# Searching in REST Service Catalogues with OpenAPI Descriptions

IOANNA MARIA STERGIOU

**Supervisor:** Prof. Euripides G.M. Petrakis
Assoc. Prof. Vasilios Samoladas
Assoc. Prof. Michail G. Lagoudakis

A thesis submitted in fulfilment of
the requirements for the degree of
Diploma in Electrical and Computer Engineering

School of Electrical and Computer Engineering
Technical University of Crete

October 2021

# Abstract

This work presents OpenPI QL, a query language for querying OpenAPI service descriptions. The basic idea behind OpenAPI QL approach is that the OpenAPI document is a description of a REST request with the corresponding responses. The request that is sent from client to the server, and the responses returned back to the client, are actions that OpenAPI describes thoroughly. An important factor of this work is the capability of determining whether data are usable for being searched. Equally important is the handling of data complexity. OpenAPI QL is modeled to deal with JSON files, which are semi-structured data. Moreover, OpenAPI QL requires that the user be syntactically familiar with SQL-like query languages. However, it is necessary for a user to deepen in REST architectural style but not in OpenAPI 3.0. A simple knowledge of API calls is enough to understand and use OpenAPI QL. To show proof of concept, OpenAPI QL system has been developed that supports query translation, execution and results viewing.

# Acknowledgements

I would like to express my very great appreciation to my supervisor Prof. Euripides Petrakis for the continuous support of my thesis despite the pandemic circumstances.

I would also like to give my grateful thanks to Nikos Mainas for his helpful ideas, especially for his understanding.

Besides my advisors, I would like to thank my friends and generally all the people I loved through this wonderful trip. The first person I would like to thank is Ilias for his continuous understanding, and secondly Persefoni, Marilena, Sofia, Alexandra and Konstantina for making my study trip memorable.

Finally, I wish to thank my parents and my sisters Dimitra and Christina, for their love, support and encouragement.

# Contents

Chapter 1

# Introduction

The World Wide Web is realized as a composition of Web services. A Web service is a unit that provides a variety of functions (often referred to as services) which are activated over HTTP. Web services comprise a great tool for the Web developer community. Typically, Web services are described in plain text which users have to browse and read, in order to determine whether a service meets their needs [1]. However, text descriptions are not readable by machines and in some cases are inaccurate or vague. Web services need to be formally described in a way that is understandable by both humans and machines. The last requirement would not only improve the accuracy of service descriptions but also, would allow for services to be discovered by other services and be orchestrated in composite services or applications. A Web service description is a document by which the service provider communicates the specification of the Web service to the service requester. OpenAPI specification (OAS) [2] is a widely adopted standard for describing REST APIs which is supported by large industry users like Google, Microsoft, IBM, Oracle and many others.

## 1.1 Problem Definition

OpenAPI Specification 3.0 defines a standard interface to describe REST APIs. Firstly, Swagger 2.0 and secondly, OpenAPI Specification 3.0, introduced a structure that facilitates the collecting of API information and documenting them in a file [3].

OpenAPI Specification format, is designed to offer both humans and computers the capability to discover Web Services, without access to any documentation or source code. However, the descriptions of Web services are mainly understandable by humans and not absolutely by machines. In order for the services to be able to be searched, discovered and used by other services we need to map OpenAPI

[1]https://idratherbewriting.com/learnapidoc/pubapis_openapi_intro.html
[2]https://swagger.io/specification/
[3]https://medium.com/@tgtshanika/open-api-3-0-vs-swagger-2-0-94a80f121022

descriptions to a hierarchical database model. In this model the OpenAPI fields are organized into a tree-like structure, forming a hierarchy of connected data.

## 1.2 Proposed Solution

In order for a machine to be able to understand the meaning of OpenAPI service description, the service has to be formally described and its content clearly defined. In this thesis, we provide a mechanism for searching and discovering Web Services described using OpenAPI.

OpenAPI QL relies on the idea of searching Web Services, with asking simple API characteristics, such is the data exchange format of a request. The most remarkable spots of this work, are (a) the deep understanding of the Web Services that OpenAPI 3.0 describes, (b) the search mechanism that deals with the levels of nesting in OAS 3.0 structure, (c) the information extraction from properties and objects and (d) the OpenAPI QL syntax and restrictions.

## 1.3 Contributions

The contributions of the present work are summarized below

- We introduce OpenAPI QL, a language for querying OpenAPI.
- We define the OpenAPI QL search model to approach OAS 3.0 descriptions.
- We suggest a translation from OpenAPI QL queries to N1QL, a No-SQL query language which can be applied for searching JSON data.

## 1.4 Thesis Outline

In chapter 2, we present background and information related to our work. Chapter 3 presents the search model we were based on, in order to implement the new query language. In the following chapter, we discuss language syntax and rules. In this chapter, we also explain how OpenAPI QL queries are processed and executed over OpenAPI data. In chapter 5, we show the implementation of the query language analyzing the algorithm. Chapter 6 includes example queries and results collected experimentally using example queries on the database. These Finally, chapter 7 consists of conclusions and some future work ideas.

Summarizing,

**Chapter 2** presents the background.

**Chapter 3** introduces the OpenAPI Model based on OpenAPI description structure.This Chapter is actually an extended diagram representation of our search mechanism.

**Chapter 4** presents to an extended description of OpenAPI QL's syntax and use.

**Chapter 5** describes the OpenAPI QL system.

**Chapter 6** contains example queries and results.

**Chapter 7** proposes ideas for further research.

Chapter 2

# Background

Over the past few years, APIs have been widespread with more and more companies appreciating the business value of creating APIs. The massive growth of Web Services, created the need for clear API documentation. API documentation is a technical document, containing instructions about how to effectively use and integrate an API into an application. In particular, for RESTful services the need for a concise reference document containing all the information required to work with APIs, led to the creation of the Swagger Specification. Swagger contract describes what the API does, it's request parameters and response objects, all without any indication of code implementation.

## 2.1 OpenAPI Specification

OpenAPI specification was created for the description of RESTful services. OpenAPI Specification (OAS) 3.0 renamed from Swagger Specification, is an open-source, language-agnostic specification. When properly defined, the consumer is able to understand and interact with the service with minimal amount of implementation logic. The Service descriptions can be written in either JSON or YAML. The implementation of the APIs may follow a top - down or a bottom - up approach. When the top - down approach is used, the service is implemented after the service's description has been written, while in the bottom - up approach the implementation comes first and the description is later generated by the service's implementation.

For implementation purposes, OpenAPI community offered numerous API management tools. The most popular tool is an editor called Swagger. Swagger Editor provides instant visualization and is able to run locally or online. Thus, a consumer may use the Swaggerhub. The Swaggerhub [1] is the old tool known as Swagger Editor, which has been integrated in order to give the users the capability of team working with OAS. It is a powerful tool which allows teams to fully document RESTful services.

[1]https://stackshare.io/swaggerhub/alternatives

Moreover, OAS provides an open-source code generator - Swagger Codegen 7 - which makes it possible to build server code directly from an OpenAPI service description in almost any programming language and framework (PHP, Java etc). Last but not least, OpenAPI Specification gives the client the chance to visually render documentation for an OpenAPI service description, using the Swagger UI 8, an open - source HTML5 - based user interface. Other competitive tools to Swagger are Postman, Amazon API Gateaway and Paw.

OpenAPI Specification is part of the OpenAPI Initiative, which is supported by widely known companies such as Google, Microsoft and IBM. Its structure will be further described in the following chapters.

Figure 1 [2] illustrates the structure of an OpenAPI document version 3.0:

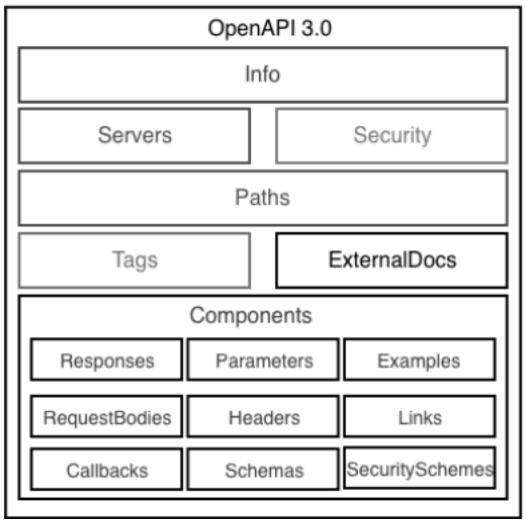


Figure 1. OpenAPI Specification 3.0 structure

## 2.2 OAS2 vs OAS3

The current version of OpenAPI Specification is OAS 3.1.0 and was released in 2020 [3]. It was the first major update of the specification since 2015. OAS 3.0 features a more elaborate (yet simple) structure and format than its predecessor OAS 2.0. The requirement for a single host server is relaxed (allowing a service to be installed on multiple servers). The request body is more flexible and allows consumption of different media types, such as JSON, XML, HTML, plain text and others. The descriptions for

[2] Aikaterini Karavasileiou. “An ontology for describing OpenAPI version 3 services in the cloud”.In: (2019)
[3] https://swagger.io/blog/news/whats-new-in-openapi-3-0/

parameters have changed: FormData parameter was removed and, the cookie parameter type was introduced for documenting APIs that use cookies. The definition of Schema objects is enhanced with additional properties (e.g. anyOf, oneOf, not) allowing a creation of more complex schemas of various data types. Regarding security definitions, OAS v3.0 is enhanced with support for OpenID Connect Discovery 1.0. OAS 3.0 now features a Components field where various reusable objects can be defined (i.e. responses, parameters, headers, links, callbacks, schemas and security schemes).

