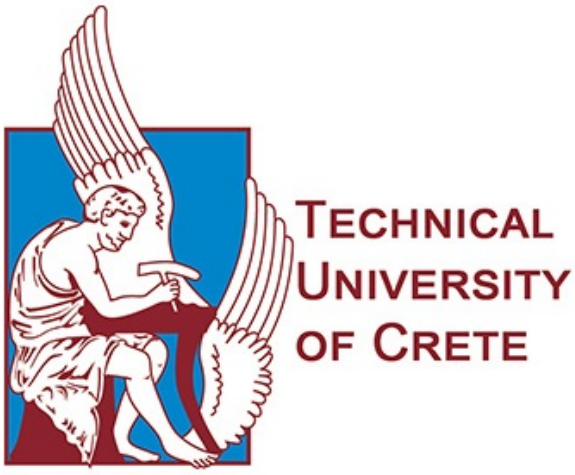

DEPARTMENT OF ELECTRICAL AND
COMPUTER ENGINEERING

Diploma Thesis

An ontology for describing OpenAPI version 3 services in the cloud

Aikaterini Karavasileiou

Comitee
Supervisor : Prof. Euripides G.M. Petrakis
Assoc. Prof. Deligiannakis Antonios
Assoc. Prof. Samoladas Vasileios

Chania, 2019

Abstract

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 adopt the OpenAPI Specification (OAS), a simple and yet powerful specification for the description of REST APIs, as the description language of Cloud services. OAS descriptions are mainly understandable by humans. However, OAS descriptions must be also understandable by machines so that, the services can be searched, discovered and used by other services. In order for a machine to understand the meaning of OAS, service descriptions need to be formally defined and their content be semantically enriched in a way that eliminates ambiguities. Taking advantage of the extension features foreseen in OAS 3.0, our approach suggests that in order to eliminate ambiguities in OAS descriptions, OAS properties must be semantically annotated. Building-upon the latest version of OAS, this work analyses the reasons that cause ambiguities in service descriptions and proposes Semantic OAS (SOAS 3.0). Building-upon SOAS descriptions, we designed and implemented a mechanism to transform SOAS (and therefore OAS) descriptions to ontologies. As a result, the ontology will enable application of querying languages (e.g. SPARQL) for service discovery and of reasoning tools for detecting inconsistencies and inferred relationships in SOAS descriptions.

Acknowledgements

I would really like to express my sincere appreciation to my Supervisor, Professor Euripides G.M. Petrakis for the help and support from the beginning till the end of this thesis.

Moreover, I am grateful to Nikos Mainas for his great suggestions and thoughtful discussions we had together.

I would also like to thank Professor Antonios Deligiannakis and Professor Vasileios Samoladas who agreed to participate in the presentation and evaluation of my thesis.

Last but not least, I would like to thank my family for their enormous help and unconditional support.

Contents

Chapter 1

Introduction

1.1 Motivation

We live in a world where the World Wide Web (WWW) is everywhere. The Web is realized as a composition of Web services. A web service is a unit of managed code that can be remotely invoked using HTTP. That is, it can be activated using HTTP requests. Web services enable the exposure of the functionality of an existing code over the network. Once it is exposed on the network, other applications can invoke it over HTTP.

With Web Services, each application must adhere to a set of standardized protocols for sharing and accessing data. This way, two programs can talk to each other, regardless of operating system, database or programming language. Instead, everyone agrees on a set of rules by which these interactions will take place.

For applications to communicate with each other their services must be formally described in a way that is understandable by machines. The last requirement would not only improve the accuracy of service descriptions but also, would allow services to be discovered by other services. A web service description is a document by which the service provider communicates the specification of the web service

to the service requester.

Web Service descriptions are available in plain text, which users have to browse and read in order to determine whether a service meets their needs. In the last years, more and more services tend to be written using OpenAPI Specification (OAS). However, OAS service descriptions are mainly intended to be readable by humans and not by machines, while in some cases are inaccurate or vague.

1.2 Problem Definition

OpenAPI Specification (OAS) is a description format for REST APIs. OAS descriptions are mainly understandable by humans. However, OAS descriptions must be also understandable by machines so that, the services can be searched, discovered and used by other services. In order for a machine to understand the meaning of OAS, service descriptions need to be formally defined and its content be semantically enriched in way that eliminates ambiguities. The focus of this work is on improving the description of Web Services in order to provide descriptions which are both uniquely defined and discoverable.

1.3 Problem Solution

OpenAPI Specification (OAS) 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 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. In this work we propose that OAS service description can be semantically annotated using extension properties. As a result we propose an extension of OAS referred to in the following as Semantic OpenAPI Specification (SOAS 3.0). Taking a step forward we then create a mechanism that achieves the association of OAS entities to entities of an Ontology (e.g. domain ontology). The new approach 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 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 following summarizes the contributions of this work :

- Building-upon the work by N.Mainas [1] we propose SOAS as an extension to the OpenAPI Specification (OAS) that semantically enriches service descriptions in order to eliminate ambiguities and offer descriptions readable by both humans and machines.
- Creates 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.

[1] https://dias.library.tuc.gr/view/68268

- Demonstrates how SOAS can be applied using two examples of web services described in chapter 6 of this thesis and also presents the results and benefits that are derived by using our mechanism for the transformation of a SOAS (and thus OAS) service description to an Ontology.

1.5 Thesis Structure

In chapter 2 we present the background and we briefly describe technologies that were used in this thesis. Chapter 3 provides information about our decision to choose OpenAPI Specification, its features and our proposed solution for the description of web services, the Semantic OpenAPI Specification 3.0. In Chapter 4 we present in an abstract level the algorithm that was created during this thesis and transforms SOAS services to Ontology. Chapter 5 presents the full algorithm behind the procedure of instantiating Services to Ontology. Chapter 6 contains the results that were derived by applying our mechanism to two Web services examples. Finally, chapter 7 presents our conclusions and our plans for future work.

Chapter 2

Background

2.1 SOAP

SOAP is an XML-based service invocation protocol and was originally developed for distributed applications that communicate over HTTP. It was meant to access services, objects and servers in a platform-independent manner.

This protocol was created as a response to the fact that HTTP was mainly used just for communication from a client to a server by passing files. However, intercommunication is very important and one-way communication was not enough in order to create competitive web services. Hence, an extension of the HTTP was needed. SOAP indeed added a set of HTTP headers and an XML payload and as a result it enabled complex two-way communication between applications.

The communication between a server and a client is implemented through an "envelope" which contains a mandatory "body" part that contains all the call and response info and a non obligatory "header" part that provides all the header information. Inside the body it is possible to exist a "fault" part that contains all the information about errors that could have occurred while processing the mes-

sage.

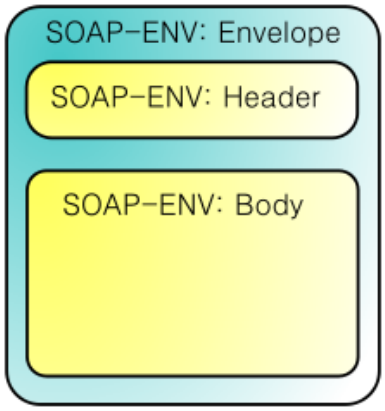

Figure 2.1: SOAP envelope

2.2 WSDL and UDDI

WSDL stands for Web Services Description Language, it was written in XML and was created in order to describe SOAP web services.

An WSDL document specifies the location and the methods of the service. It comprised mainly of the elements <types>, <message>, <portType>and <binding>. The first one specifies the datatypes that a web service uses. The <message>part contains the messages that are required in order to interact with a service operation, while the <portType>element 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. Last but not least the <binding>element provides the protocol and data format for each <portType>.

UDDI was also introduced as a registry for storing information about web services. UDDI stands for Universal Description Discovery and Integration. The services were

described via WSDL, while the communication between a service provider and a consumer used to happen via SOAP. In fact, a service provider had to describe its service using WSDL before it was published in a service registry, like UDDI.Then the service consumer had to issue a query to the registry to locate a service. The WSDL description of the service was passed to the service consumer, informing him how to communicate with the service. The service consumer used WSDL to send a request to the service provider and received the expected response by the service provider.

2.3 REST

REST (REpresentational State Transfer) is an architectural style for developing Web services. It was introduced in Ray Fielding's dissertation on 2000[1] and by then it obtained massive adoption. REST defines a set of constraints that need to be used while creating web services in order for the services to be called "RESTful".

The following key-terms are associated with REST. The client is the person or the software that uses the API. A **client** can be a developer that uses Twitter API, as well as a Web browser that uses Twitter API and then renders the returned data as information on the screen. The second key-term is a **resource** which is actually any object that the API can provide information about (e.g. an article or a photograph on a Website). Each resource has a unique identifier, known as URL. Resources can be static like a chapter on a book or dynamic like the news.

During the communication between a client and a

[1]https://www.ics.uci.edu/ fielding/pubs/dissertation/fielding_dissertation.pdf

server it is obligatory for the first to provide the latter with an identifier for the resource it is interested in. Moreover the client has to inform the service provider about the operation that the server needs to perform on that specific resource. REST was originally designed based on the HTTP protocol. The operations that a client may ask a server to perform are usually in the form of an HTTP method (e.g. GET,PUT,POST, DELETE).

The key-point of REST are the constraints that a service should follow in order to become "RESTful". The first one is the *Client Server* constraint. This is responsible for a separation between the client and the server. The separation allows the components to evolve independently. In fact only the client is able to begin an interaction with the server. The server is responsible for responding to the request. Next, we have the *Stateless* constraint. This requires the client to provide all the information that the server needs in order to understand the request. The third is the *Cache* constraint that requires that the data sent from the client to the server are labeled explicitly as "cacheable" or "non-cacheable". If we have the label "cacheable", then it's permitted to the client to store the response in order to reuse it later in an equivalent request. This fact obviously increases the system performance as it reduces the requests that the server has to manage. Following Cache constraint we have the *Layered System*. Indeed, an architecture can be composed of hierarchical layers in the form of servers between the service consumer and the service provider. Each component cannot "see" beyond the immediate layer with which they are interacting and obviously the layers do not affect neither the request nor the response. The *Code-On-Demand* is optional and en-

ables the client to download and execute the code from the server.

The final constraint is *Uniform Interface*. This is the central feature that distinguishes REST from other network-based styles. In REST architecture there are four interface constraints. The *identification of resources*, meaning that every request about a resource has to include a URI. Next there is the *manipulation of resources through representations*, which means that a client can modify the resource given that the server has given the permission. In addition the *self-descriptive messages* impose that every message to and from the server must contain all the information needed in order to be efficiently processed. The final interface constraint is *Hypermedia*. Although it is usually violated HATEOAS is an important constraint and it means that the server should allow the client to discover all the available actions and resources it needs by using *hyperlinks*.

2.4 Differences between SOAP and REST

SOAP is a protocol, while REST is an architectural style. SOAP is characterized by strict rules and it also provides good security features. Specifically, SOAP supports WS-Security for enterprise-level protection.[2] For instance, when an application deals with crucial private information like bank account numbers, it makes more sense to use SOAP. However, SOAP's extra security is not necessary in an application that sends the day's forecast. In addition, SOAP requires more resources and bandwidth, while the response data cannot be stored in a cache. Last but not least, SOAP

[2]https://www.soapui.org/soapui-projects/ws-security.html

is limited to the use of XML and requires additional parsing for messages.

REST was actually created to cope with the problems that were derived from SOAP, such as bigger complexity and inflexible architecture. REST allows different messaging formats, such as JSON, HTML, XML or even plain text and it's also characterized by better performance. Moreover, it provides caching and scale-ability. In order to achieve the above features REST architectural style lacks in security. As a result even though both SOAP and REST can be used in any application, SOAP is *usually* preferred in enterprise applications, while REST is rather handled by the web where high-flexibility is a more important feature.

2.5 WADL

As soon as REST appeared, WADL was proposed as an XML - based description language for HTTP - based web services. WADL provides a machine readable XML description in order to model the resources provided by a service as well as the relationships between them. In fact, WADL is the REST equivalent of SOAP's WSDL and thus it was created in order to describe RESTful services.

However, WADL is not popular. The main reason is that WADL, similarly to WSDL, provides only a syntactic description of the service, with limited support for describing the meaning of service's resources. In addition, WADL does not support semantic annotation hence it is not possible to enrich the meaning of a service description.

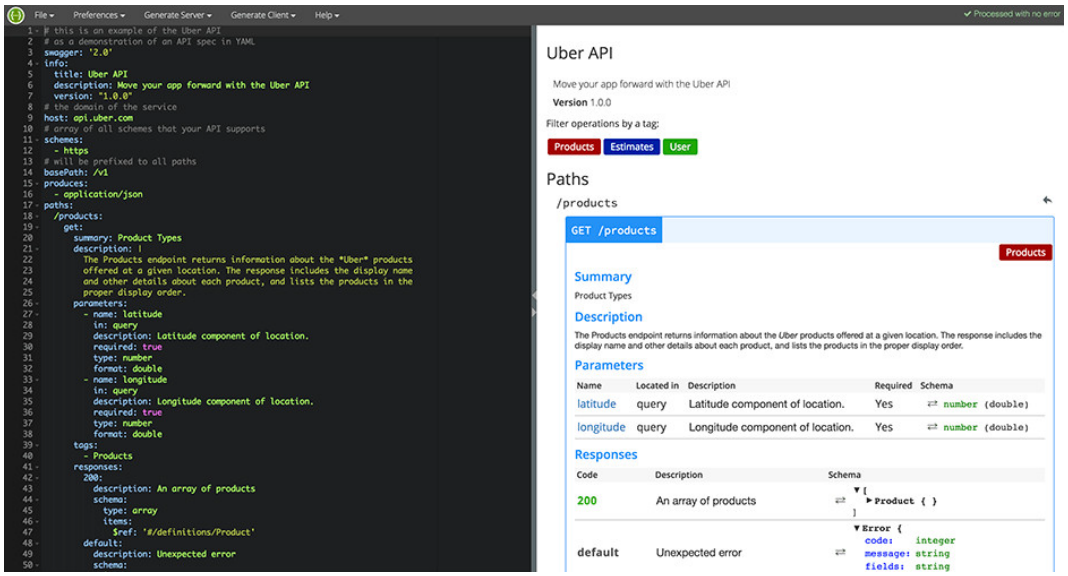

Figure 2.2: Swagger Editor example

2.6 OpenAPI Specification

As WADL was not adopted massively, another specification was created and became very fast the most common approach for the description of RESTful services. OpenAPI Specification (OAS)[3] was initially called Swagger Specification and it is an open - source, language - agnostic specification. When properly defined, the consumer is able to understand and interact with the service with minimal amount of implementation logic.

The descriptions of the services can be written in either JSON[4] or YAML[5]. The implementation of the APIs may follow a top - down or a bottom - up approach. When the top - down approach is used, the service is implemented after the service's description has been written, while in the bottom - up approach the implementation comes first and the description is later generated by the service's implementation.

[3]https://swagger.io/specification/
[4]https://www.json.org/
[5]https://yaml.org/

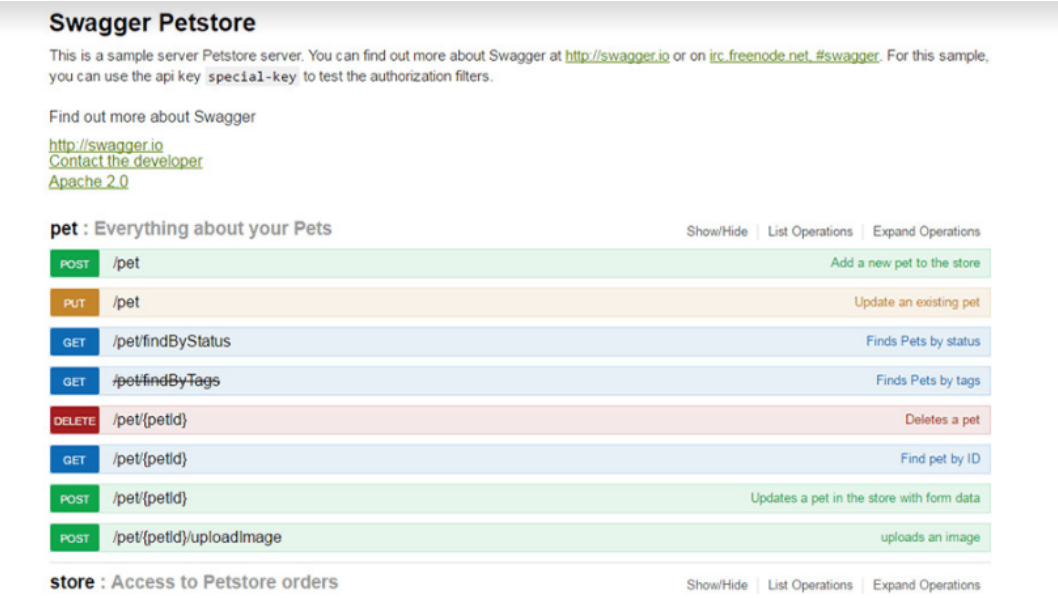

Figure 2.3: Swagger UI

More specifically, OAS provides an editor called Swagger Editor[6], an example of which appears in figure 2.2. Swagger Editor runs locally or online. It also provides instant visualization which means that the client may interact with the specification while still defining it (e.g. the client may see the changes in the code appear instantly on the interface). Thus, a consumer may use the Swagger Editor in order to fully document RESTful services. Moreover, OAS provides an open - source code generator - Swagger Codegen[7]- which makes it possible to build server code directly from an OpenAPI service description in almost any programming language and framework (PHP, Java etc). Last but not least, OpenAPI Specification gives the client the chance to visually render documentation for an OpenAPI service description, using the Swagger UI[8], an open - source HTML5 - based user interface. An example of a Swagger UI appears in figure 2.3.

[6]https://editor.swagger.io/
[7]https://swagger.io/tools/swagger-codegen/
[8]https://swagger.io/tools/swagger-ui/

It is important to mention that the OpenAPI Specification is part of the OpenAPI Initiative (OAI)[9], which is supported by widely known companies such as Google, Microsoft and IBM. The OAS structure is going to be further described in the following chapters.

2.7 OAS v2 - OAS v3

The current version of OpenAPI Specification is OAS 3.0 and was released in 2017. It was the first major update of the specification since 2015. OAS 3.0 features a more elaborate (yet simple) structure and format than its predecessor OAS 2.0. The requirement for a single host server is relaxed (allowing a service to be installed on multiple servers). The request body is more flexible and allows consumption of different media types, such as JSON, XML, HTML, plain text and others. The descriptions for parameters have changed: FormData parameter was removed and, the cookie parameter type was introduced for documenting APIs that use cookies. The definition of Schema objects is enhanced with additional properties (e.g. anyOf, oneOf, not) allowing a creation of more complex schemas of various data types. Regarding security definitions, OAS v3.0 is enhanced with support for OpenID Connect Discovery[10]. OAS 3.0 now features a Components field where various reusable objects can be defined (i.e. responses, parameters, headers, links, callbacks, schemas and security schemes).

In the new update two new features were added - referred to as LINKS and CALLBACKS. LINKS are defined

[9] https://www.openapis.org/

[10] https://swagger.io/docs/specification/authentication/openid-connect-discovery/

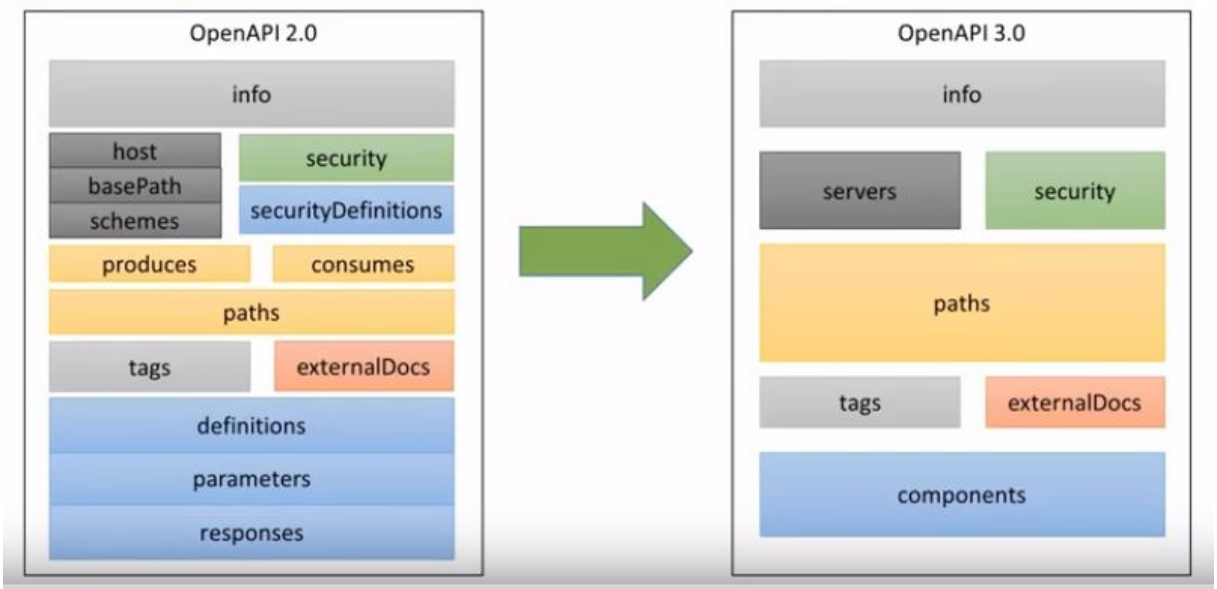

Figure 2.4: Differences between the two versions of OAS

in the service response section to allow values returned by a service call to be used as input for a next call. This is an attempt of OAS 3.0 to incorporate HATEOAS functionality in the specification. Finally, CALLBACKS is a feature for defining asynchronous APIs or Webhooks. CALLBACKS define the requests that the described service will send to another service in response to certain events. An application of this feature would be for describing publish-subscribe mechanisms which allow services to publish information and other services subscribing to them to get notified when this information becomes available.

The differences between the two versions of OAS appear in figure 2.4[11]. As we may see, the structure in OAS 3.0 has become more simplified. As identified by the color in the figure *definitions, parameters, responses* and *securityDefinitions* appear now under the *components* item. New objects have been added within this item. Item *definitions* in OAS 2.0 has been renamed as *schemas* in OAS 3.0, while *securityDefinitions* item has been re-

[11]https://swagger.io/blog/news/whats-new-in-openapi-3-0/

named as *securitySchemes*, placed under *components* item. The items *produces* and *consumes* have disappeared and absorbed by the *paths* item. The same happened with the sub-items *host, basePath, schemes* that have been replaced by the *servers* item.

