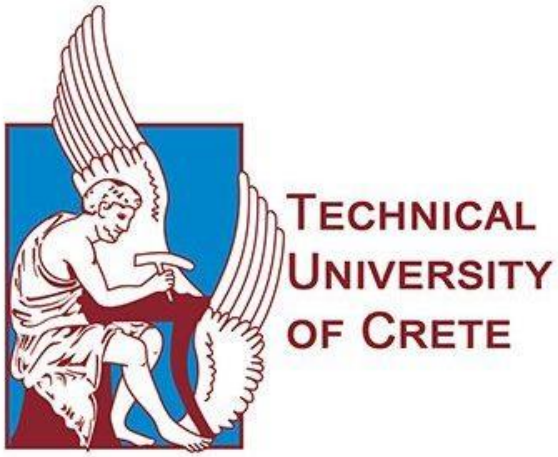


School of Electrical and
Computer Engineering

Diploma Thesis

Instantiating OpenAPI Descriptions to the REST Services Ontology

Bouraimis Fotios

Committee

Supervisor: Prof. Euripides G. M. Petrakis

Assoc. Prof. Michail G. Lagoudakis

Assoc. Prof. Samoladas Vasileios

Abstract

The increasing interest in Web Service architectures over the past years has led to the proliferation of Web service offerings over the internet. Consequently, the need for efficient and accurate service discovery based on user needs has become a significant challenge. In order for services to become understandable and discoverable by humans and machines they need to be formally described. In this work, we use the OpenAPI Specification (OAS), a widely used specification for the description of REST APIs. OpenAPI descriptions are mainly understandable by humans. However, OpenAPI descriptions need to 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 OpenAPI, service descriptions need to be formally defined and their content be semantically enriched in a way that eliminates ambiguities. Taking into consideration the advantages of OpenAPI 3.0, our approach suggests that in order to eliminate ambiguities in OpenAPI descriptions, OpenAPI properties must be semantically annotated. Building-upon the latest version of OpenAPI and taking advantage of Semantic OpenAPI (SOAS 3.0) this work provides a complete mechanism to transform OpenAPI descriptions to ontologies. As a result, the ontology will make service discovery possible with the application of query languages (e.g., SPARQL) and reasoning tools for detecting inconsistencies and inferred relationships in SOAS descriptions.

Acknowledgments

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 Vasileios Samoladas and Professor Michail G. Lagoudakis for their constructive comments and for participating in the evaluation committee. I would also like to thank all the members of the Intelligence Lab for their excellent communication and generous support.

Contents

Chapter 1

Introduction

1.1 Motivation

The World Wide Web has become an integral part of our daily life. The Web is realized as a composition of Web services. A Web service is a unit that provides a variety of functions (often referred to as services) which are activated over HTTP. Web services comprise a great tool for the Web developer community. Typically, Web services are described in plain text which users have to browse and read, in order to determine whether a service meets their needs. However, text descriptions are not readable by machines and in some cases are inaccurate or vague. Web services need to be formally described in a way that is understandable by both humans and machines. The last requirement would not only improve the accuracy of service descriptions but also, would allow for services to be discovered by other services and be orchestrated in composite services or applications.

A Web service description is a document by which the service provider communicates the specification of the Web service to the service requester. OpenAPI specification (OAS)[1] is a widely adopted standard for describing REST APIs which is supported by large industry users like Google, Microsoft, IBM, Oracle and many others. However, the problem of inaccuracy remains.

1.2 Problem Definition

OpenAPI Specification (OAS) is a description format for REST APIs. The descriptions of Web services written in this format are mainly understandable by humans. However, OpenAPI 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 an OpenAPI description, the OpenAPI properties must be semantically annotated and mapped to a semantic model. 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 Proposed Solution

In order for a machine to understand the meaning of an OpenAPI service description, a service description needs to be formally defined and its content semantically enriched. In a previous work [1], we proposed that OpenAPI service descriptions can be semantically annotated using extension properties. The extended model was the Semantic OpenAPI Specification (Semantic OpenAPI [2]). Taking a step forward we then created a mechanism that achieves the association of OpenAPI entities to entities of an Ontology. The present work complements and improves this approach in certain ways.

The new ontology instantiation algorithm can handle a wider range of OpenAPI Schema Objects. This is feasible by supporting the instantiation of new property keywords in Schema Objects that are introduced with OpenAPI Specification v3.0. Along with the property keywords, the algorithm now supports model composition which allows the definition of composition Schema. In addition, the algorithm supports model polymorphism which allows a model to accommodate more than one OpenAPI schemas. Continuing, we introduced the concept of inheritance between the ontology classes that correspond to OpenAPI Schema Objects. As a last step concerning the instantiation algorithm, we managed to semantically enrich OpenAPI descriptions in a more efficient way. In particular, the algorithm creates ontology entities even if a description is not semantically annotated. Therefore, the newly defined semantic entities can be used in other OpenAPI descriptions. Finally, we provided a Web Application to the community. The Web Application allows for users to translate their OpenAPI description in to an ontology and offers several other additional features as well.

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

1.4 Contributions of the Work

The following summarizes the contributions of this work:

- Supports the instantiation of all Schema Object property keywords introduced with OpenAPI Specification v3.0.
- Improves the existing instantiation algorithm of [1] to support model composition between OpenAPI schemas. Model composition allows for the combination of OpenAPI schemas.
- Introduces the concept of polymorphism between OpenAPI schemas. Polymorphism allows a model to accept more than one OpenAPI schemas.
- Introduces the concept of inheritance concerning the entities of the ontology. Inheritance creates relations between the ontology classes that correspond to OpenAPI schemas.
- Expands the range of service discovery by semantically enriching OpenAPI descriptions in a more efficient way.
- Provides a Web application to the community, that supports the translation of OpenAPI descriptions to instances of the OpenAPI ontology as well as making SPARQL queries online to every available ontology on the server.
- Demonstrates the use of the algorithm for service discovery in several service descriptions which are available on the Web and also presents the results and benefits that are derived by using our mechanism for the transformation of a Semantic OpenAPI service description to an ontology.

1.5 Thesis Outline

In chapter 2 we present the necessary background for this work and we briefly describe technologies that were used in this thesis. In chapter 3 we describe our method on OpenAPI properties and explain our approach on every step. Also, we present in an abstract level the part of the algorithm that was created during this thesis and handles each case presented in chapter 3. Continuing, in chapter 4 we present the whole instantiation algorithm in order to give the reader the opportunity to have an overview of our mechanism. Chapter 5 contains the results that were derived by applying our mechanism to Google services and demonstrates real life situations of service discovery. In addition, in chapter 5 we describe the functionality and user interface of the Web Application. Finally, chapter 6 presents our conclusions and our plans for future work.

Chapter 2

Background

2.1 Introduction

The Semantic Web is presented as an extension of the World Wide Web following the standards of World Wide Web Consortium[2] (W3C). Information in Semantic Web is offered both in human-readable and machine-readable data, making the communication between machines possible. It also provides a set of standards and technologies contributing to an environment where data and their relationships are represented in a common data format.

The Resource Description Framework (RDF)[3] is the heart of Semantic Web. RDF is a data model for expressing information about resources using statements. A resource is identified by an International Resource Identifier (IRI). IRIs are global identifiers and can be used to identify the same resource. RDF describes the relationship between two resources in the form of a triple. A triple contains a *Subject* connected to an *Object* with a relation represented by the *Predicate*. A collection of triples builds a graph, and a collection of graphs forms a dataset. An example is given in Figure 2.1.

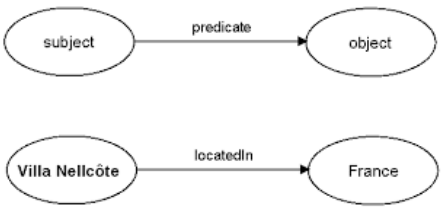


Figure 2.1: Semantic triple

RDF Schema (RDFS)[4] and Web Ontology Language (OWL)[5] provide tools for creating vocabularies as well as ontologies that capture knowledge in an area of interest. More specifically, ontologies provide the means for representing high level concepts, their properties and interrelationships. Of course, there are also other ways to represent knowledge, like vocabularies or logical models but ontologies offer many more advantages. In ontologies we meet terms such as classes, instance of classes (individuals), properties, attributes, restrictions, relations as well as axioms and rules. The advantage of ontologies is that they dictate an easy but also formal way to express relationships and linking data to specific concepts. The difference with RDFS is that RDFS provides a data-modeling vocabulary for RDF as well as mechanisms for describing classes, class hierarchies, data types or properties. Unlike RDFS, the Web Ontology Language (OWL) is a family of knowledge representation language offering increased expressiveness for describing classes and properties. Among others, OWL allows for the definition of relations between classes (e.g., disjointness), equality, restrictions over properties (e.g., cardinality restrictions) and partial order or equivalence relations between properties (e.g., transitive, symmetric properties).

In order to check the validity of the relations between its entities, the Semantic Web offers tools called Semantic reasoners. A Semantic reasoner (e.g., Pellet) is a piece of software able to infer logical consequences from a set of asserted facts or axioms. First, is the most popular of all, the *consistency* checking. Consistency checking ensures that ontologies do not contain any contradictory facts. Next is checking a class of an ontology whether it is possible to have instances. That is called *concept satisfiability* and if a class is unsatisfiable, defining an instance of a class will cause an inconsistency problem. Next is the *classification*, which creates the class hierarchy by checking classes and their subclasses. Finally, is the *realization*, which finds the most specific classes of individuals.

[2] https://www.w3.org/
[3] https://www.w3.org/RDF/
[4] https://www.w3.org/TR/rdf-schema/
[5] https://www.w3.org/OWL/

The last piece of Semantic Web is the ability for querying RDF data. SPARQL[6] an acronym for Protocol and RDF Query Language, is the W3C standard recommendation for querying and manipulating RDF data as well as a protocol to invoke queries over HTTP and a number of result formats (XML, JSON, CSV).

2.2 REST

REST (Representational State Transfer) is a software architectural style for creating Web Services. All REST-compliant systems are characterized by five principles (and one optional) that must be met in order to call a Web Service RESTful. REST is the most dominant architectural style through Web Services since it`s introduction to the public and that is because it offers great convenience for Web Services to communicate with each other.

The separation of concerns between client and server in RESTful Web Services is of high importance. Clients interact with the server through standard operations on resources and they are completely agnostic over the underlying service logic and implementation. Web resources consist of any object (document, file, etc.) on the Web and are identified by their URL`s. The communication between a client and a service starts always with the client`s initiative. The client sends a request to the server asking for a resource and the server responds to this request. The state of the client does not affect the state of the server. This means, that every request contains the necessary information to interact with the service, as the service doesn't store any information on previous requests. Also, in order for their communication the client needs to know what operation to use on the server to retrieve the requested resource. These operations, are HTTP methods and the most common between them are the GET, POST, PUT and DELETE method. As well as the operation, the client also needs to know what header to use inside the request message and the path that leads to the resource.

After the Client-Server and Statelessness, next is the Cache constraint. The Cache constraint requires the data sent from the server to be either *cacheable* or *non-cacheable*. The first means that the client is allowed to store the data from a server response and the second does not permit their storage. The need for data storage is to prevent repeated and unnecessary requests to the server. It is obvious that this constraint improves any Web Service in terms of performance as it reduces the work load of requests a server has to manage. Next is the Layered constraint. A layered system is organized hierarchically, each layer providing services to the layer above it and using services of the layer below it. Layered-client-server adds proxy and gateway components to the client-server style. These additional mediator components can be added in multiple layers to add features like load balancing and security checking to the system. Another constraint is the Code-On-Demand. Although it is optional, it offers the opportunity to the client to download executable code from the server.

The last and most characteristic constraint that make Web Services RESTful is that of Uniform Interface. The uniform interface constraint is fundamental to the design of any RESTful system. The Uniform Interface constraints that exist in REST architecture are four. The first constraint dictates that individual resources are identified in requests and it is called *resource identification in requests*. Second is the *resource manipulation through representation* which allows the client to modify a resource given that the server permits it. Next is the *self-descriptive messages*, which means that messages to and from the server must include all the necessary information to be efficiently processed. The last interface constraint is that of *HATEOAS*. This constraint dictates that a server must provide all the available actions and resources to the client`s disposal through hyperlinks.

2.3 OpenAPI Specification

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

Figure 2.2 illustrates the structure of an OAS 3.0 service description. It comprises of many parts (objects). Each object has a list of properties which can be objects as well. This way, are linked to each other. The Info object

[6] https://www.w3.org/TR/rdf-sparql-query/

provides non-functional information such as the name of the service, service provider, license information and terms of the service. The Servers object provides information on where the API's servers are located. The Servers object specifies the base-path (the part of the URL that appears before the endpoint) of an API request. There are also variables that can be populated at run-time. Servers can be defined for different operations (i.e., a Servers object can be added as property in Path object of an Operation object). These locally declared servers will override the base (i.e., global) servers.

The Security requirement object lists the security schemes that the service applies to execute an operation. Its name must correspond to a security scheme declared in Security Schemes under the Components object. The specification offers support for basic HTTP authentication, API keys, OAuth2 common flows or grants (i.e., ways of retrieving an access token) and OpenID Connect. The Paths object describes the available operations (i.e., HTTP methods) and contains the relative paths for the service endpoints (which is appended to a server URL in order to construct the full URL of an operation).

Components object holds a set of reusable objects which can be responses, parameters, schemas, request bodies and more. Schemas object define data structures that are used to describe the request and response messages. A Schema object can be a primitive (string, integer), an array or a model. A Schema object may also have properties of its own accord (i.e., externalDocs) and properties supporting model composition and polymorphism.

The Responses object describe the expected responses of an operation, by mapping them to a specific HTTP status code. A response object defines the message content, as well as HTTP headers that a response may contain. Parameters object describes parameters that operations use. The specification, categorizes parameters into:

- ¬ 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.

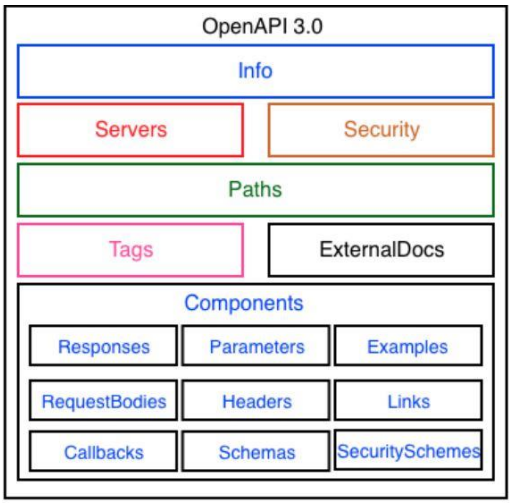


Figure 2.2: OpenAPI Document Structure

2.4 Hydra Core Vocabulary

Hydra[7] is a set of technologies that simplify the development of interoperable, hypermedia-driver 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 purpose 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. 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 contributing to the separation of concerns between client and server.

In the center of the vocabulary (Figure 2.3) is the ApiDocument class which builds the foundation for the description of a Web API. Hydra, describes an API by giving it a tittle, a short description, and documenting its

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

server-defined main entry point (Entry Point), all the operations (Operation) as well as the entities (Classes) and their properties (Properies). Also, the classes known to be supported by the Web API and all the addition information about status codes from response objects can be documented.

Typically, in Web services, a client decides whether to follow a link or not based on the link relation which defines its semantics. The Resource class is used to inform a client that an IRI is dereferenceable, meaning that when an IRI is accessed a representation of a resource is retrieved. This allows a client to distinguish Linked Data from IRIs that are used exclusively as identifiers. Similarly, the Link class is used in order to define properties whose properties are known to be dereferenceable IRIs.

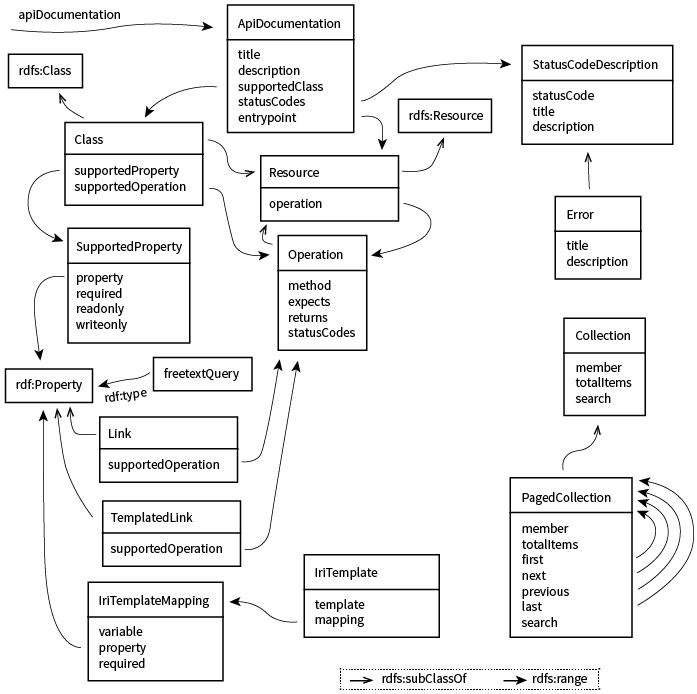


Figure 2.3: Hydra Core Vocabulary

However, in some cases the server cannot create links although they are essential for interacting with the Web service. For example, in order to query a service a link may contain parameters that a client must fill at runtime. Hydra provides us with the IriTemplate class to handle these cases. An IriTemplate describes a template and a number of mappings. An IriTemplateMapping maps a variable in the IRI template to a property and may optionally specify whether that variable is required or not. To better understand this, the example in Figure 2.4 represents a description of an IRI Template. The variable *lastname* maps to the property *givenName* from Schema.org vocabulary. Utilizing this information, the client can fully understand the meaning of variables and generate a complete URI.

```
{
    "@context": "http://www.w3c.org/ns/hydra/context.jsonld",
    "@type": "IriTemplate",
    "template": "http://api.example.com/users{?lastname}",
    "mapping": [
        {
            "@type": "IriTemplateMapping",
            "variable": "lastname",
            "property": "schema.org/givenName",
            "required": true
        }
    ]
}
```

Figure 2.4: Description of an IRI Template

In addition to IriTemplate, another equally important Hydra class is the Operation class. This class contains all the necessary information in order for an HTTP request from the client to be valid. The method property specifies the HTTP method, while the *expects* and *returns* properties define the expected data in request and response messages. Furthermore, the *status-Code* property specifies a *StatusCodeDescription* that provides a developer with information regarding what to expect when invoking an operation.

Another interesting feature of Hydra is presented via the SupportedProperty Class. Hydra, as well as using classes to describe information expected or returned by an operation, it also defines a new concept which describes the properties of a class. More specifically, it is possible to define a property as *required*, *read-only* or *write-only* depending on which class is associated with. For example, in Figure 2.5 we can define the property which is supported by a class as required (means that its presence is obligatory in the request), as readable, (means that the client cannot see its value) and as writeable (means the client cannot change the property`s value). In a similar manner, Hydra introduces the SupportedOperation property which makes it possible to define operations supported by all instances of a class.

```
{
    "@context": "http://www.w3c.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 2.5: Hydra SupportedProperty Class

2.5 SHACL

SHACL[8] stands for Shapes Constraint Language and is an RDF vocabulary that is used to describe and validate the structure of RDF data. The RDF data, are validated against a set of conditions which are expressed in SHACL as shapes. The RDF graphs used for providing the constraints are called *shapes graphs* and the RDF graph that contains the data to be validated it is called *data graph*. The SHACL processor accepts as an input the data graph and the shapes graphs and generates validation reports based on constraints. In SHACL, the nodes of the shapes graph are divided in two major categories. First is the node shape which contains constraints about a given focus node. Next, is the property shape which contains constraints about a property and the values of a path for a node. A node shape contains *targets* which specify which nodes in the data graph must conform to a shape and constraint components which determine how to validate a node. For example, in Figure 2.6 a node shape is presented by the shapes graph and three nodes from the data graph in order to be validated. Due to the constraints that exist in the node shape, only two nodes from the data graph are valid. The first invalid node, *(:bob)* does not have a *name* property but a *firstName* property. The second invalid node *(:alice)*, on the *email*

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

property has as a string type value and not an IRI one.

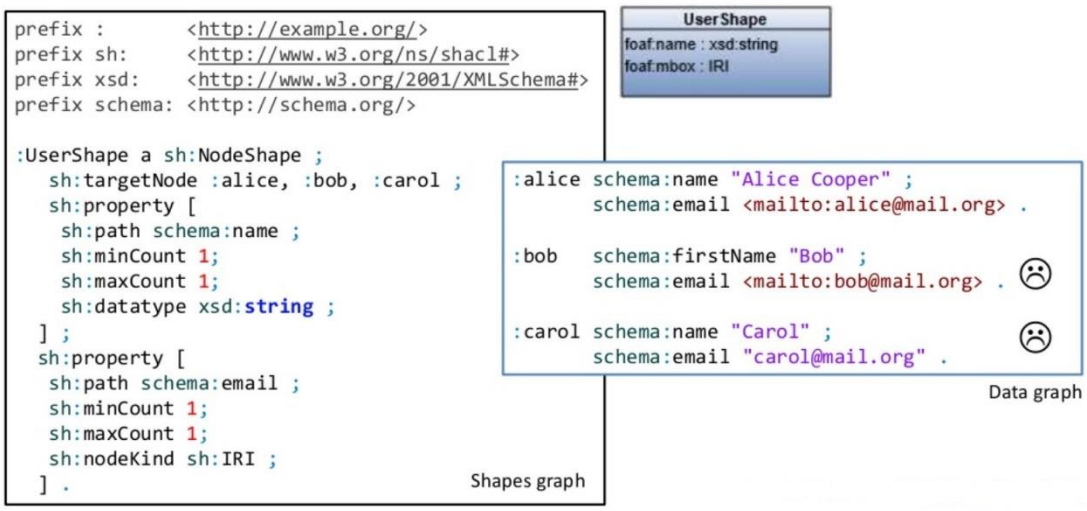


Figure 2.6: SHACL validation, sh:targetNode

Except for the sh:targetNode, which specifies directly the nodes to be validated, there is also the sh:targetClass. This is encountered in the majority of times as it signifies that all the nodes of a given type need to conform with a particular shape. In Figure 2.7, an example of the sh:targetClass is presented. Here, the three nodes of the data graph are all of the same type *(:User),* meaning that all are instances of the User Class. The "*:UserShape*" can point to these nodes for validation, with sh:targetClass. The last two nodes in the data graph remain invalid for the same reasons as before.

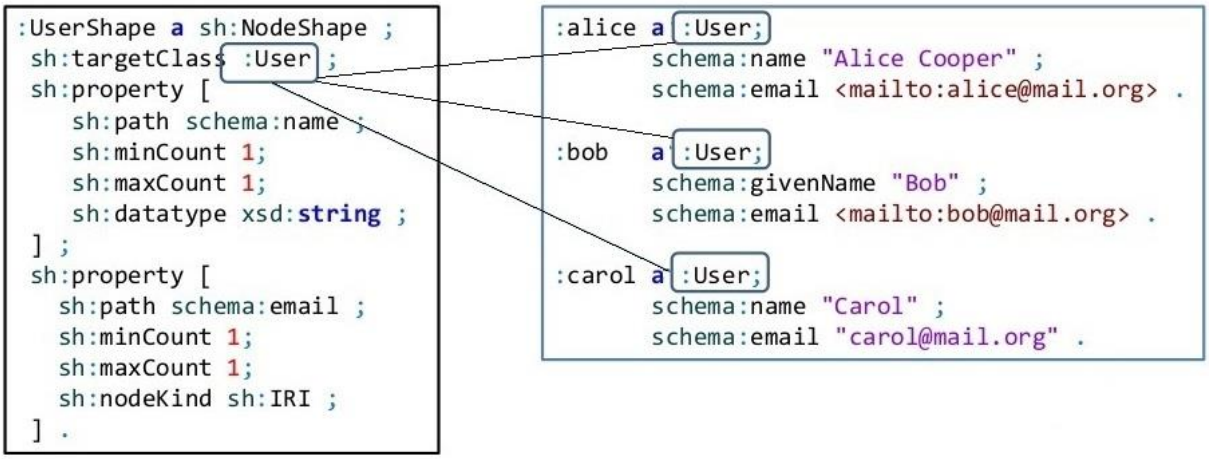


Figure 2.7: SHACL validation, sh:targetClass

It is necessary to distinguish between the use of SHACL and the use of OWL. Some of the major differences about their usage is that even though OWL has restrictions, they are not designed to validate data, they are designed to allow inference of data. Another difference is that OWL adopts the open world assumption. What this means, is that if a class of *Person* is defined with properties, *surname, name* and *date of birth* and a node from the data graph has only two of these values, we cannot claim that this node is invalid. In this case we can say that a property is missing but not that is invalid. On the other hand, SHACL is close world assumption which means that for the current example the node would be invalid.

Chapter 3

Handling Schema Objects

3.1 Introduction

The Schema Object allows the definition of input and output data types. Schema Objects are placed under the Components section and can be referenced by every other Object of an OpenAPI document. They are of high significance due to their frequent appearance in core elements of OpenAPI descriptions such as Responses, Request Bodies, Parameters etc. A formal procedure for mapping of Schema Objects to the OpenAPI ontology is presented below.

OpenAPI v3.0 introduced keywords like *anyOf, allOf, oneOf* and "*not"* and allowed for the creation of more complex schemas with various data types. These properties support and express concepts like model composition and polymorphism. These new features in conjunction with the usage of semantic annotations brings in more complexity to our algorithm as well as more capabilities. In this chapter all of these issues are addressed. Schema Objects need to be semantically enriched in order eliminate ambiguities [2]. In OpenAPI v3.0, Schema Objects are enhanced with additional properties and offered much more potential for further clarification of their intended meaning.

3.2 Schema Object in OpenAPI v3 Ontology

First, we explain how Schema Objects and their structural parts are expressed in the ontology. Schema Objects are expressed as classes, object and data properties using SHACL vocabulary[9]. SHACL is an RDF vocabulary that can be used to describe and validate the structure of RDF data, similarly to XML-Schema or JSON Schema. SHACL can be used to define classes together with constraints on their properties. It provides built-in types of constraints (e.g., cardinality: *minCount*/*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 1 shows the direct mapping of Schema Object properties with the SHACL vocabulary.

Table 3.1: Mapping OpenAPI Schema Object properties to SHACL

Schema Object Property	SHACL Property
exclusiveMaximum	sh:exclusiveMaximum if OpenAPI exclusiveMaximum is true
exclusiveMinimum	sh:exclusiveMinimum if OpenAPI exclusiveMinimum is true
maxLength	sh:maxLength
minLength	sh:minLength
pattern	sh:pattern
maxItems	sh:maxCount
minItems	sh:minCount
enum	sh:in
allOf	sh:and
oneOf	sh:xone
anyOf	sh:or

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

not	sh:not
default	sh:defaultValue

A Schema Object of OpenAPI v3.0 is mapped in Shape Class in the ontology. A Shape Class is distinguished into NodeShape Class and PropertyShape Class. A shape determines how to validate a focus node (a node from the data graph) based on the values of properties and other characteristics of the focus node. The two types of shapes are defined by the SHACL Core language. Shapes about the focus node itself are called node shapes and shapes about the values of a particular property for the focus node are called property shapes. In OpenAPI ontology, the NodeShape class represents the classes that describe the models of an OpenAPI v3.0 description (schemas) and PropertyShape class represents the properties of a class, their datatype and restrictions.

Listing 3.1 shows the Error model from the Swagger Petstore example. The model is of type object and contains two required properties code and message. Each one of the properties contains information about the type and format constrains that follows. Listing 3.2 shows how the Error model of Listing 3.1 is represented in the OpenAPI ontology. The model will be translated into an instance of the NodeShape Class. Inside the focus node, the *rdfs:label* predicate is used to provide human-readable version of the resource`s name. The *sh:property* declares that the specified node has one or more property shapes (instances of the PropertyShape class). Also each value of the *sh:property*, according to SHACL Core language, is a well-formed property shape. The predicate *sh:targetClass* will be explained in detail later on.

```yaml
Error:
  type: object
  required:
    - code
    - message
  properties:
    code:
      type: integer
      format: int32
    message:
      type: string
```

Listing 3.1: OAS Model Swagger Petstore *Error* Schema Object

```
<ErrorNodeShape>  a     sh:NodeShape ;
        rdfs:label      "ErrorNodeShape" ;
        sh:property     <Error_messagePropertyShape> , <Error_codePropertyShape> ;
        sh:targetClass  <Error> .

<Error>  a  owl:Class .
```

Listing 3.2: Representation of a Schema Object in OpenAPI ontology (*ErrorNodeShape*)

Listing 3.3 shows how both properties of Error model, code and message, are represented in the OpenAPI ontology. Each one will become an instance of PropertyShape class. The values of *sh:datatype* come from the type and format of each property. In particular a property of *type* integer with *format* "*int32*" will give an "*xsd:int*" value and one with *format* "*int64*" will give an "*xsd:long*" value. Continuing, predicate *openapi:name* corresponds to the OpenAPI property name inside the Schema Object and finally *sh:path* has a value of an *rdf:Property* instance. In particular, *sh:path* predicate points at the URI of the property that is being restricted.