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

OAS 3.0 has become more simplified. The parameters, responses and securityDefinitions appear now under the components item. New objects have been added within this item. Item definitions in OAS 2.0 has been renamed as schemas in OAS 3.0, while securityDefinitions item has been renamed as securitySchemes, placed under components item. The items produces and consumes have disappeared and absorbed by the paths item. The same happened with the sub-items host, basePath, schemes that have been replaced by the servers item.

## 2.3 REST

REST (Representational State Transfer) is an architectural style for developing Web services [4]. It was introduced in Ray Fielding's dissertation on 2000 and by then it obtained massive adoption. REST defines a set of constraints that need to be used while creating web services in order for the services to be called “RESTful”.

The main key-terms related to REST, are the client and resource [5]. The client is the software uses the API. As resource is actually called any object that the API can provide information about, for example a photograph or a Website. Each resource has a unique identifier, known as URL. Resources

[4] https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/

[5] Aikaterini Karavasileiou. “An ontology for describing OpenAPI version 3 services in the cloud”.In: (2019)

can be static or dynamic. For example, static resource is considered to be an image because it remains the same for each and every request. Dynamic resource can be a Web page that experiences ongoing modifications.

While establishing the communication between a client and a server, it is necessary for the first to provide the latter with an identifier pointing the resource it is interested in. Moreover the client has to inform the service provider about the operation needs to be executed on that specific resource. The operation is actually a verb defined by HTTP protocol. Considering that REST is based on such protocol, these verbs can be the GET, the PUT, the POST and so on.

Apart from single term description, is useful to mention that REST defines a list of constraints that a service should follow in order to become “RESTful”. These constraints are namely the Uniform Interface, the Stateless, the *Cacheable*, the *Client-Server*, the *Layered System* and finally the *Code on Demand*.

## 2.4 REST API meets OpenAPI

REST (REpresentational State Transfer) describes how clients and servers interact with each other. REST communication is based on HTTP protocol. The requests are made through to a resource URI. This URI possibly contains additional data for request execution, and replies in many formats such as JSON.

To access a REST service, the client needs to know information about the REST API. With OpenAPI this step is automated. OpenAPI provides a machine parse-able file that explains computers how a REST API works. It consists of information about what requests exist in this API, and what replies to expect. In other words, this file shows developers the accessible features of an API. Besides the fact that it offers readable documentation, it is capable of generating code in different languages.

Summarizing, REST and OpenAPI have important differences. The first establishes communication rules between client and server and is used from humans to create an API. The second produces a parse-able file by machines, but is also useful for a developer to discover and use API features. So, the only connection between them is that OpenAPI describes APIs that conform to the REST principles and architectural constraints.

Chapter 3

# OpenAPI Query Model

Designing a query system involves making a large number of decisions, further complicated by the difficult structure of an OpenAPI document. In this thesis, we designed a data model in order to be used in information retrieval from OpenAPI documents. Specifying the data elements we desire to query and defining how they relate to one another, led us to the composition of the OpenAPI Query model.

OpenAPI Query model is a simplified representation of the services described in OpenAPI. With the term query model, is meant to be the creation of a query system involving decisions about how OpenAPI elements can be queried efficiently. According to OpenAPI Specification 3.0, a service is described using a combination of objects and properties, in JSON or YAML format. Finding the most appropriate search model for our data, we had to define where these data can be stored and how can be discovered efficiently. For this reason, we created a model based on the classic OpenAPI 3.0 structure with additions or conversions [1]. Hierarchical relationships between objects are now decomposed to simpler ones in order for users to be able to extract information from one statement. Accessing properties from nested OpenAPI objects and setting new values if the requested property don't exist, are the most important features provided by our model.

## 3.1 Services in OpenAPI Specification

The OpenAPI Specification 3.0 is an open-source format for producing, describing, consuming, and visualizing RESTful services in machine-readable files [2]. An OpenAPI description details the actions exposed by a REST API. For example, OpenAPI is capable of representing API endpoints, paths,

[1]https://www.infoworld.com/article/2977045/n1ql-brings-sql-to-nosql-databases.html
[2]https://dbpedia.org/page/OpenAPI_Specification

headers or parameters in a document which are basic components describing a request and the corresponding responses. Moreover, a remarkable improvement from previous versions is the representation of security information, such is authentication and authorization of an API action. So, the latest version of OpenAPI Specification is more powerful when it comes to describing request or response models.

All API service descriptions are defined under Paths Object, within OpenAPI Specification 3.0 files. Paths object contains the components needed to execute a request and study the responses returned by the API. More specific, according to REST, a request is made up of four components such as endpoint, HTTP method, headers and body. In some cases, a request requires security definition for authentication and authorization purposes. The following figure illustrates an example of a Paths Object from Petstore API:

```
{ "paths": {
    "/pet": {
        "put": {
            "tags": [
                "pet"
            ],
            "summary": "Update an existing pet",
            "operationId": "updatePet",
            "requestBody": {
                "$ref": "#/components/requestBodies/Pet"
            },
            "responses": {
                "400": {
                    "description": "Invalid ID supplied"
                },
                "404": {
                    "description": "Pet not found"
                }
            },
            "security": [
                { "petstore_auth": [
                        "write:pets",
                        "read:pets"
                    ]...}
```

Listing 3.1. Paths Object with request body, response and authentication scheme definition

Ignoring the hierarchical structure of OpenAPI documents, OpenAPI model offers a new perspective on information extraction. Objects like “Info” and “Servers”, are objects representing general information and servers, respectively. These objects are worth to be displayed in results, but not to be used as conditions in WHERE clause. The reason for this is that their property values are generalized API information which do not actually describe API services. So, evaluating the information provided by OpenAPI Specification 3.0, led us to the properties that are worth to be searched, composing the OpenAPI Query model. In this section we will show which these objects are. After providing the description, it would be convenient to explain how the OpenAPI fields are converted in order to serve our querying purposes. All these changes are presented further in the chapter.

### 3.1.1 Request

A request in RESTful APIs, generally involves:

- ***HTTP Method***: defines what kind of operation to perform
- ***Request Media Type***: declares the format of the request data
- ***Request Body***: contains the representation of the resource to be created
- ***Request Parameters***: describes the additional information sent to the server. A parameter is basically a part of request URL. The possible locations of a parameter are ”query”, ”header”, ”path” or ”cookie”.
- ***Request Security Requirement***: defines the security scheme for authentication and authorization of an individual request. The available security schemes are the HTTP, the OAuth2, the OpenIdConnect Discovery and Api Keys.
- ***Tags***: declares the names of the groups that a particular operation is assigned to. Operation Tags are not directly associated with request execution. They are actually labels categorizing an operation so that the tools and libraries can handle these actions differently. So, tagged operations provide tag names that are useful to describe fully a request.

### 3.1.2 Response

A response in RESTful APIs, is composed of:

- ***HTTP status code***: defines a number which characterizes the status of a particular response. Every operation returns one successful status code and one or more error statuses.
- ***Response Media Types***: declares the format of the response data
- ***Response body***: contains the representation of the already created response. A response body is described by a particular schema.

- ***Response Headers***: defines the custom headers provided by an API call

## 3.2 Dealing with objects and properties

The OpenAPI objects and properties describing a service are presented in hierarchical JSON documents. Each API document includes information, spread among objects, making the data processing a difficult procedure. Moreover, the relationships between objects and their deeply nested structure considered to be the most serious problems to deal with, when implementing a search mechanism. For this reason, we suggest the flattening of nested JSON objects until the target properties to be approached as individual elements. With the term flattening we mean the defining of simple expressions that help us to query OpenAPI properties, ignoring the level of nesting in JSON objects. Even though the desirable information is located deep in a JSON data tree, the flattening technique allows us to define expressions which approach hierarchical structures as flat data.