2.8 Semantic Web

The Semantic Web is an extension of the World Wide Web through standards by the World Wide Web Consortium (W3C). It promotes common data formats and exchange protocols on the Web, most importantly Resource Description Framework (RDF). Moreover, Semantic Web provides software programs with metadata that make the process of finding Web pages a lot more accurate. The main goal is to allow data to be "machine readable" and "machine understandable".

In addition, there are technologies in the context of Semantic Web that enable people to create data stores on the Web, build vocabularies, and write rules for handling data. In fact Semantic Web is about linked data which are empowered by technologies such as RDF, SPARQL, OWL. This is very useful as the above technologies enable the use of reasoners, the process of writing queries for the data etc.

In a few words, semantic web is the idea of linking data in the whole World Wide Web and making the integration of connected information found in different web sites possible.

2.9 Ontologies and OWL

An ontology is a formal description of knowledge as a set of concepts within a domain and the relationships that hold between them. In ontologies we meet terms such as individuals (instances of objects), classes, attributes and relations as well as restrictions, rules and axioms.

As already mentioned, the main use of ontologies is to represent knowledge. However, there are other ways of doing that - some of which are vocabularies and logical models. The advantage of ontologies though is that they make the process of expressing relationships and linking data to specific concepts very easy and precise. In a Semantic World, the ontologies are one of the component - keys by providing the necessary structure in order for information to be connected with other similar to it on the Web of Linked Data.

In order for ontologies to be expressed the W3C Web Ontology Language (OWL) has been created. OWL is a Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and the relations between them. In fact, OWL gives the user the opportunity to create classes, property of classes and the relationships that exist between them.

However, probably the most valuable feature that OWL provides us with is the ability of using a reasoner on the created ontology. The use of the reasoner comes with all the advantages that the latter has, which means consistency checking (find any logical inconsistencies) and satisfiability checking (whether a class can have instances or not). By the use of reasoners, the user may also discover "hidden" relationships, such as the follow :

$$A \rightarrow B$$
$$B \rightarrow C$$
So, it holds that $A \rightarrow C$.

In order to conclude, ontologies offer a lot of benefits. Except reasoning, which has already been analyzed, we could say that ontologies "understand" concepts and relationships in ways that are close to the way humans perceive interlinked concepts. In addition, they provide a more coherent and easy navigation as users move from one concept to another in the ontology structure. Last but not least, ontologies are easy to extend as relationships may be added without much implementation effort to existing ontologies. As a result, this model evolves with the growth of data without impacting dependent processes and systems if something goes wrong or needs to be changed.

2.10 Hydra Core Vocabulary

Hydra[12] is a lightweight vocabulary for creating hypermedia-driven Web APIs. More specifically, Hydra defines a number of concepts in RDF Schema that allow machines to understand how to interact with an API. The main idea is to provide a vocabulary through which the 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. Since all the information about the valid state transitions is exchanged in a machine-processable way at runtime instead of being hardcoded into the client at design time, clients can be decoupled from the server

[12]https://www.hydra-cg.com/spec/latest/core/

and adapt to changes more easily.

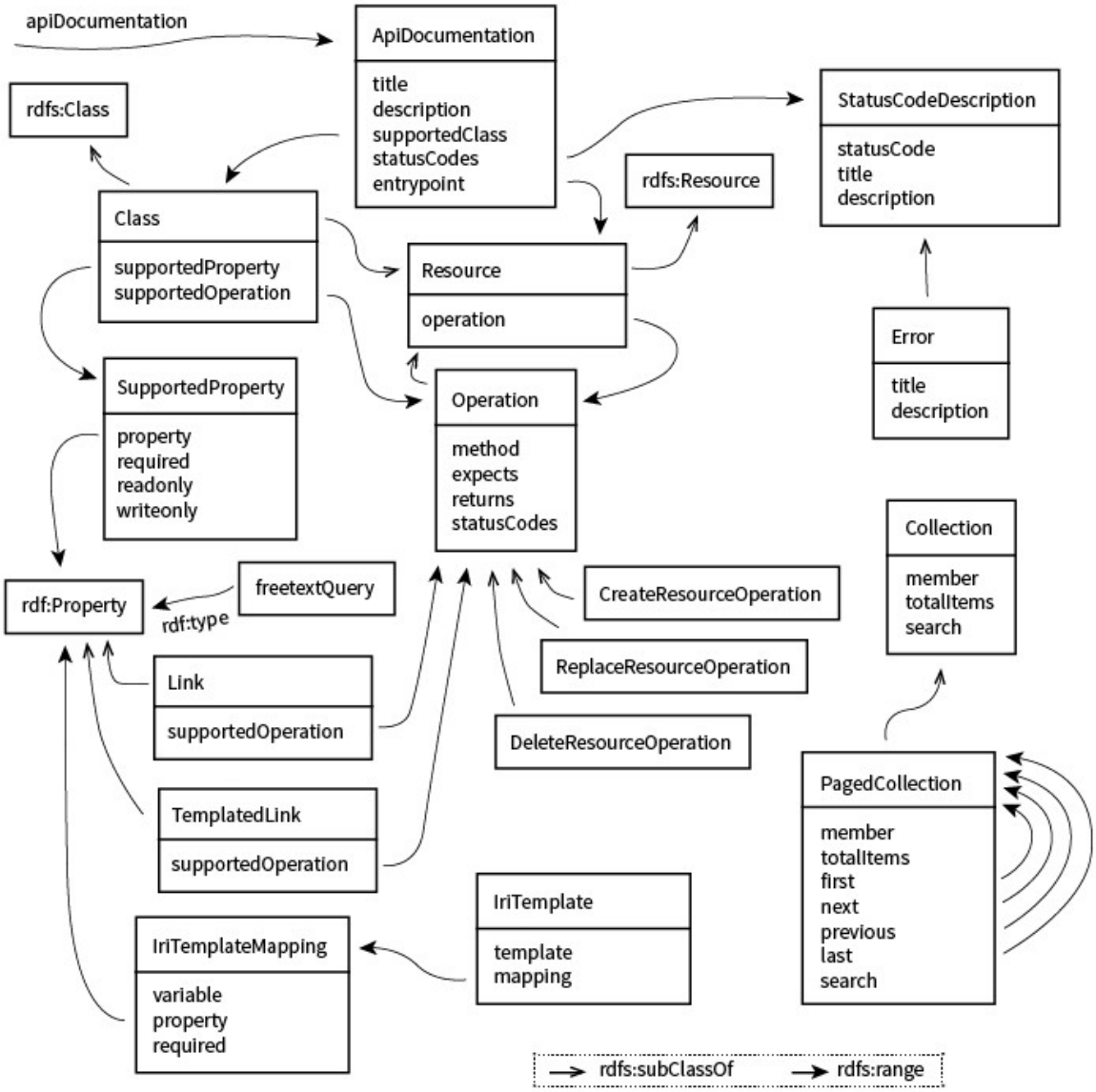

Figure 2.5: Hydra Core Vocabulary

As we can see in figure 2.5, the centre of the vocabulary is actually the *ApiDocumentation* class, which builds the foundation for the description of a Web API. Hydra describes an API by giving it a title, a short description, and documenting its main entry point. Furthermore, the classes known to be supported by the Web API and additional information about status codes that might be returned can be documented.

Generally, a client decides whether to follow a link or

not based on the link relation (or property in the case of Linked Data) which defines its semantics. There are however also clients such as Web crawlers which simply follow every link intended to be dereferenced. In HTML this usually means that all links in anchor elements (the <a>tag) are followed but most references in link elements (the <link>tag), are ignored. Since in RDF serializations no such distinction exists, the best a client can do is to blindly try to dereference all URIs. It would thus be beneficial to describe in a machine-readable manner if a property represents a link intended to be dereferenced or solely an identifier. In Hydra Vocabulary this is represented by the *Link* class. It can be used to define properties that represent dereferenceable links.

Moreover, in the Hydra Vocabulary there is the *IriTemplate* class, which is connected with the *IriTemplateMapping* class. Sometimes, 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 and a number of mappings. An IriTemplateMappging maps a variable in the IRI template to a property and may optionally specify whether that variable is required or not. An example for better understanding is presented in figure 2.6. 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.

Another important Hydra class is the *Operation* class, which in fact contains the necessary information in order

```
{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
  "@type": "IriTemplate",
  "template": "http://api.example.com/issues{?q}",
  "variableRepresentation": "BasicRepresentation",
  "mapping": [
    {
      "@type": "IriTemplateMapping",
      "variable": "q",
      "property": "hydra:freetextQuery",
      "required": true
    }
  ]
}
```

Figure 2.6: Description of an IRI Template

for an HTTP request from a client to a server to be valid. Operation class has a list of properties, each of which is used for a specific reason. The property *method* describes the HTTP method that is used and it is the only required property. Optionally, the *expects* property identifies the information which is expected from the web API, while the *returns* property specifies the information returned by the Web API on success. Finally, the *statusCodes* provides information about the statuses that might be returned.

Hydra Vocabulary has also an interesting feature which is presented via the *Supported Property* class. Since Hydra uses classes to describe the information expected or returned by an operation, it also defines a concept to describe the properties known to be supported by a class. More specifically, it is possible to define whether a specific property is *required* or whether it is *read-only* or *write-only* depending on the class it is associated with. An

example is presented in figure 2.7. As we may see, we can specify whether a property which is supported by a class is *required* - which means obligatory for the validity of a request, *readable* - whether the client is able to retrieve the property's value and *writeable* - whether the client is able to change the property's value. In the exact same

```
{
  "@context": "http://www.w3.org/ns/hydra/context.jsonld",
  "@id": "http://api.example.com/doc/#Comment",
  "@type": "Class",
  "title": "The name of the class",
  "description": "A short description of the class.",
  "supportedProperty": [
    ... Properties known to be supported by the class ...
    {
      "@type": "SupportedProperty",
      "property": "#property", // The property
      "required": true, // Is the property required in a request to be valid?
      "readable": false, // Can the client retrieve the property's value?
      "writeable": true // Can the client change the property's value?
    }
  ]
}
```

Figure 2.7: Hydra Supported Property Class

way, Hydra introduces the *Supported Operation* property which defines the operations supported by all instances of a class.

The Hydra core vocabulary is used along with JSON-LD, in order to enable the creation of hypermedia-driven APIs. JSON-LD[13] 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.

Hydra tried to combine the REST architectural style

[13]https://json-ld.org/

and the Linked Data principles. This combination is able to create opportunities to advance the Web of machines in a similar way that hypertext did for the human Web. However, Hydra did not become very popular, even though the idea behind it has a lot of benefits. In case we need to make an assumption on why Hydra has not gained the popularity it deserved we could say it is because sometimes ontologies are treated with suspicion due to their complexity. We however were inspired by this idea and tried to create a simple mechanism that would solve the above problems. Our work will be presented in the following chapter.

2.11 Apache Jena

Apache Jena (or Jena in short) is a free and open source Java framework for building semantic web and Linked Data applications. The framework is composed of different APIs interacting together to process RDF data. The interaction between the different APIs appear in the figure 2.8.

The most important feature of Apache Jena for our work is its ability to read and handle ontologies using a Java framework. In Jena each of the ontology languages has a profile, which lists the permitted constructs, the names of the classes and properties. The profile is bound to an ontology model, which is an extended version of Jena's Model class. The base Model allows access to the statements in a collection of RDF data, which is in fact a collection of triples (resource, property, value). *OntModel* extends this by adding support for the kinds of constructs expected to be in an ontology: classes (in a

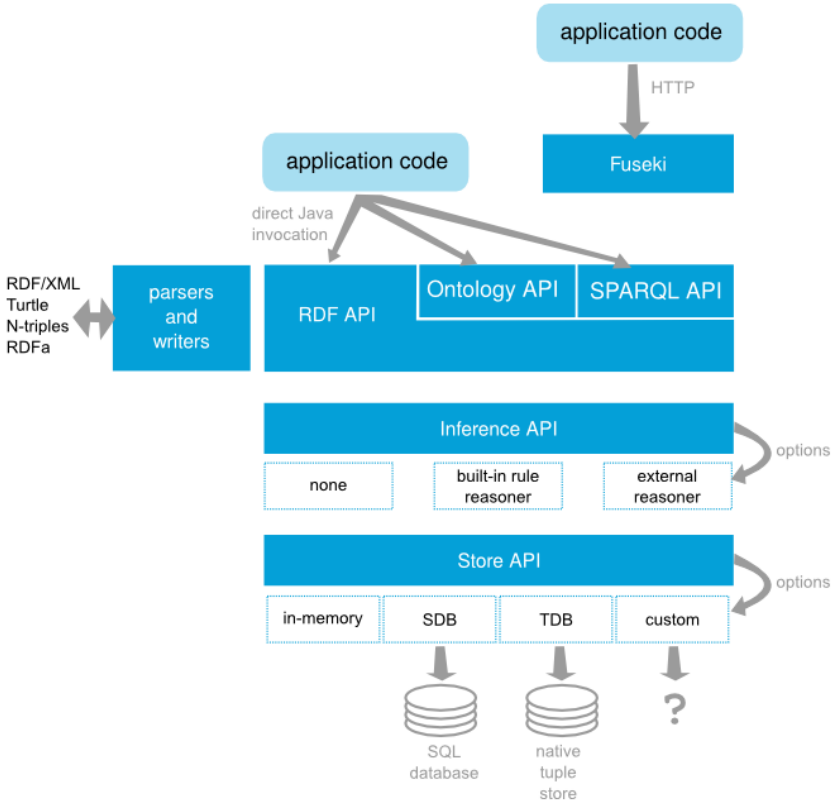

Figure 2.8: Apache Jena Framework

class hierarchy), properties (in a property hierarchy) and individuals.

Jena provides its users also with the opportunity to conduct queries on ontologies by using the SPARQL query language via the ARQ - A SPARQL Processor for Jena. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. The results of SPARQL queries can be result sets or RDF graphs.

2.12 Semantic Reasoners - Pellet

Pellet is a semantic reasoner, which is in fact a piece of software able to infer logical consequences from a set of asserted facts or axioms.

A semantic reasoner provides a standard set of De-

scription Logic inference services. First of all, there is the *consistency checking*, which ensures that an ontology does not contain any contradictory facts. Secondly, a semantic reasoner checks for *concept satisfiability*, which checks if it is possible for a class to have any instances. If class is unsatisfiable, then defining an instance of the class will cause the whole ontology to be inconsistent. Then, there is the *classification*, which computes the subclass relations between every named class to create the complete class hierarchy. The class hierarchy can be used to answer queries such as getting all or only the direct subclasses of a class. Finally, there is the *realization*, which finds the most specific classes that an individual belongs to. Pellet reduces all of the above services to consistency checking.

Chapter 3

The Semantic OpenAPI Specification 3.0

In this chapter, we present the Semantic OpenAPI Specification (SOAS) version 3.0, 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 ambiguities in OAS descriptions. To show proof of concept we discuss OAS giving emphasis to ambiguities inherent in OAS properties using the Swagger Petstore[14] example as a service use case.

3.1 Why OpenAPI Specification

When we were called to choose a description language for services we had to consider many factors such as the protocol or architecture style upon which the description language was built, the range of adoption from the computer science society, the documentation and the tools that were

[14]https://petstore.swagger.io/

provided.

Regarding the protocol or architecture style, it is reasonable that we wanted a description language for RESTful services, given that the majority of Cloud services are offered by means of Web services based on the REST architecture style. Hence, WSDL except being characterized as complex and not widely adopted is also not suitable for describing RESTful services. Similarly, WADL despite the fact that it was created for services based on REST architecture, it has the same disadvantages with WSDL.

Our other option was Hydra which is a very promising technology based entirely on Semantic Web. However, as already mentioned in the chapter 1, section 1.10 Hydra has not gained massive adoption probably because of its complexity.

For all the above reasons and because it covers all the factors that were mentioned in the first paragraph, we chose to adopt the OpenAPI Specification (OAS) as a description language for web services.

3.2 Swagger Petstore

Swagger Petstore is the most common example of an OpenAPI service. It is a virtual petstore, where we are able to see information about the pets (name, photoUrl, status etc), information about the client (id, username, firstname etc) as well as information about an order. We may also see the parameters that are required or not, in order to send a query to the service in the form of GET, PUT, POST and DELETE. By using Swagger Petstore it is also possible to be informed about the various responses returned (200 - successful operation or 404 - pet not found).

3.3 Description of the OAS

In this section we further describe the OpenAPI Specification. In the figure below is represented the structure of an OAS version 3 service description.

In OpenAPI Specification there are many objects, each

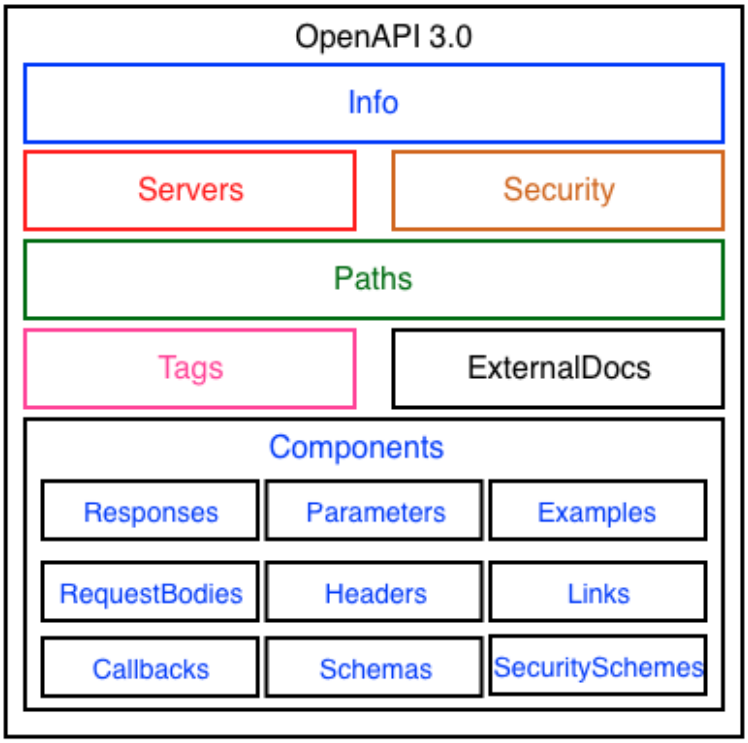

Figure 3.1: OAS v3 document Structure

one of which has a list of properties. The majority of the objects contain themselves other objects as properties. Therefore objects are linked to each other.

First of all, there is the *Info Object*. What it does is to provide metadata about the API. 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 illustrates an example of an Info Object. We may see the title of the application, its description, termsOfService, contact information,

license and version.

```
title: Sample Pet Store App
description: This is a sample server for a pet store.
termsOfService: http://example.com/terms/
contact:
  name: API Support
  url: http://www.example.com/support
  email: support@example.com
license:
  name: Apache 2.0
  url: https://www.apache.org/licenses/LICENSE-2.0.html
version: 1.0.1
```

Listing 3.1: Example of an Info Object

Next, there is the *Servers Object* which is in fact an object representing a server. It is required to contain the url to the target host. Specifically in Servers Object we specify the basepath (the part of the URL that appears before the endpoint) used in the API requests. Here, we may also find a description of the host designated by the URL, as well as variables that can be populated at runtime by the server (such as *username*). Additionally, if different paths (endpoints) require different server URLs, the Servers Object may be added as a property in the Path Object's Operation Object. The locally declared servers (i.e. the servers that are declared in Servers Object) URL will override the global servers (i.e. the servers that are declared in OpenAPI Object) URL.

```
servers:
- url: https://{username}.gigantic-server.com:{port}/{basePath}
  description: The production API server
  variables:
    username:
      # note! no enum here means it is an open value
      default: demo
      description: this value is assigned by the service provider,
    in this example `gigantic-server.com`
    port:
      enum:
        - '8443'
        - '443'
```

```
      default: '8443'
    basePath:
      # open meaning there is the opportunity to use special base
    paths as assigned by the provider, default is `v2`
      default: v2
```

Listing 3.2: Example of a Servers Object

As we can see in listing 3.2 the "variables" property of Server Object contains itself some properties as it is in fact an object itself, called *Server Variable Object*. The most important property of the Server Variable Object is the "enum" one. By using this property we are able to derive multiple server URL from one server declaration. In the listing we may see that we there are two servers with part ids 8443 and 443 with the former being the default (443 is used when explicitly mentioned in the request).

Next in the OAS Document Structure there is the *Security Requirement Object*. It lists the required security schemes to execute this operation. The name used for each property must correspond to a security scheme declared in the Security Schemes under the Components Object (will be explained later). If the security scheme is of type "oauth2" or "openIdConnect", then the value is a list of scope names required for the execution. For example, in listing 3.3 we may see an oauth2 SecurityRequirement Object which defines the type of object that we are able to write or read, in this case *pets*.

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

Listing 3.3: OAuth2 Security Requirement Example

Another important object in OAS Document Structure is the *Path Object*. It contains the relative paths for the service endpoints. Each Path item describes the avail-

able operations based on HTTP methods. The path is appended to the expanded URL from the Server Object's url field in order to construct the full URL. For example, in listing 3.4 the Path Object has value */pets*. It describes that with the *get* operation we get as a result a description with all the pets registered in the system. It also describes that when we get as a response the number '200' then we have a successful query that returns a list of all the pets.

```
/pets:
  get:
    description: Returns all pets from the system that the user
   has access to
    responses:
      '200':
        description: A list of pets.
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/pet'
```

Listing 3.4: Path Object Example

More specifically, the *get* operation that appears in listing 3.4 is in fact part of another object called *Path Item Object*. This object describes all operations available on a single path (i.e get, put, post, delete etc) as well as a list of parameters that are applicable for all the operations described under this path. In listing 3.5 we see an example of a Path Item Object. It contains the results that a *get* operation returns as well as the parameters which are the *name* of the id of the pet, its description and whether or not it is required.

```
get:
  description: Returns pets based on ID
  summary: Find pets by ID
  operationId: getPetsById
  responses:
    '200':
```

```
      description: pet response
      content:
        '*/*' :
          schema:
            type: array
            items:
              $ref: '#/components/schemas/Pet'
    default:
      description: error payload
      content:
        'text/html':
          schema:
            $ref: '#/components/schemas/ErrorModel'
parameters:
- name: id
  in: path
  description: ID of pet to use
  required: true
  schema:
    type: array
    style: simple
    items:
      type: string
```

Listing 3.5: Path Item Object Example

Next in the OAS Document Structure is the *Tag Object*, which adds metadata to a single tag that is used by the *Operation* Object. It is not mandatory to have a Tag Object per tag defined in the Operation Object instances. It is required to contain the name of the tag, while it might also contain a short description and an additional external documentation for the tag. An example of a Tag Object appears in listing 3.6, where we can see the name of the tag as well as its description. Moreover, in listing 3.7 we can see the tag *pet* and then the rest of the properties that an Operation Object has, including the summary of what that specific operation does (in this case : update a pet), the parameters it has to contain in order to be performed, the available responses and the scopes allowed by the security schemes.

```
name: pet
```

```
description: Pets operations
```

Listing 3.6: Tag Object Example

```
tags:
- pet
summary: Updates a pet in the store with form data
operationId: updatePetWithForm
parameters:
- name: petId
  in: path
  description: ID of pet that needs to be updated
  required: true
  schema:
    type: string
requestBody:
  content:
    'application/x-www-form-urlencoded':
      schema:
       properties:
          name:
            description: Updated name of the pet
            type: string
          status:
            description: Updated status of the pet
            type: string
       required:
         - status
responses:
  '200':
    description: Pet updated.
    content:
      'application/json': {}
      'application/xml': {}
  '405':
    description: Method Not Allowed
    content:
      'application/json': {}
      'application/xml': {}
security:
- petstore_auth:
  - write:pets
  - read:pets
```

Listing 3.7: Operation Object Example

Another object is the *External Documentation Object*. What this object does it to allow referencing an external resource for extended documentation.

```
description: Find more info here
url: https://example.com
```

Listing 3.8: External Documentation Object Example

One of the most important objects of OAS Document Structure is the *Components object*. It holds a set of reusable objects which can be responses, parameters, schemas, request bodies and more.

