TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

OpenAPI SPARQL: Querying OpenAPI Ontologies in OWL

Author:
Nikolaos Lagogiannis

Committee:
Euripides G.M. Petrakis
Vasileios Samoladas
Antonios Deligiannakis

A thesis submitted in fulfillment of the requirements for the degree of 5-year Diploma

October 17, 2022

TECHNICAL UNIVERSITY OF CRETE

Abstract

School of Electrical and Computer Engineering

OpenAPI SPARQL: Querying OpenAPI Ontologies in OWL

by Nikolaos Lagogiannis

In this work, we present OpenAPI SPARQL Language (OASL). OASL is an RDF query language specified OpenAPI ontologies. OpenAPI is a language agnostic format describing REST services in YAML or JSON form. In previous work, we showed how valid OpenAPI descriptions of RESTful services could be mapped to ontologies. However, queries on the OpenAPI ontology are very complex and require that the user be familiar with the peculiarities of the ontology. This is precisely the problem OASL deals with. To formulate an OASL query, a user needs only a basic understanding of SPARQL and no knowledge of Ontology OpenAPI definition. Also, the proposed language is easier to write from their equivalent SPARQL queries.OASL builds on top of SPARQL and simplifies query complexity, so even highly complex SPARQL queries can be expressed using only a few OASL statements.

Acknowledgements

First and foremost, I would like to thank my Supervisor Euripides Petrakis for his support and guidance during preparation and it's immediate response during every step of this work.

I would also like to thank my friends and family their support during all these years as a university student.

Contents

List of Figures

Chapter 1

Introduction

The available Web applications nowadays have risen remarkably, supplying consumers and developers with multiple options. Software companies associated with the generation of web services are also trying to create standard and easily readable documentation. Many of the formats introduced compete with each other to cover various applications. One of the dominant description formats, which is also an industry standard, it is OpenAPI Specification. OperAPI defines a standard, language-agnostic interface to RESTful APIs that humans and machines can easily understand.For a machine to understand the meaning of OpenAPI, service descriptions need to be formally defined, and their content is semantically enriched in a way that eliminates ambiguities.

1.1 Motivation

Semantic Web project is an extension of the World Wide Web through standards set by the World Wide Web Consortium (W3C)[9]. The goal of the Semantic Web is to make Internet data machine-readable.Taking into consideration the dynamics that generated about the expansion of semantic web , it is also emerges the opportunity to create machine-readable descriptions.In previous work by Fotios Bouraimis [2] , it is shown a way to convert OpenAPI descriptions into ontologies and SPARQL was the language that is used to make queries about them. However, even the smaller ones are complicated and require enough SPARQL expertise to be expressed from any user.

1.2 Proposed Solution

We introduce OpenAPI SPARQL, a query language for OpenAPI descriptions. This query language is similar to SPARQL and aims to define an easier-to-write syntax, avoiding the syntax complexity of SPARQL queries addressing the OpenAPI ontology. This would require that the user be familiar with OpenAPI syntax and and especially with the axioms (including rules) that have been defined in the OpenAPI ontology

[6]. SPARQL queries expressed using ontology axioms may become particularly complicated, involving many statements that an ordinary SPARQL user (with no understanding of the ontology) is almost impossible to express. OASL deals with precisely this problem. An OASL query involves much fewer statements using HTTP properties rather than OpenAPI or the OpenAPI ontology.

OASL syntax is proposed and support on features of SPARQL with the addition of operators capable of filtering effectively the OpenAPI ontology database (eg. comparison operators). The ontology database is implemented in a Virtuoso server. OASL is evaluated in a relatively large database with 100 ontologies by query complexity level (e.g. simple, complex, very complex queries) using a set of example queries.

1.3 Thesis Outline

The rest of this thesis is organized as follows:

- **Chapter 2 - Background Knowledge:** Background knowledge and introduction to concepts and technologies essential to our work.
- **Chapter 3 - OpenAPI Objects Analysis:** Contains all available Open API properties for this language according to our flattened objects.
- **Chapter 4 - OpenAPI SPARQL :** This chapter analyzes all details about OpenAPI SPARQL Language syntax.
- **Chapter 5 - System Implementation:** Technical details and the algorithmic structure about the implementation mechanism of this work.
- **Chapter 6 - Results and Measurements:** Examples about execution and discussing results.
- **Chapter 7 - Future Work:** Thesis conclusion and future work directions are provided .

Chapter 2

Background Knowledge

The number of web services have been rapidly increasing over the last years. This created the need for a standard description format of web services where can be understood by both humans and machines. Also another emerging need is to find a way to query data using a language easily understandable also by humans and machines. This chapter describes the basic knowledge to understand this work as also refers on previous work where is involved to this problem.

2.1 Semantic Web

The term "Semantic Web" refers to the W3C's vision of the Web of linked data. Semantic Web technologies enable people to create data stores on the Web, build vocabularies, and write rules for handling data [9]. Linked data are empowered by technologies such as RDF [8], SPARQL, and OWL[10]. The Semantic Web is not a separate Web but an extension of the current one. Information is given a well-defined meaning that enables computers and people to work in better cooperation. It extends the network of hyperlinked human-readable web pages by inserting machine-readable metadata about pages and how they are related to each other, enabling automated agents to access the Web more intelligently and perform tasks on behalf of users.

An ontology is a formal description of knowledge representation as a set of concepts within a domain and the relationships between them. To enable such a description, we need to formally specify components such as individuals (instances of objects), classes, attributes, and relations, as well as restrictions, rules, and axioms. As a result, ontologies do not only introduce a shareable and reusable knowledge representation but can also add new knowledge about the domain.

The ontology data model can be applied to a set of individual facts to create a knowledge graph – a collection of entities where nodes and edges express the types and the relationships between them. By describing the knowledge structure in a domain, ontology sets the stage

for the knowledge graph to capture the data. RDF [8] and OWL[10] are the technologies associated with this concept.

RDF stands for Resource Description Framework and is a standard for describing web resources and data interchange, developed and standardized with the World Wide Web Consortium (W3C). While there are many conventional tools for dealing with data and, more specifically, dealing with the relationships between data, RDF is the most straightforward, potent, and expressive standard designed now. [1] RDF connects data pieces via triples (three positional statements).
The structure of a triple contains tree statements named: Subject, Predicate, Object

- **Subject** represents the entity described with the following information.
- **Predicate** is the property that is requested to link the entity with the final information.
- **Object** describe the final information where it is aimed to be attached on the Subject.

Below is presented a simple statement in triple format.

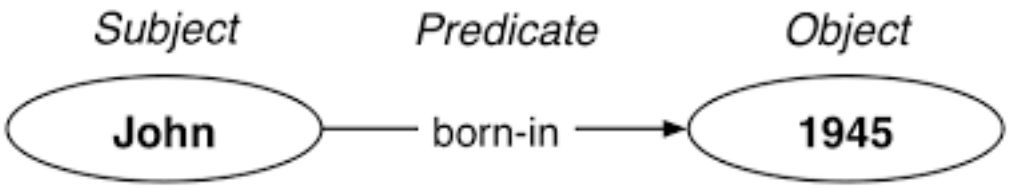

FIGURE 2.1: Basic Triple Structure Example

2.2 Open API

The OpenAPI Specification [1], previously known as the Swagger Specification, defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. [2] When properly defined, a consumer can understand and interact with the remote service with minimal implementation logic. Previously part of the Swagger framework, it became a separate project in 2016, overseen by the OpenAPI

[1] https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf/
[2] `https://en.wikipedia.org/wiki/OpenAPI_Specification`

Initiative, an open-source collaboration project of the Linux Foundation.OpenAPI Specification descriptions are written in JSON or YAML data serialization formats.

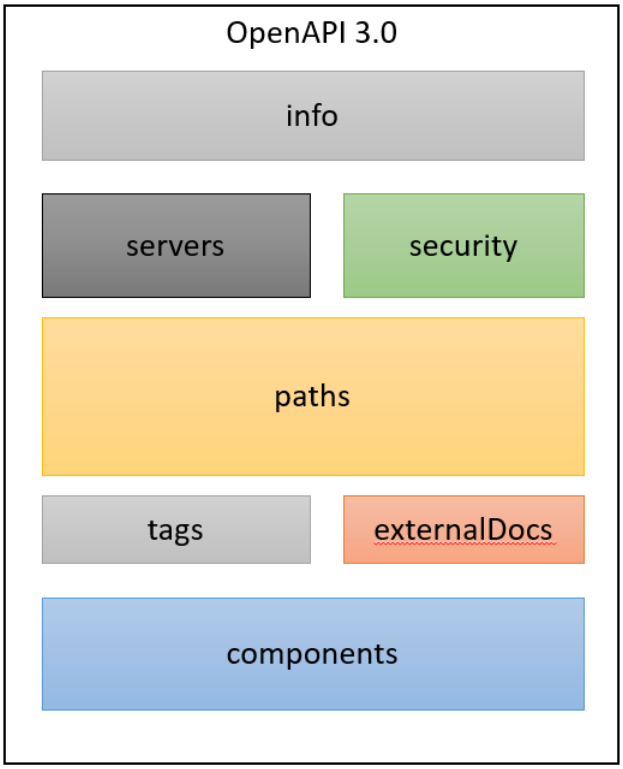

FIGURE 2.2: OpenAPI 3rd version Structural Objects Diagram

An OpenAPI document has defined many structural Objects, such as info Paths, ,Server ,Security, tag , external docs , and components.

- The **info** field contains high-level information about the API, such as version, title, and license.
- In **Paths** field are described the specific methods that can be reached, as well as their parameters, responses, human-readable descriptions of functionality, accepted content types, tags, and other expected behaviors.
- **ExternalDocs** as an object is used to enrich with additional information
- **Components** object, holds a set of reusable objects for different OAS elements to reference throughout the spec.These objects are Responses, Request Bodies, Headers , Links, Callbacks, Examples, Schemas and Security Schemas .

