Semantically Enriched API Descriptions for Improving Service Discovery in Cloud Environments

Mainas Nikolaos

School of Electrical and Computer Engineering
Technical University of Crete

A thesis submitted in fulfillment of the requirements for the degree of Master of Science.

Committee
Supervisor: Euripides G.M Petrakis, Professor
Stelios Sotiriadis, Research Fellow
Stavros Christodoulakis, Emeritus Professor

June 2017

Abstract

The rapid growth and development of Cloud Computing has allowed the implementation of scalable applications and services in lower costs based on the principles of Service-Oriented Architecture (SOA). The increasing interest in Cloud Computing solutions, has led to the proliferation of Cloud service offerings over the Internet by multiple Cloud providers. The need for efficient and accurate service discovery based on user needs has become a significant challenge. Cloud services are mainly offered by means of Web services based on the REST architecture style and need to be formally described in a way that is both understandable by humans and machines. In this work, we propose the adoption of the OpenAPI Specification (OAS), a simple and yet powerful specification for the description of REST APIs, as the description language of Cloud services. In addition, we present a set of extension properties, forming the Semantic OpenAPI Specification (SOAS), that semantically enrich the OAS service descriptions, so as to be understandable by both humans and machines. Finally, we demonstrate that is plausible to transform SOAS descriptions to ontologies as this enables application of state-of-the-art querying languages (e.g. SPARQL) for service discovery and of reasoning tools for detecting inconsistencies and inferred relationships in SOAS descriptions.

Acknowledgments

This thesis would not have been possible without the help of several people who contributed in the preparation and completion of this study.

Firstly and foremost I would like to thank my advisor, Professor Euripides G. M. Petrakis for his constant supervision. For guiding, advising and supporting me in every step of this thesis. I am grateful for giving me the opportunity to work on this very interesting field of research.

Also, I would like to thank Professor Stavros Christodoulakis and Dr. Stelios Sotiriadis who agreed to evaluate my thesis.

Moreover, I would like to thank my laboratory colleagues for their patient and constructive comments.

Most of all, I would like to thank my family for their enormous help, understanding and support.

Contents

List of Figures

List of Tables

1

Introduction

The rapid growth of the World Wide Web (WWW) in recent years has drastically affected many aspects of our life. The Web provides access to the world's information and enables communication in matter of seconds. According to big data info-graphic contributed by Ben Walker [40], 2.5 quintillion bytes of data are being created every day. In order to use and analyze this vast amount of data, systems need to directly communicate to each other. Services became the prevalent option, as they allow systems to remain independent and self-contained. More and more organizations build their systems based on the principles of Service-Oriented Architecture (SOA) [15], in which every service offers a specific functionality.

Service-Oriented Architecture (SOA) is an evolution of distributed computing based on the request/reply design paradigm for synchronous and asynchronous applications. An application's business logic or individual functions are modularized and presented as services for consumer/client applications. A service, in most cases, is implemented as a Web Service [23] due to its loosely coupled nature i.e. the service interface is independent of its implementation. Every service needs to introduce itself to other services in order to be able to use it by making its API and functionality public and accessible to others. Service description languages have been introduced to describe the requirements for establishing a connection with a service as well as message formats to successfully communicate with it.

In today's highly competitive environments, organizations are striving to

manage the cost of infrastructure and information architecture. Cloud Computing [33], a new paradigm of computing, has emerged allowing users to gain access over unlimited computing resources that could be managed effectively and more efficiently. Individuals and organizations can make use of scalable IT infrastructures in lower costs, while processing power can be accessed based on demand and on budget allowance.

SOA and cloud computing are complementary technologies as they both attempt to increase the agility and cost savings of software development and maintenance. Typically, Cloud services are offered by means of Web Services and thus they are service-oriented. Therefore, systems driven by SOA design principles, may include Cloud services as part of their architecture. Moreover, Cloud providers offer environments that integrate SOA technologies, such as service management and orchestration platforms as well as service registries, for the development of services or applications that fully exploit Cloud Computing capabilities. As the number and diversity of cloud providers is increasing the need for standardizing technologies for publishing their services to developers is becoming of crucial importance for their adoption and market success.

1.1 Motivation

The increasing interest in Cloud Computing solutions, has led to the proliferation of Cloud service offerings over the Internet by multiple Cloud providers. Cloud providers (such as Amazon and Microsoft) offer services that range from software to hardware through Web interfaces or APIs (Application Programming Interface). The need for efficient and accurate service discovery based on user needs has become a significant challenge [39].

In order to increase the adoption of cloud services by software developers and enterprises, these must be accompanied by appropriate service descriptions. Typically, Cloud providers describe their services in plain text (such

as the FIWARE catalogue[1]), which users have to browse and read in order to determine whether a service meets their needs. However, text descriptions of services are mainly intended to be readable by humans and not by machines, while in some cases are inaccurate or vague. Cloud services need to be formally described in a way that is both understandable by humans and machines. The last requirement would not only improve the accuracy of service descriptions but also, would allow for services to be discovered by other services and also orchestrated in composite services or applications e.g. in the example of AWS formation services[2].

1.2 Problem Definition

One of the key components in any service-oriented architecture is service discovery. It is an active area of research, as the increasing amount of services imposes suitable discovery and selection capabilities for the identification of the services of interest. For instance, ProgrammableWeb[3], the most well-known Web service directory, has registered more than 15000 public services, not considering the amount of private services created by organizations.

Extensive research has been conducted over the years [14] in order to provide tools and mechanisms that would efficiently describe any aspect of a service (functional and non-functional). UDDI (Universal Description Discovery and Integration) [11], WSDL (Web Service Description Language) [9] were introduced as a first approach towards the syntactic description of services. SAWSDL [16], OWL-S [32] and other approaches were proposed as an effort to enrich the existing service descriptions with semantics based on Semantic Web technologies. However, as new technologies and architectural styles (REST [18]) emerged, many of these approaches became obsolete. The need for better service descriptions and consequently effective discovery

[1] https://catalogue.fiware.org/
[2] https://aws.amazon.com/cloudformation/
[3] http://www.programmableweb.com/

fueled new research efforts such as WADL (Web Application Description Language) [24], Hydra [30], OpenAPI Specification (OAS) [35].

The focus of this work is on improving the description of Cloud services, and consequently allow the implementation of efficient and accurate service discovery mechanisms. Cloud services need to be described in a way that eliminates ambiguities and provides descriptions which are both uniquely defined and discoverable. Therefore, a description language is required that would allow for both syntactic and semantic description of Cloud services. To achieve it, the existing description languages need to be reviewed, analyze their features and characteristics in order to determine their suitability for the description of Cloud services.

1.3 Proposed Solution

Our research effort resulted in the adoption of the OpenAPI Specification (OAS) as the main description language for Cloud services. OAS (formerly known Swagger) is a simple and yet powerful specification for the description of REST APIs, as it provides both human-readable and machine-readable descriptions. Given an OpenAPI service description, a consumer client is able to understand and discover the functionality of a service, as well as to interact with it with a minimum implementation logic. In fact, OAS is a complete framework providing tools for interactive documentation (Swagger Editor[4] and Swagger UI[5]) and client SDK generation (Swagger Codegen[6]).

Despite its capabilities, OAS is mainly focused to human-readable service descriptions. In order for a machine to understand the meaning of an OpenAPI service description, a service description need to be formally defined and its content be semantically enriched. This work proposes that OAS service description can be semantically annotated by associating OAS entities

[4] https://github.com/swagger-api/swagger-editor
[5] https://github.com/swagger-api/swagger-ui
[6] https://github.com/swagger-api/swagger-codegen

to entities of an ontology (e.g. domain ontology). The new approach, henceforth referred to as Semantic OAS (SOAS) eliminates any ambiguities in the original OAS descriptions and produces service descriptions that are understandable by both humans and machines. Moreover, this work suggests that is plausible to transform SOAS descriptions to ontologies as this enables application of state-of-the-art querying languages (e.g. SPARQL) for service discovery and of reasoning tools (e.g. Pellet) for detecting inconsistencies and inferred relationships in SOAS descriptions.

1.4 Contributions of the Work

The major contributions of this thesis can be summarized as follows:

- Provides a comprehensive review of approaches for service description and of related technologies including SOA, cloud and semantic Web technologies. This review provides a critical analysis of each technology and highlights its characteristics that restrict its adoption or limit its usefulness in providing unambiguous and machine readable service descriptions.
- Proposes Semantic OpenAPI Specification (SOAS) as the description language for Cloud services. SOAS is an extension to the OpenAPI Specification (OAS) that semantically enriches service descriptions in order to eliminate any ambiguities and offer descriptions readable by both humans and machines.
- Describes a mechanism for transforming SOAS service descriptions to ontologies so as to benefit from semantic web tools such as reasoners and query languages for service discovery and for enabling service orchestration.
- Demonstrates how SOAS can be applied to describe FIWARE services (as the FIWARE catalogue of services is available in text only) following the example of ORACLE which first provided the API Catalog Cloud Service as a collection of machine-readable OpenAPI descriptions, rep-

resenting some of Oracle's most popular SaaS and PaaS applications.

1.5 Thesis Outline

Chapter 2 provides a brief introduction to basic concepts and technologies which are used throughout the thesis. In addition, it summarizes and presents the most common approaches for the description of Cloud services. In Chapter 3 our proposed solution for the description of Cloud services, the Semantic OpenAPI Specification, is presented. At first, we discuss the reasons the led us to the adoption of OpenAPI Specification. Then, we introduce the extension properties that allow OpenAPI service description to be semantically enriched. Moreover, we describe a mechanism for transforming SOAS service descriptions to ontologies. Chapter 4 demonstrates how our proposed solution can be applied in a Cloud provider, such as the FIWARE platform. Finally, conclusions and issues for future work are discussed in Chapter 5.

2

Background and Related Work

2.1 Service-Oriented Architecture (SOA)

Service-oriented architecture (SOA) represents an architectural model that aims to enhance the efficiency, agility, and productivity of an enterprise by designing, developing, deploying and managing systems, based on service-orientation [15]. Service-orientation is a design paradigm that suggests that all functional components of a system are viewed as services that communicate with each other through well defined interfaces by message passing.

A service is a function that is well-defined, self-contained, and independent of other services. The invoker of a service need to be aware of its interface only and not its implementation. SOA as an architectural model doesn't impose a specific technology for the communication of services. With the advent of machine communication protocols such as HTTP and data formats such as XML, RDF and JSON, SOA is becoming the mainstream approach for building distributed systems (communicating systems in general) in terms of communicating services.

The following principles provides general guidance for the design and development of services and Service Oriented Architectures:

- *Standardized Service Contract*
  A service contract is a specification or description that provides infor-

mation about a service. It defines information about service endpoint, service operations, the exchanged message formats and the conditions which need to be fulfilled before the service can be executed

- *Service Loose Coupling*
  Service minimizes dependencies between service consumers and the underlying service logic and implementation.
- *Service Abstraction*
  Services hide the logic they encapsulate from the outside world
- *Service Reusability*
  Services are reusable resources and thus can be used in multiple solutions.
- *Service Autonomy*
  Services are independent and control the functionality they encapsulate.
- *Service Statelessness*
  Services minimize retaining information specific to an activity.
- *Service Discoverability*
  Services are designed to be outwardly descriptive so that they can be found and accessed via available discovery mechanisms
- *Service Composability*
  Collections of services can be coordinated and assembled to form composite services.

A SOA building block (i.e. a service) can take one of the following roles according to the "*find-bind-execute*" system design paradigm:

- *Service registry*, which is a repository containing service contracts (descriptions) for service consumers to know how services may be accessed.
- *Service Providers*, who build and execute services. Moreover, they are responsible to register services in Service registries.
- *Service Consumers*, who search for services in Service registries based on some criteria and when found a dynamic binding is performed.

2.2 SOAP-based Services

Initially, the Web services technology platform was built on existing standards such as, Extensible Markup Language (XML), Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description Discovery and Integretion (UDDI). XML [7] was selected, due to its popularity at the time, as the main data format for machine to machine communication.

SOAP [12] designed by Microsoft in 1998, specifies a messaging framework consisting of an XML-based message format, a protocol binding specification, and a mechanism for specifying additional application-level features. In SOAP-based services, all messages are typically envelopes that contain header and body elements. Figure 2.1[1] illustrates the structure of a SOAP message. An *envelope* defines the SOAP message, an XML document with a start and end point so that the receiver knows where the message starts and where it ends. A *header* is an optional element which contains application specific information about the message. Finally, the *body* is a mandatory element that contains the application-defined XML data being exchanged in the SOAP message, that is the information sent to the receiver. An important characteristic of SOAP-based services is the ability to leverage different transport protocols, including HTTP and SMTP, as well as others.

WSDL [9] is a XML-based interface description language that is used for describing the functionality offered by a web service (Section 2.6.1) e.g. in a UDDI registry. UDDI [11] was introduced as a registry for the discovery of services based on their WSDL description files. Figure 2.2[2] demonstrates the role of each component in a original Web Service technology platform. A service provider describes its service using WSDL before it is published in a service registry, like UDDI (1). A service consumer issues queries to the registry to locate a service (2). The WSDL description of the service

[1]Original figure from `https://en.wikipedia.org/wiki/SOAP`
[2]Original figure from `http://www.service-architecture.com/`

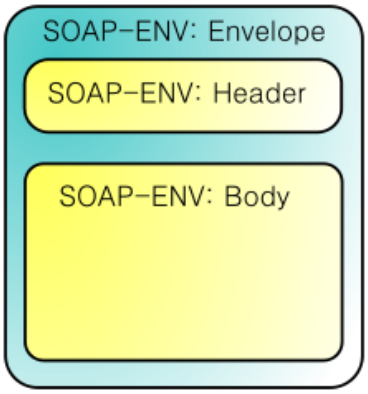

Figure 2.1: SOAP message structure

is passed to the service consumer, informing him how to communicate with the service (3). The service consumer uses the WSDL to send a request to the service provider (4) and receives the expected response by the service provider (5).

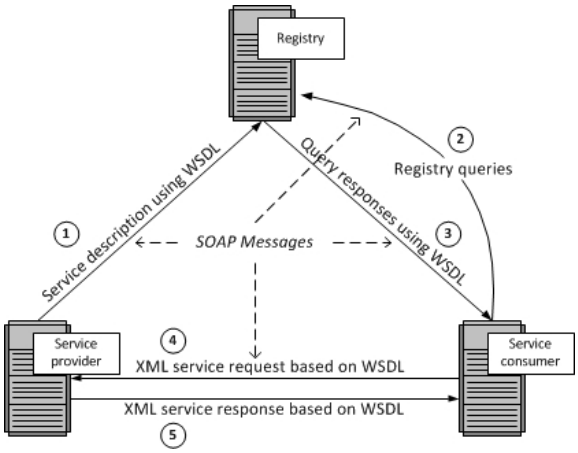

Figure 2.2: Web service basic architecture

2.3 REST-based Services

A different flavor of Web Services was inspired by Fielding's *Representational State Transfer* (REST) architectural style [18]. REST defines a set of archi-

tectural principles by which Web services are designed to focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages.

The key abstraction of information in REST is a resource. A resource is anything that is important enough to be referenced as a thing in itself [36], such as a document or image, a collection of other resources, a non-virtual object (e.g. a person). Resources can be static, like a book, or dynamic, like a weather report (i.e., it always changes, but still is a resource). REST uses a resource identifier (URI) to identify the particular resource involved in an interaction between components.

According to Fielding's doctoral dissertation, any Web Service considered RESTful is characterized by the following principles:

- *Client-Server*
  Client applications and server application must be able to evolve separately without any dependency on each other. This principle has a tight relation with the loosely coupled SOA principle, as discussed in Section 2.1
- *Stateless*
  Each request from a service consumer should contain all the necessary information in order to interact with the service, as the service doesn't store information from previous requests. The same principle is also found in SOA (Section 2.1).
- *Cacheable*
  Response messages from the service to its consumers are explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests. Caching brings performance improvement for both clients and server, as this reduces the number of requests a server has to manage, while clients may instantly reuse existing information without the need to send a new request and wait for server's response.
- *Uniform interface*

The uniform interface simplifies the service architecture, enabling every component of the service to evolve independently. Resources are identified using URIs, which are different from the representation (XML, JSON) sent to the client. The client can manipulate the resource representations provided they have the permissions. Every message sent between the client and the server is self-descriptive and includes enough information to describe how it is to be processed. Server's responses should include hyperlinks that a client can use in order to discover all the available actions and resources it needs (HATEOAS[3]).

- *Layered system*
  It allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting.
- *Code on demand*
  An optional constraint where the server temporarily extends the functionality of a client by the transfer of executable code.

REST has gained massive adoption, including Cloud Services, compared to other approaches, such as SOAP and WSDL. REST-based services are simpler to express, faster to process and make efficient use of bandwidth, as they don't require additional parsing for messages and are much less verbose than SOAP-based services. Unlike SOAP-based services, REST-based services are designed to be stateless and also enable caching that improves performance and scalability. Moreover, REST-based services may support multiple data formats such as XML and JSON, whereas SOAP-based services are limited to the use of XML.

However, the term REST has been misused as most Web services that claim to be RESTful (REST APIs) are not, in fact. In most cases services are based on the REST architecture, but often they violate the hypermedia constraint (HATEOAS). Fielding wrote a blog post[4] explaining that a service is considered RESTful if all REST principles are met. In order to describe

[3] `https://en.wikipedia.org/wiki/HATEOAS`
[4] `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`

services that are implemented incorporating the hypermedia constraint, the term *Hypermedia API* [1] has been emerged.

2.4 Cloud Computing

The US National Institute of Standards and Technology[5] (NIST) defines Cloud computing as "*a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three delivery models and four deployment models*" [33].

Essential Characteristics

Cloud computing characteristics that differentiate it from the traditional computing

- *On-demand self-service*
  A customer can easily access the required/necessary computing services without requiring human collaboration with the service providers
- *Broad network access*
  It provides the ability to access various cloud services from different devices, such as smartphones, desktops and other handheld internet capable devices.
- *Resource pooling*
  The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. A consumer may be using resources from across the world

[5] `https://www.nist.gov/`

without knowing the exact location, but with the exception of highly abstract information such as country, state or data-center.

- *Rapid elasticity*
  Consumers are able to purchase capabilities that can rapidly be scaled up or down, depending on the requirement. Often, services available to the consumer appear to be unlimited.
- *Measured service*
  Cloud systems automatically monitor and measure resource usage enabling this way the implementation of billing strategies per customer and per use of resources in certain periods of time.

Service Models

Cloud computing has been categorized into three models depending on the services provided by the cloud.

- *IaaS (Infrastructure as a Service)*
  Basic computing services are provided to a consumer, such as storage, networking and processing. A consumer is responsible for installing and maintaining the operating system and other software, but the responsibility of upgrading or maintaining the hardware resides to the provider. Examples of IaaS services are Amazon EC2[6] and Azure Virtual Machines[7]
- *PaaS (Platform as a Service)*
  A consumer is provided with an environment containing libraries, services and other tools through which he is able to deploy his own created or acquired applications onto the cloud infrastructure. A consumer doesn't control or manage cloud infrastructure, however he may be able to configure settings for the application hosting environment on a cloud. Examples of PaaS services are Google App Engine[8] and AWS

[6] https://aws.amazon.com/ec2/
[7] https://azure.microsoft.com/en-us/services/virtual-machines/
[8] https://cloud.google.com/appengine/

Elastic Beanstalk[9]

- *SaaS (Software as a Service)*
  A consumer is able to use the provider's applications running on a cloud environment. The applications are typically accessible through a thin client interface, such as a web browser. A consumer doesn't control either the underlying infrastructure or platform. However, there is a possibility to configure user-specific settings for the application in use. Examples of SaaS services are Google Docs[10] and Dropbox[11].

Deployment Models

There are four models for cloud computing service deployment, regardless of the service or delivery model adopted. These deployment models may have different derivatives which may address different specific needs or situations.