Schemas object allows the definition of input and output data types. These types can be objects, but also primitives and arrays. The specification introduces also additional properties supporting polymorphism (*discriminator* property). Schema object is an extended subset of the JSON Schema Specification. A Schema Object may borrow some properties directly from JSON Schema and use them in the exact same way (i.e properties *title, required*). In addition, it may borrow some properties but adjust them to the OpenAPI Specification (i.e properties *format, default*). Finally, a Schema Object may also have some properties of its own accord in order to have further documentation (i.e properties *discriminator, externalDocs*). An example of a Primitive Schema Object appears in listing 3.9, an example of a Simple Model appears in listing 3.10 and that of a Model with Polymorphism Support appears in listing 3.11. Listing 3.9 represents an e-mail with type String, Listing 3.10 illustrates properties of an object (*name* with type String, *address* which refers to a Schema called Address in Components Object and *age* with a type of Integer. Finally, listing 3.11 illustrates the use of the *discriminator* property in Schema *Pet*. In fact using this property we are able to define two more pet Schemas - *Cat* and *Dog*.

```
type: string
format: email
```

Listing 3.9: Primitive Schema Object Example

```yaml
type: object
required:
- name
properties:
  name:
    type: string
  address:
    $ref: '#/components/schemas/Address'
  age:
    type: integer
    format: int32
    minimum: 0
```

Listing 3.10: Simple Model Schema Object Example

```yaml
components:
  schemas:
    Pet:
      type: object
      discriminator:
        propertyName: petType
      properties:
        name:
          type: string
        petType:
          type: string
      required:
      - name
      - petType
    Cat: ## "Cat" will be used as the discriminator value
      description: A representation of a cat
      allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
        properties:
          huntingSkill:
            type: string
            description: The measured skill for hunting
            enum:
            - clueless
            - lazy
            - adventurous
            - aggressive
        required:
        - huntingSkill
    Dog: ## "Dog" will be used as the discriminator value
      description: A representation of a dog
      allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
        properties:
          packSize:
            type: integer
            format: int32
            description: the size of the pack the dog is from
            default: 0
            minimum: 0
        required:
```

```
        - packSize
```
Listing 3.11: Model with Polymorphism Support Schema Object

In order to define the connection between the three schemas defined in listing 3.11 via the discriminator value, we present the *Discriminator* Object. The Discriminator is a specific object in a schema which is used to inform the consumer of the specification of an alternative schema based on the value associated with it. The discriminator object is legal only when using one of the composite keywords oneOf, anyOf, allOf.

In listing 3.12, a response payload may be described to be exactly one of the types Cat, Dog or Lizard.

```
MyResponseType:
  oneOf:
  - $ref: '#/components/schemas/Cat'
  - $ref: '#/components/schemas/Dog'
  - $ref: '#/components/schemas/Lizard'
  discriminator:
    propertyName: petType
```
Listing 3.12: Discriminator Object Example

The expectation now is that a property with name petType MUST be present in the response payload, and the value will correspond to the name of a schema defined in the OAS document. Thus the response payload will be similar to the one appearing in listing 3.13. This would indicate that the Cat schema can be used in conjunction with this payload.

```
{
  "id": 12345, //the id of the pet
  "petType": "Cat"
}
```
Listing 3.13: Discriminator Object Example - Response Payload

To avoid redundancy, the discriminator may be added to a *parent* schema definition, and all schemas comprising the parent schema in an *allOf* construct may be used as an alternate schema. An example appears in listing 3.14 where we may see that all the pets (Dog, Cat and Lizard) are mapped to the parent schema aka *Pet* Schema.

```
{
  components:
  schemas:
    Pet:
      type: object
      required:
      - petType
      properties:
        petType:
          type: string
      discriminator:
        propertyName: petType

    Cat:
      allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
        # all other properties specific to a `Cat`
        properties:
          name:
            type: string
    Dog:
      allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
        # all other properties specific to a `Dog`
        properties:
          bark:
            type: string
    Lizard:
      allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
        # all other properties specific to a `Lizard`
        properties:
          lovesRocks:
            type: boolean
}
```

Listing 3.14: Discriminator Object Example - use of allOf

Next in the OAS Document under Components Object we find the *Responses Object*, which contains all the expected responses of an operation. It maps an expected

response to a specific HTTP status code, describing the message content and HTTP Headers that an operation's response may contain. An example appears in listing 3.15, which indicates that a response '200' signifies success while the default code is "unexpected error" in case where a "put" or "post" request does not return anything.

```
'200':
  description: a pet to be returned
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/Pet'
default:
  description: Unexpected error
  content:
    application/json:
      schema:
        $ref: '#/components/schemas/ErrorModel'
```

Listing 3.15: Responses Object Example

Under the Components Object appears also the *Security Scheme Object*. It contains the security schemes that the service uses for authentication. The specification offers support for basic HTTP authentication, API keys, OAuth2's common flows[15] and OpenID Connect[16].

However, there is another object connected with the Security one, which is called *OAuth Flow Object*. This object contains some mandatory properties, which are the authorizationUrl, the tokenUrl and the available scopes for the OAuth2 security scheme. An example for better understanding appears in listing 3.16. Under the OAuth Flow Object we define the scopes of an operation. Scopes illustrate what we are enabled to do (in this case *write* and *read* pets) The properties authorizationUrl, authorization-

[15]https://auth0.com/docs/protocols/oauth2

[16]https://openid.net/connect/

Code and tokenUrl are defined according to the OAuth procedure.

```
type: oauth2
flows:
  implicit:
    authorizationUrl: https://example.com/api/oauth/dialog
    scopes:
      write:pets: modify pets in your account
      read:pets: read your pets
  authorizationCode:
    authorizationUrl: https://example.com/api/oauth/dialog
    tokenUrl: https://example.com/api/oauth/token
    scopes:
      write:pets: modify pets in your account
      read:pets: read your pets
```

Listing 3.16: OAuth Flow Object Example

Another object that appears under Components is the *Parameters Object*. It contains all the parameters that a query to the server (aka an operation) may use. A unique parameter is defined by a combination of a name and location. In listing 3.17, we have a parameter (*token*) that is mandatory (*required : true*) in order for an operation to be performed. The specification categorizes parameters into five types specified by the **in** field :

- *Path* parameters are used in cases where the parameter values are part of operation's path.
- *Query* parameters are appended to the url when sending a request.
- *Header* parameters define additional custom headers that may be sent in a request.
- *Cookie* parameters are passed in the Cookie header.

```
name: token
in: header
description: token to be passed as a header
```

```
required: true
schema:
  type: array
  items:
    type: integer
    format: int64
style: simple
```

Listing 3.17: A header parameter with an array of 64 bit integer numbers

3.4 Enriching the OpenAPI Specification

In an OAS service document, there are many elements that share the same semantics. A human may easily infer these semantic similarities, but a machine cannot. In order for a machine to understand the meaning of OAS 3.0, a service description needs to be be semantically enriched. SOAS 3.0, introduces extra properties to annotate existing OAS properties which are proved (in the following) to be ambiguous. Table I summarizes the extension properties, their scope and their meaning. We will refer to these properties as *x-properties*.

First of all, we have added the *x-RefersTo* extension property that specifies the association between an OAS element and a concept in a semantic model. Listing 3.18 shows how x-refersTo is used to semantically annotate a Pet model and its properties: it associates the model with Pet class in schema.org vocabulary.

However, it is possible for a model to have a narrower meaning. For example, if the Pet model describes a specific group of pet (e.g. dogs), *x-kindOf* extension property is used instead to denote that the model is a subclass of the referred semantic concept. Both properties can be used

Table 3.1: OAS extension properties for semantic annotations

Property	Applies to	Meaning
x-refersTo	Schema Object	The concept in a semantic model that describes an OAS element.
x-kindOf	Schema Object	A specialization between an OAS element and a concept in a semantic model.
x-mapsTo	Schema Object	An OAS element which is semantically similar with another OAS element.
x-collectionOn	Schema Object	A model describes a collection over a specific property.
x-onResource	Tag Object	The specific *Tag* object refers to a resource described by a *Schema* object.
x-operationType	Operation Object	Clarifies the type of operation.

only with elements in Schema object and accept a URI that represents the concept in a semantic model.

```
parameters:
  Query:
    name: name
    in: query
    description: Pet's name for filtering
    required: true
    schema:
      type: string
      x-mapsTo: '#/components/schemas/Pet.name'
schemas:
  Pet:    # A Pet model extended with SOAS 3.0 properties
    type: object
    x-refersTo: http://schema.org/Pet
    properties:
      name:
        type: string
        x-refersTo: http://schema.org/petName
      photoUrls:
        type: string
        x-refersTo: http://schema.org/petPhoto
      id:
```

```
        type: integer
        x-refersTo: http://schema.org/petId
    required:
      - name
      - photoUrls
    discriminator:
      propertyName: name
Dog:  # A Dog model extending the Pet Model
  description: A representation of a dog pet
  x-kindOf: http://schema.org/Pet
  allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
        properties:
            height:
                type: integer
                    description: height in cms
          weight:
              type: integer
                 description: weight in kgs
       required:
           - height
           - weight
Cat:  # A Cat model extending the Pet Model
   description: A representation of a cat pet
   x-kindOf: http://schema.org/Pet
   allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
            properties:
            eyesColor:
                type: string
      required:
        - eyesColor
```

Listing 3.18: OAS Model polymorphism example

Next in the series of the x-properties there is the *x-mapsTo*. It is used to define Schema object elements that share the same semantics. In Listing 3.18, x-mapsTo property is used in Parameters Object to dictate that query parameter name refers to Pet.Name in Schemas object.

The next x-property is *x-collectionOn*. It is used to indicate that a model in Schemas object is actually a collection. Typically, a collection (or a list) of resources in OAS 3.0 is described using the array type. However, it is very common a collection's definition to be encapsulated within an object type with additional properties. Then, x-collectionOn property is used to denote the data types of

the objects of the collection. Listing 3.19 defines a model as a collection of Pet objects (totalItems property denotes population).

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

Listing 3.19: Model definition representing a collection

The x-onResource extension property is used in Tag Objects to specify the resource that a tag refers. In OAS 3.0, tags are used to group operations either by resources or any other qualifier. 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 Schemas but a machine cannot. The x-onResource property is used to associate the tag with a Schema object that describes a specific resource. In Listing 3.20 x-onResource property is assigned on a pet tag that provides information regarding the operations that are available for Pet model in Schemas object.

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

paths:
  /pet/findByStatus:
    get:
      x-operationType: 'http://schema.org/SearchAction'
      tags:
```

```
    - pet
  summary: Finds Pets by status
  description: Multiple status values can be provided
  operationId: findPetsByStatus
  parameters:
    - $ref: '#/components/parameters/statusQuery'
  responses:
    '200':
      description: successful operation
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/Pets"
    '400':
      description: Invalid status value
  security:
  - petstore_auth:
  - 'write:pets'
  - 'read:pets'
```

Listing 3.20: Semantically enriched Example - Swagger Petstore OAS service description

Finally, x-operationType extension property is used to specify the type of an Operation object. A request is characterized by the HTTP method that it uses. However, the semantics of the HTTP methods are too generic and may have a more specific meaning. For example, in Listing 3.20, this property is used to clarify that a GET request on path /pet/findByStatus is a search operation on pets based on their status. The value of the property is a URL pointing to the concept that semantically describes the operation type. The *Action* type of the *Schema.org* vocabulary provides a detailed hierarchy of Action sub-types that can be used by the property.

3.5 OpenAPI v3 Ontology

OpenAPI ontology in Figure 3.2 captures all information specified by SOAS 3.0 description. Properties of classes are related with other other classes explaining them (i.e.

SOAS 3.0 properties are mapped to classes as well).

At the heart of the ontology is Hydra Core Vocabulary. Schemas, Operations, Resources and Properties are mapped to Hydra models. For models not supported by Hydra (i.e. security, headers, constraints) new models are introduced in SOAS 3.0 ontology.

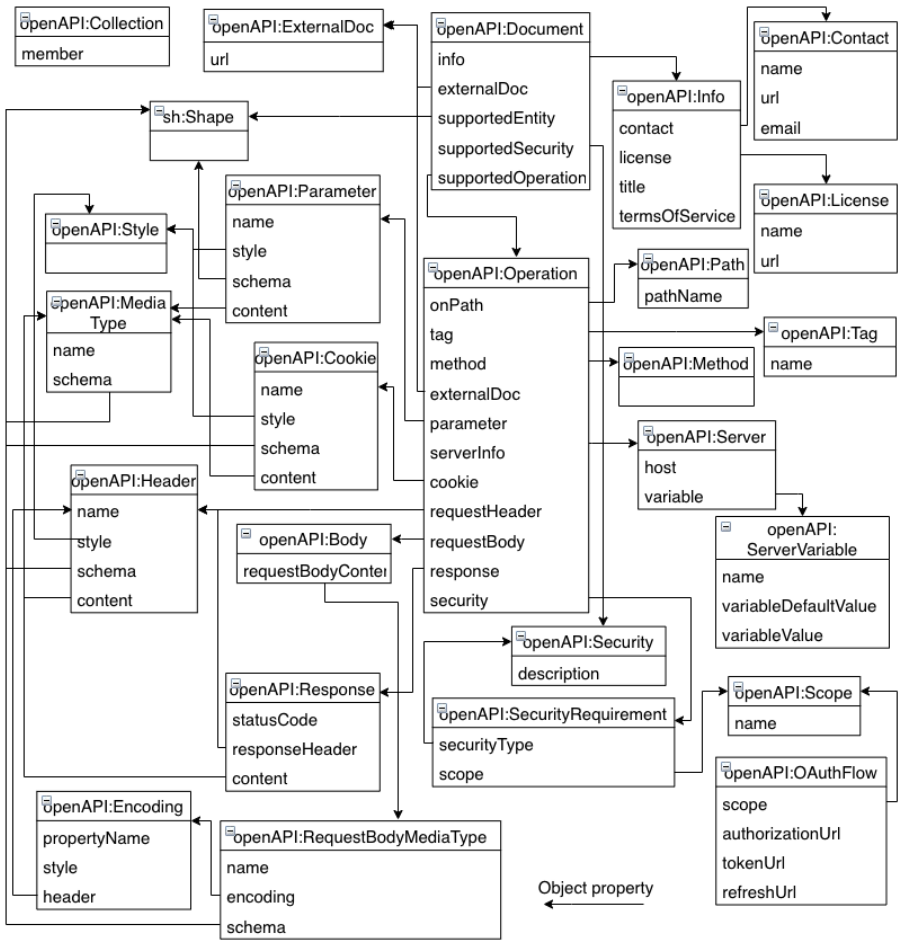

Figure 3.2: OpenAPI Version 3 Ontology

In accordance to OAS 3.0 structure, *Document Class* provides general information (Info class) regarding the service; it specifies service paths, the entities and the security schemes that it supports. *Path class* represents (relative) service paths (pathName property). *Operation class* provides information for sending HTTP requests to the service as well as the HTTP responses. Responses are fur-

ther described by *Response class*, specifying the status code and the data returned. The entire range of HTTP response values is represented. *Operation Class* refers to a security scheme in *SecurityRequirement Class*. Listing 3.21 is an example representation of *Info* as seen in OAS 3.0 document. The example is written in turtle language for ontologies. As we may see, we have a representation of all properties belonging to Info Object, such as service-Title and description.

```
...
ex:PetStoreDocument
  a openapi:Document ;
  openapi:info [
    openapi:serviceTitle "Sample Pet Store App";
    openapi:description "This is a sample server for a pet store."
    ;
    openapi:termsOfService <http://example.com/terms/> ;
    openapi:contact [
      openapi:creator "API Support" ;
      openapi:url <http://www.example.com/support> ;
      openapi:email <support@example.com> .
    ] ;
    openapi:license [
      openapi:licenseName "Apache 2.0" ;
      openapi:url <http://www.apache.org/licenses/LICENSE-2.0.html
   >
    ]
  ] ;
  openapi:version "1.0.1" ;
  \# the defined entities
  openapi:supportedEntity ex:PetShape, ex:PetCollectionShape, ex:
   ErrorShape ;
  \# the defined operations
  openapi:supportedOperation ex:op1, ex:op2, ex:op3, ex:op4 .
...
```

Listing 3.21: Representation of an OAS Document in the Ontology

In addition, Listing 3.22 illustrates how an OAS *Path* item and *Operation* are defined in the ontology using the example of Listing 3.20. More specifically, we may see a

Path Individual (path2) that describes the path of finding a pet using its status. Moreover, we can see an Operation Individual (path2_op1) and its properties, such as tag, parameter, response and security.

```
...
ex:path2
  a openapi:Path ;
  openapi:pathName "/pets/findByStatus" ;
ex:path2_op1
  a openapi:Operation, schema:SearchAction ;
  openapi:onPath ex:path2
  openapi:method openapi:GET ;
  openapi:tag ex:tag_pet ;
  openapi:parameter ex:query_status ;
  openapi:response [
    openapi:statusCode 200 ;
    openapi:content [
      openapi:mediaTypeName "application/json" ;
      openapi:schema ex:PetCollectionShape
    ] ;
    openapi:description "successful operation"
  ] ;
  openapi:response [
    openapi:statusCode 400 ;
    openapi:description "Invalid status value" .
  ] ;
  openapi:security [
    openapi:securityType ex:petstore_oauth ;
    opneapi:scope ex:read_pets, ex:write_pets
  ] ;
  openapi:name "findPetsByStatus" ;
  openapi:summary "Finds Pets by Status" ;
  openapi:description "Multiple status values  with comma
   seperated strings" .
...
```

Listing 3.22: Representation of *Path* and *Operation* in the ontology

Operation Individual *path2_op1* refers to a *SecurityRequirement* individual, specifying an OAuth2 security scheme (i.e. *petstore_oauth* individual) and the corresponding scope (i.e. *read_pets* and *write_pets* individuals). Individual *path2_op1* is also considered to be an individual of *SearchAction* type defined in *Schema.org* vocabulary (i.e. as defined by the *x-operationType* extension prop-

erty).

Next, in figure 3.3 the security schemes supported by OAS 3.0 are displayed. We may see that Security Class is represented by ApiKey, Http, OAuth2 and OAuthIdConnect, which are in fact the types that a Security Scheme Object can have. In addition, we see that OAuth2 Class in specific is represented by OAuth Flows Object that has the properties implicit, password, clientCredentials, authorizationCode and scope.

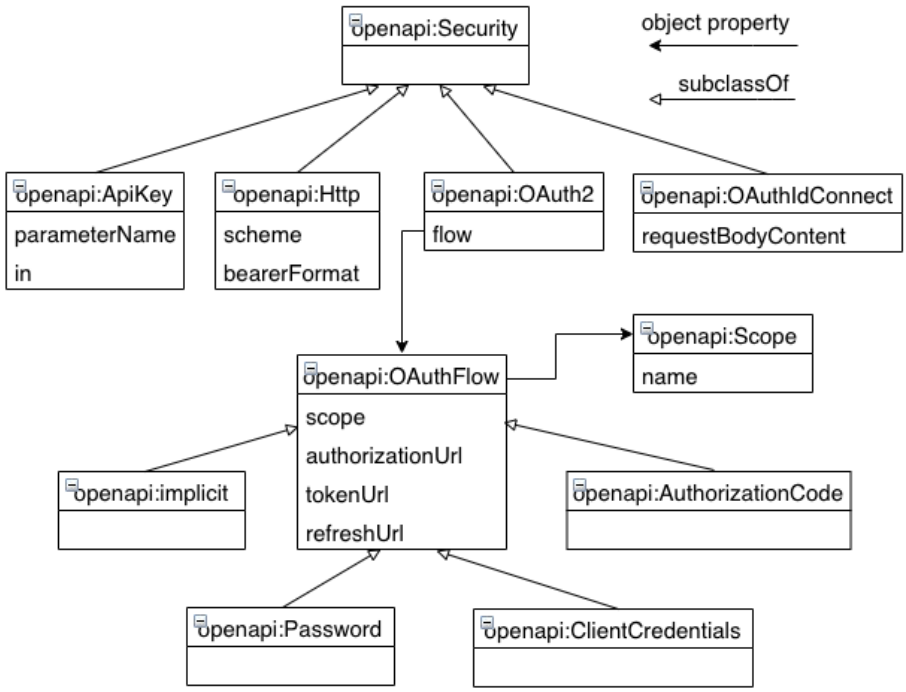

Figure 3.3: Security Class in OpenAPI v3 Ontology

Schema Objects are expressed as classes, object and data properties using SHACL vocabulary[16]. SHACL is an RDF vocabulary that can be used to describe and validate the structure of RDF data, similarly to XML-Schema or JSONSchema. SHACL can be used to define classes together with constraints on their properties. It provides built-in types of constraints (e.g. cardinality : minCoun-

[16]https://www.w3.org/TR/shacl/

t/maxCount) and allows expression of constraints (as well as logical combinations of such constraints) on the type of properties and on the values the properties can take. Table 3.2 shows the direct mapping of Schema Object properties with the SHACL vocabulary. The OpenAPI ontology also defines additional properties for describing the remaining Schema Object properties. While Schema Object of OAS 3.0 is mapped in Shape Class in the ontology. Shape Class is distinguished into in *NodeShape* Class and *Property-Shape* Class. The *Node- Shape* class represents the classes that describe the models of an OAS 3.0 description. It represents operations related to a class (supportedOperation), which comes from x-onResource extension property. It defines the properties of a class and specifies whether a class may contain additional properties (additionalProperties) of a specific type. Class *PropertyShape* represents the properties of a class, their datatype and restrictions (e.g. a maximum value for a numeric property) and indicates whether the supported property is required, read-only in the class definition.

Listing 3.23 shows how the Pet model of Listing 3.18 is represented in the OpenAPI ontology. The model contains references to the Schema.org vocabulary using the x-refersTo extension property. The SHACL class PetShape is now defined according to Schema object definition of Pet with the addition of new data properties and constraints (e.g. each pet has exactly one name, photo and id). A model defined using the combination of allOf property and discriminator property, is represented in the OpenAPI ontology as a subclass of the model that is extended. A subclass defined using x-kindOf become a subclass of the

Table 3.2: Mapping OpenAPI Schema Object properties to SHACL

Schema Object property	SHACL property
maximum	sh:exclusiveMaximum if openAPI
exclusiveMaximum	exclusiveMaximum is true sh:inclusiveMaximum if openAPI exclusiveMaximum is false
minimum	sh:exclusiveMinimum if openAPI
exclusiveMinimum	exclusiveMinimum is true sh:inclusiveMinimum if openAPI exclusiveMinimum is false
maxLength	sh:maxLength
minLength	sh:minLenght
pattern	sh:pattern
maxItems	sh:maxCount
minItems	sh:minCount
enum	sh:in
allOf	sh:and
oneOf	sh:xone
anyOf	sh:or
not	sh:not
default	sh:defaultValue

referenced semantic concept.