*Flattening* for nested JSON, suggests that the objects and their nested values can be represented in a single level. In our case, we use this technique to approach OpenAPI objects. Each element in the JSON file considered as an *OpenAPI Element*. Each *OpenAPI Element* can have hierarchical object structure. The goal is to convert the these elements into a flat list of fields and values. However, it is not desirable to flatten all OpenAPI elements. This approach requires only the objects describing a request execution with the corresponding results to be unpacked. More specific the OpenAPI objects transformed, are the basic objects defined under Paths Object. Operation Object, Parameter Object, Tag Object, Header Object, Request Body Object, Response Object, Responses Object, Media Type Object, Schema Object, Security Requirement Object and Reference Object.

### 3.2.1 Dealing with Operation Object

Operation Object describes a single API operation with a verb. This verb is placed as a field name, nested inside Paths Object. Typically, this kind of operation definition can be queried only by the use of access paths. Access paths are actually expressions which have the capability to locate a property names within objects and extract information from them. In this case, the access path for a GET operation is [paths[< any endpoint >].get]. However, this method does not serve the purposes of our search model. For this reason, we convert this field into a key/value pair in which the key is "method" and its value is "get" as shown below:

$$\text{"get": \{ "description", "responses", e.t.c. \}} \quad \rightarrow \quad \textbf{method}\text{: "get"}$$

After that, a service operation can be queried by **method** property values. The content of the property **method** considered to be a unique distinct string with values one of "get", "put", "post", "delete", "patch", "trace", "options" and "head".

### 3.2.2 Dealing with Parameters

Every request definition in OpenAPI, includes one or more parameter definitions. As a result, parameters in OpenAPI requests can be an extended list of Parameter Objects. Parameter Object defines a single operation parameter. Each parameter within Parameter Object is characterized uniquely by a combination of name and location. Field “name” is responsible for parameter’s name definition and the field “in” for location. The possible parameter locations are query, path, cookie and header. The following table illustrates a parameter with name “skip” sent in a query string:

"parameters": { "skipParam": { "name": "skip", "in": "query" }

The OpenAPI model, represents the parameters with four *OpenAPI elements.* The elements declared in our model are the *QueryParam*, *PathParam*, *CookieParam* and *Header.* Firstly, a *QueryParam* is actually a Parameter Object in which the value of the field ***in*** is "query". Secondly, a *PathParam* is a Parameter Object in which the value of the field ***in*** is "path". Next, a *CookieParam* has the same representation with the above parameters, except the location which is "cookie". The header parameter is merged with the response headers in the Header element described further in Headers subsection.

Apart from location, each parameter definition contains three properties. More specific, the properties defined are “name”, “schema” and “required”. The “required” property is the field that determines whether this parameter is mandatory. The “name” and “schema” properties contain the name and the schema of the parameter, respectively. In our model, the elements considered as parameter properties, are the exact same field/value pairs. This means that the properties that can be queried by their values are the ***name***, the ***schema*** and the ***required***, as the following example demonstrates.

Field "name" can be queried by value, which is a case sensitive string. Next, the field "required" has as value the boolean strings such is true and false. As schema considered to be a specific object which

"parameters":
"name": "skip" → ***name***: name of the parameter,
"required": true → ***required***: true or false,
"schema":"type": "string" → ***schema***: body of the parameter

contains a list of properties. The "schema" field is a special case that is not completely covered in this section. In particular, the OpenAPI QL model takes into consideration the case of the property "properties" appearing within a Schema Object.

### 3.2.3 Dealing with Tags

OpenAPI Specification 3.0 uses tags to group operations either by resources or any other qualifier. Although tags are optional in OpenAPI, applying them on large APIs can help users to organize services by endpoint. Considering that APIs operate through request and responses, the tag shows the location where the API sends a request and where the response emanates. For this reason, tag is a valuable addition to our model when comes to describe an operation.

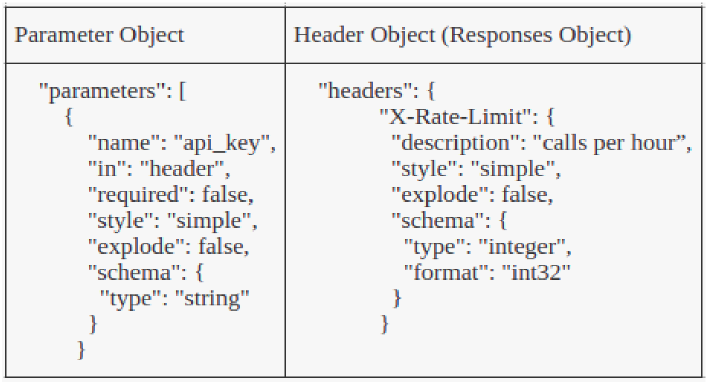
A tag is a grouping name applied on a particular operation. An operation can have more than one tags. At the root level, the Tags Object lists all the tags of an operation within Paths Object. Every Tag Object includes some fields, the most important of them to be the field "name". Hence, we consider tag as a property that characterizes an operation, but also through operation, characterizes a request. As a result, in our model the field "tag" is defined as a Request property, as the following example illustrates.

"tags": [ "pet", "user" ] → ***tag***: name of the tag

Field "tag" can be queried by its value. The name of the tag is a case sensitive string such is "pet" and "user".

### 3.2.4 Dealing with Headers

OpenAPI responses can include custom headers to provide additional information on the result of an API call. In OpenAPI, the information referring to headers, are presented by Header Object, within Response Object. Also, headers can appear as operation parameters in the same way with the parameters described in a previous subsection.



| Parameter Object | Header Object (Responses Object) |
|---|---|
| ```"parameters": [ { "name": "api_key", "in": "header", "required": false, "style": "simple", "explode": false, "schema": { "type": "string" } }``` | ```"headers": { "X-Rate-Limit": { "description": "calls per hour", "style": "simple", "explode": false, "schema": { "type": "integer", "format": "int32" } }``` |

Figure 2. Header representation within Paths and Responses Objects

Both definitions have name and body. However, the header structure presented in responses it is not in the form of a key/value pair, which would be convenient for our model, because of the name of the header is defined as a JSON key. Therefore, we add a new property "name" which will have the name of the header as string value.

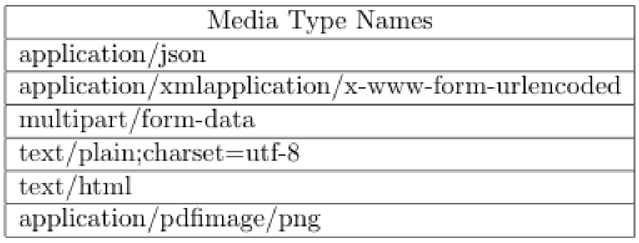
1. "X-Rate-Limit": { "description", "schema" } $\rightarrow$ ***name***: name of the header
2. "limitHeader": { "name": "limit", "in": "header" } $\rightarrow$ ***name***: name of the header

Field "name" can be queried by value, which is a string including special characters, such is "X-Rate-limit" and "limit".

### 3.2.5 Dealing with Media Type Object

API service operations can accept and return data in different formats, such as JSON, XML or images. In OpenAPI these formats are defined in Media Type Object within a request or a response, one or multiple times. Each Media Type Object is identified by a key. Within a Media Type Object are declared a body and a optionally a list of media type examples. Possible media type names are listed below:

Although media type is defined clearly, the actual value needed to be searched is the key itself, which is the "application/json" presented in the figure below. So, we choose to flatten this field and convert it into a key/value pair to serve the querying purposes. The new pair has the "media_type" as key and the name of the media type as value:



| Media Type Names |
|---|
| application/json |
| application/xmlapplication/x-www-form-urlencoded |
| multipart/form-data |
| text/plain;charset=utf-8 |
| text/html |
| application/pdfimage/png |

Figure 3. Media type formats that OpenAPI 3.0 supports

"responses": { "application/json": { "schema": { } } $\rightarrow$ "***media_type***": "application/json"

Field "media_type" can be queried by value, which is a case sensitive string.

### 3.2.6 Dealing with HTTP status codes

Web service operations typically return status codes in order to provide successful or unsuccessful statuses. Every status code defines a range of status codes using the following notation: 1XX, 2XX, 3XX, 4XX, and 5XX. Nevertheless, OpenAPI Specification 3.0 supports all possible HTTP response codes, such are the successful and error statuses. For example, a 404 Not Found response for an operation that returns a resource by ID, is a status code that appears in almost all responses in an API documentation.

A status code is the number that defines operation status. In OpenAPI descriptions this number is a JSON key. This means that the results can be fetched by querying the name of the key which is a specific number. This is not a convenient search method for our model, because the user is not able to know the exact value of the status code of a response. In order to simplify the search process, we suggest key-value pair conversion to the field responsible for status code representation. More specific, we introduce the key ***status_code*** that holds the status code representation, a shown below:

"responses": { "404": { "application/json": { "schema" } } } $\rightarrow$ "***status_code***": 404

Considering this property as a number, we proposed a query model that examines whether a status code is within a range. For example, if a user wants to check if an operation fails due to an internal server error, he would probably end up querying random status code values. This happens because the range definition of this kind of errors is [500 - 599] and the user is not able to know the exact status code in advance. So, modeling properties like status code, offers to users search flexibility. This idea would be explained in detail in a next chapter.