```
<Error_messagePropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      “Error_messagePropertyShape” ;
        openapi:name    “message” ;
        sh:datatype     xsd:string ;
        sh:path         <Error_message> .

<Error_message>  a  rdf:Property .

<Error_codePropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      “Error_codePropertyShape” ;
        openapi:name    “code” ;
        sh:datatype     xsd:int ;
        sh:path         <Error_code> .

<Error_code>  a  rdf:Property .
```

Listing 3.3: Representation of Schema Object properties in OpenAPI ontology (*Error_messagePropertyShape*, *Error_codePropertyShape*)

In OpenAPI descriptions and services, a variety of properties are defined. As properties are declared, conflicts of names may often occur because the same property may appear many times in the document with the same or different meaning. The algorithm handles these issues by prefixing all property shapes with the name of the schema object that the property belongs to, hence the *Error_codePropertyShape* which comes from the property code in the error schema object. The same approach is followed with *rdf:Property* instances, (i.e. *Error_code*, *Error_message*) which are used as objects to the *sh:path* predicates.

Another great example is that of *Dog* Schema object. The *Dog* schema in Listing 3.4 has two properties, “*bark*” and “*packSize*” but only “*packSize*” is required. The “*bark*” property is of type boolean and “*packSize*” is of type integer and as seen in the description value, expresses the size of the dog pack. The example in Listing 3.5 is the part of the ontology that the *Dog* Schema object is mapped to. Inside the ontology, a node shape is created for the Schema object with two property shapes, one for each property. The *openapi:description* predicate in *Dog_packSizePropertyShape* contains the value of the OpenAPI property “*description*”. Inside *Dog_barkPropertyShape,* the value of *sh:datatype* is *xsd:boolean* which maps to “*boolean”*. Lastly, although only the “*packSize*” property is required, both properties need to be mapped in property shapes and both individuals (of the property shapes) need to be included in Dog node shape.

```
Dog:
  type: object
  properties:
    bark:
      type: boolean
    packSize:
      type: integer
      format: int64
      description: the size of the pack the dog is from
      default: 0
  required:
    - packSize
```

Listing 3.4: OAS Model Swagger Petstore *Dog* Schema Object

```
<DogNodeShape>  a          sh:NodeShape ;
        rdfs:label         "DogNodeShape" ;
        sh:property        <Dog_packSizePropertyShape> , <Dog_barkPropertyShape> ;
        sh:targetClass     <Dog> .

<Dog>  a  owl:Class .

<Dog_packSizePropertyShape>
        a                       sh:PropertyShape ;
        rdfs:label              "Dog_packSizePropertyShape" ;
        openapi:description     "the size of the pack the dog is from" ;
        openapi:name            "packSize" ;
        sh:datatype             xsd:long ;
        sh:path                 <Dog_packSize> .

<Dog_packSize>  a  rdf:Property .

<Dog_barkPropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      "Dog_barkPropertyShape" ;
        openapi:name    "bark" ;
        sh:datatype     xsd:boolean ;
        sh:path         <Dog_bark> .

<Dog_bark>  a  rdf:Property .
```

Listing 3.5: Representation of *Dog* Schema Object and its properties in the ontology (*DogNodeShape*), (*Dog_packSizePropertyShape*), (*Dog_barkPropertyShape*)

It is clear that Instances of PropertyShape Class are created from the name of the Schema Object they belong to appended with the name of the property and the string "*PropertyShape*" (i.e., *Error_codePropertyShape*, *Dog_barkPropertyShape* etc.). These Property Shapes are then connected to the Node Shape through the *sh:property* predicate. Continuing, the Property Shapes contain the *sh:path* predicate which takes as an object, an RDF property. This RDF property is created from the name of the Schema Object appended with the name of the property (i.e., *Error_code*, *Error_message*, *Dog_packSize*, *Dog_bark*). This approach is followed in order to avoid duplication between properties with the same name which are defined in different Schema Objects. This is better explained in the later sections of this chapter where we describe our algorithm.

3.3 Semantic Annotations

OpenAPI service documents often obtain elements that share the same semantics. For a human it might be easy to infer these semantic similarities but a machine needs a formal description. This led to the creation of our semantic annotations in the form of extension properties inside the OpenAPI service document. Table 3.2 summarizes the extension properties, their scope and their meaning. In this chapter we are mainly interested in extensions regarding schema objects and more specific in *x-refersTo, x-kindOf* and *x-mapsTo*.

Table 3.2: OAS extension properties for semantic annotations

Property	Applies to	Meaning
x-refersTo	Schema Object	The concept in a **semantic** model that describes an OpenAPI element.
x-kindOf	Schema Object	A specialization between an OpenAPI element and a concept in a **semantic** model.
x-mapsTo	Schema Object	An OpenAPI element which is **semantically** similar with another OpenAPI 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

The properties that apply to Schema Objects are, *x-refersTo, x-kindOf, x-mapsTo* and *x-collectionOn.* From these properties, only the first three are analyzed in this chapter because they are the only ones that semantically influence a Schema Object. The *x-collectionOn* describes a collection over a specific property of the Schema Object. The handling of this extension property is explained later in this chapter along with the functions that handle extension properties in a Schema Object. The *x-onResource* property is found in a Tag Object and refers to a Schema Object that describes a resource. The *x-operationType* is found in an Operation Object and clarifies the type of an operation. Both these properties, *x-onResource* and *x-operationType,* are described in the next chapter where we present the whole instantiation algorithm.

The *x-refersTo* property is responsible for associating OpenAPI elements and concepts in a semantic model. Listing 3.6 presents the usage of *x-refersTo* in the Pet model. For demonstration purposes we use the example domain "*https://example.com/ontology*". In this case it associates the Schema Object with the "*Pet*" class and the "*id*" with the "*Id*" class inside the example domain "*https://example.com/ontology*". As shown in the same example the property *x-kindOf* has a slightly different meaning. Some models have a narrower meaning and this means that whenever this extension property is used, will denote the model as a subclass of that semantic concept. In our example is implied that the *Dog* model is a subclass of the *Animal* class. Finally, the extension property *x-mapsTo* is destined to define schema object elements that share the same semantics. In Listing 3.6, *x-mapsTo* property is used to dictate that "*SecondPet*" Schema Object refers to "*Pet*" and "*SecondId*" to "*Pet.id*".

```
Pet:
  x-refersTo: https://example.com/ontology/Pet
  type: object
  required:
    - id
  properties:
    id:
      x-refersTo: https://example.com/ontology/Id
      type: integer
      format: int64

Dog:
  x-kindOf: https://example.com/ontology/Animal
  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

SecondPet:
  x-mapsTo: '#/components/schemas/Pet'
  type: object
  required:
    -secondId
  properties:
    secondId:
      x-mapsTo: '#/components/schemas/Pet.id'
      type: integer
      format: int64
```

Listing 3.6: OAS model *x-refersTo*, *x-kindOf* , *x-mapsTo* usage example

3.4 OpenAPI Keywords

In OpenAPI v3.0 the definition of Schema Objects is enhanced with additional properties. Some of these properties *allOf* (also existed in OpenAPI v2.0), *anyOf*, *oneOf*, and "*not*". The purpose of these keywords is to allow for the creation of more complex schemas and give greater flexibility to users. These four keywords are of high importance regarding model definitions because they are responsible for model composition and polymorphism. Also, this work capitalizes of this opportunity and implements the concept of inheritance between classes on the ontology.

Very often APIs have schemas that share common properties. Instead of defining these properties for each schema repeatedly, it is possible to describe the schemas as a composition of the common property set and schema-specific properties. Using *allOf* keyword in Listing 3.7, the *ExtendedErrorModel* schema includes its own set of properties as well as properties inherited from the *BasicErrorModel* schema. In order to validate data against the combined model, a client (or a server) needs to validate against each sub-model it consists of (Listing 3.8).

```
components:
  schemas:
    ErrorModel:
      type: object
      required:
      - message
      - code
      properties:
        message:
          type: string
        code:
          type: integer
          minimum: 100
          maximum: 600

    ExtendedErrorModel:
      allOf:
      - $ref: '#/components/schemas/ErrorModel'
      - type: object
        required:
```

```
        - rootCause
        properties:
          rootCause:
            type: string
```

Listing 3.7: OpenAPI model inheritance example

```
# Valid
{
  "message": "Page Not Found",
  "code": 404,
  "rootCause": "not responding"
}

# Invalid (the payload is missing the “message” property although it is a
  required one due to allOf)
{
  "code": 500
  "rootCause": "interval server"
}
```

Listing 3.8: *allOf* validation example (JSON data)

The keywords *oneOf* and *anyOf* are mostly used to describe OpenAPI elements that can take one or more of several alternative schemas. In the following example (Listing 3.9) we present two similar responses. In the first case, where the keyword *oneOf* is used, the data needs to be validated with exactly one of the listed schemas. The data is invalid when matches with more than one of the listed schemas. In contrast to *oneOf*, the *anyOf* keyword is used to validate data against either of the listed schemas, often with all of them simultaneously. By accepting several alternative schemas, the concept of polymorphism is showcased in OpenAPI v3.0. In Listing 3.10, a number of JSON data are presented to better understand the validation against these two keywords.

```
components:
  responses:
    sampleResponse_oneOf:
      content:
        application/json:
          schema:
            oneOf:
              - $ref: '#/components/schemas/Cat'
              - $ref: '#/components/schemas/Dog'
    sampleResponse_anyOf:
      content:
        application/json:
          schema:
            anyOf:
              - $ref: '#/components/schemas/Cat'
              - $ref: '#/components/schemas/Dog'
  …
  schemas:
    Dog:
      type: object
      properties:
        bark:
          type: boolean
        breed:
          type: string
          enum: ['Dingo', 'Husky', 'Retriever', 'Shepherd']
    Cat:
      type: object
```

```
      properties:
        hunts:
          type: boolean
        age:
          type: integer
```

Listing 3.9: OpenAPI model polymorphism example

```
# Valid (the payload is valid against the Dog schema).
{
  "bark": true,
  "breed": "Dingo"
}

# Invalid (the payload is not valid against neither Cat nor Dog schema).
{
  "bark": true,
  "hunts": true
}

# Invalid for oneOf (the payload conforms with both schemas and should conform
  with only one).
# Valid for anyOf (the payload conforms with both schemas).
{
  "bark": true,
  "hunts": true,
  "breed": "Husky",
  "age": 3
}
```

Listing 3.10: *oneOf*, *anyOf* validation example (JSON data)

The Listing 3.10 gives us an insight of the keywords *anyOf, oneOf.* The first example is valid for both Response Objects (*SampleResponse_anyOf* and *SampleResponse_oneOf*). This is because the payload contains the properties of *Dog* Schema Object. The next example instead, is not valid. This payload contains one property of each schema and therefore does not meet the conditions of either *anyOf* (one or more schemas must conform) or *oneOf* (only one schema must conform). The third and final example is a little more complicated. This payload is valid for *anyOf* (valid for the Response Object *SampleResponse_anyOf*) but is invalid for *anyOf* (invalid for the Response Object *SampleResponse_oneOf*). This is because the payload conforms simultaneously to both *Cat* and *Dog* Schema Objects.

Last but not least the "*not*" keyword is used to modify a schema and make it more specific. It doesn`t help to combine any schemas but it declares which type of value is not accepted for a specific property. In Listing 3.11, the "*pet_type*" value can be of any type except integer (that is, it should be an array, boolean, number, object or string).

```
components:
  schemas:
    Pet:
      type: object
      properties:
        pet_type:
          not:
            type: integer
      required:
        - pet_type
```

Listing 3.11: OpenAPI model "*not*" keyword example

```
# Valid.
{
  "pet_type": "Cat"
}

# Invalid (the payload should not be an integer).
{
  "pet_type": 11
}
```

Listing 3.12: "*not*" validation example (JSON data)

So far, the Schema Objects are translated to an ontology using SHACL as mentioned in previous paragraphs. SHACL provides us with four logical constraints. These components as well as each of the corresponding property inside the ontology are shown in Table 3.4. In order to understand the use of these components, we will also present the usage examples from the SHACL documentation.

Table 3.4: OpenAPI keywords corresponding components and ontology properties

OpenAPI Keyword	Ontology Property
allOf	sh:and
anyOf	sh:or
oneOf	sh:xone
not	sh:not

Regarding the logical component *sh:and,* its value is a SHACL list of shapes and specifies the condition that each value node conforms to all provided shapes inside that list. This is compatible to conjunction and the logical "*and*" operator. In Listing 3.13, *the* example illustrates the use of *sh:and* in a shape to specify the condition that certain focus nodes have exactly one value of *ex:property*. Inside the bold text, the *sh:and* value is a list containing two blank nodes. A blank node is an unnamed node, usually inside square brackets. The first node dictates whatever value is accepted from the example data graph must have at least one *ex:property*. In the same way, the second node dictates that it accepts only values that have at most one *ex:property*. Consequently, having both nodes in the *sh:and* list declares that *ex:ExampleAndShape* will accept values with exactly one *ex:property*.

```
ex:ExampleAndShape
    a sh:NodeShape ;
    sh:targetNode ex:Person ;
     sh:and (
         [
             sh:path ex:property ;
             sh:minCount 1 ;
```

```
        ]
        [
            sh:path ex:property ;
            sh:maxCount 1 ;
        ]
    ) .
```

Listing 3.13: SHACL *sh:and* example

Similarly with *sh:and*, *sh:or* takes as its value a SHACL list of shapes. It is comparable to disjunction and the logical "*or*" operator, and dictates that each value node conforms to at least one of the provided shapes. The example is Listing 3.14 illustrates the use of *sh:or* in a shape to specify the condition that certain focus nodes have at least one value of *ex:firstName* or at least on value of *ex:givenName*.

```
ex:OrConstraintExampleShape
    a sh:NodeShape ;
    sh:targetNode ex:Person ;
    sh:or (
        [
            sh:path ex:firstName ;
            sh:minCount 1 ;
        ]
        [
            sh:path ex:givenName ;
            sh:minCount 1 ;
        ]
    ) .
```

Listing 3.14: SHACL *sh:or* example

The last component that accepts a list of shapes as its value is *sh:xone*. It specifies that each value node conforms to exactly one of the provided shapes in that list. The example in Listing 3.15 specifies the condition that certain focus nodes must either have a value for *ex:fullName* or values for *ex:firstName* and *ex:lastName*, but not both. Inside the *sh:xone* list there exist two blank nodes, the first accepts a value node with at least one property of *ex:fullName*, and the next accepts also a value node with at least one *ex:firstName* property and at least one *ex:lastName* property. Therefore, the value node must conform to either one of the shapes, but not both.

```
Ex:XoneConstraintExampleShape
    a sh:NodeShape ;
    sh:targetClass ex:Person ;
    sh:xone (
        [
            sh:property [
                sh:path ex:fullName ;
                sh:minCount 1 ;
            ]
        ]
        [
            sh:property [
```

```
                sh:path ex:firstName ;
                sh:minCount 1 ;
            ] ;
            sh:property [
                sh:path ex:lastName ;
                sh:minCount 1 ;
            ]
        ]
    ) .
```

Listing 3.15: SHACL *sh:xone* example

In contrast with the other components, the *sh:not* does not have a list of shapes as its value. It accepts one shape, and specifies the condition that each value node cannot conform to that shape. It is comparable to negation and the logical "*not*" operator. The following example, Listing 3.16 illustrates the use of *sh:not* in a shape to specify the condition that certain focus nodes cannot have any value of *ex:property*.

```
ex:NotExampleShape
    a sh:NodeShape ;
    sh:targetNode ex:Person ;
    sh:not [
        a sh:PropertyShape ;
        sh:path ex:property ;
        sh:minCount 1 ;
    ] .
```

Listing 3.16: SHACL *sh:not* example

3.5 Handling Schema Objects without Semantic Annotations

A previous work has shown that in order to eliminate ambiguates, OpenAPI properties must be semantically annotated and mapped to a semantic model. Our approach is that every Schema object and every property (of the Schema object) acquires a semantic value. Listing 3.17, is a simple version of a Schema object that obtains property "*id*". Pet Schema Object is of type "*object*" and therefore, as shown in previous paragraphs, it will be mapped with a node shape. The node shape will be the *PetNodeShape.* Also, property "*id*" will be mapped to the property shape *Pet_idPropertyShape*. The semantic value for each of our current shapes is presented in bold in Listing 3.18.

```
Pet:
  type: object
  required:
    - id
  properties:
    id:
      type: integer
      format: int64
```

Listing 3.17: *Pet* Schema Object

```
<PetNodeShape>  a        sh:NodeShape ;
        rdfs:label       "PetNodeShape" ;
        sh:property      <Pet_idPropertyShape> ;
        sh:targetClass   <Pet> .

<Pet>   a        owl:Class .


<Pet_idPropertyShape>
        a             sh:PropertyShape ;
        rdfs:label    "Pet_idPropertyShape" ;
        openapi:name  "id" ;
        sh:datatype   xsd:long ;
        sh:path       <Pet_id> .

<Pet_id>  a      rdf:Property .
```

Listing 3.18: *Pet* Node Shape, *Pet_id* Property Shape

For the node shape, we create a class of *Pet* (given by the name of the schema) and it is used as an object to the *sh:targetClass* predicate. According to SHACL, the *sh:targetClass*[10] as well as all target declarations, is used to produce focus nodes for a shape. A focus node is a node that is validated against the shape where it is used. In our current example, by creating a *Pet* class and using it as an object to the *sh:targetClass* predicate, we declare two things. First, we justify the *Pet* class as a semantic value and secondly each node that is an instance of *Pet* class will conform to this node shape. As a sidenote for *sh:targetClass* and focus nodes we cite the following example. In this example, only *ex:Dog* and *ex:Cat* are focus nodes.

```
ex:PetNodeShape
    a sh:NodeShape ;
    sh:targetClass <Pet> .

ex:Dog a <Pet> .
ex:Cat a <Pet> .
ex:NewYork a <Place> .
```

Listing 3.19: *sh:targetClass* usage

For the property shape, we create a property called *Pet_id*. In property shapes we use the predicate *sh:path* to signify which property is used in the current property shape. By definition, the *sh:path* predicate takes only one value. Also, *rdf:Property*[11] is the class of RDF properties and an instance of *rdfs:Class*.

In Listing 3.20 and 3.21 we present the algorithms for node and property shape creation respectively. The "*ontModel*" argument in both functions represents the ontology model that is created for the Web Service and where the algorithm writes all the triples. In function *CreateNodeShape* (Listing 3.20), the "*schemaName*" argument contains the name of the schema so for the *Pet* Schema Object, (Listing 3.17) its value is "*Pet*". The "*schemaObject*" argument contains the body of the schema (type, property schemas, description etc.) and the "*schemas*" argument is a list containing all the schemas that are encountered in the current OpenAPI description

[10] https://www.w3.org/TR/shacl/#targetClass
[11] https://www.w3.org/TR/rdf-schema/#ch_property

document, either Schema Objects or property schemas (Pet, id etc.). In order to instantiate the *Pet* object to the *Pet* class, the function is called as *createNodeShape (ontModel, Pet, petBodyObject, schemas)*.

In function *createPropertyShape*, the "*ontModel*" and "*schemas*" arguments remain the same. Also, the "*schemaName*" contains the name of the schema but in this case the name of the schema property. Therefore, regarding *Pet* Schema Object, (Listing 3.17) its value is "*id*". The "*schemaObject*" argument, contains the body of the property schema (type, format etc.). Finally, the "*ownerName*" argument (Listing 3.21), represents the name of the Schema Object which the schema property belongs to (i.e., Pet). In order to instantiate the "*id*" property schema, the function is called as *createPropertyShape (ontModel, Pet, id, idBodyObject, schemas).*

These two functions are responsible for parsing Schema Objects and their properties and translating them into shapes. Additionally, they are responsible for handling different situations (annotations, OpenAPI keywords, inheritance, semantic validation etc.) which are going to be discussed in the following sections. Therefore, the majority of the following sections contain these two functions. Each time, the corresponding part of each of these functions is going to be presented. In this section, the simple case of handling Schema Objects and property schemas without any annotation or OpenAPI v3.0 keywords is showcased. The Listings 3.20 and 3.21 show how the functions work together to fully map a Schema Object and its properties. Starting from *createNodeShape,* we create an ontology class from the "*schemaName*" argument and we connect it to the node shape. Then, for every property schema, we call *createPropertyShape* and we map the returned property shape Individuals with the node shape Individual. In *createPropertyShape,* after making an ontology property from the argument "*schemaName*" we connect it to the property shape. Finally, we extract and map all the properties (e.g., format, description, name etc.) to the property shape, before returning it.

function createNodeShape (ontModel, schemaName, schemaObject, schemas)
- Create a NodeShape from schemaName + "NodeShape".
- Create an owl class from the schemaName.
- Add the class to the ontology (ontModel).
- Add the tripe: NodeShape sh:targetClass class .
- For every property schema inside the schemaObject
  - Call createPropertyShape and return the property shape Individual
  - Map the property shape Individual to the current node shape Individual: NodeShape sh:property PropertyShape
- Return NodeShape Individual.

Listing 3.20: NodeShape algorithm, no annotation handling

function createPropertyShape (ontModel, ownerName, schemaName, schemaObject, schemas)
- Create a PropertyShape from ownerName + "_" + schemaName + "PropertyShape".
- Create a resource property from the ownerName + "_"+ schemaName.
- Add it to the ontology (ontModel) as an ontology property.
- Add the triple: PropertyShape sh:path property.
- Scan schemaObject (the body of the property schema) for the rest properties (e.g., format) and add them to the PropertyShape by writing triples with PropertyShape as Subject and the corresponding Predicate – Object, according to each property.
- Return PropertyShape Individual.

Listing 3.21: PropertyShape algorithm, no annotation handling

So far, we showcased that a Schema Object and its property schema that are not semantically annotated, will acquire an ontology class and an ontology property respectively. This means that the node shape of the Schema Object and the property shape of the property schema will become semantically enriched. This offers some benefits. The first benefit is that a user does not necessarily need to semantically annotate a Schema Object or a property schema. Consequently, we introduce new semantic values. The second benefit is that the semantic values of a Web Service description (i.e., classes and properties) can be used to semantically annotate another Web Service description. This is presented in chapter 5 where we semantically annotate a custom Web Service description with ontology classes created for a different Web Service.

3.6 Semantically Annotated Schema Objects

The semantic annotations that apply in Schema Objects are *x-refersTo, x-kindOf* and *x-mapsTo*. With the new approach, new combinations emerge between semantic values in Schema Objects. In Listing 3.22 both *Pet* Schema object and its property "*id*" are semantically annotated with the use of *x-refersTo*. The corresponding shapes in the ontology are presented in Listing 3.23. Both shapes (property and node) are associated with the semantic value inside their extension property (*x-refersTo*). The algorithm in Listing 3.24 and Listing 3.25 extracts the referred string and then assigns it as an object to *sh:targetClass* or *sh:path*.

```
Pet:
  x-refersTo: https://example.com/ontology/Pet
  type: object
  required:
    - id
  properties:
    id:
      x-refersTo: https://example.com/ontology/Id
      type: integer
      format: int64
```

Listing 3.22: *Pet* Schema object, *x-refersTo*

```
<PetNodeShape>  a       sh:NodeShape ;
          rdfs:label            "PetNodeShape" ;
          sh:property           <Pet_idPropertyShape> ;
          sh:targetClass        <https://example.com/ontology/Pet> .

<Pet_idPropertyShape>
        a                   sh:PropertyShape ;
        rdfs:label          "Pet_idPropertyShape" ;
        openapi:name        "id" ;
        sh:datatype         xsd:long ;
        sh:path             <https://example.com/ontology/Id> .
```

Listing 3.23: *Pet* NodeShape and *Id* PropertyShape, *x-refersTo*

The handling of the *x-refersTo* extension property inside Schema Objects and property schemas falls to the same two functions. Both functions (Listing 3.24 and Listing 3.25) after detecting the extension property, they extract its value. Afterwards, they create an ontology class (in *createNodeShape* function) or an ontology property (in *createPropertyShape* function) and assign it to the current shape, continuing with the scan of the rest schema body. Finally, the functions return the Shape Individuals. The function calls for the current example are *createNodeShape (ontModel, Pet, petBodyObject, schemas)* and *createPropertyShape (ontModel, Pet, id, idBodyObject, schemas)*.

function createNodeShape (ontModel, schemaName, schemaObject, schemas)
- Create a NodeShape from schemaName + “NodeShape”.
- Scan schemaObject for the x-refersTo extension property.
- Extract string from the x-refersTo property.
- Create a class with the extracted string.
- Add the class to the ontology (ontModel).
- Add the tripe: NodeShape sh:targetClass class.
- For every property schema inside the schemaObject
  - Call createPropertyShape and return the property shape Individual
  - Map the property shape Individual to the current node shape Individual: NodeShape sh:property PropertyShape
- Return NodeShape Individual.

Listing 3.24: NodeShape algorithm, *x-refersTo* handling

function createPropertyShape (ontModel, ownerName, schemaName, schemaObject, schemas)
- Create a PropertyShape from ownerName + ”_” + schemaName + “PropertyShape”.
- Scan schemaObject (the body of the property schema) for the x-refersTo extension property.
- Extract string from the x-refersTo property.
- Create a property with the extracted string.
- Add the property to the ontology (ontModel).
- Add the triple: PropertyShape sh:path property .
- Scan schemaObject (the body of the property schema) for the rest properties and add them to the PropertyShape by writing triples with PropertyShape as Subject and the corresponding Predicate - Object according to each property.
- Return PropertyShape Individual.

Listing 3.25: PropertyShape algorithm, *x-refersTo* handling

When a Schema object has the extension property *x-kindOf*, the model (schema), it will become a subclass of the referred semantic value. Consequently, the *Pet* class (<*Pet*>) will become a subclass of *Pet* class inside the example domain “*https://example.com/ontology*” with the IRI “*https://example.com/ontology/Pet*”. Similarly, the *Pet_id* property will become a subproperty of the “*Id*” property with the IRI “*https://example.com/ontology/Id*”. The algorithm in Listings 3.29 and 3.30 creates the semantic values of each shape and then denotes it as a subclass or a subproperty.

```
Pet:
  x-kindOf: https://example.com/ontology/Pet
  type: object
  required:
    - id
  properties:
    id:
      x-kindOf: https://example.com/ontology/Id
      type: integer
      format: int64
```

Listing 3.26: *Pet* Schema object, x-kindOf

```
<PetNodeShape>  a          sh:NodeShape ;
        rdfs:label         "PetNodeShape" ;
        sh:property        <Pet_idPropertyShape> ;
        sh:targetClass     <Pet> .

<Pet>   a                 owl:Class ;
        rdfs:subClassOf   <https://example.com/ontology/Pet> .
```

Listing 3.27: *Pet* NodeShape, x-kindOf

```
<Pet_idPropertyShape>
        a                 sh:PropertyShape ;
        rdfs:label        "Pet_idPropertyShape" ;
        openapi:name      "id" ;
        sh:datatype       xsd:long ;
        sh:path           <Pet_id> .

<Pet_id>  a                          rdf:Property ;
        rdfs:subPropertyOf      <https://example.com/ontology/Id> .
```

Listing 3.28: *Id* PropertyShape, x-kindOf

The extension property *x-kindOf* is handled in a similar fashion as the *x-refersTo*. The functions (Listing 3.29 and Listing 3.30) detect the extension property and extract its value. Afterwards, they create an ontology class or an ontology property from this extracted value. Additionally, they create an ontology class or an ontology property from the "schemaName" argument. The value created from the "schemaName", will become a subclass or a subproperty to the value created from the extension property. This is the point where the handling of *x-refersTo* and *x-kindOf* differ. Both functions continue by scanning the rest schema body and mapping any additional encountered properties. In the end, the functions return the Shape Individuals. Again, the function calls do not differ from the previous examples, *createNodeShape (ontModel, Pet, petBodyObject, schemas)* and *createPropertyShape (ontModel, Pet, id, idBodyObject, schemas)*.

function createNodeShape (ontModel, schemaName, schemaObject, schemas)
- Create a NodeShape from schemaName + "NodeShape".
- Scan schemaObject for the x-kindOf extension property.
- Extract string from the x-kindOf property.
- Create a class A with the extracted string.
- Add class A to the ontology (ontModel).
- Create a class B from the schemaName.
- Add class B to the ontology (ontModel).
- Make class B a subclass of A (add the connection triple: B rdfs:subclassOf A to the ontology).
- Add the tripe: NodeShape sh:targetClass class B.
- For every property schema inside the schemaObject
  - Call createPropertyShape and return the property shape Individual
  - Map the property shape Individual to the current node shape Individual: NodeShape sh:property PropertyShape
- Return NodeShape Individual.

Listing 3.29: NodeShape algorithm, *x-kindOf* handling

function createPropertyShape (ontModel, ownerName, schemaName, schemaObject, schemas)

- Create a PropertyShape from ownerName + ”_” + schemaName + “PropertyShape”.
- Scan schemaObject (the body of the property schema) for the x-refersTo extension property.
- Extract string from the x-kindOf property.
- Create a property A with the extracted string.
- Add property A to the ontology (ontModel).
- Create a property B from the ownerName + “_” + schemaName.
- Add property A to the ontology (ontModel).
- Make property B a subproperty of A (add the connection triple: B rdfs:subPropertyOf A to the ontology).
- Add the triple: PropertyShape sh:path property B.
- Scan schemaObject (the body of the property schema) for the rest properties and add them to the PropertyShape by writing triples with PropertyShape as Subject and the corresponding Predicate - Object according to each property.
- Return PropertyShape Individual.

Listing 3.30: PropertyShape algorithm, *x-kindOf* handling

Next in the series of x-properties is x-mapsTo. When *x-mapsTo* is used in a Schema object, it points to another schema object or property schema to dictate that it shares the same semantics. This extension property can be confused with the property *$ref* of the OpenAPI, but it differs. The *x-mapsTo* connects two schemas semantically. The *$ref* property is used as an inline substitution. This property allows an OpenAPI Object to refer to other components inside the OpenAPI description and avoid repetition.

Continuing with *x-mapsTo*, in Listing 3.31 inside *SecondPet* Schema object, *SecondPet* and *Pet* share the same semantic value, as well as “*id*” and “*secondId*”. The algorithm that handles this operation, first needs to extract the value from the *x-mapsTo* property. Then, extracts the schema name and if this Schema Object has not yet been created, it calls the *createNodeShape* to create the Schema Object. Finally, after getting the semantic value, it will assign it to the current shape.