```
...
ex:PetShape
  a sh:NodeShape ;
  sh:targetClass schema:Pet ;
  sh:property [
    sh:path schema:petName ;
    sh:name "name" ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ];
  sh:property [
    sh:path schema:petPhoto ;
    sh:name "photoUrls" ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ];
  sh:property [
```

```
    sh:path schema:petId ;
    sh:name "id" ;
    sh:datatype xsd:integer ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
  ] .
...
```

Listing 3.23: Representation of an OAS model in the openAPI ontology

Collections are represented using Collection class. Listing 3.24 is the representation of PetCollection class of Listing 3.19. Class PetCollection becomes a subclass of class Collection. Without the x-collectionOn extension property in Listing 3.19, the PetCollection model would be a simple class without any reference of being a collection.

```
...
ex:PetCollection
  rdfs:subClassOf openAPI:Collection .

  ex:PetCollectionShape
  a sh:NodeShape ;
  sh:targetClass ex:PetCollection ;
  sh:property [
    sh:path openapi:member ;
    sh:name "pets" ;
    sh:class ex:Pet ;
  ] ;
  sh:property [
    sh:path ex:totalItems ;
    sh:name "totalItems" ;
    sh:datatype xsd:integer ;
  ] .
...
```

Listing 3.24: Representation of collections in the openAPI ontology

Finally, OAS parameters are represented as separate classes for every parameter type. Class *Header* of Figure 3.2 contains definitions of header parameters that are used in HTTP requests and responses. Class *Cookie* defines the cookies that are sent through HTTP requests and re-

sponses. Class *Parameter* defines parameters that are attached to operation's URL.

Chapter 4

Instantiating OpenAPI Services to the OpenAPI Ontology

In this chapter, we first present the main idea of mapping an OpenAPI service to the OpenAPI version 3 ontology so as to benefit from semantic web tools such as reasoners and query languages for service discovery. We do that by providing some tables in which the OpenAPI Objects and their properties are assigned to ontology Classes and their properties. Then, we provide an abstract form of the algorithm on which our mechanism is based on. The complete algorithm is presented in chapter 5. The input is an OAS description of a service that contains Objects. The output is an instantiated Ontology where all services properties are represented. For each Object we present the table and then its algorithm.

OpenAPI Object		*OpenAPI Document Class*	
Field Name	**Type**	**Property**	**Range**
openapi	string	Not applied.	
info	Info Object	openapi:info	openapi:Info
servers	Server Object	Not applied.	
paths	Paths Object	Not applied.	
components	Components Object	Not applied.	
security	Security Requirement Object	Not applied.	
tags	Tag Object	Not applied.	
externalDocs	External Documentation Object	openapi:externalDoc	openapi:ExternalDoc
		openapi:supported Operation	openapi:Operation
		openapi:supported Entity	sh:Shape

Table 4.1: OpenAPI Object to OpenAPI Document Class

4.1 Mapping of OpenAPI Object to Document Class

The *openapi* property is not associated with a property in the ontology. This happens because this property is only used in order to access its components. The same thing happens with *components* property.

Moreover, the *servers* property is not associated with a specific property in our ontology. The property declares Server information that is used across the API. If an alternative server object is specified at the Path Item Object or Operation level, it will be overridden by this value. Therefore, in the OpenAPI ontology, server info is defined for every operation.

The *paths* property is actually replaced by the openapi:supportedOperation which, in turn, is described by OpenAPI:Operation class. Next, we have the *security* property which declares security schemes that are used across the API. However, an operation can override this global declaration and introduce other security schemes. Therefore, in the OpenAPI ontology, security is defined for every operation.

Next, property *tags* is not associated directly to a property of Document Class as tags are only kept in property tag of Operation.

Property *supportedOperation* has been added in order to relate Operations with classes while using the x-onResource property. We have also added property *supportedEntity* in order to keep the Schema (Shape) when related to a Tag via the x-onResource property. It is important to be mentioned here that *Document* is the main class. This means that the additional properties mentioned above are used in the parts of the algorithm where x-onResource is implemented aka Tag and Operation Object.

Listing 4.1 illustrates the mapping between the OpenAPI Object and the Document Class in the form of a simple algorithm.

```
function parseDocumentObject (openAPI_Document)
 - Initialize the Ontology Model
 - Create Document Individual
 - Call Info Function and save Info Individual in Document
   Individual
 - Keep a list with all the Global Servers
 - For every Server Object in OpenAPI Service :
   - Call Server Function and save Server Individual in Document
   Individual
   - Add the Server Individual to the list
 - For every Security Scheme Object in OpenAPI Service :
   - Call Security Scheme Function and save Security Scheme
   Individual in Document Individual
```

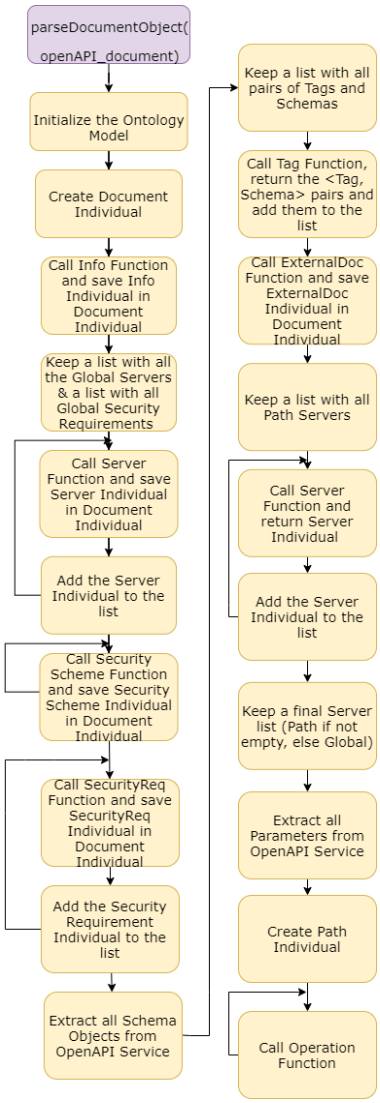

Figure 4.1: Workflow of OpenAPI Object to Document Individual mapping

```
- Keep a list with all the Global Security Requirements
- For every Security Requirement Object in OpenAPI Service :
  - Call Security Requirement Function and save Security
  Requirement Individual in Document Individual
  - Add the Security Requirement Individual to the list
- Extract all Schema Objects from OpenAPI Service
- Keep a list with all pairs of Tags and Schemas // used for x-
  onResource
- Call Tag Function, return the <Tag, Schema> pairs and add them
   to the list
- Call ExternalDoc Function and save ExternalDoc Individual in
  Document Individual
- Keep a list with all Path Servers
- For every Path Server :
  - Call Server Function and return Server Individual
  - Add the Server Individual to the list
- Keep a final Server list (Path if not empty, else Global)
- Extract all Parameters from OpenAPI Service
```

```
- Create Path Individual
- For every Operation Object, call Operation Function
```

Listing 4.1: OpenAPI Object to Document Individual

4.2 Mapping of Info Object to Info Class

Info Object		*OpenApi Info Class*	
Field Name	**Type**	**Property**	**Range**
title	string	openapi:serviceTitle	xsd:string
description	string	openapi:description	xsd:string
termsOfService	string	openapi:termsOfService	xsd:anyURI
contact	Contact Object	openapi:contact	openapi:Contact
license	License Object	openapi:license	openapi:License

Table 4.2: Info Object to Info Class

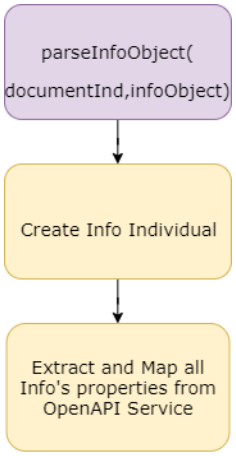

Figure 4.2: Workflow of Info Object to Info Individual mapping

```
function parseInfoObject (documentInd,infoObject)
 - Create Info Individual
 - Extract and map all Info's properties from OpenAPI Service
 (title, description, termsOfService, contact, etc)
```

Listing 4.2: Info Object to Info Individual

4.3 Mapping of Contact Object to Contact Class

Contact Object		*OpenAPI Contact Class*	
Field Name	**Type**	**Property**	**Range**
name	string	openapi:contactName	xsd:String
url	string	openapi:url	xsd:anyURI
email	string	openapi:email	xsd:anyURI

Table 4.3: Contact Object to Contact Class

```
function parseContactObject (contactObject)
  - Create Contact Individual
  - Extract and map all Contact's properties from OpenAPI
    Service
  - Return Contact Individual
```

Listing 4.3: Contact Object to Contact Individual

4.4 Mapping of Licence Object to Licence Class

License Object		*OpenAPI License Class*	
Field Name	**Type**	**Property**	**Range**
name	string	openapi:licenseName	xsd:String
url	string	openapi:url	xsd:anyURI

Table 4.4: License Object to License Class

```
function parseLicenseObject (contactObject)
  - Create License Individual
  - Extract and map all License's properties from OpenAPI
    Service
  - Return License Individual
```

Listing 4.4: License Object to Licence Individual

4.5 Mapping of a Server Object to Server Class

Here we present the mapping of both Server Object and Server Variable Object. For the correspondence of the properties we use table 4.5 and 4.6, while the parsing to Classes appears in listing 4.5.

Server Object		*OpenAPI Server Class*	
Field Name	**Type**	**Property**	**Range**
url	string	openapi:host	xsd:anyURI
description	string	openapi:description	xsd:String
variables	Map[string, Server Variable Object]	openapi:variable	openapi: ServerVariable

Table 4.5: Server Object to Server Class

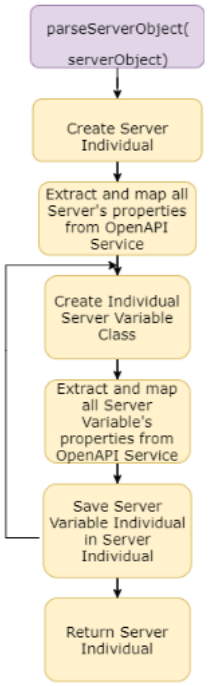

Figure 4.3: Workflow of Server Object to Server Individual mapping

As we may see in table 4.6 the property *name* in our ontology is not associated with a property in the OpenAPI

Server Variable Object		*OpenAPI Server Variable Class*	
Field Name	**Type**	**Property**	**Range**
		openapi:name	xsd:String
enum	string	openapi:variableValue	xsd:String
default	string	openapi: variableDefaultValue	xsd:String
description	string	openapi:description	—xsd:String

Table 4.6: Server Variable Object to Server Variable Class

service. This property was added, because it is required since the Server Object has a Map of a key-value pair where a String is the key and a Server Variable Object is the value. The property *name* is the key of the map.

```
function parseServerObject(serverObject)
  -Create Server Individual
  -Extract and map all Server's properties from OpenAPI Service
  -for (serverVariableObject in serverObject.getServerVariables)
    -Create Individual Server Variable Class
    -Extract and map all Server Variable's properties from OpenAPI
      Service
    -Save Server Variable Individual in Server Individual
  - Return Server Individual
```

Listing 4.5: Server Object to Server Individual

4.6 Mapping of Operation Object to Operation Class

Since each Path contains an Operation, property *onPath* is used in order to keep the Path Individual. Moreover, property *method* has been added in order to define the type of an Operation (GET, PUT, POST etc). In addition, since Parameters can have the type Path, Query, Header or Cookie we have added the properties *cookie*, *query* and *requestHeader* to our ontology. Operation is a very important Object, since it contains many other Ob-

Operation Object		*OpenAPI Operation Class*	
Field Name	**Type**	**Property**	**Range**
		openapi:onPath	openapi:Path
		openapi:method	openapi:Method
tags	string	openapi:tag	openapi:Tag
summary	string	openapi:summary	xsd:String
description	string	openapi:description	xsd:String
external Docs	External Documentation Object	openapi:externalDoc	openapi:ExternalDoc
operationId	string	openapi:name	xsd:String
parameters	Parameter Object	openapi:parameter	openapi:Parameter
		openapi:cookie	openapi:Cookie
		openapi:query	openapi:Query
		openapi: requestHeader	openapi:Header
requestBody	Request Body Object	openapi:requestBody	openapi:Body
responses	Response Object	openapi:response	openapi:Response
deprecated	boolean	openapi:deprecated	xsd:Boolean
security	Security Requirement Object	openapi:security	openapi: SecurityRequirement
servers	Server Object	openapi:serverInfo	openapi:Server

Table 4.7: Operation Object to Operation Class

jects. In addition, the Algorithm that shows the parsing of Operation Object to Operation Individual contains also the implementation of the x-operationType property. Listing 4.6 illustrates the abstract algorithm.

```
function parseOperationObject(documentInd, pathInd,
    OperationObject, tagShapeList, pathParametersList,
    pathServersList, globSecReqList)
  - Create Operation Individual
  - Extract and map all Operation's properties from OpenAPI
    Service
  - Call ExternalDoc Function and save ExternalDoc Individual to
    Operation Individual
```

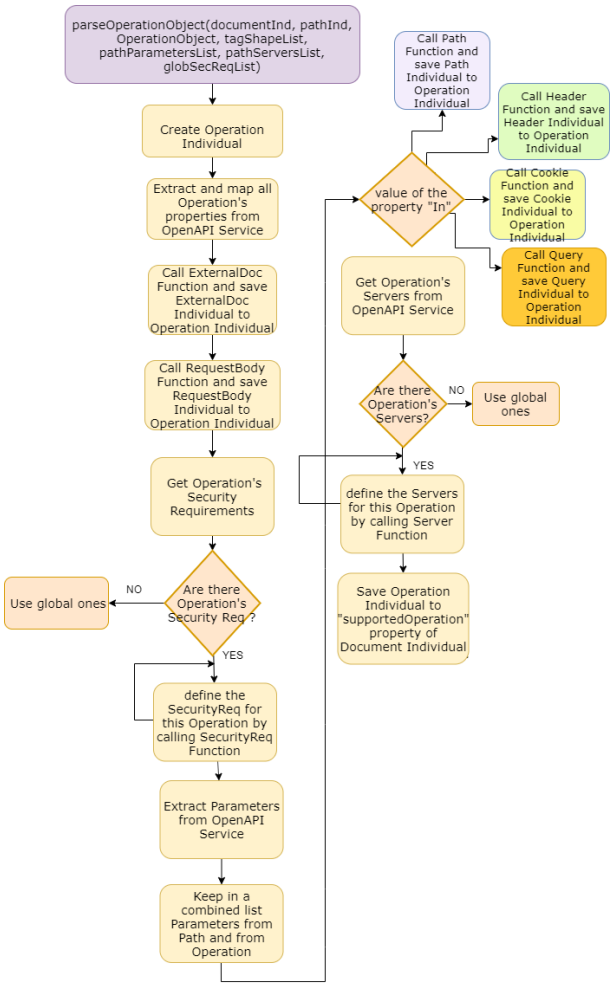

Figure 4.4: Workflow of Operation Object to Operation Individual mapping

```
– Call RequestBody Function and save RequestBody Individual to
  Operation Individual
– Get Operation's Security Requirements
    – if there are no Operation's SecurityReq, use global ones
    – else define the SecurityReq for this Operation by calling
  SecurityReq Function

  **Parameters shared by all operations of a path can be defined
   on the path level instead of the operation level. If any extra
   parameters defined at the operation level are used together
  with path-level parameters. Specific path-level parameters can
  be overridden on the operation level, but cannot be removed.**

  – Extract Parameters from OpenAPI Service
  – Keep in a combined list Parameters from Path and from
  Operation
    //Parameters could be in different positions
    – Depending on the value of the property "In" the
  corresponding function is called and the Parameter Individual
  returned is saved on      Operation Individual (Path, Query,
  Header, Cookie)
    – Get Operation's Servers from OpenAPI Service
```

```
    - if there are no Operation's Servers, use global ones
    - else define the Servers for this Operation by calling
Server Function
  - Save Operation Individual to "supportedOperation" property
of Document Individual

  ***x-operationType implementation***
- Get the resource where x-operationType points
- Set resource-class  as second class of the new Operation
    Individual
```

Listing 4.6: Operation Object to Operation Individual

4.7 Mapping of External Doc Object to External Doc Class

External Documentation Object		*OpenAPI External Documentation Class*	
Field Name	**Type**	**Property**	**Range**
url	String	openapi:url	xsd:String
description	String	openapi:description	xsd:String

Table 4.8: External Doc Object to External Doc Class

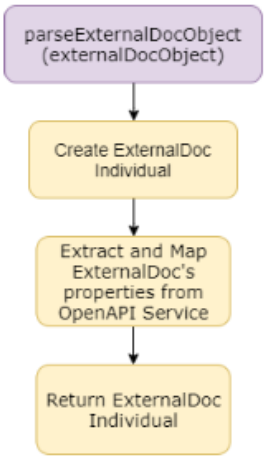

Figure 4.5: Workflow of External Doc Object to External Doc Individual mapping

```
function parseExternalDocObject(externalDocObject)
  - Create ExternalDoc Individual
```

```
  - Extract and Map ExternalDoc's properties from OpenAPI Service
  - Return ExternalDoc Individual
```

Listing 4.7: External Doc Object to External Doc Individual

4.8 Mapping of Parameter Object to Path, Query, Cookie, Header Class

There are four possible parameter locations specified by the *in* field. Path - where the parameter values actually part of the operation's URL. Query - Parameters that are appended to the URL. Header - Custom headers that are expected as part of the request. Cookie - Used to pass a specific cookie value to the API. Depending from this field, the Parameter Object is mapped to one of the above Classes. Each of the listings 4.8-4.11 represent the mapping of the Parameter Class to the corresponding Class depending of the value of the *in* property.

```
function parseCookieParameterObject(cookieObject, componentSchemas
    )
  - Create CookieParameter Individual
  - Extract and Map CookieParameter's properties from OpenAPI
    Service
  - Extract Schema Object from CookieParameter Object
  - Call Schema function and return Shape Individual
  - Save Shape Individual to "schema" property of CookieParameter
    Individual
  - Extract MediaType Object from CookieParameter Object
  - Call MediaType function and return MediaType Individual
  - Add MediaType Individual to MediaType list
  - Save MediaType list to "content" property of CookieParameter
    Individual
  - Return CookieParameter Individual
```

Listing 4.8: Parameter Object to Cookie Individual

```
function parseHeaderParameterObject(headerName, headerObject,
    componentSchemas)
  - Create HeaderParameter Individual
  - Assign "headerName" value from OpenAPI Service to "headerName"
     property of Header Individual
  - Extract and Map HeaderParameter's properties from OpenAPI
    Service
  - Extract Schema Object from HeaderParameter Object
  - Call Schema function and return Shape Individual
```

Parameter Object		*OpenAPI Parameter (Path, Query) Class, Cookie, Header Classes*	
Field Name	**Type**	**Property**	**Range**
name	String	openapi:name	xsd:String
in	String		
description	String	openapi:description	xsd:String
required	boolean	openapi:required	xsd:Boolean
deprecated	boolean	openapi:deprecated	xsd:Boolean
allowEmptyValue	boolean	openapi: allowEmptyValue	xsd:Boolean
style	String	openapi:style	openapi:Style
explode	boolean	openapi:explode	xsd:Boolean
allowReserved	boolean	openapi: allowReserved	xsd:Boolean
schema	Schema Object	openapi:schema	sh:Shape
content	Map[string, Media Type Object]	openapi:content	openapi: MediaType

Table 4.9: Parameter Object to Parameter Class

```
  - Save Shape Individual to "schema" property of HeaderParameter
    Individual
  - Extract MediaType Object from HeaderParameter Object
  - Call MediaType function and return MediaType Individual
  - Add MediaType Individual to MediaType list
  - Save MediaType list to "content" property of HeaderParameter
    Individual
  - Return HeaderParameter Individual
```

Listing 4.9: Parameter Object to Header Individual

```
function parseQueryParameterObject(queryObject, componentSchemas)
  - Create QueryParameter Individual
  - Extract and Map QueryParameter's properties from OpenAPI
    Service
  - Extract Schema Object from QueryParameter Object
  - Call Schema function and return Shape Individual
  - Save Shape Individual to "schema" property of QueryParameter
    Individual
  - Extract MediaType Object from QueryParameter Object
  - Call MediaType function and return MediaType Individual
  - Add MediaType Individual to MediaType list
  - Save MediaType list to "content" property of QueryParameter
    Individual
```

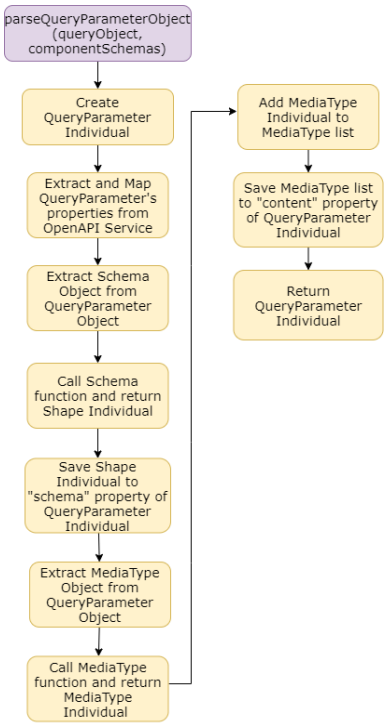

Figure 4.6: Workflow of QueryParameter Object to QueryParameter Individual mapping