- *Private cloud*
  The cloud infrastructure is provided solely for a single organization's use. It may be operated, managed and owned by the organization, a third party or combination.
- *Community cloud*
  The cloud infrastructure is provided for restrictive use by a particular group of users from organization that have common concern (e.g., policy or mission). It may be operated, managed and owned by one or more organizations in the society or a third party.
- *Public cloud*
  The cloud infrastructure is made available to the general public for open use. In this deployment model, the cloud infrastructure may be operated, managed and owned by an academic institution, government, business or a combination.
- *Hybrid cloud*

[9] `https://aws.amazon.com/elasticbeanstalk/`
[10] `https://docs.google.com/`
[11] `https://www.dropbox.com/`

> The cloud infrastructure is a combination of two or more discrete cloud infrastructures, such as community, public or private, that remaim exclusive units. However, those units are bound together by the proprietary technology or the standardization that facilitates application and data portability.

2.4.1 SOA and Cloud Computing

SOA (Section 2.1) and Cloud Computing (Section 2.4) are technologies that can exist separately — neither depends on the other. However, the benefits of an integrated architecture with these two solutions is shown as a largely favorable strategy in the categories of cost, speed of development, ease of maintenance, and management.

Cloud Computing provides computational resources, including hardware and software, for the delivery and deployment of scalable applications and services. However, it doesn't impose any specific method for the efficient use and management of its offering services. SOA promises to fill this gap by providing guidelines, principles, and techniques for the development of applications and services, and strictly defines the architecture of service-oriented systems.

Cloud services are typically API or service-driven, and thus service-oriented. Cloud providers organize their services in directories or service registries, in order to enable discovery of services that best fit the needs of customers as well as reuse and better management of services.

2.4.2 Openstack

OpenStack[12] is an open-source cloud operating system platform for supporting the management of large pools of compute, storage, and networking re-

[12] `https://www.openstack.org/`

sources through a datacenter. The U.S National Aeronautics and Space Administration[13] (NASA) and Rackspace[14] were the initials contributors of the project, back in 2010. Nowadays, the community has over 200 participating members including companies like IBM[15], Intel[16], HP[17] and Red Hat[18].

OpenStack is comprised of several open-source sub-projects or services that manage its resources using either a web-based dashboard (provided by OpenStack), or through RESTful APIs (Section 2.3). Figure 2.3 demonstrates OpenStack's core services.

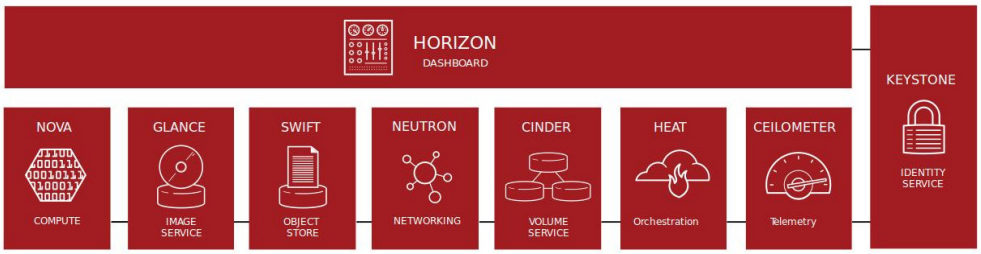

Figure 2.3: OpenStack Conceptual Architecture

- *Horizon* a web-based self-service portal for assisting user interaction with the underlying OpenStack services, such as launching an instance, assigning IP addresses and configuring access controls.
- *Nova* provides on-demand access to compute resources by providing mechanism for the management of large networks of virtual machines. A user is able to create and manage virtual servers using the machine images provided by *Glance*.
- *Glance* provides a system that manages disk and server images. In addition, the service has the ability to copy a server image and store it. These images can be used as templates to create new servers using *Nova* or backup running virtual machines.

[13] https://www.nasa.gov/
[14] https://www.rackspace.com/
[15] http://www.ibm.com/us-en/
[16] http://www.intel.com/content/www/us/en/homepage.html
[17] http://www8.hp.com/us/en/cloud/cloud-leadership.html
[18] https://www.redhat.com/en

- *Swift* provides a highly available, distributed, eventually consistent storage system in which users may store and retrieve arbitrary data objects.
- *Neutron* manages the virtual networks in the cloud infrastructure. Using this service a user may create and control its own networks and manage the IP addresses that are assigned to its virtual machines.
- *Cinder* provides the storage resources that are consumed by the virtual machines created by *Nova*. The service manages the creation, attachment and detachment of block storage devices to the virtual machines according to user's storage needs.
- *Heat* provides a template-based deployment orchestration for cloud applications. The service is similar to the AWS CloudFormation[19], allowing users to deploy and manage a service as a collection of services.
- *Ceilometer* monitors and collects usage data from OpenStack services in order to be used for billing, benchmarking, scalability, and statistical purposes.
- *Keystone* provides an authentication and authorization service for other OpenStack services. The service contains information regarding user authorization rights so that an OpenStack service will grant access only to the resources that each user owns or is authorized to access.

2.4.3 FIWARE

FIWARE[20] is a European initiative that aims to provide an open platform for developing and deploying Future Internet (FI) applications and services. The FIWARE platform is a cloud-based infrastructure built on top of OpenStack, providing a set of predefined basic services called Generic Enablers (GEs) which can be viewed as building blocks for an application. Generic Enablers are considered as software modules that offer various functionalities along with protocols and interfaces for their communication. Generic Enablers are

[19] `https://aws.amazon.com/cloudformation/`
[20] `https://www.fiware.org/`

implementations of open specifications of the most common functionalities that are provided by FIWARE and are stored in a public catalogue[21].

The FIWARE catalogue is organized into the following technical chapters:

- *Cloud Hosting* offers Generic Enablers that allow the creation and management of compute, storage and network resources. This chapter contains services which are mainly provided by Openstack, as well as Generic Enablers implemented by the FIWARE community such as the *SDC GE*[22] which allows the automatic deployment (installation and configuration) of software on running virtual machines.
- *Data/Context Management* aims at providing Generic Enablers that will ease the development and provisioning of innovative applications that require management and processing of context information (data which are relevant to a particular entity) as well as data streams in real-time and at massive scale. The *Orion Context Broker GE*[23] offers operations regarding the storage, subscription and management of context information. The *Complex Event Processing GE*[24] allows the analysis of data in real-time, detecting and reporting events that occur within a dynamic time window.
- *Internet of Things (IoT) Services Enablement* provides Generic Enablers that allow Things to become available, searchable, accessible, and usable resources to FIWARE-based applications enabling interaction with real-life objects. According to the FIWARE architecture[25] a Thing is any physical object, living organism, person or concept interesting from the perspective of an application and whose parameters are totally or partially tied to sensors, actuators or combinations of them.

[21] https://catalogue.fiware.org/

[22] https://catalogue.fiware.org/enablers/software-deployment-configuration-sagitta

[23] https://catalogue.fiware.org/enablers/publishsubscribe-context-broker-orion-context-broker

[24] https://catalogue.fiware.org/enablers/complex-event-processing-cep-proactive-technology-online

[25] https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/Internet_of_Things_(IoT)_Services_Enablement_Architecture

The services of the IoT chapter mainly act as gateways in which devices are registered to be discovered by other services in order to obtain access to the information they collect, such as the *IoT Discovery GE*[26].

- *Applications/Services Ecosystem and Delivery* offers a set of Generic Enablers that form a business framework through which service providers manage the entire lifecycle of their offerings. The *Business API Ecosystem GE*[27] allows users to publish their offerings (services, applications, data) and manage the pricing models, accounting and revenue settlement. On the other hand, the *Application Mashup GE*[28] provides a framework through which users without programming skills can easily create applications by combining services obtained from the *Business API Ecosystem GE*.
- *Security* provides a comprehensive set of services for applications to comply with major security requirements such as authentication and authorization. The *Identity Management (IDM) GE*[29] is one of the most important services in the FIWARE platform. The service manages all FIWARE users accounts providing access to the platform's services. In addition, users can register their services and configure their security (roles, authorization policy). Finally, the service acts as an authorization server providing access to third-party applications using the OAuth2 flow.
- *Interface to Networks and Devices (I2ND)* provides Generic Enablers that allow the creation and management of advanced network infrastructures such as the *Network Information and Control GE*[30].
- *Advanced Web-based User Interface* provides Generic Enablers that significantly improve the user experience for the Future Internet by adding new user input and interaction capabilities, such as interactive 3D graphics, immersive interaction with the real and virtual world

[26] https://catalogue.fiware.org/enablers/iot-discovery

[27] https://catalogue.fiware.org/enablers/business-api-ecosystem-biz-ecosystem-ri

[28] https://catalogue.fiware.org/enablers/application-mashup-wirecloud

[29] https://catalogue.fiware.org/enablers/identity-management-keyrock

[30] https://catalogue.fiware.org/enablers/network-information-and-control-ofnic

(Augmented Reality), virtualizing and thus separating the display from the (mobile) computing device for ubiquitous operations, and many more.

Overall, FIWARE is an IaaS and PaaS platform that aims to provide an innovative and cost-effective way of building cloud applications while maintaining high quality of service and security.

2.5 Semantic Web and Linked Data

Semantic Web

The Semantic Web is an extension of the current Web in which information is offered not only in the form of natural language documents but also as machine-readable data, enabling machine to machine communication in addition to human to machine communication that is supported by the existing Web [5]. In order to achieve its purpose a set of standards and technologies have been developed offering an environment where data and their relationships are represented in a common data format, and also inferences can be drawn from existing data. Vocabularies and data can be accessed using query languages.

The Resource Description Framework (RDF) [13] builds the foundation of the Semantic Web. RDF is a data model for expressing information about resources using statements. Resources can be anything, including documents, people, physical objects, and abstract concepts. A resource is identified by an International Resource Identifier (IRI). IRIs are global identifiers, so that an IRI can be re-used to identify the same thing. An RDF statement expresses a relationship between two resources, in the form of triple (Figure 2.4). The Subject and the Object represent the two resources being related, the predicate represents the nature of their relationship. Multiple triples

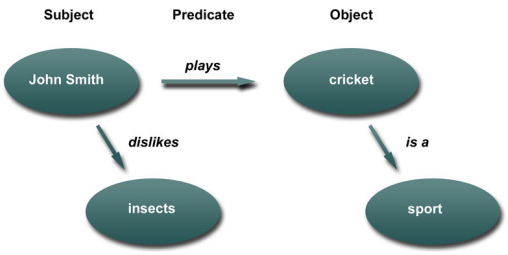

Figure 2.4: RDF data model

build a graph, and multiple graphs form a dataset.

The RDF data model doesn't make any assumptions about what resource IRIs stand for. RDF Schema (RDFS) and Web Ontology Language (OWL) offer the tools for creating vocabularies or, more formally, ontologies that capture knowledge in an area of interest. More specifically, ontologies provides the means for representing high-level concepts, their properties and interrelationships. This information is supplemented by classificatory information for such entities (i.e. making it possible to define classification hierarchies for objects and object properties) as well as, entity and relation constraints. Ontology axioms provide the semantics allowing for checking existing information for consistency or for inferring new information based on information defined explicitly in the ontology.

RDF Schema (RDFS) [8] provides a data-modeling vocabulary for RDF data. It provides mechanisms for describing classes, class hierarchies, data types, or properties similar to object-oriented programming languages. Unlike RDFS, the Web Ontology Language (OWL) [21] is a family of knowledge representation languages offering increased expressiveness for describing classes and properties. Among others, OWL allows the definition of relations between classes (e.g. disjointness), equality, restrictions over properties (e.g. cardinality restrictions) and partial order and equivalence relations between properties (e.g. transitive, symmetric properties).

The last piece of the Semantic Web is the ability for querying RDF data.

SPARQL [26], a recursive acronym for SPARQL Protocol and RDF Query Language, is the W3C recommendation not only for querying and manipulating RDF data but also a protocol to invoke such queries over HTTP and a number of result formats (XML, JSON, CSV).

Linked Data

In 2006, Sir Tim Berners-Lee[31] introduced the term Linked Data [4], describing how RDF data should be published on the Web, fulfilling part of the vision of Semantic Web. According to [6] Linked Data refers to data published on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external data sets, and can in turn be linked to from external data sets. Moreover, Berners-Lee introduced a set of principles for successfully creating and publishing Linked Data.

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names (dereferencing the URI).
3. When someone looks up a URI, provide useful information, using the standards (RDF*, SPARQL).
4. Include links to other URIs so that they can discover more things.

2.6 Interface Description Languages

A service description (or contract) is a fundamental part in SOA [15], as they expose the purpose and the functionalities of a service and thus allowing for a service to be discovered and used. It provides all the needed information about service endpoint, service operations, the exchanged message formats

[31] https://www.w3.org/People/Berners-Lee/

and the conditions which need to be fulfilled before the service can be executed.

In cloud environments, providers describe their services in plain text (such as the FIWARE catalogue[32]), which users have to browse and read in order to determine whether a service meets their needs. However, text descriptions of services are mainly intended to be readable by humans and not by machines, while in some cases are inaccurate or vague. Cloud services need to be formally described in a way that is both understandable by humans and machines.

In this section, we will discuss the most noteworthy approaches to describe interface description languages for Cloud services that are offered by means of Web services (SOAP-based or RESTful services).

2.6.1 WSDL and SAWSDL

The Web Service Description Language [9] (WSDL) is a XML-based interface description language that is used for describing the functionality offered by a Web service. WSDL is part of the initial Web Services technology platform (Section 2.2). The current version of the specification is 2.0, which is a W3C recommendation. However, version 1.1 [10] is still in use.

The structure of a WSDL document can be seen in Figure 2.5[33]. WSDL 2.0 introduced changes in document structure, naming conventions, while it added support for the HTTP 1.1 protocol. SOAP-based services (Section 2.2) were described in WSDL 1.1, using a predefined binding extension for the SOAP protocol. WSDL 2.0 introduced a corresponding binding extension for the HTTP 1.1 protocol[34], in order to allow the description of RESTful services (Section 2.3).

[32] https://catalogue.fiware.org/

[33] Original figure from https://en.wikipedia.org/wiki/Web_Services_

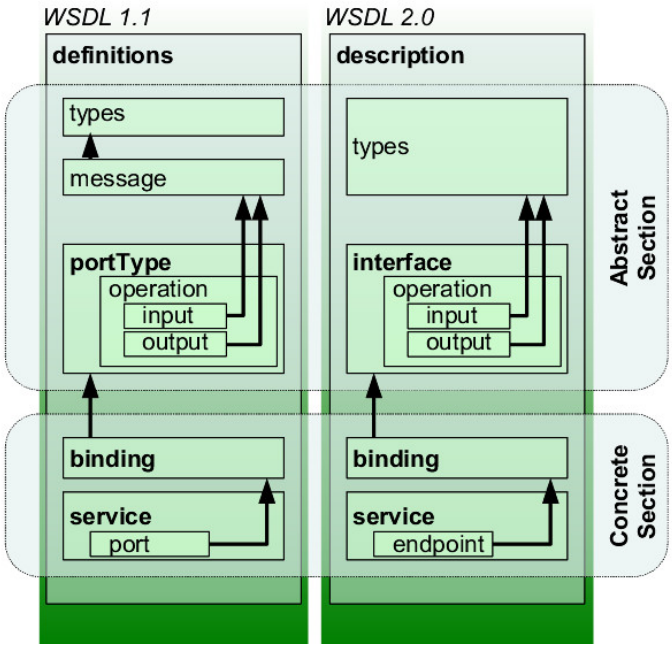

Figure 2.5: WSDL document structure

In WSDL 1.1 the *types* element contains the data types definitions that are required by the request and response message definitions of the service operations. The XML Schema language (XSD) [20] is used (inline or referenced) for this purpose. The *messages* element defines the messages that are required in order to interact with a service operation. A message is described by the data types defined in *types* element. WSDL 2.0 preserved the *types* element in its syntax, but removed the *messages* element, as operations could directly refer to the data types in *types* element.

The *portType* element in WSDL 1.1, renamed *interface* in WSDL 2.0, defines all the operations that a service may perform. An *operation* element represents a function of the service and describes the input, output and fault messages produced upon successful or unsuccessful invocation of the service.

The *binding* element is common in both versions of WSDL, and specifies the network protocol and data format of messages used in operations defined

Description_Language

[34] https://www.w3.org/TR/wsdl20-adjuncts/

in *portType* (or *interface* for WSDL 2.0).

The *service* element describes the interface through which service operations may be accessed. In particular, a *port* (*endpoint* in WSDL 2.0) specifies the address to which the client should request the service. It also has reference to *binding* used by the *service*.

WSDL only describes a service at a syntactic level. In 2007, W3C introduced the Semantic Annotations for WSDL and XML Schema (SAWSDL) [16] recommendation, in order to provide a mechanism to associate semantics with service interfaces and message schemas. SAWSDL defines how to add semantic annotations to various parts of a WSDL document such as inputs, outputs, interfaces and operations by providing three extensibility attributes to WSDL and XML Schema elements.

The *modelReference* attribute defines the association between a WSDL or XML Schema component and a concept in an ontology. It is used to annotate XML Schema type definitions, element and attribute declarations as well as WSDL interfaces, operations and faults.

The *liftingSchemaMapping* and *loweringSchemaMapping* attributes are added in XML Schema element declarations and type definitions for specifying mappings between semantic data and XML data. A reference using the *liftingSchemaMapping* attribute defines the mechanism through which XML data are transformed to data that conform to some semantic model. On the other hand, a reference using the *loweringSchemaMapping* attributes defines the mechanism through which data from a semantic model are transformed to XML data.

Listing 2.1[35] demonstrates how SAWSDL extension attributes are applied on an XML Schema. The listing describes the confirmation response of a purchase order. As seen the *modelReference* property is used to indicate that the confirmation element is described by the concept "Order Confirmation" in

[35]Example taken from `https://www.w3.org/TR/sawsdl/`

the referenced semantic model. Similarly the *liftingSchemaMapping* attribute refers to a XSLT mechanism (represented in Listing 2.2), which a client processor could use in order to transform the XML data in RDF.