Below is presented a detailed OpenAPI Example.

```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in [CommonMark](http://commonmark.org/help/) .
  version: 0.1.9
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

FIGURE 2.3: OpenAPI example

This example describes a service with 2 servers available ,an endpoint '/users' that accepts GET requests. We also see that for response status code 200 ,which means successful request , the client receives a json file which contains data as String array.As we can also see the structure that follows is an info object who contains general information (title, description an version), then there is path object, which contains any endpoint this service describes and inside that endpoint are included every detail about this request.

2.3 Previous Work

In previous work has defined a way for expressing OpenAPI descriptions in Ontology. Also, another work implemented describes an algorithm that converts an openAPI document to a consistent ontology in OWL format.

Specifically, the first work claims that for a machine to understand the meaning of OpenAPI service descriptions, this content must be semantically defined in a way that eliminates ambiguities. This semantically enriched description is named SOAS 3.0 [6]. Taking advantage of SOAS 3.0, another previous work emerged [2], that creates a complete mechanism that transforms OpenAPI description to an OpenAPI Ontology.

The diagram below describes the structure of OpenAPI as an ontology.

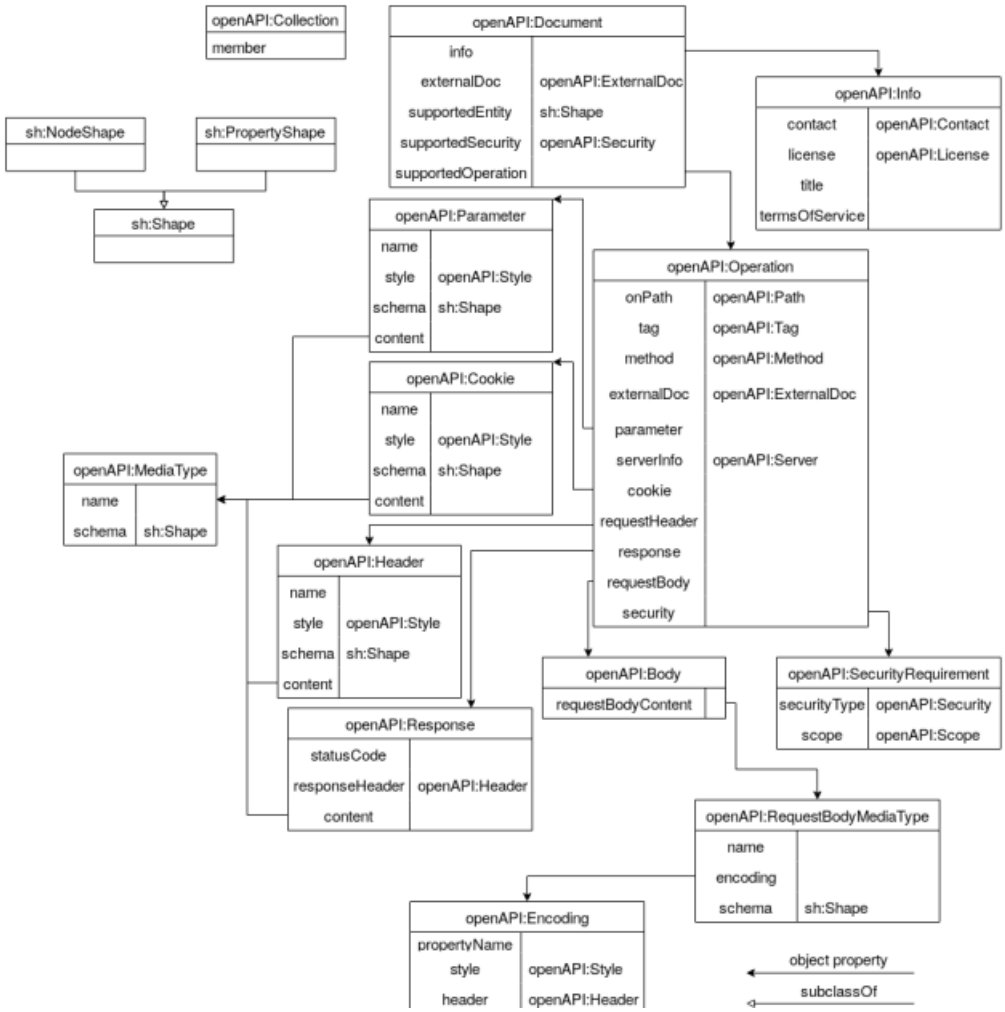

FIGURE 2.4: Semantic OpenAPI Structure

This work annotates entirely and with great semantic detail most OpenAPI Objects except Link and Callback objects. The following graph structure has the Document Class at the top level. This class contains the info that describes a service and binds together any description details about the Operation, Entity and Security objects where they are expressed. First, Entity objects refer to any schema or Property object that an OpenAPI service supports. This is supported through Shapes Constraint Language (SHACL) by creating equivalent sub-class PropertyShape class and NodeShape class. Operation class is the backbone of any Semantic
OpenAPI Document. Specifically, this class binds information about Response, RequestBody, Header, Parameter, Tag, Server, Cookie, onPath, and Security objects.

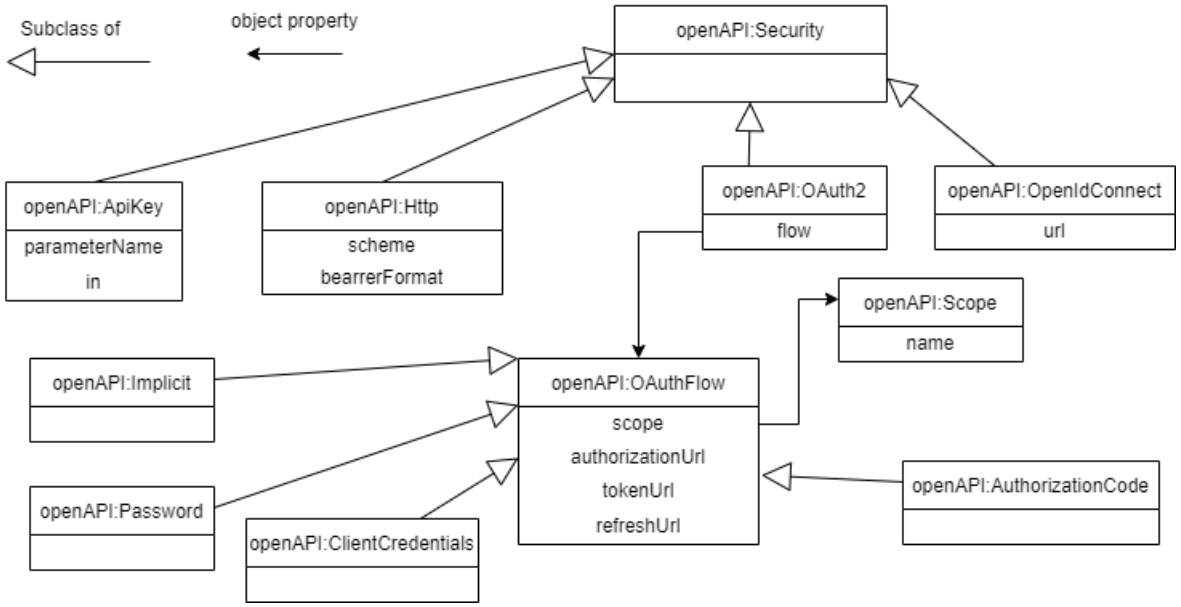

FIGURE 2.5: Security Object diagram

Security Class is a superclass which achieves to group the properties about every supported security type. Each one of these types individually , is defined as a subclass of Security with their specific properties.Same procedure is followed to describe OAuth flows.Flows has properties such as authorization url where they are member of OAuthflow Class which is linked with the subclass of security OAuth2 type.

Bottom level of diagram, contains classes that represent openAPI object such as MediaType , ExternalDocs , Contact, Liscense etc.These objects are more often used at many classes to express their multiple endpoints inside an openAPI description . Other classes however they defined to restrict their use at incorrect endpoints, so if it happens the ontology should be inconsistent.

2.4 SPARQL as a Query Language

SPARQL is an RDF query language that can retrieve and manipulate RDF data. It is an official recommendation standard by W3C (the only standard W3C query language) and SPARQL allows users to write queries against what can loosely be called "key-value" data or, more specifically, data that follow the RDF specification of the W3C. Thus, the entire database is a set of "subject-predicate-object" triples. This is analogous to some NoSQL databases' usage of the term "document-key-value". SPARQL syntax follows an SQL-like format. It uses the SELECT clause to retrieve data as SQL does, the FROM clause to define the dataset to be queried, the WHERE clause expresses the conditions about what to

query from a dataset and the ORDER BY clause is responsible for rearranging these results. Below is displayed a query example in SPARQL[3] :

SPARQL Example