```
SecondPet:
  x-mapsTo: '#/components/schemas/Pet'
  type: object
  required:
    -secondId
  properties:
    secondId:
      x-mapsTo: '#/components/schemas/Pet.id'
      type: integer
      format: int64
```

Listing 3.31: *SecondPet* Schema object, *x-mapsTo*

```
<SecondPetNodeShape>  a  sh:NodeShape ;
        rdfs:label           "SecondPetNodeShape" ;
        sh:property          <SecondPet_secondIdPropertyShape> ;
        sh:targetClass       <Pet> .

<SecondPet_secondIdPropertyShape>
        a               sh:PropertyShape ;
```

```
        rdfs:label      "SecondPet_secondIdPropertyShape" ;
        openapi:name    "secondId" ;
        sh:datatype     xsd:long ;
        sh:path         <Pet_id> .
```

Listing 3.32: *SecondPet* NodeShape and *SecondId* PropertyShape, *x-mapsTo*

The *x-mapsTo* is also handled inside the c*reateNodeShape* and *createPropertyShape* functions. This extension property does not require to create any ontology class or property because it "borrows" them from the referred shape. After extracting the value of the *x-mapsTo* property, the algorithm needs to decompose it to find the referred schema, whether it is an object or a property. After extracting the referred schema name, the algorithm checks if it is already encountered on another OpenAPI element which means that a shape has already been created. Then, we can take the value from the sh:targetClass predicate or the sh:path predicate and connect it with the current shape. If the schema is not already encountered, we call the function *createNodeShape* to create the requested shape and then we take its values. The last step is always to return the Shape Individuals so that the algorithm can carry on with the mapping of the rest OpenAPI Objects. The function calls for the current example are *createNodeShape (ontModel, SecondPet, secondPetBodyObject, schemas)* and *createPropertyShape (ontModel, SecondPet, secondIdBodyObject, schemas)*.

function createNodeShape (ontModel, schemaName, schemaObject, schemas)

- Create a NodeShape from schemaName + "NodeShape".
- Scan schemaObject for the x-mapsTo extension property.
- Extract string from x-mapsTo property.
- Extract the referred schema name from the string.
- Check in the list of schemas (schemas argument) if the Schema object that this schema name is referring to, has already been created.
- If not, create the Schema object by recursively calling the createNodeShape function, then take it`s semantic value (class).
- If yes, take the semantic value (class).
- Add the tripe: NodeShape sh:targetClass class.
- For every property schema inside the schemaObject
  - Call createPropertyShape and return the property shape Individual
  - Map the property shape Individual to the current node shape Individual: NodeShape sh:property PropertyShape
- Return NodeShape Individual.

Listing 3.33: NodeShape algorithm, *x-mapsTo* handling

function createPropertyShape (ontModel, ownerName, schemaName, schemaObject, schemas)

- Create a PropertyShape from ownerName + ”_” + schemaName + “PropertyShape”.
- Scan schemaObject (the body of the property schema) for the x-mapsTo extension property.
- Extract string from the x-mapsTo property.
- Extract the referred schema name A (of the Schema object) from the string.
- Extract the referred schema name B (of the property of the Schema object) from the string.
- Check in the list of schemas (schemas argument) if the Schema object that this schema name A is referring to, has already been created.
- If not, create the Schema object by calling the createNodeShape function, then take from it`s property (with schema name B) the semantic value (property).
- If yes, take the semantic value from its property (with schema name B).
- Add the triple: PropertyShape sh:path property.
- Scan schemaObject (the body of the property schema) for the rest properties and add them to the PropertyShape by writing triples with PropertyShape as Subject and the corresponding Predicate - Object according to each property.
- Return PropertyShape Individual.

Listing 3.34: PropertyShape algorithm, *x-mapsTo* handling

At this point, a reasonable question would be, what happens when a Schema Object should not have a semantic value. Sometimes, an OpenAPI description contains schemas that are not widely used or they are so uniquely written that do not contribute to a general purpose. For example, when an author of an API creates a Response object schema specifically for the debugging of his server, or some other purpose that satisfies only his service structure. This Response Object does not have any use for other authors of OpenAPI descriptions and therefore is not necessary to acquire a semantic value. Another reason is that it is pointless to create classes for a Schema Object that uses polymorphism (this is explained later on). The solution to this problem is the *x-refersTo* property with the value of “*none*”. When “*x-refersTo: none*” is used in a Schema object, the corresponding node shape in the ontology will not have an *sh:targetClass* predicate or if it is a property shape, will not have an *sh:path* predicate. If the value of *x-refersTo* is “*none*”, the algorithm will not write the triple (NodeShape, *sh:targetClass*, class) to the ontology. That means the node shape will not contain an *sh:targetClass* predicate. In the same way, if a property contains the “*x-refersTo: none*” the algorithm will not write the triple (PropertyShape, sh:path, property) and the corresponding property shape will not contain an *sh:path* predicate.

```yaml
Pet:
  x-refersTo: none
  type: object
  required:
    - id
  properties:
    id:
      x-refersTo: none
      type: integer
      format: int64
```

Listing 3.35: *Pet* Schema Object, “*x-refersTo: none”*

```
<PetNodeShape>  a     sh:NodeShape ;
        rdfs:label         “PetNodeShape” ;
        sh:property        <Pet_idPropertyShape> .

<Pet_idPropertyShape>
        a                sh:PropertyShape ;
        rdfs:label       “Pet_idPropertyShape” ;
```

```
        openapi:name    “id” ;
        sh:datatype     xsd:long .
```

Listing 3.36: *Pet* NodeShape and *Id* PropertyShape, “*x-refersTo: none”*

The handling of the *x-refersTo* extension property with the value of “*none*” is also found inside the *createNodeShape* and *createPropertyShape* functions. When the *x-refersTo* extension property is detected, before creating a class or property we check for the “*none*” value. In that case the shapes, whether it is a node shape or property shape do not acquire a semantic value and the algorithm continues scanning the rest of the properties and returns the Shape Individuals. The function calls are *createNodeShape (ontModel, Pet, petBodyObject, schemas)* and *createPropertyShape (ontModel, Pet, idBodyObject, schemas)*.

function createNodeShape (ontModel, schemaName, schemaObject, schemas)
- Create a NodeShape from schemaName + “NodeShape”.
- Scan schemaObject for the x-refersTo extension property.
- Extract string from x-refersTo property.
- If value is “none” continue without creating a triple.
- For every property schema inside the schemaObject
  - Call createPropertyShape and return the property shape Individual
  - Map the property shape Individual to the current node shape Individual: NodeShape sh:property PropertyShape
- Return NodeShape Individual.

Listing 3.37: NodeShape algorithm, “*x-refersTo: none*” handling

function createPropertyShape (ontModel, ownerName, schemaName, schemaObject, schemas)
- Create a PropertyShape from ownerName + ”_” + schemaName + “PropertyShape”.
- Scan schemaObject (the body of the property schema) for the x-mapsTo extension property.
- Extract string from the x-refersTo property.
- If value is “none” continue without creating a triple.
- Scan schemaObject (the body of the property schema) for the rest properties and add them to the PropertyShape by writing triples with PropertyShape as Subject and the corresponding Predicate - Object according to each property.
- Return PropertyShape Individual.

Listing 3.38: PropertyShape algorithm, “*x-refersTo: none*” handling

These features of the algorithm by creating and adding new classes and properties to the ontology, contribute to ontology expansion. Connecting semantic values and Schema Objects is of high importance for this cause. Creating semantic values for unannotated schemas is a big step for the algorithm which massively introduces new semantic values in the semantic web. Nevertheless, due to “*x-refersTo: none*” this is not a one-way affair. This addition gives the author the opportunity to skip this step and thus the algorithm offers a lot of flexibility.

3.7 General Case

In this section we showcase how the functions *createNodeShape* and *createPropertyShape* handle all the extension properties that may exist in a Schema Object. Besides the properties *x-refersTo, x-kindOf* and *x-mapsTo* there is also the *x-collectionOn* property. This property is handled inside the functions *createNodeShape* and but does not semantically affect a Schema Object. It is used to indicate that a Schema Object is actually a collection. Typically, a collection (or a list) of resources in OpenAPI v3.0 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 3.39 defines a model as a collection of *Pet* objects.

Collections are represented using *Collection* class. The class *PetCollection* will become a subclass of *Collection* (*openapi:Collection*) class. This is showcased in Listing 3.40 where we present the mapping of the *PetCollection* Schema Object. The *PetCollection_petsPropertyShape* shape that corresponds with property "*pets*" of *PetCollection*, is defined as a member of a collection because its *sh:path* predicate has the *openapi:member* value. Also, this property shape contains the *sh:node* predicate which specifies the node shape that a value node conforms to.

```yaml
PetCollection:
  x-collectionOn: pets
  type: object
  properties:
    pets:
      type: array
      items:
        $ref: '#/components/schemas/Pet'
```

Listing 3.39: *PetCollection* Schema Object, *x-collectionOn*

```
<PetCollectionNodeShape>
        a               sh:NodeShape ;
        rdfs:label      "PetCollectionNodeShape" ;
        sh:property     <PetCollection_petsPropertyShape> ;
        sh:targetClass  <PetCollection> .

<PetCollection>  a   owl:Class ;
        rdfs:subClassOf  openapi:Collection .

<PetCollection_petsPropertyShape>
        a             sh:PropertyShape ;
        rdfs:label    "PetCollection_petsPropertyShape" ;
        openapi:name  "pets" ;
        sh:node       <PetNodeShape> ;
        sh:path       openapi:member .
```

Listing 3.40: *PetCollection* NodeShape, *x-CollectionOn*

The *x-collectionOn* property can only be found in a Schema Object and not in a property schema. Therefore, the only function that handles this extension property is the *createNodeShape.* When the *x-collectionOn* property is detected, the algorithm creates an ontology class from the argument "schemaName". Then, the algorithm will declare this class as a subclass of the *openapi:Collection*. Continuing, the algorithm scans and maps the remaining properties of the Schema Object and returns the node shape Individual. The function call for this example is *createNodeShape (ontModel, PetCollection, petCollectionBodyObject, schemas)*.

function createNodeShape (ontModel, schemaName, schemaObject, schemas)
- Create a NodeShape from schemaName + “NodeShape”.
- Scan schemaObject for the x-collectionOn extension property.
- Extract string from x-collectionOn property.
- Create a class from the schemaName.
- Add class to the ontology (ontModel).
- Make the class a subclass of openapi:Collection (add the connection triple: class rdfs:subclassOf openapi:Collection)
- For every property schema inside the schemaObject
  - Call createPropertyShape and return the property shape Individual
  - Map the property shape Individual to the current node shape Individual: NodeShape sh:property PropertyShape
- Return NodeShape Individual.

Listing 3.41: NodeShape algorithm, *x-CollectionOn* handling

At this point, it is necessary to present the two functions in a general form to give the reader an overview. The general form of *createNodeShape* is showcased in Listing 3.42. The first step is to create the node shape name from the argument “*schemaName*” and the “*No*deShape” string. Next, we check the Schema Object for any of our defined extension properties and we handle them according to what we have seen so far. If there are no extension properties, we handle the Schema Object as an unannotated one. Continuing, we call *createPropertyShape* for every property schema inside the Schema Object and map the returned property shape Individual with the node shape Individual. This is done with the tripe “*NodeShape sh:property ProperyShape”.* As a last step we return the node shape Individual.

function createNodeShape (ontModel, schemaName, schemaObject, schemas)
- Create a NodeShape from schemaName + “NodeShape”.
- If the schemaObject contains any extension properties:
  - Check for the extension property x-refersTo
    - Check is the value is “none”
  - Check for the extension property x-kindOf
  - Check for the extension property x-mapsTo
  - Check for the extension property x-collectionOn
- If the schemaObject does not contain any extension properties:
  - Handle it according to an unannotated Schema Object
- For every property schema inside the schemaObject
  - Call createPropertyShape and return the property shape Individual
  - Map the property shape Individual to the current node shape Individual: NodeShape sh:property PropertyShape
- Return NodeShape Individual.

Listing 3.42: *createNodeShape* function

The general form of *createPropertyShape* is presented in Listing 3.43. Our first action here is to create the name of the property shape from the arguments “*ownerName*”, “*schemaName*” appended with the string “*PropertyShape*”. The next step is to check for extension properties. If the property schema contains any extension properties, the corresponding actions are performed in each case. However, if there are no extension properties the property schema is handled as an unannotated one. Continuing, we extract and map all the properties that can be found in a property schema such as description, format etc. The last step is to return the property shape Individual.

function createPropertyShape (ontModel, ownerName, schemaName, schemaObject, schemas)

- Create a PropertyShape from ownerName + ”_” + schemaName + “PropertyShape”.
- Do semantic validation //This is explained later on
- If the schemaObject (the body of the property schema) contains any extension properties:
  - Check for the extension property x-refersTo
    - Check is the value is “none”
  - Check for the extension property x-kindOf
  - Check for the extension property x-mapsTo
- If the schemaObject does not contain any extension properties:
  - Handle it according to an unannotated property schema
- Check property schema for OpenAPI keywords (anyOf, oneOf, not) //This is explained later on
- Scan schemaObject (the body of the property schema) for the rest properties and add them to the PropertyShape by writing triples with PropertyShape as Subject and the corresponding Predicate - Object according to each property.
- Return PropertyShape Individual.

Listing 3.43: *createPropertyShape* function

3.8 Composition and Inheritance

In previous paragraphs we described the use of the *allOf* keyword and its importance for the connection between schemas. The *allOf* keyword expresses the concept of inheritance between two schemas. With the help of SHACL and the logical component “*and*” (*sh:and* predicate) we were able to implement the composition logic into a shape. In Listing 3.44, the *Pet* Schema object extends the *OldPet* with the additional property of “*id*”. Next, in Listing 3.45 we present the node shape of *Pet*. Inside the *sh:and* list there is the focus node of *OldPetNodeShape* which represents the *OldPet* Schema object and a blank node that represents the additional properties added to it (inline Schema Object). That is the formal and practical way to implement the “*and*” logical constraint and by extension the concept of composition in the ontology. Also, in the same example, the two classes *Pet* and *OldPet* that were created become the objects of *sh:targetClass* predicates. The relation between the classes must be the same as the relation between the schemas. Consequently, in Owl terminology the *Pet* class inherits the *OldPet* class as shown in Listing 3.46.

```
Pet:
  allOf:
    - $ref: '#/components/schemas/OldPet'
    - type: object
      required:
      - id
      properties:
        id:
          type: integer
          format: int64

OldPet:
  type: object
  required:
    - name
    - tag
  properties:
    name:
      type: string
    tag:
      type: string
```

Listing 3.44: *Pet*, *OldPet* Schema Objects, allOf

```
<PetNodeShape>  a       sh:NodeShape ;
        rdfs:label      "PetNodeShape" ;
        sh:and           (  [ a            sh:NodeShape ;
                              sh:property  <Pet_idPropertyShape>
                            ]
                            <OldPetNodeShape>
                           );
        sh:targetClass  <Pet>

<OldPetNodeShape>  a    sh:NodeShape ;
        rdfs:label      "OldPetNodeShape" ;
        sh:property     <OldPet_tagPropertyShape> , <OldPet_namePropetShape> ;
        sh:targetClass  <OldPet>
```

Listing 3.45: *Pet*, *OldPet* nodeshapes, sh:and list

```
<Pet>   a               owl:Class ;
            rdfs:subClassOf  <OldPet> .

<OldPet>  a     owl:Class .
```

Listing 3.46: *Pet* class and *OldPet* class relation (Inheritance)

The function *parseSchemaObject* in Listing 3.47 is responsible for handling Schema Objects with OpenAPI v3.0 keywords. The arguments of this function have already been explained and discussed as they are the exact arguments passed on the *createNodeShape* function. The algorithm that implements the above functionality first checks if the keyword *allOf* exists in the Schema Object. If the keyword exists, an empty RDF list will be created. Continuing, we call the *parseSchemaObject* function for every schema under the *allOf* keyword in order to create node Shape Individuals. These individuals will be inserted inside the list. When we are done adding shape Individuals, we map the list to the node shape by writing the triple *NodeShape sh:and RDF List*. Continuing, the algorithm gets the created classes from every shape Individual and creates the triple where the *rdfs:subClassOf* predicate is used to declare the relation between the two classes. Finally, it returns the NodeShape Individual. The function call of *parseSchemaObject* for this example is *parseSchemaObject (ontModel, Pet, petBodyObject, schemas)*.

At this point it is important to make the distinguish between the named Individuals and the blank nodes. Under the *allOf* keyword, there are two schemas, the *OldPet* and one unnamed schema. For each of these, the *parseSchemaObject* will be called and will create a node shape. For the *OldPet* the function call is *parseSchemaObject (ontModel, OldPet, oldPetBodyObject, schemas)* and for the unnamed schema the function call is *parseSchemaObject (ontModel, NULL, unnamedBodyObject, schemas)*. These function calls are placed inside the first "for loop" of Listing 3.47. The difference is that the body of the named Individual is written as a separate node and only the name of the Individual is presented in the RDF list. Instead, the body of the other node which is called blank node, is written inside the RDF list as shown in Listing 3.45 in bold. The Blank node cannot be presented elsewhere because it has no name to be referenced by. Also, blank nodes do not contain an *sh:targetClass* predicate. This is why in the second "for loop" in Listing 3.47 below, we only retrieve the classes of named Shape Individuals.

If the Schema Object does not contain the *allOf* keyword, it is a simple case of mapping any given schema. If the type of the schema is object, we call *createNodeShape*. If the type of the schema is either *integer, string* or *boolean* we call the *createPropertyShape* function. If the type of schema is array then we call *createCollectionNodeShape.* This function works similarly to the *x-collectionOn* property. More specifically, it creates a NodeShape Individual and creates an ontology class from its schema name. This class later becomes a subclass of the Collection class. Finally, it calls *createPropertyShape* to map the body of the array schema. This function is analyzed in the next chapter.

function parseSchemaObject (ontModel, schemaName, schemaObject, schemas)

- If the schemaObject has the allOf keyword then:
  - Create a NodeShape for the schemaObject by calling createNodeShape with schemaName.
  - Create an empty RDF list.
  - For every schema under the allOf keyword:
  - Call parseSchemaObject and return the node shape (named Shape Individual or blank node).
  - Add the returned node shape to the RDF list.
- Connect the NodeShape with the RDF list by writing the triple: NodeShape sh:and RDF list.
- For every named Shape Individual inside the RDF list:
  - Extract the class from its sh:targetClass predicate.
  - Make the class of the NodeShape a subclass of the extracted class (from the previous step).
- Else if the schemaObject does not contain the allOf keyword then:
  - If the type of the schemaObject is "object" then call createNodeShape
  - Else if the type of the schemaObject is "array" then call createCollectionNodeShape
  - Else call createPropertyShape
- Return Shape Individual.

Listing 3.47: parseSchemaObject function, allOf handling

This section gives us the opportunity to clarify some important differences between property schemas and Schema Objects. The main reason for creating Schema Objects is to have a group of properties that exist under a specified schema. This is easy to understand as all of the examples of Schema Objects presented in this chapter contain property schemas. Therefore, it is valid to say the structural components of a Schema object are it`s properties. A property schema is a unit. A Schema object is created with at least one unit. A property schema much like a unit cannot be composed from other units. This is why the keyword *allOf* is not permitted in a property schema and therefore composition between property schemas is not supported.

3.9 Semantically Annotated Composed Schema Objects

In the previous section we analyzed inheritance and composition in Schema Objects that did not contain any semantic annotations. When we add semantic annotations in composed Schema Objects, we implement inheritance relations between semantic values. In Listing 3.48, we present two Schema Objects connected with the keyword *allOf*. The *Pet* Schema Object uses the extension property *x-refersTo* to acquire a semantic value from the example domain. The algorithm will create a class with the IRI "*https://example.com/ontology/Pet*" and not a class with the name *Pet* because the schema is annotated. In contrast to *Pet*, *OldPet* will acquire a class of *OldPet* given to it by the functions explained earlier. Consequently, the "*https://example.com/ontology/Pet*" class will become a subclass of *OldPet* class as shown in Listing 3.49.

```
Pet:
  x-refersTo: https://example.com/ontology/Pet
  allOf:
    - $ref: '#/components/schemas/OldPet'
    - type: object
      required:
      - id
      properties:
        id:
          type: integer
          format: int64

OldPet:
  type: object
```

```
  required:
    - speed
  properties:
    speed:
      type: string
    petType:
      type: string
```

Listing 3.48: *Pet* (annotated), *OldPet* Schema Objects

```
<PetNodeShape>  a   sh:NodeShape ;
        rdfs:label      "PetNodeShape" ;
        sh:and          ( [ a            sh:NodeShape ;
                                sh:property  <Pet_idPropertyShape>
                          ]
                          <OldPetNodeShape>
                        ) ;
        sh:targetClass  <https://example.com/ontology/Pet> .

<https://example.com/ontology/Pet>
        a               owl:Class ;
        rdfs:subClassOf  <OldPet> .

<OldPetNodeShape>
        a                sh:NodeShape ;
        rdfs:label       "OldPetNodeShape" ;
        sh:property      <OldPet_petTypePropertyShape> ,
<OldPet_speedPropertyShape> ;
        sh:targetClass   <OldPet> .

<OldPet>  a  owl:Class .
```

Listing 3.49: *https://example.com/ontology/Pet* subclass of *OldPet*

Another example, is when only the subschema (a schema under a keyword) has an extension property. In this situation, the algorithm creates a class for the unannotated Schema Object *Pet* and registers this class as a subclass of "*https://example.com/ontology/OldPet*". In this example the subschema has an extension property.

```
Pet:
  allOf:
    - $ref: '#/components/schemas/OldPet'
    - type: object
      required:
      - id
      properties:
        id:
          type: integer
          format: int64

OldPet:
  x-refersTo: https://example.com/ontology/OldPet
  type: object
  required:
```

```
    - speed
  properties:
    speed:
      type: string
    petType:
      type: string
```

Listing 3.50: *Pet, OldPet* (annotated) Schema Objects

```
<PetNodeShape>
        a               sh:NodeShape ;
        rdfs:label      "PetPetNodeShape" ;
        sh:and          ( [ a            sh:NodeShape ;
                                sh:property  <Pet_idPropertyShape>
                          ]
                          <OldPetNodeShape>
                        ) ;
        sh:targetClass  <Pet> .

<Pet>  a   owl:Class ;
        rdfs:subClassOf  <https://example.com/ontology/OldPet> .

<OldPetNodeShape>
        a               sh:NodeShape ;
        rdfs:label      "OldPetNodeShape" ;
        sh:property     <Old_petTypePropertyShape> ,
                         <OldPet_speedPropertyShape> ;
        sh:targetClass  <https://example.com/ontology/OldPet> .
```

Listing 3.51: *Pet* subclass of *https://example.com/ontology/OldPet*

Finally, the last example contains two schemas enhanced with extension properties. The algorithm then, will create a subclass relation between the semantic values that the Schema Objects referred to.

```
Pet:
  x-refersTo: https://example.com/ontology/Pet
  allOf:
    - $ref: '#/components/schemas/OldPet'
    - type: object
      required:
      - id
      properties:
        id:
          type: integer
          format: int64

OldPet:
  x-refersTo: https://example.com/ontology/OldPet
  type: object
  required:
    - speed
  properties:
```

```
    speed:
      type: string
    petType:
      type: string
```

Listing 3.52: *Pet* (annotated), *OldPet* (annotated) Schema Objects

```
<PetNodeShape>
        a               sh:NodeShape ;
        rdfs:label      "PetNodeShape" ;
        sh:and          ( [ a            sh:NodeShape ;
                                sh:property  <Pet_idPropertyShape>
                          ]
                          <OldPetNodeShape>
                        ) ;
        sh:targetClass  <https://example.com/ontology/Pet> .

<https://example.com/ontology/Pet>
        a                owl:Class ;
        rdfs:subClassOf  <https://example.com/ontology/OldPet> .

<OldPetNodeShape>
        a               sh:NodeShape ;
        rdfs:label      "OldPetNodeShape" ;
        sh:property     <OldPet_petTypePropertyShape> ,
                         <OldPet_speedPropertyShape> ;
        sh:targetClass  <https://example.com/ontology/OldPet> .
```

Listing 3.53: "*https://example.com/ontology/Pet*" subclass of "*https://example.com/ontology/OldPet*"

As a last example of this section, we present the use of *x-refersTo* with the value of "*none*" in the current functionality. Earlier, we explained that with "*x-refersTo: none*" a semantic value for the current Schema object will not be created. Inheritance is a relation between two classes, therefore in this situation it cannot exist. This is why in Listing 3.55 *PetNodeShape* is missing an *sh:targetclass* predicate and there is no subclass relation written to the ontology concerning these node shapes.

```
Pet:
  x-refersTo: none
  allOf:
    - $ref: '#/components/schemas/OldPet'
    - type: object
      required:
      - id
      properties:
        id:
          type: integer
          format: int64

OldPet:
  type: object
  required:
    - name
  properties:
    name:
```

```
      type: string
    tag:
      type: string
```

Listing 3.54: *Pet* (none), *OldPet* Schema Objects

```
<PetNodeShape>  a   sh:NodeShape ;
        rdfs:label  "PetNodeShape" ;
        sh:and      ( [ a            sh:NodeShape ;
                        sh:property  <Pet_idPropertyShape>
                      ]
                      <OldPetNodeShape>
                    ) .

<OldPetNodeShape>  a    sh:NodeShape ;
        rdfs:label      "OldPetNodeShape" ;
        sh:property     <OldPet_tagPropertyShape> , <OldPet_namePropetShape> ;
        sh:targetClass  <OldPet> .


<OldPet>  a     owl:Class .
```

Listing 3.55: *OldPet*, *Pet*, no subclass relation

In the previous section we presented the function *parseSchemaObject* which handles the existence of the *allOf* keyword inside a Schema Object. This function (Listing 3.47), creates a shape for each Schema Object involved with the *allOf* keyword calls the *createNodeShape* function. This is where the extension properties (if there are any) of every Schema Object are handled inside the algorithm. Therefore, the *parseSchemaObject* function can handle Schema Objects with the *allOf* keyword regardless of whether they contain any extension property or not.

3.10 Polymorphism

It is very often for APIs to have requests and responses that can be described by several alternative schemas. For this functionality OpenAPI specification provides the keywords *oneOf* and *anyOf* as we saw earlier. Schema Objects that contain these keywords are translated with the use of *sh:xone* and *sh:or* respectively. In Listing 3.56, there is a very representative example for the concept of polymorphism. This unnamed schema representing the response with code "*201*" can be described by one of the following schemas, *Dog*, *Cat* or *Lizard*. Because the schema has no name, a Blank node will be created containing a predicate of *sh:xone* and an RDF list as an object. Inside the RDF list, as we can see in Listing 3.57 there are the named Individuals *CatNodeShape*, *DogNodeShape* and *LizardNodeShape*. For ease of presentation, we show only the value of the *openapi:schema* predicate which in this case represents the Schema Object inside the Response Object.

```
responses:
    "201":
      description: variety
      content:
        application/json:
          schema:
            oneOf:
              - $ref: "#/components/schemas/Cat"
              - $ref: "#/components/schemas/Dog"
```

```
              - $ref: "#/components/schemas/Lizard"

components:
    schemas:
        Dog:
          type: object
          properties:
            bark:
              type: boolean
            breed:
              type: string

        Cat:
          type: object
          properties:
            hunts:
              type: boolean
            age:
              type: integer

        Lizard:
          type: object
          properties:
            lovesRocks:
              type: boolean
```

Listing 3.56: *Cat*, *Dog*, *Lizard* Schema Objects, *oneOf*

```
openapi:schema  [ a         sh:NodeShape ;
                  sh:xone   ( <LizardNodeShape> <DogNodeShape> <CatNodeShape> )
                ]

<CatNodeShape>  a       sh:NodeShape ;
        rdfs:label      "CatNodeShape" ;
        sh:property     <Cat_agePropertyShape> , <Cat_huntsPropertyShape> ;
        sh:targetClass  <Cat> .