Listing 2.1: Example usage of SAWSDL

```
...
<xs:element name="OrderResponse" type="confirmation" />
<xs:simpleType name="confirmation" sawsdl:modelReference="
    http://.../spec/ontology/purchaseorder#OrderConfirmation"
    sawsdl:liftingSchemaMapping="
    http://.../spec/mapping/Response2Ont.xslt">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Confirmed" />
    <xs:enumeration value="Pending" />
    <xs:enumeration value="Rejected" />
  </xs:restriction>
</xs:simpleType>
...
```

Listing 2.2: The XSLT mechanism for transforming XML data in RDF

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:po="http://www.w3.org/2002/ws/sawsdl/spec/wsdl/order#"
    xmlns:POOntology="http://.../spec/ontology/purchaseorder#"
    version="2.0">
  <xsl:output method="xml" version="1.0" encoding="iso-8859-1"
      indent="yes"/>
  <xsl:template match="/">
    <rdf:RDF>
      <POOntology:OrderConfirmation>
        <hasStatus rdf:datatype="
            http://www.w3.org/2001/XMLSchema#string">
          <xsl:value-of select="po:OrderResponse"/>
        </hasStatus>
      </POOntology:OrderConfirmation>
    </rdf:RDF>
  </xsl:template>
</xsl:transform>
```

SAWSDL allows multiple semantic annotations to be associated with WSDL elements. Both schema mappings and model references can contain multiple pointers. Multiple schema mappings are interpreted as alternatives whereas multiple model references all apply. SAWSDL does not specify any other relationship between them. SAWSDL is criticized as it comes without any formal semantics [27]. This hinders logic-based discovery and composition of Web services described with SAWSDL but calls for "magic mediators outside the framework to resolve the semantic heterogeneities" [27].

Despite the fact that WSDL 2.0 is able to describe both SOAP-based and RESTful services, it is not perceived as suitable by developers. WSDL is limited to the description of traditional SOAP-based services. Moreover, the adoption of WSDL 2.0 is poor as more tools are offered only for WSDL 1.1.

2.6.2 WADL

The Web Application Description Language [24] (WADL) is an XML-based interface description language that is used for the description of HTTP-based Web services. WADL is specifically designed for describing RESTful services (Section 2.3), by modeling the resources provided by the service and the relationships between them.

Listing 2.3: WADL description for the Yahoo News Search application

```
<application xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:schemaLocation="
    http://wadl.dev.java.net/2009/02 wadl.xsd" xmlns:tns="
    urn:yahoo:yn" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:yn="urn:yahoo:yn" xmlns:ya="urn:yahoo:api" xmlns="
    http://wadl.dev.java.net/2009/02">

<grammars>
  <include href="NewsSearchResponse.xsd"/>
  <include href="Error.xsd"/>
</grammars>
```

```
<resources base="
    http://api.search.yahoo.com/NewsSearchService/V1/">
  <resource path="newsSearch">
    <method name="GET" id="search">
      <request>
        <param name="appid" type="xsd:string" style="query"
            required="true"/>
        <param name="query" type="xsd:string" style="query"
            required="true"/>
        <param name="type" style="query" default="all">
          <option value="all"/>
          <option value="any"/>
          <option value="phrase"/>
        </param>
        <param name="results" style="query" type="xsd:int"
            default="10"/>
        <param name="start" style="query" type="xsd:int"
            default="1"/>
        <param name="sort" style="query" default="rank">
          <option value="rank"/>
          <option value="date"/>
        </param>
        <param name="language" style="query" type="xsd:string"
            />
      </request>
      <response status="200">
        <representation mediaType="application/xml" element="
            yn:ResultSet"/>
      </response>
      <response status="400">
        <representation mediaType="application/xml" element="
            ya:Error"/>
      </response>
    </method>
  </resource>
</resources>
</application>
```

Listing 2.3[36] demonstrates an example of a WADL document. The *grammars* element contains all data types definitions that are used by the service. In our example, the *grammars* element specifies the XML Schema files that contain the data types definitions.

The *resources* element contains all the resources provided by the service. The *base* attribute provides the base path for any *resource* element. A *resource* element describes a set of methods that define the behavior of a resource. The *path* attribute defines a relative URI (static or template) identifying the resource. It is also possible a *resource* element to contain *resource* elements that act as sub-resources.

A *method* element describes the input and outputs when applying an HTTP method to a resource. The *name* attribute indicates the HTTP method that is used, while the *id* attribute provides an identifier for the method. In our example, a method, named "search", describes a GET request on the path "newsSearch". A *method* element contains a *request* element and zero or more *response* elements.

A *request* element specifies the *query* and *headers* parameters as well as the message body that may be included in the HTTP request message. A *response* element describes the expected response body messages, HTTP headers and the HTTP status code that results from performing an HTTP method on a resource. A body message is described by a *representation* element, specifying the media type and the data format as defined in the *grammars* element. In our example, the "newsSearch" method specifies a *request* element describing multiple *query* parameters that may be required to be included in the HTTP request. In addition, the "newsSearch" method specifies the expected responses for 200 and 400 HTTP status codes. In both cases, the expected response contains a message body in XML representation, whose data format is defined in *grammars* element.

Despite the efforts for standardization, WADL hasn't attracted any sig-

[36]WADL example taken from `https://www.w3.org/Submission/wadl/`

nificant adoption among developers. The main critique is focused on the fact that WADL is closely related to WSDL, providing only a syntactic description of the service, with limited support for describing the meaning of service's resources. In contrast to WSDL, no mechanism to semantically annotate service descriptions exists for WADL.

2.6.3 OpenAPI Specification, RAML, API Blueprint

As the REST architecture style has gained massive adoption over the years, a number of various interface description languages have been proposed, promising to provide efficient and accurate descriptions for RESTful services. OpenAPI Specification, RAML and API Blueprint are the most commonly used approaches for describing RESTful services. They follow quite a similar approach to WADL (Section 2.6.2), meaning that everything is bound to the URLs for accessing resources, which, however, contradicts to REST's hypermedia constraint that calls for the dynamic discovery of resources at runtime (HATEOAS[37]). However, their adoption is mainly due to the large tooling support they offer covering the whole API lifecycle from design to sharing.

OpenAPI Specification

The OpenAPI Specification (OAS) [35], formerly known as the Swagger specification, is probably the most heavily adopted approach, for the description of RESTful services (Section 2.3). It is an open-source, language-agnostic specification, through which a consumer can understand and use a service by applying minimal implementation logic. Service descriptions are offered in either JSON or YAML [3] format, which can be produced and served statically, or be generated dynamically from the application. This allows the design and implementation of APIs to follow either a top-down (the service description is initially created and then the service is implemented) or

[37] https://en.wikipedia.org/wiki/HATEOAS

bottom-up approach (the service description is generated from the service implementation). A comprehensive analysis of the specification is presented in Chapter 3.

OAS is, a complete framework supported by a set of core tools for designing, building and documenting RESTful services. The Swagger Editor[38] is an open-source Web-based editor for designing, defining and documenting RESTful services (Figure 2.6). It provides instant visualization and interaction with the API while still defining it. The Swagger Codegen[39] is an open-source code generator to build server code and client SDKs directly from an OpenAPI service description in almost any programming language and framework (PHP, Java, NodeJS). Swagger UI[40] is an open-source HTML5-based user interface to visually render documentation for an OpenAPI service description (Figure 2.7).

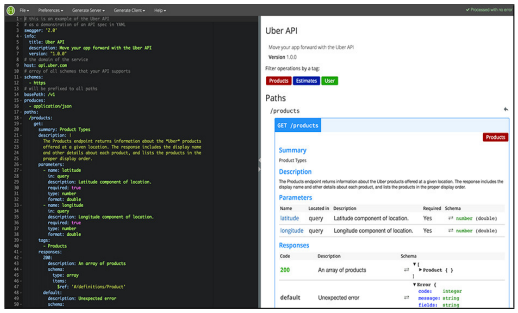

Figure 2.6: Swagger Editor preview

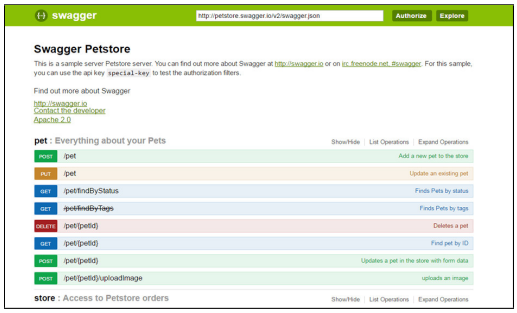

Figure 2.7: Swagger UI preview

The OpenAPI Specification is part of the Open API Initiative[41], a collaborative project supported by the Linux Foundation[42], which aims at "*creating, evolving and promoting a vendor neutral API description format*". The initiative is supported by a constantly increasing community including compa-

[38] https://github.com/swagger-api/swagger-editor
[39] https://github.com/swagger-api/swagger-codegen
[40] https://github.com/swagger-api/swagger-ui
[41] https://www.openapis.org/
[42] https://www.linuxfoundation.org/

nies like Google[43], Microsoft[44], IBM[45] and many others. The current version of the OpenAPI Specification is 2.0, since September of 2014, however there is an ongoing process for creating a new version of the specification that will provide additional features based on users' requests and needs.

RAML

The RESTful API Modeling Language (RAML) [41] is a YAML-based [3] language for describing "practically"-RESTful services. Because of the way RAML is designed, it can support documentation of REST-based services (Section 2.3) in addition to services that don't strictly adhere to REST principles, such as SOAP-based services (Section 2.2). Contrary to OpenAPI Specification, RAML is only a top-down specification, meaning that the API is first designed and then the rest of the system is implemented.

RAML is also supported by a number of native and third-party tools that facilitates the management of the whole API lifecycle from design to sharing. API Designer[46] is a web-based API development tool that allows API providers to design their API quickly, efficiently, and consistently and socialize the design. It consists of a RAML editor side-by-side with an embedded RAML console (API Console). The API Console[47] provides live interactive documentation that lets users try out an API in real time. Unlike OpenAPI Specification, client code generation from RAML API documents is mainly provided by third-party commercial tools.

[43] https://www.google.com/intl/en/about/
[44] https://www.microsoft.com
[45] http://www.ibm.com
[46] https://www.mulesoft.com/platform/api/anypoint-designer
[47] https://github.com/mulesoft/api-console

API Blueprint

The API Blueprint [2] follows a different approach compared to OpenAPI Specification and RAML. It is a documentation-oriented description language based on a set of semantic assumptions laid on top of the Markdown syntax [22]. Unlike OpenAPI Specification and RAML, API Blueprint doesn't dictate a specific style for the description of a service. A service provider is free to describe the functionality of his service in any way he prefers. For example, a service may be described by only providing examples for the various request and response messages without including any data type definitions (XML Schema, JSON Schema) that specify the structure of request and responses messages.

Due to the nature of the API Blueprint specification, there is limited tool support. The most known tool around API Blueprint is the Apiary platform[48], that provides a collaborative design editor, interactive documentation and other tools to improve user's experience and interaction with a described web API. The main drawback of API Blueprint is that it lacks tooling for code generation.

2.7 Ontologies and Vocabularies

Interface description languages normally offer only syntactic descriptions of service API. However, such descriptions are insufficient to enable the automation of tasks such as service discovery and composition. In order to solve this problem, earlier as well as more recent research suggest describing services also semantically.

[48] http://apiary.io

2.7.1 OWL-S

The Web Ontology Language for Web Services (OWL-S) [32] is an upper ontology based in OWL, used to semantically annotate Web services. OWL-S consists of the following main upper ontologies shown in Figure 2.8.

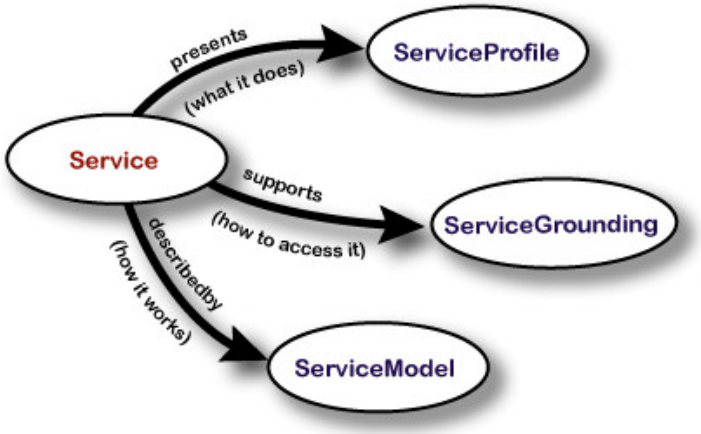

Figure 2.8: The OWL-S ontology

The *Service Profile* is used to describe what the service does, and is meant to be mainly used for the purpose of service discovery. A *Service Profile* includes information about the service provider as well as functional and non-functional service parameters. The functional description of the service specifies the inputs required by the service and the outputs generated. Furthermore, since a service may require external conditions to be satisfied, the *Service Profile* describes the preconditions required by the service and the results from the execution of the service. The non-functional description of the service may specify parameters such as the category and the rating of the service.

The *Service (Process) Model* gives a detailed description of a service's operation and describes the composition (choreography and orchestration) of one or more services. A process in OWL-S can be atomic, simple, or composite.

An atomic process describes a specific action that a service performs in a single step. An atomic process is described by specifying the input and output messages as well as preconditions and effects for successful interaction with the service. A composite process describes a hierarchically defined workflow, consisting of atomic and other composite processes. Finally, a simple process represent an abstract process which have no specific binding to a physical service. A simple process can be used to group atomic processes that provide multiple descriptions of the same process, as well as to provide a more simplified description of a composite process.

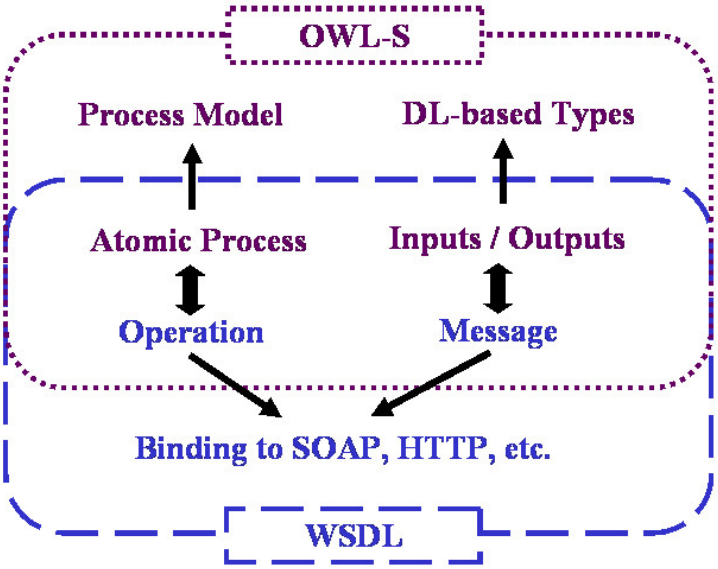

Figure 2.9: Mapping between OWL-S and WSDL

Service Grounding provides the required information for accessing the service. Particularly, *Service Grounding* specifies an mapping between the logic-based and the syntactic description in order to facilitate service execution. Figure 2.9 demonstrates how the binding between OWLS-S and WSDL is performed. The atomic processes of the *Service Process* are mapped to WSDL's operations, while the inputs and outputs of atomic processes are mapped to WSDL's message elements that contain the XML data types that define the input and output of WSDL's operations.

2.7.2 WSMO

The Web Service Modeling Ontology (WSMO) [31] defines a conceptual model and a formal language WSML (Web Service Modeling Language) for the semantic description of Web services. In WSMO, a Web service is described under the notion of four top-level elements, as shown in Figure 2.10.

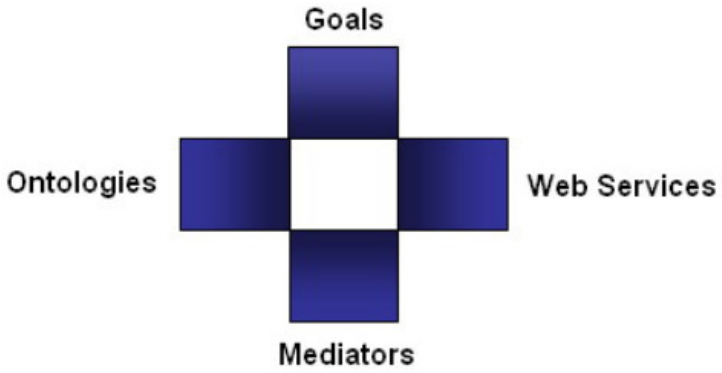

Figure 2.10: The top-level elements of WSMO

WSMO Ontologies provide the terminology used by other WSMO elements to describe the relevant aspects of the domain of interest. *Goals* specify the requirements that a user might have when searching for a Web service. *WSMO Service* describes the functional, non-functional and behavioral aspects of a Web service.

Non-functional properties include financial properties that refer to the charging model of the service as well as properties that describe the performance (execution time, latency), accuracy and security regarding the authorization and authentication of users. The functionality of the service is defined as capability which, similar to the *atomic process* of OWL-S, specifies the input and output messages as well as preconditions and effects to successfully interact with the service. Finally, the *WSMO Service* provides the necessary information so that a client may communicate and interact with the service.

Mediators describe elements that overcome interoperability problems that arise between different WSMO elements in data, process and protocol level. *Mediators* in data level resolve the terminological mismatches that exist between heterogeneous data sources (ontology integration). *Mediators* in protocol level resolve mismatches that arise between the heterogeneous communication protocols (e.g. SOAP, HTTP). Lastly, *Mediators* in process level resolve the mismatches that arise during the orchestration and cooperation of services.

WSMO was heavily criticized due to the fact that it has been developed without any compliance with the existing W3C standards [27]. WSMO Ontologies defined Classes, subClass hierarchies and properties in a format that wasn't compatible with any of the existing standards of the Semantic Web (RDFS, OWL). In addition, WSMO didn't provide any connection to the WSDL service descriptions. To overcome these issues, WSMO-Lite has been created, that is discussed in the next section.

2.7.3 WSMO-Lite

WSMO-Lite [17] is a lightweight ontology of service descriptions, that can be used for annotations of various WSDL elements using the SAWSDL annotation mechanism [16]. As stated in Section 2.6.1, SAWSDL doesn't impose any formal semantics. The WSMO-Lite ontology aims to fill the SAWSDL annotations with concrete semantic service descriptions.

WSMO-Lite introduces the following service semantics, in order to provide better service descriptions and enable service discovery, and orchestration:

- *Information semantics* are represented using domain ontologies that describe the data model of a service used in input, output and fault messages.
- *Functional semantics* represent the capabilities and the classification of service functionalities. Capabilities define the preconditions and the

effects that must hold before and after a client invoke a service, while classification is achieved using a classification ontology that provides a hierarchy of categories.

- *Nonfunctional semantics* are represented using ontologies that describe nonfunctional properties, such as pricing models and service performance.
- *Behavioral semantics* are expressed by annotating service operations with *Functional semantics*.

WSMO-Lite has been designed not to be bound to a particular service description format such as WSDL. Therefore, it can be also used for the description of RESTful services (Section 2.3), that are documented in HTML pages. However, HTML documentation of RESTful services is not machine-readable. In [37], hRests and MicroWSMO, two HTML microformats that act similarly to WSDL and SAWSDL (Figure 2.11) have been introduced for annotating HTML service documentation so that it becomes machine-readable.

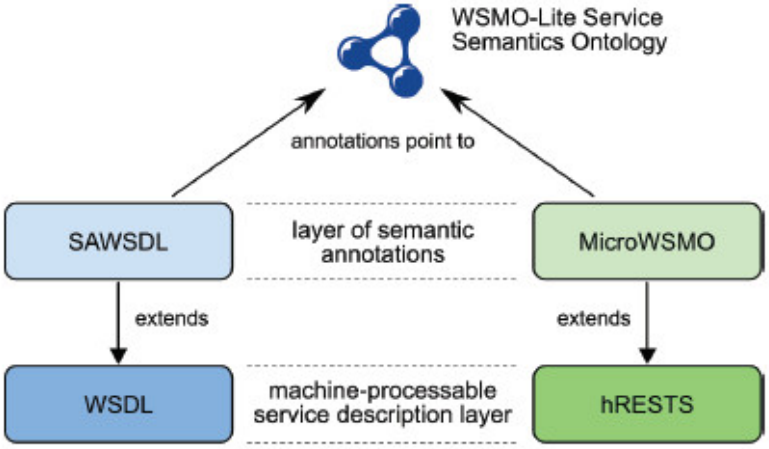

Figure 2.11: Web service descriptions with WSMO-Lite

2.7.4 Hydra Core Vocabulary

Hydra [30] is a set of technologies that simplify the development of interoperable, hypermedia-driven Web APIs. At the heart of this approach lies the Hydra Core Vocabulary. It defines a number of concepts in RDF Schema that allow machines to understand how to interact with an API. The main idea behind Hydra is to provide a vocabulary through which a server may be able to advertise valid state transitions to a client. This means that response messages from the server contain enough information that a client can use in order to discover all the available actions and resources it needs, and thus construct new HTTP requests to achieve a specific goal.

Figure 2.12 provides a conceptual view of the Hydra Core Vocabulary. At the center of the vocabulary stands the *ApiDocumentation* class, through which the server defines the main entry point (*EntryPoint*) and documents all the operations (*Operation*) as well as the entities (*Class*) and their properties (*Property*) it supports.

The *Resource* class is used to inform a client that an IRI is dereferenceable, meaning that when a IRI is accessed a representation of a resource is retrieved. This is an important feature as it allows a client to distinguish Linked Data from IRIs that are used exclusively as identifiers. Similarly, the *Link* class is used in order to define properties whose properties are known to be dereferenceable IRIs.

However, there are cases that the interaction with the service requires links that cannot be created by a server. For example, in order to query a service a link may contain parameters that a client must fill at runtime. In Hydra, such cases are described by the *IriTemplate* class. An *IriTemplate* consists of a *template* that describes an IRI template[49] and a number of *mappings*. An *IriTemplateMappging* maps a variable in the IRI template to a property. Listing 2.4 demonstrates an example of an IriTemplate description, where

[49] `https://tools.ietf.org/html/rfc6570`

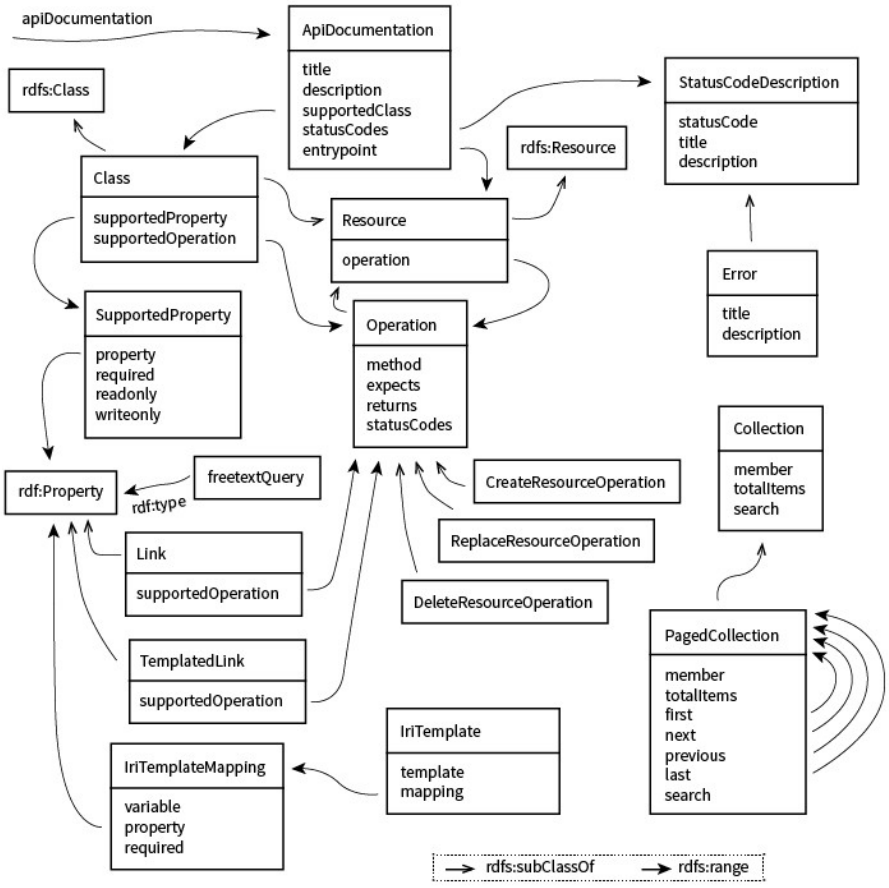

Figure 2.12: The Hydra Core Vocabulary

the variable "*lastname*" maps to the property "*givenName*" from Schema.org vocabulary. With this information, a client may understand the meaning of variables and generate a complete IRI.

Listing 2.4: Description of an IriTemplate in Hydra

