconds, using Google Chrome Developer Tools [52], also known as *DevTools*, which provide web authoring and debugging tools built right into the browser. Test result is shown in figure 7.3. The same test has been performed also for the delete order with a duration of about 11 seconds, and the obtained profile is shown in figure 7.4. In particular, the values obtained are the following ones:

- CPU time for rendering operations: 254 milliseconds and 194 milliseconds
- CPU time for scripting operations: 105 and 230 milliseconds, respectively
- CPU time for painting operations: 187 and 89 milliseconds
- Frames Per Second (FPS): around 60/70 FPS at most
- Network: in this case obtained values are not relevant since they do not depend on the application itself

The FPS indicator provides a measure of animations efficiency. For responsive web pages, the value recommended is exactly the one obtained also in these tests: around 60 fps. Also HTML pages rendering is optimal and added to scripting and painting times produces a good response-time, since it is under the threshold of 1 second recommended by HCI[1] guidelines [54]. Finally time to complete API requests

[1] *Human Computer Interaction* (HCI) [53] is an academic discipline which studies the interaction between humans (the users) and computers.

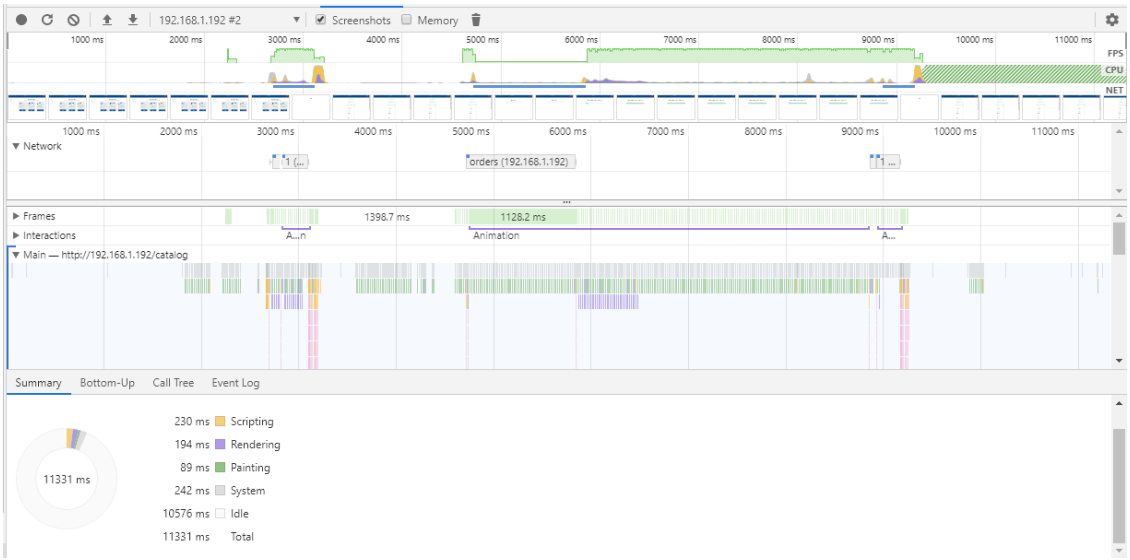


Figure 7.4. Performance profile for product order (delete)

is not relevant since it depends on network performance.

In conclusion, Service Portal offers a good overall user experience, which combined with advantages of using NGINX and the fact backend and database allow fast request processing, data retrieval and manipulation, give the application a good performance level.

Chapter 8

Conclusions and Future work

8.1 Conclusions

The work performed in this thesis can be considered as a confirmation of the fact that CSPs have started a revolution process regarding their network infrastructure and the way in which services are delivered to customers. Moreover, it emphasizes the importance of studying new technologies according to the change of customers' requests. This can be translated in the need of understanding the potentiality of new paradigms such as SDN and NFV and how they will affect the future of networks, as well as the spread of Cloud-based technologies.

The main result reached with this thesis work is the practical contribution given to two Telecommunications Provider companies to support their evolution process. To sum up, the improvement of Service Resolver for ONAP was a key achievement since it was not working correctly before. For what concerns Service Portal, it represents a model of how a BSS application working at Product layer should be implemented. Moreover, it is worth to remark that its proof-of-concept is a consequence of provider's specific needs, but the fact that its implementation is compliant with the new dominant technologies represents an added-value feature.

In conclusion, the target of understanding CSPs needs, analysing the technologies leading their transformation and helping them to innovate their infrastructure was satisfied.

8.2 Future work

This work can be further extended based on continuously evolving providers' needs. In particular, both Service Resolver and Service Portal can be improved.

Regarding Service Resolver, as already explained in the corresponding chapter, it is not yet mature to be used in production environment. Moreover, it can be also enriched with new features to allow BSS applications at Product layer to exploit better ONAP capabilities. A possible implementation of a HUB API and the support

of E2E instantiation method are an example.

For what concerns Service Portal, a possible improvement is backend API extension to be compliant to TMF specifications about Product Ordering, Product Catalog and Product Inventory management. Moreover, users management is simplified for now, but in order to use the application to satisfy customers' request from an entire country, customer management should be performed by an application working at eTOM Customer layer and it should be integrated with the BSS system handling products.

To sum up, thanks to their design, Service Resolver and Service Portal offer the possibility to be easily extended in case of future necessity, following a CI/CD[1] approach.

[1] CI/CD stands for *Continuous Integration, Continuous Delivery* and it is a process for continuous development, testing, and delivery of new code [55].