```
#prefix declarations

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
...
#dataset definiton
FROM ...
#result clause
SELECT ?name ?email ...
#query patern
WHERE{
?person foaf:name ?name .
?person foaf:mbox  ?email .
...
}
```

FIGURE 2.6: Demonstrative Example in SPARQL

This example contains a query pattern that asks to project name and other variables for matching triples where the subject has no limitations (a variable in the issue can match anything), but the predicate follows two conditions. It is necessary to match as property with **foaf:name** URI and **foaf:mbox** property according to the same URI , defined in the prefix declaration clause.

The query above has an order of SPARQL query structure concerning the position of clauses.
First are declared any *prefixes* that will be used. It should be noted that prefixes are optional. Second follows the *SELECT clause*, responsible for variable declarations where will be projected. *WHERE clause* is the last and most crucial part because it includes the query patterns requested to match. Next, it follows the *FILTER clause*, which adds more restrictions about fetching the data, such as logical conditions or comparisons for expected values (string contains substrings or values are higher than specified prices, etc.). At last, there are 'rearrange' clauses such as *LIMIT OFFSET, ORDER BY*, and each one rearrange the query answer differently.

[3] https://en.wikipedia.org/wiki/SPARQL

Chapter 3

OpenAPI SPARQL Objects Analysis

RDF data are stored in graph databases. The challenge is to query a path of property triples, leading to a node expressing the requested object. However, SPARQL queries sometimes require multiple triples to describe some openAPI fields, leading to confusing and extended SPARQL queries. Moreover, many of these queries use the same paths to reach a group of semantic fields. As a result, someone who wants to ask a query in an RDF database with openAPI descriptions must understand openAPI specification and know precisely the semantic path of these ontologies. At last, he needs to write an extensive SPARQL query, with the risk of expressing it wrong.

OpenAPI SPARQL Language (OASL) applies flattering,a reduction of long path experssions into a simple statement with the property of interest. OASL requires that the user be familiar with SPARQL syntax (SELECT WHERE and triple understanding). This section explains the model and the fields that language supports. Every field of this model is expressed with no more than two triples (in most cases, they are defined with one). Notice that if more than one properties required, these can be properties of different OpenAPI Objects (eg. a schema property may describe objects of both requests or responses).

3.1 Service

Service object, includes the fields of openAPI, Info, Contact ,License and External Documentation objects.They contain general information about the described web service such as its title , description or its version.

The fields of Service object along with OpenAPI and semantic OpenAPI are shown below:

Service	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
title: string	title in info object	serviceTitle in info class
description: string	description in info object	description in info class
email: string	e-mail in contact object	e-mail in contact class
externalDocUrl: string	url in external documentation object	url in ExternalDocumentation class
externalDocDescription: string	description in external documentation object	description in ExternalDocumentation class
contactUrl: string	url in contact object	url in contact class
contactName: string	name in contact object	name in contact class
version: string	version in Document object	version in Document class
liscenseName: string	name in license object	name in license class
liscenseDescription: string	description in license object	description in license class

FIGURE 3.1: Service Table

3.2 Request

Response Object includes fields of Paths, Operation, and Request Body Objects. In semantic openAPI, all these objects are part of Operation Class which they have access through supported operation property. These objects describe the requests and their details supported by a web service method(ex GET) , operationName etc.

The fields of Request object along with OpenAPI and semantic OpenAPI are shown below:

Request	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
method : ['get', 'put', 'post', 'delete']	Direct property by name in Paths Object	method property in Operation class object with Method class accepted fields
path: string	property name inside paths object	onPath property in Operation class object
operationName: string	operationId in Operation Object	name property in Operation class object
schema: string	Schema in RequestBody object	schema in requestBody class object inside Operation
contentType: string	Property name inside content field Request Body Object	Property name in Mediatype class object
extDocUrl: string	url in contact object	url in ExternalDocumentation class
extDocDescription: string	Description in External Documentation Object	description in ExternalDocumentation class
summary: string	Summary in Operation object	summary in Operation class object
bodyDescription: string	Description in Request Body object	description in RequestBody class object
tag: string	Tags in Operation Object	name property of Tag class located in operation object

FIGURE 3.2: Request Table

Method field in a Semantic OpenAPI description has been declared as a property of Operation object domain and in OpenAPI is referred directly with their property names in Path object (GET , PUT, POST, DELETE). In OASL, any question about the method field has accepted answers, only the previous four keywords. Also, the operationName field represents operationId in openAPI, which is defined with name property at the Operation object domain in its semantic version. The schema keyword is used to access the schema field in RequestBody Object.

3.3 Security

Security object include fields from Security Requirement, Security Scheme, OAuth Flows and OAuth Flow. These objects describe all security mechanisms defined by web services. In semantic openAPI these objects are described with Security objects and their subclasses (APIKey , Http ,OAuth2 , openIdConnect) as also OAuthFlow class with its subclasses (ClientCredentials , AuthorizationCode ,Password ,Implicit) as shown in the figure: 2.5 .

The fields of Security object along with OpenAPI and semantic OpenAPI are displayed below::

Security	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
apiKeyIn: string	*In* located in Security Scheme object	In located in ApiKey class (subclass of Security)
apiKeyName: string	name in Security Scheme object	name located in ApiKey class (subclass of Security)
httpBearrerFormat:string	bearrerFormat in Securty Scheme Object	format in Http class (subclass of Security)
httpBearrerScheme: string	scheme in Security Scheme object	scheme in Http class (subclass of Security)
openIdConnectUrl: string	name in Security Scheme object	Url in OpenIdConnect class (subclass of Security)
OAuth2AuthUrl: string	authorizationUrl in OAuthflow object	authorizationUrl in OAuthflow class
OAuth2RefreshUrl: string	refreshUrl in OAuthflow object	refreshUrl in OAuthflow class
OAuth2TokenUrl: string	tokenUrl in OAuthflow object	tokenUrl in OAuthflow class
OAuth2flowType: ['Implicit', 'Client Credentials' , 'Password' , 'AuthCode']	property names in OAuthflows Object	Subclasses of AuthFlow class (non existent direct field)
securityType: ['Oauth2' , 'Api Key', 'OpenIdConnect' , 'Http']	Type in Security Scheme object	Subclasses of Security class (non existent direct field)

FIGURE 3.3: Security Table

SecurityType field represents the field type in the Security Object at OpenAPI. In semantic OpenAPI it is defined with the creation of the appropriate subclasses (OAuth2, ApiKey, etc.) and there is not specific property to describe it. In (OASL) it is accepted to ask using only the subclasses names as keywords (Oauth2 , Api Key, OpenIdConnect and Http).

Also, it defines OAuth2flowType field using subclasses names as keywords (Implicit, Client Credentials , Password , AuthCode). The other fields are flattened properties from every security type and OAuthflow type mentioned above.

3.4 SecurityScope

A request mechanism can also require specific scopes. The fields of Security Scope object relates the name and descriptions defined in the OAuthFlow object's scopes.
The fields of Security Scope object, along with OpenAPI and semantic OpenAPI are displayed below:

SecurityScope	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
description: string	property value inside scopes in OAuthFlow object	name property inside Scope Class
name: string	property value inside scopes in OAuthFlow object	description property inside Scope Class

FIGURE 3.4: Security Scope Table

3.5 Tag

Tag corresponds to fields in Tag and External Documentation objects and they are used to group operations as objects in a description.

The fields of Tag object along with OpenAPI and semantic OpenAPI are displayed below:

Tag	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
name: string	name in Tag object	Name in Tag class
description: string	description in Tag object	description in Tag class
externalDocUrl: string	Url in Tag object	Url in Tag class
*externalDoc*Description*: string*	External Documentation description in Tag Object	description in External Doc Class inside Tag

FIGURE 3.5: Tag Table Object

3.6 Response

The field of Response table contains properties from Response , Responses and Media Type objects.These objects describe, about a request, details for its defined responses.
The fields of Response object along with OpenAPI and semantic OpenAPI are shown below:

Response	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
description: string	description in Response object	description in Response class
schema	schema in Mediatype object at content field of Response object	schema in Mediatype Class at content field of Response inside Operation object
statusCode: integer	property name inside Responses object	statusCode in Response class object
statusCodeRange: ['1xx' , '2xx', '3xx', '4xx' , '5xx']	property names inside Responses object	Subclasses of Response class (non existent direct field)

FIGURE 3.6: Response object table

In OpenAPI description statusCode property is described with property names as a string either is a number about the status code or expresses a response range using "-xx" string symbol (1xx which means 100-199, 2xx ,3xx ,4xx and 5xx). In semantic openAPI status code range is expressed by different subclasses of Response class object. In (OASL) status code defined with two separate fields. If a user want to ask for status code range uses statusCodeRange property giving the above "-xx" keywords as it is shown in figure 3.6 . Asking about a certain status code number uses the statusCode field in integer format in a question.

3.7 Server

Server objects are available to describe about each Request its own servers. Server table contain fields from Server Object. The fields of Server object along with OpenAPI and semantic OpenAPI are shown below:

Server	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
description: string	description in Server object	description in Server class
url: string	url in Server object	url in Server class

FIGURE 3.7: Caption

3.8 Header

Header Table contains all Header Object properties. This object has the same structure with Parameter object and describes any header that are returned with response or a request. In semantic OpenAPI it is placed either as a ResponseHeader or as a RequestHeader inside Operation Object.

The fields of Header object along with OpenAPI and semantic OpenAPI are shown below:

ResponseHeader or RequestHeader	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
name: string	name in Header object	name in Header class (and its sublasses)
required: boolean	required in Header object	required in Header class (and its sublasses)
schema	schema in Parameter object	schema in Parameter class (and its sublasses)
description: string	description in Header object	description in Header class (and its sublasses)
style: [‘simple’ , ‘form’ , ‘label’ , ‘matrix’ , ‘pipeDelimited’ , ‘spaceDelimited’]	style in Headerobject	style in Header class (and its sublasses)
allowEmptyValue: boolean	allowEmptyValue in Header object	allowEmptyValue in Header class (and its sublasses)
deprecated: boolean	deprecated in in Header object	decprecated in Header class (and its sublasses)

FIGURE 3.8: Header Table

Because of the existence of the Header object at two critical positions inside an OpenAPI document, this table expresses two ways of querying. The RequestHeader keyword is responsible for searching directly from the requestHeader property domain of the Operation object. The ResponseHeader keyword is responsible for searching directly from responseHeader property, which is in the domain of the Response object located inside the Operation.

3.9 Parameter

Parameter table contains fields from Parameter Object in OpenAPI description.As an object describes parameters where a request can acquire with it.

The fields of Parameter object along with OpenAPI and semantic OpenAPI are shown below:

Parameter	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
name: string	name in Parameter object	name in Parameter class inside operation
required: boolean	required in Parameter object	required in Parameter class inside operation
Schema:string	schema in Parameter object	Schema in Parameter class inside operation
description: string	description in Parameter object	description in Parameter class inside operation
style:[‘simple’ , ‘form’ , ‘label’ , ‘matrix’ , ‘pipeDelimited’ , ‘spaceDelimited’]	style in Parameter object	style in Parameter class inside operation
allowEmptyValue: boolean	allowEmptyValue in Parameter object	allowEmptyValue in Parameter class inside operation
allowReserved: boolean	allowReserved in Parameter object	allowReserved in Parameter class inside operation
deprecated: boolean	deprecated in in Parameter object	deprecated in Parameter class inside operation
location:[‘path’ , ‘header’, ‘query’ ,‘cookie]	In field located in Parameter object	Subclasses of Parameter class (non existent direct field)

FIGURE 3.9: Parameter Table

The field location is defined in (OASL) to enable questions about ’in’ field in an OpenAPI document. It should be noted that in Semantic OpenAPI description there is not a property to describe it, however it is represented by equivalent subclasses of Parameter class, such as Path-Parameter, Cookie etc. The accepted keywords in (OASL) are "path", "header" , "query" and "cookie". Also style field has specific accepted keywords that they simplify the original define objects in a Semantic OpenAPI description.

3.10 Schema

Schema table contains fields of Schema object. This object of openAPI has defined to conclude all fields that defined in JSON Schema. However in (OASL) are supported only the most important ones.Their use is to describe any data model where a request , response, parameter or header can support.

In Semantic OpenAPI Schema and its fields, it is described through SHACL objects (i.e., NodeShape or PropertyShape objects), which, in turn, are defined as subclasses of a general ShapeClass.

The fields of Schema object along with OpenAPI and semantic OpenAPI are shown below:

Schema	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
dataType:['integer' , 'number' , 'string' , 'boolean' ,'array']	type in JSON schema	datatype property in PropertyShape inside property field of NodeShape
type	type in JSON schema	targetClass in NodeShape
property	properties in Schema object	property Field in NodeShape
title: string	title in JSON schema	title property in NodeShape
description: string	description in JSON schema	description in NodeShape
minProperties: integer	minProperties in JSON schema	minProperties in NodeShape
maxProperties: integer	maxProperties in JSON schema	maxProperties in NodeShape
minLength: integer	minLength in JSON schema	minLength in NodeShape
maxLength: integer	maxLength in JSON schema	maxLength in NodeShape
minCount: integer	minCount in JSON schema	minCount in NodeShape
maxCount: integer	maxCount in JSON schema	maxCount in NodeShape
minimum: number	minimum in JSON schema	minimum in NodeShape
maximum: number	maximum in JSON schema	maximum in NodeShape

FIGURE 3.10: Schema Table

The 'Schema' subject keyword is used, then the query asks for the supported entity field. In other objects, as aforementioned, there are also schema properties. In SPARQL, when a user wants to ask a specific question about a schema property, he writes several triples to reach the appropriate property-value endpoint. In (OASL), because of the importance of most of the schema questions, it is selected the SPARQL-like approach of using variables to avoid reducing the available query variety. Also property field, its purpose is the direct queries about name, type and datatype for the Property object.

3.10.1 Schema field in other objects

The schema objects can be used in multiple positions inside an OpenAPI description. For example, we can declare a schema in the Response object but also in Request, Header e.t.c. In order to allow the user to query schemas in every position from the document, access to that position is achieved using the schema fields from these objects. In detail, the user enters a variable as an object in the specific triple and next can access all schema properties using the variable as the triple's subject.

OASL Example	***SPARQL Example***
1) Response schema ?var1 *2)* ?var1 title "Dog"	*1)* ?service openapi:supportedOperation ?operation . *2)* ?operation openapi:response ?response . 3) ?response openapi:content ?r_con . 4) ?r_con openapi:schema ?var1 . 5) ?var1 rdf:type sh:NodeShape 6) ?var1 rdfs:label "DogNodeShape"
1) Schema title "Dog"	*1)* ?service openapi:supportedEntity ?node_shape . 2) ?node_shape rdf:type sh:NodeShape . 3) ?node_shape rdfs:label "DogNodeShape"

FIGURE 3.11: Schema property example

This figure has two different (OASL) query parts. The first one describes a triple that asks about the schema property in Response Object with the title value of "Dog". As it is pointed with red color, using a variable (?var1) it connects the triples (1st as an object and 2nd as a subject), and similarly queries schema fields with SPARQL. The next column has the equivalent SPARQL query and lines 4-6, which is noted with red color, an identical definition.

The second example shows a triple using the defined Schema keyword as the subject and the equivalent query is about the 'supported entity' endpoint in a semantic OpenAPI description.

It is also accepted as value , except from variables , to add string or URI making this an abbreviation triple.More details about the implementation are in Chapter 5

3.11 Property

In OpenAPI properties field contains also object that can be described as Schema Object. In semantic openAPI this is also viable with the definition of PropertyShape class.

The fields of Property object along with OpenAPI and semantic OpenAPI are shown below:

Property	Equivalent OpenAPI field	Equivalent Semantic OpenAPI Field
dataType: ['integer' , 'number' , 'string' , 'boolean' ,'array']	type in JSON schema	Datatype field inside PropertyShape class
name: string	property names inside properties object field	Name field property inside PropertyShape class
description: string	description in JSON schema	description in JSON schema
type	type in JSON Schema	Path field property inside PropertyShape class
minLength: integer	minLength in JSON schema	minLength in PropertyShape class
maxLength: integer	maxLength in JSON schema	minLength in PropertyShape class
minCount: integer	minCount in JSON schema	minCount in PropertyShape class
maxCount: integer	maxCount in JSON schema	maxCount in PropertyShape class
minimum: number	minimum in JSON schema	minimum in PropertyShape class
maximum: number	maximum in JSON schema	maximum in PropertyShape class

FIGURE 3.12: Property Table

3.11.1 Property field

The property field works similarly to the schema field in other objects. Using a variable is the standard way to query specific details about a property object. It is also accepted as a value, except from variables, to add string or URI, making this an abbreviation triple. According to the value is translated into a different triple group. More details about the implementation are in Chapter 5

Chapter 4

OpenAPI SPARQL Language

OpenAPI SPARQL Language is a query language for searching Semantic OpenAPI descriptions. The syntax of this language is similar to the SPARQL language, so it is easier for a user with SPARQL knowledge to understand and adapt to (OASL) syntax. Every OASL query consists of the following structural components:

- **PREFIX clause** (optional): This clause is responsible for defining keywords that correlate to specific URIs.
- **SELECT clause**: This clause contains which variables value are to be projected as results in the user
- **WHERE clause**: This clause contains all conditions and rules in triple form. These conditions filter these OpenAPI triples and match them with the user's defined variables.
- **Rearrange clause** (optional) :This clause includes 3 different clauses , with purpose to rearrange the projected data. These clauses are **LIMIT , ORDER BY** and **OFFSET**.

4.1 Language Syntax

The syntax of OpenAPI SPARQL (OASL) is presented below in Backus Naur form (BNF).After BNF is presented the accepted language triples for OpenAPI SPARQL Language :

```
<query>     ::=<prefixes> <select query> <where query> <rearrange clause>
              | <select query> <where query> <rearrange clause>
              | <prefixes> <select query> <where query>
              | <select query> <where query>


<prefixes>              ::= PREFIX  prefix_name : URI
                          | PREFIX  prefix_name : URI <prefixes>

<select query>          ::= SELECT       <select clause>
                          | SELECT DISTINCT   <select clause>

<where query>           ::= WHERE{  <where Body> }

<where Body>            ::= <triples>
                          | {<triples>} OR{<triples >}
                          |  <triples> <where Body>
                          | {<triples>} OR{<triples>} <where Body>


<rearrange clause>  ::= (<order clause> |  <limit clause> )
                      | (  <order clause>  (<limit clause>| <offset clause>))
                      | (<order clause> <limit clause>  <offset clause>)
                      | (<order clause> <offset clause> <limit clause>)
```

<select clause> ::= <variables >

<variable> ::= ?<identifier>

<variables> ::= <variable>
::= <variable> ,<variables>

<triples> := <triple>
:= <triple> <triples>
:= NOT {<triple> }
:= OPTIONAL {<triple>}

<Subject> := <variable>
| <Classes defined in column 1>

<Predicate> ::= <Column 2 : *All properties based on Subject*>

<Object> ::= <variable>
|<number>
|<string>
| <boolean>
|<URI>
| <identifier> : <identifier>
| <between clause>
| <compare clause >

<number> ::=<integer> . <integer>
| <integer>

<string> | " <*literal*> "

<*integer*> ::= <*digit*>
| <*digit*> <*integer*>

<*boolean*> ::= (*true* | *false*)
<*identifier*> ::= (<*character*> | <*digit*>)
| <*identifier*> (<*character*> | <*digit*> | "_")

<*URI*> ::= "<" <*literal*> ">"
<*literal*> ::= <*character*> | <*literal*> <*character*>
<*character*> ::= <*letter*>|<*digit*>|<*symbol*>

```
<letter>          ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P |
                      Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k |
                      l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<digit>           ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<symbol>          ::= | ! | # | $ | % | & | ( | ) | * | + | , | - | . | / | : | ; | > | = | <
                      | ? | @ | [ | ] | ^ | _ | ` | { | } | ~ | ' | \

<between clause>        ::= BETWEEN(<number>, <number>)

<compare clause>        ::=  ( > | >= | < | <= ) <number>

<order clause>    ::= ORDER BY desc <variable>
                    | ORDER BY asc  <variable>