```
{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
  "@type": "IriTemplate",
  "template": "http://api.example.com/users{?lastname}",
  "mapping": [
    {
      "@type": "IriTemplateMapping",
      "variable": "lastname",
      "property": "schema.org/givenName",
```

```
      "required": true
    }
  ]
}
```

The *Operation* class represents the information that is necessary so that a client may send valid HTTP requests to the server. The *method* property specifies the HTTP method, while the *expects* and *returns* properties define the expected data in request and response messages. In addition, the *statusCode* property specifies a *StatusCodeDescription* that provides a developer with information regarding what to expect when invoking an operation.

An interesting feature of the vocabulary is the *Class* class that extends a class definition by providing the *supportedProperties* that belong to the class. This is important as in RDF there is not any mechanism informing which properties belong to a class and also enables properties from other vocabularies to be reused directly. A *SupportedProperty* defines the property that is used and specifies whether it is required, readonly or writeonly.

In a Hydra-driven Web API, the service description may be discovered automatically by a client if the API provider marks his responses with a HTTP Link Header to direct a client to the corresponding API document. This enables the dynamic discovery of API descriptions at runtime. Moreover, due to the use of RDF's unique identifiers parts of the API descriptions can be shared and reused improving interoperability of services.

The Hydra core vocabulary is used along with JSON-LD, in order to enable the creation of hypermedia-driven APIs [29]. JSON-LD [38] is a lightweight format for the representation of Linked Data in JSON. Its design allows existing JSON to be interpreted as Linked Data with minimal changes. Moreover, it is 100% compatible with JSON, and thus the large number of existing JSON parsers and libraries can be reused. The combination of Hydra and JSON-LD enables the creation of machine-readable and understandable contracts that can be discovered at runtime. This allows the implementation of

completely generic clients[50], such as API consoles or client libraries.

Hydra is a promising effort towards the evolution of RESTful services. In its current state, Hydra is endorsed by the W3C and a community group is working to extend Hydra and provide tools and guidelines for designing and creating Hydra-driven Web APIs.

2.8 Service Catalogues

Service descriptions allow services to be stored and organized in registries or catalogues, enabling the creation of mechanisms for effective service discovery based on specific user needs. Service registries are important components in SOA (Section 2.1) as they promote service discoverability and reusability. Typically, cloud providers don't provide any specific platform for service storage and discovery, as Cloud services are described in plain text. However, Oracle has created an API Catalog service, which is described in the following section, in an effort to provide formal service descriptions for its Cloud offerings[51].

2.8.1 Oracle API Catalog Service

The API Catalog Cloud Service[52] is a collection of OpenAPI service descriptions [35], representing some of Oracle's most popular SaaS and PaaS applications. Through the service a user is able to search, browse and try out Oracle's Cloud Services using either a web interface or the service API. The API catalog organizes the APIs by hierarchical categories, allowing a user to locate an API and export its OpenAPI description either by browsing the category lists, or by conducting full-text search.

[50] http://www.hydra-cg.com/#tooling
[51] https://cloud.oracle.com/home
[52] https://cloud.oracle.com/en_US/api-catalog

Oracle's API Catalog Cloud Service is a first attempt towards the creation of Cloud service registries. Oracle utilizes all the capabilities that the OpenAPI Specification offers. However, service discovery is still inaccurate and inefficient, as the search operation relies on the existence of text terms in API descriptions. The approach described in Chapter 3 aims to improve Cloud service discovery by semantically annotating OpenApi service descriptions and thus reducing unambiguity in service descriptions

3

The Semantic OpenAPI Specification

In the previous chapter, we analyzed the importance of formal Cloud service descriptions in SOA. Moreover, we reviewed the most notable and commonly used approaches for the syntactic and semantic description of services. However, the majority of Cloud providers don't use any of the previously discussed approaches to describe their offering services. Thus, neither providers nor potential users can benefit from the capabilities that formal service descriptions offer.

In the following sections, we present our approach the Semantic OpenAPI Specification (SOAS), an extension of the OpenAPI Specification, for the effective and efficient description of Cloud services. We analyze the reasons that led us to the adoption of OAS, and we demonstrate how OpenAPI service descriptions can be semantically enriched in order to resolve any ambiguities that may exist in service descriptions. Moreover, we describe a mechanism for transforming SOAS service descriptions to ontologies so as to benefit from semantic web tools such as reasoners and query languages for service discovery and for enabling service orchestration.

3.1 Adopting the OpenAPI Specification

In order to choose a description language for the Cloud services we had to consider many aspects that would affect our decision. First of all, a descrip-

tion language for RESTful services was needed, as the majority of Cloud services are offered by means of Web services based on the REST architecture style. In addition, the description language should be simple enough so that users with different knowledge background could easily learn and adopt. It is also important the description language to be supported with tools that facilitate users at every step of their interaction with a service.

Based on these considerations, WSDL and WADL couldn't serve our goals. WSDL, despite being a W3C recommendation, hasn't been adopted by developers, as they found it complex and with no tooling support. In fact, WSDL is identified and used mainly as the description language of SOAP-based services (Section 2.2). On the other hand, WADL although it was created to support the description of RESTful services, had small adoption rate by developers. WADL is mainly supported by JAVA frameworks (JAX-RS[1]), but there is not known support for other programming languages.

Hydra is a novel approach, that is entirely based on Semantic Web. In its current state, Hydra is under constant development and evaluation by the corresponding W3C group in order to provide guidelines and tools for the development and design of RESTful services. So far, there is not any official recommendation, which is a deterrent factor for considering Hydra suitable for the description of Cloud services. However, we expect Hydra to gain popularity after the release of a recommended specification.

In our work, we adopt the OpenAPI Specification as the description language for Cloud services. The selection was based primarily on the popularity of the specification, as well as the capabilities and tools it offers facilitating user's interaction with a service (Section 2.6.3). Its active community is constantly working to improve its tooling support as well as the specification itself. In addition, the OpenAPI Initiative is powered by companies such as Microsoft and Google, attempting to standardize a description mechanism for RESTful services. It is worth mentioning that OpenStack (Section 2.4.2)

[1] `https://jax-rs-spec.java.net/`

has announced[2] the adoption of OAS for the documentation and description of its services.

3.2 Describing the OpenAPI Specification

Figure 3.1 illustrates the structure of an OAS service description[3]. An OAS service description may be written in either JSON or YAML format [3]. YAML is a more human friendly data serialization format, and is considered as a superset of JSON. In addition, some parts of the document can be split into separate files, allowing the reuse of these files across multiple service descriptions.

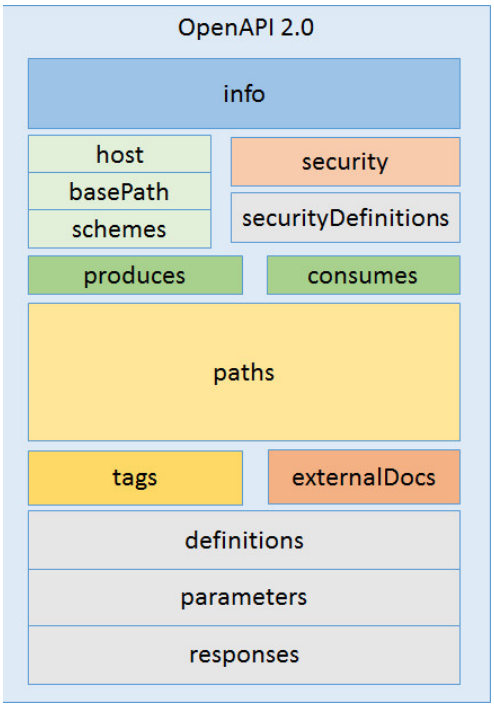

Figure 3.1: OAS document structure overview

[2] https://www.openstack.org/summit/austin-2016/
[3] Examples of the OAS specification are represented in YAML format

The specification adopts some features of JSON-Schema[4] in order to describe the content of HTTP requests and responses. In addition, the specification introduces the "file" data type, which is used for describing the content of HTTP requests and responses that require files. Table 3.1 provides a brief summary of the components of an OAS service document.

The *Info Object* provides non-functional information about the described service. It is required to contain the service name as well as the version of the service API. In addition, it may provide information regarding the service's license, the terms of service, and contact information of the service provider. Listing 3.1 provides an example of the *Info Object*.

Listing 3.1: OAS Info Object example from Swagger Petstore

```
info:
  description: This is a sample Petstore server. You can find out
      more about Swagger at [http://swagger.io](http://swagger.io
     ) or on [irc.freenode.net, swagger](http://swagger.io/irc/).
      For this sample, you can use the api key 'special-key' to
     test the authorization filters.
  version: 1.0.0
  title: Swagger Petstore
  termsOfService: 'http://swagger.io/terms/'
  contact:
    email: apiteam@swagger.io
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
```

The *SecurityDefinitions Object* contains the security schemes that the service uses for authentication. The OAS offers support for basic authentication [19], an API key[5] and OAuth2's common flows [25]. For an API key definition, it is necessary to specify the name of the parameter that holds the API key, as well as its location in the HTTP request (a query string parameter or header). In case of an OAuth2 security scheme, the used authorization flow

[4] http://json-schema.org/latest/json-schema-core.html#anchor8
[5] https://en.wikipedia.org/wiki/Application_programming_interface_key

Field	Description
info	Provides non-functional information about the described service.
host	Specifies the host (name or ip) of the service.
basePath	Provides the base path on which the service can be accessed which is relative to the host.
schemes	Defines the used transfer protocols of the service (http, https).
produces	A list of media types that the service supports for responses. Operations are allowed to override this declaration and define alternative media types.
consumes	A list of media types that the service supports for requests. Operations are allowed to override this declaration and define alternative media types.
paths	The available paths of the service along with supported operations.
tags	A list of tags that can be used to group operations by resources or any other qualifier.
definitions	Contains all the data types (arrays, models, primitives) definitions that are used by service's operations.
parameters	Contains the defined parameters that can be reused across operations.
responses	Contains the defined responses that can be reused across operations.
securityDefinitions	Defines the security schemes that can be used across operations.
security	A declaration of which security schemes are applied for the service as a whole. Operations may override this declaration if alternative security schemes are used.
externalDocs	Provides link to external resources for additional documentation if needed

Table 3.1: OAS document structure from [35]

(implicit, accessCode, password, application) must be specified, along with the scopes and the urls, for obtaining a valid token.

Listing 3.2 demonstrates the security schemes defined in Swagger Pet-

store. The API defines two security schemes that are used in various HTTP requests. The API key security scheme defines an HTTP header, named "api_key", which must be provided in every HTTP request that requires it. On the other hand, the OAuth2 security scheme describes the procedure, in this case an "implicit" flow, that must be applied so that a valid token to be obtained. The OAuth2 security definition specifies the authorization endpoint, as well as the available levels of access (scopes) that the application can request.

Listing 3.2: OAS Security Definition example from Swagger Petstore

```
securityDefinitions:
  # an API key definition
  api_key:
    type: apiKey
    name: api_key
    in: header

  # an implicit flow definition
  petstore_auth:
    type: oauth2
    authorizationUrl: http://petstore.swagger.io/api/oauth/dialog
    flow: implicit
    scopes:
      write_pets: modify pets in your account
      read_pets: read your pets
```

The *Paths Object* contains the relative paths for the service endpoints. Each *Path item* describes the available operations based on HTTP methods, as seen in Listing 3.3.

The core of an OAS service document is the *Operation Object*, as it practically reveals the functionality of a service path, and guides the user to interact with it. It contains all the needed information in order to construct HTTP requests to the service. In addition, it provides information regarding the HTTP responses that the service returns. Table 3.2 summarizes the fields that describe an *Operation Object*.

Listing 3.3: OAS Path Item example from Swagger Petstore

```yaml
paths:
  /pets/{petId}:
    get:
      tags:
        - Pet
      description: Returns pets based on ID
      summary: Find pet by ID
      operationId: getPetById
      produces:
        - application/json
        - text/html
      parameters:
        - name: petId
          in: path
          description: ID of pet to use
          required: true
          type: string
      responses:
        '200':
          description: pet response
          schema:
            $ref: '#/definitions/Pet'
        default:
          description: error payload
          schema:
            $ref: '#/definitions/ErrorModel'
      security:
        - petstore_auth:
          - write:pets
          - read:pets
```

The *Responses Object* contains all the expected responses of an *Operation Object*. It maps an expected response to a specific HTTP status code. The *Response Object* describes the message content and HTTP Headers that an operation's response may contain. It is not required to declare all the possible responses of an operation, as it is expected to describe a successful operation

Field	Description
summary	A short summary of operation's functionality.
description	A verbose analysis of operation's behavior.
operationId	A unique name for the operation. The operationId may be used by tools and libraries for creating clients.
externaDocs	Provides link to external resources for additional documentation if needed
tags	A list of tags that can be used to group operations by resources or any other qualifier.
consumes	A list of media types that the operation supports for requests. This overrides the *consumes* definition at the beginning of the document.
produces	A list of media types that the operation supports for responses. This overrides the *consumes* definition at the beginning of the document.
schemes	The transfer protocols used by the operation. This overrides the *schemes* definition at the beginning of the document.
deprecated	A boolean property specifying that the operation is deprecated.
parameters	A list of the parameters that the operation uses.
responses	A list of possible responses that are returned from invoking this operation.
security	The security schemes that the operation uses. This overrides the *security* definition at the beginning of the document.

Table 3.2: OAS Operation Object structure from [35]

response and the most common error responses.

The *Parameters Object* contains all the parameters that operations use. The specification, categorizes parameters into five types:

- *Path parameters* are used in cases where the parameter's value is part of operation's path. For example, in Listing 3.3 the */pets/*{*petId*} describes a template path containing a *Path parameter* (petId). In order to send a request for the specific endpoint, it is necessary to declare a value in the place of the parameter.

- *Query parameters* are appended to the url when sending a request. For example, in */pets?orderBy=name*, the *Query parameter* is orderBy.
- *Header parameters* define additional custom headers that may be sent in a request.
- *Body parameters* define the message content of a service request. A *Body parameter* refers to the data types defined in *Definitions Object* to describe the expected structure of the payload.
- *Form parameters* are a special case of *Body parameters*, that describe the content of a service request when the media types "*application/x-www-form-urlencoded*" and "*multipart/form-data*" are used as the content type of the request. Form parameters are used to allow files to be sent as part of the message in service requests.

Listing 3.4 demonstrates how parameters are declared in a OAS document. All *Parameter Objects* are defined in a similar manner. The *name* field declares the parameter's name, while the *in* field specifies the parameter's type (path, body, query, form, header). The *required* field specifies whether the parameter is required for the successful execution of an operation. For *Path parameters* the value of the *required* field must always be "true".

Listing 3.4: OAS Parameters definitions example from Swagger Petstore

```
parameters:
  userBody:     # A body parameter definition
    name: user
    in: body
    description: user to add to the system
    required: true
    schema:
      $ref: '#/definitions/User'

  tokenHeader: # A header parameter definition
    name: token
    in: header
    description: token to be passed as a header
    required: true
    type: array
```

```
    items:
      type: integer
      format: int64
    collectionFormat: csv

  queryID: # A query parameter definition
    name: id
    in: query
    description: ID of the object to fetch
    required: false
    type: array
    items:
      type: string
    collectionFormat: multi

  userPath:  # A path parameter definition
    name: username
    in: path
    description: username to fetch
    required: true
    type: string

  formFileAvatar:  # A form parameter definition
    name: avatar
    in: formData
    description: The avatar of the user
    required: true
    type: file
```

The *type* field describes the data type that each parameter accepts. A parameter's type is based on the data types supported by the JSON-Schema[6] (string, number, integer, boolean, array). Table 3.3 summarizes the properties that can also be used in the definition of a parameter. In addition, *Form parameters* can use the "file" data type, allowing files to be sent in service requests. Note that, the *type* field is not applicable in *Body parameters*. Instead, the *schema* field is used, referring to the data type structure that

[6] http://json-schema.org/latest/json-schema-core.html

describes the message content of an HTTP request.

Field	Description
format	Extends the format of the parameter's type. For example, a number can have either a "float" or a "double" format.
allowEmptyValue	A boolean property allowing a parameter to be sent with an empty value. It is valid only for query and form parameters. The default value is "false".
items	It is required for array types, specifying the data type of the items in the array.
collectionFormat	Specifies the format of items in an array type. The available values are: • csv - comma separated values (foo,bar). • ssv - space separated values (foo bar). • tsv - tab separated values (foo\tbar). • pipes - pipe separated values (foo|bar). • multi - generates multiple parameter instances for every item in the array (foo=bar&foo=baz). It is valid only for query and form parameters. The default value is "csv".
default	Contains the default value of the parameter if none is provided.
maximum	Specifies a maximum value in a numeric type.
exclusiveMaximum	A boolean property, representing whether the maximum value should be excluded or not.
minimum	Specifies a minimum value in a numeric type.
exclusiveMinimum	A boolean property, representing whether the minimum value should be excluded or not.
maxLength	Specifies the maximum length of a string.
minLength	Specifies the minimum length of a string.
pattern	Restricts a string to a particular regular expression.
maxItems	Specifies the maximum items of an array.
minItems	Specifies the minimum items of an array.
uniqueItems	A boolean property specifying that the items of an array should be unique.
enum	Specifies a fixed set of values.
multipleOf	Restricts a numeric value to be multiple of a given number.

Table 3.3: OAS parameter properties from [35]

The *Definitions Object* contains all data types (called *Schema Objects*) that are used to describe the request and response messages. A *Schema Object* can be a primitive (string, integer), an array or a model. For the definition of *Schema Objects* the specification is based on JSON Schema[7] and uses a predefined subset of it.

Field	Description
discriminator	Provides polymorphism support to models. The property's value specifies the model's property that is used to differentiate between the models that inherit this model.
readOnly	A boolean property that is used in model's properties to declare them as "read-only", meaning that the property may be sent as part of a response but mustn't be sent as part of the request.
xml	Describes the XML representation of model's properties.
externalDoc	Provides link to external resources for additional documentation if needed
example	Provides an example of an instance for the *Schema Object*

Table 3.4: Schema Object additional properties introduced by the OAS

Table 3.5 summarizes the properties that are adopted by the JSON-Schema and can be used for the description of *Schema Objects*. In addition, Table 3.4 contains the properties introduced by the specification in order to improve a *Schema Object* definition as well as support the representation of XML data types.

Listing 3.5 demonstrates a "User" model definition in the specification. A model is defined as an object (in *type* field) by specifying its properties within the *properties* field. A *required* field, describes the properties that are required to exist when the model is used by an operation. In addition, model's properties can be declared as "read-only" (*readOnly* field), or may contain various restriction regarding their type (e.g. maximum or minimum restrictions for numeric types).

[7] `http://json-schema.org/latest/json-schema-core.html`

Field	Description
format	Extends the format of the specified *type* field. For example, a number type can have either a "float" or a "double" format.
default	Contains the default value of the data type if none is provided.
multipleOf	Restricts a numeric value to be multiple of a given number.
maximum	Specifies a maximum value in a numeric type.
exclusiveMaximum	A boolean property, representing whether the maximum value should be excluded or not.
minimum	Specifies a minimum value in a numeric type.
exclusiveMinimum	A boolean property, representing whether the minimum value should be excluded or not.
maxLength	Specifies the maximum length of a string.
minLength	Specifies the minimum length of a string.
pattern	Restricts a string to a particular regular expression.
maxItems	Specifies the maximum items of an array.
minItems	Specifies the minimum items of an array.
uniqueItems	A boolean property specifying that the items of an array should be unique.
maxProperties	Specifies the maximum properties that an object type can have.
minProperties	Specifies the minimum properties that an object type can have.
required	Specifies a list with the properties of an object type that are required.
enum	Specifies a fixed set of data type's values.
items	It is required for array types, specifying the data type of the items in the array.
allOf	The property is used in order to extend the definition of an object type.
properties	Specifies the properties of an object type.
additionalProperties	Specifies whether an object type can contain additional properties of a specific data type.

Table 3.5: Schema Object properties adopted from the JSON-Schema

Models can be extended using the *allOf* property of JSON Schema. Note that, the *allOf* property is adopted by the JSON-Schema but its definition

and usage is different in the OAS. As seen in Listing 3.6, the *PremiumUser* model is formed by extending the *User* model (from Listing 3.5). The *PremiumUser* model inherits all the properties of *User* model while defining additional properties (*dateOfBirth*). This feature is very helpful as it allows the reuse of existing models and also provides a notion of inheritance among models.

Listing 3.5: OAS Model definition example from Swagger Petstore