<Cat>  a    owl:Class .

<DogNodeShape>  a       sh:NodeShape ;
        rdfs:label      "DogNodeShape" ;
        sh:property     <Dog_breedPropertyShape> , <Dog_barkPropertyShape> ;
        sh:targetClass  <Dog> .

<Dog>  a    owl:Class .

<LizardNodeShape>  a    sh:NodeShape ;
        rdfs:label      "LizardNodeShape" ;
        sh:property     <Lizard_lovesRocksPropertyShape> ;
        sh:targetClass  <Lizard> .
```

```
<Lizard>  a  owl:Class .
```

Listing 3.57: *Cat*, *Dog*, *Lizard*, shapes, *sh:xone*

An unnamed schema is the most common way to express a Schema Object with polymorphism. However, this is not mandatory. The following example in Listing 3.58 will help to better understand this argument. There, we present the same Response Object, this time with the named schema *ThreePets*. When one of the schemas (*Cat*, *Dog* or *Lizard*) is returned as a value, it will take the place of *ThreePets*. Consequently, the *ThreePets* node shape, has no utility in this case. Another approach on this matter would be to enrich the *ThreePets* Schema Object with the extension property *x-refersTo* with the value of "*none*". This way, the *ThreePetsNodeShape* would not contain the *sh:targetClass* predicate and therefore, it would not acquire a semantic value, much like a blank node. The algorithm supports polymorphism in both named and unnamed Schema Objects.

```
responses:
    "201":
      description: variety
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/ThreePets"

components:
    schemas:
        ThreePets:
          oneOf:
            - $ref: "#/components/schemas/Cat"
            - $ref: "#/components/schemas/Dog"
            - $ref: "#/components/schemas/Lizard"
```

Listing 3.58: *ThreePets*, *oneOf*

```
openapi:schema          <ThreePetsNodeShape>

<ThreePetsNodeShape>  a  sh:NodeShape ;
        rdfs:label      "ThreePetsNodeShape" ;
        sh:targetClass  <ThreePets> ;
        sh:xone         ( <LizardNodeShape> <DogNodeShape> <CatNodeShape> ) .

<ThreePets>  a  owl:Class .
```

Listing 3.59: *ThreePets* node shape, *sh:xone*

The semantic annotated Schema Objects with polymorphism fall under the same category. Although they are supported by the algorithm, the case remains the same. There is no real benefit to semantically annotate the Schema Object that contains polymorphism (*ThreePets* in this case) for the same reason as to name this kind of Schema Object. Also, the semantic values between the shapes are not connected in any way, in contrast to *allOf* as we presented in a previous section. So, if any of the *Dog*, *Cat* or *Lizard* were semantically annotated their semantic values would not have been interrelated.

In Listing 3.60 we present the part of the algorithm concerning polymorphism implementation in Schema Objects. This is also handled in *parseSchemaObject* function therefore the arguments are already explained. The

first step is to check if the keywords *anyOf* or *oneOf* exist inside the body of the Schema Object. If they exist, we create an empty RDF list. Continuing, inside the "for loop" we call *parseSchemaObject* for every one of the *Cat, Dog* and *Lizard* schemas. Then, we add the returned node shapes, which in this case are named Individuals, to the list. Next, we map the list to the node shape by writing the appropriate triples. The "*NodeShape sh:or RDF list"* tripe for *anyOf,* and the "*NodeShape sh:xone RDF list"* for *oneOf*. Finally, we return the NodeShape Individual. The function call for the unnamed schema in Listing 3.56 would be *parseSchemaObject (ontModel, NULL, unnamedBodyObject, schemas)* and for the *Threepets* Schema Object of Listing 3.58 would be *parseSchemaObject (ontModel, Threepets, threePetsBodyObject, schemas)*. If the schema does not contain neither *anyOf* nor *oneOf* then we call the appropriate function according to its type and return the Shape Individual.

function parseSchemaObject (ontModel, schemaName, schemaObject, schemas)
- If the schemaObject has the anyOf or oneOf keyword then:
- Create a NodeShape for the schemaObject by calling createNodeShape with schemaName.
- Create an empty RDF list.
- For every schema under the anyOf or oneOf keyword:
  - Call parseSchemaObject and return the node shape (named Shape Individual or blank node).
  - Add the returned node shape to the RDF list.
- Connect the NodeShape with the RDF list by writing the triple:
  NodeShape sh:or RDF list or NodeShape sh:xone RDF list
- Else if the schemaObject does not contain the anyOf or oneOf keyword then:
  - If the type of the schemaObject is "object" then call createNodeShape
  - Else if the type of the schemaObject is "array" then call createCollectionNodeShape
  - Else call createPropertyShape
- Return Shape Individual.

Listing 3.60: *parseSchemaObject*, *anyOf*, *oneOf* handling

Polymorphism is also supported in property schemas. Properties can also take alternative schemas, and it is expressed the same way as in Schema Objects. An example triple would look like "*PropertyShape sh:or RDF list*" or "*PropertyShape sh:xone RDF list*". Under the keywords *anyOf, oneOf*, an array of schemas is expected. These schemas can be either inline or referred, as presented in Listing 3.61. In this example the property "*speed*" of *Pet* Schema Object, is defined by either one of the listed schemas. Consequently, the mapped shape will be a property shape with the name "*Pet_speedPropertyShape*" and an object of "*Pet_speed*" to the *sh:path* predicate, according to what we have seen so far. Also, the property shape will contain the RDF list *sh:or*, with three nodes. The first one is a node of the *SpecifiedPropertyShape* and the rest are blank nodes. In Listing 3.62 this part of the ontology is presented.

```
Pet:
  type: object
  required:
    - speed
  properties:
    speed:
      anyOf:
        - type: integer
          format: int64
        - type: string
        - $ref: '#/components/schemas/Specified'

components:
    schemas:
      Specified:
        type: integer
  format: int32
```

Listing 3.61: speed property, anyOf

```
<Pet_speedPropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      "Pet_speedPropertyShape" ;
        openapi:name    "speed" ;
        sh:or           ( <SpecifiedPropertyShape>
                          [ a             sh:PropertyShape ;
                            sh:datatype   xsd:string
                          ]
                          [ a             sh:PropertyShape ;
                            sh:datatype   xsd:long
                          ]
                        ) ;
        sh:path         <Pet_speed> .


<SpecifiedPropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      "SpecifiedPropertyShape" ;
        sh:datatype     xsd:int ;
        sh:path         <Specified> .

<Specified>  a  rdf:Property .
```

Listing 3.62: *speed* property shape, *sh:or*

The function *createPropertyShape* is responsible for handling the keywords *anyOf*, and *oneOf* inside the body of a property schema. For ease of presentation the Listing 3.63 presents the case in which the keywords exist. In a similar fashion as the previous cases, the first step is to create an empty RDF list. Then, for every schema under the keywords we call the *parseSchemaObject* and create a property shape Individual. Every individual is added to the list. Continuing, we map the RDF list with the property shape. This is done either with the "*PropertyShape sh:or RDF list*" triple or with "*PropertyShape sh:xone RDF list*". The function call for the example above would be *createPropertyShape (ontModel, Pet, speed, speedBodyObject, schemas)*. In addition, inside the "for loop" the *parseSchemaObject* would be called three times. For the first inline schema would be *parseSchemaObject(ontModel, NULL, unnamed1BodyObject, schemas)* for the second inline schema would be *parseSchemaObject(ontModel, NULL, unnamed2BodyObject, schemas)* and for the referenced property schema *Specified* would be *parseSchemaObject (ontModel, specified, specifiedBodyObject, schemas)*.

function createPropertyShape (ontModel, ownerName, schemaName, schemaObject, schemas)

- If the schemaObject (the body of the property schema) has the anyOf or oneOf keyword then:
- Create an empty RDF list.
- For every schema under the anyOf or oneOf keyword:
  - Call parseSchemaObject and return the node shape (named Shape Individual or blank node).
  - Add the returned property shape to the RDF list.
- Connect the PropertyShape with the RDF list by writing the triple:
  PropertyShape sh:or RDF list or PropertyShape sh:xone RDF list.
- Return PropertyShape Individual.

Listing 3.63: *createPropertyShape* algorithm, polymorphism handling

3.11 Annotations within Property Schema Components

In section 3.8, Composition and Inheritance, we made an important distinction between Schema Objects and property schemas. Another difference between them is that in contrast to Schema Objects, semantically annotating polymorphed property schemas is beneficial and not at all pointless. That lies in the nature of property schemas with polymorphism. A property schema with polymorphism basically means that it accepts alternatives datatypes. There are two cases. The first is when a semantic value is connected with different schemas through a property. The second is when a property is connected with semantic values through property attributes (i.e., properties of properties). Both categories are introduced in property schemas with polymorphism support along with semantic annotations.

When a property is annotated with a semantic value, the schema property and the semantic value are connected. When a semantically annotated property uses polymorphism, it means that the semantic value is connected to the schemas represented under the *oneOf*/*anyOf* keywords. Such a case, is presented in Listing 3.64 in bold. In Listing 3.65, the mapped property shape "*Speed*" has the semantic value "*https://example.com/ontology/speed*". Consequently, this semantic value maps with only one (due to *oneOf*) of the schemas listed inside the *sh:xone* RDF list. The value "*https://example.com/ontology/speed*" can be one of string, long or mapped with the *SpecifiedPropertyShape*, which is an int (integer). To better understand this example as well as the following, it is valid to say that in this case the semantic value uses polymorphism to connect with different schemas.

```
Pet:
  type: object
  required:
    - speed
  properties:
    speed:
      x-refersTo: https://example.com/ontology/Speed
      oneOf:
        - type: integer
          format: int64
        - type: string
        - $ref: '#/components/schemas/Specified'

components:
    schemas:
      Specified:
        type: integer
        format: int32
```

Listing 3.64: semantically annotated property with polymorphism

```
<Pet_speedPropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      "Pet_speedPropertyShape" ;
        openapi:name    "speed" ;
        sh:path         <https://example.com/ontology/Speed> ;
        sh:xone         ( <SpecifiedPropertyShape>
                          [ a             sh:PropertyShape ;
                            sh:datatype   xsd:string
                          ]
                          [ a             sh:PropertyShape ;
                            sh:datatype   xsd:long
                          ]
                        ) .

<SpecifiedPropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      "SpecifiedPropertyShape" ;
        sh:datatype     xsd:int ;
        sh:path         <Specified> .

<Specified>   a   rdf:Property .
```

Listing 3.60: speed property shape, semantic value

The second case is when the property attributes are semantically annotated instead of the property. The property "*Speed*" does not acquire a semantic value but it`s attributes do. Therefore, in parallel with our previous saying, this time the schema uses polymorphism to connect with different semantic values. This example is presented in Listing 3.61 in bold. In Listing 3.62, the mapped property shape of "*speed*" is presented along with the semantic value of each of the nodes (bold). Here, because the blank nodes inside the *sh:xone* RDF list contain the *sh:path* predicate, the property shape of "*speed*" cannot have such a predicate. In the previous example (Listing 3.60), the blank nodes did not contain the *sh:path* predicate, therefore validly the property shape has an *sh:path* predicate.

```
Pet:
  type: object
  required:
    - speed
  properties:
    speed:
      oneOf:
        - type: integer
          format: int64
          x-refersTo: https://example.com/ontology/Int64_property
        - type: string
          x-refersTo: https://example.com/ontology/String_property
        - $ref: '#/components/schemas/Specified'


components:
    schemas:
      Specified:
        x-refersTo: https://example.com/ontology/Int32_property
        type: integer
        format: int32
```

Listing 3.61: semantically annotated attributes in polymorphism

```
<Pet_speedPropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      "Pet_speedPropertyShape" ;
        openapi:name    "speed" ;
        sh:xone         ( <SpecifiedPropertyShape>
                          [ a             sh:PropertyShape ;
                            sh:datatype   xsd:string ;
                            sh:path       <https://example.com/ontology/String_property>
                          ]
                          [ a             sh:PropertyShape ;
                            sh:datatype   xsd:long ;
                            sh:path       <https://example.com/ontology/Int64_property>
                          ]
                        ) .

<SpecifiedPropertyShape>
        a               sh:PropertyShape ;
        rdfs:label      "SpecifiedPropertyShape" ;
        sh:datatype     xsd:int ;
        sh:path         <https://example.com/ontology/Int32_property> .
```

Listing 3.62: speed property shape, semantically annotated components

These are the only examples that represent a valid semantically annotated property schema with polymorphism. An invalid case of such property, is when all of the components as well as the main property have a semantic value. The contradiction here is that a semantic value “tries” to be mapped with other semantic values. This is not permitted. The reason why is that the two semantic values, that of the property and either one of the property attributes come into conflict. This is always the case as more than one semantic value cannot define a single model simultaneously. This example is presented in Listing 3.63. In this case the algorithm will exit with the appropriate message, without making the desired translation.

```
OtherPet:
  type: object
  required:
    - speed
  properties:
    speed:
      x-refersTo: https://example.com/ontology/Speed
      oneOf:
        - type: integer
          format: int64
          x-refersTo: https://example.com/ontology/Int64_property
        - type: string
          x-refersTo: https://example.com/ontology/String_property
        - $ref: '#/components/schemas/Specified'

components:
    schemas:
      Specified:
        x-refersTo: https://example.com/ontology/Int32_property
        type: integer
        format: int32
```

Listing 3.63: semantic malfunction, conflict

Another invalid example and an easier to understand is that of semantic inconsistency. In case where the property is not semantically annotated but it`s property attributes are, all attributes need to be semantically annotated. Otherwise, if some attributes have and others do not, all of them will not be semantically equivalent. This case is presented in Listing 3.64. The two inline property schemas obtain a semantic value but the referenced property schema "*Specified*" does not.

```
OtherPet:
  type: object
  required:
    - speed
  properties:
    speed:
      oneOf:
        - type: integer
          format: int64
          x-refersTo: https://example.com/ontology/Int64_property
        - type: string
          x-refersTo: https://example.com/ontology/String_property
        - $ref: '#/components/schemas/Specified'

components:
    schemas:
      Specified:
        type: integer
        format: int32
```

Listing 3.64: semantic malfunction, inconsistency

The last convention of the algorithm appears in the case of the annotated components. In Listing 3.61, we presented a valid example of polymorphism. However, if the type of the "*Specified*" schema was "*object*", then the example would become invalid as presented in Listing 3.65. In this Listing, it`s semantic value of "*https://example.com/ontology/Specified*" characterizes it as a semantically annotated Schema Object and not as a semantically annotated property schema. Consequently, under property "*speed*" would result a semantic inconsistency containing two semantically annotated property schemas (inline schemas) and one semantically annotated Schema Object which is not equivalent. A method to overcome this stalemate is to allow the Schema Object "*Specified*" to be annotated as property schema like in Listing 3.65 (black bold). In this example we present this exact situation, where the "*Specified*" schema is double annotated, first as a Schema Object and secondly as property schema. However, Swagger Parser does not allow this type of format (black bold) because an array object which is referenced and not inline, cannot have properties (in this case an extension property) outside it`s referenced body.

```
OtherPet:
  type: object
  required:
    - speed
  properties:
    speed:
      oneOf:
```

```
        - type: integer
          format: int64
          x-refersTo: https://example.com/ontology/Int64_property
        - type: string
          x-refersTo: https://example.com/ontology/String_property
        - $ref: '#/components/schemas/Specified'
          x-refersTo: https://example.com/ontology/Annotate_as_Property

components:
    schemas:
      Specified:
        x-refersTo: https://example.com/ontology/Specified
        type: object
        ...
```

Listing 3.65: semantic malfunction, inconsistency

In Listing 3.66, we present the algorithm that handles the validity of Schema Object properties. The function name is *semanticValidation* and has two arguments. The first argument "*schemaObject*" contains the body of the schema property in discuss. The second argument "*schemas*" contains all the schemas encountered so far in the OpenAPI document. The algorithm essentially performs a double check. The first check is for the uniformity of attributes concerning their extension properties. Either all of them will contain an extension property or none of them. Secondly checks if the "parent" property has an extension and acts appropriately according to the first check. This function will exit and cause the whole algorithm to stop if the above conditions are not met. In conclusion, if a rule is violated the function will exit otherwise it returns to the function which called it and the algorithm continues.

function semanticValidation(schemaObject, schemas)

- Check if the schemaObject (the body of the property schema) contains an extension property.
- Check all the components for extension properties (use the schemas argument for referenced schemas because they are not contained inside the body of the schemaObject).
- If the property contains an extension property, exit at the first component that will contain an extension property.
- If all of the components do not contain extension properties continue.
- If the property does not contain an extension property, check the components.
- If some of the components contain extension properties, exit.

Listing 3.66: algorithm semantic validation handling

3.12 Keyword Not

In this subsection of the chapter, we study the case of the not keyword provided by the OpenAPI Specification. The not keyword is used in property schemas and defines what type of values is not acceptable for the current property. It mainly helps to modify schemas and make them more specific. In particular, in Listing 3.67 we claim that the property "*byType*" of the *Pet* Schema Object can be anything but a string. As mentioned in a previous section, the SHACL constraint for this keyword is the *sh:not*. In contrast with the other constraints (*sh:or*, *sh:and*, *sh:xone*), this one does not accept as a value an RDF list but a single node. This is clear in Listing 3.68 where the corresponding property shape of property "*byType*" is presented. There, the *sh:not* predicate has a single blank node as an object.

```
Pet:
  type: object
  required:
    - id
  properties:
    id:
```

```
      type: integer
      format: int64
    byType:
      not:
        type: string
```

Lising 3.67: Pet Schema object, byType property, not

```
<Pet_byTypePropertyShape>
        a             sh:PropertyShape ;
        rdfs:label    "Pet_byTypePropertyShape" ;
        openapi:name  "byType" ;
        sh:not        [ a            sh:PropertyShape ;
                        sh:datatype  xsd:string
                      ] ;
        sh:path       <Pet_byType> .

<Pet_byType>  a  rdf:Property .
```

Listing 3.68: byType property shape, sh:not

In Listing 3.69, we present the same Schema Object of Listing 3.67 with the addition of an extension property that dictates a semantic value. Although the Schema Object at first seems valid, it is not. That is because if the property under the not keyword has a semantic value it will designate that only this value is not allowed. So, for this example, every other string will be permitted except the a string that is semantically enriched with the "*https://example.com/ontology/SpecificString*" value. This negates the universality of the not keyword and its original purpose. In particular, the keyword not is intended to exclude a whole category of properties (e.g., strings, integers etc.) and not only one specific property. So, in these cases we assume it is a misuse of the not keyword from the author of the OpenAPI description. Therefore, our algorithm will ignore any extension property and will continue to the translation as presented in Listing 3.70. The mapped shaped as a result from Listing 3.69 is the same as above (Listing 3.68).