### 3.2.7 Dealing with Responses

API responses are specified in every API documentation. Each operation must have at least one defined response, most of the times a successful one. As about response contents, a response is described using its HTTP status code and the data returned in the response body and/or headers.

More specifically, every response starts with HTTP status code definition. As already mentioned in a previous section, a status code in our model is declared as a key-value pair which consists of a key named ***status_code*** and a numeric value. Next, after status code, follows the media type for data exchange, under *content* keyword. In OpenAPI Query model a media type is represented by a key-value pair, in which the key is ***media_type*** and the value is a case sensitive string. As for the response body, it is described using *schema* keyword in various ways. A response schema can be an object, an array, a primitive or a file. However, OPENAPI model does not support all schema cases, but only one case, related to *properties* keyword. More specific, our model illustrates schema definitions adding a key labeled ***property***. With this key, a user is able to query a schema that contains a specific property definition.

### 3.2.8 Dealing with Security Requirement Object

Security, in OpenAPI, is declared by the term *security scheme*. This term defines the required authentication and authorization schemes for an operation execution. APIs are protected using four security schemes, such are OAuth 2.0, OpenID Connect Discovery, Api Keys and HTTP. Each security defined under operations, must correspond to a security scheme which is declared in the Security Schemes under the Components Object. Apart from security definition, composing queries for dealing with API security information, is a challenging part of our work.

Our model proposes a security scheme to be queried by type, where each scheme can be of type “oauth2”, “openIdConnect”, “apikey” and “http”. Each type of security provides its own properties, but not in Paths section. Therefore, our model is designed to identify the authentication required for request authorization inside Paths Object and then deal with four different cases within Components Object.

The first security scheme is *OAuth 2.0* which is declared by OAuth Flow Object. This object contains mandatory properties defined by keys *flows* and *scopes*. *OAuth 2.0* flows are scenarios an API client performs to get an access token from the authorization server. Scopes are access rights that control whether the credentials a user provides allow to perform the needed call to the resource server. So, in order to be able to query these information, we embody in our model these scenarios as *OAuth 2.0* properties named ***flow*** and ***scope***. If an operation authentication scheme is of type “oauth2” then the value of a flow is a string with values one of ”authorizationCode”, ”implicit”, ”password” and ”clientCredentials”. The value of a scope is a string defining an action, like “read” or “write”.

1. ”flows”: { ”implicit” } → ***flow***: name of OAuth 2.0 flow
2. ”scopes”: [ ”write”, ”read” ] → ***scope***: name of scope

Next, *OpenID Connect Discovery* security scheme is on the top of *OAuth 2.0* authentication protocol and based on its providers. More specifically, *OpenID Connect* security declares a sign-in flow that enables a client application to obtain user information via authentication. If the security scheme is of type “openIdConnect”, then value is a list of scope names required for the execution. Additional information extracted from this scheme is the *openIdConnectUrl* property. Summarizing, this type of security can be queried by ***scope*** and ***openIdConnectUrl*** property values, which are properties established by OpenAPI Query model. Our model supports as scope values, stings defining an action. The values of the property ***openIdConnectUrl*** are also strings in Url format.

1. ”openIdConnectUrl”: ”< Url string >” → ***openIdConnectUrl***: Url of the scheme
2. ”scopes”: [ ”write”, ”read” ] → ***scope***: name of the scope

Another important authorization method is the use of API Keys. An API key is actually a token that a client provides when making API calls. In OpenAPI 3.0, API Keys are described as a combination of name and location, defined as *name* and *in* properties, respectively. In our implementation, we use the exact same fields for these properties to be searched for. The value of key ***name*** is a case sensitive string and the value of ***in*** defining the location, is a string with value one of “header”, “query” and “cookie”. The example follows, shows the field conversions applied on API Key Authentication:

1. ”name”: ”apikey_auth” → ***name***: name of the security scheme
2. ”in”: ”header” → ***in***: location of the API Key

The final security scheme is HTTP. OpenAPI 3.0 includes numeric cases of security definitions built in HTTP protocol, such as bearer authentication. In Paths Object, HTTP security for a request to be authorized, is declared as an empty array list. Each HTTP scheme defines the property "type" with value *http* and *scheme* with values like "bearer", "basic" or another arbitrary string. Using the exact same field names, our technique embodies in our model's structure, the property ***scheme***, in order for users to be able to query its values.

"scheme": "basic" $\rightarrow$ ***scheme***: name of the HTTP Authorization scheme

### 3.2.9 Dealing with Request Bodies

Request bodies specify the message that will be sent in a request. Therefore, the operations that usually define a request body are PUT, POST and PATCH. n OpenAPI, a request body is provided by *requestBody* keyword and its contents are media types and schemas. Request media types are defined under *content* keyword, the most common of them being JSON, XML, form data, plain text and some more presented in 3.2.5 sub-section. Also, OpenAPI Query model supports a specific schema case which includes the properties object defined by *properties* keyword. More specific, this object is a collection of property/value pairs that can be modeled in the same way by declaring the property ***property***. The following example shows the presence of a media type and a schema within a request body in OpenAPI descriptions:

```
{  "requestBody": {
        "content": {
            "application/x-www-form-urlencoded": {
                "schema": {
                   "properties": {
                      "id": {
                        "type": "integer",
                        "format": "int64"

                   }
                 }
               }
```

Listing 3.2. Schema with property definition in a Request Body Object

The information we desire to be extracted from a request body is the media type name and the name of the property within a schema. So when comes to query a request body, we search for the key/value pairs "media_type": string and "property": string, as discussed in sections 3.2.5 and 3.3.1 respectively.

## 3.3 Dealing with special cases

JSON uses data objects consisting of attribute–value pairs and array data types in order to describe API Services, according to OpenAPI Specification 3.0. Even though JSON gives the capability of representing API data in a document, the final structure produced by OpenAPI Specification 3.0 is complex. The main problems we were called to deal with when trying to implement OpenAPI Query model, are analyzed further in the following subsections.

### 3.3.1 Dealing with Schema Object

Schema object is responsible for defining input and output data types. According to OpenAPI Specification 3.0, a schema can be an object, a primitive like sting or number, an array, a file a combination of properties and objects specifying special data types such as form data. Additionally, a schema may contain keywords like *empty* for no body definitions and *default* for common structured schemas. Also, a schema object can have several alternative definitions that can be described by *oneof* and *anyof* keywords. Although OpenAPI supports polymorphism, our query model cannot support these cases.

Therefore, considering that the implementation of a model describing all possible schemas is not a simple procedure, we chose to embody in our model only the case described by "properties" keyword. The "properties" key is used to define the object properties. It actually lists the property names and specifies a schema for each one of them. In order to query these names, we establish a new field named ***property***, as shown below:

"properties": { "id": { "type": "integer" } } $\rightarrow$ "***property***": "id"

The field ***property*** has case sensitive string values such is the "id".

## 3.4 Modeling OpenAPI Services

The diagram following, represents all the parts of the OpenAPI description we considered as *OpenAPI Elements* and the hierarchical relations between them. Although this work is about searching for services in OpenAPI Specification descriptions, our approach is based mostly on REST architectural style and how API actions are described through this. Considering that a RESTful action is actually a request made to the server through an HTTP method, we adapted the OpenAPI 3.0 Model according to our search needs. The OpenAPI Query Model defines what is an OpenAPI service and how it is described as a request with one or more responses. Thus, the following representation of OpenAPI Query Model also simplifies the hierarchical relations between objects and properties as the following picture illustrates.