```
definitions:
  User:  # A User object
    type: object
    properties:
      id:
        type: integer
        format: int64
        readOnly: true
      username:
        type: string
      firstName:
        type: string
      lastName:
        type: string
      email:
        type: string
      password:
        type: string
      address:
        $ref: '#/definitions/Address'
    required:
      - username
      - email
      - password
```

As the *allOf* property provides model extensibility, the use of the *discriminator* property offers polymorphism support among models. The *discriminator* property defines the model's property whose value will determine the model that is provided in requests or responses. The value of the property

must be the same with the name of models that are used. Listing 3.7 demonstrates how the *discriminator* property is used. In the example, a *Person* model is defined with its property *gender* used as a discriminator determining whether the *User* model is treated either as a *Male* or a *Female* model.

Listing 3.6: OAS Model composition example

```
definitions:
  PremiumUser: # A PremiumUser object extending the User object
    allOf:
      - $ref: '#/definitions/User'
      - type: object
        properties:
          dateOfBirth:
            type: string
            format: date
        required:
          - dateOfBirth
```

Listing 3.7: OAS Model polymorphism example

```
definitions:
  Person:    # An Person model that will be extended
    type: object
    properties:
      firstname:
        type: string
      lastname:
        type: string
      gender:    # The property is declared as a discriminator
        type: string
    required:
      - firstname
      - lastname
      - gender
    discriminator: gender

  Male:  # A Male model extending the Person Model
    description: A representation of a male person
    allOf:
      - $ref: '#/definitions/Person'
```

```
    - type: object
      properties:
        height:
          type: integer
          description: height in cms
        weight:
          type: integer
          description: weight in kgs
      required:
        - height
        - weight

Female:    # A Female model extending the Person Model
  description: A representation of a female person
  allOf:
    - $ref: '#/definitions/Person'
    - type: object
      properties:
        eyesColor:
          type: string
      required:
        - eyesColor
```

The OpenAPI Specification provides a basic format for describing RESTful services (Section 2.3). In addition, the specification offers an extension mechanism that allows the format to be enriched with additional features that may be considered useful in certain cases. The extension properties are declared using the prefix "x-" and can have any valid JSON value. This feature of the specification is used in order to semantically enrich the service descriptions as described in the following section.

3.3 Enriching the OpenAPI Specification

The OpenAPI Specification is mainly targeted to provide a human-friendly environment for discovering and consuming RESTful services (Section 2.3).

The specification offers a syntactic service description that is enriched with text descriptions (through the extensive use of *description* property) so that users may easily discover and understand the functionalities of the service and interact with it.

Despite its user friendliness the specification is not machine understandable, thus limiting the availability of tools that facilitate machine tasks such as service discovery, which is essential especially in Cloud environments. Using the extension mechanism that the OAS offers, we introduced some additional properties that semantically annotate an OpenAPI service description. Using semantic annotations various parts of an OpenAPI service description may obtain a semantic content that machines may understand and interpret similarly to humans. Table 3.6 summarizes the extension properties we defined for the semantic enrichment of OpenAPI service descriptions that form the Semantic OpenAPI Specification.

The *x-refersTo* extension property specifies the association between an OAS element and a concept in a semantic model. The property accepts a URI, that represents the concept in the semantic model. The property can be used in *Schema Objects* and *Parameters Objects*.

Listing 3.8 demonstrates how the property *x-refersTo* is used to semantically annotate a *Person* model and its properties. The model *Person* is associated with the concept *Person*[8] from the Schema.org vocabulary, while its properties are also associated with the corresponding properties of the semantic concept.

However, it is not always possible for a model to have a relation with an equivalent semantic concept, as a model may have a narrower meaning over the referenced semantic concept. For example, consider the *Person* model in Listing 3.8. If the model was defined to describe a specific group of people, e.g. teenagers, it would be inappropriate to associate the model with a generic

[8] `http://schema.org/Person`

Property	Applies to	Description
x-refersTo	Schema Object & Parameter Object	It specifies the concept in a semantic model that best describes an OAS element.
x-kindOf	Schema Object	It specifies a specialization that exists between an OAS element and a concept in a semantic model. The property is mainly used to declare that a concept in a semantic model is too generic to describe the specified model, whereas a more specific subtype (if existed) should be considered more appropriate.
x-mapsTo	Schema Object & Parameter Object	It indicates that an OAS element is semantically similar with another OAS element.
x-collectionOn	Schema Object	It indicates that a model describes a collection over a specific property.
x-onResource	Tag Object	It Denotes that the specific *Tag Object* refers to a resource described by a *Schema Object*.
x-operationType	Operation Object	The property is used providing semantic information on the type of operation. The subtypes of the *Action* concept in the Schema.org vocabulary can be used as values of the property.

Table 3.6: OAS extension properties for semantic annotations

concept such as the *Person* type of the Schema.org vocabulary, whereas a more specific subtype (if existed) should be considered more appropriate. In this case, the *x-kindOf* extension property is used denoting that the model is actually a subclass of the referred semantic concept. The property accepts a URI, representing the concept in the semantic model, and is only used for models in *Definitions Object*.

Listing 3.8: Semantic annotations for OAS in Definitions Object

```
definitions:
  Person:
    type: object
    x-refersTo: http://schema.org/Person
    properties:
      firstname:
        type: string
        x-refersTo: http://schema.org/givenName
      lastname:
        type: string
        x-refersTo: http://schema.org/familyName
      gender:
        type: string
        x-refersTo: http://schema.org/gender
    required:
      - firstname
      - lastname
      - gender
```

The *x-mapsTo* extension property is used to define OAS elements that share the same semantics. In a OAS service document, there are many elements that share the same semantic information that a human may understand but a machine cannot. Listing 3.9 demonstrates an excerpt from the Swagger Petstore[9] service description. As seen, there is a semantic similarity among the *petId* property of the *Order* model and the *id* property of the *Pet* model . In addition, this semantic similarity also exists between the *status* property of the *Pet* model and the *status* query parameter.

Listing 3.9: Excerpt from Swagger Petstore OAS service description

```
parameters:
  statusQuery:
    name: status
    in: query
    description: Status values that need to be considered for
       filter
```

[9] http://petstore.swagger.io/

```
    required: true
    type: array
    items:
      type: string
      enum:
        - available
        - pending
        - sold
      default: available
    collectionFormat: multi
    x-mapsTo: '#/definitions/Pet.status'

definitions:
  Pet:   # A Pet model definition
  type: object
  required:
    - name
    - photoUrls
  properties:
    id:
      type: integer
      format: int64
    category:
      $ref: '#/definitions/Category'
    name:
      type: string
      example: doggie
    photoUrls:
      type: array
      items:
        type: string
    tags:
      type: array
      items:
        $ref: '#/definitions/Tag'
    status:
      type: string
      description: pet status in the store
      enum:
```

```
      - available
      - pending
      - sold

  Order: # An Order model definition
  type: object
  properties:
    id:
      type: integer
      format: int64
    petId:
      type: integer
      format: int64
      x-mapsTo: '#/definitions/Pet.id'
    quantity:
      type: integer
      format: int32
    shipDate:
      type: string
      format: date-time
    status:
      type: string
      description: Order Status
      enum:
        - placed
        - approved
        - delivered
    complete:
      type: boolean
      default: false
```

A human may easily infer these semantic similarities, either by the element names or by the description that may be provided. However, in order for a machine to act similarly to a human it is necessary to provide additional information, that specifies these relations. This is the case of the *x-mapsTo* property applicability, as can be seen in Listing 3.9.

The *x-collectionOn* extension property is used to indicate that a model in

Definitions Object is actually a collection. A collection (or a list) is defined in OAS using the *array* type as seen in Listing 3.10. However, it is very common a collection's definition to be enriched with additional properties and thus to be defined as an *object* type.

Listing 3.10: A simple collection definition

```
definitions:
  PetCollection:   # A Pet Collection definition
    type: array
    items:
      $ref: '#/defintions/Pet'
```

Listing 3.11 represents a model definition that is, in fact, a collection containing items of the *Pet* model, while the *totalItems* property contains the total number of elements in the collection, serving as metadata information. Using the *x-collectionOn* property a model is considered as a collection. The property's value is the name of the property that contains the collection's items.

Listing 3.11: A model definition representing a collection

```
definitions:
  PetCollection:   # A Pet Collection definition
    x-collectionOn: pets
    type: object
    properties:
      pets:
        type: array
        items:
          $ref: '#/defintions/Pet'
      totalItems:
        type: integer
```

The *x-onResource* extension property is used in *Tag Objects* in order to specify the resource that a tag refers. As stated in the previous section, tags are used to group operations either by resources or any other qualifier. A human may understand the purpose of a tag, by reading its description. In

addition, if the tag is used to group operations by resources, a human may recognize that the referred resource is described by a *Schema* Object in *Definitions Object*. For example, in the Swagger Petstore[10] service description, the tag *pet* groups all the operations which are related with the pet resource, that is described from the "Pet" model (the definition of the Pet model is presented in Listing 3.9). As Figure 3.2 illustrates, tags provide a more comprehensive visualization of service's functionalities, allowing users to easily browse and discover the operations that refer to a specific resource.

Listing 3.12: Excerpt from Swagger Petstore OAS service description

```
tags:
  - name: pet
    description: Everything about your Pets
    externalDocs:
      description: Find out more
      url: 'http://swagger.io'
    x-onResource: '#/definitions/Pet'

paths:
  /pet:
    post:
      tags:
        - pet
      summary: Add a new pet to the store
      operationId: addPet
      consumes:
        - application/json
        - application/xml
      produces:
        - application/xml
        - application/json
      parameters:
        - $ref: '#/parameters/newPet'
      responses:
        '405':
          description: Invalid input
          security:
```

[10] http://petstore.swagger.io/

```
    - petstore_auth:
      - 'write:pets'
      - 'read:pets'
```

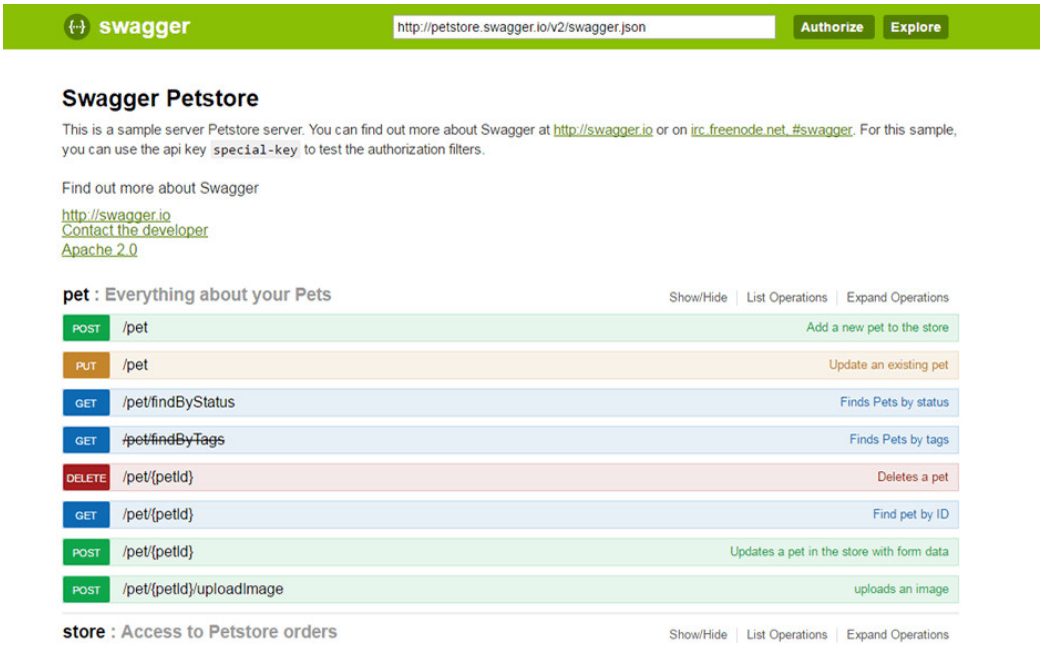

Figure 3.2: Swagger Petstore operations grouped by tag "pet"

This information a machine is not able to infer and thus the *x-onResource* property is introduced, by associating the tag with a *Schema* Object that describes a specific resource. Note that, *Schema Objects* can be semantically annotated, using the *x-refersTo* property and thus refer to a semantic content with a specific meaning. Listing 3.12 demonstrates the usage of the *x-onResource* property on an excerpt of the Swagger Petstore service description. The property is assigned on a *pet* tag that provides information regarding the operations that are available for the resource that is described by the *Pet* model in *Definitions Object* (the definition of the Pet model can be seen in Listing 3.9).

Finally the *x-operationType* extension property is used in order to semantically specify the type of an *Operation object*. A request (an operation for

OAS) is characterized by the HTTP method that it uses. However, the semantics of the HTTP methods are too generic and in the context of a service they may have a more specific meaning. For example, in the Swagger Petstore[11] service description a GET request on the path */pet/findByStatus* (Listing 3.13) is a search operation of pets based on their status. A human may refer to the description of the operation in order to understand its intended purpose, but a machine needs additional information as the HTTP method itself cannot provide such specific information.

Listing 3.13: Excerpt from Swagger Petstore OAS service description