Bibliography

[1] *Cloud Central Office Reference Architectural Framework*, BroadBand Forum, January 2018

[2] BroadBand Forum, URL: https://www.broadband-forum.org/

[3] TM Forum's Business Process Framework (eTOM), URL: https://www.tmforum.org/business-process-framework/

[4] TeleManagement Forum, URL: https://www.tmforum.org/

[5] TM Forum Open API program, URL: https://www.tmforum.org/open-apis/

[6] Open Network Automation Platform (ONAP), URL: https://www.onap.org/

[7] ONAP Virtual Infrastructure Manager (VIM), URL: https://docs.onap.org/en/elalto/submodules/multicloud/framework.git/docs/MultiCloud-Architecture.html

[8] *Network Functions Virtualisation (NFV); Management and Orchestration*, NFV ETSI Industry Specification Group (ISG), 2014

[9] ONAP Architecture, URL: https://docs.onap.org/en/elalto/guides/onap-developer/architecture/onap-architecture.html

[10] Docker official documentation, URL: https://docs.docker.com/

[11] Kubernetes official web site, URL: https://kubernetes.io/

[12] ONAP Service Design and Creation (SDC) project, URL: https://wiki.onap.org/display/DW/Service+Design+and+Creation+%28SDC%29+Portal

[13] OASIS TOSCA Technical Committee, URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca

[14] ONAP SDC CSAR Structure, URL: https://wiki.onap.org/display/DW/Csar+Structure

[15] ONAP Service Orchestrator (SO) project, URL: https://wiki.onap.org/display/DW/Service+Orchestrator

[16] ONAP Virtual Infrastructure Deployment (VID) project, URL: https://wiki.onap.org/display/DW/VID

[17] ONAP Service Orchestrator instantiation flow, URL: https://wiki.onap.org/display/DW/ARCHCOM%3A+InfoFlow+-+SO+Service+Instantiation+Flow

[18] Business Process Model and Notation (BPMN) specification, URL: https://www.omg.org/spec/BPMN/2.0/PDF, January 2011

[19] ONAP Active and Available Inventory (AAI, A&AI) project, URL: https://wiki.onap.org/display/DW/Active+and+Available+Inventory+Project

[20] ONAP Data Movement as a Platform (DMaaP) project, URL: https://wiki.onap.org/display/DW/Data+Movement+as+a+Platform+Project
[21] Apache Kafka official documentation, URL: https://kafka.apache.org/
[22] ONAP External APIs (NBI) project, URL: https://wiki.onap.org/display/DW/External+API+Framework+Project
[23] ONAP External APIs documentation, URL: https://docs.onap.org/en/dublin/submodules/externalapi/nbi.git/docs/offeredapis/offeredapis.html
[24] ONAP External APIs ServiceCatalog API documentation, URL: https://docs.onap.org/en/elalto/_downloads/b48637d161b9ca4807c181140d7405a0/swagger.json
[25] ONAP External APIs ServiceOrder API documentation, URL: https://docs.onap.org/en/elalto/_downloads/c5cff8a7b7fb8de1902794f57d4adf0f/swagger.json
[26] ONAP External APIs ServiceInventory API documentation, URL: https://docs.onap.org/en/elalto/_downloads/ac240fa96f8a911b357f7ad955aef3a2/swagger.json
[27] *Service Ordering Management API REST Specification*, TM Forum, April 2019
[28] *Service Inventory Management API REST Specification*, TM Forum, April 2019
[29] *Service Catalog Management API REST Specification*, TM Forum, April 2019
[30] Service Resolver for ONAP public repository, URL: https://gitlab.com/Orange-OpenSource/lfn/onap/service-resolver
[31] Python Connexion official documentation, URL: https://connexion.readthedocs.io/en/latest/
[32] Open API Specification, URL: http://spec.openapis.org/oas/v3.0.3
[33] MongoDB official web site, URL: https://www.mongodb.com/
[34] Service Resolver installation guide, URL: https://gitlab.com/Orange-OpenSource/lfn/onap/service-resolver/blob/master/install_and_run.md
[35] Swagger official web site, URL: https://swagger.io/
[36] MySQL official web site, URL: https://www.mysql.com/
[37] Flask official documentation, URL: https://flask.palletsprojects.com/en/1.1.x/
[38] PEP333 WSGI Specification, URL: https://www.python.org/dev/peps/pep-3333/
[39] React official web site, URL: https://reactjs.org/
[40] JavaScript XML (JSX) official documentation, URL: https://jsx.github.io/doc.html
[41] Bootstrap official web site, URL: https://getbootstrap.com/
[42] Flask-JWT-Extended's documentation, URL: https://flask-jwt-extended.readthedocs.io/en/stable/

[43] RFC 7519, JSON Web Token (JWT), URL: https://tools.ietf.org/html/rfc7519
[44] Python mysql-connector-python project, URL: https://pypi.org/project/mysql-connector-python/
[45] The Official YAML Web Site, URL: https://yaml.org/
[46] npm Documentation, URL: https://docs.npmjs.com/
[47] React Bootstrap official documentation, URL: https://react-bootstrap.github.io/
[48] JavaScript fetch documentation, URL: https://github.com/github/fetch
[49] Docker Compose official documentation, URL: https://docs.docker.com/compose/
[50] NGINX official web site, URL: https://www.nginx.com/
[51] NGINX official documentation: URL: https://nginx.org/en/docs/
[52] Chrome DevTools, URL: https://developers.google.com/web/tools/chrome-devtools
[53] *HCI and Usability for e-Inclusion*, 5th Symposium of the Workgroup Human-Computer Interaction and Usability Engineering of the Austrian Computer Society, USAB 2009, Linz, Austria, November 9-10, 2009, Proceedings, Springer 2009.
[54] Jakob Nielsen, *Website Response Times*, June 2010, URL: https://www.nngroup.com/articles/website-response-times/
[55] Isaac Sacolick, *What is CI/CD? Continuous integration and continuous delivery explained*, January 2020