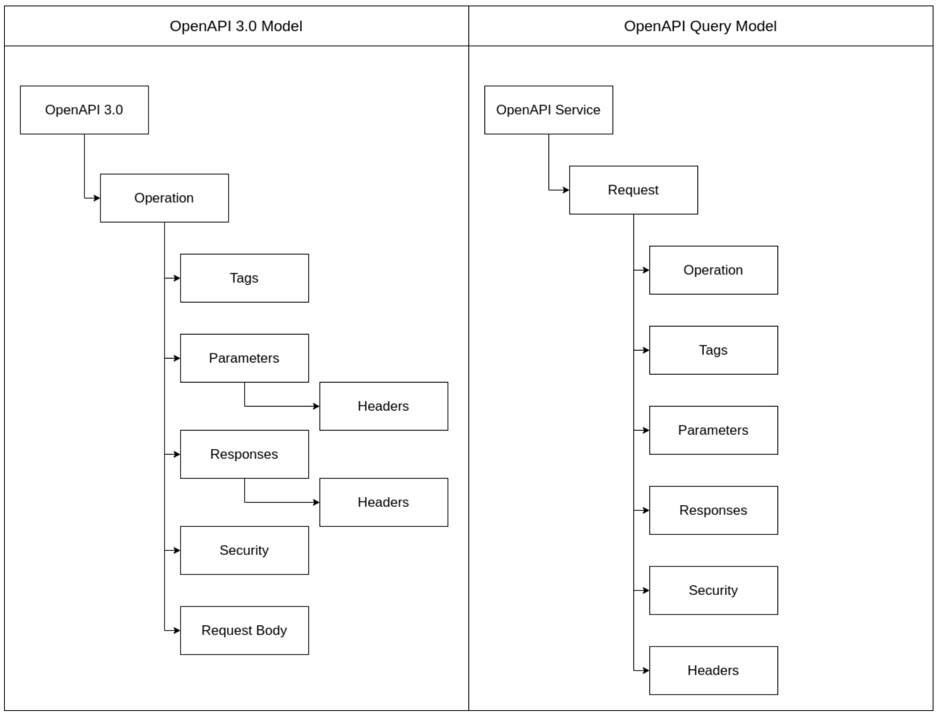


Figure 4. From OpenAPI 3.0 Model to OpenAPI Query Model

At the left side of the diagram, the OpenAPI 3.0 Model is described, which is based on OpenAPI 3.0 Specification. The hierarchy of the JSON objects presented in an OpenAPI document, starts with the Operation Object. Within Operation Object, OpenAPI defines Tags, Responses, Parameters, Security and Request Bodies as the next level elements. Also, headers are described as a part of a Response or Parameter Object. The hierarchical relationships between these objects do not allow the user to locate information without the knowledge of the information of a previous level. In particular, if a user desires to search for a specific parameter, he needs to know the operation verb (GET, PUT, POST, PATCH, DELETE, OPTIONS, HEAD, TRACE), in order to extract this information from specific Operation Objects. Thus, we proposed a query model that allows users to search for any OpenAPI property ignoring the level of nesting. This approach aims to flatten hierarchical relationships between objects by considering all the above information as parts of a Request.

A request, a response, a parameter, a security scheme, a header or a tag are elements that describe a RESTful action. Each action provided by an API is actually a request made to the server from the client. So, the right part of the above diagram treats all the above elements as components of a Request. Every component has it's own characteristics that describe it thoroughly, for example a name, or a schema. Finally, one or more requests considered to be components of a Web Service. In this way, we achieve to define a model that helps user who hasn't deepen in OpenAPI 3.0, to search for specific service characteristics.

### 3.4.1 OpenAPI Query Model Synopsis

Service is the top level element and has to be defined in every OPENAPI QL query. The reason for this is that every SELECT statement fetches the whole OpenAPI documents - document id and content- that matches the conditions. Next, the Request belongs to the second level of hierarchy because every API action is related to it. More specifically, REST APIs work by requesting information from a server and then receiving the responses. OpenAPI security schemes are used to authenticate requests in RESTful APIs. Parameters are passed through APIs when a request executes. Headers are used to an HTTP request and responses is information returned after a complete request execution. Hence, each element describing a request parameter, response, header or authentication scheme is considered to be components of a Request.

Except from OpenAPI elements hierarchical representation, we need to summarize the fields defined for each *OpenAPI Element* separately. Each one element in the diagram 4 is a reserved word for our model. Each element defines a list of field-value pairs which are described thoroughly in chapter 3 and summarized in table 1. The left column of the table below contains all the elements and fields extracted from OpenAPI Query model. The expressions in the form OpenAPI_Element.[field], use the dot notation to define the *OpenAPI Element* to be queried, followed by the search fields. The column appearing at the right of the table, shows the OpenAPI objects and properties our elements correspond to. This table is valuable because OPENAPI QL uses this notation to express queries.

| OPENAPI QL Expression Syntax | OpenAPI Object and Property Reference |
|---|---|
| Service | API's description. The root object. |
| Request | Parts of Operation Object. More specific, RequestBody Object, Tag Object, Security Object and Parameter Object. |
| Request.method | We refer to properties with particular method names like 'get', 'put', 'post', 'delete', 'patch', 'options', 'head','trace'. |
| Request.tag | We refer to the property 'name' of every tag in Tags Object's list. |
| Request.media_type (requestBody) | Property name that is the media type used for the request body. Media type object starts with the definition of media type as property name, for example 'application\json', 'application/xml' and it refers to the current request or response body. |
| Request.schema | Schema Object that is included in Request Body Object. |
| Response | Responses Object (including Response Object which describes a single response) |
| Response.status_code (responses) | HTTP status code property name that is described in every response. In Responses Object status code appears once for every response. |
| Response.media_type (responses) | Property name that is the media type used for the response body.Media type object starts with the definition of media type as property name, for example 'application\json', 'application/xml' and it refers to the current request or response body. |
| Response.schema | Schema Object that is included in Response Object. |

| OPENAPI QL Expression Syntax | OpenAPI Object and Property Reference |
|---|---|
| QueryParam | Parameter Object where property ‘in‘ has value ‘query‘. In fact we refer to a query type parameter. |
| QueryParam.name | Value of property ‘name‘ in Parameter’s Object, where ‘in‘ property has value ‘query‘. |
| QueryParam.schema | Schema Object that is included in Parameter’s Object, where ‘in‘ property has value ‘query‘. |
| PathParam | Parameter Object where property ‘in‘ has value ‘path‘. In fact we refer to a path type parameter. |
| PathParam.name | Value of property ‘name‘ in Parameter’s Object, where ‘in‘ property has value ‘path‘. |
| PathParam.schema | Schema Object that is included in Parameter’s Object, where ‘in‘ property has value ‘query‘. |
| CookieParam | Parameter Object where property ‘in‘ has value ‘cookie‘. In fact we refer to a parameter passed as cookie. |
| CookieParam.name | Value of property ‘name‘ in Parameter’s Object, where ‘in‘ property has value ‘cookie‘. |
| CookieParam schema | Schema Object that is in Parameter’s Object, where ‘in‘ property has value ‘cookie‘. |
| Schema | Schema Object as described in requestBody or responses.Schema property or Content property contains the data we ask for. |

| OPENAPI QL Expression Syntax | OpenAPI Object and Property Reference |
|---|---|
| Oauth2Auth | Security Scheme Object whose property ‘type‘ has value ‘oauth2‘. |
| Oauth2Auth.flow | OAuth Flow Object starts with property name ‘implicit‘, ‘password‘, ‘clientCredentials‘, ‘authorizationCode‘. Oauth2Auth.flow in OpenAPI QL refers to this property name with values ‘implicit‘, ‘password‘, ‘clientCredentials‘ or ‘authorizationCode‘. |
| Oauth2Auth.scope | OAuth Flow Object scope property names like ‘read‘ or ‘write‘.OAuth Flow Object describes scopes with key-value pairs like ‘read:pets‘:‘<sort description>‘, but we need only the information about the type of scope, like ‘read‘ or ‘write‘. |
| OpenIdConAuth | Security Scheme Object whose property ‘type‘ has value ‘openIdConnect‘. |
| OpenIdConAuth.openIdConnectUrl | Security Scheme Object whose property ‘openIdConnectUrl‘ has a particular Url string value. |
| ApiKeyAuth | Security Scheme Object whose property ‘type‘ has value ‘apiKey‘. |
| ApiKeyAuth.name | Value of property ‘name‘ which is in Security Scheme Object and property ‘type‘ has value ‘apiKey‘. |
| ApiKeyAuth.in | Value of property ‘in‘ which is in Security Scheme Object and property ‘type‘ has value ‘apiKey‘. |
| HttpAuth | Security Scheme Object whose property ‘type‘ has value ‘http‘. |

| OPENAPI QL Expression Syntax | OpenAPI Object and Property Reference |
|---|---|
| HttpAuth.scheme | Value of property ‘scheme‘ which is in Security Scheme Object and property ‘type‘ has value ‘http‘. |
| Header | Header Object. Mapping both Response and Request headers, the common elements between them, are the name of the header and it’s schema. Parameter Object where property ‘in‘ has value ‘header‘. It is actually header information passed with the request. |
| Header.name | Property name that gives the exact name of the header in Header Object. For example Header Object starts with ‘X-Rate-Limit‘ property which is the header’s name we ask for.<br>OR<br>Value of property ‘name‘ in Parameter’s Object, where ‘in‘ property has value ‘header‘. |
| Header.schema | Schema Object which is described in Header Object.<br>OR<br>Schema Object that is in Parameter’s Object, where ‘in‘ property has value ‘header‘. |

Table 1. OpenAPI query model Synopsis

Chapter 4

# OPENAPI Query Language

In this chapter we introduce OPENAPI QL, a query language for searching API Services. The new language is based on a powerful set of properties and operators introduced by OpenAPI Query model. The advantages of this query model are two: (a) it is capable of querying properties that describe the actions provided by APIs and (b) defines new properties that are in the form of key/value pairs. Moreover, the OPENAPI QL suggests a syntax which follows the basic SQL syntax (SELECT - FROM - WHERE) and treats properties and objects, most like columns in tables. The working version of OPENAPI QL is presented in the system implementation discussed further in a following chapter. Notice that, OPENAPI QL queries are translated into equivalent N1QL queries.