<limit clause>    ::= LIMIT integer
<offset clause>   ::= OFFSET integer
```

4.1.1 Triple Syntax

The table below contains, in the first column, the OpenAPI objects that OASL recognizes. The user can type the presented keywords as the **subject** in his triple. The second column contains all the recognized properties in an ontology that every OpenAPI object accepts. The keywords of the second column are the **predicate** in his triple. At last, column 3 expresses the **object** in his triple. Anything inside quotes describes the only keywords that accept as an answer. String, boolean, and integer tell the acknowledged format in this specific field. Fields such as type and schema object cells are empty because they accept either more than one type or, according to their answer, generate a different SPARQL code, as the previous section mentions. Of course, identical SPARQL variables are accepted in the object's position to protect our data (using the same variable with the SELECT clause).

TRIPLE COLUMNS		
Column1 SUBJECT	**Column 2 PREDICATE**	**Column 3 OBJECT**
Service	title	string
	description	string
	email	string
	externalDocUrl	string
	externalDocDescription	string
	contactUrl	string
	contactName	string
	version	string
	liscenseName	string
	liscenseDescription	string
Request		
	method	‘get’ , ‘put’ , ‘post’ , ‘delete’
	path	string
	operationName	string
	schema	string
	contentType	string
	header	string
	extDocUrl	string
	extDocDescription	string
	summary	string
	bodyDescription	string
	tag	string
Server	description	string
	url	string
SecurityScope	description	string
	name	string
Parameter	name	string
	required	boolean
	schema	

	description	string
	style	‘simple’ , ‘form’ , ‘label’ , ‘matrix’ , ‘pipeDelimited’ , ‘spaceDelimited’
	allowEmptyValue	boolean
	allowReserved	boolean
	deprecated	boolean
	location	‘path’ , ‘header’, ‘query’ ,‘cookie’
Tag	name	string
	description	string
	extDocUrl	string
Security	apiKeyIn	string
	apiKeyName	string
	httpBearrerFormat	string
	httpBearrerScheme	string
	OpenIdConnectUrl	string
	OAuth2AuthUrl	string
	OAuth2RefreshUrl	string
	OAuth2TokenUrl	string
	OAuth2flowType	‘Implicit’, ‘Client Credentials’ , ‘Password’ , ‘AuthCode’
	securityType	‘Oauth2’ , ‘Api Key’, ‘OpenIdConnect’ , ‘Http’
Response	schema	string
	header	string
	statusCode	integer
	statusCodeRange	‘1xx’ , ‘2xx’, ‘3xx’, ‘4xx’ , ‘5xx’
	description	string
ResponseHeader RequestHeader Header		
	name	string
	required	boolean
	schema	
	description	string
	style	‘simple’ , ‘form’ , ‘label’ , ‘matrix’ , ‘pipeDelimited’ , ‘spaceDelimited’

	allowEmptyValue	boolean
Schema <variable>	dataType	‘integer’ , ‘number’ , ‘string’ , ‘boolean’ , ‘array’
	type	
	property	
	style	‘simple’ , ‘form’ , ‘label’ , ‘matrix’ , ‘pipeDelimited’ , ‘spaceDelimited’
	title	string
	minProperties	number
	maxProperties	number
	minLength	number
	maxLength	number
	minCount	number
	maxCount	number
	minimum	number
	maximum	number
	description	string
Property <variable>	dataType	‘integer’ , ‘number’ , ‘string’ , ‘boolean’ , ‘array’
	name	string
	type	
	description	string
	minLength	number
	maxLength	number
	minimum	number
	maximum	number
	minCount	number
	maxCount	number

4.1.2 Prefix and SELECT clause

The PREFIX clause is an optional part of the query where a user can define prefixes. Every prefix is matched with a URI separating the prefix part from the URI part with the ":" symbol, similarly to SPARQL. Prefixes are used as URI shortcuts inside a query, allowing the user to express smaller and easier questions.

The SELECT clause is the position inside a query where the user defines any variables where is content wants to expose as the output of his query. Any results are usually presented in table format, and the variable name is also the column's name. A variable starts with the "?" symbol followed by an identifier. It is also available for the user with the "DISTINCT" keyword. Using this keyword, every duplicated tuple from the expected results are removed. It should be noted that these syntax rules are identical to the SPARQL syntax structure.

Below is presented an example of the implementation of these rules:

Syntax Example	Syntax description
PREFIX example1: <http://example.com/resources/> PREFIX example2: <http://example2.com/tagCompanies/>	Prefix declaration syntax (example1, example2)
SELECT ?variable1, ?variable2	Select syntax about 2 variables (variable1, variable2)
SELECT DISTINCT ?variable1, ?variable2	Select syntax following DISTINCT keyword

FIGURE 4.1: Prefix and Select clause examples

In this example is shown two declaration of prefixes named *example1* and *example2*. A colon symbol follows them to separate from the URI's and, at last, is declared the URI's where they are replaced. The SELECT clause contains two variables separated by a comma and presents the use of
DISTINCT keyword.

4.1.3 WHERE Clause

This clause is responsible for choosing which triple conditions must be true and defining the fields the user wants to express. The accepted triples and most commonly accepted combinations derived from their objects have been described in the previous section. Except for intersection between triples,it is also supported OR clauses, equivalent to

UNION clauses in SPARQL, OPTIONAL clauses, and NOT clauses as like in SPARQL.

- **OR clause** syntax is identified correctly by the use of a number of triples inside to brackets following by *OR* keyword and at last followed by another count of triples inside brackets.
- **OPTIONAL** syntax contains inside brackets only one triple. If the user wants to add more optional triples , follows the same procedure for each triple
- **NOT** syntax follow the same structure with OPTIONAL clause. It is used inside brackets only one triple.

Below is presented an example with the implementation of these rules:

Syntax Example	Syntax description

WHERE {
Service title ?variable1
Tag name example2:GoogleCompany
Request operationName ?variable2
Response schema example1:Workers
}
	WHERE syntax about intersection triples. Every line that is written as a normal triple (without OR, NOT etc.) it represents an intersection clause.

WHERE {
Service title ?variable1
Tag name example2:GoogleCompany
Request operationName ?variable2
{Response schema example1:Managers}
OR {
Response schema example1:Employers}
}
	This where syntax expresses using OR keyword, an equivalent to SPRQL UNION. This clause projects some field in variables who validates one of these two conditions (schema refers to Managers or Employers)

WHERE {
Service title ?variable1
NOT{Tag name example2:GoogleCompany}
}
	This where syntax expresses the use of NOT keyword. Similarly, to SPARQL if this condition enclosed by brackets is true. Then then any marched value is false.

WHERE {
Service title ?variable1
Tag name example2:GoogleCompany
OPTIONAL {Request operationName ?variable2}
Response schema example1:Workers
}
	OPTIONAL use is designed identical to SPARQL. Inside brackets is valid syntax only one triple.

FIGURE 4.2: Where syntax example

The first example describes a WHERE clause asking the title from services and their operations' names, with operations tagged about GoogleCompany according to a declared uri. Also, its response is a type of schema that has been declared according to another uri endpoint (example1:Workers).

The second example describes a query which contains OR clause inside.OR clause have similar use and syntax to SPARQL UNIONS .The keyword change it is selected to be similar to SQL language, because most of users are not familiar with SPARQL as also is more memorable for a simple user. This example is asking for web services, which they have tag name as example2:GoogleCompany defines, and Response schema can be *(example1:Managers and example1:Employers)*. We ask to retrieve as result its service title and its operation name.

NOT example fetches every service title that matches, but only if operations tag is not as example2:GoogleCompany declares.

The last example expresses that if is declare an operation name to that endpoint , in expected answer will be expressed in variable2.Else this tuple will still fetch the rest requested info, even though it will be void.

Overall WHERE clause in OASL includes the equivalent syntax rules from SPARQL WHERE and FILTER clause. Because of the restricted range of queries where a user can search about the fields of an openAPI ontology description, the only useful filter clauses that could be used are about comparisons between numbers.As a result it is selected to integrate these functionalities inside where clause to achieve a more convenient language for user.

4.1.4 Rearrange Clause

This clause describes three different clauses whom its purpose is to rearrange any data where the query fetches. Their syntax behaviour is identical to the SPARQL syntax. These clauses are : **order by clause LIMIT clause** and **offset clause**.These rearrange clauses are optional and they have specific order between them. For example ORDER BY clause syntax comes first and later on follows LIMIT and OFFSET.

LIMIT command is used to contain to a specific number the expected results. Syntax only requires *LIMIT* keyword followed by an integer. If the answers exceed this number, then they will not be included to the answer table. below it is presented an example and its results about this section of query:

Example without rearranging	LIMIT Example

SELECT DISTINCT ?serviceName ?opName
WHERE {
Service title ?serviceName
Request method GET
Request operationName ?opName
}
	SELECT DISTINCT ?serviceName
	?opName
	WHERE {
	Service title ?serviceName
	Request method GET
	Request operationName ?opName
	}
	LIMIT 15

FIGURE 4.3: Query syntax comparison between no rearrangement and LIMIT use.

The example above retrieves the titles from the matched services and the operation names, which their request methods GET requests. We expect, as a result, operations with GET request methods and furthermore the name of the service where it is located. All 'rearrange examples' that are displayed in this section are similar variations containing the same triples.

#	serviceName	opName
1	Google Fit	list-searchable-fields
2	Gmail API	list-searchable-fields
3	Swagger Petstore - OpenAPI 3.0	list-searchable-fields
4	USPTO Data Set API	list-searchable-fields
5	Google Blogger	list-searchable-fields
6	Youtube API	list-searchable-fields
7	Greenvivir-Backend	list-searchable-fields
8	Google Fit	retrieveHumidProperty
9	Gmail API	retrieveHumidProperty
10	Swagger Petstore - OpenAPI 3.0	retrieveHumidProperty
11	USPTO Data Set API	retrieveHumidProperty
12	Google Blogger	retrieveHumidProperty
13	Youtube API	retrieveHumidProperty
14	Greenvivir-Backend	retrieveHumidProperty
15	Google Fit	retrieveListOfSubscriptions
16	Gmail API	retrieveListOfSubscriptions
17	Swagger Petstore - OpenAPI 3.0	retrieveListOfSubscriptions
18	USPTO Data Set API	retrieveListOfSubscriptions
19	Google Blogger	retrieveListOfSubscriptions
20	Youtube API	retrieveListOfSubscriptions
21	Greenvivir-Backend	retrieveListOfSubscriptions
22	Google Fit	getPersonsByCURP
23	Gmail API	getPersonsByCURP
24	Swagger Petstore - OpenAPI 3.0	getPersonsByCURP
25	USPTO Data Set API	getPersonsByCURP
26	Google Blogger	getPersonsByCURP
27	Youtube API	getPersonsByCURP

FIGURE 4.4: Results without Rearrangement

#	serviceName	opName
1	Google Fit	list-searchable-fields
2	Gmail API	list-searchable-fields
3	Swagger Petstore - OpenAPI 3.0	list-searchable-fields
4	USPTO Data Set API	list-searchable-fields
5	Google Blogger	list-searchable-fields
6	Youtube API	list-searchable-fields
7	Greenvivir-Backend	list-searchable-fields
8	Google Fit	retrieveHumidProperty
9	Gmail API	retrieveHumidProperty
10	Swagger Petstore - OpenAPI 3.0	retrieveHumidProperty
11	USPTO Data Set API	retrieveHumidProperty
12	Google Blogger	retrieveHumidProperty
13	Youtube API	retrieveHumidProperty
14	Greenvivir-Backend	retrieveHumidProperty
15	Google Fit	retrieveListOfSubscriptions

FIGURE 4.5: Results Using LIMIT

Comparing the two result tables is is obvious that after the use of LIMIT the expected tuples reduced to 15 (as the number after LIMIT)

OFFSET command is used also for containing the number of expected results and its syntax is the same with LIMIT. However, the declared number expresses how much triples are going to be skipped.

below it is presented an example and its results about this section of query:

LIMIT Example	LIMIT and OFFSET Example

SELECT DISTINCT ?serviceName ?opName
WHERE {
Service title ?serviceName
Request method GET
Request operationName ?opName
}
LIMIT 15
	SELECT DISTINCT ?serviceName
	?opName
	WHERE {
	Service title ?serviceName
	Request method GET
	Request operationName ?opName
	}
	LIMIT 15
	OFFSET 2

FIGURE 4.6: Query syntax comparison between LIMIT use and adding OFFSET use.

#	serviceName	opName
1	Google Fit	list-searchable-fields
2	Gmail API	list-searchable-fields
3	Swagger Petstore - OpenAPI 3.0	list-searchable-fields
4	USPTO Data Set API	list-searchable-fields
5	Google Blogger	list-searchable-fields
6	Youtube API	list-searchable-fields
7	Greenvivir-Backend	list-searchable-fields
8	Google Fit	retrieveHumidProperty
9	Gmail API	retrieveHumidProperty
10	Swagger Petstore - OpenAPI 3.0	retrieveHumidProperty
11	USPTO Data Set API	retrieveHumidProperty
12	Google Blogger	retrieveHumidProperty
13	Youtube API	retrieveHumidProperty
14	Greenvivir-Backend	retrieveHumidProperty
15	Google Fit	retrieveListOfSubscriptions

FIGURE 4.7: Previous results using LIMIT

#	serviceName	opName
1	Swagger Petstore - OpenAPI 3.0	list-searchable-fields
2	USPTO Data Set API	list-searchable-fields
3	Google Blogger	list-searchable-fields
4	Youtube API	list-searchable-fields
5	Greenvivir-Backend	list-searchable-fields
6	Google Fit	retrieveHumidProperty
7	Gmail API	retrieveHumidProperty
8	Swagger Petstore - OpenAPI 3.0	retrieveHumidProperty
9	USPTO Data Set API	retrieveHumidProperty
10	Google Blogger	retrieveHumidProperty
11	Youtube API	retrieveHumidProperty
12	Greenvivir-Backend	retrieveHumidProperty
13	Google Fit	retrieveListOfSubscriptions
14	Gmail API	retrieveListOfSubscriptions
15	Swagger Petstore - OpenAPI 3.0	retrieveListOfSubscriptions

FIGURE 4.8: Results Using LIMIT and OFFSET

Comparing previous results with the addition of OFFSET the expected results are still 15 , but it starts from the 3rd tuple of in Figure 4.5 .Also the last 2 extra tuples according to 4.3 Figure are next 2 after position 15. So it is obvious that the first and second tuples of Figure 4.5 have been skipped.

Order clause is responsible to rearrange data either by ascending or by descending order from a selected variable.The syntax form requires the keyword ORDER BY followed by either "*asc*" or "*desc*" keywords accordingly. In OASL these keywords are not case sensitive. At last , follows the variable where according to its results the data will be rearranged.

below it is presented an example and its results about this section of query:

LIMIT and OFFSET Example	LIMIT and OFFSET and ORDER BY Example

SELECT DISTINCT ?serviceName ?opName
WHERE {
Service title ?serviceName
Request method GET
Request operationName ?opName
}
LIMIT 15
OFFSET 2
	SELECT DISTINCT ?serviceName
	?opName
	WHERE {
	Service title ?serviceName
	Request method GET
	Request operationName ?opName
	}
	ORDER BY asc ?serviceName
	LIMIT 15
	OFFSET 2

FIGURE 4.9: Query syntax comparison between OFFSET and LIMIT use with OREDER BY clause with ascending order.

#	serviceName	opName
1	Swagger Petstore - OpenAPI 3.0	list-searchable-fields
2	USPTO Data Set API	list-searchable-fields
3	Google Blogger	list-searchable-fields
4	Youtube API	list-searchable-fields
5	Greenvivir-Backend	list-searchable-fields
6	Google Fit	retrieveHumidProperty
7	Gmail API	retrieveHumidProperty
8	Swagger Petstore - OpenAPI 3.0	retrieveHumidProperty
9	USPTO Data Set API	retrieveHumidProperty
10	Google Blogger	retrieveHumidProperty
11	Youtube API	retrieveHumidProperty
12	Greenvivir-Backend	retrieveHumidProperty
13	Google Fit	retrieveListOfSubscriptions
14	Gmail API	retrieveListOfSubscriptions
15	Swagger Petstore - OpenAPI 3.0	retrieveListOfSubscriptions

FIGURE 4.10: Previous results using LIMIT and ORDER

#	serviceName	opName
1	Gmail API	retrieveListOfSubscriptions
2	Gmail API	getPersonsByCURP
3	Gmail API	list-data-sets
4	Google Blogger	list-searchable-fields
5	Google Blogger	retrieveHumidProperty
6	Google Blogger	retrieveListOfSubscriptions
7	Google Blogger	getPersonsByCURP
8	Google Blogger	list-data-sets
9	Google Fit	list-searchable-fields
10	Google Fit	retrieveHumidProperty
11	Google Fit	retrieveListOfSubscriptions
12	Google Fit	getPersonsByCURP
13	Google Fit	list-data-sets
14	Greenvivir-Backend	list-searchable-fields
15	Greenvivir-Backend	retrieveHumidProperty

FIGURE 4.11: Results also ORDER BY ascending order

Adding also ORDER BY clause with ascending order it is obvious the rearrangement that happened at every tuple if we use the results from code without rearrangement and after that LIMIT and OFFSET are taking place .

4.2 Enriched Non-SPARQL Syntax rules

Most of the syntax rules this language follows are very similar to SPARQL syntax, and any clause in (OASL) also exists in SPARQL. However, in

some cases, to avoid the writing of extensive queries and to integrate the FILTER clause, this language introduces two keywords about comparison where they are placed as objects in (OASL) triples.

4.2.1 Comparison Query

Many fields of OpenAPI descriptions contain properties where their value are numbers. Even though SPARQL can make a query and ask about a property in ontology description using as object a number, range queries are more complex. It is supported with FILTER clause, but to avoid its integration in OASL we introduce the comparing mechanism positioned at the object placed inside WHERE clause. The syntax of a comparing triple contains an accepted subject, a property, which means a field where the object accepts numbers and, in object position, uses one of the comparator symbols below, followed by a number.

>	Greater than next number
>=	Greater or equal than next number
<	Lesser than next number
<=	Lesser or equal than than next number

below is presented an example containing its query syntax as also the effects in results:

Example without comparator