```
  - Return QueryParameter Individual
```

Listing 4.10: Parameter Object to Query Individual

```
function parsePathParameterObject(pathObject, componentSchemas)
  - Create PathParameter Individual
  - Extract and Map PathParameter's properties from OpenAPI
    Service
  - Extract Schema Object from PathParameter Object
  - Call Schema function and return Shape Individual
  - Save Shape Individual to "schema" property of PathParameter
    Individual
  - Extract MediaType Object from PathParameter Object
  - Call MediaType function and return MediaType Individual
  - Add MediaType Individual to MediaType list
  - Save MediaType list to "content" property of PathParameter
    Individual
  - Return PathParameter Individual
```

Listing 4.11: Parameter Object to Path Individual

4.9 Mapping of Request Body Object to Request Body Class

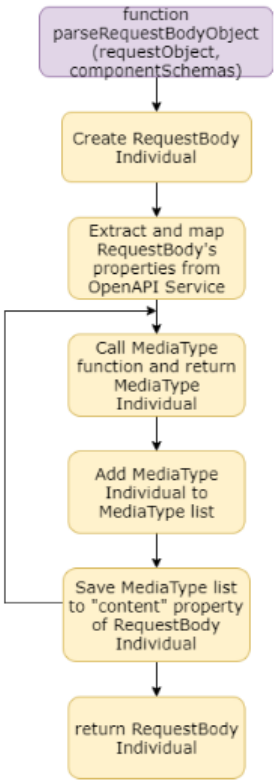

Figure 4.7: Workflow of RequestBody Object to RequestBody Individual mapping

Request Body Object		*OpenAPI Request Body Class*	
Field Name	**Type**	**Property**	**Range**
description	String	openapi:description	xsd:String
content	Map[string, Media Type Object]	openapi:content	openapi: RequestBodyMediaType
required	boolean	openapi:required	xsd:Boolean

Table 4.10: Request Body Object to Request Body Class

```
function parseRequestBodyObject(requestObject, componentSchemas)
  -Create RequestBody Individual
  -Extract and map RequestBody's properties from OpenAPI Service
  -for every MediaType Object :
    - Call MediaType function and return MediaType Individual
    - Add MediaType Individual to MediaType list
    - Save MediaType list to "content" property of RequestBody
    Individual
  - return RequestBody Individual
```

Listing 4.12: Request Body Object to Request Body Individual

4.10 Mapping of Media Type Object to Media Type Class

Property *mediaTypeName* as seen in table 4.11 is used as the key to the Map[string, Media Type Object] of table 4.10. This map as seen in table 4.10 represents the type of the *content* property.

Media Type Object		*OpenAPI Media Type Class*	
Field Name	**Type**	**Property**	**Range**
		openapi: mediaTypeName	xsd:String
schema	Schema Object	openapi:schema	sh:Shape
encoding	Map[string, Encoding Object]	openapi:encoding	openapi:Encoding

Table 4.11: Media Type Object to Media Type Class

```
function parseMediaTypeObject(mediaName, mediatypeObject,
    componentSchemas)
  - Create MediaType Individual
  - Assign "mediaName" value from OpenAPI Service to "mediaName"
    property of MediaType Individual
  - Call Schema function and return Shape Individual
  - Create a list for Encoding Individuals
  - For each Encoding Object in Media Type Object :
    - Call Encoding function and return Encoding Individual
    - Add the Encoding Individual to the list
  - Assign the list to the "encoding" property of MediaType
    Individual
  - Return MediaType Individual
```

Listing 4.13: Media Type Object to Media Type Individual

4.11 Mapping of Encoding Object to Encoding Class

Similarly, property *propertyName* has been added in table 4.12 in order to work as the key to the Map[string, Encoding Object] which is the type of the property *encoding*

seen in table 4.11.

Encoding Object		*OpenAPI Encoding Class*	
Field Name	**Type**	**Property**	**Range**
		openapi: propertyName	xsd:String
contentType	String	openapi: contentType	xsd:String
headers	Map[string, Header Object]	openapi: encodingHeader	openapi:Header
style	string	openapi:style	openapi:Style
explode	boolean	openapi:explode	xsd:Boolean
allowReserved	boolean	openapi: allowReserved	xsd:Boolean

Table 4.12: Encoding Object to Encoding Class

```
function parseEncodingObject(encodName, encodingObject,
   componentSchemas)
  - Create Encoding Individual
  - Assign "encodName" value from OpenAPI Service to "encodName"
    property of Encoding Individual
  - Extract and map Encoding's properties from OpenAPI Service
  - Create a list for Header Individuals
  - For each Header Object in Encoding Object :
    - Call Header function and return Header Individual
    - Add the Header Individual to the list
  - Assign the list to the "encodingHeader" property of Encoding
    Individual
  - Return Encoding Individual
```

Listing 4.14: Encoding Object to Encoding Individual

4.12 Mapping of Response Object to Response Class

As we may see in table 4.16 Response Class has some subclasses. Their main purpose is to describe responses of grouped status codes. Property *statusCode* is a string used to define the status of a response (i.e. successful/

unsuccessful).

Response Object		*OpenAPI Response Class (DefaultResponse, 1xxResponse, 2xxResponse, 3xxResponse, 4xxResponse, 5xxResponse)*	
Field Name	**Type**	**Property**	**Range**
		openapi:statusCode	xsd:String
description	string	openapi:description	xsd:String
headers	Map[string, Header Object]	openapi: responseHeader	openapi:Header
content	Map[string, Media Type Object]	openapi:content	openapi:MediaType

Table 4.13: Response Object to Response Class

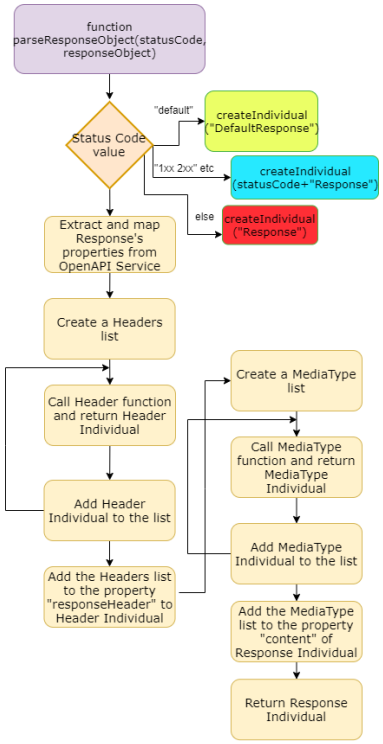

Figure 4.8: Workflow of Response Object to Response Individual mapping

```
function parseResponseObject(statusCode, responseObject,
    componentSchemas)
```

```
  - Depending on the "statusCode", the corresponding Individual is
    created
  - Extract and map Response's properties from OpenAPI Service
  - Create a Headers list
  - For each Header Object :
    - Call Header function and return Header Individual
    - Add Header Individual to the list
  - Add the Headers list to the property "responseHeader" to
    Header Individual
  - Create a MediaType list
  - For each MediaType Object :
    - Call MediaType function and return MediaType Individual
    - Add MediaType Individual to the list
  - Add the MediaType list to the property "content" of Response
    Individual
  - Return Response Individual
```

Listing 4.15: Response Object to Response Individual

4.13 Mapping of Tag Object to Tag Class

Listing 4.16 illustrates the mapping of a Tag Object to a Tag Individual, while 4.17 shows the implementation of the *x-onResource* extension property.

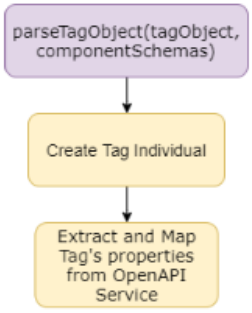

Figure 4.9: Workflow of Tag Object to Tag Individual mapping

```
function parseTagObject(tagObject, componentSchemas)
  - Create Tag Individual
  - Extract and Map Tag's properties from OpenAPI Service
```

Listing 4.16: Tag Object to Tag Individual

```
- Is implemented in parseTagObject
- Get all the extensions with name 'x-onResource' from Tag Object
- If x-onResource points to a Schema
    - Find that Schema in componentSchemas
    - Create the Shape Individual by calling the
    createNode/PropertyShape function
```

Tag Object		*OpenAPI Tag Class*	
Field Name	**Type**	**Property**	**Range**
name	string	openapi:name	openapi:String
description	string	openapi:description	openapi:String
externalDoc	External Doc Object	openapi:externalDoc	openapi:ExternalDoc

Table 4.14: Tag Object to Tag Class

```
– Save <Tag Individual, Shape Individual> pair
```

Listing 4.17: x-onResource implementation

4.14 Mapping of Schema Object to Shape Class

Schema Object		*OpenAPI Shape Class*	
Field Name	**Type**	**Property**	**Range**
multipleOf	string	openapi:multipleOf	xsd:Integer
readOnly	boolean	openapi:readOnly	xsd:Boolean
maxProperties	integer	openapi: maxProperties	xsd:Integer
minProperties	integer	openapi: minProperties	xsd:Integer
writeOnly	boolean	openapi:writeOnly	xsd:Boolean
xml	Xml Object	openapi:xml	openapi:Xml
externalDocs	External Documentation Object	openapi: externalDoc	openapi:ExternalDoc
deprecated	boolean	openapi: deprecated	xsd:Boolean

Table 4.15: Schema Object to Shape Class

In the implementation of the mapping of a Schema Object to a Shape Individual are also implementated the *x-refersTo*, *x-kindOf*, *x-mapsTo* and *x-collectionOn* ex-

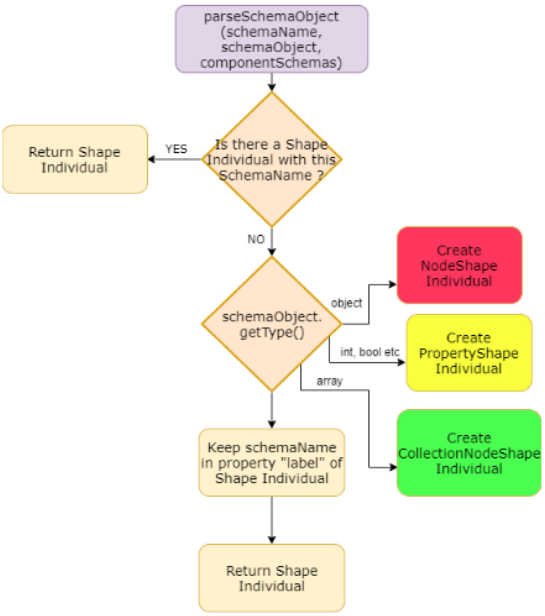

Figure 4.10: Workflow of Schema Object to Shape Individual mapping

tension properties which can be seen in listings 4.19, 4.20, 4.21, 4.22.

```
function parseSchemaObject(schemaName, schemaObject,
    componentSchemas)
  - If there is a Shape Individual with this schemaName, return it
  - Else get the Type of Schema Object :
    - If(getType==object) : Create NodeShape Individual
    - Else if(getType==int,bool etc) : Create PropertyShape
    Individual
    - Else if(getType==array) : Create CollectionNodeShape
    Individual
  - Keep schema name in property "label" of Shape Individual
  - Return Shape Individual
```

Listing 4.18: Schema Object to Shape Individual

```
- Is part of createNodeShape(..)/createPropertyShape(..)
- Get the resource where x-refersTo points
- Save the resource as target class/property of the new Shape
    individual
```

Listing 4.19: x-RefersTo implementation

```
- Implemented in createNodeShape(..)/createPropertyShape(..)
- Get the resource where x-kindOf points
- Create a new Class/property with name "schemaName"
- Make the new class subclass/subproperty of the class-resource
      where x-kindOf points
```

```
- Save the new class/property as target class/property of the new
      Shape individual
```

Listing 4.20: x-kindOf implementation

```
- Is part of createNodeShape(..)/createPropertyShape(..)
- Get the resource where x-mapsTo points
- Find Node/PropertyShape Individual with same name as the
   resource
- Copy semantics of the resource Individual
- Paste semantics in Node/PropertyShape Individual
```

Listing 4.21: x-mapsTo implementation

```
- Is implemented in createNodeShape(..)
- Get the model where x-collectionOn points
- Create new ontology class with name "schemaName"
- Set new class subclass of "openapi:Collection" class
- Save new class as target Class of the new NodeShape individual
```

Listing 4.22: x-collectionOn implementation

4.15 Mapping of XML Object to XML Class

XML Object		*OpenAPI XML Class*	
Field Name	**Type**	**Property**	**Range**
name	string	openapi:xmlName	xsd:String
namespace	string	openapi:namespace	xsd:String
prefix	string	openapi:prefix	xsd:String
attribute	boolean	openapi:attribute	xsd:Boolean
wrapped	boolean	openapi:wrapped	xsd:Boolean

Table 4.16: XML Object to XML Class

```
function parseXMLObject(xmlObject)
    - Create XML Individual
    - Extract and map XML's properties from OpenAPI Service
    - Return XML Individual
```

Listing 4.23: XML Object to XML Individual

4.16 Mapping of Security Scheme Object to Security Class

As seen in table 4.17 property *type* doesn't have a correspondence to a property in our ontology. OpenAPI 3.0 supports four Security Schemes "apiKey", "http", "oauth2", "openIdConnect" which in the OpenAPI ontology are represented as subclasses of Security Class. This may be seen in the tables 4.18, 4.19, 4.20 and 4.21.

Security Scheme Object		*OpenAPI Security Class*	
Field Name	**Type**	**Property**	**Range**
description	string	openapi:string	xsd:String
type	string		

Table 4.17: Security Scheme Object to Security Class

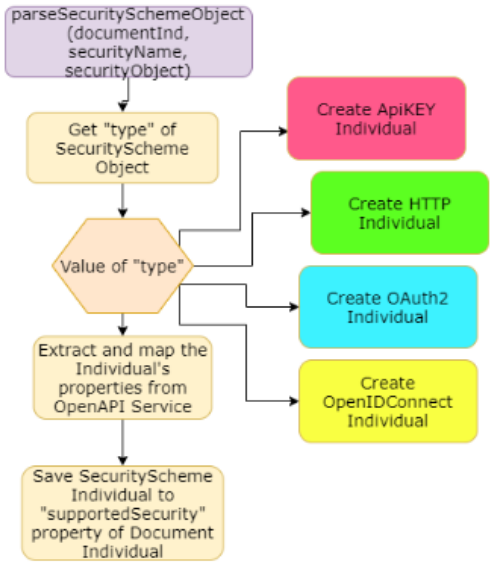

Figure 4.11: Workflow of SecurityScheme Object to Security Individual mapping

```
function parseSecuritySchemeObject(documentInd, securityName,
    securityObject)
  - Get "type" of SecurityScheme Object
  - Depending on the type, create the corresponding Individual (
    ApiKEY, HTTP, OAuth2, OpenIDConnect)
```

```
- Extract and map the Individual's properties from OpenAPI
  Service
- Save SecurityScheme Individual to "supportedSecurity" property
  of Document Individual
```

Listing 4.24: SecurityScheme Object to SecurityScheme Individual

Security Scheme Object		*OpenAPI ApiKey Class*	
Field Name	**Type**	**Property**	**Range**
name	string	openapi:parameterName	xsd:String
in	string	openapi:in	xsd:String

Table 4.18: ApiKey Class

Security Scheme Object		*OpenAPI Http Class*	
Field Name	**Type**	**Property**	**Range**
scheme	string	openapi:scheme	xsd:String
bearerFormat	string	openapi:bearerFormat	xsd:String

Table 4.19: Http Class

Security Scheme Object		*OpenAPI OpenIdConnect Class*	
Field Name	**Type**	**Property**	**Range**
openIdConnectUrl	string	openapi:openIdConnectUrl	xsd:String

Table 4.20: OpenIdConnect Class

For the ApiKey, Http, OpenIdConnect Classes all we need to do is create an Individual as seen in listing 4.24.

Security Scheme Object		*OpenAPI OAuth2 Class*	
Field Name	**Type**	**Property**	**Range**
flows	OAuthFlowsObject	openapi:flow	openapi:OAuthFlow

Table 4.21: OAuth2 Class

4.17 Mapping of Security Requirement Object to Security Requirement Class

Finally in table 4.22 we may see the Security Requirement Class in our ontology which defines the Security Scheme used and the permitted Scope.

OpenAPI SecurityRequirement Class	
Property	**Range**
openapi:securityType	openapi:Security
openapi:scope	openapi:Scope

Table 4.22: Security Requirement Class

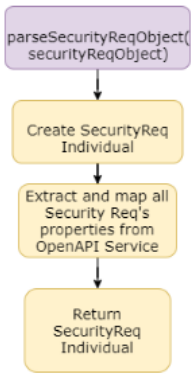

Figure 4.12: Workflow of SecurityRequirement Object to SecurityRequirement Individual mapping

```
function parseSecurityReqObject(securityReqObject)
  - Create SecurityReq Individual
  - Extract and map all Security Req's properties from OpenAPI
    Service
  - Return SecurityReq Individual
```

Listing 4.25: Security Req Object to Security Req Individual

Chapter 5

Implementation

We have at this point described the abstract algorithm that shows the mappings between OpenAPI Objects and Classes in OpenAPI version 3.0 Ontology. In this chapter, we proceed with the full implementation containing all the technical details of our algorithm for whom is interested in.

The *Document* Class is the most important one in the Ontology. Figure 3.1 represents the OpenAPI Object that maps to the Document Class in our ontology. What convert2ontology method does is call all the other methods that proceed the mapping of each Object to an ontology Individual (Info, Server, SecuritySchemes, SecurityRequirement, Tag, ExternalDocs, Operations.)

In addition, this method creates a list of all the Objects that must be kept as global (Servers, SecurityRequirement and Tags). More specifically, servers may be defined globally in the *Document* level, but may also be defined in *Path* and in *Operation*. Moreover, in this method we create the Document Individual.

```
procedure convert2ontology(openApiDoc) {
  // initialize the open api ontology
  Ontology ontModel = initOntologyModel();
  // create an individual for Document Class
  Individual documentInd = createIndividual(ontModel, "openapi:
```

```
  Document");
// info object into ontology
parseInfoObject(ontModel, documentInd, openApiDoc.getInfo());
// parse global servers and crate individuals
List<Individual> globServerIndList;
for (ServerObject serverObject in openApiDoc.getServers()) {
  Individual serverInd = parseServerObject(ontModel,
 serverObject);
  globServerIndList.add(serverInd);
}
// parse all security schemes
for (Entry<String, SecuritySchemeObject> secSchemeEntry in
 openApiDoc.getComponents().getSecuritySchemes()) {
  parseSecuritySchemeObject(ontModel, secSchemeEntry.getKey(),
 securityObject.getValue());
}
// parse global security requirements and parse individuals
List<Individual> globSecReqIndList;
for (SecurityRequirementObject securityReqObject in openApiDoc.
 getSecurityRequirements()) {
  Individual securityReqInd = parseSecurityReqObject(ontModel,
 securityReqObject);
  globSecReqIndList.add(securityReqInd);
}
//parse Tag
Map<String, Individual> tagShapeMap;
for (TagObject tag in openApiDoc.getTags()) {
   Entry<String, Individual> tagShapeEntry = parseTagObject(
 ontModel, tag, openApiDoc.getComponents().getSchemas());
   documentInd.setProperty("openapi:supportedEntity",
 tagShapeEntry.getValue());
   tagShapeMap.add(tagShapeEntry);
}
//External Docs in Document level
Individual exDocInd=parseExternalDocObject(ontModel,openApiDoc.
 getExternalDoc());
operationInd.setProperty("openapi:externalDoc", exDocInd);
// parse path objects into ontology
for (Entry <String, PathItemObject> pathEntry in openApiDoc.
 getPaths()) {
  //get Path Servers
  List<Individual> pathServerIndList;
  for (ServerObject serverObject in pathEntry.getValue().
 getServers()) {
    Individual serverInd = parseServerObject(ontModel,
 serverObject);
    pathServerIndList.add(serverInd);
  }
  //If there no path servers, keep global servers
  List<Individual> operationServerList;
  if(pathServerIndList.isEmpty()){
    operationServerList = globServerIndList;
  }
  else {
    operationServerList = pathServerIndList;
  }
  //Get path parameters
  List<Individual> pathParameterIndList;
```

```
    for (ParameterOject parameterObject in pathEntry.getValue().
   getParameter()) {
      Individual parameterInd = parsePathParameterObject(ontModel,
    parameterObject, openApiDoc.getComponents().getSchemas());
      pathParameterIndList.add(parameterInd);
    }
    //Parse all operation types (Get,Post,..,Delete)
    for (OperationObject operationObject in pathEntry.getValue().
   getOperations()){
      //path
      Individual pathInd=parsePathObject(ontModel, pathName)
      parseOperationObject(ontModel, documentInd, pathInd,
   operationObject, tagShapeMap, openApiDoc.getComponents().
   getSchemas(), pathParameterIndList,operationServerList,
   globSecReqIndList);
    }
  }
```

Listing 5.1: Initialization and Document Method

If there are not any servers in Operation, then we use the ones in Path and if we don't have any Path servers either, then we use the global ones. The same thing may happen with the servers defined in SecurityRequirement. This way we are able to override the global servers, by defining new or we may stick with the global ones. Then, we also get all the parameters defined in Path level and we call the method *parsePathObject()* which returns a Path Individual. In addition, for every Path we get the Operations under it and we call *parseOperationObject()* which will be explained in the following section along with what happens with globally defined Tags and those defined in Operation.

Finally, the format is almost the same in this method for every Object. As we can see, we get the Object from the OpenAPI service, then we call the method that parses the Object to an Individual of the corresponding class in the ontology and then we add the Individual to the global list (when there is one). The only difference is when we have Tags. Then, we create a Map of a <String, Individual>. The string contains the name of the Tag, while

the Individual is the Shape Individual that exists only in case that a Tag has been connected with a Schema by using the *x-onResource* property. Then, we put the Shape Individual in the *supportedEntity* property of the Document. Every entry is saved in the Map we said before.

5.1 ParseOperationObject Method

This method starts with initializing a new Operation Individual, which is later going to be filled with all the appropriate properties of the Operation Object. The extension property x-operationType is implemented here. What is done is get the Operation Individual and add a superclass. The superclass is the class where x-operationType points. For example, if we had a */pet/findByStatus* command then the superclass would be the class SearchAction of Schema.org.

Then, some properties of Operation Object are mapped to the Operation Individual, having in mind the table 4.7. When a property is datatype and not Object, the mapping is trivial while when it is Object we call complementary methods where the parse of the corresponding Object to Individual is done and the Individual is returned.