## 4.1 Syntax

OpenAPI QL statements present similarities with SQL statements. The syntax includes the three basic clauses, SELECT, FROM and WHERE, and some common operators such as BETWEEN, AND, OR. OPENAPI QL also supports properties referring to each element we need to search for, and *OpenAPI Elements* declaring the data to query from.

Every OPENAPI QL query, starts with SELECT clause. Expressions following SELECT clause, define the data to be shown as results. Retrieved data, in our case, is ment to be the id or the content of the OpenAPI description file.

As mentioned in a previous chapter, our data is actually a collection of OpenAPI descriptions. To fufill the purposes of our search model we considered every OpenAPI description as a composition of eleven elements which are the Request, Response, QueryParam, Header, PathParam, CookieParam, OAuth2Auth, OpenIdConnectAuth, ApiKeyAuth, HTTPAuth and Tag. Each element defined in OpenAPI Query model, specifies a basic API functionality, such is a request or a response. In our language we refer to them as *OpenAPI Elements* and they are following FROM clause, the second clause defined after SELECT statement.

An OpenAPI element consists of a number of customized properties. OPENAPI QL first determines the components of a request which can be a response, a parameter, a header or a security scheme element. Secondly, defines a new property for each component separately, in the form of a key/value pair. Since the properties are defined clearly, the user is able to use them in order to compose OpenAPI QL expressions, using between them the appropriate conjunctive or disjunctive operator. As a result, WHERE clause is able to illustrate a combination of expressions and make comparisons, in order to retrieve the results from database.

## 4.2 Clauses and Operators

The clauses that OpenAPI QL supports, are described below:

- ***SELECT***: specifies the data to be returned. In our case the returned data can be the document id, the entire OpenAPI document or both.

- ***FROM***: declares the data elements to query from. Always follows SELECT clause.

- ***WHERE***: specifies logic operations between properties. WHERE clause always follows FROM clause.

OPENAPI QL supports the following operators:

- ***AS***: renames the data parts of a query. Aliasing a property, changes the name performed in the results. AS operator, connects properties with aliases in SELECT clause. In case of FROM clause, this operator allows using an instance of a data part in a query.

- ***AND***: connects logically two expressions. These expressions are connected by logical AND functionality in WHERE clause. Both expressions need to be true.

- ***OR***: connects logically two expressions. These expressions are connected by logical OR functionality in WHERE clause. At least one expression needs to be true.

- ***BETWEEN***: searches criteria for a query where the value is inside the range of two values. These values can be numbers. BETWEEN operator is used only when ”status_code” property expressions, within WHERE clause.

Also, OPENAPI QL declares a comparison operator:

- ***EQUALSIGN*** or =: introduces the meaning of “equal to”. It is used to WHERE clause expressions as a comparison operator between a field and a value.

- ***WILDCARD*** or ∗: represents one or more characters. In SELECT clause it is used to return all fetched results from a data source.

Summarizing OPENAPI QL syntax:

| Syntax |
|---|
| **SELECT** < projected property > AS < aliasing string > |
| **FROM** < OpenAPI Element > AS < aliasing string > |
| **WHERE** < condition > AND / OR < condition > BETWEEN ”integer” AND ”integer” |

Table 2. Generic OPENAPI QL syntax

The table above presents the generic syntax of OPENAPI QL. The projected properties are the properties returned as query results. Next, the OpenAPI Elements are the data parts of the OpenAPI description, declared in our model. Aliasing strings are used after properties and elements giving to them temporary names.

This language supports operators allowing users to specify more than one criteria after WHERE clause. The term *criteria* implies basically, to field/value pairs. A special case is the BETWEEN operator within WHERE statement. This operator is used to define a range of numeric values. So, it appears in OPENAPI QL statements only when the property has numeric values. For example, the field *status_code* can be queried either by a specific value or by a given range of values including BETWEEN operator. Both representations are illustrated in the examples below:

(A)

```
SELECT ... AS ...
FROM ... AS ...
WHERE status_code = {numeric value}
```

(B)

```
SELECT ... AS ...
FROM ... AS ...
WHERE status_code
BETWEEN
{numeric value} AND {numeric value}
```

Table 3. Querying status code

More specifically, the case (A) illustrates an expression that selects all records where the value of the status_code property is a specific number. The case (B) asks to be returned the data that have status codes in a given range.

## 4.3 Query Composition

OPENAPI QL is a high level language independent from the actual representation of objects and properties in OpenAPI Specification 3.0. OpenAPI elements are transformed and implemented in a simple and understandable way according to users needs. A user has only to be familiar with basic meanings of REST and SQL-like syntax.

As mentioned in a previous section, OPENAPI QL queries are a composition of SELECT, FROM and WHERE expressions. SQL deals with data in tables. On the other hand, OPENAPI QL handles data as JSON-formed documents. Considering these files as our data source, OPENAPI QL is capable of running queries over specific parts of the documents, called *OpenAPI Elements*. As a result, both languages have the basic SELECT-FROM-WHERE structure for their queries, but the main difference between them is the data selection. A more detailed description of the expressions appearing within OPENAPI QL clauses will be discussed further in this chapter.

### 4.3.1 SELECT clause

The standard SELECT statement is "SELECT * FROM OpenAPI Elements". More specific, SELECT clause defines the data to be returned as results. These results are actually a number of OpenAPI service descriptions matching the conditions defined by WHERE clause. In order for a description to be queried, SELECT clause specifies a list of the OpenAPI fields, separated with commas. The SELECT cases that OPENAPI QL supports, are shown below:

| **Case 1** |
|---|
| SELECT s.id (AS) service_id<br>FROM Service (AS) s |
| **Case 2** |
| SELECT s . *<br>FROM Service (AS) s |
| **Case 3** |
| SELECT s . *, s.id (AS) service_id<br>FROM Service (AS) s |

Table 4. SELECT clause cases

Case (1) shows a SELECT statement asking for the id of the services to be returned as results. The "s" alias can be any case sensitive letter or string. Any alias defined in FROM clause can be followed by the id field or the wildcard (*) because in this work we aim to define queries able to retrieve the id or the content of an OpenAPI description. Finally, case (2) seeks for the description content and case (3) for both. Summarizing, the SELECT statement in OpenAPI QL returns the list of the descriptions that match the conditions in WHERE clause and not specific OpenAPI elements. The only choice that the user can make, is to return the content or the id key of the of the OpenAPI file.

### 4.3.2 FROM clause

The FROM clause declares the data to query from. In our case, these data sources are specific objects and properties we consider as *OpenAPI Elements*. Each *OpenAPI Element* provides properties that characterize a request and the corresponding responses. The connection between these elements is hierarchical, as the figure 4 depicts. We consider as top-level *OpenAPI Element* the *Service*, which is every action provided by APIs. The section following in a lower level of hierarchy, is the *Request*. As *Request*, is regarded to be the data element specifying API information, such are parameters, headers, tags, responses and security scheme definitions. For this reason, is required to be defined before any *OpenAPI Element* instance related to it in a lower layer, as diagram 4 shows. For example, a *Response* definition in a query requires for *Service* and *Request* to be declared before:

| Example 4.3.2 | |
|---|---|
| SELECT | s.id (AS) id |
| FROM | *Service* (AS) s, |
| | *Request* (AS) req, |
| | *Response* (AS) resp |
| WHERE | s.request = req |
| | AND req.response = resp |
| | AND resp.status_code BETWEEN 200 AND 400 |

Table 5. Querying request and response properties

In the table above, the query defines the data in which to search for, in hierarchical order. When a user queries a response property has to define first the Service, which is the element to be returned as result. Secondly, OPENAPI QL requires for the *Request Element* to be defined in order to get access to a response, a parameter, a header or a security scheme. The hierarchy followed in FROM clause elements, is the hierarchy that OpenAPI defines, in a more simplified way. So, one or a combination of these API elements have to be defined after *Request Element* as the diagram 4 depicts. Finally, the condition within WHERE clause asks for the status_code property that belongs to a *Response Element* and filters the data in a range of numbers.