```
SELECT DISTINCT ?serviceName
?statusCode

WHERE {
Service title ?serviceName
Response statusCode ?statusCode
}
```

Example with comparator

```
SELECT DISTINCT ?serviceName
?statusCode

WHERE {
Service title ?serviceName
Response statusCode ?statusCode
Response statusCode >=500
}
```

FIGURE 4.12: Query syntax comparison between two examples.

#	serviceName	statusCode
1	Google Fit	200
2	Gmail API	200
3	Swagger Petstore - OpenAPI 3.0	200
4	Youtube API	200
5	Greenvivir-Backend	200
6	Google Fit	500
7	Gmail API	500
8	Swagger Petstore - OpenAPI 3.0	500
9	Youtube API	500
10	Greenvivir-Backend	500

FIGURE 4.13: Results without using comparing operators.

#	serviceName	statusCode
1	Google Fit	500
2	Gmail API	500
3	Swagger Petstore - OpenAPI 3.0	500
4	Youtube API	500
5	Greenvivir-Backend	500

FIGURE 4.14: Results using greater equal operator

4.2.2 Between Query

Another addition was the keyword *BETWEEN*.This keywords allows to compare between a range of values. It is equivalent to the typing of two triples about the same field using symbols "<=" ">=". Syntax positioning is the same with a compare query ,but in object position starts with keyword *BETWEEN* followed by parentheses, containing two number values separated by comma. Below there is an example using this rule.

Example without BETWEEN use

```
SELECT DISTINCT ?serviceName
?statusCode

WHERE {
Service title ?serviceName
Response statusCode ?statusCode
}
```

Example using BETWEEN

```
SELECT DISTINCT ?serviceName
?statusCode

WHERE {
Service title ?serviceName
Response statusCode ?statusCode
Response statusCode BETWEEN(100,400)
}
```

FIGURE 4.15: Query syntax comparison between two examples.

#	serviceName	statusCode
1	Google Fit	200
2	Gmail API	200
3	Swagger Petstore - OpenAPI 3.0	200
4	Youtube API	200
5	Greenvivir-Backend	200
6	Google Fit	500
7	Gmail API	500
8	Swagger Petstore - OpenAPI 3.0	500
9	Youtube API	500
10	Greenvivir-Backend	500

FIGURE 4.16: Results without BETWEEN use

#	serviceName	statusCode
1	Google Fit	200
2	Gmail API	200
3	Swagger Petstore - OpenAPI 3.0	200
4	Youtube API	200
5	Greenvivir-Backend	200

FIGURE 4.17: Results after triple containing BETWEEN

Chapter 5

System Implementation

This chapter describes all technical details about the system implementation and also the algorithmic procedures about the parser.

5.1 Tools Selection

This mechanism uses as database Openlink Virtuoso[7]. Virtuoso Universal Server is a middleware and database engine that combines the functionality of a traditional relational database management system (RDBMS), object-relational database (ORDBMS), virtual database, RDF, XML, free-text, web application server in a single system. However, rather than dedicate servers for each of the functionality mentioned above, Virtuoso is a "universal server" where it enables a single multithreaded server process that implements multiple protocols. Another considered database was Neo4j[1]. However, it selected Virtuoso because it has an integrated SPARQL endpoint, which is a feature that offers HTTP-based Query Services. Neo4j supports its query language (Cypher) and not SPARQL, something that would require an extra layer of translation for SPARQL to Cypher[2].

To parse OASL queries, it is used a lexical analyzer and a parser that was generated using JFlex and Java CUP tools, respectively. Java CUP [5] generates a Java program that will parse input that satisfies that grammar. On the other hand, Jflex [4] is the main lexical analyzer written in Java and works together with Java cup.

The user interface is written using Flask [3]. Flask is a micro web backend framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions. As a web template engine, Flask's default engine is Jinja2. [3]

[1] `https://neo4j.com/`
[2] `https://en.wikipedia.org/wiki/Neo4j`
[3] `https://en.wikipedia.org/wiki/Flask_(web_framework)`

5.2 Mechanism Description

This system comprises three basic components the openlink virtuoso , which is the database of our system, an user interface which is starting any uploading or querying action , and of course the translator which is responsible for parsing an OASL query and producing a equivalent SPARQL query.

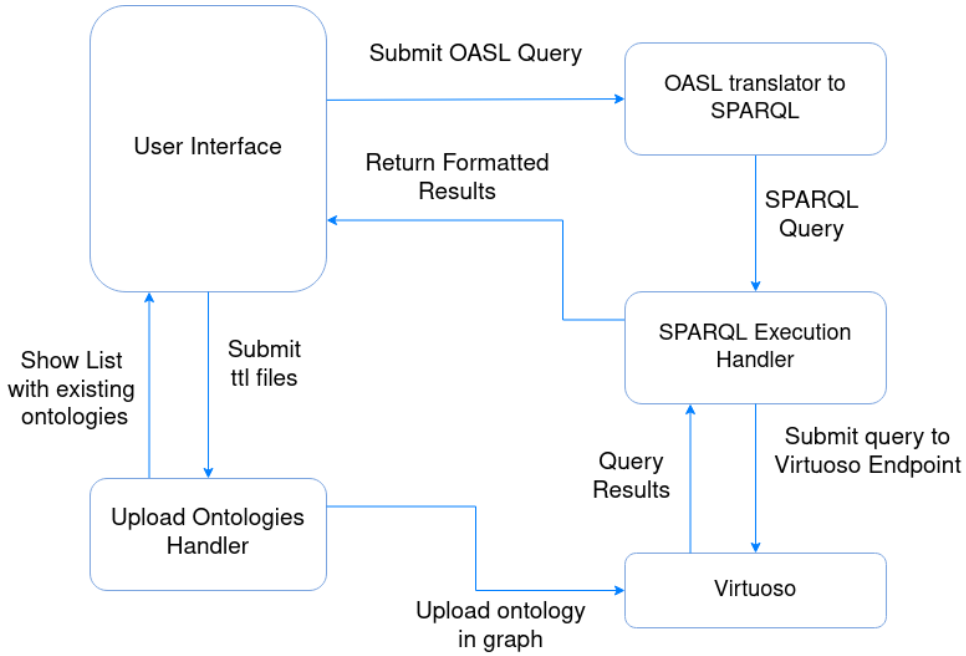

FIGURE 5.1: System Architecture Components

Communication between virtuoso is possible from virtuoso SPARQL endpoint.Virtuoso has the port '8890' available to execute SPARQL queries or to upload ontologies to the graph database.

User interface contains four pages which two of them are responsible for uploading files and writing queries in (OASL).The other two pages contain information about the way this app works. Specifically:

- **Home Page** is the starting page where a user is informed about the feature this site supports.
- **Upload Page** loads an environment where the user selects which ttl files want to upload to the virtuoso database for storage. It should be noted that the uploaded files are already OpenAPI ontologies, and any conversion process does not happen to the files. (Ontologies have been created in previous work. [2])
- **Execute Page** transfers the user to a simple environment where he can write an OASL query and after submitting it, can see the results in table format. If the query is not correct, throws an appropriate error.

- **Instructions Page** contains a PDF file with our simplified triples where this language accepts. As a result even if a user has not high level of openAPI knowledge he can just look what kind of information to ask and just type it.

Execute Query page consists from 3 elements . The text field is a re-adjustable rectangle space.When Execute page is selected is already containing a simple query as an example. On submit (the submit button is pressed), it posts the content of text field and after the translation and execution of the query in Virtuoso is returned in XML form the results. The next field presents the fetched results formatted as enumerated table rows. If an error has been occurred this field will present the exact error message that was happened.

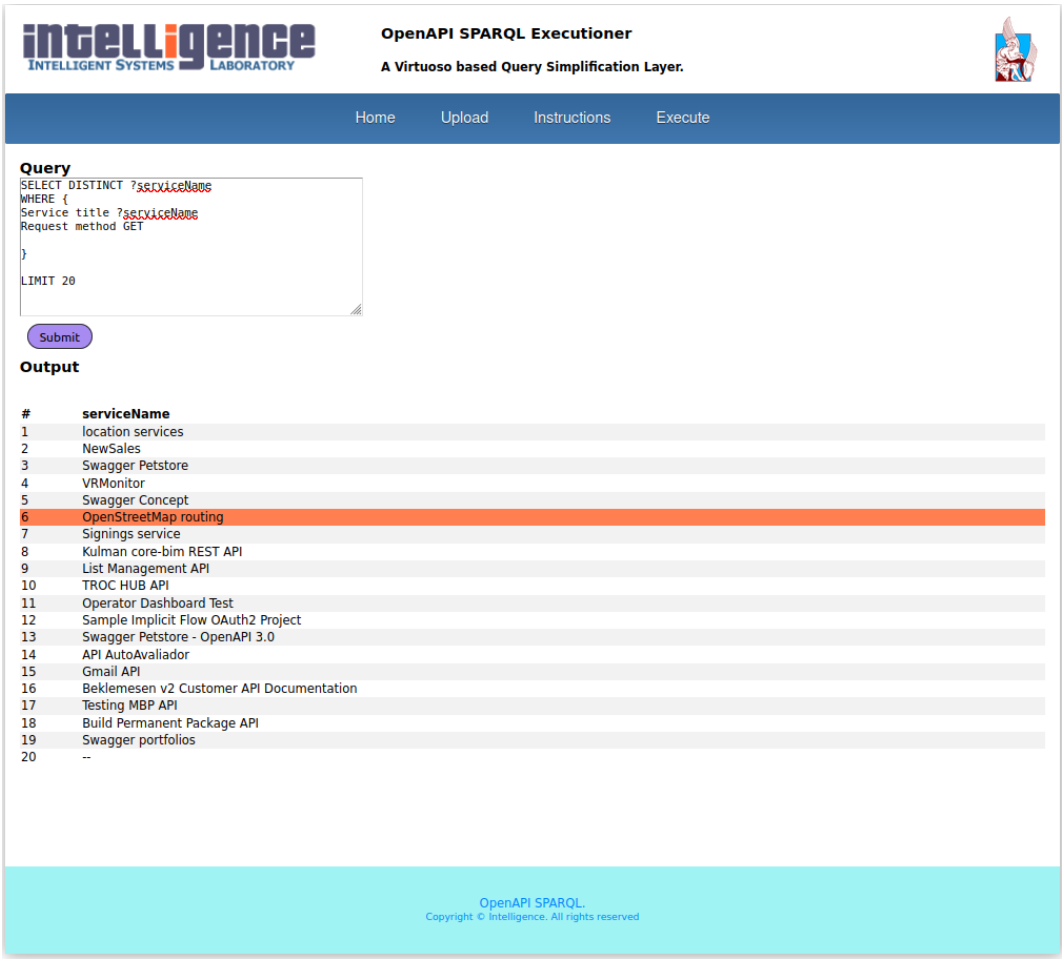

FIGURE 5.2: Execute OASL query page

The Upload page contains two simple buttons and one table.The purpose of this page is to store in database ontologies, which from the execute page a user can ask queries about them. The table displays all ontology file names that have been uploaded to our system. In case no files have been uploaded, the Virtuoso database is empty and this field is blank. The browse button opens a tab where the user can select which files will be uploaded. The user can upload more than one file at once. After their selection, he could press the Upload button. Then these files using the virtuoso endpoint are uploaded to Virtuoso to be stored, meaning they become part of Virtuoso's graph database. After completing the uploading process, the display table also includes the new files. Every file is uploaded with a specific URI endpoint and from this endpoint every triple are starting to extend their properties.

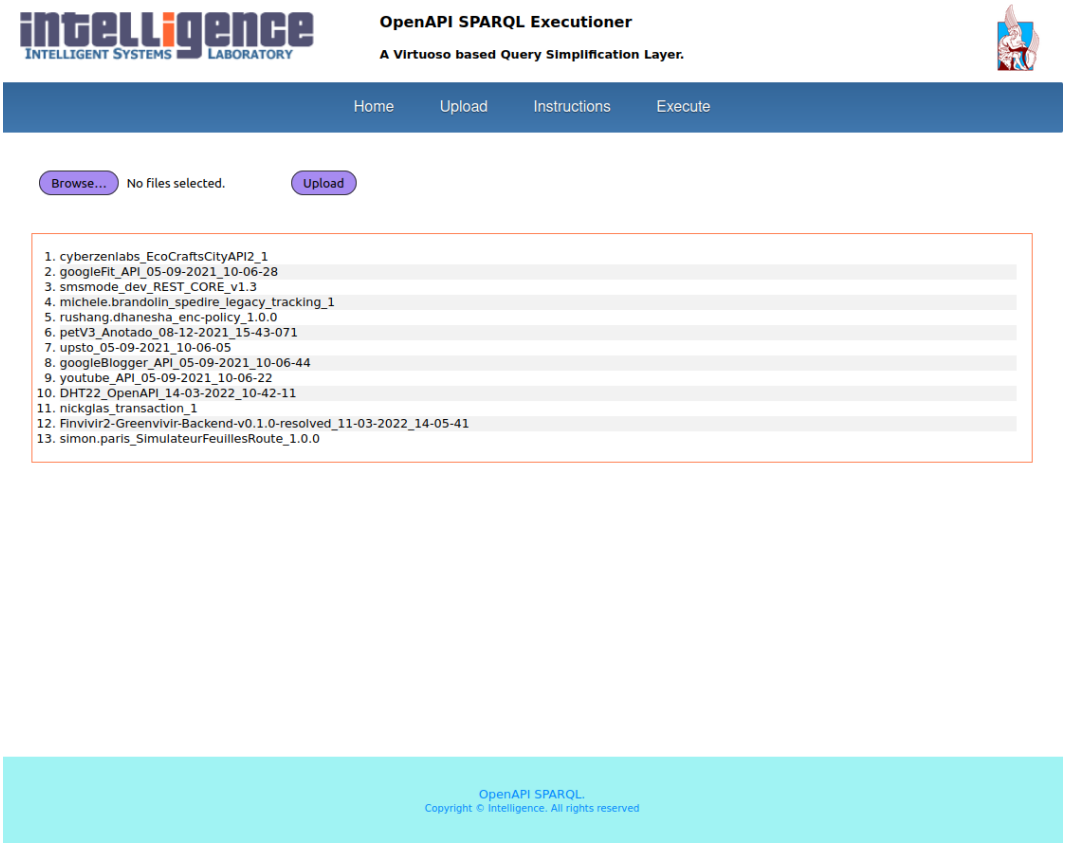

FIGURE 5.3: Execute OASL query page

Finally parsing and Translation mechanism is written in java. It is integrated into the system as an executable JAR file as argument input takes in string format query where the user submits and returns the generated SPARQL query as output. If an error occurs during the process is accessible from the *system.err* output stream.

5.3 Parsing Algorithm

Below are presented the algorithms for parsing a query. It is described in three main procedures:

Algorithm 1 Query Parsing Algorithm