```
method void parseOperationObject(OntModel ontModel, Individual
    documentInd, Individual pathInd, OperationObject
    operationObject, Map<String,Individual> tagShapeMap, Map<String
    , SchemaObject> schemas, List<ParameterObject>
    parametersListFromPath, List<Individual> serverList, List<
    Individual> globSecReqIndList) {
  operationInd = createIndividual(ontModel, "openapi:Operation");
  //x-operationType
  String operationType = operationObject.getExtensions("x-
    operationType");
  if(operationType != null){
    operationInd.addOntClass(operationType);
  }
  operationInd.setProperty("openapi:name", operationObject.
    getOperationId());
```

```
operationInd.setProperty("openapi:onPath", pathInd);
operationInd.setProperty("openapi:deprecated", openapi.
 getDeprecated());
operationInd.setProperty("openapi:summary", openapi.getSummary()
 );
operationInd.setProperty("openapi:description", openapi.
 getDescription());
//Type of Operation
Individual methodInd=getMethodIndividual(ontModel,
 operationObject.getOperationType());
operationInd.setProperty("openapi:method", methodInd);
//ex
if(operationObject.getExternalDoc()!=null){
  Individual exDocInd=parseExternalDocObject(ontModel,
 operationObject.getExternalDoc());
  operationInd.setProperty("openapi:externalDoc", exDocInd);
}
//tags
for (String tag in operationObject.getTags()) {
  Individual tagInd;
  Entry<String, Individual> tagShapeEntry = tagShapeMap.getEntry
 (tag);
  if (tagShapeEntry == null) {
    tagInd = createIndividual(ontModel, "openapi:Tag");
    tagInd.setProperty("openapi:name", tag);
  }
  else {
    tagInd = findTagIndividual(ontModel, tag);
    Individual shape = tagShapeEntry.getValue();
    if(shape!=null){
       shape.setProperty("openapi:supportedOperation",
 operationInd);
     }
  }
  operationInd.setProperty("openapi:tag", tagInd);
}
//RequestBody
if(openapi.getRequestBody()!=null){
  Individual requestInd = parseRequestBodyObject(ontModel,
 operationObject.getRequestBody(),schemas);
  operationInd.setProperty("openapi:requestBody", requestInd);
}
//Response
for(Entry<String, ResponseObject> responseEntry in
 operationObject.getResponses()){
  Individual responseInd=parseResponseObject(ontModel,
 responseEntry.getKey(), responseEntry.getValue(),schemas);
  operationInd.setProperty("openapi:response",responseInd));
}
//Security Req
if(operationObject.getSecurityRequirements().isEmpty()){
  for (Individual securiryReqInd in globSecReqIndList) {
    operationInd.setProperty("openapi:security", securiryReqInd)
 ;
  }
}
else{
  for (SecurityRequirementObject securiryReqObject in
```

```
    operationObject.getSecurityRequirements()) {
      Individual securityReqInd = parseSecurityReqObject(ontModel,
    securiryReqObject);
      operationInd.setProperty("openapi:security", securityReqInd)
    ;
     }
  }
  //parameters
  List<ParameterObject> combinedParametersList= combineParameters(
    parametersListFromPath,operationObject.getParameters());
  for (ParameterObject parameterObject in combinedParametersList){
    switch(parameterObject.getIn())
    {
      case "path":
        Individual parameterInd=parsePathParameterObject(ontModel,
    parameterObject,schemas);
        operationInd.setProperty("openapi:parameter",parameterInd)
    ;
      case "query":
        Individual queryInd=parseQueryObject(ontModel,
    parameterObject,schemas);
        operationInd.setProperty("openapi:query",queryInd);
      case "header":
        headerName=parameterObject.getName();
        Individual headerInd=parseHeaderObject(ontModel,headerName
    ,parameterObject,schemas);
        operationInd.setProperty("openapi:requestHeader",headerInd
    );
      case "cookie":
        Individual cookieInd=parseCookie(ontModel,parameterObject,
    schemas);
        operationInd.setProperty("openapi:cookie",cookieInd);
     }
  }

  //Servers
  if(operationObject.getServers().isEmpty()){
    for (Indinidual serverInd in finalServerList) {
      operationInd.setProperty("openapi:serverInfo", serverInd);
    }
  }
  else{
    for (ServerObject serverObject in operationObject.getServers()
    ) {
      Individual serverInd = parseServerObject(ontModel,
    serverObject);
      operationInd.setProperty("openapi:serverInfo", serverInd);
     }
  }
  documentInd.setProperty("openapi:supportedOperation",
    operationInd);
}
```

Listing 5.2: Operation Method

Later we have the parsing of Tags inside Operation. What happens with Tags is the following : We have the

globally defined Tags but we may also have some Tags defined in Operation level. The ones defined in Operation level are in the form of strings and not Tag Objects. When a Tag is defined in Operation, but has not been globally defined (this might happen) then what we do is create a Tag Individual and assign to it the name of the tag. On the other hand, when there is a Tag defined in Operation that has been already predefined globally then we find it in the global list, we get the Shape (which exists in case we have used the x-onResource property) and make it point to the Operation Individual by using the property *supportedOperation*.

In addition, Request Body and Response are Objects which, as mentioned above, are returned as mapped Individuals after calling the right method that implements the parsing. Moreover, what happens with Security Requirement is that we check if there are not any Security Requirement Objects in Operation level and in this case we keep the global ones. However in case there are Security Requirement Objects defined in Operation, then for each one the method *parseSecurityReqObject()* is called. The returned Security Requirement Individual is saved in the property security under Operation.

In order to implement the map of the Parameter Object we call the method *combineParameters* sending as parameters the ones from Path and the ones from Operation. Parameters existed on Path are always used. When there are additional Parameters defined on Operation, then we use both the ones from Path and the ones from Operation. There is however a case where a Parameter on Path may be overrided on Operation level. This happens only when it is declared using the same *name* and

in type. In that case, the Operation Parameter is kept. An operation according to its *in type* may be inside a *Query*, a *Header*, a *Cookie* or a *Path*. For each of these cases a different parse method is called.

The same thing happens with Server Objects. If there are not Server Objects defined in Operation level, the global ones are used. Otherwise for each Server Object the method *parseServerObject* is called and the returned Individuals are saved in property *serverInfo* of Operation. Finally, Document Individual keeps all the Operations that have been created in property *supportedOperation*.

5.2 CombineParameters Method

This method is the one mentioned before in Operation. A list is declared and then we check if and which Parameters are overridden in Operation level. In this case we keep these Parameters in the combined list. In the case where there is not a Parameter overridden, we keep in the list the Parameters from Path. Finally, we make a check for all the Parameters declared in Operation level and for the ones that do not override a Path Parameter (since they haven't been added to the list yet), we put them too in the combined list.

```
method List<ParameterObject> combineParameters(List<
    ParameterObject> fromPath, List<ParameterObject> fromOperation)
{
  List<ParameterObject> combinedList;
  //Specific path-level parameters can be overridden
  for(ParameterObject paramFromPath in fromPath){
      bool override=false;
      for(ParameterObject paramFromOperation in fromOperation){
         if(paramFromPath.getName()==paramFromOperation.getName())
            if(paramFromPath.getIn()==paramFromOperation.getIn()){
              combinedList.add(paramFromOperation);
```

```
                override=true;
                break;
              }
        }
        if(override==false)
           combinedList.add(paramFromPath);
  }
  /*Any extra parameters defined at the operation level
  * are used together with path-level parameters*/
  for(ParameterObject paramFromOperation in fromOperation){
    if(paramFromOperation not in combinedList)
      combinedList.add(paramFromOperation);
  }
  return combinedList;
}
```

Listing 5.3: combineParameters Method

5.3 ParsePathObject Method

This is a simple method where we create an Individual of Path Class and put in its property *name* the value that is returned from the OpenAPI service.

```
method Individual parsePathObject(OntModel ontModel, String name)
    {
  Indinidual pathInd = createIndividual(ontModel, "openapi:Path");
  pathInd.setProperty("openapi:pathName", name);
  return pathInd;
}
```

Listing 5.4: Path Method

5.4 ParseInfoObject Method

In this method the mapping of an Info Object to Info Individual is implemented. The datatype properties of the OpenAPI service are corresponded to the ones in our ontology by using the *setProperty* command. For the Contact and License Object that are object properties of the Info Object, a Contact and License Individual are created and

then passed to the corresponding property of Info Individual.

```
method void parseInfoObject(OntModel ontModel, Individual
    documentInd, InfoObject infoObject) {
  Individual infoInd = createIndividual(ontModel, "openapi:Info");
  infoInd.setProperty("openapi:serviceTitle", infoObject.getTitle
    ());
  infoInd.setProperty("openapi:description", infoObject.
    getDescription());
  infoInd.setProperty("openapi:termsOfService", infoObject.
    getTermsOfService());
  //Contact Object
  if (infoObject.getContact() != null) {
    Individual contactInd = createIndividual(ontModel, "openapi:
    Contact");
    contactInd.setProperty("openapi:contactName", infoObject.
    getContact().getName());
    contactInd.setProperty("openapi:url", infoObject.getContact().
    getUrl());
    contactInd.setProperty("openapi:email", infoObject.getContact
    ().getEmail());
    infoInd.setProperty("openapi:contact", contactInd);
  }
  //License object
  if (infoObject.getLicense() != null) {
    Individual licenseInd = createIndividual(ontModel, "openapi:
    License");
    licenseInd.setProperty("openapi:licenseName", infoObject.
    getLicense().getName());
    licenseInd.setProperty("openapi:url", infoObject.getLicense().
    getUrl());
    infoInd.setProperty("openapi:license", licenseInd);
  }
  documentInd.setProperty("openapi:info", infoInd);
  documentInd.setProperty("openapi:version", infoObject.getVersion
    ());

}
```

Listing 5.5: Info Method

5.5 ParseServerObject Method

In this method the Server Object is parsed to a Server Individual. An individual of Server Class is created and then it is filled with its properties in which we have added the value that we got from the OpenAPI service. For the property *variables* that returns a Server Variable Object,

which is a map of its name and the Object we need to create a ServerVariable Individual. After the Server Variable Individual has been created and filled with the correct properties it is passed back to the Server Individual.

```
method Individual parseServerObject(OntModel ontModel,
    ServerObject serverObject){
  Individual serverInd = createIndividual(ontModel, "openapi:
    Server");
  serverInd.setProperty("openapi:host", serverObject.getUrl());
  serverInd.setProperty("openapi:description", serverObject.
    getDescription());
  //ServerVariable object
  for (Entry <String, ServerVariableObject> serverVariableEntry in
     serverObject.getServerVariables()){
    Individual serverVariableInd = createIndividual(ontModel, "
    openapi:ServerVariable");
    ServerVariableObject sv=serverVariableEntry.getValue();
    serverVariableInd.setProperty("openapi:name",
    serverVariableEntry.getKey());
    for (String enum in sv.getEnum()){
    serverVariableInd.setProperty("openapi:variableValue",enum);
    }
    serverVariableInd.setProperty("openapi:variableDefaultValue",
    sv.getDefault());
    serverVariableInd.setProperty("openapi:description", sv.
    getDescription());
    serverInd.setProperty("openapi:variables",serverVariableInd);
  }
  return serverInd;
}
```

Listing 5.6: Server Method

5.6 ParseExternalDocObject Method

In this method the External Documentation Object is mapped to the External Documentation Individual as seen in table 4.8.

```
method Individual parseExternalDocObject(OntModel ontModel,
    ExternalDocObject exdoc) {
  Individual exdocInd = createIndividual(ontModel, "openapi:
    ExternalDoc");
  exdoc.setProperty("openapi:url", exdoc.getUrl());
  exdoc.setProperty("openapi:description", exdoc.getDescription())
    ;
```

```
  return exdocInd;
}
```

Listing 5.7: External Doc Method

5.7 ParseXMLObject Method

In this method the XML Object is mapped to the XML Individual as seen in table 4.16.

```
method Individual parseXmlObject(OntModel ontModel,XmlObject xml)
    {
  Individual xmlInd = createIndividual(ontModel, "openapi:XML");
  xmlInd.setProperty("openapi:name", xml.getName());
  xmlInd.setProperty("openapi:namespace", xml.getNameSpace());
  xmlInd.setProperty("openapi:prefix", xml.getPrefix());
  xmlInd.setProperty("openapi:attribute", xml.getAttribute());
  xmlInd.setProperty("openapi:wrapped", xml.getWrapped());
  return xmlInd;
}
```

Listing 5.8: XML Method

5.8 GetMethodIndividual Method

In the OpenAPI ontology there are predefined Individuals of Class Method. This is why we keep the URI (name) of each one of the OpenAPI Methods (put, post, head etc) and then we get the associating Individual from the ontology.

```
method Individual getMethodIndividual(OntModel ontModel, String
    method){
  switch(method)
  {
    case PUT :
      method_uri= "openapi:PUT";
    case POST:
      method_uri= "openapi:POST";
    case HEAD:
      method_uri="openapi:HEAD";
    case PATCH:
      method_uri= "openapi:PATCH";
    case OPTIONS:
```

```
      method_uri ="openapi:OPTIONS";
    case GET:
      method_uri= "openapi:GET";
    case DELETE:
      method_uri= "openapi:DELETE";
    case TRACE:
      method_uri= "openapi:TRACE";
  }
  return ontModel.getIndividual(method_uri);
}
```

Listing 5.9: Method

5.9 GetStyleIndividual Method

Similarly to Method, in the OpenAPI ontology there are predefined Individuals of Class Style. This is why we keep the URI (name) of each one of the OpenAPI Styles (form, matrix etc) and then we get the associating Individual from the ontology.

```
method Individual getStyleIndividual(OntModel ontModel, StyleEnum
    style){
  switch(style)
  {
    case FORM :
      style_uri= "openapi:form";
    case MATRIX:
      style_uri= "openapi:matrix";
    case DEEPOBJECT:
      style_uri="openapi:deepObject";
    case LABEL:
      style_uri= "openapi:label";
    case SIMPLE:
      style_uri ="openapi:simple";
    case SPACEDELIMITED:
      style_uri= "openapi:spaceDelimited";
    case PIPEDELIMITED:
      style_uri= "openapi:pipeDelimited";
    }
    return ontModel.getIndividual(style_uri);
```

Listing 5.10: Style Method

5.10 ParseMediaTypeObject Method

In this method the Media Type Object is parsed to Media-Type Individual and its properties. For two of the properties, *encoding* and *schema* we call the corresponding parsing methods in order to return the Encoding and Schema Individuals.

```
method Individual parseMediaTypeObject(OntModel ontModel, String
    mediaTypeName, MediaTypeObject mediaTypeObject, Map<String,
    SchemaObject> schemas){
  Individual mediatypeInd = createIndividual(ontModel, "openapi:
    MediaType");
  mediatypeInd.setProperty("openapi:mediaTypeName", mediaTypeName)
    ;
  for(Entry<String, EncodingObject> encodingEntry in
    mediaTypeObject.getEncoding()){
    Individual encodingInd = parseEncodingObject(ontModel,
    encodingEntry.getKey(), encodingEntry.getValue(),schemas);
    mediatypeInd.setProperty("openapi:encoding", encodingInd);
  }
  if(mediaTypeObject.getSchema()!=null){
    Individual schemaInd=parseSchemaObject(ontModel,null,
    mediaTypeObject.getSchema(),schemas);
    mediatypeInd.setProperty("openapi:schema", schemaInd);
  }
  return mediatypeInd;
}
```

Listing 5.11: Media Type Method

5.11 ParseEncodingObject Method

This method implements the parsing of an Encoding Object to an Encoding Individual. For the property *encodingHeader* the method *parseHeaderObject* is called in order to return the Header Individual needed for this property.

```
method Individual parseEncodingObject(OntModel ontModel, String
    encodingName, EncodingObject encodingObject,Map<String,
    SchemaObject> schemas){
  Individual encodingInd = createIndividual(ontModel, "openapi:
    Encoding");
```

```
  encodingInd.setProperty("openapi:propertyName", encodingName);
  encodingInd.setProperty("openapi:contentType", encodingObject.
    getContentType());
  encodingInd.setProperty("openapi:explode", encodingObject.
    getExplode());
  encodingInd.setProperty("openapi:allowReserved", encodingObject.
    getAllowReserved());
  Individual styleInd=getStyleIndividual(ontmodel,encodingObject.
    getStyle());
  encodingInd.setProperty("openapi:style", styleInd);
  for(Entry<String,HeaderObject> headerEntry in encodingObject.
    getHeaders()){
     headerInd=parseHeaderObject(ontModel, headerEntry.getKey(),
    headerEntry.getValue(),schemas);
     encodingInd.setProperty("openapi:encodingHeader", headerInd);
  }
  return encodingInd;
}
```

Listing 5.12: Encoding Method

5.12 ParseHeaderObject, ParseCookieObject, ParseQueryObject, ParsePathParameterObject Methods

Header, Cookie Query and PathParameter Objects are part of Parameter Object. For properties *schema* and *content* the methods *parseSchemaObject* and *parseMediaTypeObject* are called to return the associating Individuals.

```
method Individual parseHeaderObject(OntModel ontModel, String
    headerName, HeaderObject headerObject, Map<String, SchemaObject
    > schemas) {
  Individual headerInd = createIndividual(ontModel, "openapi:
    Header");
  headerInd.setProperty("openapi:name", headerName);
  headerInd.setProperty("openapi:description", headerObject.
    getDescription());
  headerInd.setProperty("openapi:required", headerObject.
    getRequired());
  headerInd.setProperty("openapi:deprecated", headerObject.
    getDeprecated());
  Individual styleInd=getStyleIndividual(ontmodel, headerObject.
    getStyle());
  headerInd.setProperty("openapi:style", styleInd);
  headerInd.setProperty("openapi:explode", headerObject.getExplode
    ());
```

```
  if(headerObject.getSchema()!=null){
    Individual schemaInd=parseSchemaObject(ontModel,null,
    headerObject.getSchema(),schemas);
    headerInd.setProperty("openapi:schema", schemaInd);
  }
  for(Entry<String,MediaTypeObject> mediaTypeEntry in headerObject
    .getContent()){
    Individual mediatypeInd=parseMediaTypeObject(ontModel,
    mediaTypeEntry.getKey(),mediaTypeEntry.getValue(),schemas);
    headerInd.setProperty("openapi:content", mediatypeInd);
  }
  return headerInd;
}
```

Listing 5.13: Header Method

```
method Individual parseCookie(OntModel ontModel, ParameterObject
    parameterObject, Map<String, SchemaObject> schemas) {
  Individual cookieInd = createIndividual(ontModel, "openapi:
    Cookie");
  cookieInd.setProperty("openapi:name", parameterObject.getName())
    ;
  cookieInd.setProperty("openapi:description", parameterObject.
    getDescription());
  cookieInd.setProperty("openapi:required", parameterObject.
    getRequired());
  cookieInd.setProperty("openapi:deprecated", parameterObject.
    getDeprecated());
  Individual styleInd=getStyleIndividual(ontmodel,parameterObject
    .getStyle());
  cookieInd.setProperty("openapi:style", styleInd);
  cookieObject.setProperty("openapi:explode", parameterObject.
    getExplode());
  if(cookieObject.getSchema()!=null){
    Individual schemaInd=parseSchemaObject(ontModel,null,
    cookieObject.getSchema(),schemas);
    cookieInd.setProperty("openapi:schema", schemaInd);
  }
  for(Entry<String,MediaTypeObject> mediaTypeEntry in
    parameterObject.getContent()){
    Individual mediatypeInd=parseMediaTypeObject(ontModel,
    mediaTypeEntry.getKey(),mediaTypeObject.getValue(),schemas);
    cookieInd.setProperty("openapi:content", mediatypeInd);
  }
  return cookieInd;
}
```

Listing 5.14: Cookie Method

```
method Individual parseQueryObjectParameter(OntModel ontModel,
    ParameterObject queryObject, Map<String, SchemaObject> schemas)
    {
  Individual queryInd = createIndividual(ontModel, "openapi:Query
    ");
  queryInd.setProperty("openapi:name", queryObject.getName());
  queryInd.setProperty("openapi:description", queryObject.
    getDescription());
  queryInd.setProperty("openapi:required", queryObject.getRequired
    ());
```

```
  queryInd.setProperty("openapi:deprecated", queryObject.
    getDeprecated());
  Individual styleInd=getStyleIndividual(ontmodel,queryObject .
    getStyle());
  queryInd.setProperty("openapi:style", styleInd);
  queryInd.setProperty("openapi:explode", queryObject.getExplode()
    );
  queryInd.setProperty("openapi:allowEmptyValue", queryObject.
    getAllowEmptyValue());
  queryInd.setProperty("openapi:allowReserved", queryObject.
    getAllowReserved());
  if(queryObject.getSchema()!=null){
    Individual schemaInd=parseSchemaObject(ontModel,null,
    queryObject.getSchema(),schemas);
    queryInd.setProperty("openapi:schema", schemaInd);
  }
  for(Entry<String,MediaTypeObject> mediaTypeEntry in queryObject.
    getContent()){
    Individual mediatypeInd=parseMediaTypeObject(ontModel,
    mediaTypeEntry.getKey(),mediaTypeEntry.getValue(),schemas);
    queryInd.setProperty("openapi:content", mediatypeInd);
  }
  return queryInd;
}
```

Listing 5.15: Query Method

```
method Individual parsePathParameterObject(ontModel,
    parameterObject, Map<String, SchemaObject> schemas) {
  Indinidual parameterInd = createIndividual(ontModel, "openapi:
    Parameter", parameterObject.getName());
  parameterInd.setProperty("openapi:name", parameterObject.getName
    ());
  parameterInd.setProperty("openapi:description", parameterObject.
    getDescription());
  parameterInd.setProperty("openapi:required", true);
  parameterInd.setProperty("openapi:deprecated", parameterObject.
    getDeprecated());
  styleInd=getStyleIndividual(ontmodel,parameterObject.getStyle())
    ;
  parameterInd.setProperty("openapi:style", styleInd);
  parameterInd.setProperty("openapi:explode", parameterObject.
    getExplode());
  if(parameterObject.getSchema()!=null){
    Individual schemaInd=parseSchemaObject(ontModel,null,
    parameterObject.getSchema(),schemas);
    parameterInd.setProperty("openapi:schema", schemaInd);
  }
  for(Entry<mediaTypeName,mediaTypeObject> mediaTypeEntry in
    parameterObject.getContent()){
    mediatypeInd=parseMediaTypeObject(ontModel,mediaTypeEntry.
    getKey(), mediaTypeEntry.getValue(), schemas);
    parameterInd.setProperty("openapi:content", mediatypeInd);
  }
  return parameterInd;
}
```

Listing 5.16: Path Parameter Method

5.13 GetDatatype Method

The mapping between datatypes from OAS and datatypes in the ontology happens according to the table 5.1.

OpenAPI DataTypes		*OpenAPI Ontology DataTypes*
Type	**Format**	
integer	int32	xsd:int
integer	int64	xsd:long
number	float	xsd:float
number	double	xsd:double
string		xsd:string
string	byte (base64 -encoded file contents)	xsd:base64Binary
string	binary (binary file contents)	There isn't a binary datatype in xsd. Therefore, we need to introduce a new. openAPI:binary which is an array of xsd:Byte
boolean		xsd:boolean
string	date	xsd:date
string	date-time	xsd:dateTime

Table 5.1: Datatypes

```
method Resourse getDatatype(String type, String format){
  if(type=="integer" && format=="int32") return XSD.xint;
  else if (type=="integer" && format=="int64") return   XSD.xlong;
  else if (type=="number" && format=="float") return XSD.xfloat;
  else if (type=="number" && format=="double") return XSD.xdouble;
  else if (type=="string" && format=="byte") return  XSD.
   base64Binary;
  else if (type=="string" && format=="binary") return Resourse("
   openapi:binary");
  else if (type=="string" && format=="date") return XSD.date;
  else if (type=="string" && format=="date-time") return  XSD.
   dateTime;
  else if (type=="string" && format==null) return XSD.xstring;
  else if (type=="boolean" && format==null) return XSD.xboolean;
  else if (type=="integer" && format==null) return XSD.xint
  else if (type=="number"&& format==null) return XSD.decimal;
}
```

Listing 5.17: Datatype Method

5.14 ParseResponseObject Method

This method implements the mapping of a Response Object to Response Individual. Depending on the *statusCode*, an Individual of the corresponding Class (Default, 1xx, 2xx etc) is created. In order to give value to the properties *responseHeader* and *content* methods *parseHeaderObject* and *parseMediaTypeObject* are called.