```
Pet:
  type: object
  required:
    - id
  properties:
    id:
      type: integer
      format: int64
    byType:
      not:
        x-refersTo: https://example.com/ontology/SpecificString
        type: string
```

Listing 3.69: Pet Schema object, byType property, not, x-refersTo

The OpenAPI keyword "not" is also handled inside the *createPropertyShape* function. When the keyword is detected, the function calls the *parseSchemaObject* function to map the schema under the "*not*" keyword and returns the shape Individual. Then. the algorithm write the connection triple "*PropertyShape sh:not ShapeIndividual*". However, before calling *schemaObject* we set all extension properties to NULL. The last step is to return the PropertyShape Individual. For the examples above, the function calls would be *createPropertyShape (ontModel, Pet, byType, byTypeBodyObject, schemas)* and inside the function, *parseSchemaObject (ontModel, NULL, unnamedBodyObject, schemas)*.

function createPropertyShape (ontModel, ownerName, schemaName, schemaObject, schemas)

- Check if the schemaObject has the not keyword.
- If there are any extension properties
  - Set extension properties to NULL.
- Call parseSchemaObject to create a shape Individual for the schema under not keyword.
- Write the triple: PropertyShape sh:not shape Individual.
- Add the triple to the ontology (ontModel).
- Return PropertyShape Individual.

Listing 3.70: algorithm keyword “not” handling

3.13 Synopsis

In this chapter we analyzed how we handle the Schema Objects of an OpenAPI description. We also showcased the different concepts that may exist inside the body of a Schema Object. Composition, polymorphism and semantic annotations both in Schema Object but also in property schemas create countless combinations. In this section we present these features along with some basic combinations in a hierarchical way.

1.1 Schema Object to Node Shape (with or without extension properties)

The OpenAPI Object will call *parseSchemaObject* (sections 3.8 and 4.6) and if it does not contain any OpenAPI keywords (*allOf, anyOf, oneOf*) it will get the type of the Schema Object. Then, it will call the appropriate function. We assume that the type is ”*object*”. Then, the function *createNodeShape* (sections 3.7 and 4.6) will be called. After handling any extension property inside the body of the Schema Object, *then*, call *createPropertyShape* (sections 3.7 and 4.6) and map every property schema to a PropertyShape Individual. Map every PropertyShape Individual to the NodeShape Individual and return it to *parseSchemaObject.* The last function, will return the Shape Individual to the OpenAPI Object.

1.2 Schema Object to Node Shape (with or without extension properties) with OpenAPI keywords

Again, the handling of this case begins with an OpenAPI Object calling *parseSchemaObject* (sections 3.8 and 4.6) to map its schema. Here, at the first decision branch of Figure 3.1 we assume the Schema Object contains either one of the *allOf, anyOf* or *oneOf.* Then, we call *createNodeShape* (sections 3.7 and 4.6) to make a NodeShape Individual and handle any extension properties that it might have. We also create an empty RDF list. Then for every component under the keywords, we call *parseSchemaObject* in order to create a Shape Individual for each one of them. In particular, the *createNodeShape* is called for the Schema Object that contains the mentioned keywords. Then, we call *parseSchemaObject* for each of the components because they might also contain one of these keywords.

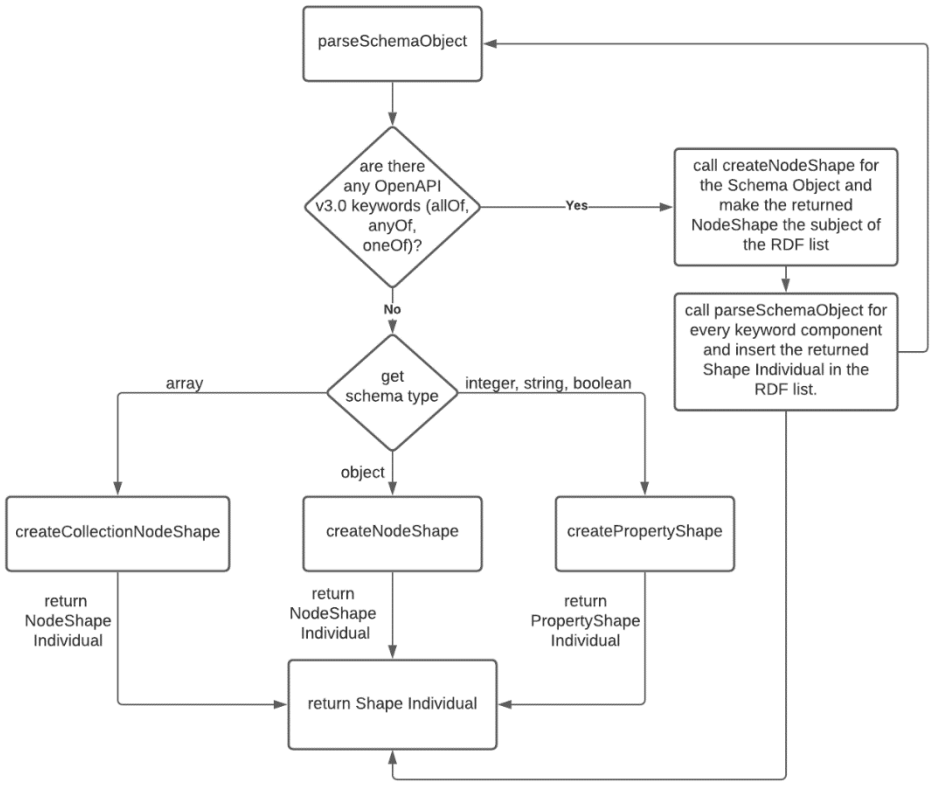


Figure 3.1: parseSchemaObject flowchart

If the components contain semantic annotations, they will get handled like the case 1.1 above. These Shape Individuals, will be added to the list and the list will be mapped to the NodeShape Individual (subject), with either one of the *sh:and, sh:or* or *sh:xone* predicates. However, if the keyword is the *allOf,* we make an additional action in order to create subclass relations between the classes of the Shape Individuals. When we are done with the RDF list and its components, we continue in *parseSchemaObject* and return the NodeShape Individual. In summary, this is the way we handle composition and inheritance but also polymorphism inside a Schema Object.

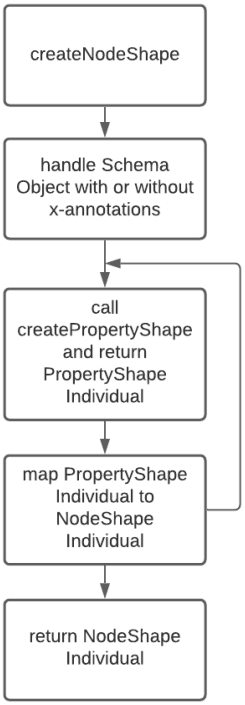


Figure 3.2: createNodeShape flowchart

2.1 Property Schema to Property Shape (with or without extension properties)

All property schemas inside the OpenAPI description are handled in *createPropertyShape* function (sections 3.7 and 4.6). This function can be called by either, *parseSchemaObject* (sections 3.8 and 4.6)*, createNodeShape* (sections 3.7 and 4.6) or *createCollectionNodeShape* (section 4.6). In any case, the function handles a schema with either one of the types *integer, string* or *boolean.* The function starts by handling any extension properties that may be found inside its schema body. Then, (assuming it does not contain the keywords *anyOf, oneOf,* or "*not*") maps all its property attributes (e.g., format, description etc.) to the PropertyShape Individual. The last step is to return the PropertyShape Individual.

2.2 Property Schema to Property Shape (with or without extension properties) with OpenAPI keywords

For this case, in Figure 3.3 we assume that the property schema indeed has an OpenAPI keyword. At, the start of the function we handle any extension properties that might exist inside the property schema. Then, if any of the keywords *oneOf, anyOf* or "*not*" is encountered we call *parseSchemaObject* (sections 3.8 and 4.6) and map the components under these keywords with Shape Individuals. In case of "*not*" we have only one component and its Shape Individual will become an object to the *sh:not* predicate. If this is not the case, we create an RDF list and insert the Shape Individuals. The list becomes the object of either *sh:or* or *sh:xone*. Finally, we map the list to our PropertyShape Individual and return it. In conclusion, this is they way a property schema with polymorphism or the keyword "not" is handled.

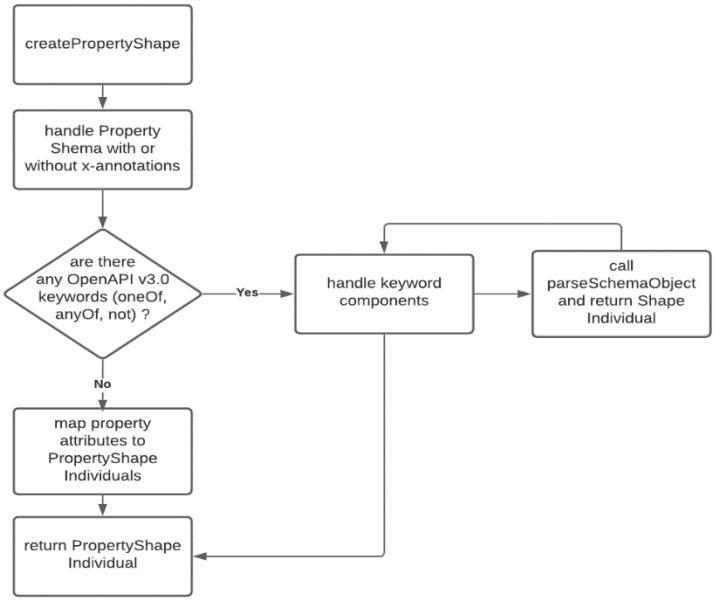


Figure 3.3: createPropertyShape flowchart

3.1 Array Schema to Node Shape

In Figure 3.4 we handle any schema that is of type array. This is a very straightforward situation to handle. In *createCollectionNodeShape* (section 4.6) we create an Individual of NodeShape and then we make a class from its schema name. This class will become a subclass of the Collection class. The for the schema inside its "*items"* property, we call the *createPropertyShape* (sections 3.7 and 4.6) and return a PropertyShape Individual. This is mapped to the NodeShape Individual. As a last step we return the NodeShape Individual. In this case we do not encounter any semantic annotations.

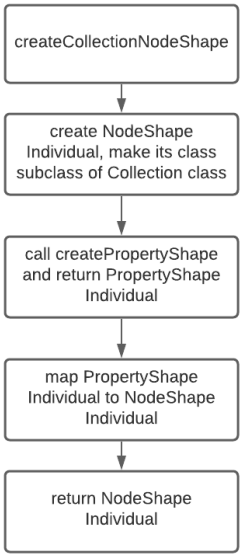


Figure 3.4: *createCollectionNodeShape* flowchart

This section sums up the combinations that are presented in this chapter. However, in OpenAPI descriptions there might be many more combinations. In particular, a Schema Object or a property schema might contain both polymorphism and composition. Specifically, a schema that exists under an *allOf* keyword might also contain an *allOf* keyword or even an *oneOf* keyword and so on. This is why for every component under the OpenAPI v3.0 keywords we call the *parseSchemaObject* (sections 3.8 and 4.6). This function will handle the OpenAPI keywords. In this way we allow any possible variation concerning composition and polymorphism.

Chapter 4

Instantiation Algorithm

4.1 Introduction

In this chapter, we present the main idea of mapping an OpenAPI service to the OpenAPI v3 Ontology. Besides Schema Objects, it is important to showcase the whole algorithm along with our approach on handling OpenAPI elements. As mentioned at the beginning of this thesis, the input of the algorithm is an OpenAPI description document. The output is an instantiated ontology where all the service properties are represented. The chapter is divided in sections that contain the most important functions of the algorithm. The functions are analyzed in terms of their input arguments, their output and the logic that they follow.

All OpenAPI Objects are mapped to the corresponding class in our OpenAPI v3 Ontology. The main responsibility of every function presented in this chapter is to map an OpenAPI Object and its properties to the appropriate class inside the ontology. In order to better comprehend the actions that follow we present the OpenAPI v3 Ontology in Figure 4.1.

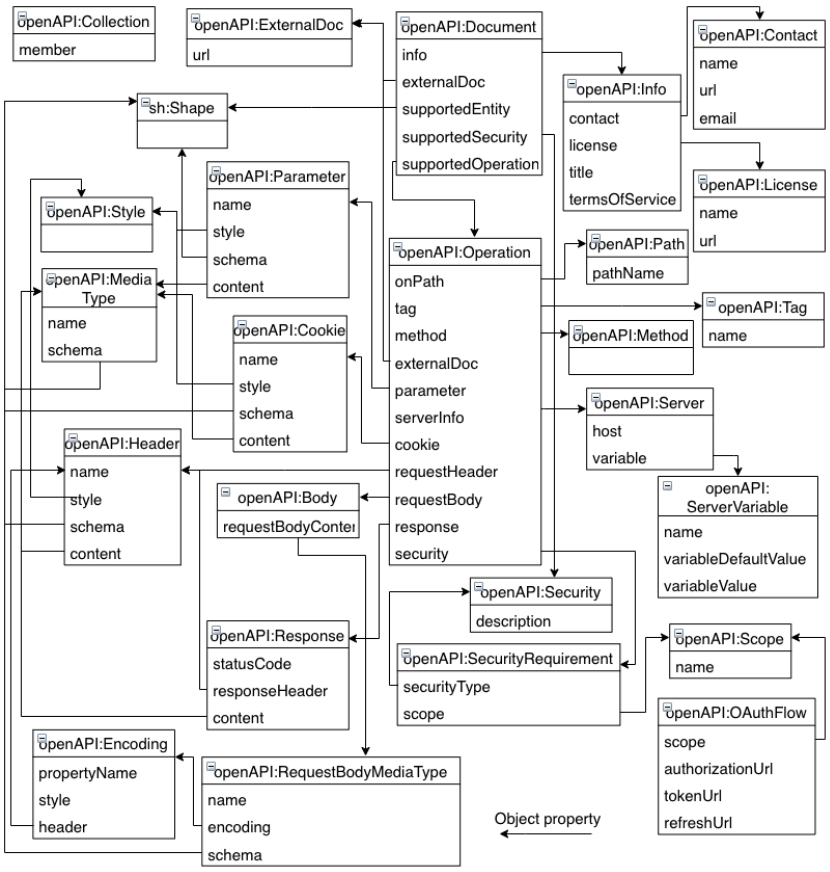


Figure 4.1: OpenAPI Version 3 Ontology

4.2 OpenAPI Object

The algorithm scans the OpenAPI document of a service and instantiates OpenAPI objects to classes of the ontology. In particular, after uploading the ontology in the memory, the algorithm will scan the OpenAPI file to extract info, servers, security schemes, tags and paths objects. These objects will become individuals of their corresponding classes.

The OpenAPI object (the root object of the OpenAPI document) is mapped to class Document. There may exist more than one appearance of *servers* or *securiySchemes* in an OpenAPI file. Property *servers* declares server information which applies across the description (global servers). This will be overwritten by server information defined in Path or Operations objects. Similarly, Security schemes declared by an operation will also override global declaration of security schemes. Property *Tags* contains the Tag objects for operations which are grouped. Through *x-onResource* property, Tag objects can associate operations with Schema Objects. Listing 4.1 illustrates the mapping between OpenAPI Object and the Document Class in the form of a simple algorithm. The input of the algorithm is the OpenAPI description document.

function parseDocumentObject (OpenAPI Document)

- Initialize the Ontology Model
- Create Document Individual
- Call Info Function and save Info Individual in Document Individual
  → parseInfoObject (infoObject) //Listing 4.2
- 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
    → parseServerObject (serverObject) //Listing 4.5
  - 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
    → parseSecuritySchemeObject (documentInd, securityObject) //Listing 4.6
- Keep a list (globalSecReqList) 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
    → parseSecurityReqObject (securityReqObject) //Listing 4.7
  - Add the Security Requirement Individual to the list.
- Call ExternalDoc Function and save ExternalDoc Individual in Document Individual
  → parseExternalDocObject (externalDocObject) //Listing 4.8
- Extract all Schema Objects from OpenAPI Service
- Keep a list (tagShapeMap) 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 (tagShapeMap)
  → parseTagObject (tagObject, componentSchemas) //Listing 4.9
- For every Path Object
- Keep a list (pathServersList) with all Path Servers
- For every Path Server:
  - Call Server Function and return Server Individual
    → parseServerObject (serverObject) //Listing 4.5
  - Add the Server Individual to the list
  - Keep a final Server list (Path if not empty, else Global)
  - Extract all Parameters from OpenAPI Service and put them in a list (pathParametersList)
  - Create Path Individual
  - For every Operation Object call Operation Function
    → parseOperationObject (documentId, pathInd, OperationObject, tagShapeMap, pathParametersList, pathServersList, globalSecReqlist) //Listing 4.10

Listing 4.1: OpenAPI Object to Document Individual

In Listing 4.1 we present the main function of the algorithm. Starting from the parseDocument function, we initialize the ontology (uploading the ontology in the memory) and continue by creating Individuals for several OpenAPI Objects. The main idea behind every function is to handle one OpenAPI Object and map its properties to the corresponding Individual. The first Individual is that of the Info Class which is created by calling the function parseInfoObject presented in Listing 4.2. The Info Object contains the Contact and the License Object which are mapped to Individuals by calling parseContactObject (Listing 4.3) and parseLicenseObject (Listing 4.4) respectively.

function parseInfoObject (infoObject)
- Create Info Individual
- Extract and map all Info's properties from OpenAPI Service (title, description, termsOfService, etc)
- Call Contact Function
  → parseContactObject (contactObject) //Listing 4.3
- Call License Function
  → parseLicenseObject (licenseObject) //Listing 4.4
- Return Info Individual

Listing 4.2: Info Object to Info Individual

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

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

Listing 4.4: License Object to License Individual

Next, we create a list for global Server Individuals and call the parseServerObject (Listing 4.5) to map all the properties of a Server Object as well as to create the corresponding Individual. Inside the Server function we check for server variables which will become Individuals of Server Variable Class and will be stored in the current Server Individual. As a last step, we return the Server Individual to be added to the list. All functions we have seen so far take as argument an OpenAPI Object and after mapping all its properties they return an Individual.

function parseServerObject (serverObject)
- Create Server Individual
- Extract and map all Server's properties from OpenAPI Service
- For every Server Variable Object in Server Object Variables:
  - Create Individual Server Variable Class
  - Extract and map all Server Variable's properties from OpenAPI Service
  - Save Sever Variable Individual in Server Individual
- Return Server Individual

Listing 4.5: Server Object to Server Individual

After dealing with Info and Server Objects, we proceed with Security Schemes and Security Requirements. With Security Schemes we follow the standard procedure of calling the function parseSecuritySchemeObject (Listing 4.6) and creating one Individual for every Security Scheme Object. A Security Scheme Object becomes an Individual of a certain class (ApiKEY, HTTP, OpenIDConnect, OAuth2) depending on property *type* inside the Security Scheme Object. This Individual is connected to the Document Individual with the property *supportedSecurity*. Handling Security Requirement Objects is a case more similar to that of the Server Objects. We need to keep a list of global Security Requirements which later might be overwritten by a Security Requirement declared in an operation. The function parseSecurityReqObject in Listing 4.7 maps the properties of the Security Requirement Object to an Individual and returns this Individual to be added on the list. Continuing inside the parseDocumentObject function, we need to handle External Document Objects. For these Objects we call the function parseExternalDocObject (Listing 4.8) and map its properties to an Individual. Finally, we return the ExternalDoc Individual to the main function.

function parseSecuritySchemeObject (documentInd, 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 the SecurityScheme Individual to "supportedSecurity" property of Document Individual of Document Individual

Listing 4.6: SecurityScheme Object to SecurityScheme Individual

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

Listing 4.7: Security Req Object to Security Req Individual

function parseExternalDocObject (externalDocObject)
- Create ExternalDoc Individual
- Extract and map ExternalDoc's properties from OpenAPI Service
- Return ExternalDoc Individual

Listing 4.8: External Doc Object to External Doc Individual

Continuing, we proceed with extracting all Schema Objects from the OpenAPI service which are listed under Components section of an OpenAPI document. These schemas are placed inside the list "schemas" that we presented in the previous chapter (Handling Schema Objects). Storing all schemas in one list at the start of the algorithm contributes a great deal in speed and flexibility when later on we might come across a "$ref" property which refers to a schema.

Next, we handle Tag Objects. The x-onResource connects a Tag Object with a Schema Object. To track this relation between the objects, we keep a list that contains pairs of Tags and Schemas. To add such a pair in the list, first we need to map it. Consequently, we call the function parseTagObject (Listing 4.9). This function creates a Tag Individual along with all its properties. Also, checks for the x-onResource property and then, if the extension property exists, it creates a Shape Individual for the referred schema. To map a Schema Object to a Shape Class we call parseSchemaObjcet which was thoroughly explained in the previous chapter. Afterwards, the function returns a pair of Tag – Shape Individual. Otherwise, returns a pair of Tag Individual – Null. The pair is returned to the main function parseDocumentObject to be added in the appropriate list. Tag Individuals and their associated Shapes are kept in a Map structure that will be used when instantiating Operation objects in a following section.

function parseTagObject (tagObject, componentSchemas)
- Create Tag Individual
- Extract and map Tag's properties from OpenAPI Service
- If x-onResource points to a Schema:
  - Find the Schema in componentsSchemas
  - Create the Shape individual by calling the Schema Function
    → parseSchemaObject (schemaName, schemaObject, componentSchemas) //Listing 4.19
- Return Tag Individual – Shape Individual if x-onResource was used, else return Tag Individual - Null

Listing 4.9: Tag Object to Tag Individual

The last OpenAPI object that is handled inside the parseDocumentObject function is the Path Object. For every Path Object we perform the same steps. Firstly, we extract any defined servers if the *server* property is set. For every server inside the Path Object, we call the Server Function and we add the returned Server Individual inside a list which represents the servers used in the current path. If the property *servers* is not set, then the list is replaced by the globalServerList created previously. This simply means that a Path Object can either have servers defined in its body or the global servers. Continuing, we extract all the parameters inside the current path and we create a Path Individual.

The final step for a Path Object is to handle its operations. An Operation Object can overwrite a lot of the previously mapped OpenAPI objects such as tags, servers, parameters and security requirements. For this reason, these OpenAPI objects are given as arguments to the parseOperationObject function which is responsible for handling the Operation Objects.

4.3 Operation Object

In Listing 4.10 we present the parseOperationObject function. This function shows how the Individuals of Class Operation are created. The Operation method can be any of put, get, post, etc. and it is mapped under the property *method* of an Operation Individual along with other properties. Among these properties, we map the *onPath* property which associates an Operation Individual with a Path Individual. If there is an External Document Object inside the operation, we call the parseExternalDoc function and save the returned External Document Individual to Operation Individual. The same logic is followed with any Tag Objects inside the operation. Continuing, in an Operation Object there might be new Security Requirements as well as new Servers introduced. If this is the case, we call parseSecurityReqObject and parseServerObject functions respectively to overwrite the global ones. In addition, in order to connect the Operation Individual with Document Individual we use the *supportedOperation* property. The *x-operationType* extension property which may exist in an Operation Object, clarifies the type of the current operation. When an Operation Object contains this extension property, its Operation Individual will become a member of the corresponding operationType class.

function parseOperationObject (documentId, pathInd, OperationObject, tagShapeMap, pathParametersList, pathServersList, globalSecReqlist)

- Create Operation Individual
- Extract and map all Operation's properties from OpenAPI Service
- Save Operation Individual to Path Individual with property onPath
- Call ExternalDoc Function and save ExternalDoc Individual to Operation Individual
  → parseExternalDocObject (externalDocObject) //Listing 4.8
- Call Tag Function and save Tag Individual to Operation Individual.
  → parseTagObject (tagObject, componentSchemas) //Listing 4.9
- 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
    → parseSecurityReqObject (securityReqObject) //Listing 4.7
- 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
    → parseServerObject (serverObject) //Listing 4.5
- Save Operation Individual to "supportedOperation" property of Document Individual
- Get the resource where x-operationType points
- Set resource-class as second class of the new Operation Individual
- Extract Parameters from OpenAPI Service
- Keep in a combined Parameters list from Path and from Operation
- Map every parameter depending on the value of property "in" by calling Parameter Function (for each parameter call one of the functions below)
  → parseCookieParameterObject (cookieObject, componentSchemas) //Listing 4.11
  → parseHeaderParameterObject (headerName, headerObject, componentSchemas) //Listing 4.12
  → parseQueryParameterObject (queryParameter, componentSchemas) //Listing 4.13
  → parsePathParameterObject (pathParameter, componentSchemas) //Listing 4.14
- Call RequestBody Function for every Request Body
  → parseRequestBodyObject (requestObject, componentSchemas) //Listing 4.17
- Call Response Function for every Response
  → parseResponseObject (statusCode, responseObject, componentSchemas)

Listing 4.10: Operation Object to Operation Individual

Other structural elements of an operation are parameters, request bodies and responses. Regarding parameters, after extracting them from the Operation Object, we need to map each one of them depending on their property *in*. Parameter Objects can be any of type Path, Query, Header or Cookie and are instantiated to the corresponding classes (i.e., PathParameter, Query, Header and Cookie respectively). After creating the Parameter Individuals, each time by calling the appropriate function, we add properties to the Operation Individual in order to associate them with it. Parameters, request bodies and responses are going to be explained in the following sections.

4.4 Parameter Object

As mentioned in the previous section, there are four possible parameter locations specified by the *in* field inside the Parameter Object. Path – Where the parameter value is 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 on the *in* field, the Parameter Object is mapped to one of the above Classes.

Each of the Listings 4.11 – 4.14 represent the mapping of the Parameter Class to the Corresponding Class depending on the value of *in* property. We start by creating the appropriate Individual. Then extract and map all of the properties of a Parameter Object (description, explode, name etc.). Next, we need to handle the Schema Object of the parameter. We do this either by extracting the schema from the Parameter Object body, or by retracting it from the schemas (*componentSchemas* argument) when the schema is not placed inside the Parameter Object body but it is referred with the use of *$ref* property. Then, we save the Shape Individual to the Parameter Individual. The last action we take, is to handle the Media Type Objects inside the Parameter Objects. After extracting all the Media Type Objects from the parameter, we add them to a list and call the parseMediaTypeObject for each one of them. Finally, we save the MediaType Individuals to the Parameter Individual under the property *content*.

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
  → parseSchemaObject (schemaName, schemaObject, componentSchemas) //Listing 4.19
- Save Shape Individual to "schema" property of CookieParameter Individual
- Extract MediaType Object from CookieParameter Object
- Call MediaType Function and return MediaType Individual
  → parseMediaTypeObject (mediaName, mediaTypeObject componentSchemas) //Listing 4.15
- Add MediaType Individual to MediaType list
- Save MediaType list to "content" property of CookieParameter Individual
- Return CookieParameter Individual

Listing 4.11: 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 SchemaObject from HeaderParameter Object
- Call Schema Function and return Shape Individual
  → parseSchemaObject (schemaName, schemaObject, componentSchemas) //Listing 4.19
- Save Shape Individual to “schema” property of HeaderParameter Individual
- Extract Mediatype Object from HeaderParameter Object
- Call MediaType Function and return MediaType Individual
  → parseMediaTypeObject (mediaName, mediaTypeObject componentSchemas) //Listing 4.15
- Add MemberType list to “content” property of HeaderParameter Individual
- Return HeaderParameter Individual

Listing 4.12: Parameter Object to Header Individual

function parseQueryParameterObject (queryParameter, 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
  → parseSchemaObject (schemaName, schemaObject, componentSchemas) //Listing 4.19
- Save Shape Individual to “schema” property of QueryParameter Individual
- Extract MediaType Object from QueryParameter Object
- Call MediaType Function and return MediaType Individual
  → parseMediaTypeObject (mediaName, mediaTypeObject componentSchemas) //Listing 4.15
- Add MediaType Individual to MediaType list
- Save MediaType list to “content” property of QueryParameter Individual
- Return QueryParameter Individual

Listing 4.13: Parameter Object to Query Individual

function parsePathParameterObject (pathParameter, componentSchemas)
- Create PathParameter Individual
- Extract and Map PathParameter’s properties from OpenAPI Service
- Extract Schema Object from PathParameter Object
- Call parseSchemaObject Function and return Shape Individual
  → parseSchemaObject (schemaName, schemaObject, componentSchemas) //Listing 4.19
- Save Shape Individual to “schema” property of PathParameter Individual
- Extract MediaType Object from PathParameter Object
- Call MediaType Function and return MediaType Individual
  → parseMediaTypeObject (mediaName, mediaTypeObject componentSchemas) //Listing 4.15
- Add MediaType Individual to MediaType list
- Save MediaType list to “content” property of PathParameter Individual
- Return PathParameter Individual

Listing 4.14: Parameter Object to Path Individual

In Listing 4.15 we present the parseMediaTypeObject function. We start by creating the MediaType Individual and mapping the *mediaName* value of the object to the *mediaName* property of the MediaType Individual. Then, we create a Shape Individual by calling the parseSchemaObject function. The schema may exist inside the Media Type Object or being referred to in the *componentSchemas*. Continuing, we create a list for all Encoding Individuals that need to be mapped with the Media Type Individual. The Encoding Objects are mapped with the use of the parseEncodingObject (Listing 4.16). Finally, we return the Media Type Individual.

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
  → parseSchemaObject (schemaName, schemaObject, componentSchemas) //Listing 4.19
- Create a list for Encoding Individuals
- For each Encoding Object in MediaType Object:
  - Call Encoding Function and return Encoding Individual
    → parseEncodingObject (encodName, encodingObject, componentSchemas) //Listing 4.16
  - Add the Encoding Individual to the list
- Assign the list to the "encoding" property of MediaType Individual
- Return MediaType Individual

Listing 4.15: Media Type Object to Media Type Individual

The parseEncodingObject function is responsible for handling Encoding Objects in an OpenAPI document. Following the standard procedure, the first step is to create an Encoding Individual. Then we map the *encodName* value to the *encodName* property of the Encoding Individual. Inside an Encoding Object there might be Header Objects which are mapped to Header Individuals and assigned to the Encoding Individual through the *encodingHeader* property. The Header Objects might be more than one therefore we keep a list and then map the Header Individuals list to the Encoding Object. Then, we return the Encoding Individual.

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 poperties from OpenAPI Service
- Create a list for Header Individuals
- For each Header Object in Encoding Object:
  - Call Header Function and return Header Individual
    → parseHeaderParameterObject (headerName, headerObject, componentSchemas) //Listing 4.12
  - Add the Header Individual to the list
- Assign the list to the "encodingHeader" property of Encoding Individual
- Return Encoding Individual

Listing 4.16: Encoding Object to Encoding Individual

A Header Object according to the OpenAPI Specification 3.0 has the same properties as a Parameter Object which contains the value *header* inside the *in* property. Therefore, the actions we make for a Header Object are the same as that of a Parameter Object of type *header*. The handling of this OpenAPI Object is already described in Listing 4.12.

4.5 Response and Request Body Objects

Both of these OpenAPI Objects can be found inside an Operation Object. Request bodies and responses are structural components of an operation. They are widely used inside an API to either declare the payloads that an action requires, or what a client receives after an action.

The parseRequestObject function (Listing 4.17) is called by the parseOperationObject (Listing 4.10) in order to handle the Request Body Object inside an operation. After creating a Request Body Individual, the function maps all the properties of the current Object from the OpenAPI Service. Then, it creates a list of MediaType Individuals and calls the Media Type function for each one. The list with the Individuals is mapped to the *content* property of the RequestBody Individual.

function parseRequestBodyObject (requestObject, componentSchemas)
- Create RequestBody Individual
- Extract and Map RequestBody's properties from OpenAPI Service
- Create MediaType list
- For ever MediaType Object in RequestBody Object:
  - Call MediaType Function and return MediaType Individual
    → parseMediaTypeObject (mediaName, mediaTypeObject, componentSchemas) //Listing 4.15
  - Add MediaType Individual to MediaType list
- Save MediaType list to "content" property of RequestBody Individual
- Return RequestBody Individual

Listing 4.17: Request Body Object to Request Body Individual

In Listing 4.18 we present the function parseResponseObject. Depending on the value *statusCode* of the Response Object, we create an Individual of the corresponding Class. Then, we map all the properties of the Response Object and we create one list for the Header Individuals and one list for the Media Type Individuals. In order to fill these lists, we call the Header function as well as the Media Type function. The Header list of Individuals is mapped to the *responseHeader* property and the Media Type list of Individuals is mapped to the *content* property of the Response Individual. As a final step, we return the Response Individual to the parseOperationObject (Listing 4.10) where was originally called.

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 in Response Object
  - Call Header Function and return header Individual
    → parseHeaderParameterObject (headerName, headerObject, componentSchemas) //Listing 4.12
  - Add Header Individual to the list
- Add the Headers list to the property "responseHeader" of Response Individual
- Create MediaType list
- For ever MediaType Object in Response Object:
  - Call MediaType Function and return MediaType Individual
    → parseMediaTypeObject (mediaName, mediaTypeObject, componentSchemas) //Listing 4.15
  - Add MediaType Individual to MediaType list
- Save MediaType list to "content" property of Response Individual
- Return Response Individual

Listing 4.18: Response Object to Response Individual

4.6 Schema Objects

As we have seen so far, almost every OpenAPI Object contains a Schema Object. Some, contain a Schema Object directly (parameters, media type etc.) or through another OpenAPI Object (request bodies, responses, paths etc.). Consequently, the modification that took place in Schema Objects have a great impact on every OpenAPI Object and therefore the whole ontology.

In the previous sections we mentioned that Schema Objects become Individuals of Shape Class with the help of parseSchemaObject. This function calls either createNodeShape or createPropertyShape depending on the property *type* of a Schema Object. Although these three functions have already been analyzed in chapter 3, here we present them in a more spherical way to give the reader a better overview.

The parseSchemaObject function is presented in Listing 4.19. The first step is to check if the Schema Object with the current schema name (*schemaName* argument) has already been mapped with an Individual in a previous OpenAPI Object. If this is the case, we return the already created Shape Individual. Else, we proceed by checking the body of the Schema Object for OpenAPI keywords (allOf, anyOf, oneOf). The function then will handle the Schema Object (this case has already been analyzed in Chapter 3) and will return the Shape Individual. If the Schema Object is neither already created nor contains OpenAPI keywords, we check its *type* property and we call the appropriate function. As a last step we return the Shape Individual which we received from either createNodeShape, createPropertyShape or createCollectionNodeShape.

function parseSchemaObject (schemaName, schemaObject, componentSchemas)
- If there is a Shape Individual with this schemaName, return it
- Else check the Schema Object for the keywords allOf, anyOf, oneOf, handle them and call createNodeShape
  → createNodeShape (schemaName, schemaObject, componentSchemas) //Listing 4.20
    - For every component under the keywords, call parseSchemaObject
      → parseSchemaObject (compSchemaName, compSchemaObject, componentSchemas)
- Else check the property "type" of the Schema Object
  - If the property "type" is "object" call createNodeShape
    → createNodeShape (schemaName, schemaObject, componentSchemas) //Listing 4.20
  - If the property "type" is "int", "boolean" or "string" call createPropertyShape
    → createPropertyShape (ownerName, schemaName, schemaObject, componentSchemas) //Listing 4.21
  - If the property "type" is "array" call createCollectionNodeShape
    → createCollectionNodeShape (schemaName, schemaObject, componentSchemas) //Listing 4.22
- Keep schemaName in property "label" if Shape Individual
- Return Shape Individual

Listing 4.19: parseSchemaObject function

The algorithm in Listing 4.20 sums up the functionality of createNodeShape. After creating a Node Shape Individual and extracting all the properties (description etc.) we then check for any extension properties. The handling of extension properties for Node Shapes has already been analyzed in Chapter 3. Continuing, for every property schema inside the Schema Object we call the createPropertyShape function to create the Property Shapes. Finally, we return the Node Shape Individual.

function createNodeShape (schemaName, schemaObject, componentSchemas)
- Create Node Shape Individual
- Extract and map Schema's properties from OpenAPI Service
- Check the Schema Object for extension properties
- For every property schema inside the Schema Object
  - Call createPropertyShape
    → createPropertyShape (ownerName, schemaName, schemaObject, componentSchemas) //Listing 4.21
- Return Shape Individual

Listing 4.20: Schema Object to Node Shape Individual

The createPropertyShape function is already explained in detail in Chapter 3. After creating the Property Shape Individual, we make a semantic validation on it. Then we continue by checking for extension properties and any OpenAPI keywords that may exist in it. Continuing, we map all its properties (description, tittle, pattern etc) with the Individual. Finally, if there exists an External Doc or an XML Object, we call the corresponding functions and we map the returned Individuals to the Property Shape Individual. An XML Object can be found only in a property schema, and its function is presented below in Listing 4.21

function createPropertyShape (ownerName, schemaName, schemaObject, componentSchemas)
- Create Property Shape Individual using Schema and OwnerName
- Do semantic validation on the property schema
- Check property schema for extension properties
- Check property schema for OpenAPI keywords (anyOf, oneOf, not)
- Extract and map property schema's properties from OpenAPI Service
- Call External Doc Function and save it to the Property Shape Individual
  → parseExternalDocObject (externalDocObject) //Listing 4.8
- Call XML Object Function and save it to the Property Shape Individual
  → parseXMLObject (xmlObject) //Listing 4.21
- Return Property Shape Individual

Listing 4.21: Property Schema to Property Shape Individual

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

Listing 4.21: XML Object to XML Individual

The createCollectionNodeShape function is the last function presented in this chapter. This function is responsible for handling Schema Objects that are of *type array.* This is a very simple function which creates an ontology class from the schema name of the Schema Object. Then, makes this class a subclass of the Collection Class. Next, it calls createPropertyShape for the schemas under the *items* property and maps them with the current Node Shape Individual. Finally, it returns the Node Shape Individual.

function createCollectionNodeShape (schemaName, schemaObject, componentSchemas)
- Create Node Shape Individual
- Create a class from the schemaName and make it a subclass of the Collection Class
- Call createPropertyShape for items in array
  → createPropertyShape (ownerName, schemaName, schemaObject, componentSchemas) //Listing 4.21
- Map the returned Property Shape Individual to the Node Shape Individual
- Return Node Shape Individual

Listing 4.22: Schema Object to Shape Individual with Collection Class

4.7 Synopsis

The last section of this chapter provides an overview of the instantiation algorithm. In Figure 4.2, we present the flowchart of the instantiation algorithm. Starting with the *parseDocument* function, we extract and map the following OpenAPI Objects: *Info, License, Contact, Server, Security Scheme, Security Requirement, External Document* and *Tag*. Then, for every *Path Object* inside the OpenAPI description we call *parseServerObject* to map any newly defined servers inside each path. Continuing, we call the *parseOperationObject* to map the operations inside a *Path Object*. Inside *parseOperationObject* we extract any *Tag* and *External Document Objects* that the *Operation Object* might contain. In addition, the *Operation Object* might contain new servers and security requirements. If this is the case, we call *parseServerObject* and *parseSecurityReqObject* respectively. Otherwise, we use the global servers and security requirements. Then, we map the defined parameters for operation according to the type of each parameter (Header, Query, Path, Cookie). Finally, we map the *Request Body* and *Response Object* of the operation.

The *parseSchemaObject* function handles a *Schema Object* inside an OpenAPI Object. In particular, this function is called by all the parameter functions, the *parseTagObject* and the *parseMediaTypeObject*. Then the *parseSchemaObject* depending on the type of the schema will call either *createNodeShape, createPropertyShape* or *createCollectionNodeShape*.

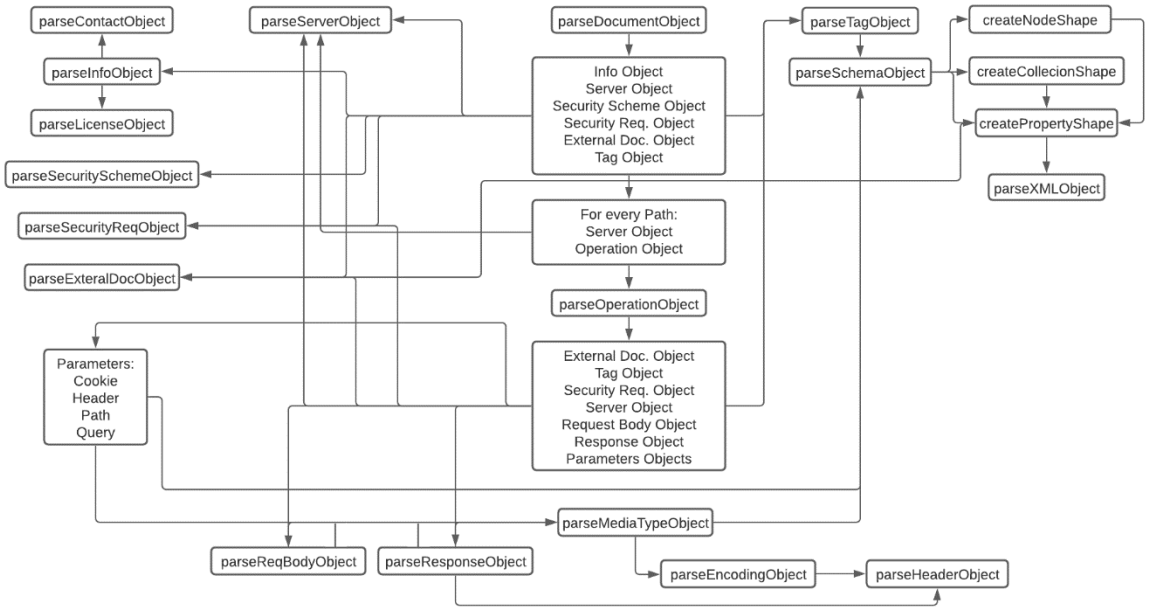


Figure 4.2: Instantiation Algorithm flowchart

Chapter 5

Web Application and SPARQL Results

5.1 Introduction

In this chapter we present our work aside the Instantiation algorithm. We create a Web Application with the implementation of the algorithm and we expose it on the Web (http://www.intelligence.tuc.gr/semantic-open-api/) so that other investigators and practitioners can test it with real life examples. Along with our Web Application, in this chapter we also present the results from SPARQL Queries on several Google APIs that are instantiated to the OpenAPI ontology.

5.2 Web Application

The Web Application disposes a very simple user interface. Our purpose was to avoid an unnecessary and complex user interface so as not to repel users and confuse them about the purpose of the Web Application. The system first of all provides an uploading mechanism where the user can upload an OpenAPI description. A user may upload an OpenAPI description in YAML[12] format. When the description is uploaded, the system will automatically take the user to the *Ontologies* page where the instantiated description is placed.

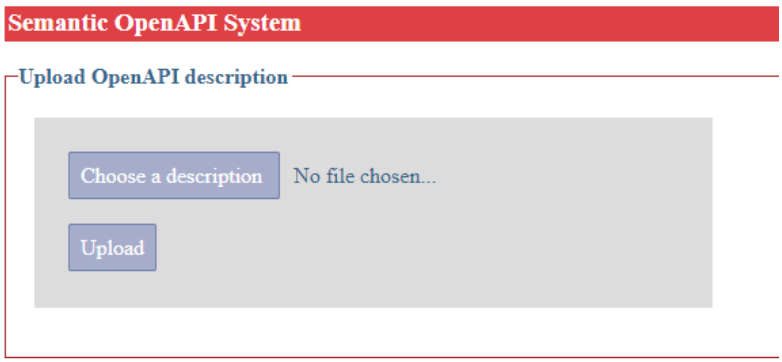


Figure 5.1: User Interface Upload

The ontology is available on TTL[13] (turtle) format. The name of the file where the ontology exists, consists of the original name of the OpenAPI description appended with the upload date and time. The system also provides information about the size of the file in kilobytes (KB). In addition, aside from the description that a user has uploaded, the user may also browse through all ontologies available in the database. In Figure 5.2 an example of the *Ontologies* page is presented.

[12] https://en.wikipedia.org/wiki/YAML
[13] https://en.wikipedia.org/wiki/Turtle_(syntax)

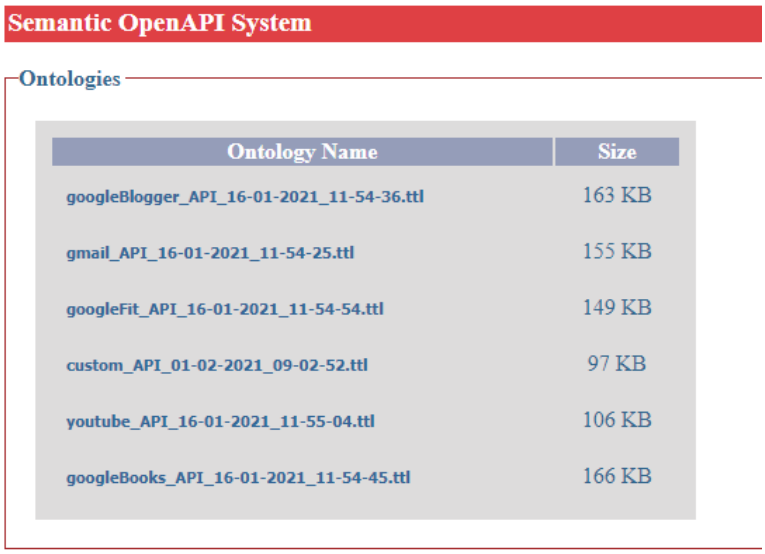


Figure 5.2: User Interface Ontologies

In addition, our system offers the opportunity to perform SPARQL Queries on all ontologies that our Web Application has stored. For this purpose, we added a Virtuoso Universal Server[14] which contains all the ontologies previously created with the Web Application in the form of graphs. When navigating to *Queries* page, a list of data graphs is listed. The graphs represent the ontologies inside the Virtuoso Database[15]. This way the user is able to inspect the graphs that desires, and perform SPARQL Queries on a specific graph. In Figure 5.3 the graphs list is presented.

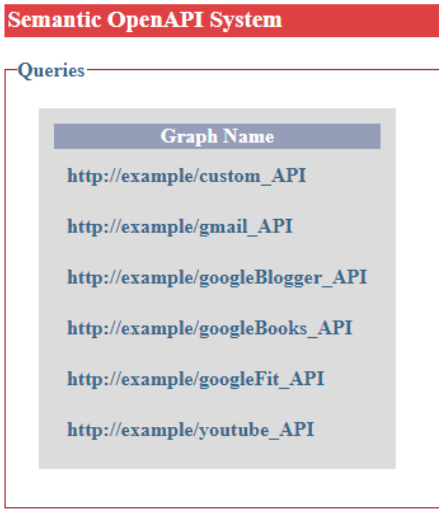


Figure 5.3: User Interface Graphs

[14] https://en.wikipedia.org/wiki/Virtuoso_Universal_Server
[15] https://virtuoso.openlinksw.com/

Next to the data graphs list, a user can find a text area where a SPARQL Query can be written. After writing the SPARQL Query, a user can submit it with the *Submit Query* button under the text area. Upon submitting the query, next to the SPARQL Query text area, another one will show up containing the results of the Query. If the user has made a mistake inside the Query, an appropriate message with the exact error will get returned as the answer. Also, the user does not have to rewrite the entire SPARQL Query from scratch as the Query will not get erased upon returning the answer. If the SPARQL Query is correct, the answer is divided in the variables that the user chose inside the *SELECT* clause of the Query. As we previously mentioned, the SPARQL Queries are performed inside the Virtuoso Database. However, the return format might be confusing for new users. This is why we chose to return the answer as a key – value pair. This gives the opportunity to work faster and perform many SPARQL Queries as well as receive the answer in a clear text format. In Figure 5.4 we showcase a Query where it returns all the service titles from the available graphs inside the database.

Figure 5.4: User Interface Queries

Finally, we present the menu of the Web Application. The *Home* page is where a user can upload a REST service description. The *Ontologies* page is where all the available ontologies are listed and the *Queries* page is where a user can perform SPARQL Queries. The *Paper* option allows the user to download the published article of the present work.

Figure 5.5: User Interface Menu

5.3 Services and SPARQL Queries

In this section we present realistic examples of using the algorithm. In order to show a full representation of our work we chose some of the most mainly used services of Google taken from the Google APIs Explorer[16]. The Google services first were written in an OpenAPI description and then partially annotated to show the full potential of the algorithm. All of the OpenAPI descriptions were loaded in our Web Application. In addition, the SPARQL Queries took place in the Web App in order to simulate the use of the system for a developer who is in search of generic service information or specific endpoints of services.

Most of our SPARQL Queries follow the same logic. The scenario is that a developer is looking for a Web Service by searching a semantic value which should be related with the relevant Web Service description. The user in most cases first retrieves some generic information about the Web Service which is related to the semantic value and then proceeds with more informative SPARQL Queries. We chose this approach in order to go step by step on the complexity of SPARQL Queries presented in this section and also, to follow a realistic scenario for new users. Although, this is not mandatory. A user can perform all types of SPARQL Queries and retrieve as many data as possible. Also, for ease of presentation and in order to avoid duplication we present in Listing 5.1 the prefixes for every SPARQL Query used in this section. The prefixes listed below are pretty common. Firstly, we define our OpenAPI ontology with the prefix *openapi*. Then, we continue with the well-known ontologies of *rdf* (RDF), *rdfs* (RDF Schema), *sh* (SHACL) and *owl* (OWL).