```
paths:
  /pet/findByStatus:
    get:
      x-operationType: 'http://schema.org/SearchAction'
      tags:
        - pet
      summary: Finds Pets by status
      description: Multiple status values can be provided with
         comma separated strings
      operationId: findPetsByStatus
      produces:
        - application/xml
        - application/json
      parameters:
        - $ref: '#/parameters/statusQuery'
      responses:
        '200':
          description: successful operation
          schema:
            $ref: '#/definitions/PetCollection'
        '400':
          description: Invalid status value
      security:
        - petstore_auth:
          - 'write:pets'
          - 'read:pets'
```

[11] http://petstore.swagger.io/

This information is specified by the use of the *x-operationType* as seen in Listing 3.13. The value of the property is a URL pointing on the concept that semantically describes the operation type. The *Action*[12] type of the Schema.org vocabulary provides a detailed hierarchy of *Action* subtypes that can be used by the property.

3.4 The OpenAPI Ontology

The extension properties described in the previous section allow an OpenAPI document to become machine-understandable. In this section, we present an ontology[13], in which any OpenAPI document can be transformed in order to benefit from Semantic Web tools such as reasoners and query languages for service discovery and for enabling service orchestration. Figure 3.4 represents the structure of the OpenAPI ontology that we propose for the transformation of the OpenAPI documents.

The idea of the OpenAPI ontology stems from [34], where the authors have demonstrated that it is possible an OpenAPI service description to be transformed into an ontology. Their proposed ontology, presented in Figure 3.3, is a direct mapping of the OAS objects and properties into the corresponding defined classes and object/data properties of their ontology. In addition, they suggest using annotations on *Schema Objects* and *Parameter Objects* of an OpenAPI service description by adding a link, inside angle brackets ("<>"), in their description property that points on a concept in a semantic model. However, since their ontology is a direct mapping of the OAS objects and properties, these annotations are not handled properly. From the given examples[14] these annotations are still part of the text description properties and thus additional processing is required for a machine to extract and use

[12] `http://schema.org/Action`

[13] All examples are provided using the Turtle syntax (`https://www.w3.org/TR/turtle/`)

[14] `https://github.com/fathoni/swg-sample`

them.

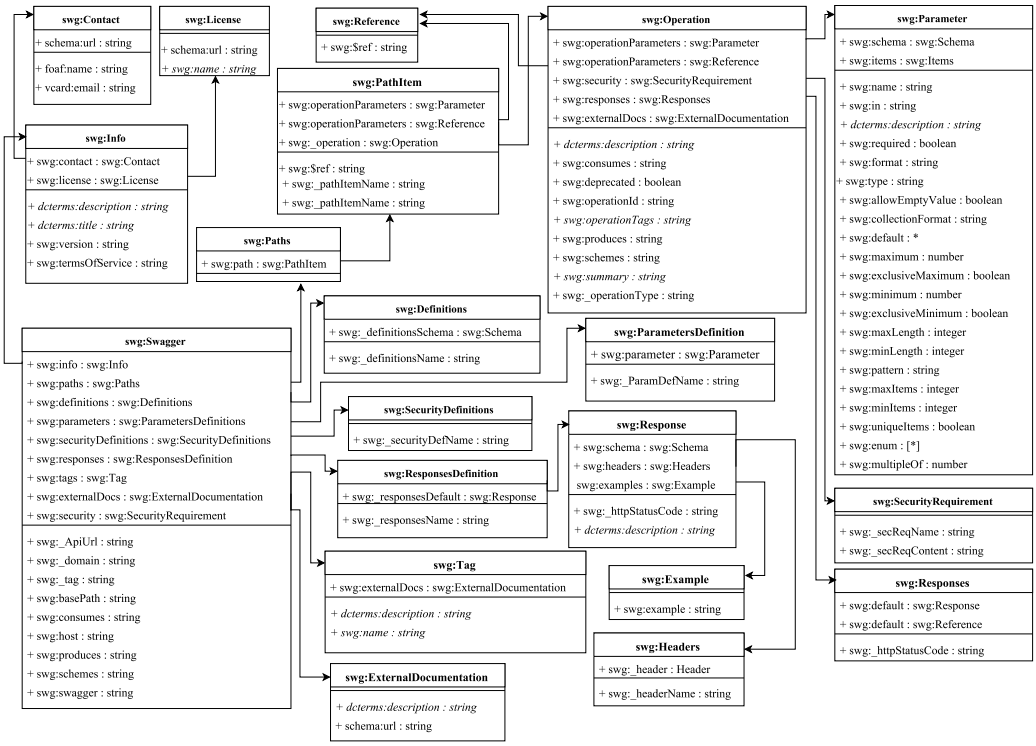

Figure 3.3: The proposed ontology for OAS from [34]

On the other hand, our proposed OpenAPI ontology is based on the structure of the OpenAPI Specification, and captures all the information that is provided by an OpenAPI service description, as well as the semantic annotations introduced previously. The rest of the section describes the transformation of an OAS service description into our proposed ontology. The *Document* class represents the documentation and the entry point of the service. Similarly to the OAS structure, it provides general information (*Info* class) regarding the described service, and also specifies the service paths and the entities that it supports.

A service path is described by the *Path* class containing the relative path for the service endpoint (using the *pathName* property). The *Operation* class is similar to the *Operation object* in the OAS structure. It contains all the

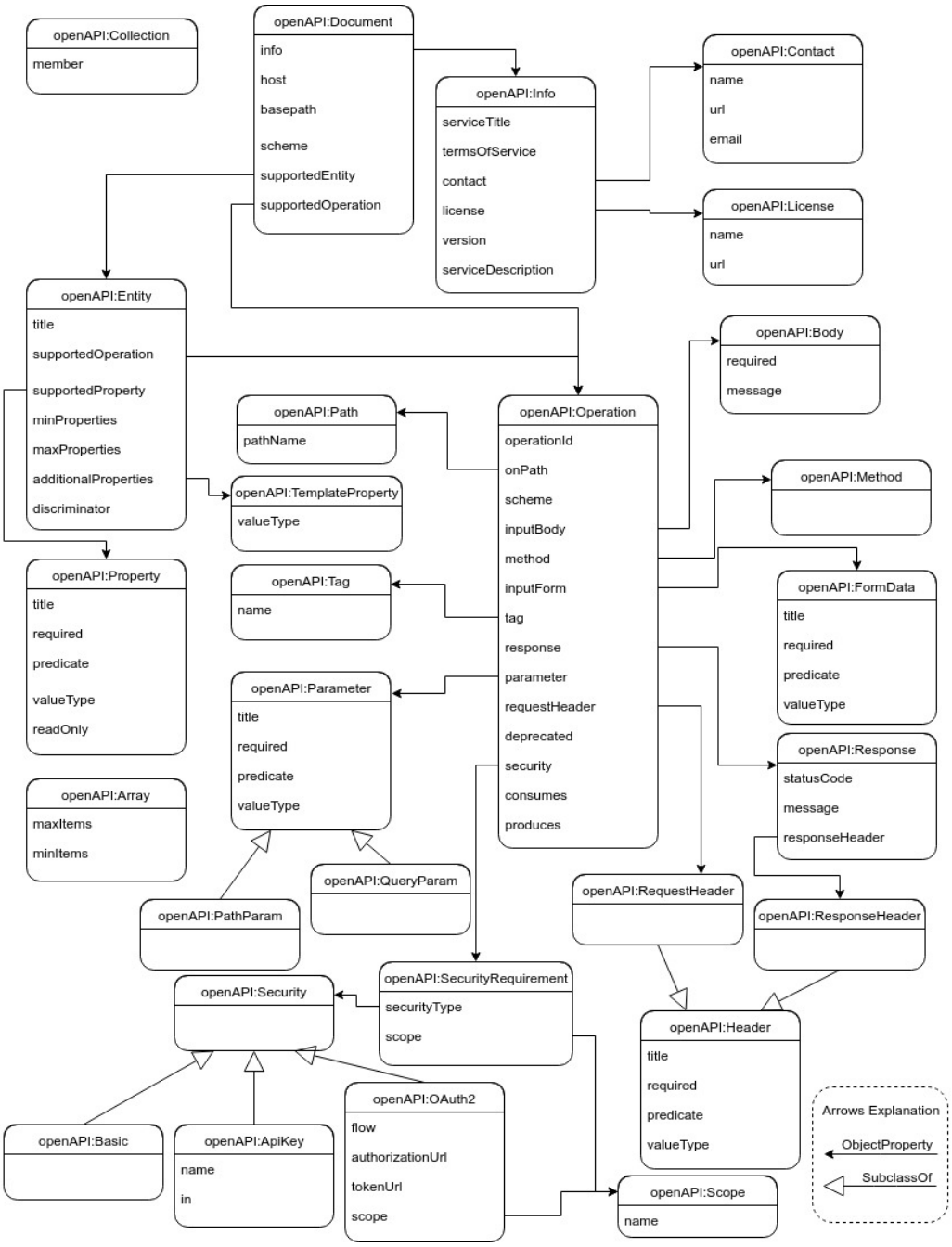

Figure 3.4: The OpenAPI Ontology

needed information in order to construct HTTP requests to the service as well as the defined HTTP responses.

Listing 3.14 demonstrates how an OAS *Path Item* and *Operation* are defined in our proposed OpenAPI ontology, using the example of the Swagger Petstore in Listing 3.13. As seen, the mapping of the OAS elements into the ontology is straightforward. A *Path* individual, named "*path2*", defines the "*/pets/findByStatus*" service path, as well as the operations it supports. The "*path2_op1*" is an *Operation* individual containing all the information that also exists in the *Operation object*. Note that the "*path2_op1*" individual is also considered as an individual of the *SearchAction* type of the Schema.org vocabulary, due to the existence of the *x-operationType* extension property.

Listing 3.14: Representation of an OAS Path Item and Operation in the openAPI ontology

```
@prefix : <http://www.intelligence.tuc.gr/PetStoreOntology#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix openAPI: <http://www.intelligence.tuc.gr/openAPI#> .
@base <http://www.intelligence.tuc.gr/PetStoreOntology> .

<http://www.intelligence.tuc.gr/PetStoreOntology> rdf:type owl:
    Ontology ;
  owl:imports <http://www.intelligence.tuc.gr/2016/openAPI> .

...

:path2 rdf:type openAPI:Path ;
  openAPI:pathName "/pets/findByStatus" ;

:path2_op1 rdf:type openAPI:Operation, <http://schema.org/
    SearchAction> ;
  openAPI:onPath :path2
  openAPI:method openAPI:Get ;
  openAPI:tag :tag_pet ;
  openAPI:parameter :query_status ;
  openAPI:response :path2_op1_200 , :path2_op1_400 ;
```

```
  openAPI:security :oauth_requirement ;
  openAPI:operationId "findPetsByStatus" ;
  openAPI:produces "application/json" , "application/xml" ;
  openAPI:summary "Finds Pets by Status" ;
  openAPI:description "Multiple status values can be provided
      with comma seperated strings" .

:path2_op1_200 rdf:type openAPI:Response ;
  openAPI:message :PetCollection ;
  openAPI:statusCode 200 ;
  openAPI:description "successful operation" .

:oauth_requirement rdf:type openAPI:SecurityRequirement ;
  openAPI:securityType :petstore_oauth ;
  openAPI:scope :read_pets , :write_pets .

...
```

The *Security* class defines the security schemes that the specification supports. The *Operation* class refers to the defined security scheme through the *SecurityRequirement* class, which in the case of the OAuth2 security scheme contains the scopes that are needed in order to successfully execute the operation. In Listing 3.14 the "*path2_op1*" operation refers to a *SecurityRequirement* individual, specifying an OAuth2 security scheme ("*petstore_oauth*" individual) and the corresponding scopes ("*read_pets*" and "*write_pets*" individuals).

The most challenging part in the design of the OpenAPI ontology was the representation of *Schema Objects* that describe the input and output messages of operations in an OpenAPI service document. Since *Schema Objects* can be associated with semantic concepts, it is necessary to express them in terms of classes, object and data properties, and also capture all the information provided from the OpenAPI document. Therefore, we have adopted an approach similar to the one used in the Hydra Core Vocabulary (Section 2.7.4).

The *Entity* class represents the classes that describe the models of an OpenAPI service description and captures all the information contained in the models they describe. It allows to specify the operations that are related to a class (*supportedOperation*), which comes from the use of the *x-onResource* extension property. In addition, it determines whether a class may contain additional properties (*additionalProperties*) of a specific type, described by a *TemplateProperty*, and specifies the properties that a class contains (*supportedProperty*).

The *Property* class represents the properties that a class definition contains. A *Property* specifies the supported property (*predicate*) as well as its expected value (*valueType*) and any restrictions that may exist (e.g. a maximum value for a numeric property). In addition, the *Property* indicates whether the supported property is required, read-only in the class definition. The *Array* class is introduced in order to distinguish the properties that contain a list of values (properties of type "array").

Listing 3.15: Representation of an OAS model in the openAPI ontology

```
...

<http://schema.org/Person> rdf:type openAPI:Entity , owl:Class ;
  openAPI:title Person ;
  openAPI:supportedProperty _:firstname , _:lastname , _:gender.

_:firstname rdf:type openAPI:Property ;
  openAPI:title firstname ;
  openAPI:predicate <http://schema.org/givenName> ;
  openAPI:valueType xsd:string ;
  openAPI:required true .

_:lastname rdf:type openAPI:Property ;
  openAPI:title lastname ;
  openAPI:predicate <http://schema.org/familyName> ;
  openAPI:valueType xsd:string ;
  openAPI:required true .

_:gender rdf:type openAPI:Property ;
```

```
  openAPI:title gender ;
  openAPI:predicate <http://schema.org/gender> ;
  openAPI:valueType xsd:string ;
  openAPI:required true .

...
```

Listing 3.15 demonstrates how the "Person" model in Listing 3.8 is represented in the OpenAPI ontology. Note that the "Person" model contains references to the Schema.org vocabulary using the *x-refersTo* extension property. As seen, the Person type of the Schema.org vocabulary is also considered as an *Entity* describing the Person model and its properties in the OpenAPI ontology. In case, the *x-refersTo* extension property is missing from a model definition and its properties, then the corresponding class and properties will be defined in the OpenAPI ontology. Thus, the newly defined classes and properties can be used as references to other service descriptions if needed.

A model defined using the "allOf" property, is represented in the OpenAPI ontology as a subclass of the the model that is extended. Listing 3.16 demonstrates how the models of Listing 3.7 are represented in the OpenAPI ontology. As seen, the "Male" and "Female" classes are defined as subclasses of the "Person" class due to the existence of the "allOf" property in model definitions. In addition, the "Male" and "Female" classes also inherit all the properties of the "Person" model.

A subclass is also defined in the OpenAPI ontology, when the *x-kindOf* extension property is used in a model definition. In this case, the model that is annotated with the *x-kindOf* extension property is represented in the OpenAPI ontology as a subclass of the referenced semantic concept.

Listing 3.16: Representation of OAS models defined using the "allOf" property in the openAPI ontology

```
...

:Person rdf:type openAPI:Entity, owl:Class ;
  openAPI:title Person ;
  openAPI:discriminator gender ;
  openAPI:supportedProperty _:firstname , _:lastname , _:gender.

:Male rdf:type openAPI:Entity ;
  rdfs:subClassOf :Person ;
  openAPI:title Male ;
  openAPI:supportedProperty _:firstname , _:lastname , _:gender,
                _:height, _:weight .

:Female rdf:type openAPI:Entity ;
  rdfs:subClassOf :Person ;
  openAPI:title Female ;
  openAPI:supportedProperty _:firstname , _:lastname , _:gender,
                _:eyesColor .

...
```

Collections are represented in the OpenAPI ontology through the *Collection* class by specifying the members (*member*) of a collection. Listing 3.17 demonstrates the "PetCollection" of Listing 3.11 representation in the OpenAPI ontology. A "PetCollection" class is defined in the OpenAPI ontology as a subclass of the *Collection* class. In addition, the "PetCollection" is also considered an *Entity* in order to describe its properties. Note that the "PetCollection" model in Listing 3.11 is defined as an object, and the *x-collectionOn* extension property is used in order to specify that the model is in fact a collection whose members are described by the "pets" property. Without the *x-collectionOn* extension property, the "PetCollection" model would be defined as a simple class without any reference of being a collection.

Listing 3.17: Representation of collections in the openAPI ontology

```
...

:PetCollection rdf:type openAPI:Entity ;
  rdfs:subClassOf openAPI:Collection ;
  openAPI:title PetCollection ;
  openAPI:supportedProperty _:pets , _:totalItems .

_:pets rdf:type openAPI:Property, openAPI:Array ;
  openAPI:title pets ;
  openAPI:predicate openAPI:member ;
  openAPI:valueType :Pet .

_:totalItems rdf:type openAPI:Property ;
  openAPI:title totalItems ;
  openAPI:predicate :totalItems ;
  openAPI:valueType xsd:integer .

...
```

A closer look on the definition of the *Operation* class reveals some differences that exist with the OAS structure. While in the specification a header or the body of a HTTP request are treated as *parameters objects*, in the OpenAPI ontology there is a clear distinction of the various parameter types, by defining different classes for every parameter type.

As seen in Figure 3.4, the *Header* class, which is further distinguished in *ResponseHeader* and *RequestHeader* classes, contains all the definitions of header parameters that are used in HTTP requests and responses. The *Body* class defines the body parameters describing the expected message content of a HTTP request. The *FormData* class represents the form parameters that are defined by the specification describing the message content of a HTTP request when the "*application/x-www-form-urlencoded*" and "*multipart/form-data*" media types are used. Finally, the *Parameter* class defines all parameters that are attached to operation's URL. The class is further organized in *PathParameter* and *QueryParameter* classes that refer

to the corresponding path and query parameters of the specification.

Parameters of an OpenAPI service description (excluding body parameters) are represented in the OpenAPI ontology similarly to a model's property definition. Listing 3.18 demonstrates a query parameter definition (from Listing 3.9) in the OpenAPI ontology. A status *QueryParam* is defined containing the same information that is provided in the OpenAPI service description. As seen, the same property that describes the status property in the Pet model, is also used to describe the status *QueryParam*. This is due to the existence of the *x-mapsTo* extension property, stating that the query parameter shares the same semantic information with the referred property.

Listing 3.18: Representation of Query parameter in the OpenAPI ontology

```
...

:statusQuery rdf:type openAPI:QueryParam, openAPI:Array ;
  openAPI:title status ;
  openAPI:predicate :pet_status ;
  openAPI:valueType xsd:string ;
  openAPI:enumData available , pending , sold.
  openAPI:defaultData available ;
  openAPI:collectionFormat multi ;
  openAPI:required true .

:Pet rdf:type openAPI:Entity , owl:Class ;
  openAPI:title Pet ;
  openAPI:supportedProperty _:petId , _:petCategory , _:petName
      , _:petPhotoUrls , _:petTags , _:petStatus .

_:petStatus rdf:type openAPI:Property ;
  openAPI:title status ;
  openAPI:predicate :pet_status ;
  openAPI:valueType xsd:string ;
  openAPI:enumData available , pending , sold.