```
addIntitalPrefixes()
if PREFIX clause is detected then
    while new prefix exists do
        ParsePrefixStatement()
Ensure: Prefix is unique
        PrefixList ← addnewprefixes()
    generatePrefixSPARQLCode()

Ensure: next clause = SELECT
  ParseVariables()
  generateSelectSPARQLCode(identified variables)

Ensure: next clause = WHERE
  while current clause = WHERE do
    structure ← RecogniseTripleStructuring
    ParseTriples()
    if structure = OPTIONAL then
Ensure: Optional syntax rules
    else if structure = NOT then
Ensure: NOT syntax rules
    else if structure = UNION then
Ensure: UNION syntax rules
    generateWHERESPARQLCode(structure)

  while next clause = rearrange do   ▷ LIMIT , OFFSET or ORDER BY
    if rearrange = LIMIT then
        Parse LIMIT clause
    if rearrange = OFFSET then
        Parse OFFSET clause
    if rearrange = ORDER BY then
        Parse ORDER BY clause
    correctPositioning ← checkIfClausespositionAreInOrder()
    if correctPositioning is false then return Syntax Error
    else
        generateRearrangeSPARQLCode()
```

When a user submits a query in OASL, it executes the same algorithm

for parsing described above. This algorithm, specifically at first, examines if they have been declared any prefixes and stores them in a HashSet for later validation syntax in triples where it can be used. Also, if they exist, this triple throws an error message to the user. When a SELECT keyword is scanned examines the syntax of the following variables and creates the equivalent part of the SPARQL final code.WHERE keyword means that WHERE clause has started, so the algorithm expects either to recognize triples or to recognize OPTIONAL, UNION and NOT keywords to generate an equivalent SPARQL code. At last, examine if any rearrange clauses are in the correct order. Any clause is detected by the keyword that represents (LIMIT, ORDER BY, OFFSET) and checks if their positioning is correct, else returns an error message to the user. If every step is successful, then is created the executable SPARQL query.

Algorithm 2 Parse Triples algorithm

```
while nextTriple exists do
    subject ← scanSubject()
    if subject = Service then
        predicate ← scanServicePredicateField()
    else if subject = Request then
        predicate ← scanResponsePredicateField()
        if predicate =schema then
    else if subject = Parameter then
        predicate ← scanParameterPredicateField()
    else if subject = Tag then
        predicate ← scanTagPredicateField()
    else if subject = Header or RequestHeader or ResponseHeader
then
        predicate ← scanHeaderPredicateField()
    else if subject = Server then
        predicate ← scanServerPredicateField()
    else if subject = Sequrity then
        predicate ← scanSequrityPredicateField()
    else if subject = SequrityScope then
        predicate ← scanSequrityScopePredicateField()
    else if subject = Response then
        predicate ← scanResponsePredicateField()
    else if subject = Schema then
        predicate ← scanSchemaPredicateField()
    else if subject = Property then
        predicate ← scanPropertyPredicateField()
    else if subject = variable then
        if schemaVariableList contains variable then
            Variable requires Schema fields
        else if propertyariableList contains variable then
            Variable requires Schema fields
        elsereturn Error syntax message

    [isCommon, object] ← specialTripleConditions()
    if isCommon is true then
        object ← commonTripleParse()
    SparqlTripleStack ← mergeTriple(subject, predicate, object, filterClause)
return SPARQLQuery
```

The algorithm parses and creates the executable part of SPARQL code responsible for triples follows a specific procedure. First scans the subject of any triple which means the object table contains several expected fields. In most cases , when the subject is one of the Tables described in the previous section , he parses the predicate and expects to be a

member of his field. If the subject is a variable, searches in two lists to recognise if this variable has been declared as schema or property variable and is treated accordingly. Any other case is syntactical incorrect. When recognised the subject and predicate examines if this triple expect a common syntax as in most object is happening or is used one of the special triples. If it is common , starts a procedure to recognise the object and at last merges these information and creates the equivalent SPARQL triple code part.

Algorithm 3 Common Triple Parsing Conditions

```
  Parse object value
  if object = String, Number, Boolean, uri , variable then
      Check if this Object is acceptable in triple
  else if object = prefix then
Ensure: Prefix value has been declared
  else if Comparison object then
      declare new variable
      Codeobject ← newVariable
      addFILTERClause(comparator number)
  else if Between object then
      declare new variable
      Sparqlobject ← newVariable
      addFILTERClause(newVariable ≥ num1 and newVariable ≤
  num2)
```

This algorithm describes the procedure that is followed to identify the object of a triple. As object in OASL it can be common values such as string , number, boolean or any uri. Each one of them it is recognized instantly be the lexical analysis by the rules presented above. Identical to this is treated also any common variable in object position. After its spotting, the parser examines if the syntax rule is correct and then adds the last part of SPARQL code for this triple . When a prefix is included as object , it is examined if has been declared first and then follows the same procedure as before. If a comparator with a number is the object , according with the comparator type, SPARQL object becomes a new declared variable and is adding a FILTER clause with the appropriate conditions to express this comparison. At last if BETWEEN keyword is detected the same procedure is followed comparing the variable with two conditions : greater equal (>=) and lesser equal (<=) .

5.3.1 Special behaviour parsing conditions

The procedures above describe the behavior for the most triples that they are defined in this work. However, for some triples, even though they have the same syntax in subject and predicate, the object of the

triple datatype (for example :string or uri), can determine which is the final property in generated SPARQL code.An example is schema property. If the user specifies a string as an object, the matched property inside schema class will be *rdfs:label* .When the user specifies a URI as object , then the matched property will be *sh:targetClass*. These *'special' triples* are the ones where field name (predicate) is schema or property , which they are described in Chapter 3 .
Below is the algorithm about handling these situations:

Algorithm 4 Special Triple Conditions