```
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:  <http://www.w3.org/2000/01/rdf-schema#>
PREFIX sh:    <http://www.w3.org/ns/shacl#>
PREFIX owl:   <http://www.w3.org/2002/07/owl#>
```

Listing 5.1: SPARQL Prefixes

The first service we present is the Google Books API[17]. This Web API allows a developer to bring Google Books features to a site or an application. Among other endpoints, the Google Books API contains an endpoint where a user can reach and retrieve a specific BookSelf (a Schema Object in the OpenAPI document) which represents a collection of data about several books. The Schema Object *BookSelf* is annotated with the *https://schema.org/Book* semantic value. This Schema Object is returned as response to a *GET* method on path */users/{userID}/bookshelves/{shelf}.* In Figure 5.6 we present the details of the API endpoint under examination.

[16] https://developers.google.com/apis-explorer

[17] https://developers.google.com/books/docs/overview

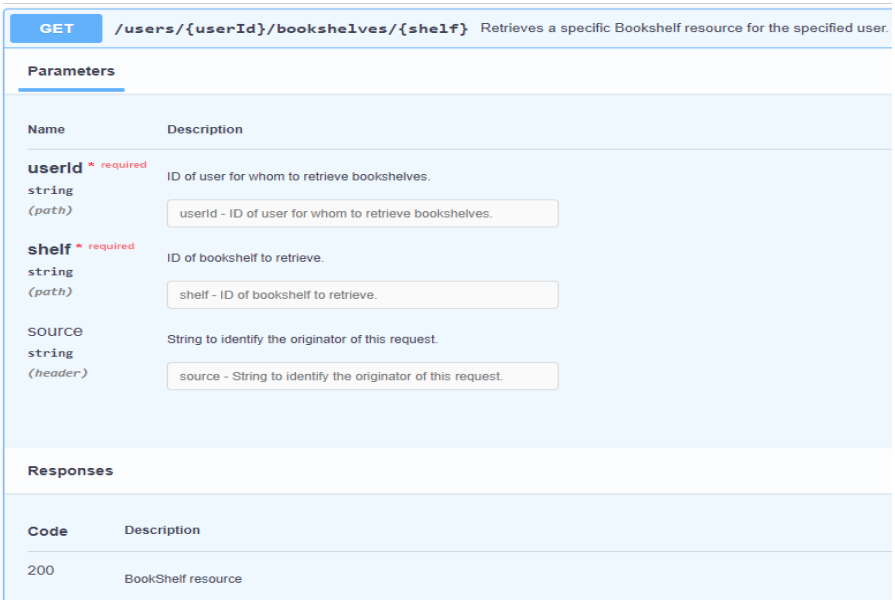


Figure 5.6: Google Books BookSelf

In Listing 5.2 we present the SPARQL Query along with the returned answer. The *SELECT* clause of the Query asks for the *name* and *external URL* of the Web Service that contains the semantic value *https://schema.org/Book.* Also, it retrieves the *method* and *path* of the endpoint related to the semantic value. In this case the semantically annotated Schema Object is return as a response. As we can see in Appendix A.1.1, the *BookSelf Schema Object* contains an *x-kindOf* property and as a value the *https://schema.org/Book*. This means that the class created for the *BookSelf Schema Object* will become a subclass of the semantic value. This class is addressed in Line 3 of the SPARQL Query below. We then proceed to return the service name using the *openapi:serviceTitle* property as well as the external document of the Web Service using the *openapi:url* property. Continuing, we address the Node Shape (line 7) created for the current Schema Object which is the *openapi:schema* of the response's content. Working our way up, from content to response and from response to operation, we retrieve the operation method as well as the path of the endpoint. In the last lines of Listing 5.2 we can see the returned answer which is consistent with the information of the endpoint as seen in Figure 5.6.

```
1:  SELECT ?name ?externalURL ?method ?pathName

2:  WHERE {
3:  GRAPH ?g {?class rdfs:subClassOf <https://schema.org/Book> .
4:  ?service_info openapi:serviceTitle ?name .
5:  ?service_externalDoc a openapi:ExternalDoc .
6:  ?service_externalDoc openapi:url ?externalURL .
7:  ?node sh:targetClass ?class .
8:  ?content openapi:schema ?node .
9:  ?response openapi:content ?content .
10: ?operation openapi:response ?response .
11: ?operation openapi:method ?method .
12: ?operation openapi:onPath ?path .
13: ?path openapi:pathName ?pathName} }

Answer:
Name: Google Books
ExternalURL: https://developers.google.com/books/docs/v1/reference
Method: GET
PathName: /users/{userId}/bookshelves/{shelf}
```

Listing 5.2: *https://schema.org/Book* name, extURL, method, path

The next SPARQL Query is based on the same semantic value as the previous. Although, this time we retrieve

more information about the specified endpoint. The *SELECT* clause in line 1 asks for *parameter name* and *parameter description* of the endpoint related to the semantic value. This Query is similar to the previous one, for the most part. After, retrieving the operation (line 7) we then proceed with its parameters line 8 – 10. In line 11, we narrow our returned answer to only required parameters. This is why the parameter *source* as seen in Figure 5.6 is missing from our answer. In the last lines of Listing 5.3 we can see the returned answer which contains the name and description of the two required parameters for the endpoint.

```
1:  SELECT ?paramName ?paramDescription

2:  WHERE {
3:  ?class rdfs:subClassOf <https://schema.org/Book> .
4:  ?node sh:targetClass ?class .
5:  ?content openapi:schema ?node .
6:  ?response openapi:content ?content .
7:  ?operation openapi:response ?response .
8:  ?operation openapi:parameter ?parameter .
9:  ?parameter openapi:description ?paramDescription .
10: ?parameter openapi:name ?paramName .
11: ?parameter openapi:required true } }

Answer:
1. Parameter Name: userId
   Parameter Description: ID of user for whom to retrieve bookshelves.

2. Parameter Name: shelf
   Parameter Description: ID of bookshelf to retrieve.
```

Listing 5.3: *https://schema.org/Book* parameter name and description

Another endpoint of the Google Books API allows the user to retrieve a volume resource. A volume resource contains information about a single book. The *Schema Object Volume* is returned as a response to the endpoint in Figure 5.7. The path of the endpoint is the */volumes/{volumeId}* which contains the required parameter *volumeId*. Also, the method for the endpoint is the *GET* method. In Appendix A.1.2, we can see the *Volume Schema Object.* This Schema Object, contains a nested property *PDF* which is annotated with the *https://schema.org/DigitalDocument* semantic value through the extension property *x-kindOf.*

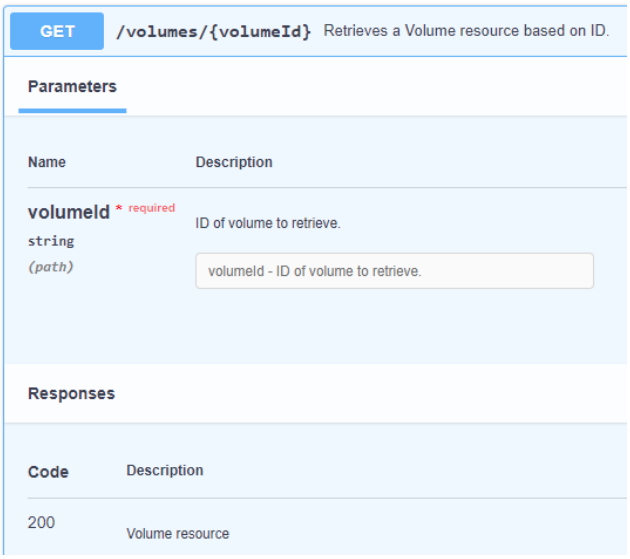


Figure 5.7: Google Books Volume

As the previous examples, the SPARQL Query in Listing 5.4 searches for a Web Service related to semantic value. This semantic value is *https://schema.org/DigitalDocument* and exists in the property *PDF* of the *Volume Schema Object.* The *SELECT* clause in line 1 contains several variables. First, asks for the name and external URL of the Web Service. Next, it retrieves information about the response containing the Schema Object as well as the

method, the path and the summary of the endpoint. In line 3 of the Query, we target the graph (ontology) which contains the semantic value as a subproperty. This is because the *PDF* property contains an *x*-kindOf and therefore, the property created for *PDF* will become a subproperty of the semantic value. Continuing, in line 4 – 6 we retrieve the external document and name of the Web Service. Because the *PDF* property is a nested property, this means that one or more Schema Objects are placed between the property and the original Schema Object. This is showcased in lines 7 – 10. Finally, we are able to reach the content of the response and from there the operation so as to retrieve all the needed information. The answer at the bottom of Listing 5.4 is consistent with the information of the endpoint in Figure 5.7.

```
1:  SELECT ?name ?externalURL ?respDescription ?respStCode ?method ?pathName ?summary

2:  WHERE {
3:  GRAPH ?g {?property rdfs:subPropertyOf <https://schema.org/DigitalDocument> .
4:  ?service_info openapi:serviceTitle ?name .
5:  ?service_externalDoc a openapi:ExternalDoc .
6:  ?service_externalDoc openapi:url ?externalURL .
7:  ?propertyShape sh:path ?property .
8:  ?node sh:property ?propertyShape .
9:  ?upperNode sh:node ?node .
10: ?schemaNode sh:node ?upperNode .
11: ?content openapi:schema ?schemaNode .
12: ?response openapi:content ?content .
13: ?response openapi:description ?respDescription .
14: ?response openapi:statusCode ?respStCode .
15: ?operation openapi:response ?response .
16: ?operation openapi:method ?method .
17: ?operation openapi:summary ?summary .
18: ?operation openapi:onPath ?path .
19: ?path openapi:pathName ?pathName .
20: ?operation openapi:summary ?summary} }

Answer:
Name: Google Books
externalURL: https://developers.google.com/books/docs/v1/reference
Response Description: Volume resource
Response Status Code: 200
Method: GET
Path: /volumes/{volumeId}
Summary: Retrieves a Volume resource based on ID.
```

Listing 5.4: *https://schema.org/DigitalDocument* name, extURL, response description, code method, path name, summary

The next Web Service we present in this chapter is the Google Blogger API[18]. According to its description, the Google Blogger API allows client applications to view and update Blogger content. The endpoints that concern us are the ones that return a blog resource as a response. The paths of these endpoints are */blogs/{blogId}* and */blogs/byurl* and are presented in Figure 5.8 and 5.9 respectively. In the Figures below, we can see that both endpoints are accessible by the *GET* operation method. The first endpoint retrieves a blog by its id and the second retrieves a blog by URL. The response that a client gets by calling these endpoints is a Schema Object with the name *Blog*. The *Blog Schema Object* is semantically annotated with the value *https://schema.org/Blog* through the *x-kindOf* property as we can see in Appendix A.2.1.

[18] https://developers.google.com/blogger/docs/3.0/reference

Figure 5.8: Google Blogger Blog by ID

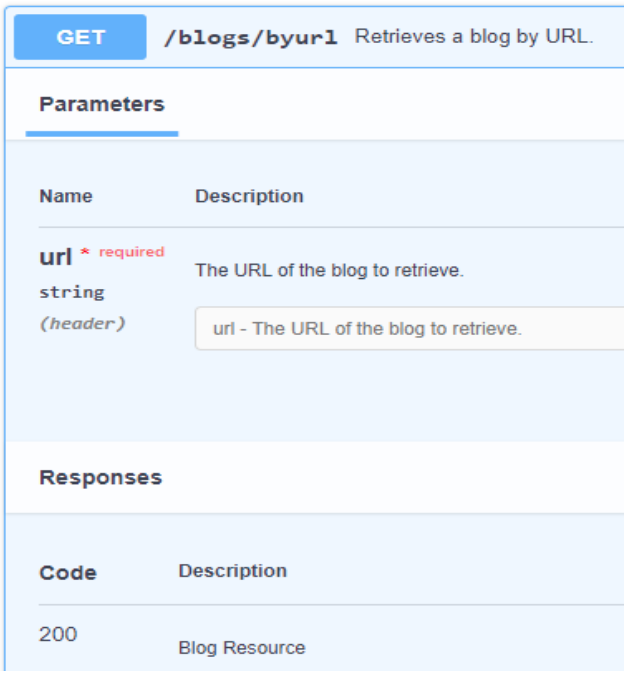


Figure 5.9: Google Blogger Blog by URL

As we can see, both endpoints require a parameter in order to return the blog resource. Although, the *blogId* parameter of the first endpoint is a path parameter and the *URL* parameter of the second endpoint is a header parameter. In order to get both parameters from a single SPARQL Query we need to use the UNION keyword. That is because an operation contains all its path parameters under the *openapi:parameter* property and all its header parameters under the *openapi:requestHeader* parameter. In Listing 5.5 we present the Query under examination.

The *SELECT* clause of the SPARQL Query returns general information about the Web Service as well as specific information about the endpoints. In line 1 of Listing 5.5 we have variables about the name and external URL of the service, the method, the path name and the summary of the operation and also the name and the description of the parameters. In lines 3 we specify the semantic value we are interested in and we use the *rdfs:subClassOf* predicate because the *Blog Schema Object* contains the *x-kindOf* property and therefore its class is a subclass of the semantic value. Continuing, in lines 4 – 6 we retrieve the name and external URL of the Web Service. Next, by specifying the Node Shape created for the Schema Object (line 7) we get to the content and response that this Schema Object is used (line 8 – 9). Continuing, in lines 10 – 14 we get the information about the path name, the summary and method of the operation. Next, is the point where the two operations of the two endpoints differ and the reason we use the keyword UNION in the Query. In the first part of the Query, in lines 15 – 17 we get the name and description of the path parameter using the predicate *openapi:parameter* which detects only path parameters. In the second part of the Query instead, we get information about the parameter with the *openapi:requestHeader* predicate which is used for header parameters.

The answer lies within the last lines of Listing 5.5. The information we get from the SPARQL Query is consistent with the information from Figures 5.8 and 5.9. In particular, we get the path of the endpoints which are */blogs/{blogId}* and */blogs/blogurl* and also, we get the request method which is *GET* for both endpoints. In addition, we get the parameter variables *blogId* and *URL* along with their descriptions and also the summary of each operation.

```
1:  SELECT ?name ?externalURL ?method ?pathName ?summary ?paramName ?paramDescription

2:  WHERE  {
3:  GRAPH ?g {?class rdfs:subClassOf <https://schema.org/Blog> .
4:  ?service_info openapi:serviceTitle ?name .
5:  ?service_externalDoc a openapi:ExternalDoc .
6:  ?service_externalDoc openapi:url ?externalURL .
7:  ?node sh:targetClass ?class .
8:  ?content openapi:schema ?node .
```

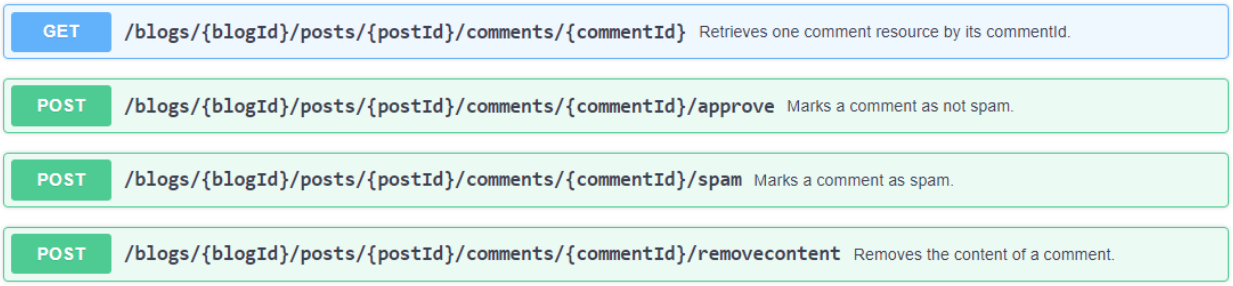
```
9:  ?response openapi:content ?content .
10: ?operation openapi:response ?response .
11: ?operation openapi:method ?method .
12: ?operation openapi:summary ?summary .
13: ?operation openapi:onPath ?path .
14: ?path openapi:pathName ?pathName .

15: {?operation openapi:parameter ?parameter .
16: ?parameter openapi:name ?paramName .
17: ?parameter openapi:description ?paramDescription}

18: UNION

19: {?operation openapi:requestHeader ?parameter .
20: ?parameter openapi:name ?paramName .
21: ?parameter openapi:description ?paramDescription} } }

Answer:
Name: Google Blogger
External URL: https://developers.google.com/blogger/docs/3.0/reference

1. Method: GET
   Path name: /blogs/{blogId}
   Summary: Retrieves a blog by its ID.
   Parameter Name: blogId
   Parameter Description: The ID of the blog to get.

2. Method: GET
   Path name: /blogs/byurl
   Summary: Retrieves a blog by URL.
   Parameter Name: URL
   Parameter Description: The URL of the blog to retrieve.
```

Listing 5.5: *https://schema.org/Blog* name, extURL, method, path name, summary, parameter name and description

The next SPARQL Query is based on the semantic value *https://schema.org/comment*. This semantic value is used to annotate the *Schema Object Comments* (Appendix A.2.2). This Schema Object is returned as a response on several endpoints of the Google Blogger API. Although, this semantic value is commonly used on other Web Service descriptions which contain some kind of comment resource. Another API which contains this semantic value to annotate a Schema Object is the YouTube API[19]. This API contains the *Comment Schema Object* (Appendix A.3.1) which is also returned as a response on some endpoints. The two Schema Objects, *Comments* of Google Blogger and *Comment* of YouTube API have different properties but are annotated with the same semantic value. This gives us the opportunity to perform a SPARQL Query and get as a return value information about both Web Services.

In Figures 5.10 and 5.11 we present the endpoints associated with the semantic value under examination. For ease of presentation, in the following Figures we present only the path, the method and the summary of the endpoints and the rest information (the response of each endpoint) are available on the Appendix. Inside the body of the Schema Objects, we notice a difference in relation to the previous examples. The Schema Objects use the *x-refersTo* property instead of *x-kindOf.*

[19] https://developers.google.com/youtube/v3/docs



Figures 5.10: Google Blogger Comments

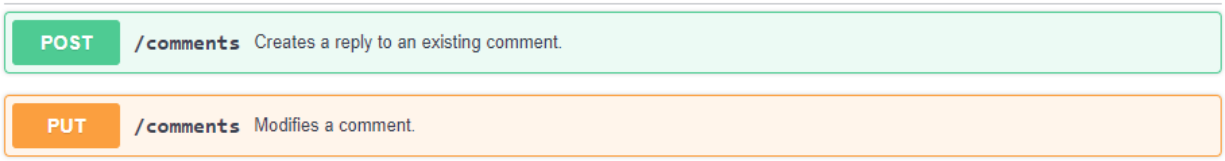


Figure 5.11: YouTube API Comment

The *SELECT* clause of the SPARQL Query below provides information about endpoints as well as the entire Web Services. In line 1 of Listing 5.6 we ask for name and external URL of each Web Service. In addition, we retrieve information about the operations of the endpoints such as summary, path name and operation method. Starting in line 3 inside the *WHERE* clause we specify the semantic value *https://schema.org/comment* as a direct class of the Node Shape of the Schema Object. We do this with the predicate *sh:targetClass.* This is because the Schema Objects related to this semantic value have an *x-refersTo* extension property and therefore, the semantic value becomes a direct class of the Node Shape. In lines 4 – 6 we retrieve general information about the Web Services (name and external URL). Then, we reach the content of the response through the Node Shape (line 7) with the *openapi:schema* predicate. From there, we get to response and then operation where we are able to retrieve the operation summary, path name and method.

The returned values of the Query are placed at the bottom of the Listing. There we can observe four endpoints from the Google Blogger API and two from the YouTube API. Along with the information about the endpoints, path name, summary, method, we also get the name and external URL for each one of the Web Services. The information inside the returned answer is consistent with Figures 5.10 and 5.11.