...
```

4

Use Case: FIWARE

The Semantic OpenAPI Specification presented in the previous chapter allows for both, syntactic and semantic description of RESTful services, which can be used for the description of Cloud services based on REST principles. In addition, the proposed OpenAPI ontology enables the exploitation of Semantic Web technologies such as reasoners and query languages that may assist in the creation of service discovery and service orchestration mechanisms.

In the following sections, we discuss how our proposed approach can be applied in a Cloud provider, such as the FIWARE platform. However, this is not a trivial process since there are many issues to resolve. Firstly, the existing descriptions of FIWARE's services (called Generic Enablers) are incomplete and vague, not allowing the RESTful APIs to be fully described in OAS. In addition, the semantic annotation of OpenAPI service descriptions requires for ontologies that describe the domain of a service, which FIWARE should provide. A closer collaboration with the FIWARE community can resolve these issues and allow FIWARE to offer better service descriptions, as well as tools that improve users' interaction with the platform.

4.1 The FIWARE Catalogue

The FIWARE platform publishes its offering Generic Enablers (GEs) in a public catalogue, called the FIWARE Catalogue[1]. The FIWARE Catalogue is a web interface, that organizes GEs based on FIWARE's technical chapters (Section 2.4.3). As Figure 4.1 illustrates, a user can navigate through a list of FIWARE's offering GEs and discover their intended purpose and functionality.

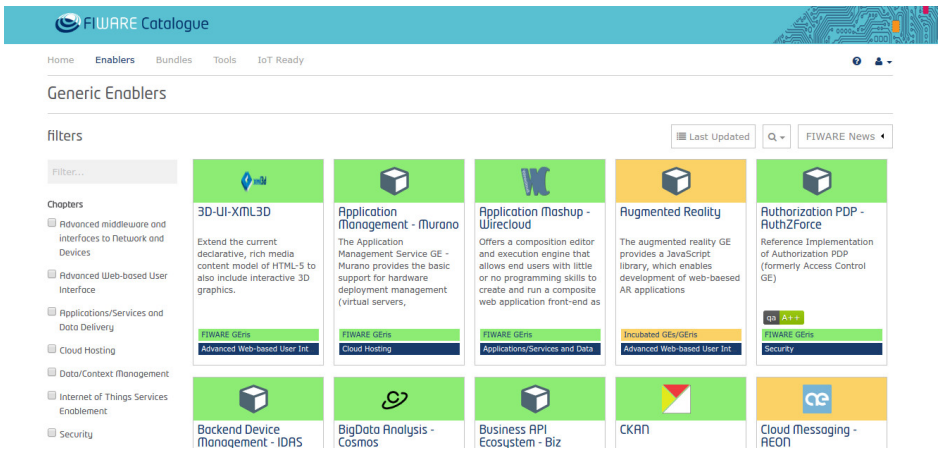

Figure 4.1: FIWARE Catalogue web interface

Once a GE is selected, the user is taken to the GE detailed view. In this page, as Figure 4.2 demonstrates, a user finds all the information that is related with a GE. The page provides contact information for the organization or the individual responsible for the specific GE and additionally a menu with links to other relevant information.

- The "*Overview*" item is the main page of a GE detailed view and provides a more detailed description of the GE, the purpose and the benefits of using it.
- The "*Creating Instances*" item provides instructions for creating an instance in the FIWARE platform for this specific GE.

[1] `https://catalogue.fiware.org/`

- The "*Documentation*" item contains all the documentation of the GE. Particularly, in most cases it provides links to external sources such as user manuals, API descriptions and reference architecture.
- The "*Downloads*" item provides links where the source code of the GE is stored, as well as example packages.
- The "*Instances*" item registers any available public instance of the specific GE that a user can use.
- The "*Terms and conditions*" item contains the license and the policy that a user must adhere in order to use the specific GE.

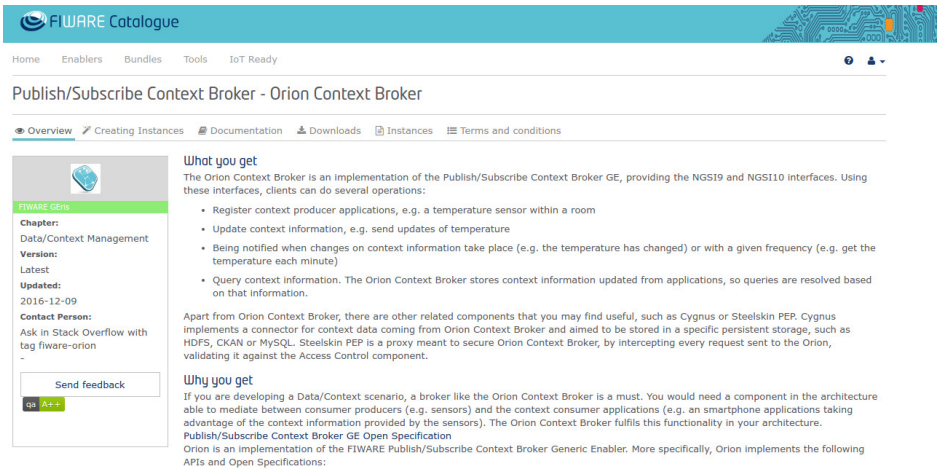

Figure 4.2: FIWARE Catalogue GE detailed view

The FIWARE Catalogue in its current structure cannot offers only a categorization of GEs in FIWARE's technical chapters. Users need to discover on their own what a specific GE offers and how it can be accessed and used. In fact, a catologue's entry acts as a mediator that takes users to the GE's project repository (such as GitHub[2]), where the actual documentation and description of the service is kept. As mentioned in Section 2.4.3, GEs are considered as software modules that offer various functionalities along with protocols and interfaces for their communication.

The majority of GEs in the FIWARE platform are provided by means of

[2] `https://github.com/`

Web services, however, there are also GEs offering a Web user interface (UI). It is also very common, GEs to provide both a Web service and a UI, as well as SDKs in specific programming languages. FIWARE doesn't impose any particular guidelines for documenting GEs, which is actually left to the GE provider's discretion and capability to decide the best way to do this. Therefore, it is interesting to review how GE providers tend to describe the offered services.

Cloud Hosting

The *Cloud Hosting* chapter contains GEs which are responsible for the provisioning of virtual machines, as well as for associating compute, storage and network resources to virtual machines. Most of the GEs are provided by OpenStack (the cloud platform FIWARE is built upon) and the FIWARE Catalogue refers users to the official documentation and support offered by OpenStack. OpenStack used to describe its RESTful APIs using WADL, however, in 2016 OpenStack Summit in Austin[3], Texas, the OpenStack API-WG defined OAS as a standard API documentation way and announced its adoption for the documentation and description of OpenStack services.

Apart from offering OpenStack services, the *Cloud Hosting* chapter includes also GEs implemented and provided by the FIWARE community (Table 4.1). *Bosun*, *Pegasus* and *Sagitta* GEs are offering additional functionalities and features for the management of infrastructure resources. The GEs are provided and maintained by the same provider (Telefónica I+D[4]) and expose a RESTful API that is described using API Blueprint[5].

On the contrary, the *Docker* GE, which provides the basic docker container hosting capabilities on the FIWARE platform, offers a RESTful API[6]

[3] https://www.openstack.org/summit/austin-2016/
[4] http://www.tid.es/
[5] https://apiblueprint.org/
[6] https://docs.docker.com/engine/api/v1.25/

Generic Enabler	Offers	API Description
Policy Manager (Bosun)	RESTful API	API Blueprint
PaaS Manager (Pegasus)	RESTful API	API Blueprint
Software Deployment & Configuration (Sagitta)	RESTful API	API Blueprint
Docker	RESTful API & SDKs	OpenAPI Spec.

Table 4.1: FIWARE GEs of the Cloud Hosting chapter

described using OAS. The API allows users to control every aspect of Docker, and can be used for the development of tools for managing and monitoring applications running on Docker. Figure 4.3 illustrates how Docker's OpenAPI description is visualized using ReDoc[7], a tool similar to Swagger UI.

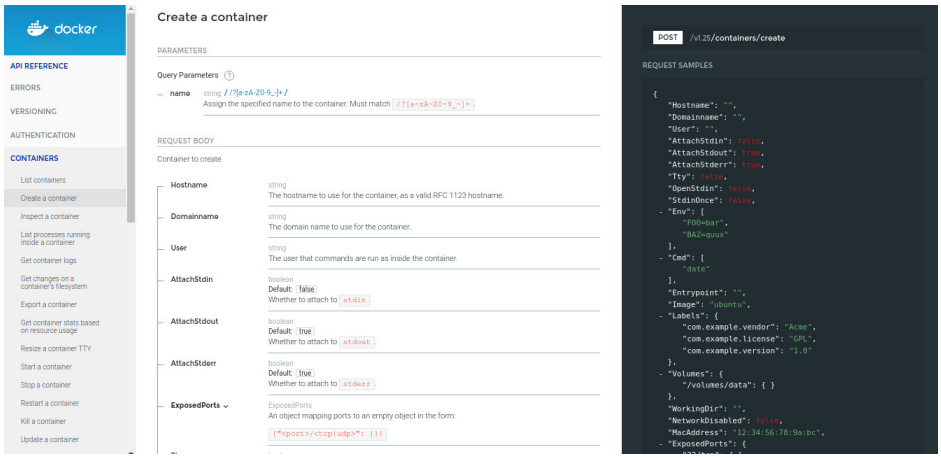

Figure 4.3: The Docker API documentation based on ReDoc

Data/Context Management

The *Data/Context Management* chapter contains GEs that provide various functionalities regarding the management and processing of data. Table 4.2 summarizes the offered GEs and illustrates the given interfaces to access their functionalities. In addition, the *BigData analysis (Cosmos)* GE is included in

[7] https://github.com/Rebilly/ReDoc

the chapter and offers a Hadoop[8] cluster as well as a Storm[9] cluster allowing the analysis of data in real-time and massive scale. The main purpose of the *Cosmos* GE is to offer a framework through which service providers may expose services that application developers may use to perform such data analysis.

Generic Enabler	Offers	API Description
CKAN	RPC-style API & Web UI	Text description
Stream-oriented (Kurento)	JSON-RPC 2.0 API & SDKs	Text description
Publish/Subscribe Context Broker (Orion)	RESTful API	API Blueprint
Complex Event Processing (CEP)	RESTful API & Web UI	API Blueprint

Table 4.2: FIWARE GEs of the Data/Context Management chapter

The *CKAN* GE offers a Web interface through which a user can manage and organize datasets and their resources. Moreover, a RPC-style API[10] is provided, given a text description of its functionalities. Overall, the description of the API is poor as it doesn't provide sufficient information regarding the structure of HTTP requests and responses in order to allow a user interact with it.

The *Kurento* GE provides a framework for the development of interactive multimedia applications. The Kurento media server is accessed through an API via WebSockets, which is based on JSON-RPC 2.0 protocol[11]. A text description of the API is provided, which is quite informative as it provides a complete description of API's request and response elements. Moreover, Java and JavaScript clients are available for developers to connect to Kurento media server.

[8] `http://hadoop.apache.org/`
[9] `http://storm.apache.org/index.html`
[10] `http://docs.ckan.org/en/latest/api/index.html`
[11] `http://www.jsonrpc.org/specification`

The *Orion Context Broker* GE is one of the most important and commonly used GEs in the FIWARE platform. It allows the management of context information (data which are relevant to a particular entity) including updates, queries, registrations and subscriptions. The GE offers a RESTful API implementing the NGSI9/10[12] specification that is described using API Blueprint, as well as an informative user manual. Note that, a new version of *Orion's* API is implemented and is about to replace the existing one.

The *CEP* GE allows the analysis of event data in real time. A user may interact with the GE through a Web interface or a RESTful API. Although an informative description of the technology used by the GE is provided, and a guide for interacting with the Web interface, the description of the API is incomplete. The API is described using API Blueprint, however, it doesn't provide sufficient information regarding the structure of HTTP requests and responses or even examples.

Internet of Things (IoT) Services Enablement

The *Internet of Things (IoT) Services Enablement* chapter contains GEs that allow devices and sensors to be registered and discovered by other FIWARE services in order to obtain access to the information they collect. In most cases, GEs act as gateways gathering information from the registered sensors or devices and offering it as NGSI context information, that is the FIWARE standard data exchange model. The offered chapter's GEs are specified in Table 4.3, which also describes the given interfaces for interacting with them.

The *IDAS* GE enables IoT devices and sensors to be connected to FIWARE-based services and provides the tools for translating IoT specific protocols to the NGSI context information protocol. Thus, the information retrieved from devices can be used by other GEs, such as the *Orion Context Broker* GE.

[12] `https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE_NGSI_Open_RESTful_API_Specification`

Generic Enabler	Offers	API Description
Backend Device Management (IDAS)	RESTful API	API Blueprint
IoT Discovery	RESTful APIs & Web UI	API Blueprint & Text Description
IoT Broker	RESTful API	API Blueprint
IoT Data Edge Consolidation (Cepheus)	RESTful API	Text Description & API Blueprint

Table 4.3: FIWARE GEs of the Internet of Things (IoT) Services Enablement chapter

The *IDAS* GE offers a RESTful API, called IoT Agent Provision API, which is described using API Blueprint, allowing the registration of new devices and the management of existing ones.

The *IoT Discovery* GE allows the registration and discovery of IoT sensors and devices using either the NGSI context information protocol or as Linked Data, based on the IoT-A[13] ontologies. Therefore, the IoT Discovery GE exposes two modules, an NGSI server and the Sense2Web platform. The NGSI server is a repository for the storage of NGSI entities exposing a RESTful API that implements the NGSI-9[14] specification described in API Blueprint. The Sense2Web platform is also a repository that contains the semantic descriptions of IoT sensors and devices, offering a Web interface as well as a RESTful API. The Sense2Web API is described as text, and the given documentation is incomplete as it merely states the available paths which are accompanied by a small description of their intended functionality.

The *IoT Broker* GE provides a repository allowing the collection and aggregation of context information from IoT devices and sensors. The GE exposes a RESTful API implementing the NGSI-10[15] specification described in API Blueprint.

[13] www.iot-a.eu

[14] https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE_NGSI-9_Open_RESTful_API_Specification

[15] https://forge.fiware.org/plugins/mediawiki/wiki/fiware/index.php/FIWARE_NGSI-10_Open_RESTful_API_Specification

The *Cepheus* GE supports the processing and management of data in real time. The GE exposes two modules, a NGSI server (Cepheus-Broker) and a Complex Event processor (Cepheus-CEP). The Cepheus-Broker acts as a repository collecting data provided by IoT devices and sensors. This information can be forwarded to other GEs, such as the *Orion Context Broker*, or can be used by the Cepheus-CEP module.

The Cepheus Broker exposes a RESTful API, which according to the given text documentation implements the NGSI9/10 specification. However, as the project's Github repository[16] states some features of the NGSI9/10 specification are not provided. The Cepheus-CEP module is based on the Esper[17] engine allowing the process and analysis of data real time, such as filtering, aggregating and merging real time data from different sources. The GE exposes a RESTful API, which allows the configuration of the CEP engine and the communication with other GEs. The description of Cepheus-CEP API is given as text, however, as seen in the project's Github repository an API Blueprint description is being implemented.

Applications/Services Ecosystem and Delivery

The Generic Enablers of the *Applications/Services Ecosystem and Delivery* chapter form an ecosystem that enable developers to publish their products, services and applications and manage their offerings in order to generate revenue. In addition, the chapter offers tools that support application mashup, which is mainly focused on the creation of visualization dashboards based on underlying services and data. Table 4.4 demonstrates the chapter's offered GEs and specifies the given interfaces for accessing their functionalities.

The *Biz Ecosystem RI* GE offers a framework through which developers can publish their products, services, applications and manage the entire life

[16] `https://github.com/Orange-OpenSource/fiware-cepheus`
[17] `http://www.espertech.com/esper/`

Generic Enabler	Offers	API Description
Business API Ecosystem (Biz Ecosystem RI)	Web UI & RESTful API	API Blueprint
Data Visualization (SpagoBI)	Web UI, RESTful API & SDK	API Blueprint
Application Mashup (Wirecloud)	Web UI & RESTful API	API Blueprint

Table 4.4: FIWARE GEs of the Applications/Services Ecosystem and Delivery chapter

cycle of their offerings in order to obtain some revenue. The Generic Enabler provides a Web interface, through which a user can interact with it, having a twofold role. On one hand, a user acts as a seller publishing and managing his offerings, and on the other hand a user acts as a customer buying products, services or applications. The functionalities of the *Biz Ecosystem RI* GE are also exposed by a RESTful API, described in API Blueprint.

The *SpagoBI* GE offers analytical capabilities for business intelligence and Big Data analytics. The GE provides tools ranging from traditional reporting and charting features to generating insights on data and turning them into actionable knowledge for effective decision-making processes. A user can interact with the GE using the provided Web interface, while a RESTful API is also offered, described in API Blueprint, allowing a user to manage the documents and datasets. In addition, *SpagoBI* GE provides a Javascript SDK that helps users to embed parts of the offering functionalities inside a web page or to retrieve information about datasets and documents.

The *Wirecloud* GE provides a web application mashup platform. Web application mashups integrate heterogeneous data, application logic, and UI components (widgets) to create new value-adding composite applications. A Web interface is offered so that users can choose the best suited widgets and create new composite applications such as dashboards for visualizing the data of interest. In addition, the *Wirecloud* GE provides a RESTful API, called Apllication Mashup API, and described in API Blueprint, which

allows users to manage their workspaces as well as the widgets available at the Application Mashup server.

Security

The purpose of the *Security* chapter is to provide a comprehensive set of services for applications to comply with major security requirements such as authentication and authorization. Table 4.5 summarizes the offered GEs, as well as the offered interfaces, that users can use in order to secure their services and applications based on FIWARE security layer. Note that, the *Security* chapter may also includes software components, such as the *PEP Proxy (Wilma)* GE, which are added in the backend of applications, in order to ensure authorization and authentication of FIWARE users.

Generic Enabler	Offers	API Description
Identity Management (KeyRock)	Web UI & RESTful API	API Blueprint
Authorization PDP (AuthZForce)	Web UI & RESTful API	WADL

Table 4.5: FIWARE GEs of the Security chapter

The *Keyrock* GE is one of the most important components in the FIWARE platform, as it provides user, organization and application identity management, and authentication. The Generic Enabler is based on Openstack Keystone[18] fully implementing its APIs as well as additional functionalities that only *Keyrock* offers. Using the offered Web interface developers can register their applications, and manage the security of their applications (credentials, authorization policy, and roles). In addition, users and client applications interact with *Keyrock* GE for authentication and even provide access to third-party applications using the OAuth2 flow. Users can also interact with the GE using the RESTful API that is also provided and described in API Blueprint.

[18] `https://developer.openstack.org/api-ref/identity/v3/index.html`

The *AuthZForce* GE allows the creation and management of authorization policies, which are responsible for authorizing or denying access requests of services. The Generic Enabler offers a RESTful API, described in WADL, that complies with XACML[19] (eXtensible Access Control Markup Language), an OASIS standard for authorization policy format. The functionalities of the *AuthZForce* GE are used by the *Keyrock* GE to manage policies defined by developers for their applications.

Interface to Networks and Devices (I2ND)

The Generic Enablers of the *Interface to Networks and Devices* chapter are mainly focused to offer services for the creation and management of Software-Defined Networks[20] (SDN) in private, enterprise and public settings. Moreover, the chapter may contain libraries and tools (middlewares) that are integrated within application, supporting a wide range of communication scenarios.

The *Network Information and Control (OFNIC)* GE provides the means for the management of network infrastructures. It exposes network status information and enables a certain level of programmability within the network. Users of the *OFNIC* GE may use the provided Web interface, or the RESTful API, described in API Blueprint, in order to access the networks and retrieve information and statistics as well as set control policies exploiting networks' capabilities.

Advanced Web-based User Interface

The main objective of the *Advanced Web-based User Interface* chapter is to provide Generic Enablers that will improve the user experience in Future Interner applications by adding new features such as interactive 3D graph-

[19] http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html
[20] https://en.wikipedia.org/wiki/Software-defined_networking

ics and immersive interaction with the real and virtual world (Augmented Reality[21]).

The *3D-UI-XML3D* GE adds new HTML elements for describing 3D scenes, including geometry, material, textures, lights, and cameras. The GE is the reference implementation of XML3D[22], a proposal for an extension to HTML5 to provide Web developers an easy way to create interactive 3D Web applications.

The *Synchronization* GE allows multiple instances of 3D and 2D environments (scenes) running on different clients to synchronize in real-time. Clients may interact with the GE either using the WebSocket protocol, or a RESTful API (Scene API) for infrequent queries and modifications. The *Cloud Rendering* GE is a service that connects to a synchronization server and is responsible for the rendering of a scene based on a chosen camera view and streams the results back to the client.

The *GIS Data Provider (Geoserver/3D)* GE enables the storage of geographical data which can also be represented in 3D form. The GE's implementation is based on the open source GeoServer[23] project, a Java server that allows users to share and edit geospatial data. A user may interact with the GE either through a Web interface or a RESTful API in order to access and manage its resources.

The *POI Data Provider* GE offers a repository for the management and retrieval of Point of Interest[24] (POI) data. Thus, the GE can be used for the positioning of content in the context of 3D or 2D scenes. A user is provided with a RESTful API, described in API Blueprint, that allows the management and retrieval of information related to locations.

The *Interface Designer* GE offers a Web editor for the creation and manip-

[21] https://en.wikipedia.org/wiki/Augmented_reality
[22] http://xml3d.org/
[23] http://geoserver.org/
[24] https://en.wikipedia.org/wiki/Point_of_interest

ulation of scene objects (entities). The GE doesn't fully support the creation of content, but depends on pre-existing content that is added to a scene, in order to be properly positioned and whose parameters and animations can be modified. On the other hand, the *Virtual Characters* GE allows the insertion and control of animated virtual characters in 3D scenes.

4.2 API Descriptions of FIWARE's GEs

The FIWARE catalogue offers developers with tools for the creation of innovative applications or the integration of new functionalities to existing applications. The majority of FIWARE's Generic Enablers offer a RESTful API, through which developers can access their functionalities and manage their resources. Therefore, it is important the GEs to offer an accurate and complete description of their APIs, so that developers may easily interact with them.

API Blueprint is the most commonly used service description specification among GE providers, but it cannot be considered very useful for GE consumers. Due to its nature, the specification is mainly focused providing a human readable text description of services, without imposing any particular restrictions and guidelines for the description of APIs. Thus, a service provider is free to describe the functionality of his service in any way he prefers. However, this freedom of expression comes with a cost of limited tool support, such as client generation.

In the case of FIWARE's GEs, the description of RESTful APIs is based on providing examples for every HTTP request and its corresponding response. In addition, these examples may be accompanied by a small text that briefly explain their components. However, these type of descriptions tend to be incomplete or even misleading, not allowing developers to properly communicate and interact with the Generic Enabler.

For example, consider the API Blueprint description of the *Bosun* GE, which allows the management of Cloud resources based on rules. Figure 4.4 illustrates a part of the API's description that explains the representation of a rule in JSON format. As seen, a rule comprises of three parts: name, condition and action. The name property specifies the rule's name, while the condition property describes the conditions that must be met (i.e. the CPU usage of a server is greater than a certain limit) so that specific actions (i.e. a server scale up) to be performed as described in the action property. The description of a rule representation continues by explaining the structure of condition and action objects.

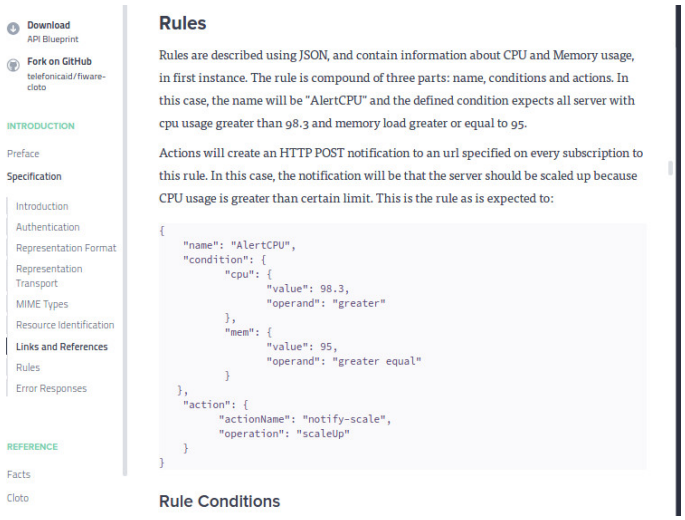

Figure 4.4: Representation of rules in Bosun GE

Nevertheless, as the service description presents API's endpoints regarding the creation and management of rules, a different and non-compatible representation of rule is provided. Figure 4.5 demonstrates a part of the API's description, which describes the HTTP request for the creation of a new rule. As seen, a JSON Schema is also provided for validating the JSON message that will be sent for the creation of a new rule. However, a closer look at the given JSON Schema reveals some inconsistencies regarding what was previously described at the representation of a rule. According to the JSON Schema the condition and action properties are expected to contain string

values, while the value of these properties should be other JSON objects.

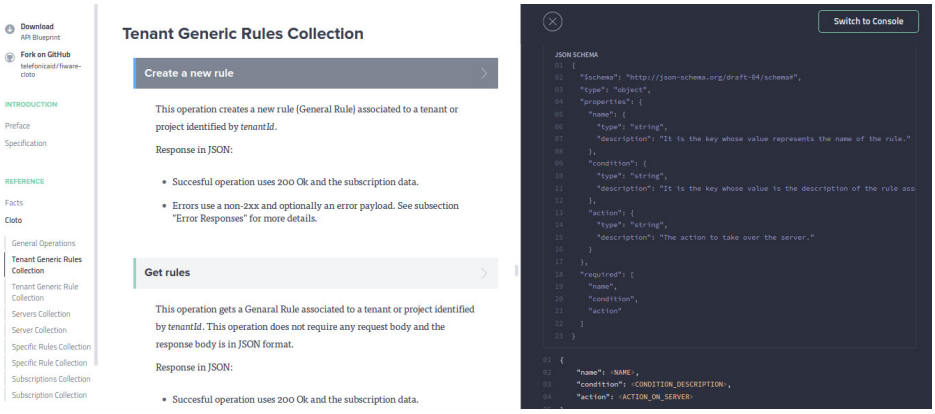

Figure 4.5: The provided JSON Schema of a rule in HTTP requests of Bosun GE

Another example of incomplete description is the RESTful API of *Keyrock* GE. As Figure 4.6 illustrates, the API is described by simply providing examples for every HTTP request and its corresponding response, without providing any description of the properties that are used in JSON structures. Therefore, developers with no previous experience with *Keyrock* GE may face many difficulties during their interaction with it. Such descriptions are found on the majority of Generic Enablers in the FIWARE catalogue, which in many cases may prevent potential developers from using them.

4.3 Improving the FIWARE Catalogue

The FIWARE platform needs to define guidelines and instructions in order to offer better service descriptions for its Generic Enablers, and consequently to improve user's experience and interaction with the platform. In this context, FIWARE should consider the adoption of the OpenAPI Specification for the description of RESTful APIs and take advantage of its tools and features, such as our proposed approach SOAS that semantically enriches the OpenAPI service descriptions. During our research, we attempted to generate

the OpenAPI service descriptions of FIWARE's GEs. However, these descriptions are not completed and cannot be used yet, since they are based on information collected from the existing API descriptions which as seen previously in many cases are incomplete and inadequate. Nevertheless, this is a work in progress since the FIWARE platform and its Generic Enablers are constantly evolving, and thus a cooperation with the FIWARE community is needed in order to achieve this integration. The generated OAS descriptions are stored in a public repository[25], so as to be easily retrieved and further improved.

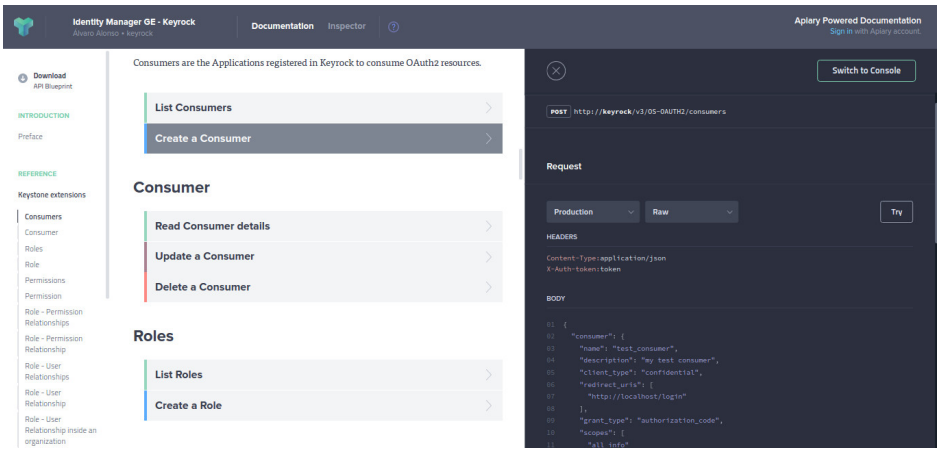

Figure 4.6: API Blueprint description of HTTP requests of *Keyrock* GE

Listings 4.1 and 4.2 provide an excerpt of *Bosun's* RESTful API described in OAS. Listing 4.1 demonstrates how a Rule (Figure 4.4) can be described in OAS. As seen, a Rule is represented as a *rule* Object containing 3 properties: the *name* property that specifies the rule's name, and the *condition* and *action* properties, which are described by other Objects. A Rule's condition is described by the *ruleCondition* Object whose properties contain conditions about CPU, Memory, Disk and Network usage to be evaluated. The *action* Object describes the available actions that will be performed. Based on the existing documentation there are two types of actions, *notify-email* and

[25] `https://www.dropbox.com/sh/gy6j6ia6ce9kdpl/AADWzXB7lq4-vovk0ZVFR9X_a?dl=0`

notify-scale. According to the value of the *actionName* property, a different action is specified. This is a typical example of Polymorphism that OAS supports through the use of the "discriminator" property.

Listing 4.1: Rule representation of Bosun GE in OAS