```
if predicate is schema then
    if object = string then
        declare new variable
        Codeobject ← newVariable
        add schema triple until name property
    else if object = uri then
        declare new variable
        Codeobject ← newVariable
        add schema triple until targetClass endpoint
    else if object = prefix then
Ensure: Prefix value has been declared
        declare new variable
        Codeobject ← newVariable
        add schema triple until targetClass endpoint
    else if object = common datatypes then
        declare new variable
        Codeobject ← newVariable
        add triples until property class datatype field
    else if object = variable then
        add variable to schemaVariableList
else if predicate = property then
    if object = string then
        declare new variable
        Codeobject ← newVariable
        add property triples until name property
    else if object = uri then
        declare new variable
        Codeobject ← newVariable
        add property triple until path property
    else if object = prefix then
Ensure: Prefix value has been declared
        declare new variable
        Codeobject ← newVariable
        add schema triple until path endpoint
    else if object = variable then
        add variable to propertyVariableList
```

This algorithm examines if predicate is schema or property. If it is schema keyword then scans the object expecting one of the following options from lexical analysis. When is string creates a SPARQL code part referring to the name property endpoint of Schema object in ontology document . Uri and prefixes are matched with the targetClass property endpoint.In prefix option it is also examined if the expected prefixed has been declared in prefix clause. Another object field that is allowed are the common datatypes where this language uses .When lexical analysis finds one of these types ,this shortcut creates a SPARQL code part referring to the datatype property endpoint of Property Object in ontology document. At last if a variable is detected , is declared by adding it to the schema variable list as schema variable. This variable can be used in later triples as subject. If predicate is property keyword , in string creates SPARQL code part name property endpoint of Property object, in uri and prefix is the same procedure but until the path endpoint an finally, variables are stored in property variable list to be used as Property subjects.

5.4 Translation and Execution Procedure

Below it is explained the whole sequence of action that take place from submitting a query until the appearance of results.

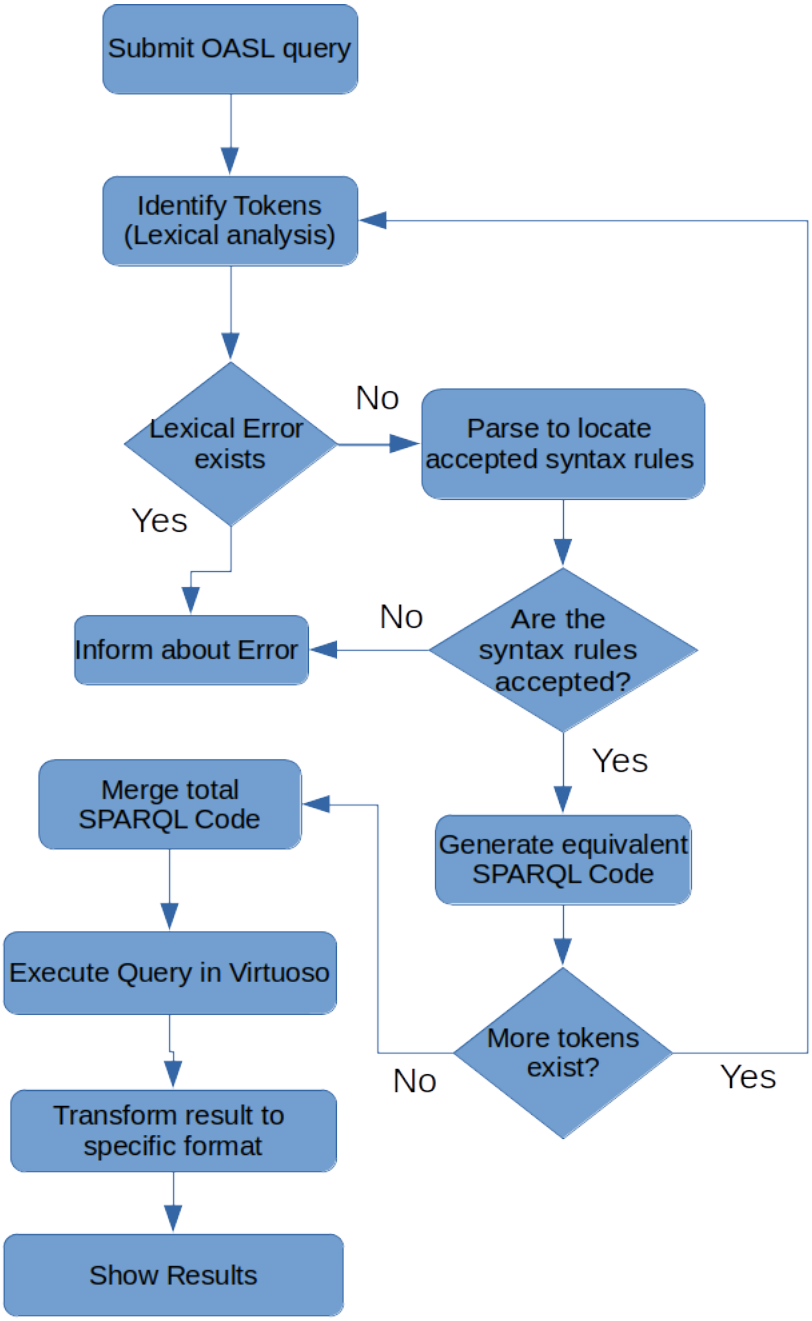

FIGURE 5.4: Translation and Execution sequence diagram

The whole procedure starts from the moment the user submits a query.

The compiler scans this query word by word and executes the algorithm described in Algorithm 1. Suppose there are no errors to return in the user interface and stop the procedure. In that case, the generated SPARQL code is given as input through the SPARQL endpoint, from the virtuoso Jena provider, in Virtuoso and executes this query. Any results are returned in XML format, where they are transformed to be presented in table format in User Interface.

Chapter 6

Results and Measurements

6.1 Performance Analysis

In order to take some solid measurements in numbers about the performance of this work. We test by uploading 300 Semantic openAPI descriptions.Afterwards we test some different queries as examples , simple and more complex and extended , counting their reaction time and analyzing these results. These reaction time measurements are:

- The time difference between start of translation mechanism until the production of the executable SPARQL code.
- The time difference between the moment the SPARQL query is executed in virtuoso until the time where results are became available.
- The total time from the submission of OASL query until the final results presentation to the user.

6.1.1 Non-optimal SPARQL Triples Analysis

It should be noted that the produced SPARQL queries are not optimal because some RDF triples are sometimes repeated in the final code. This phenomenon happens when the parser recognizes a triple from the same OpenAPI object and generates some parts of the final RDF graph triples endpoint, which is possible to already used. For example most triples such Request , Response and Tag in OASL, their triple paths until their endpoints in ontology are sharing the *openapi:supportedOperation* property. As a consequence , the generated SPARQL code will contain more than once this triple.

To examine if there are significant delays, we test a SPARQL query in Virtuoso SPARQL endpoint with an extreme (bad) and nonrealistic example containing a repeated pattern of triples 300 times. Compared to an equivalent optimized one (3 triples), this example revealed a minimal delay (1ms), which is insignificant compared to the time a SPARQL

query needs to be executed. Considering that the number of these repeated triples will never exceed 30 triples, we conclude that there is no point in optimizing this issue.

6.2 Query Categories

In order to examine the reaction times, we must determine which criteria we categorized the tested queries. The OASL translation process contains the scanning process, the SPARQL code generation process following each recognized rule, and any additional control that the rules are producing a meaningful query(does not contain unnecessary declarations, wrong variables, etc.). Any delay in the translation mechanism proceeds from how extensive is the OASL code and the variety of different clauses. The complexity of the SPARQL execution process depends on the number of different triple endpoints. Every triple which is added in the WHERE clause means that an extra condition must be matched in the database's graph.

The queries below are scaling according to these rules :

- **Triple increase** :More triples, more time consuming
- **Clause variety** : More clauses more difficult to process.

6.2.1 Simple queries

As simple query we define any query where it is requested to match with maximum 2 or 3 different triples, which means that any intersections inside graph is very limited, from execution perspective. From the translation perspective it should contain only the necessary clauses with a minimal amount of triples. below is presented an example of a simple OASL code and its SPARQL equivalent code as also its timings.

Simple Example	Equivalent SPARQL Example

SELECT ?serTtile ?statCode ?opName
WHERE {
Service title ?serTtile .
Response statusCode ?statCode .
Request operationName ?opName .
 	
}
	PREFIX rdf:
	<http://www.w3.org/1999/02/22-rdf-
	syntax-ns#>
	PREFIX owl:
	<http://www.w3.org/2002/07/owl#>
	PREFIX xsd:
	<http://www.w3.org/2001/XMLSchema#>
	PREFIX rdfs:
	<http://www.w3.org/2000/01/rdf-
	schema#>
	PREFIX myOnt:
	<https://www.example.com/service/goo
	gleBooks_API#>
	PREFIX openapi:
	<http://www.intelligence.tuc.gr/ns/o
	pen-api#>
	PREFIX sh:
	<http://www.w3.org/ns/shacl#>
	SELECT ?serTtile ?statCode ?opName
	WHERE {
	?service a openapi:Document .
	?service openapi:info ?info .
	?info openapi:serviceTitle ?serTtile
	.
	?service openapi:supportedOperation
	?operation .
	?operation openapi:response
	?response .
	?response openapi:statusCode
	?statCode .
	?service openapi:supportedOperation
	?operation .
	?operation a openapi:Operation .
	?operation openapi:name ?opName .
	
	}

FIGURE 6.1: Presentation of a Simple OASL and SPARQL translation query

This example describes a query in (OASL) where the user wants from each matched answer the service title of the openAPI document ,the operation name about each operation inside and the response status code for every responses described.In OASL it is a simple and small query containing only the necessary clauses (*SELECT* and *WHERE*) . They are requested only 3 triples , which they belong to 3 separate Table Object groups.The SPARQL equivalent query is more extended , however contains only and operations between triples.

The below figure shows the structure of retrieved data from the simple example. The results were significantly more, so they are cropped for better displaying them in the figure:

#	serTtile	statCode	opName
1	Swagger Petstore	200	showPetById
2	Swagger Petstore	200	showPetById
3	Swagger Petstore	200	showPetById
4	Swagger Petstore	200	showPetById
5	Swagger Petstore	200	showPetById
6	Swagger Petstore	200	showPetById
7	Swagger Petstore	200	showPetById
8	location services	200	showPetById
9	Bridge Point Insurance	200	showPetById
10	EXAMPLE	200	showPetById
11	NewSales	200	showPetById
12	VRMonitor	200	showPetById
13	Swagger Petstore - OpenAPI 3.0	200	showPetById
14	Swagger Petstore - OpenAPI 3.0	200	showPetById
15	Swagger Petstore - OpenAPI 3.0	200	showPetById
16	Swagger Petstore - OpenAPI 3.0	200	showPetById

FIGURE 6.2: Results fetched by Simple Query.

After executing the simple query, we return many results, showing only a tiny fraction. We can see that the first data column projected the names of the service in variable *?serTitle*, as exactly was expected. Also, *?statCode* variable displays in second column the status code numbers as exactly were instructed to fetch according to the second triple (Response statusCode ?statCode). At last name from operations appear in the third column. We can we that many results are repeated, which is normal because the DISTINCT keyword is not used. These repeated tuples exist because the OpenAPI document matches the requested triple patterns from the example above many times.

Below are presented delay timings for simple queries:

Translation Time :	35ms
SPARQL Execution time :	248ms
Total Time :	2193ms

After running similar simple queries as the average time for translating procedure was 35 milliseconds , which is a small fracture comparing to the execution time and insignificant to the total time needed for revealing the results.

6.2.2 Complex queries

Complex query in SPARQL we define any query that contains many triple endpoints as also many different clauses.Clauses such as FILTER or rearrangement clauses , they should be more time consuming because they require extra data processing . In OASL translation mechanism a query is more complex, when is demanded to run extra procedures to decide the final executable code output or to ensure that some syntax rules are properly used.These procedures

below is presented an example of a simple OASL query and its SPARQL equivalent code ,as also its timings. This example contains 3 clauses (*PREFIX SELECT* and *WHERE*) and Where clause contains in its body also *OPTIONAL* and *OR*.Comparing this query to the previous one in

Figure 6.2, is enriched by asking also In response Object the required content schemas to be compatible ,as they have been declared, in two different URI's (*myOnt:Pet* and *myOntb:Pets*). The status code field is removed from this example. At last information about Tag and Server objects that are requested are declared optional . The equivalent SPARQL query is far more extended as it seems and expresses using rdf triples the graph paths that is followed until the endpoints of the field where it described above.

Extended query Example

```
PREFIX
myOntb:<https://www.example.com/service/petstore#>
SELECT ?serTtile ?opName ?tgName ?ServerUrl
?parName ?parDesc
WHERE {
Service title ?serTtile .
{Response schema myOnt:Pet .}
OR
{Response schema myOntb:Pets .}
Request operationName ?opName .
OPTIONAL {Tag name ?tgName .}
OPTIONAL {Server url ?ServerUrl. }
}
```

Equivalent SPARQL Example

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
schema#>
PREFIX myOnt:
<https://www.example.com/service/googleBooks_API#>
PREFIX openapi:
<http://www.intelligence.tuc.gr/ns/open-api#>
PREFIX sh:    <http://www.w3.org/ns/shacl#>
PREFIX myOntb:
<https://www.example.com/service/petstore#>

SELECT ?serTtile ?opName ?tgName ?ServerUrl
?parName ?parDesc
WHERE {
?service a openapi:Document .
?service openapi:info ?info .
?info openapi:serviceTitle ?serTtile .
{
?service openapi:supportedOperation ?operation .
?operation openapi:response ?response .
?response openapi:content ?r_con .
?r_con openapi:schema ?schemaTypeProp1 .
?schemaTypeProp1 a        sh:NodeShape .
?schemaTypeProp1 sh:targetClass myOnt:Pet .
}
UNION {
?service openapi:supportedOperation ?operation .
?operation openapi:response ?response .
?response openapi:content ?r_con .
?r_con openapi:schema ?schemaTypeProp2 .
?schemaTypeProp2 a        sh:NodeShape .
?schemaTypeProp2 sh:targetClass myOntb:Pets .
}
?service openapi:supportedOperation ?operation .
?operation a openapi:Operation .
?operation openapi:name ?opName .
OPTIONAL { ?tag a openapi:Tag .
?tag openapi:name ?tgName .
}
OPTIONAL { ?service openapi:supportedOperation
?operation .
?operation  openapi:serverInfo      ?server .
?server a   openapi:Server .
?server openapi:host ?ServerUrl .
}
}
```

FIGURE 6.3: Medium complexity query

below are presented delay timings in extended queries:

Translation Time :	39ms
SPARQL Execution time :	1643ms
Total Time :	8341ms

After running similar queries with medium complexity the average time for translating procedure was **39 milliseconds** , which is a minimal delay compared to the execution time and total time where a significant increase is observed about to 1.6 and 8.4 seconds respectively .