```
1:  SELECT ?name ?externalURL ?summary ?pathName ?method

2:  WHERE {
3:  GRAPH ?g {?node sh:targetClass <https://schema.org/comment> .
4:  ?service_info openapi:serviceTitle ?name .
5:  ?service_externalDoc a openapi:ExternalDoc .
6:  ?service_externalDoc openapi:url ?externalURL .
7:  ?content openapi:schema ?node .
8:  ?response openapi:content ?content .
9:  ?operation openapi:response ?response .
10: ?operation openapi:method ?method .
11: ?operation openapi:onPath ?path .
12: ?operation openapi:summary ?summary .
13: ?path openapi:pathName ?pathName} }

Answer:
Name: Google Blogger
External URL: https://developers.google.com/blogger/docs/3.0/reference

1. Summary: Retrieves one comment resource by its commentId.
   Path name: /blogs/{blogId}/posts/{postId}/comments/{commentId}
   Method: GET
```

```
2. Summary: Marks a comment as not spam.
   Path name: /blogs/{blogId}/posts/{postId}/comments/{commentId}/approve
   Method: POST

3. Summary: Marks a comment as spam.
   Path name: /blogs/{blogId}/posts/{postId}/comments/{commentId}/spam
   Method: POST

4. Summary: Removes the content of a comment.
   Path name: /blogs/{blogId}/posts/{postId}/comments/{commentId}/removecontent
   Method: POST

Name: YouTube API
External URL: https://developers.google.com/youtube/v3/docs

1. Summary: Modifies a comment.
   Path name: /comments
   Method: PUT

2. Summary: Creates a reply to an existing comment.
   Path name: /comments
   Method: POST
```

Listing 5.6: *https://schema.org/comment* name, extURL, method, path name and summary

Leaving the Google Blogger and YouTube API we continue with the Google Fit API. This Web Service gives us the opportunity to demonstrate the significance of the keyword *allOf* in a REST API description. As mentioned in previous chapters, the *allOf* keyword creates relations between the classes of Schema Objects that are involved with this particular keyword. When we search for a class that is created under these conditions, we can retrieve information about all the Schema Objects involved. This means, we get a wider range of information that can help the client decide which endpoint is more suitable for every given situation. It also provides more information for the particular Web Service.

In this case, the semantic value that we are interested in, is the *https://schema.org/UserInteraction*. This semantic value helps to determine user actions on a web page. Inside the Google Fit API description, the *Schema Object UseDataSourcesResource* (Appendix A.4.1) uses the *x-kindOf* property to refer to this semantic value. This Schema Object is provided as a request body for the endpoint with path */users/{userId}/dataSources* and operation method *POST*. In addition to this particular Schema Object, another one is useful to us. This is the *UseDataSourcesResourceExtra Schema Object* (Appendix A.4.1). The *UseDataSourcesResourceExtra* contains the *allOf* keyword and listed under it, the *UseDataSourcesResource*. According to what we have seen so far, the classes of these Schema Objects are related. The *UseDataSourcesResourceExtra* is also used as request body payload with the method *PUT* on the path *users/{userId}/dataSources/{dataSourceId}*. All the endpoints mentioned in this paragraph are presented in Figure 5.12 and the request bodies of the endpoints are available on the Appendix.

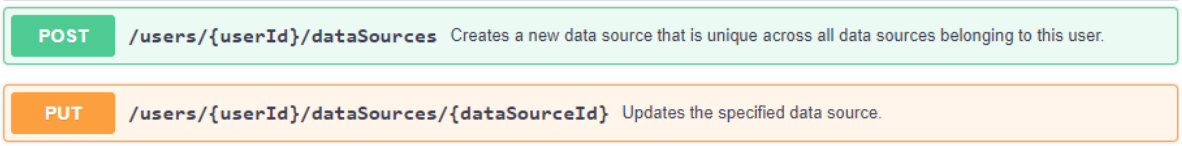


Figure 5.12: Google Fit API UserDataSourcesResource-Extra

The SPARQL Query presented in Listing 5.7 is similar to the previous ones. The variables in the *SELECT* clause retrieve information about the name and external URL of the Web Service as well information about each operation. Starting the usual way, we define the semantic value *https://schema.org/UserInteraction* as a subclass of the node (*UseDataSourcesResource*) class. Continuing, we get the class of the *UseDataSourcesResourceExtra Schema Object* which is a subclass (due to *allOf*) of the previous class. Then, after getting name and external URL, we work our way up the operation through content and request body. There, we get the rest information we

specified in the *SELECT* clause. Finally, at the bottom of the Listing, we present the returned answer which is matching with what we discussed in the previous paragraph.

```
1:  SELECT ?name ?exteranlURL ?summary ?pathName ?method WHERE  {
2:  GRAPH ?g { ?class rdfs:subClassOf <https://schema.org/UserInteraction> .
3:  ?node sh:targetClass ?class .
4:  ?classExtra rdfs:subClassOf ?class .
5:  ?nodeExtra sh:targetClass ?classExtra .
6:  ?service_info openapi:serviceTitle ?name .
7:  ?service_externalDoc a openapi:ExternalDoc .
8:  ?service_externalDoc openapi:url ?externalURL .

9:  {?content openapi:schema ?node .}
10: UNION
11: {?content openapi:schema ?nodeExtra .}

12: ?reqBody openapi:content ?content .
13: ?operation openapi:requestBody ?reqBody .
14: ?operation openapi:summary ?summary .
15: ?operation openapi:onPath ?path .
16: ?path openapi:pathName ?pathName .
17: ?operation openapi:method ?method } }

Answer:
Name: Google Fit
External URL: https://developers.google.com/fit/rest/v1/reference

1. Summary: Updates the specified data source.
   Path name: /users/{userId}/dataSources/{dataSourceId}
   Method: PUT

2. Summary: Creates a new data source that is unique across all data sources belonging to
   this user.
   Path name: /users/{userId}/dataSources
   Method: POST
```

Listing 5.7: *https://schema.org/ UserInteraction* name, extURL, method, path name, summary and status code

The last API we used to present our results is the Gmail API[20]. We use this example in order to demonstrate again the usage of the *allOf* keyword but this time in a wider range. Inside the Gmail API description we observe a Schema Object named *Message* (Appendix A.5.1). This Schema Object is semantically annotated with the value *https://schema.org/EmailMessage.* Also, the *Message Schema Object* is used in another Schema Object, the *Draft.* The *Draft* contains the keyword *allOf* and the *Message* is listed under it. The two schemas together, make up for the most responses inside the Gmail API. Also, because the two schemas are related through the *allOf* keyword we are able to get all of these responses along with other information. This information is accessible to us through a single SPARQL Query.

[20] https://developers.google.com/gmail/api/reference/rest

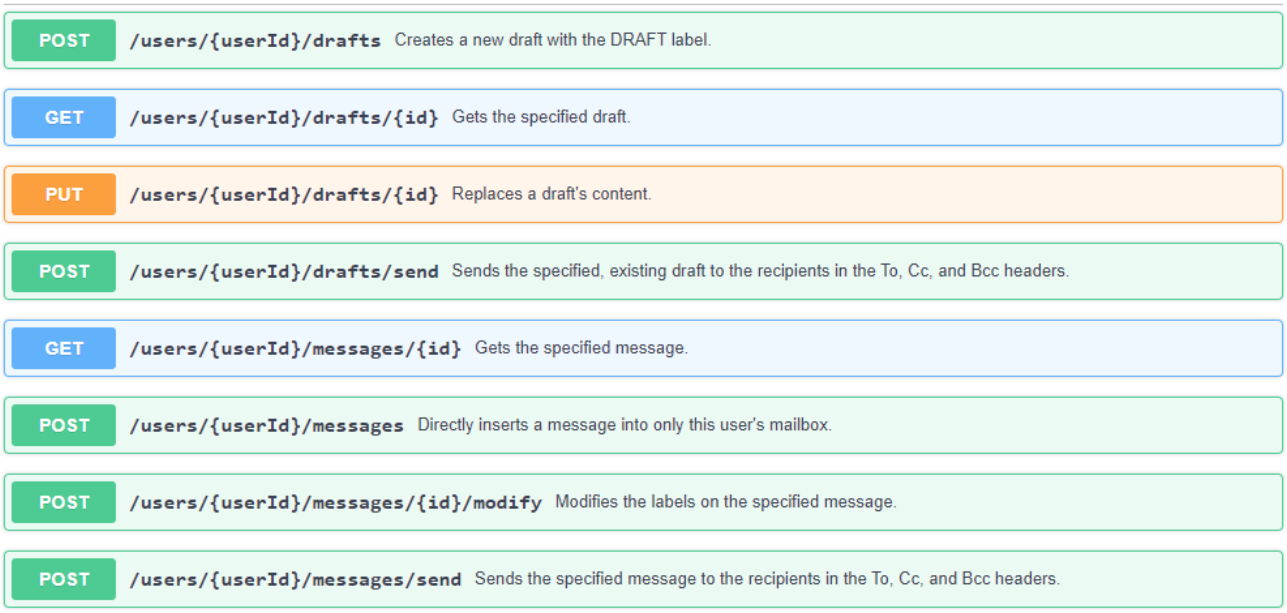


Figure 5.13: Gmail API, Draft Message

The SPARQL Query for the Gmail API is listed below. Here, we use the same structure as our previous examples and also, we retrieve the same information. The returned answer of the SPARQL Query, contains all eight API endpoints we presented in the Figure above. This demonstrates the benefits we gain from the existence of the *allOf* inside a Schema Object.

```
1:  SELECT ?name ?exteranlURL ?summary ?pathName ?method WHERE  {
2:  GRAPH ?g { ?class rdfs:subClassOf <https://schema.org/EmailMessage> .
3:  ?node sh:targetClass ?class .
4:  ?classDraft rdfs:subClassOf ?class .
5:  ?nodeDraft sh:targetClass ?classDraft .
6:  ?service_info openapi:serviceTitle ?name .
7:  ?service_externalDoc a openapi:ExternalDoc .
8:  ?service_externalDoc openapi:url ?externalURL .

9:  {?content openapi:schema ?node .}
10: UNION
11: {?content openapi:schema ?nodeDraft .}

12: ?response openapi:content ?content .
13: ?operation openapi:response ?response .
14: ?operation openapi:summary ?summary .
15: ?operation openapi:onPath ?path .
16: ?path openapi:pathName ?pathName .
17: ?operation openapi:method ?method } }

Answer:
Name: Gmail API
External URL: https://developers.google.com/gmail/api/reference

1. Summary: Gets the specified message.
   Path name: /users/{userId}/messages/{id}
   Method: GET

2. Summary: Sends the specified message to the recipients in the To, Cc, and Bcc headers.
   Path name: /users/{userId}/messages/send
   Method: POST

3. Summary: Creates a new draft with the DRAFT label.
   Path name: /users/{userId}/drafts
   Method: POST
```

```
4. Summary: Directly inserts a message into only this user`s mailbox.
   Path name: /users/{userId}/messages
   Method: POST

5. Summary: Gets the specified draft.
   Path name: /users/{userId}/drafts/{id}
   Method: GET

6. Summary: Replaces a draft`s content.
   Path name: /users/{userId}/drafts/{id}
   Method: PUT

7. Summary: Modifies the labels on the specified message.
   Path name: /users/{userId}/messages/{id}/modify
   Method: POST

8. Summary: Sends the specified, existing draft to the recipients in the To, Cc, and Bcc
   headers.
   Path name: /users/{userId}/drafts/send
   Method: POST
```

Listing 5.8: *https://schema.org/EmailMessage* name, extURL, method, path name and summary

In addition to the Google APIs, we also created a custom API with the title "Service Bundle" for demonstration purposes. The description of this API contains Schema Objects that are annotated with semantic values from other API ontologies and not from vocabularies, in contrast to what we have seen so far. In chapter 3 we presented our approach on Schema Objects that do not contain any extension properties and therefore they are not semantically annotated. These Schema Objects, even if they are not semantically annotated, they acquire an ontology class that comes from the name of each schema. One of the benefits of this approach is that a Web Service description can be annotated with ontology classes created for another Web Service. Such cases are going to be discussed in the following examples.

In our custom API, the first example that concerns us is that of the *Subscription Schema Object* (Appendix A.6.1). This Schema Object is returned as a response on the API endpoint with the path */{userId}/subscription_info*. This endpoint is reached with the *GET* method and it contains one path parameter the *userId* parameter. Information for this endpoint can be found below, in Figure 5.14 and in the Appendix. The *Subscription Schema Object* contains the *x-kindOf* property with the value of "*https://www.example.com/service/youtube_API#Subscription*". This value is divided in two parts. The first part (before the hash mark) refers to the YouTube ontology inside our Web Application. The second part (after the hash mark) refers to the *Subscription Schema Object* inside the YouTube ontology. The *Subscription Schema Object* (Appendix A.3.2) is used in the request body as well as the response body of an endpoint inside the YouTube API. The endpoint is reachable with the *POST* method on the */subscriptions* path (Figure 5.15). It also requires a query parameter with the name of *part* and is responsible for adding a subscription to a user's channel. It is important to emphasize that the *Subscription Schema Object* of the YouTube API does not contain any extension properties that semantically enrich it. Therefore, the class that is created for this Schema Object is not related to any external semantic value (i.e., from schema.org vocabulary).

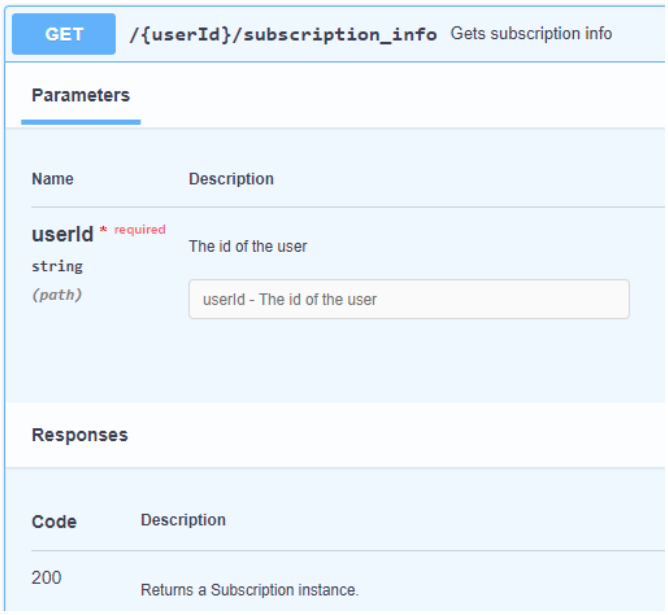


Figure 5.14: Service Bundle API, Subscription

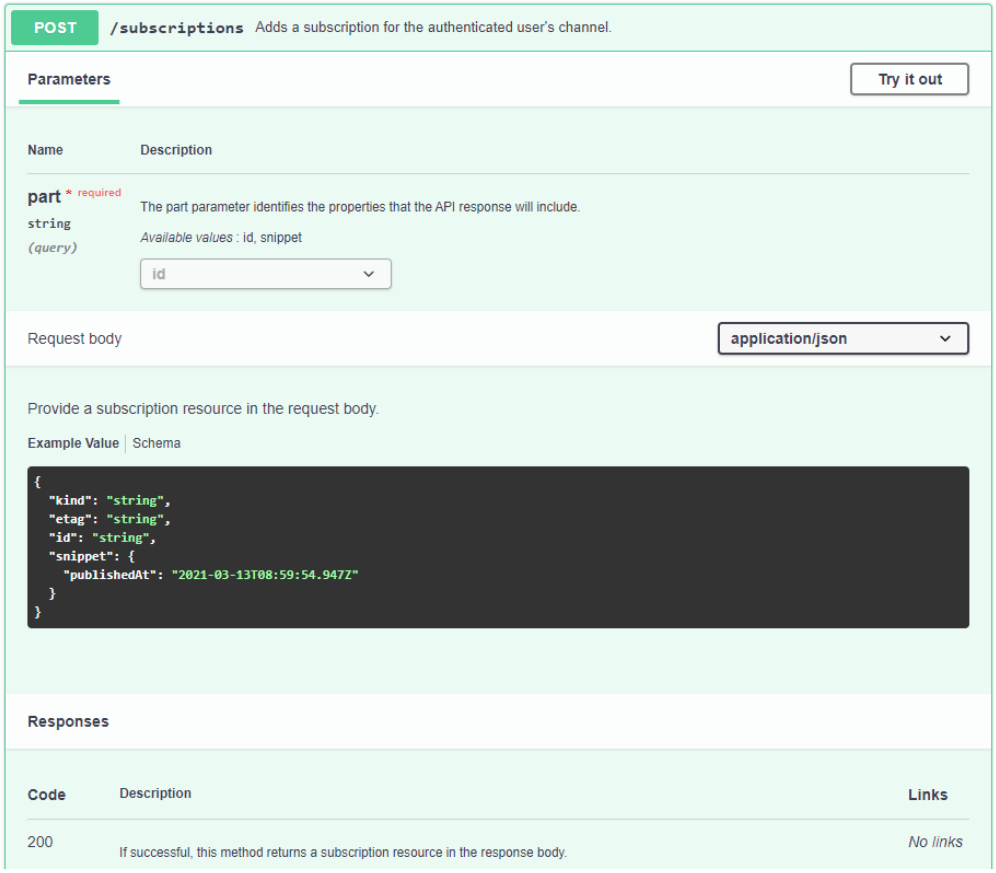


Figure 5.15: YouTube API, Subscription

The SPARQL Query for the endpoints mentioned in the previous paragraph is listed below (Listing 5.9). This Query is rather long. In line 1 of the Query, we define another prefix in addition to the prefixes we already use (Listing 5.1) in order to avoid rewriting the entire IRI in lines 3, 6 and 19. Consequently, the value

"*https://www.example.com/service/youtube_API#Subscription*" is replaced by the *val:Subscription* value. The *SELECT* clause of the Query returns plenty of information. It contains variables for the name of the graph, the title of the Web Service as well as necessary information about the operation, such as path name, method, summary and parameters. In lines 3 and 4 we look for an Owl class defined by the value we mentioned earlier and return any graph that contains this class along with the corresponding title of the Web Service.

Continuing, we have two *OPTIONAL* keywords (line 5 and line 18). We use the *OPTIONAL* keyword for two reasons. Firstly, we want to return the variables outside the keywords (i.e., graph and service name) regardless of the variables inside the keywords. Secondly, because the bodies of the Query inside the two *OPTIONAL* keywords differ. The lines 6 – 17 are responsible for retrieving information about the YouTube API. This is clear because the *val:Subscription* is a targetClass of the Node Shape (line 6). On the other hand, the body on the second *OPTIONAL* keyword is responsible for retrieving information about the Service Bundle API. This is also showcased in lines 19 and 20 where the class of the Node Shape is a subclass of *val:Subscription* due to the *x-kindOf* property inside the *Subscription Schema Object* (Service Bundle API). Besides this variation, both bodies return information about the endpoints as seen in the *SELECT clause.*

```
1:  PREFIX val: <https://www.example.com/service/youtube_API#>

2:  SELECT ?graph ?service_name ?summary ?method ?pathName ?paramName ?paramDesc
3:  WHERE {GRAPH ?graph {val:Subscription a owl:Class .
4:  ?service_info openapi:serviceTitle ?service_name .
5:  OPTIONAL {
6:  ?node sh:targetClass val:Subscription .
7:  ?content openapi:schema ?node .
8:  ?reqBody openapi:content ?content .
9:  ?operation openapi:requestBody ?reqBody .
10: ?operation openapi:summary ?summary .
11: ?operation openapi:method ?method .
12: ?operation openapi:parameter ?parameter .
13: ?parameter openapi:name ?paramName .
14: ?parameter openapi:description ?paramDesc .
15: ?operation openapi:onPath ?path .
16: ?path openapi:pathName ?pathName .
17: }
18: OPTIONAL {
19: ?class rdfs:subClassOf val:Subscription .
20: ?node sh:targetClass ?class .
21: ?content openapi:schema ?node .
22: ?response openapi:content ?content .
23: ?operation openapi:response ?response .
24: ?operation openapi:summary ?summary .
25: ?operation openapi:method ?method .
26: ?operation openapi:parameter ?parameter .
27: ?parameter openapi:name ?paramName .
28: ?parameter openapi:description ?paramDesc .
29: ?operation openapi:onPath ?path .
30: ?path openapi:pathName ?pathName .}}}
```

Answer:

```
1. Graph Name: http://example/youtube_API
   Service Name: Youtube API
   Summary: Adds a subscription for the authenticated user's channel.
   Method: POST
   Path Name: /subscriptions
   Parameter Name: part
   Parameter Description: The part parameter identifies the properties that the
   API response will include.

2. Graph Name: http://example/custom_API
   Service Name: Service Bundle
   Summary: Gets subscription info.
   Method: GET
   Path Name: /{userId}/subscription_info
   Parameter Name: userId
   Parameter Description: The id of the user.
```

Listing 5.9: *https://www.example.com/service/youtube_API#Subscription*

The Service Bundle API also contains a Schema Object related to the Google Blogger API. The *Post Schema Object* (Appendix A.6.1) is returned as a response to the */{userId}/post/{postId}* endpoint with the *DELETE* method. This endpoint contains two required parameters, the *userId* and the *postId* path parameters and it aims to delete a post. This endpoint is presented in Figure 5.16. The *Post Schema Object* contains the *x-refersTo* property with the value "*https://www.example.com/service/googleBlogger_API#Post*". The first part of this value refers to the ontology created for the Google Blogger and the second part refers to the *Post Schema Object* (Appendix A.2.3) inside the Google Blogger. The *Post Schema Object* inside the Google Blogger is also returned as a response to an API endpoint. This endpoint has the path */blogs/{blogId}/posts/{postId}* and it is reachable with the operation method *GET.* To sum up, the current example differs from the previous one mainly in terms of the extension property. The previous Schema Object in our Service Bundle (*Subscription*) used the *x-kindOf* property to refer to another Schema Object in YouTube API. The current Schema Object (*Post*) is using the *x-refersTo* extension property to refer to the *Post Schema Object* of the Google Blogger API. This difference is showcased inside the SPARQL Query in Listing 5.10.

Figure 5.16: Service Bundle API, Post

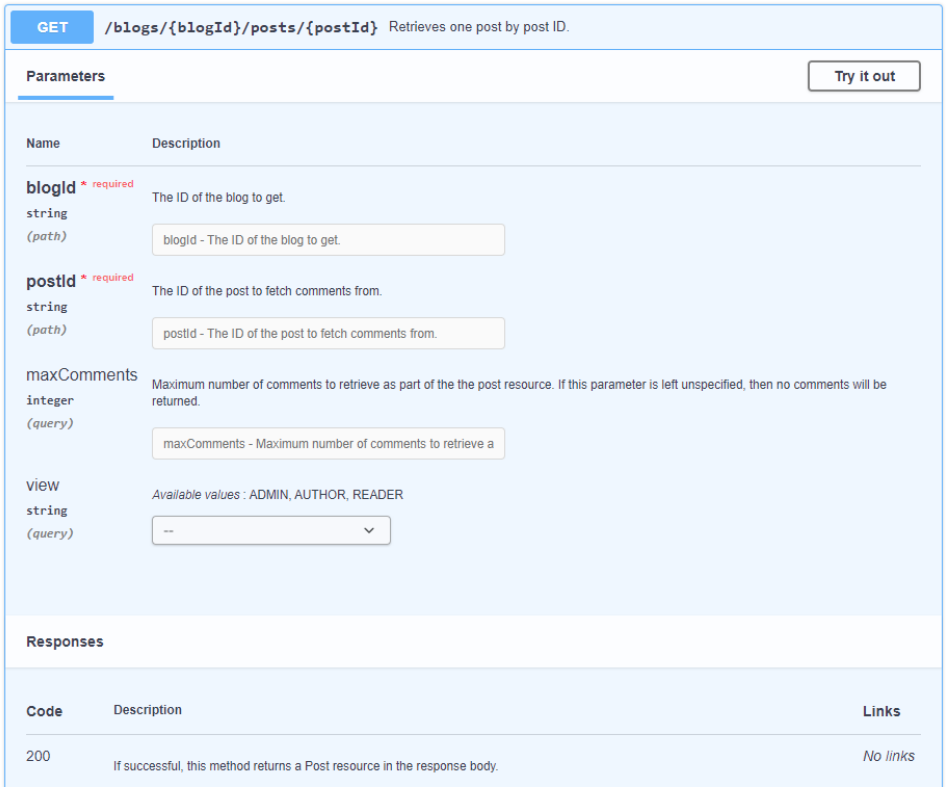


Figure 5.17: Google Blogger API, Post

In a similar manner as the previous SPARQL Query, we add an extra prefix to the existing ones (Listing 5.1). In line 1, we assign the "*val*" variable to the "*https://www.example.com/service/googleBlogger_API#*" value in order to search the corresponding class of the Node Shape by using the term "*val:Post*". In line 2 we present the *SELECT* clause which returns the graph name, the service name as well as information about the operation and parameters of the endpoint related to the current value. In line 5 we search for the value under examination as a sh:targetClass of the Node Shape. This is because in the Service Bundle it is used inside an *x-refersTo* property and in Google Blogger it represents the Node Shape class. Continuing, we get all the necessary information and when it comes to parameters, we return only the required ones (line 16). The answer at the bottom of the Listing is consistent to the data of the two endpoints above.