```
definitions:
  rule:
    type: object
    properties:
      name:
        type: string
      condition:
        $ref: '#/definitions/ruleCondition'
      action:
        $ref: '#/definitions/action'
    required:
      - name
      - condition
      - action

  ruleCondition:
    type: object
    properties:
      cpu:
        $ref: '#/definitions/condition'
      mem:
        $ref: '#/definitions/condition'
      hdd:
        $ref: '#/definitions/condition'
      net:
        $ref: '#/definitions/condition'

  condition:
    type: object
    properties:
      value:
        type: number
        format: float
      operand:
```

```
      type: string
      enum:
        - greater
        - greater equal
        - less
        - less equal
  required:
    - value
    - operand

action:
  type: object
  properties:
    actionName:
      type: string
  discriminator: actionName
  required:
    - actionName

notify-email:
  allOf:
    - $ref: '#/definitions/action'
    - type: object
      properties:
        email:
          type: string
        body:
          type: string
      required:
        - email
        - body

notify-scale:
  allOf:
    - $ref: '#/definitions/action'
    - type: object
      properties:
        operation:
          type: string
```

```
            enum:
              - scaleUp
              - scaleDown
        required:
          - operation
```

Listing 4.2 illustrates the description of a HTTP POST request and its corresponding responses from the API of *Bosun* GE (Figure 4.5) for the creation of a Rule in OAS. The listing describes an operation (named "createRule", as the operationId property states) on the path "/{tenant_id}/rules". The operation specifies a *path parameter* ("tenant_id") as well as a *body parameter* ("newRule") using the rule definition, as seen in Listing 4.1, in order to describe the body of the HTTP request. In addition, the operation also describes the responses that are sent from the server. A response with status code 200 informs that the new rule was successfully created and a JSON message is also received containing the ID of the new rule. The "default" response describes the message that is returned in case of an error. According to the existing documentation the returned error message is described similarly for every status code (except for status code 200).

Listing 4.2: Creating a Rule operation of Bosun GE in OAS

```
paths:
  /{tenant_id}/rules:
    post:
      operationId: createRule
      description: This operation creates a new rule (General
         Rule) associated to a tenant or project identified by
         tenantId.
      consumes:
        - application/json
      produces:
        - application/json
      parameters:
        - name: tenant_id
          in: path
          type: string
          description: Alphanumeric identifier of the Tenant to
```

```
          perform action with, following the OpenStack ID
          format e.g. d3fdddc6324c439780a6fd963a9fa148
        required: true
      - name: newRule
        in: body
        required: true
        schema:
          $ref: '#/definitions/rule'
    responses:
      200:
        description:  successful rule creation
        schema:
          $ref: '#/definitions/newRuleId'
      default:
        description: default response for errors
        schema:
          $ref: '#/definitions/errorRespose'
```

Developers can benefit from the OpenAPI service descriptions by generating clients in any programming language using Swagger Codegen[26]. The generated client library provides a connection with the service and implements all the functionality for successfully interacting with the service. In addition, the Swagger Codegen allows developers to configure custom templates in order to generate client libraries based on their preferences. Thus, developers can integrate the functionalities of a Generic Enabler into their working applications with minimal programming effort.

Apart from client generation, the existence of OpenAPI service descriptions allows FIWARE to develop and implement tools and services that may further improve users' experience with the platform such as service discovery mechanisms. As seen in Section 2.8.1, Oracle has already released an API catalogue for its Cloud services based on the OpenAPI Specification. However, service discovery is not yet efficient as it solely relies on searching for keywords that exist in API descriptions. Service discovery could be improved using the SOAS approach as presented in Chapter 3.

[26] https://github.com/swagger-api/swagger-codegen

Through the proposed extension properties various parts of an OpenAPI service description can be semantically enriched and obtain a semantic content that machines may understand and interpret similarly to humans. The SOAS approach may eliminate any ambiguities and provide descriptions which are both uniquely defined and discoverable. Regarding FIWARE, the semantic enrichment of OpenAPI service descriptions can be performed using any existing vocabulary, such as Schema.org, that best describes the domain of a service. Nevertheless, in cases where an appropriate vocabulary doesn't exist, FIWARE should consider defining a new one.

The semantically annotated OpenAPI service descriptions allows the implementation of a semantic API catalogue based on the proposed OpenAPI ontology, as presented in Section 3.4. The OpenAPI ontology preserves all the syntactic and semantic information of an OpenAPI service description and thus allowing the utilization of Semantic Web tools, such as reasoning and querying. In fact, the semantic API catalogue is a triplestore[27] (or RDF store), containing all the OpenAPI service descriptions transformed in the OpenAPI ontology. Reasoners, such as Pellet, can be used in order to check the consistency of ontologies and also infer additional relations that may exist. An important feature of the semantic API catalogue is the querying support that is offered through SPARQL, the standard query language for RDF data.

Using SPARQL, users can perform various queries in order to find the services that meet their needs. For example, consider a user searching for a service to create a virtual machine in the FIWARE platform. In this case, a SPARQL query should require all services of the API catalogue using the virtual machine entity, as well as any operations that are responsible for creating a virtual machine. This discovery method should be more accurate and efficient, compared to the text search that Oracle's API catalogue offers. The implementation of an semantic API catalogue and its discovery mechanism is a challenging project, which we are willing to undertake as future work.

[27] https://en.wikipedia.org/wiki/Triplestore

5

Conclusions and Future Work

The advent of Cloud Computing and its rapid development led organizations and individuals to redesign their strategy and products. However, as the number of Cloud services is constantly increasing, the need for efficient and accurate service discovery has become a significant challenge. This is mainly due to the lack of formal service descriptions, as the majority of Cloud providers describe their offerings in plain text. Therefore, the purpose of our work was to propose a description language for Cloud services, that both humans and machines could understand, and thus allow the implementation of various tools, such as service discovery mechanisms.

5.1 Conclusions

During our research we reviewed many approaches that would efficiently describe any aspect of a service, both syntactically and semantically. However, we were mainly focused on approaches for describing RESTful services, as the majority of Cloud services are offered by means of Web services based on the REST architecture style. Unlike SOAP-based services, using the standard WSDL, there are many approaches for the description of RESTful services. This is quite normal, as REST is not a specific framework, rather a set of principles and guidelines by which Web services are designed to focus on a system's resources. Therefore, there are diverse techniques for implementing RESTful services, which don't facilitate the existence of a standard descrip-

tion language.

For the description of Cloud services, we ended up proposing and using the OpenAPI Specification. The selection of the OAS, was motivated by the popularity of the specification, its powerful tool support, and the active community. In addition, there was a series of events that also affected our decision. At the end of 2015, the OpenAPI Initiative was announced, founded by organizations such as Google, Microsoft and IBM, in order to extend the OpenAPI Specification (formerly known as Swagger) and standardize a description mechanism for RESTful services. Openstack, in mid 2016, has announced the adoption of the OAS for the description of its offering services. Oracle, was the first organization that released an API catalog of its offering Cloud services described in OAS, while Microsoft preserves a Github repository[1], where the OpenAPI descriptions of Azure's Cloud services can be found.

As we demonstrated, the OpenAPI Specification offers a human-friendly environment for discovering and consuming RESTful services. However, despite being machine-readable the specification is not machine understandable, thus limiting the availability of tools that facilitate machine tasks such as service discovery. Our proposed solution, the Semantic OpenAPI Specification, attempted to fill this gap, allowing the description of RESTful services both semantically and syntactically. Using the extension mechanism that the OAS offers, we defined some additional properties that semantically enrich various parts of an OpenAPI service description, resolving any ambiguities that may exist in service descriptions and allowing machines to better understand the described services. In addition, we developed an ontology allowing any OpenAPI service description to be transformed in, enabling the use of Semantic Web tools such as reasoners and query languages.

The adoption of the whole OAS ecosystem in conjunction with our proposal can be substantially beneficial for both Cloud providers and users as it offers many opportunities. Nevertheless, this is not a straightforward process,

[1] `https://github.com/Azure/azure-rest-api-specs`

as significant work is required. In the case of the FIWARE platform, we realized that the offering services are insufficiently described, complicating the generation of OpenAPI service descriptions. Thus, the FIWARE community is needed to resolve these deficiencies. In addition, the semantic annotation of OpenAPI service descriptions requires for ontologies that best describes services' domains. The FIWARE community should provide such ontologies, either using existing ones or defining new.

5.2 Future Work

Regarding future work, there are many issues worth considering further. Currently, the OpenAPI Specification is evolving, and a new version[2] of the specification is expected to be released. The new version will bring structural improvements as well as new features. For example, a mechanism for describing links contained in HTTP response messages is introduced, while additional features from JSON-Schema will be adopted, such as the "oneOf", "anyOf", and "not" properties. Therefore, a new analysis must be performed, in order to determine whether additional properties are required for the semantic annotation of new features. The OpenAPI ontology will also be affected, since all changes and the new features must also be defined.

Regarding the OpenAPI ontology, we should consider the alignment with emerging technologies, such as the Shapes Constraint Language (SHACL)[28]. SHACL, is a specification produced by the W3C RDF Data Shapes Working Group[3], offering an RDF vocabulary that can be used to describe the structure of data, similarly to XML-Schema or JSON-Schema. Therefore, we need to examine if it can be used for the representation of *Schema Objects* of an OpenAPI service description in the OpenAPI ontology.

The implementation of a semantic API catalogue is also a project that we

[2] https://www.openapis.org/blog/2017/03/01/openapi-spec-3-implementers-draft-released

[3] https://www.w3.org/2014/data-shapes/wiki/Main_Page

are willing to undertake. The system should transform the semantically annotated OpenAPI service descriptions into our proposed OpenAPI ontology and store them, allowing users to query them in order to discover the services that best meet their needs. The existence of a semantic API catalogue would also give us the opportunity to further extend its usage in more complicated tasks, such as service orchestration.

Bibliography

[1] Mike Amundsen. *Building Hypermedia APIs with HTML5 and Node.* " O'Reilly Media, Inc.", 2011.

[2] Apiary. Api blueprint. Technical report, Technical report, https://github.com/apiaryio/api-blueprint.

[3] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml)(tm) version 1.2. yaml. Technical report, org, Tech. Rep, 2009.

[4] Tim Berners-Lee. Linked data-design issues (2006). *URL http://www. w3. org/DesignIssues/LinkedData. html*, 2006.

[5] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[6] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pages 205–227, 2009.

[7] Tim Bray, Jean Paoli, CM Sperberg-McQueen, Eve Maler, Franois Yergeau, and John Cowan. Extensible markup language (xml) 1.1-w3c recommendation. *World Wide Web Consortium. http://www. w3. org/TR/xml11/*, 2006.

[8] Dan Brickley and R Guha. Rdf schema 1.1. w3c recommendation (25 february 2014). *World Wide Web Consortium*, 2014.

[9] Roberto Chinnici, J Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language, w3c recommendation, june 2007, 2007.

[10] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1, w3c note, 2001, 2001.

[11] Luc Clement, Andrew Hately, Claus von Riegen, Tony Rogers, et al. Uddi version 3.0. 2, uddi spec technical committee draft. *OASIS UDDI Spec TC*, 2004.

[12] World Wide Web Consortium et al. W3c: Simple object access protocol, soap, version 1.2 part 0: Primer,(2003). *Web site: http://www.w3.org/TR/soap12-part0*.

[13] Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. *W3C Recommendation*, 25:1–8, 2014.

[14] Hai Dong, Farookh Khadeer Hussain, and Elizabeth Chang. Semantic web service matchmakers: state of the art and challenges. *Concurrency and Computation: Practice and Experience*, 25(7):961–988, 2013.

[15] Thomas Erl. *Soa: principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008.

[16] Joel Farrell and Holger Lausen. Semantic annotations for wsdl and xml schema. w3c recommendation, 28 august 2007. *World Wide Web Consortium (W3C), Tech. Rep*, 2007.

[17] D Fensel, F Fischer, J Kopeckỳ, R Krummenacher, D Lambert, and T Vitvar. Wsmo-lite: Lightweight semantic descriptions for services on the web, w3c member submission. 2010.

[18] Roy Fielding. Fielding dissertation: Chapter 5: Representational state transfer (rest). *Recuperado el*, 8, 2000.

[19] John Franks, Phillip Hallam-Baker, Jeffrey Hostetler, Scott Lawrence, Paul Leach, Ari Luotonen, and Lawrence Stewart. Http authentication: Basic and digest access authentication. Technical report, 1999.

[20] Shudi Gao, CM Sperberg-McQueen, and Henry S Thompson. W3c xml schema definition language (xsd) 1.1 part 1: structures: W3c recommendation 5 april 2012. *Available at h ttp://www. w3. o rg/TR/xmlschema11-1*, 2012.

[21] OWL Working Group et al. W.: Owl 2 web ontology language: Document overview. w3c recommendation (27 october 2009), 2012.

[22] John Gruber. Daring fireball: Markdown syntax documentation, 2004.

[23] Hugo Haas and Allen Brown. Web services glossary. *W3C Working Group Note (11 February 2004)*, 9, 2004.

[24] Marc J Hadley. Web application description language (wadl). w3c member submission. *World Wide Web Consortium, W3C (November 2006)*, 2009.

[25] Dick Hardt. The oauth 2.0 authorization framework. 2012.

[26] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. Sparql 1.1 query language. *W3C Recommendation*, 21, 2013.

[27] Matthias Klusch. Semantic web service description. In *CASCOM: intelligent service coordination in the semantic web*, pages 31–57. Springer, 2008.

[28] Holger Knublauch and Arthur Ryman. Shapes constraint language (shacl). *W3C First Public Working Draft*, 8:W3C, 2015.

[29] Markus Lanthaler. Creating 3rd generation web apis with hydra. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 35–38. ACM, 2013.

[30] Markus Lanthaler and Christian Gütl. Hydra: A vocabulary for hypermedia-driven web apis. *LDOW*, 996, 2013.

[31] Holger Lausen, Axel Polleres, Dumitru Roman, et al. Web service modeling ontology (wsmo). *W3C Member Submission*, 3, 2005.

[32] D Martin, M Burstein, J Hobbs, O Lassila, D McDermott, S McIlraith, S Narayanan, P Paolucci, B Parsia, T Payne, et al. Owl-s: Semantic markup for web services. w3c submission (2004), 2004.

[33] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[34] Fathoni A Musyaffa, Lavdim Halilaj, Ronald Siebes, Fabrizio Orlandi, and Sören Auer. Minimally invasive semantification of light weight service descriptions. In *Web Services (ICWS), 2016 IEEE International Conference on*, pages 672–677. IEEE, 2016.

[35] Open API Initiative (OAI). Open api specification. Technical report, Technical report, https://github.com/OAI/OpenAPI-Specification.

[36] Leonard Richardson and Sam Ruby. *RESTful web services.* " O'Reilly Media, Inc.", 2008.

[37] Dumitru Roman, Jacek Kopeckỳ, Tomas Vitvar, John Domingue, and Dieter Fensel. Wsmo-lite and hrests: Lightweight semantic annotations for web services and restful apis. *Web Semantics: Science, Services and Agents on the World Wide Web*, 31:39–58, 2015.

[38] Manu Sporny, Gregg Kellogg, Markus Lanthaler, W3C RDF Working Group, et al. Json-ld 1.0: a json-based serialization for linked data. *W3C Recommendation*, 16, 2014.

[39] Le Sun, Hai Dong, Farookh Khadeer Hussain, Omar Khadeer Hussain, and Elizabeth Chang. Cloud service selection: State-of-the-art and future research directions. *Journal of Network and Computer Applications*, 45:134–150, 2014.

[40] Ben Walker. Every day big data statistics–2.5 quintillion bytes of data created daily. *VCloudNews. April*, 5, 2015.

[41] RAML Workgroup. Restful api modeling language (raml). Technical report, Technical report, https://github.com/raml-org/raml-spec.