6.2.3 Extended and Complex queries

It is defined as an extremely complex and extended query, if its body includes any available clause and using in where clause between or comparison triples and furthermore a variety of triples from different OpenAPI objects. A query with these characteristics has much more important delays for two reasons. First an extended query with many OASL triples requires much more parsing time, even though it is not expected to influence significantly any results.The second reason is that prefixes , comparisons and the use of variables as subject in triples are using additional functions something that according to the query could cause additional.However these extreme extended and complex queries have not any practical value for a user to make them. In most cases a users query it won't contain more than 20 lines of code.

The example below contains every possible clause in OASL.The purpose of its presentation is mainly to express how the translation mechanism can handle it. Initially they are declared a number of Prefixes, so they will be available later in where clause . *SELECT* clause follows projecting the variables where the user demands. The core of this query contains some OPTIONAL and UNION bodies along with normal intersection between OASL triples.The requested thirteen fields originated from seven Semantic OpenAPI Objects (Service, Request, Parameter, Response,Tag ,Server and Response Server. In comparison with the previous example it is requested status codes at response fields should be between 100 and 450. Additionally the Header in Response object in requested as style option to be simple.Finally the after *WHERE* clause are following three rearrangement rules .They limit any results to 60 lines skipping the first 5 , but first they have been ascending ordered by the service title variable (*?serviceName*) .

Complex and Extended query Example

```
PREFIX myOntb1:<https://www.example.com/service/petstore1#>
PREFIX myOntb2:<https://www.example.com/service/petstore2#>
PREFIX myOntb3:<https://www.example.com/service/petstore2#>
PREFIX myOntb4:<https://www.example.com/service/petstore4#>
PREFIX myOntb5:<https://www.example.com/service/petstore5#>
PREFIX myOntb6:<https://www.example.com/service/petstore6#>
PREFIX myOntb7:<https://www.example.com/service/petstore7#>
PREFIX myOntb8:<https://www.example.com/service/petstore8#>
PREFIX myOntb9:<https://www.example.com/service/petstore9#>
PREFIX myOntb10:<https://www.example.com/service/petstore10#>
PREFIX myOntb11:<https://www.example.com/service/petstore11#>
PREFIX myOntb12:<https://www.example.com/service/petstore12#>
PREFIX myOntb:<https://www.example.com/service/petstore#>
SELECT DISTINCT ?serviceName ?serDesc ?opName ?tgName ?serverUrl
?parName ?headName ?headDesc
WHERE {
Service title ?serviceName .
Service description ?serDesc .
Parameter name ?parName .
OPTIONAL {Parameter required TRUE . }
Response statusCode BETWEEN(100,450) .
{Response schema myOnt:Pet .}
OR
{Response schema myOntb:Pets .}
Request operationName ?opName .
OPTIONAL {Tag name ?tgName .}
OPTIONAL {Server url ?serverUrl .}
ResponseHeader name ?headName .
OPTIONAL {ResponseHeader description ?headDesc .}
ResponseHeader style simple .

}

ORDER BY asc ?serviceName
LIMIT 60
OFFSET 5
```

FIGURE 6.4: Extended Query containing every major clause

Below are presented response times for highly extensive and complex queries with many matching results:

Translation Time :	41ms
SPARQL Execution time :	16215ms
Total Time :	21003ms

below are presented response times for highly extensive and complex queries without matching results:

Translation Time :	41ms
SPARQL Execution time :	1451ms
Total Time :	1547ms

After running queries containing as much as possible parameters , the average time for translating procedure was **41 milliseconds**. However because of the amount of fields with specific conditions that are requested the response times about SPARQL execution and total time presenting some variations. When Virtuoso returned enough rows as results which means SPARQL mechanism has found enough triples to match , execution time was almost 17 seconds.The total amount of time was approximately to 21 seconds. When similar queries return no results the SPARQL execution times are much smaller to 1451ms following similar trend of course and total response time.

In addition is presented the generated SPARQL query which is executed in Figure 6.4:

Equivalent SPARQL Example

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX myOnt: <https://www.example.com/service/googleBooks_API#>
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api#>
PREFIX sh:    <http://www.w3.org/ns/shacl#>
PREFIX myOntb1: <https://www.example.com/service/petstore1#>
PREFIX myOntb2: <https://www.example.com/service/petstore2#>
PREFIX myOntb3: <https://www.example.com/service/petstore2#>
PREFIX myOntb4: <https://www.example.com/service/petstore4#>
PREFIX myOntb5: <https://www.example.com/service/petstore5#>
PREFIX myOntb6: <https://www.example.com/service/petstore6#>
PREFIX myOntb7: <https://www.example.com/service/petstore7#>
PREFIX myOntb8: <https://www.example.com/service/petstore8#>
PREFIX myOntb9: <https://www.example.com/service/petstore9#>
PREFIX myOntb10: <https://www.example.com/service/petstore10#>
PREFIX myOntb11: <https://www.example.com/service/petstore11#>
PREFIX myOntb12: <https://www.example.com/service/petstore12#>
PREFIX myOntb: <https://www.example.com/service/petstore#>

SELECT DISTINCT ?serviceName ?serDesc ?opName ?tgName ?serverUrl ?parName ?headName
?headDesc
WHERE {
?service a openapi:Document .
?service openapi:info ?info .
?info openapi:serviceTitle ?serviceName .
?service openapi:description ?serDesc .
?operation openapi:parameter ?parameter .
?parameter openapi:name ?parName .
OPTIONAL { ?operation openapi:parameter ?parameter .
?parameter openapi:required TRUE .
}
?service openapi:supportedOperation ?operation .
?operation openapi:response ?response .
?response openapi:statusCode  ?compareVariable1 .
FILTER ( STR(?compareVariable1) >= "100" && STR(?compareVariable1) <= "450" )
{
?service openapi:supportedOperation ?operation .
?operation openapi:response ?response .
?response openapi:content ?r_con .
?r_con openapi:schema ?schemaTypeProp1 .
?schemaTypeProp1 a       sh:NodeShape .
?schemaTypeProp1 sh:targetClass myOnt:Pet .
}
UNION {
```

```
?service openapi:supportedOperation ?operation .
?operation openapi:response ?response .
?response openapi:content ?r_con .
?r_con openapi:schema ?schemaTypeProp2 .
?schemaTypeProp2 a       sh:NodeShape .
?schemaTypeProp2 sh:targetClass myOntb:Pets .
}
?service openapi:supportedOperation ?operation .
?operation a openapi:Operation .
?operation openapi:name ?opName .
OPTIONAL { ?tag a openapi:Tag .
?tag openapi:name ?tgName .
}
OPTIONAL { ?service openapi:supportedOperation ?operation .
?operation  openapi:serverInfo     ?server .
?server a   openapi:Server .
?server openapi:host ?serverUrl .
}
?service openapi:supportedOperation ?operation .
?operation openapi:response ?response .
?response openapi:responseHeader ?response_header .
?response_header openapi:name ?headName .
OPTIONAL { ?service openapi:supportedOperation ?operation .
?operation openapi:response ?response .
?response openapi:responseHeader ?response_header .
?response_header openapi:description ?headDesc .
}
?service openapi:supportedOperation ?operation .
?operation openapi:response ?response .
?response openapi:responseHeader ?response_header .
?response_header openapi:style openapi:simple .

}
ORDER BY asc (?serviceName)
LIMIT 60
OFFSET 5
```

FIGURE 6.5: Generated SPARQL Code from Complex and Extensive Example

This example indicates clearly how complex could be a SPARQL query and of course because of its extent ,that any human user would be prone errors when is writing directly a query like that.

6.3 Comparison Between Response Times

The next table presents the response times from the previous experiments and furthermore expresses in comparison, translation and execution delays with the total amount of time for submitting a query until the display of the results to the user.

Query Type:	Simple	Complex	Complex & Extended (no results)	Complex & Extended (results)
Translation Time :	35ms	39ms	41ms	41ms
Execution time :	248ms	1643ms	1451ms	16215ms
Total Time :	2193ms	8341ms	1547ms	21003ms
Translation/Total Time % :	1.59 %	0.46 %	2.45 %	0.19 %
Execution/Total Time % :	11.3 %	19.7 %	93.8 %	77.2 %

Observing and comparing the results from the previous queries it is noticeable that translation mechanism has a reliable and small delay to the total system timings.When a query is simple the generated SPARQL code includes less RDF triple lines than more complex and extended queries. When a query becomes more extensive there is a slight increase (4ms) to the translation mechanism but according to the increasing delays from Virtuoso database response and standard delays from the system processes is minimal. When an OASL query becomes contains every possible clause and includes many difficult conditions to tackle, such as *BETWEEN* , there also a slighter translation time increase.The basic response delay at this experiment arises from the execution of generated SPARQL queries in Virtuoso and the results fetching process.

In conclusion the OASL compiler and translation mechanism executes its tasks at a stable and insignificant amount of time. Most important delays are due to the execution of the SPARQL as they contain a large number of RDF triples, searching in files with thousands lines of RDF data .

6.4 Actual Ontology OpenAPI Data

An issue worth exploring was the size gap between OpenAPI document files and OpenAPI Ontology equivalent descriptions. Specifically, we select some files from our database and convert them to ontology. Virtuoso database, to recognize an ontology document as valid and consistent, must include every triple statement from the Ontology's schema. The ontology's schema is every RDF statement necessary to properly define every property that describes the individuals of a document. However, this information has no value to the querying process.

To understand the impact of this code on the total execution, it is necessary to examine how many triples are explicitly referred to in the OpenAPI information. The files (openAPI[1] and ontologies[2]) are selected randomly from Semantic OpenAPI from previous work.

The files which selected are:

- File1: Cisco521_SaaS-Connect-Provision
- File2: EliteFintech_list_management
- File3: youtube_API
- File4: petV3_Anotado
- File5: googleBlogger_API

Below is presented a table which contains details about the selected files:

Files Selected:	File1	File2	File3	File4	File 5
Yaml file lines :	20	493	170	844	552
Ontology Individual lines :	10	580	357	1031	1128
Ontology Schema lines :	2107	2173	2180	2192	2287
Individual triples % :	0.47 %	26.69 %	14.07 %	32%	33%
Yaml file size (KB) :	0.505	16.8	6.5	23.5	19.8
TTL file size (KB) :	90.3	127.1	116	174.5	179

Comparing the number of lines referring to the individuals with the rest of the code, we observe that Ontology's schema lines have a stable number of triples. This derives from the conversion algorithm adding all the statements to describe the openAPI schema in every document, irrespectively from the real OpenAPI related object and properties. Additionally, we see that Individual triple lines are proportional to the number of lines from the initial OpenAPI descriptions (YAML format), which validates that genuine information is related only to individual triples. Considering also that information for ontology (TTL) files in most cases is below 30%, which means that 70% of data have no importance in the querying procedure but take part delaying the execution time.

[1] `https://www.intelligence.tuc.gr/semantic-open-api/descriptions`
[2] `https://www.intelligence.tuc.gr/semantic-open-api/ontologies`

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This work had as its goal to introduce a new query language for Semantic OpenAPI descriptions written in Semantic OpenAPI based in SPARQL.
Comparing OASL with SPARQL as a different RDF query language approach, OASL achieves the following advantages:

- An easier syntax and more familiar to a user knowing REST architecture without knowing how an OpenAPI description is transformed into an ontology.
- It is familiar with SPARQL, a standard language for RDF ontologies, but it does not require extensive knowledge of this language.
- Translation process is fast, so it is not causing significant delays compared with an equivalent SPARQL execution system.
- Imports clever methods to express more minor triples and avoid implementing extra SPARQL clauses.

7.2 Future Work

An idea for future work could be expanding the OpenAPI objects described in this language. Because the conversion of an OpenAPI description to its Semantic form is based on previous work, Links and Callbacks are not supported. Furthermore, Webhooks is another addition for future work because this language is made for the 3.0 OpenAPI version and not 3.1.

Another idea for future work could also be the fragmentation of RDF data storage. Every ontology saved in the database contains many triples repeated each time, creating huge files and causing significant delays to the execution process.

This work does not alter any data inside ontologies. Another direction worth exploring is the use of indices. The system can insert an id

x-property (PropertyShape in the ontology) during the uploading process. To keep a syntax similar to SPARQL, the FROM clause (not used in OASL) can be implemented.

An interesting idea in addition to future work is the creation of an environment where the user can also download one of the ontology files shown in his result tab. This can be implemented either by importing a unique id for retrieving each document or using the SPARQL GRAPH function.

Bibliography

[1] D. Miller et al. *OpenAPI Specification v3.0.0*. 2017.

[2] F. Bouraimis. "Instantiating OpenAPI Descriptions to the REST Services Ontology". Diploma Thesis. School of Electrical and Computer Engineering, Technical University of Crete, 2021.

[3] Miguel Grinberg. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.", 2018.

[4] *JFlex*. `https://www.jflex.de/docu.html`.

[5] *LALR Parser Generator for Java*. `http://www2.cs.tum.edu/projects/cup/index.php`.

[6] Nikolaos Mainas. "Semantically enriched API descriptions for improving service discovery in cloud environments". Master Thesis. School of Electrical and Computer Engineering, Technical University of Crete, 2017.

[7] *OpenLink Virtuoso Universal Server*. `https://docs.openlinksw.com/virtuoso/`.

[8] *Resource Description Framework*. `https://www.w3.org/RDF/`.

[9] *Semantic Web*. `https://www.w3.org/standards/semanticweb/`.

[10] *Web Ontology Language*. `https://www.w3.org/TR/owl2-overview/`.