```
method Individual parseResponseObject (OntModel ontModel, String
    statusCode, ResponseObject responseObject, Map<String,
    SchemaObject> schemas){
  if (statusCode == "default")
       Individual responseInd = createIndividual(ontModel, "
    openapi:DefaultResponse");
  else if (statusCode == "1XX")
       Individual responseInd = createIndividual(ontModel, "
    openapi:1xxResponse");
  else if (statusCode == "2XX")
       Individual responseInd = createIndividual(ontModel, "
    openapi:2xxResponse");
  else if (statusCode == "3XX")
       Individual responseInd = createIndividual(ontModel, "
    openapi:3xxResponse");
  else if (statusCode == "4XX")
       Individual responseInd = createIndividual(ontModel, "
    openapi:4xxResponse");
  else if (statusCode == "5XX")
       Individual responseInd = createIndividual(ontModel, "
    openapi:5xxResponse");
  else {
       Individual responseInd = createIndividual(ontModel, "
    openapi:Response");
       responseInd.setProperty("openapi:statusCode", statusCode);
  }
  responseInd.setProperty("openapi:description", responseObject.
    getDescription());
  for(Entry<String, HeaderObject> headerObjectEntry in
    responseObject.getHeaders()){
     Individual headerInd=parseHeaderObject(ontModel,
    headerObjectEntry.getKey(),headerObjectEntry.getValue(),
    schemas);
     responseInd.setProperty("openapi:responseHeader", headerInd);
  }
  for(Entry<String, MediaTypeObject> mediaTypeEntry in
    responseObject.getContent()){
     Individual mediaInd=parseMediaTypeIndividual(ontModel,
    mediaTypeEntry.getKey(),mediaTypeEntry.getValue(),schemas);
     responseInd.setProperty("openapi:content",mediaInd);
  }
  return responseInd;
```

```
}
```

Listing 5.18: Response Method

5.15 ParseRequestObject Method

This method implements the mapping of a Request Object to a Request Individual. In order to set the value of the *content* property, the method *parseMediaTypeObject* is called.

```
method Individual parseRequestBodyObject (OntModel ontModel,
    RequestObject requestObject, Map<String, SchemaObject> schema){
  Individual requestInd = createIndividual(ontModel, "openapi:
    RequestBody");
  requestInd.setProperty("openapi:description", requestObject.
    getDescription());
  requestInd.setProperty("openapi:required", requestObject.
    getRequired());
  for(Entry<String, mediaTypeObject> mediaTypeEntry in
    requestObject.getContent()){
     mediaInd=parseMediaTypeObject(ontModel,mediaTypeEntry.getKey()
    ,mediaTypeEntry.getValue(),schemas);
     requestInd.setProperty("openapi:content",mediaInd);
  }
  return requestInd;
}
```

Listing 5.19: Request Method

5.16 ParseTagObject Method

This method implements the parsing of a Tag Object to Tag Individual. It also supports the use of the x-onResource property which links a Tag with a Schema. The implementation of the extension property is based on the following procedure : First, we keep in a string the value where the x-onResource property points. For example we could have */components/schema/Pet*. After that, we extract from this string the Schema name, aka *Pet*. Finally,

the method *parseSchemaObject* is called in order to return the associating Individual which is returned from parseTagObject method along with the Tag name.

```
method Entry<String, Individual> parseTagObject(Ontmodel ontModel,
    Tag tag, Map<String,SchemaObject> schemas){
  // used for findTagIndividual(OntModel model, name_of_Tag) ->
   ontModel.getIndividual(name_of_Tag)
  Individual tagInd = createIndividual(ontModel, "openapi:Tag",
   tagObject.getName());
  tagInd.setProperty("openapi:name",tagObject.getName());
  tagInd.setProperty("openapi:description", tagObject.
   getDescription());
  Individual exdocInd=parseExternalDocIndividual(ontModel,
   tagObject.getExternalDoc());
  tagInd.setProperty("openapi:externalDoc",exdocInd);
  //x-onResourse
  Individual schemaInd=null;
  String xOnResourseExtension= tagObject.getExtension("x-
   onResourse");
  if(xOnResourseExtension!=null){
    // extractSchemaName(#/components/schema/Pet)->{Pet}
    schemaName= extractSchemaName(xOnResourseExtension);
    schemaInd=parseSchemaObject(ontModel,schemaName,schemas.get(
   schemaName),schemas);
  }
  return new Entry<String, Individual>(tag.getName(), schemaInd);
}
```

Listing 5.20: Tag Method

5.17 ParseSchemaObject Method

This method parses a Schema Object of an OpenAPI service to a a Shape Individual of the OpenAPI ontology. A Shape Individual may be a NodeShape, PropertyShape or CollectionNodeShape Individual. When the type of the Schema is Object then it maps to a NodeShape, when the type is array it maps to a CollectionNodeShape, while everything else (integer, string, boolean etc) is considered a PropertyShape. For each case a different method is called. Finally, the Shape Individual is returned.

```
method Individual parseSchemaObject(Ontology ontModel, String
    schemaName, SchemaObject schemaObject, Map<String, SchemaObject
    > schemas) {
Individual shapeInd = findShapeIndividual(ontModel, schemaName);
if (shapeInd == null) {
  if (schemaObject.getType() ==object) {
    shapeInd = createNodeShape(ontModel, schemaName, schemaObject,
    schemas);
  }
  else if (schemaObject.getType() ==array && !schemaObject.
    hasExtension()) {
    shapeInd =createCollectionNodeShape(ontModel, schemaName,
    schemaObject, schemas);
  }
  //integer, string, number ,boolean
  else {
    shapeInd = createPropertyShape(ontModel, schemaName,
    schemaObject, schemas);
  }
  shapeInd.setProperty("rdfs:label", schemaName + "Shape");
}
return shapeInd;
}
```

Listing 5.21: Schema Method

5.18 CreateNodeShape Method

In this method the NodeShape Individual is created and returned. In addition, this method supports also the x-refersTo, x-kindOf, x-mapsTo and x-collectionOn properties. In every case the Class where the property points is kept. In x-kindOf, a Class with the Schema name is created and kept in the variable *classUri*. Then the class where the extension property points, becomes superclass of the created Class. For example if the x-kindOf refers to a Schema *Dog* and points to a Class *Pet*, then we create a Class Dog and make it subclass of the Class Pet.

```
method Individual createNodeShape(Ontology ontModel, String
    schemaName, SchemaObject schemaObject, Map<String, SchemaObject
    > schemas) {
  Individual nodeShapeInd = createIndividual(ontModel, "sh:
    NodeShape");
  //handle semantics x-refersTo, x-kindOf, x-mapsTo
```

```
  String classUri;
  String collectionMember; // assigned when x-collectionTo is used
  if (schemaObject.hasExtension("x-refersTo")) {
    classUri = schemaObject.getExtension("x-refersTo");
  }
  else if (schemaObject.hasExtension("x-kindOf")) {
    String uri = schemaObject.getExtension("x-kindOf");
    Class class = ontModel.createClass(schemaName);
    class.addSuperClass(uri);
    classUri = class.getUri();
  }
  else if (schemaObject.hasExtension("x-mapsTo")) {
    String mappedSchemaName = schemaObject.getExtension("x-mapsTo
   ");
    Entry<String, SchemaObject> mappedSchemaEntry = schemas.get(
   mappedSchemaName);
    Individual mappedShapeInd = findShapeIndividual(ontModel,
   schemaName);
    if (mappedShapeInd == null) {
      mappedShapeInd = createNodeShape(ontModel, mappedSchemaEntry
   .getKey(), mappedSchemaEntry.getValue(), schemas);
    }
    classUri = mappedShapeInd.getProperty("sh:TargetClass");
  }
  else if (schemaObject.hasExtension("x-collectionOn")) {
    Class class = ontModel.createClass(schemaName);
    class.addSuperClass("openapi:Collection");
    collectionMember = schemaObject.getExtension("x-collectionOn")
   ;
    classUri = class.getUri();
  }
  nodeShapeInd.setProperty("sh:targetClass", classUri);
  for (Entry<String, SchemaObject> propertyEntry in schemaObject.
   getProperties()) {
    Individual propertyShapeInd = createPropertyShape(ontModel,
   propertyEntry.getKey(), propertyEntry.getValue(), schemas);
    propertyShapeInd.setProperty("sh:name", propertyEntry.getKey()
   );
    if (propertyEntry.getKey() == collectionMember) {
      propertyShapeInd.setProperty("sh:path", "openapi:member");
    }
    nodeShapeInd.setProperty("sh:properties", propertyShapeInd);
  }
  nodeShapeInd.setProperty("openapi:description", schemaObject.
   getDescription());
  return nodeShapeInd
}
```

Listing 5.22: NodeShape Method

In x-mapsTo we search all the Schemas and we keep the one where the extension property points. Since all Schemas are kept in a Map of a Schema Name and Schema Object, this is what it is returned. We then check if this

Schema has already been translated to a Shape in the ontology. If not, we create it. The x-mapsTo may point to a Class or to a property. Since here we create NodeShape Individuals we are in the case of a Class. Moreover, in x-mapsTo we want to extract semantics. This means that if the x-mapsTo points to a Class and then this Class uses the x-refersTo in order to point to a general Class, we want to keep the URI of this generic Class. This is kept to the variable *classUri* by getting the value of the property *targetClass* of the Shape Individual.

In x-collectionOn, we create a Class with the name of the Schema and keep its URI. Then we make the Class *Collection* superclass of the one we created and put the value where x-collectionOn points to a variable named *collectionMember*. For example, if we have the command *x-collectionOn:pet* then we create a Pet Class and make Collection Class its superclass. The value pet is kept in collectionMember.

In any of the above extension properties a value has been added to the variable *classUri*. This value is passed to the property *targetClass* of the NodeShape Individual. Each Schema Object has a property named *properties*. For each one, we create a PropertyShape Individual and we set the value of its property *name* according to its name. Finally, we set to the property *properties* value the Property Shape Individual that we have created.

5.19 CreatePropertyShape Method

This method supports also the previous extension properties but in the level of property instead of Class. This

means that in every case we keep the propertyUri instead of the classUri. Moreover, in x-kindOf after saving the URI of the property where the x-property points, we create a property with the Schema name and we make the created property subproperty of the other.

```
method Individual createPropertyShape(Ontology ontModel, String
    schemaName, SchemaObject schemaObject, Map<String, SchemaObject
    > schemas) {
  Individual propertyShapeInd = createIndividual(ontModel, "sh:
    PropertyShape");
  //handle semantics x-refersTo, x-kindOf, x-mapsTo
  String propertyUri;
  if (schemaObject.hasExtension("x-refersTo")) {
    propertyUri = schemaObject.getExtension("x-refersTo");
  }
  else if (schemaObject.hasExtension("x-kindOf")) {
    String uri = schemaObject.getExtension("x-kindOf");
    Property property = ontModel.createProperty(schemaName);
    property.addSuperProperty(uri);
    propertyUri = property.getUri();
  }
  else if (schemaObject.hasExtension("x-mapsTo")) {
    String mappedSchemaProperty = schemaObject.getExtension("x-
   mapsTo");
    String mappedSchema = extractSchemaName(mappedSchemaProperty);
    Individual mappedShapeInd = findShapeIndividual(ontModel,
   mappedSchema);
    if (mappedShapeInd == null) {
      Entry<String, SchemaObject> mappedSchemaEntry = schemas.get(
   mappedSchema);
      mappedShapeInd = createNodeShape(ontModel, mappedSchemaEntry
   .getKey(), mappedSchemaEntry.getValue(), schemas);
    }
    String mappedProperty = extractPropertyName(
   mappedSchemaProperty);
    if (mappedProperty == null) {
      propertyShapeInd.setProperty("sh:path", mappedShapeInd.
   getProperty("sh:path"));
    }
    else {
      Individual mappedPropertyShape = findNodeProperty(
   mappedShapeInd, mappedProperty);
      propertyShapeInd.setProperty("sh:path", mappedPropertyShape.
   getProperty("sh:path"));
    }
  }
  if (schemaObject.getType() == array) {
    SchemaObject itemsObject = schemaObject.getItems();
    if (itemsObject.getType() == object) {
      Individual itemsNodeShape = createNodeShape(ontModel,
   schemaName, itemsObject, schemas);
      propertyShapeInd.setProperty("sh:node", itemsNodeShape);
```

```
    }
    else {
      propertyShapeInd.setProperty("sh:datatype", getDatatype(
    itemsObject.getType(), itemsObject.getFormat());
    }
    propertyShapeInd.setProperty("sh:minCount", itemsObject.
    getMinItems());
    propertyShapeInd.setPorperty("sh:maxCount", itemsObject.
    getMaxItems());
  }
  else if (schemaObject.getType() == object) {
    Individual nodeShape = createNodeShape(ontModel, schemaName,
    schemaObject, schemas);
    propertyShapeInd.setProperty("sh:node", nodeShape);
  }
  else {
    propertyShapeInd.setProperty("sh:datatype", getDatatype(
    itemsObject.getType(), itemsObject.getFormat());
  }
  propertyShapeInd.setProperty("openapi:multipleOf",schemaObject.
    getMultipleOf());
  propertyShapeInd.setProperty("openapi:maximum",schemaObject.
    getMaximum());
  propertyShapeInd.setProperty("sh:exclusiveMaximum",schemaObject.
    getExclusiveMaximum());
  propertyShapeInd.setProperty("openapi:minimum", schemaObject.
    getMinimum());
  propertyShapeInd.setProperty("sh:exclusiveMinimum",schemaObject.
    getExclusiveMinimum());
  propertyShapeInd.setProperty("sh:maxLength",schemaObject.
    getMaxLength());
  propertyShapeInd.setProperty("sh:minLength",schemaObject.
    getMinLength());
  propertyShapeInd.setProperty("sh:pattern",schemaObject.
    getPattern());
  propertyShapeInd.addProperty("sh:description", schemaObject.
    getDescription());
  propertyShapeInd.addProperty("openapi:readOnly", schemaObject.
    getReadOnly());
  propertyShapeInd.addProperty("openapi:writeOnly", schemaObject.
    getWriteOnly());
  propertyShapeInd.addProperty("openapi:deprecated", schemaObject.
    getDeprecated());
  propertyShapeInd.setProperty("sh:defaultValue", schemaObject.
    getDefault());
  Individual exDocInd = parseExternalDocObject(ontModel,
    schemaObject.getExternalDoc());
  nodeShapeInd.addProperty("openapi:exDoc", exDocInd);
  Individual xmlInd = parseXmlObject(ontModel, schemaObject.getXml
    ());
  propertyShapeInd.setProperty("openapi:xml", xmlInd);
  return propertyShapeInd;
}
```

Listing 5.23: PropertyShape Method

In x-mapsTo we keep the property where the exten-

sion property points. Since we now have property and not Class we might have something like *pet.id*. What we do is keep only the Schema name, aka *pet* by using the *extractSchemaName* method. We check if this Schema has already been translated to a NodeShape in the ontology and if not we create it and we extract the semantics the same way as in the previous section. Next, we keep only the Schema's property name by using the *extractPropertyName* method. After that we once again extract the semantics which in this case are placed in the property *path*.

Next we get the *type* of the Schema and if it is an array then we get the property *items*. If the type of the itemsObject is array or object then we create a NodeShape Individual. Finally, we return the PropertyShape Individual.

5.20 CreateCollectionNodeShape Method

In this method we work similarly to the implementation of the x-collectionOn property. More specifically, we create a NodeShape Individual, we create a class named after the Schema name and we make it subclass of the Collection Class. We set the value of the property *targetClass* according to the class uri. Next, we create a PropertyShape Individual and we set its property *path* according to member. Finally, we return the NodeShape Individual.

```
method Individual createCollectionNodeShape(Ontology ontModel,
    String schemaName, SchemaObject schemaObject, Map<String,
    SchemaObject> schemas) {
  Individual nodeShapeInd = createIndividual(ontModel, "sh:
    NodeShape");
```

```
  Class class = ontModel.createClass(schemaName);
  class.addSuperClass("openapi:Collection");
  nodeShapeInd.setProperty("sh:targetClass", class.getUri());
  Individual memberInd = createPropertyShapeInd(ontModel, null,
   schemaObject, schemas);
  memberInd.setProperty("sh:path", "openapi:member");
  return nodeShapeInd;
}
```

Listing 5.24: CollectionNodeShape

5.21 ParseSecuritySchemeObject Method

SecuritySchemeObjects may have four types. For each type we create the corresponding Individual and we set its properties. For the *OAuth2* case we call the *parseOAuthFlowsIndividual*.

```
method void parseSecuritySchemeObject(OntModel ontModel,
    Individual documentInd, String securityName,
    SecuritySchemeObject securityObject){
  switch (securityObject.getType())
  {
  case APIKEY :
    Individual securityInd = createIndividual(ontModel, "openapi:
   APIKEY", securityName);
    securityInd.setProperty("openapi:parameterName",
   securityObject.getName());
    securityInd.setProperty("openapi:in", securityObject.getIn());
    securityInd.setProperty("openapi:description", securityObject.
   getDescription());
  case HTTP:
    Individual securityInd = createIndividual(ontModel, "openapi:
   HTTP", securityName);
    securityInd.setProperty("openapi:scheme", securityObject.
   getScheme());
    securityInd.setProperty("openapi:bearerFormat", securityObject
   .getBearerFormat());
    securityInd.setProperty("openapi:description", securityObject.
   getDescription());
  case OAUTH2:
    Individual securityInd = createIndividual(ontModel, "openapi:
   OAUTH2", securityName);
    flowsInd= parseOAuthFlowsIndividual(ontModel, securityObject.
   getFlows()));
    securityInd.setProperty("openapi:description", securityObject.
   getDescription());
    securityInd.setProperty("openapi:flow", flowsInd );
```

```
  case OPENIDCONNECT:
    Individual securityInd = createIndividual(ontModel, "openapi:
   OPENIDCONNECT", securityName);
    securityInd.setProperty("openapi:description", securityObject.
   getDescription());
    securityInd.setProperty("openapi:openIdConnectUrl",
   securityObject.getOpenIdConnectUrl());
  }
    documentInd.setProperty("openapi:supportedSecurity", secInd);
}
```

Listing 5.25: SecurityScheme

5.22 ParseSequrityReqObject Method

In this method SecurityRequirement Objects are mapped to SecurityRequirement Individuals. For every SecurityReqObject we get the SecuritySchemeObject that has the same *name* as the SecurityReqObject. Finally the Individual is returned.

```
method Individual parseSecurityReqObject(OntModel ontModel, Map<
    String, List<String>> securityreqObject){
  securityreqInd = createIndividual(ontModel, "openapi:
   SecurityRequirement");
  //get the security scheme with specific url (name)
  for(Entry<String, List<String>> securityReqEntry in
   securityreqObject){
    securitySchemeInd=ontModel.getIndividual(securityReqEntry.
   getKey());
    securityreqInd.setProperty("openapi:securityType",
   securitySchemeInd);
    for (scope in securityReqEntry.getValue()){
        scopeInd=parseScopeIndividual(ontModel,scope,null);
        securityreqInd.setProperty("openapi:scope", scopeInd);
      }
  }
  return securityreqInd;
}
```

Listing 5.26: SecurityRequirement

5.23 ParseOAuthFlowsIndividual Method

OAuthFlowsObject has properties implicit, authorizationCode, clientCredentials and Password. For the ones that are not null, we call the corresponding method. Finally, we return the OAuthFlows Individual.

```
method Individual parseOAuthFlowsIndividual(ontModel,
    OauthFlowsObject oauthflowsObject){
  if(oauthflowsObject.getImplicit()!=null)
    Individual oauthflowsInd=parseImplicitFlowIndividual(ontModel,
    oauthflowsObject.getImplicit());
  else if (oauthflowsObject.getAuthorizationCode()!=null)
    Individual oauthflowsInd=parseAuthorizationCodeFlowIndividual(
   ontModel,oauthflowsObject.getAuthorizationCode();
  else if (oauthflowsObject.getClientCredentials()!=null)
    Individual oauthflowsInd=parseClientCredentialsFlowIndividuall
   (ontModel, oauthflowsObject.getClientCredentials());
  else
    Individual oauthflowsInd=parsePasswordFlowIndividual(ontModel,
    oauthflowsObject.getPassword());
  return oauthflowsInd;
}
```

Listing 5.27: OAuth Flows

5.24 AuthorizationCode, ClientCredentials, Password, Implicit, Scope parsing Methods

For each of the following methods, we create the corresponding Individual, then we set its properties and we return the Individual.