### 4.3.3 WHERE clause

The WHERE clause is used to filter records using conditions described thoroughly in 3.2.1-3.2.8 sections. Each condition contains expressions required to be fulfilled for a query to return the results. A specific condition consists of a property name, a property value and an operator such is equal sign or BETWEEN, combining those elements. Apart from these elements, a condition begins with OpenAPI Element aliases and dot notation. With the term alias we mean the temporary name assigned to every OpenAPI Element appearing in FROM expressions. As an example:

| Example 4.3.3 | |
|---|---|
| SELECT | s.id (AS) id |
| FROM | *Service* s |
| | *Request* req |
| WHERE | s.request = req |
| | AND req.media_type = "application_json" |

Table 6. Querying for media type of a request

The query above, is One Condition query and asks for a Request with a specific media type. If the condition in WHERE clause is satisfied, then the query fetches only the necessary OpenAPI definitions. OPENAPI QL also supports more than one conditions. In fact, WHERE clause combines conditions using logical operators such is AND and OR. The following example uses the AND condition in the WHERE clause to specify multiple conditions that must be met for the record to be selected.

| Example 4.3.4 | |
|---|---|
| SELECT | s.id (AS) id |
| FROM | *Service* (AS) s |
| | *Request* (AS) req |
| | *OAuth2Auth* (AS) oa2 |
| WHERE | s.request = req |
| | AND req.media_type = "application_json" |
| | AND req.OAuth2Auth = oa2 |
| | AND oa2.flow = "implicit" |

Table 7. Querying for media type and security scheme of a request

Chapter 5

# Implementation and Algorithm

The OpenAPI QL system translates a custom SQL-like language to a No-SQL query language, thereby enabling the existing client-side application code to run against database. The main purpose of the OpenAPI QL system is to express and execute efficiently OpenAPI QL queries. The tasks performed by each module individually, compose a system in which a user can search a service by querying simple properties. The input is an OpenAPI QL query and the output is the returned services from the database.

The system uses Couchbase which is a database able to store JSON documents [1] in varied schemas. An additional reason to choose Couchbase, is because the JSON files stored can contain nested structures. Also, N1QL which is Couchbase's query language [2] , is able to query, transform and manipulate unstructured or semi-structured JSON data. For all the above reasons and considering the fact that OpenAPI descriptions have difficult structure, Couchbase is the most suitable database that meets the demands of this work.

## 5.1 Description of OpenAPI QL System

As proof of concept, we implemented a simple API whose purpose is to facilitate expressing and executing of queries in OpenAPI QL. The system consists of a query translator that converts OpenAPI QL queries to N1QL query language and a NoSQL database containing OpenAPI services. Other tools used for data exploration is an interface for Couchbase. The following figure illustrates the OpenAPI QL system:

[1] https://docs.couchbase.com/server/current/introduction/why-couchbase.html
[2] https://www.couchbase.com/products/n1ql

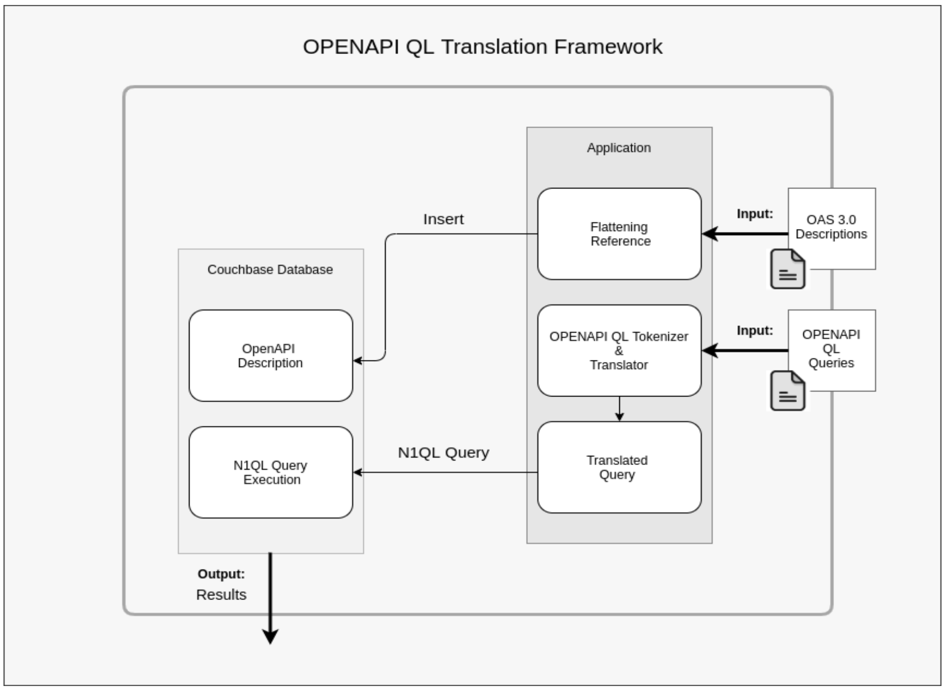


Figure 5. The OpenAPI QL System

The input of the system can be a query or an OpenAPI definition. The user uses the API to insert a definition. Then, the *Flattening Ref* module parses the definition in order to replace the ***$ref*** key with the respective OpenAPI objects or elements. If the OpenAPI file is successfully inserted into Couchbase, then the user can execute OpenAPI QL queries over the OpenAPI services.

After the definition is loaded in Couchbase, the user expresses queries in OpenAPI QL and gives them as input in the translator module. Then, the query is tokenized and translated to N1QL, which is the language manipulating Couchbase's data. When the translation process is completed, the translated N1QL query is automatically executed. The results are displayed by the API.

More specifically:

(1) The query translator module is implemented in Python. This module supports query translation and execution. More specific, it is used to convert an OpenAPI QL query into tokens. Tokens are the basic syntactical units of OpenAPI QL. A token in this work is a sequence of one or more characters that cannot contain blanks as explained further in 5.3

section. After that it is used to translate sequences of tokens into N1QL expressions. These expressions are combined to give the the translated query in N1QL.

(2) The *Flattening Ref* module is actually a function that is used while inserting the OpenAPI services in database. The basic role of this function is the replacement of all **$ref** keys across definitions. However, the size of the resulting object after flattening JSON schemas, is bigger because of the repeated blocks of code.

(3) The OpenAPI QL system consists of a NoSQL database called Couchbase. Couchbase is a distributed database which in our case, holds numerous service descriptions gathered in a bucket named *OASBucket*. Justifying the choice of Couchbase, it offers a flexible JSON schema for OpenAPI data and indexing options which are critical for OpenAPI QL query performance. As for data manipulation, it provides a SQL-like query language called N1QL.

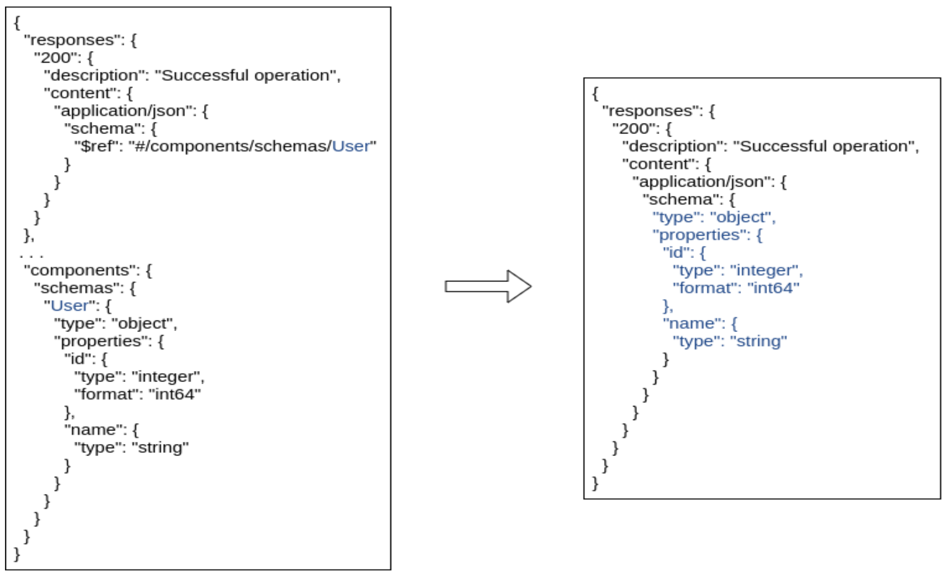
(4) The Couchbase Python module is the connector between the custom made python translator and Couchbase. It is used for data accessing. In fact, OpenAPI QL system needed this module in order to execute OpenAPI QL queries and flatten the references among objects.

(5) The Query Workbench is an integrated visual interface that is used for uploading OpenAPI files in Couchbase and collecting results. Each OpenAPI service is inserted into OASBucket via Workbench.

## 5.2 Flattening Reference function

The Reference Object is a simple object that allows referencing other components in the specification. The **$ref** key is declared in every Reference Object and is used to refer to repeated OpenAPI elements across descriptions. Using **$ref**, OpenAPI elements can specify the part of the file they are related to, which most of the times is located in Components Object.