```
1:  PREFIX val: <https://www.example.com/service/googleBlogger_API#>

2:  SELECT ?graph ?service_name ?summary ?method ?pathName ?paramName

3:  WHERE  {
4:  GRAPH ?graph {
5:  ?node sh:targetClass val:Post .
6:  ?service_info openapi:serviceTitle ?service_name .
7:  ?content openapi:schema ?node .
8:  ?response openapi:content ?content .
9:  ?operation openapi:response ?response .
10: ?operation openapi:summary ?summary .
11: ?operation openapi:method ?method .
12: ?operation openapi:onPath ?path .
13: ?path openapi:pathName ?pathName .
14: ?operation openapi:parameter ?parameter .
15: ?parameter openapi:name ?paramName .
16: ?parameter openapi:required true .
17: }
18: }
```

```
Answer:
1. Graph Name: http://example/googleBlogger_API
   Service Name: Google Blogger
   Summary: Retrieves one post by post ID.
   Method: GET
   Path Name: /blogs/{blogId}/posts/{postId}
   a. Parameter Name: blogId
   b. Parameter Name: postId

2. Graph Name: http://example/custom_API
   Service Name: Service Bundle
   Summary: Deletes a post.
   Method: DELETE
   Path Name: /{userId}/post/{postId}
   a. Parameter Name: userId
   b. Parameter Name: postId
```

Listing 5.10: *https://www.example.com/service/googleBlogger_API#Post*

5.4 Run-Time Performance

In this section we analyze the run-time performance of several SPARQL Queries that were executed inside our Web Application. According to this paper [3], the run time efficiency of a SPARQL Query depends two factors. The first is the size of the dataset (i.e., the size of the ontology graph). The second factor is the pattern of the SPARQL Query (i.e., the SPARQL expression). Consequently, the larger the graph and the larger the query pattern, the longer it will take for the answer to return.

In Table 5.1 we present the run-time performance of all the above SPARQL Queries. At the time the SPARQL Queries were performed the database of our Web Application contained 20 graphs (ontologies). The OpenAPI description that were instantiated to ontologies were taken from the Google API Explorer as well as the source for REST API specifications for Microsoft Azure[21]. The table contains the Listings where every SPARQL Query is presented along with their response time. In addition, it contains the number of triples of each pattern, and also any SPARQL operator that was used.

Table 5.1: SPARQL Queries run-time performance

SPARQL Query	Time (ms)	Triples	Operator
Listing 5.2	568	11	–
Listing 5.3	240	9	–
Listing 5.4	800	18	–
Listing 5.5	1600	18	UNION
Listing 5.6	400	11	–
Listing 5.7	3460	15	UNION
Listing 5.8	3470	15	UNION
Listing 5.9	1260	25	OPTIONAL
Listing 5.10	384	12	–

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

From the above table, some of the SPARQL Queries stand out. Starting with the Query of Listing 5.4, its pattern contains 18 triples that define the SPARQL expression. Next, is the Query of Listing 5.5 which has the keyword *UNION* in it. According to the paper, the keywords *UNION* and *OPTIONAL* (among others) need more time to be processed, therefore the answer is delayed. This is also the case in Listings 5.7 and 5.8. Additionally, in Listing 5.9 where the keyword *OPTIONAL* is used, we also observe some delay. In conclusion, with our Queries we managed to show the response times of both conjunction (triples that are connected with the period symbol) and disjunction (keyword *UNION*).

Chapter 6

Conclusion and Future Work

6.1 Conclusions

Improving the instantiation algorithm of [1] is the main focus of this work. Our approach introduces many modifications and additions. By implementing the newly introduced keywords of OpenAPI Specification v3.0 we were able to provide more flexibility and accuracy for describing OpenAPI services. With the current algorithm, users are able to take advantage of the full potential of the OpenAPI format. In addition, they are able to combine OpenAPI schemas for model composition, a feature that it is often necessary on modern Web service descriptions.

Concerning the ontology, we introduced the concept of polymorphism and the concept of inheritance. These features have a great impact on our ontology since they explore the full potential of classes, entities and properties that are created by the algorithm. In addition, we implemented a more efficient way to semantically enrich an OpenAPI description. With the current work, the classes which originate from a Web service description can be used to semantically annotate another. This feature, contributes to the expansion of the Ontology Vocabulary and widens the range of discoverability for Web Services.

In addition to the instantiation algorithm, we made a Web Application where our mechanism can be tested by the community and give us feedback and evaluation of our work. It is a useful tool for developers who wish to implement endpoints of other APIs on their application as well as providing their own to the community. Also, the Web Application will give us the direction of our future strategy upon this work.

6.2 Summary

In this section we summarize the contributions of this work and we discuss certain aspects of the OpenAPI Specification v3.0 that are not yet supported by our algorithm. Regarding OpenAPI Schema Objects we showcased a variety of new additions, some, in combination with our extension properties. The following list presents all the modifications that took place in this work.

- Instantiation of keywords *allOf, anyOf, oneOf* in Schema Objects. This includes model composition, inheritance and polymorphism support. This category also includes all the possible combinations between these keywords and the extension properties that concern Schema Objects such as *x-refersTo, x-kind-Of* and *x-mapsTo*.

- Instantiation of keywords *anyOf, oneOf* and "*not*" in the property schemas of an OpenAPI Object. This includes polymorphism support. Also in this category, all the possible combinations between the keywords and the extension properties *x-kindOf, x-refersTo* and *x-mapsTo* are included.

In conclusion, we managed to fully support the instantiation of a Schema Object inside an OpenAPI description. Not only with the keywords that the OpenAPI Specification v3.0 provides, but also in combination with our extension properties. In particular, the algorithm can handle any case of model composition or model polymorphism with or without the extension properties.

Additionally, we created a Web Application that puts our algorithm to use and manages to support the translation of OpenAPI descriptions to instances of the OpenAPI ontology. The Web Application takes as input an OpenAPI description and produces the corresponding instance of the OpenAPI Ontology. This Web Application has been tested on several Google and Azure API Services. Also, it provides a mechanism that supports SPARQL Queries on all available ontologies stored inside the Web Application Database. Lastly, in this thesis we showcased all the OpenAPI Objects that our algorithm handles such as Operation, Parameter, Response etc.

6.3 Future Work

Regarding the OpenAPI Specification v3.0 the Link Object and the Callback Object are transferred for future work. 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. Links, enable 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 OpenAPI 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.

This work has already proved its usefulness on service discovery through SPARQL queries. A query language that simplifies the complex SPARQL queries will become a great addition to our service discovery purpose. In this way the task of finding appropriate endpoints for a Web service will become much easier and approachable for a developer.

Appendix A

OpenAPI Descriptions

A.1 Google Books API

A.1.1 Bookself

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Google Books
  description: The APIs in the Google Books API Family let you bring Google Books features
to your site or application
  termsOfService: https://developer.google.com/books/terms.html
  license:
    name: Apache 2.0
    url: https://creativecommons.org/licenses/by/4.0
externalDocs:
  description: Find more info here
  url: https://developers.google.com/books/docs/v1/reference/bookshelves
servers:
  - url: https://www.googleapis.com/books/v1
paths:
  /users/{userId}/bookshelves/{shelf}:
    get:
      summary: Retrieves a specific Bookshelf resource for the specified user.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
        - $ref: "#/components/parameters/shelfParam"
        - $ref: "#/components/parameters/sourceParam"
      responses:
        "200":
          description: BookShelf resource
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/BookSelf"
components:
  parameters:
    userIdParam:
      name: userId
      description: ID of user for whom to retrieve bookshelves.
      in: path
      required: true
      schema:
        type: string

    shelfParam:
      name: shelf
      description: ID of bookshelf to retrieve.
      in: path
      required: true
      schema:
        type: string

    sourceParam:
      name: source
```

```
    description: String to identify the originator of this request.
    in: header
    required: false
    schema:
      type: string

schemas:
  BookSelf:
    x-kindOf: https://schema.org/Book
    type: object
    description: A Bookshelf resource represents the metadata for a bookshelf, it does
                 not include the volumes in the bookshelf.
    required:
      - kind
      - id
      - title
      - description
      - access
      - updated
      - created
      - volumeCount
      - volumesLastUpdated
      - selfLink
    properties:
      kind:
        description: Resource type for bookshelf metadata.
        type: string
      id:
        description: ID of this bookshelf.
        type: integer
      title:
        description: Title of this bookshelf.
        type: string
      description:
        description: Description of this bookshelf.
        type: string
      access:
        description: Whether this bookshelf is PUBLIC or PRIVATE.
        type: string
      updated:
        description: Last modified time of this bookshelf (formatted UTC timestamp with
                     millisecond resolution).
        type: string
        format: date-time
      created:
        description: Created time for this bookshelf (formatted UTC timestamp with mil
                     lisecond resolution).
        type: string
        format: date-time
      volumeCount:
        description: Number of volumes in this bookshelf.
        type: integer
      volumesLastUpdated:
        description: Last time a volume was added or removed from this bookshelf (for
                      matted UTC timestamp with millisecond resolution).
        type: string
        format: date-time
      selfLink:
        description: URL to this resource.
        type: string
```

A.1.2 PDF

```
...
  /volumes/{volumeId}:
    get:
      summary: Retrieves a Volume resource based on ID.
      parameters:
        - $ref: "#/components/parameters/volumeIdParam"
      responses:
        "200":
          description: Volume resource
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Volume"
components:
    schemas:
        Volume:
          type: object
          description: A Volume collection is used to perform a search or listing the con
                       tents of a bookshelf. This collection is a read-only collection.
          properties:
            ...
            pdf:
              x-kindOf: https://schema.org/DigitalDocument
              description: Information about pdf content. (in LITE projection).
              type: string
            ...
```

A.2 Google Blogger API

A.2.1 Blog

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Google Blogger
  description: The Blogger API v3 allows client applications to view and update Blogger
               content. Your client application can use Blogger API v3 to create new blog
               posts, edit or delete existing posts, and query for posts that match par
               ticular criteria.
  termsOfService: https://developer.google.com/books/terms.html
  license:
    name: Apache 2.0
    url: https://creativecommons.org/licenses/by/4.0
externalDocs:
  description: Find more info here
  url: https://developers.google.com/blogger/docs/3.0/reference
servers:
  - url: https://www.googleapis.com/blogger/v3
paths:
  /blogs/{blogId}:
    get:
      summary: Retrieves a blog by its ID
      parameters:
        - $ref: "#/components/parameters/blogIdParam"
      responses:
        "200":
          description: Blog Resource
```

```yaml
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Blog"
  /blogs/byurl:
    get:
      summary: Retrieves a blog by URL.
      parameters:
        - name: url
          description: The URL of the blog to retrieve.
          in: header
          required: true
          schema:
            type: string
      responses:
        "200":
          description: Blog Resource
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Blog"
components:
    parameters:
        blogIdParam:
          name: blogId
          description: The ID of the blog to get.
          in: path
          required: true
          schema:
            type: string
    schemas:
        Blog:
          x-kindOf: https://schema.org/Blog
          type: object
          description: A blog is the root data class for the Blogger API. Each blog has a
                       series of posts and pages, and each post has a series of comments.
          required:
            - kind
            - id
            - name
            - description
            - published
            - updated
            - url
            - selfLink
            - posts
            - locale
            - customerMetaData
            - pages
          properties:
            kind:
              type: string
              description: The kind of this entry. Always blogger#blog.
            id:
              type: string
              description: The ID for this resource.
            name:
              type: string
              description: The name of this blog, which is usually displayed in Blogger as
                           the blog's title. The title can include HTML.
            description:
              type: string
              description: The description of this blog, which is usually displayed in
                           Blogger underneath the blog's title. The description can include
                           HTML.
            publised:
              x-kindOf: https://schema.org/datePublished
              type: string
```

```
      format: date-time
      description: RFC 3339 date-time when this blog was published.
    updated:
      type: string
      format: date-time
      description: RFC 3339 date-time when this blog was published.
    url:
      type: string
      description: The URL where this blog is published.
    selfLink:
      type: string
      description: The Blogger API URL to fetch this resource from.
    posts:
      type: object
      description: The container for this blog's posts.
      required:
        - totalItems
        - selfLink
        - items
      properties:
        totalItems:
          type: integer
          description: The total number of posts on this blog.
        selfLink:
          type: string
          description: The URL of the collection of posts for this blog.
        items:
          type: array
          description: The list of posts for this Blog.
          items:
            type: object
    locale:
      type: object
      description: The locale this blog is set to, as broken out below.
      required:
        - language
        - country
        - variant
      properties:
        language:
          type: string
          description: The language this blog is set to, for example "en" for Eng
                       lish.
        country:
          type: string
          description: The country variant of the language, for example "US" for
                         American English.
        variant:
          type: string
          description: The language variant this blog is set to.
    customMetaData:
      type: string
      description: The JSON custom metadata for the blog.
    pages:
      type: object
      description: The container for this blog's pages.
      required:
        - totalitems
        - selfLink
      properties:
        totalItems:
          type: integer
          description: The total number of pages for this blog.
        selfLink:
          type: string
          description: The URL of the pages collection for this blog.
```

A.2.2 Comments

```
...
/blogs/{blogId}/posts/{postId}/comments/{commentId}:
    delete:
      summary: Delete a comment by ID.
      parameters:
        - $ref: "#/components/parameters/blogIdParam"
        - $ref: "#/components/parameters/postIdParam"
        - $ref: "#/components/parameters/commentIdParam"
      responses:
        "200":
          description: If successful, this method returns an empty response body.
    get:
      summary: Retrieves one comment resource by its commentId.
      parameters:
        - $ref: "#/components/parameters/blogIdParam"
        - $ref: "#/components/parameters/postIdParam"
        - $ref: "#/components/parameters/commentIdParam"
      responses:
        "200":
          description: If successful, this method returns a response body with the follow
                       ing structure.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Comments"
/blogs/{blogId}/posts/{postId}/comments/{commentId}/approve:
    post:
      summary: Marks a comment as not spam.
      parameters:
        - $ref: "#/components/parameters/blogIdParam"
        - $ref: "#/components/parameters/postIdParam"
        - $ref: "#/components/parameters/commentIdParam"
      responses:
        "200":
          description: If successful, this method returns a response body with the follow
                       ing structure.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Comments"
/blogs/blogId/posts/postId/comments/commentId/removecontent:
    post:
      summary: Removes the content of a comment.
      parameters:
        - $ref: "#/components/parameters/blogIdParam"
        - $ref: "#/components/parameters/postIdParam"
        - $ref: "#/components/parameters/commentIdParam"
      responses:
        "200":
          description: If successful, this method returns a response body with the follow
                       ing structure.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Comments"
components:
  schemas:
    Comments:
      x-refersTo: https://schema.org/comment
      type: object
      properties:
        kind:
          description: The kind of this resource. Always blogger#comment.
```

```
      type: string
    id:
      description: The ID for this resource.
      type: string
    post:
      description: Data about the post containing this comment.
      type: object
      properties:
        id:
          description: The identifier of the post containing this comment.
          type: string
    blog:
      description: Data about the blog containing this comment.
      type: object
      properties:
        id:
          description: The identifier of the blog containing this comment.
          type: string
    published:
      description: RFC 3339 date-time date-time when this comment was published, for
                   example "2012-04-15T19:38:01-07:00".
      type: string
      format: date-time
    updated:
      description: RFC 3339 date-time when this comment was last updated, for example
                   "2012-04-15T19:43:21-07:00".
      type: string
      format: date-time
    selfLink:
      description: The Blogger API URL to fetch this resource from.
      type: string
    context:
      description: The content of the comment, which can include HTML markup.
      type: string
    author:
      type: object
      description: The author of this comment.
      properties:
        id:
          description: The identifier of the comment creator.
          type: string
        displayName:
          description: The comment creator's display name.
          type: string
        url:
          description: The URL of the comment creator's profile page.
          type: string
        image:
          description: The container for the creator's avatar URL.
          type: object
          properties:
            url:
              description: The URL of the comment creator's avatar image.
              type: object
    inReplyTo:
      description: Data about the comment this is in reply to.
      type: object
      properties:
        id:
          description: The ID of the parent of this comment.
          type: string
    status:
      description: The status of the comment. The status is only visible to users who
                       have Administration rights on a blog.
      type: string
```

A.2.3 Post

```
...
/blogs/{blogId}/posts/{postId}:
    get:
      summary: Retrieves one post by post ID.
      parameters:
        - $ref: "#/components/parameters/blogIdParam"
        - $ref: "#/components/parameters/postIdParam"
        - $ref: "#/components/parameters/maxCommentsParam"
        - $ref: "#/components/parameters/viewParam"
      responses:
        "200":
          description: If successful, this method returns a Post resource in the response
                       body.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Post"

components:
  schemas:
    Post:
      type: object
      properties:
        kind:
          description: The kind of this resource. Always blogger#post.
          type: string
        id:
          description: The ID for this post.
          type: string
```

A.3 YouTube API

A.3.1 Comments

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Youtube API
  description: The YouTube Data API lets you incorporate functions normally executed on
               the YouTube website into your own website or application.
  termsOfService: https://developer.google.com/books/terms.html
```

```
  license:
    name: Apache 2.0
    url: https://creativecommons.org/licenses/by/4.0
externalDocs:
  description: Find more info here
  url: https://developers.google.com/youtube/v3/docs
servers:
  - url: https://www.googleapis.com/youtube/v3
paths:
  /comments:
    post:
      summary: Creates a reply to an existing comment.
      parameters:
        - $ref: "#/components/parameters/partParam"
      requestBody:
        description: Provide a comment resource in the request body
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Comment"
      responses:
        "200":
          description: If successful, this method returns a comment resource in the re
                       sponse body.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Comment"
    put:
      summary: Modifies a comment.
      parameters:
        - $ref: "#/components/parameters/partParam"
      requestBody:
        description: Provide a comment resource in the request body
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Comment"
      responses:
        "200":
          description: If successful, this method returns a comment resource in the re
                       sponse body.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Comment"
components:
  parameters:
    partParam:
      name: part
      description: The part parameter identifies the properties that the API response will
                   include.
      in: query
      required: true
      schema:
        type: string
        enum: [id, snippet]

  schemas:
    Comment:
      x-refersTo: https://schema.org/comment
      type: object
      properties:
        kind:
          description: Identifies the API resource's type. The value will be youtube#com
                       ment.
          type: string
        id:
```

```
      description: The ID that YouTube uses to uniquely identify the comment.
      type: string
    snippet:
      type: object
      description: The snippet object contains basic details about the comment.
      properties:
        authorDisplayName:
          description: The display name of the user who posted the comment.
          type: string
        authorProfileImageUrl:
          description: The URL for the avatar of the user who posted the comment.
          type: string
        authorChannelUrl:
          description: The URL of the comment author's YouTube channel, if available.
          type: string
        authorChannelId:
          description: This object encapsulates information about the comment author's
                       YouTube channel, if available.
          type: object
          properties:
            value:
              description: The ID of the comment author's YouTube channel, if availa
                           ble.
              type: string
        channelId:
          description: The ID of the YouTube channel associated with the comment.
          type: string
        videoId:
          description: The ID of the video that the comment refers to.
          type: string
        textDisplay:
          description: The comment's text. The text can be retrieved in either plain
                       text or HTML.
          type: string
        textOriginal:
          description: The original, raw text of the comment as it was initially
                       posted or last updated.
          type: string
        parentId:
          description: The unique ID of the parent comment.
          type: string
        canRate:
          description: This setting indicates whether the current viewer can rate the
                      comment.
          type: string
        viewerRating:
          description: The rating the viewer has given to this comment.
          type: string
          enum: [like, none]
        likeCount:
          description: The total number of likes (positive ratings) the comment has
                       received.
          type: integer
        moderationStatus:
          description: The comment's moderation status.
          type: string
          enum: [heldForView, likelySpam, published, rejected]
        publishedAt:
          description: The date and time when the comment was orignally published.
          type: string
          format: date-time
        updatedAt:
          description: The date and time when the comment was orignally published.
          type: string
          format: date-time
```

A.3.2 Subscription

```
...
/subscriptions:
    post:
      summary: Adds a subscription for the authenticated user's channel.
      parameters:
        - $ref: "#/components/parameters/partParam"
      requestBody:
        description: Provide a subscription resource in the request body.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Subscription"
      responses:
        "200":
          description: If successful, this method returns a subscription resource in the
                       response body.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Subscription"

components:
  schemas:
    Subscription:
      type: object
      properties:
        kind:
          description: Identifies the API resource's type. The value will be youtube#sub
                           scription.
          type: string
        etag:
          description: The Etag of this resource.
          type: string
        id:
          description: The ID that YouTube uses to uniquely identify the subscription.
          type: string
        snippet:
          description: The snippet object contains basic details about the subscrip
                       tion,including its title and the channel that the user subscribed
                       to.
          type: object
          properties:
            publishedAt:
              description: The date and time when the comment was orignally published.
              type: string
              format: date-time
```

A.4 Google Fit API

A.4.1 UserDataraSourcesResource – Extra

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Google Fit
  description: This API reference is organized by resource type. Each resource type has
               one or more data representations and one or more methods.
  termsOfService: https://developer.google.com/books/terms.html
  license:
    name: Apache 2.0
    url: https://creativecommons.org/licenses/by/4.0
externalDocs:
  description: Find more info here
  url: https://developers.google.com/fit/rest/v1/reference
servers:
  - url: https://www.googleapis.com/fitness/v1
paths:
  /users/{userId}/dataSources:
    post:
      summary: Creates a new data source that is unique across all data sources belonging
               to this user.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
      requestBody:
        description: In the request body, supply a Users.dataSources resource.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/UserDataSourcesResource"
      responses:
        "200":
          description: If successful, this method returns a Users.dataSources resource in
                       the response body.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/UserDataSourcesResource"
  /users/{userId}/dataSources/{dataSourceId}:
    delete:
      summary: Deletes the specified data source.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
        - $ref: "#/components/parameters/dataSourceIdParam"
      responses:
        "200":
          description: If successful, this method returns a Users.dataSources resource in
                       the response body.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/UserDataSourcesResource"
    get:
      summary: Returns the specified data source.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
        - $ref: "#/components/parameters/dataSourceIdParam"
      responses:
        "200":
          description: If successful, this method returns a Users.dataSources resource in
                       the response body.
```

```
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/UserDataSourcesResource"
    put:
      summary: Updates the specified data source.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
        - $ref: "#/components/parameters/dataSourceIdParam"
      requestBody:
        description: In the request body, supply a Users.dataSources resource with the
                     following properties.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/UserDataSourcesResourceExtra"
      responses:
        "200":
          description: If successful, this method returns a Users.dataSources resource in
                       the response body.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/UserDataSourcesResource"
components:
  parameters:
    userIdParam:
      name: userId
      description: The data stream ID of the data source to delete.
      in: path
      required: true
      schema:
        type: string

    dataSourceIdParam:
      name: dataSourceId
      description: Retrieve a data source for the person identified. Use me to indicate
                   the authenticated user. Only me is supported at this time.
      in: path
      required: true
      schema:
        type: string
  schemas:
    UserDataSourcesResource:
       x-kindOf: https://schema.org/UserInteraction
      type: object
      required:
        - application
        - dataType
        - device
        - type
      properties:
        application:
          description: Information about an application which feeds sensor data into the
                       platform.
          type: object
          required:
            - name
          properties:
            name:
              description: The name of this application. This is required for REST cli
                           ents, but we do not enforce uniqueness of this name. It is pro
                           vided as a matter of convenience for other developers who would
                           like to identify which REST created an Application or Data
                           Source.
              type: string
        dataType:
```

```
        description: The data type defines the schema for a stream of data being col
                     lected by, inserted into, or queried from the Fitness API.
        type: object
        required:
          - field
          - name
        properties:
          field:
            description: A field represents one dimension of a data type.
            type: array
            items:
              type: object
              required:
                - format
                - name
              properties:
                format:
                  description: The different supported formats for each field in a data
                               type.
                  type: string
                  enum:
                    [
                      blob,
                      floatList,
                      floatPoint,
                      integer,
                      integerList,
                      map,
                      string,
                    ]
                name:
                  description: Defines the name and format of data. Unlike data type
                               names, field names are not namespaced, and only need to
                               be unique within the data type.
                  type: string
                  ...
  UserDataSourcesResourceExtra:
    allOf:
      - $ref: "#/components/schemas/UserDataSourcesResource"
      - type: object
        properties:
          dataStreamId:
            description: A unique identifier for the data stream produced by this data
                         source
            type: string
```

A.5 Gmail API

A.5.1 Message, Draft

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Gmail API
  description: The Gmail API lets you view and manage Gmail mailbox data like threads,
               messages, and labels.
  termsOfService: https://developer.google.com/books/terms.html
  license:
    name: Apache 2.0
    url: https://creativecommons.org/licenses/by/4.0
```

```yaml
externalDocs:
  description: Find more info here
  url: https://developers.google.com/gmail/api/reference/rest
servers:
  - url: https://gmail.googleapis.com/gmail/v1/
paths:
/users/{userId}/drafts:
    post:
      summary: Creates a new draft with the DRAFT label.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
      requestBody:
        description: The request body contains an instance of Draft.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Draft"
      responses:
        "200":
          description: If successful, the response body contains an instance of Draft.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Draft"
    get:
      summary: Lists the drafts in the user's mailbox.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
        - name: maxResults
          description: Maximum number of drafts to return.
          in: query
          required: false
          schema:
            type: integer
            format: int32
        - name: pageToken
          description: Page token to retrieve a specific page of results in the list.
          in: query
          required: false
          schema:
            type: string
        - name: q
          description: Only return draft messages matching the specified query.
          in: query
          required: false
          schema:
            type: string
        - name: includeSpamTrash
          description: Include drafts from SPAM and TRASH in the results.
          in: query
          required: false
          schema:
            type: boolean
      responses:
        "200":
          description: If successful, the response body contains data with the following
                       structure.
          content:
            application/json:
              schema:
                ...
  /users/{userId}/drafts/{id}:
    delete:
      summary: Immediately and permanently deletes the specified draft. Does not simply
               trash it.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
        - $ref: "#/components/parameters/idParam"
```

```
    responses:
      "200":
        description: If successful, the response body will be empty.
  get:
    summary: Gets the specified draft.
    parameters:
      - $ref: "#/components/parameters/userIdParam"
      - $ref: "#/components/parameters/idParam"
      - name: format
        description: The format to return the draft in.
        in: query
        required: false
        schema:
          $ref: "#/components/schemas/Format"
    responses:
      "200":
        description: If successful, the response body contains an instance of Draft.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Draft"
  put:
    summary: Replaces a draft's content.
    parameters:
      - $ref: "#/components/parameters/userIdParam"
      - $ref: "#/components/parameters/idParam"
    requestBody:
      description: If successful, the response body contains an instance of Draft.
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/Draft"
    responses:
      "200":
        description: If successful, the response body contains an instance of Draft.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Draft"

/users/{userId}/drafts/send:
  post:
    summary: Sends the specified, existing draft to the recipients in the To, Cc, and
             Bcc headers.
    parameters:
      - $ref: "#/components/parameters/userIdParam"
    requestBody:
      description: The request body contains an instance of Draft.
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/Draft"
    responses:
      "200":
        description: If successful, the response body contains an instance of Draft.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Draft"
/users/{userId}/messages/{id}:
  delete:
    summary: Immediately and permanently deletes the specified message.
    parameters:
      - $ref: "#/components/parameters/userIdParam"
      - $ref: "#/components/parameters/idParam"
    responses:
      "200":
        description: If successful, the response body will be empty.
```

```
  get:
    summary: Gets the specified message.
    parameters:
      - $ref: "#/components/parameters/userIdParam"
      - $ref: "#/components/parameters/idParam"
    responses:
      "200":
        description: If successful, the response body contains an instance of Message.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Message"
/users/{userId}/messages:
  post:
    summary: Directly inserts a message into only this user's mailbox.
    parameters:
      - $ref: "#/components/parameters/userIdParam"
      - $ref: "#/components/parameters/internalDateSourceParam"
      - $ref: "#/components/parameters/deletedParam"
    requestBody:
      description: The request body contains an instance of Message.
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/Message"
    responses:
      "200":
        description: If successful, the response body contains an instance of Message.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Message"
/users/{userId}/messages/{id}/modify:
  post:
    summary: Modifies the labels on the specified message.
    parameters:
      - $ref: "#/components/parameters/userIdParam"
      - $ref: "#/components/parameters/idParam"
    requestBody:
      description: The request body contains data with the following structure.
      content:
        application/json:
          schema:
            type: object
            properties:
              addLabelIds:
                type: array
                items:
                  type: string
              removeLabelIds:
                type: array
                items:
                  type: string
    responses:
      "200":
        description: If successful, the response body contains an instance of Message.
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Message"
/users/{userId}/messages/send:
  post:
    summary: Sends the specified message to the recipients in the To, Cc, and Bcc head
             ers.
    parameters:
      - $ref: "#/components/parameters/userIdParam"
    requestBody:
      description: The request body contains an instance of Message.
```

```
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Message"
      responses:
        "200":
          description: If successful, the response body contains an instance of Message.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Message"
components:
    parameters:
        ...
    schemas:
        Draft:
          description: A draft email in the user's mailbox.
          allOf:
            - $ref: "#/components/schemas/Message"
            - type: object
              required:
                - id
              properties:
                id:
                  description: The immutable ID of the draft.
                  type: string
        Message:
              description: An email message.
              x-kindOf: https://schema.org/EmailMessage
              allOf:
                - $ref: "#/components/schemas/MessagePart"
                - type: object
                  properties:
                    id:
                      description: The immutable ID of the message.
                      type: string
                    threadId:
                      description: The ID of the thread the message belongs to
                      type: string
                    labelIds:
                      description: List of IDs of labels applied to this message.
                      type: array
                      items:
                        type: string
                        ...
```

A.6 Service Bundle

A.6.1 Subscription, Post

```
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Service Bundle
  description: Custom service for demonstration purposes
  termsOfService: https://developer.google.com/books/terms.html
servers:
  - url: http://www.intelligence.tuc/custom-service
paths:
  /{userId}/post/{postId}:
```

```
    delete:
      summary: Deletes a post.
      parameters:
        - $ref: "#/components/parameters/userIdParam"
        - $ref: "#/components/parameters/postIdParam"
      responses:
        "200":
          description: Returns a post instance.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Post"
  /{userId}/subscription_info:
    get:
      summary: Gets subscription info
      parameters:
        - $ref: "#/components/parameters/userIdParam"
      responses:
        "200":
          description: Returns a Subscription instance.
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Subscription"
components:
  parameters:
    postIdParam:
      name: postId
      description: The ID of the post to fetch comments from.
      in: path
      required: true
      schema:
        type: string

    idParam:
      name: id
      description: The ID of the draft to get.
      in: path
      required: true
      schema:
        type: string

    userIdParam:
      name: userId
      description: The id of the user
      in: path
      required: true
      schema:
        type: string

  schemas:
    Post:
      x-refersTo: https://www.example.com/service/googleBlogger_API#Post
      type: object
      properties:
        type:
          type: string

    Subscription:
      x-kindOf: https://www.example.com/service/youtube_API#Subscription
      type: object
      properties:
        theme:
          type: string
        pages:
          type: integer
```

References

[1] Karavasileiou A.: An ontology for describing OpenAPI version 3 services in the cloud, Diploma Thesis, School of Electrical and Computer Engineering, Technical University of Crete, November 2019.

[2] Nikolaos Mainas, Euripides G.M. Petrakis, "SOAS 3.0: Semantically Enriched OpenAPI 3.0 Descriptions and Ontology for REST Services", 14th IEEE International Conference on Semantic Computing (ICSC 2020), San Diego, California, February 3-5, 2020

[3] Semantics and Complexity of SPARQL Jorge P´erez , Marcelo Arenas, and Claudio Gutierrez

[4] N. Mainas, E.G.M. Petrakis and S. Sotiriadis, "Semantically enriched OpenAPI service descriptions in the cloud," in 8th IEEE International Conference on Software Engineering and Service Science, 2018, pp. 66-69. doi: 10.1109/ICSESS.2017.8342865

[5] "Automated Ontology Instantiation of OpenAPI REST Service Descriptions" Aikaterini Karavisileiou, Nikolaos Mainas, Fotios Bouraimis, and Euripides G.M. Petrakis Future of Information and Communications Conference

[6] Semantic Web, Wikipedia

[7] What Is the Semantic Web, ontotext

[8] Ontology (information science), Wikipedia

[9] What are ontologies?, ontotext.com

[10] OWL, w3.org

[11] Apache Jena, jena.apache.org

[12] Pellet: A Practical OWL-DL Reasoner

[13] OpenAPI Spec and Swagger, idratherbewriting.com

[14] OpenAPI Specification, swagger.io

[15] Web Service Description, IBM.com

[16] Why Web Services?, tutorialspoint.com

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

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

[19] Hydra: Hypermedia-Driven Web APIs, markuslanthaler.com/hydra/

[20] Roy Fielding. REST dissertation. 2000