```
method Individual AuthorizationCodeFlowIndividual(ontModel,
    OauthFlowObject oauthflowObject){
  Individual oauthflowInd=createIndividual(ontModel, "openapi:
    AuthorizationCode");
  oauthflowInd.setProperty("openapi:authorizationUrl",
    oauthflowObject.getTokenUrl());
  oauthflowInd.setProperty("openapi:tokenUrl", oauthflowObject.
    getTokenUrl());
```

```
  oauthflowInd.setProperty("openapi:refreshUrl", oauthflowObject.
    getRefreshUrl());
  for (Entry <String, String> scopeEntry in oauthflowsObject.
    getScopes()){
     Individual scopeInd=parseScopeIndividual(ontModel,scopeEntry.
    getKey(), scopeEntry.getValue());
     oauthflowInd.setProperty("openapi:scope", scopeInd);
  }
  return oauthflowInd;
}
```

Listing 5.28: AuthorizationCode

```
method Individual parseClientCredentialsFlowIndividual(ontModel,
    OauthFlowObject oauthflowObject){
  Individual oauthflowInd=createIndividual(ontModel, "openapi:
    ClientCredentials");
  oauthflowInd.setProperty("openapi:tokenUrl", oauthflowObject.
    getTokenUrl());
  oauthflowInd.setProperty("openapi:refreshUrl", oauthflowObject.
    getRefreshUrl());
  for (Entry <String, String> scopeEntry in oauthflowsObject.
    getScopes()){
     Individual scopeInd=parseScopeIndividual(ontModel,scopeEntry.
    getKey(), scopeEntry.getValue());
     oauthflowInd.setProperty("openapi:scope", scopeInd);
  }
  return oauthflowInd;
}
```

Listing 5.29: ClientCredentials

```
method Individual ParsePasswordFlowIndividual(ontModel,
    OauthFlowObject oauthflowObject){
  Individual oauthflowInd=createIndividual(ontModel, "openapi:
    Password");
  oauthflowInd.setProperty("openapi:tokenUrl", oauthflowObject.
    getTokenUrl());
  oauthflowInd.setProperty("openapi:refreshUrl", oauthflowObject.
    getRefreshUrl());
  for (Entry <String, String> scopeEntry in oauthflowsObject.
    getScopes()){
     Individual scopeInd=parseScopeIndividual(ontModel,scopeEntry.
    getKey(), scopeEntry.getValue());
     oauthflowInd.setProperty("openapi:scope", scopeInd);
  }
  return oauthflowInd;
}
```

Listing 5.30: Password

```
method Individual ParseImplicitFlowIndividual(ontModel,
    OauthFlowObject oauthflowObject){
  Individual oauthflowInd=createIndividual(ontModel, "openapi:
    Implicit");
  oauthflowInd.setProperty("openapi:authorizationUrl",
    oauthflowObject.getAuthorizationUrl());
  oauthflowInd.setProperty("openapi:refreshUrl", oauthflowObject.
    getRefreshUrl());
```

```
  for (Entry <String, String> scopeEntry in oauthflowsObject.
    getScopes()){
     Individual scopeInd=parseScopeIndividual(ontModel,scopeEntry.
    getKey(), scopeEntry.getValue());
     oauthflowInd.setProperty("openapi:scope", scopeInd);
  }
  return oauthflowInd;
}
```

Listing 5.31: Implicit

```
method parseScopeIndividual(OntModel ontModel, String scopeName,
    String scopeDescription){
  Individual scopeInd = createIndividual(ontModel, "openapi:Scope
    ");
  scopeInd.setProperty("openapi:name", scopeName);
  scopeInd.setProperty("openapi:description", scopeDescription);
  return scopeInd;
}
```

Listing 5.32: Scope

Chapter 6

Examples Mapping and Results

6.1 Swagger Petstore Mapping

In this section, we present how the mechanism that we described in the previous chapter works on the example of Swagger Petstore. In listing 6.1 the OpenAPI code for Swagger Petstore is presented.

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1

paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max
    100)
          required: false
          schema:
            type: integer
            format: int32
```

```
      responses:
        '200':
          description: A paged array of pets
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pets"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"

    post:
      summary: Create a pet
      operationId: createPets
      tags:
        - pets
      responses:
        '201':
          description: Null response
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"

  /pets/{petId}:
    get:
      summary: Info for a specific pet
      operationId: showPetById
      tags:
        - pets
      parameters:
        - name: petId
          in: path
          required: true
          description: The id of the pet to retrieve
          schema:
            type: string
      responses:
        '200':
          description: Expected response to a valid request
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pet"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
components:
  schemas:
    Pet:
```

```
83        type: object
84        x-refersTo: "http://myserver/Animal"
85        required:
86          - id
87          - name
88        properties:
89          id:
90            type: integer
91            format: int64
92          name:
93            type: string
94            x-refersTo: "http://myserver/Animal.name"
95          tag:
96            type: string
97      Pets:
98        type: array
99        x-collectionOn:  "#/components/schemas/Pet"
100       items:
101         $ref: "#/components/schemas/Pet"
102     Error:
103       type: object
104       x-refersTo: "http://myserver/Errors"
105       required:
106         - code
107         - message
108       properties:
109         code:
110           type: integer
111           format: int32
112         message:
113         type: string
```

Listing 6.1: Swagger Petstore Example

In the beginning of the example - in line 2 *info* as an *Info Object* is declared. This parses to an *Info Class* Individual in our ontology. Next, in line 5 we have the declaration of *license* which corresponds to a *License Class* Individual. The same happens in line 7 where we have the declaration of *Servers Object* that is mapped to a *Servers Class* Individual. The property *url* maps to the url property of Servers Class.

After that, we have the first declaration of a *Paths Object* in line 10 and then the first *Operation Object* in line 11. These correspond to an Individual of *Path Item Class* in OpenAPI ontology and an Individual of *Operation Class*. The command */pets* is translated in the

ontology as an Individual of a *Path Class*, whose name is */pets*. In the next line we see a *get* command which is mapped to a *Method* Individual. In line 15 we have a *Tag Object* which maps to an Individual of Tag Class whose property:name is *pets*.

In the following lines we have a declaration of a *Parameter Object*, which has the value *query* in the property *in*. This is mapped to a *Query Class* Individual to our ontology. In line 22 we have a declaration of a *Schema Object* with type:integer, which parses to an Individual of a *Property Shape Class*. After that, we have a declaration of a *Responses Object* which maps to a *Responses Class* Individual. The number "200" in the next line means that the *statusCode* of the response is 200. In line 28 by the command *content* a *Media Type Object* is declared which maps to a *Media Type* Individual. The command *application/json* is the value of the property *name* of the Media Type Individual. The ref is a pointer to /components/schemas/Pets, which means that the Schema declared in the previous lines inherets the properties of the Schema *Pet* under the Components. In line 47 *default* is a response to statusCode which means that it is mapped to an Individual of *DefaultResponse Class*, a subclass of Response Class.

In line 39 the second *Operation Object* is declared. As happened before, this is mapped to an *Operation Individual*. The procedure in the next lines is the same as the one followed in the first Operation Individual.

In line 54 there is the declaration of a second *Path Item Object*. The command */pets/petId* is an Individual of *Path Class* with name */pets/petId*. In line 60 the *Parameter Object* has as value in the *in* property the

value *path*. This is mapped to an Individual of a *Parameter Class* in our ontology. Next, in line 65 there is a declaration of a *Schema Object* with *type:string*. This is translated as a *Property Shape Individual* in OpenAPI ontology. The following lines are parsed in same manner as mentioned above.

In line 82 *Pet* is a *Schema Object* which is mapped to a *NodeShape Class* Individual. In line 84 the x-refersTo property is used in order to link a Pet to an Animal definition. In fact, http://myserver/Animal becomes the value of the property *targetClass* in Shape Class of the ontology. In line 89 *id* is a Schema Object under the property *properties*. This is mapped to a *PropertyShape Class* Individual. Similarly, *name* is also mapped to a *PropertyShape Class* Individual. The x-refersTo property in line 93 since it's now applied to a PropertyShape Individual, assigns the value http://myserver/Animal.name to the *path* property. The same thing happens in line 95 with *tag*.

Moreover, *Pets* in line 97 is a *Schema Object* declared as an array, which means that it is associated to a *CollectionNodeShape Class* Individual in OpenAPI ontology. The x-collectionOn property creates a new class [Pets], subclassof Collection Class. This new Pet Class is consisted of Pet Individuals. The next two lines which consist of the commands *items* and ref indicate that all items of the array are Pet Individuals. *Error* is also a *Schema Object* with the type:object which means, as mentioned before, that is mapped to a *NodeShape Class* Individual. In line 109 *code* is a *Schema Object* with type:integer, hence a *PropertyShape Class* Individual, while in line 112 *message* with type:string is also a *PropertyShape Class* Individual. Finally, the *type* and *format* in lines 110-111

define the datatype of each schema (sh:datatype).

6.2 UpsTo Example

Listing 6.2 illustrates the UpsTo example which is used along with Swagger Petstore in order to run queries and get results from two OpenAPI examples at the same time. UpsTo is in fact an API containing datasets. The operations available include searching for a specific dataset by providing name and version.

```
openapi: 3.0.1
servers:
  - url: '{scheme}://developer.uspto.gov/ds-api'
    variables:
      scheme:
        description: 'The Data Set API is accessible via https and
    http'
        enum:
          - 'https'
          - 'http'
        default: 'https'
info:
  description: >-
    The Data Set API (DSAPI) allows the public users to discover
    and search USPTO exported data sets. With the help of GET
    call, it returns the list of data fields that are searchable.
    With the help of POST call, data can be fetched based on the
    filters on the field names.
  version: 1.0.0
  title: USPTO Data Set API
  contact:
    name: Open Data Portal
    url: 'https://developer.uspto.gov'
    email: developer@uspto.gov
tags:
  - name: metadata
    description: Find out about the data sets
  - name: search
    description: Search a data set
paths:
  /:
    get:
      tags:
        - metadata
      operationId: list-data-sets
      summary: List available data sets
      responses:
        '200':
```

```
        description: Returns a list of data sets
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/dataSetList'
            example:
              {
                "total": 2,
                "apis": [
                  {
                    "apiKey": "oa_citations",
                    "apiVersionNumber": "v1",
                    "apiUrl": "https://developer.uspto.gov/ds-api/oa_citations/v1/fields",
                    "apiDocumentationUrl": "https://developer.uspto.gov/ds-api-docs/index.html?url=https://developer.uspto.gov/ds-api/swagger/docs/oa_citations.json"
                  },
                ]
              }
  /{dataset}/{version}/fields:
    get:
      tags:
        - metadata
      summary: Provides the general information about the API and the list of fields that can be used to query the dataset.
      operationId: list-searchable-fields
      parameters:
        - name: dataset
          in: path
          description: 'Name of the dataset.'
          required: true
          example: "oa_citations"
          schema:
            type: string
        - name: version
          in: path
          description: Version of the dataset.
          required: true
          example: "v1"
          schema:
            type: string
      responses:
        '200':
          description: >-
            The dataset API for the given version is found and it is accessible to consume.
          content:
            application/json:
              schema:
                type: string
        '404':
          description: >-
            The combination of dataset name and version is not found in the system or it is not published yet to be consumed by public.
          content:
            application/json:
```

```
            schema:
              type: string
/{dataset}/{version}/records:
  post:
    tags:
      - search
    summary: Provides search capability for the data set with the given search criteria.
    operationId: perform-search
    parameters:
      - name: version
        in: path
        description: Version of the dataset.
        required: true
        schema:
          type: string
          default: v1
      - name: dataset
        in: path
        description: 'Name of the dataset. In this case, the default value is oa_citations'
        required: true
        schema:
          type: string
          default: oa_citations
    responses:
      '200':
        description: successful operation
        content:
          application/json:
            schema:
              type: array
              items:
                type: object
                additionalProperties:
                  type: object
      '404':
        description: No matching record found for the given criteria.
    requestBody:
      content:
        application/x-www-form-urlencoded:
          schema:
            type: object
            properties:
              criteria:
                description: >-
                  Uses Lucene Query Syntax in the format of
                  propertyName:value, propertyName:[num1 TO num2] and date range format: propertyName:[yyyyMMdd TO yyyyMMdd]. In the response please see the 'docs' element which has the list of
                  record objects. Each record structure would consist of all the fields and their corresponding values.
                type: string
                default: '*:*'
              start:
```

```
                  description: Starting record number. Default value is 0.
                  type: integer
                  default: 0
                rows:
                  description: >-
                    Specify number of rows to be returned. If you run the search with default values, in the response you will see 'numFound' attribute which will tell the number of records available in
                    the dataset.
                  type: integer
                  default: 100
              required:
                - criteria
components:
  schemas:
    dataSetList:
      type: object
      properties:
        total:
          type: integer
        apis:
          type: array
          items:
            type: object
            properties:
              apiKey:
                type: string
                description: To be used as a dataset parameter value
              apiVersionNumber:
                type: string
                description: To be used as a version parameter value
              apiUrl:
                type: string
                format: uriref
                description: "The URL describing the dataset's fields"
              apiDocumentationUrl:
                type: string
                format: uriref
                description: A URL to the API console for each API
```

Listing 6.2: UpsTo example

6.3 Queries and Results

To discover services over the Web, we might opt to develop tools capable for searching SOAS 3.0 service catalogues over the Web. A more elaborate solution requires

that services are instantiated to the SOAS 3.0 ontology (a parser is capable of interpreting the meaning of SOAS 3.0 descriptions and for mapping concepts to the ontology using Apache Jena or OWL API). This would enable application of state-of-the- art query languages (e.g. SPARQL) for service discovery. Reasoning (e.g. using Pellet) can also be used for detecting inconsistencies in SOAS representations. Machine readable representations can also facilitate more complicated tasks such as service synthesis and service orchestration. For this purpose we have run some queries to the example of Swagger Petstore and UpsTo in order to check the results.

```
PREFIX rdf : <http://www.w3.org/1999/02/22_rdf_syntax_ns>
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api>
  SELECT ?pathName
  WHERE {
    ?document rdf:type openapi:Document .
    ?document openapi:supportedOperation ?operation .
    ?operation openapi:tag ?tag .
    ?tag openapi:name pets .
    ?operation openapi:onPath ?pathName .
    }

------------------
Answer:
1. /pets (petstore)
2. /pets/{petId} (petstore)
------------------
```

Listing 6.3: Retrieve all paths that use tag with name-pets

```
PREFIX rdf:<http://www.w3.org/1999/02/22_rdf_syntax_ns>
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api>

SELECT  ?description
WHERE {
?document rdf:type openapi:Document .
?document openapi:supportedOperation ?operation .
?oparation openapi:responses ?responses .
?responses openapi:statusCode 200 .
?responses openapi:description ?description .
}
------------------
Answer:
1. A paged array of pets (petstore)
```

```
2. Expected response to a valid request (petstore)
3. successful operation (UpsTo)
4. Returns a list of data sets (UpsTo)
5. The dataset API for the given version is found and it is
    accessible to consume (upsTo)
------------------
```

Listing 6.4: Retrieve descriptions of responses with status code '200'

```
PREFIX rdf:<http://www.w3.org/1999/02/22_rdf_syntax_ns>
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api>

SELECT  ?parameterName
WHERE {
?document rdf:type openapi:Document .
?document openapi:supportedOperation ?operation .
?operation openapi:paramerets ?parameter .
?parameter rdf:type openapi:Query .
?parameter openapi:name ?parameterName .
}

------------------
Answer:
1. dataset (UpsTo)
2. version (UpsTo)
3. petId (petstore)
------------------
```

Listing 6.5: Retrieve all names of parameters that are used in Path

```
PREFIX rdf:<http://www.w3.org/1999/02/22_rdf_syntax_ns>
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api>

SELECT  ?operationId, ?operationId
WHERE {
?document rdf:type openapi:Document .
?document openapi:supportedOperation ?operation .
?operation openapi:tag ?tag .
?tag openapi:name ?tagNames .
?operation openapi:operationId ?operationId.
}

-------------------------------------
Answer:
1. list-data-sets, metadata (upsTo)
2. perform-search, search (upsTo)
3. listPets, pets (petstore)
4. createPets, pets (petstore)
5. showPetById, pets (petstore)
-------------------------------------
```

Listing 6.6: Retrieve operationId's with the corresponding tag name

```
PREFIX rdf:<http://www.w3.org/1999/02/22_rdf_syntax_ns>
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api>

SELECT  ?serviceTitle
WHERE {
?document rdf:type openapi:Document .
?document openapi:info ?info .
?info openapi:serviceTitle? ?serviceTitle.
}


-------------------------------------
Answer:
1. Swagger Petstore (petstore)
2. USPTO Data Set API (upsTo)
-------------------------------------
```

Listing 6.7: Retrieve service titles

Seeing the above results and comparing with the Petstore and UpsTo examples we observe that the parameters returned are the same with the ones declared in sections 6.1 and 6.2. Hence, by using SOAS 3.0 we are able to translate an OpenAPI Service to the OpenAPI Ontology and then take advantage of the SPARQL language in order to pose queries and get the corresponding responses while having eliminated the ambiguities that could initially exist in the OpenAPI description service.

We now present a more interesting example. In listing 6.8 we may see the example of Swagger Petstore with different kind of pets (cats, dogs) enriched with the extension properties that were introduced by SOAS 3.0. Listing 6.9 shows a query used for retrieving all Operation Ids in Swagger Petstore that are related with *pets*. The results returned are *list pets, create pets, list dogs, list cats*. What we have done is to connect the Operations with their corresponding Tags, add to them the x-onResource property in order to connect them with the corresponding Schema and use the x-refersTo and x-kindOf extension properties in order to semantically refer to a Pet onto-

logy in the Web. This example illustrates the reason for which the transformation of OAS (and therefore SOAS) services in ontology is important. What we realize is that this work creates machine-understandable services that are now able to take into consideration hidden and indirect relationships.

```
tags:
  -name: pets
   description: Everything about Pets
   x-onResourse: '#/components/schemas/Pet'

   -name:cats
   description: Everything about Cats
   x-onResourse: '#/components/schemas/Cat'

   -name:dogs
   description: Everything about Dogs
   x-onResourse: '#/components/schemas/Dog'

paths:
 /pets:
    get:
     summary: List all pets
     operationId: list pets
   tag:
     -pets
   ....
    post:
     summary : Create a pet
     operationId: create pets
   tag:
     -pets
   ....
/dogs:
     get:
      summary: List all dogs
      operationId: list dogs
    tag:
     -dogs
    ....
/cats:
     get:
      summary: list all cats
      operationId: list cats
    tag:
     -cats

    ....

components:
  schemas:
     Pet:
```

```
      type: object
      x-refersTo: http://schema.org/Pet
      discriminator:
        propertyName: petType
      properties:
        name:
          type: string
        petType:
          type: string
      required:
      - name
      - petType
    Cat:
      description: A representation of a cat
      allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
       x-kindOf: http://schema.org/Pet
        properties:
          huntingSkill:
            type: string
            description: The measured skill for hunting
            enum:
            - clueless
            - lazy
            - adventurous
            - aggressive
        required:
        - huntingSkill
    Dog:
      description: A representation of a dog
      allOf:
      - $ref: '#/components/schemas/Pet'
      - type: object
       x-kindOf: http://schema.org/Pet
        properties:
          packSize:
            type: integer
            format: int32
            description: the size of the pack the dog is from
            default: 0
            minimum: 0
        required:
        - packSize
```

Listing 6.8: Swagger Petstore Example with Cats and Dogs

```
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api>
PREFIX schema : <https://schema.org/>

SELECT ?operationId
WHERE {
?document rdf:type openapi:Document .
?document openapi:supportedEntity ?schemaInd .
?schemaInd openapi:targetClass schema:Pet.
?schemaInd openapi:supportedOperation ?operationInd .
?operationInd openapi:operationId ?operationId .
}
```

```
_____________________
Answer:
1. list pets
2. create pets
3. list dogs
4. list cats
_____________________
```

Listing 6.9: Example of how a Query returns hidden relationships

Chapter 7

Conclusion and Future Work

7.1 Conclusions

Throughout this thesis 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. There are diverse techniques for implementing RESTful services. What we adopted and suggested was OpenAPI Specification.

The selection of the OAS, was motivated by the popularity of the specification, its powerful tool support, and the active community. OpenAPI Specification offers a human-friendly environment for discovering 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. Taking advantage of the extension features foreseen in OAS 3.0, our approach suggested that OAS properties must be semantically annotated. Leveraging latest results for hypermedia-based construction of Web APIs, service descriptions were translated to the OpenAPI ontology by using the mechanism that was implemented

and explained along with its algorithm in this thesis. The translation of a service to an ontology enables the use of Semantic Web tools such as reasoners and query languages, something that was shown in Chapter 6 when presenting the results from the SPARQL queries. Therefore, the application of SOAS 3.0 enables the services to become machine-understandable while taking advantage of semantic technologies.

7.2 Future Work

The last update (version 3.0) of OpenAPI Specification added support to two very imporant features. The first one is Callbacks and the other one is Links. Callbacks are asynchronous requests that the server service will send to some other service in response to certain events. This feature improves the workflow that the server API offers to its clients. Using links, enables the description of how various values returned by one operation can be used as input for other operations. Both these new features make the enrichment of our proposed mechanism - in order to support HATEOAS - possible. By doing that, we might take advantage of the possibilities that OAS has to offer such as explorable API - meaning the ability to browse around the data. This makes it a lot easier for the client developers to build a mental model of the API and its data structures.

List of Figures

Bibliography

[1] Semantically Enriched API Descriptions for Improving Service Discovery in Cloud Environments, Nikolaos Mainas, 2017

[2] SOAS 3.0: Semantically Enriched OpenAPI 3.0 Descriptions and Ontology for REST Services, Nikolaos Mainas, Euripides G.M. Petrakis, 2019

[3] Semantic Web, Wikipedia

[4] What Is the Semantic Web, ontotext

[5] Jack Koftikian. Simple Object Access Protocol (SOAP)

[6] W3C.org. Web Services Description Language (WSDL). 2011

[7] W3schools.com. XML WSDL

[8] Techopedia.com. UDDI

[9] Roy Fielding. REST dissertation. 2000

[10] Understanding Security and Dependability for SOAP and REST, apicasystems.com

[11] Ontology (information science), Wikipedia

[12] What are ontologies?, ontotext.com

[13] OWL, w3.org

[14] A Guide to What's New in OpenAPI 3.0, Swagger.io

[15] Hydra Core Vocabulary, hydra-cg.com

[16] Hydra: Hypermedia-Driven Web APIs, markus-lanthaler.com/hydra/

[17] JSON-LD 1.0, w3.org/TR/json-ld/

[18] Apache Jena, jena.apache.org

[19] SPARQL 1.1 Query Language, w3.org/TR/sparql11-query/

[20] Pellet: A Practical OWL-DL Reasoner

[21] OpenAPI Spec and Swagger, idratherbewriting.com

[22] OpenAPI Specification, swagger.io

[23] Web Service Description, IBM.com

[24] Why Web Services?, tutorialspoint.com

[25] Why Web Services are important, flylib.com