When querying a schema, the **$ref** definition do not serve our purposes, since is needed to be queried in two phases. In order to retrieve results, one should ask for a specific schema and then query on the returned **$ref** value, which is a path pointing the actual object.Therefore, OpenAPI Query model suggests a flattening of references by replacing reusable code within *Paths* schemas, as shown in the following example:

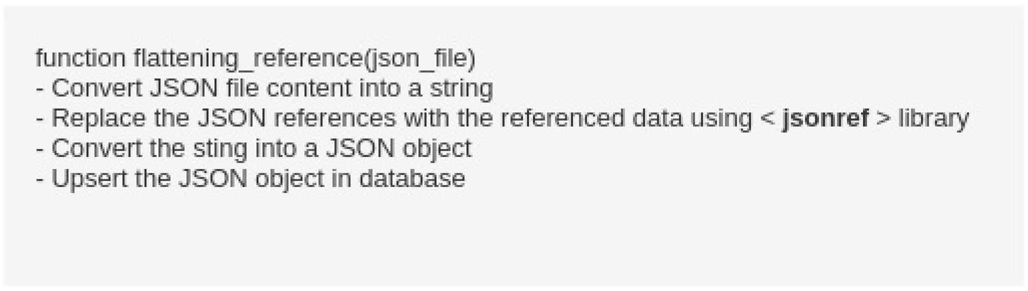


```
{
  "responses": {
    "200": {
      "description": "Successful operation",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/User"
          }
        }
      }
    }
  },
 . . .
  "components": {
    "schemas": {
      "User": {
        "type": "object",
        "properties": {
          "id": {
            "type": "integer",
            "format": "int64"
          },
          "name": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

```
{
  "responses": {
    "200": {
      "description": "Successful operation",
      "content": {
        "application/json": {
          "schema": {
            "type": "object",
            "properties": {
              "id": {
                "type": "integer",
                "format": "int64"
              },
              "name": {
                "type": "string"
              }
            }
          }
        }
      }
    }
  }
}
```

Figure 6. Hierarchical representation of a request

The *Flattening Ref* module is a function responsible for replacing JSON references across definitions. The argument of the function is the JSON file containing the OpenAPI description. First, we convert the JSON object (json_file argument) into a string in order to be processed by **jsonref** Python library. Next, the **jsonref** module is called which automatically flattens the JSON reference objects within the data structure. As a last step, the string produced is converted into a JSON object again, in order to be inserted in Couchbase. The figure below sums up the functionality of flattening_reference function.



```
function flattening_reference(json_file)
- Convert JSON file content into a string
- Replace the JSON references with the referenced data using < jsonref > library
- Convert the sting into a JSON object
- Upsert the JSON object in database
```

Figure 7. Function for JSON $ref replacement

This mechanism allows very simple and efficient traversal of OpenAPI document content. Nevertheless, the flattening method, duplicates that part of the schema everywhere across definitions, making updating or other action in the future more difficult.

## 5.3 Tokenization Algorithm

When trying to develop a text search mechanism, the first concern is how to index data. The implemented system in this thesis, is a system based on tokens. With the term token we mean an OpenAPI QL unit. A token in OpenAPI QL, consists of one or more characters, excluding blanks, control characters, and characters within a string constant or delimited identifier [3]. So, as a token considered to be every character, case sensitive string or reserved word of our query language, such is *SELECT*, *FROM* or a *Request* element e.t.c. For example the tokens of an SELECT statement are illustrated below:

| | | | | | | |
|---|---|---|---|---|---|---|
| SELECT | s.id | AS | service_id | FROM | Service | s |

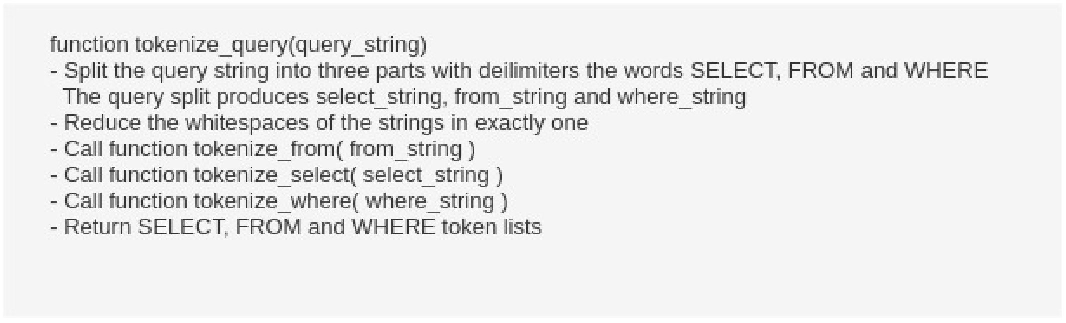
Table 8. SELECT statement tokenization

The tokenizer in our system, is enabled automatically when a query is set for execution. Therefore, when a user enters an OpenAPI QL query, the query is tokenized, and a bucket of documents is accessed in order for the best matches to be found.

The most important factor when tokenizing a query, is how users want to retrieve the results. An additional factor is the structure of the data. OpenAPI content appear to be an unstructured JSON file, but it is just structured data using a specific language model. In our case, we apply a simple word split on OpenAPI QL expressions, including case insensitivity utilities.

Considering the SQL-like syntax of our queries, our first concern is to choose the suitable split for the sequence of words. The function *tokenize_query* is responsible for the query tokenization. The OpenAPI QL query is treated as a string in order to be tokenized. The first separation is applied on *SELECT*, *FROM* and *WHERE* words, as the figure 8 illustrates. After, that the original query is divided into three clear sub-expressions, such are from_string, select_string and where_string. In each individual sub-expression we reduce the number of white spaces between words, in exactly one, in order to be able to split the strings in whitespaces. This is a necessary process in order to be able to cleave the sequence of characters to pieces.

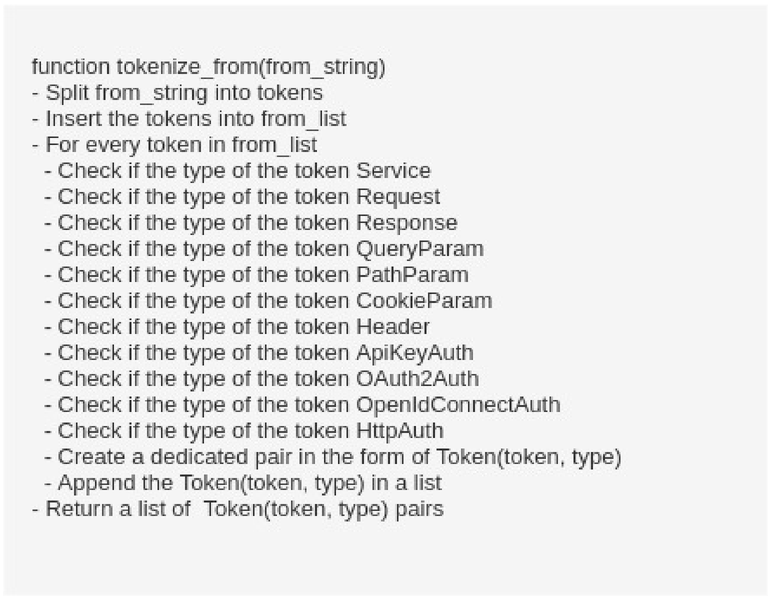
[3] https://www.ibm.com/docs/en/db2-warehouse?topic=elements-tokens



```
function tokenize_query(query_string)
- Split the query string into three parts with deilimiters the words SELECT, FROM and WHERE
  The query split produces select_string, from_string and where_string
- Reduce the whitespaces of the strings in exactly one
- Call function tokenize_from( from_string )
- Call function tokenize_select( select_string )
- Call function tokenize_where( where_string )
- Return SELECT, FROM and WHERE token lists
```

Figure 8. Function for query tokenization

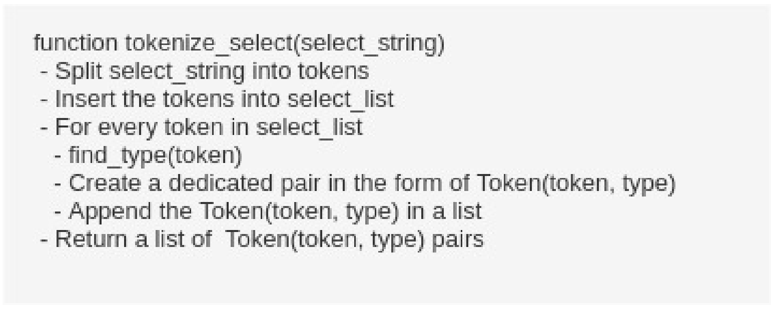
Next, every token is characterized by a specific type. Each type conforms to an OpenAPI QL element, such is a logical operator, a string value, a numeric value, a symbol, or just a *Request* element, *Response* element and so on. The type application on tokens, begins from the sub-query following the *FROM* clause, which is actually the data we want to retrieve the results from. The reason for this is that, the *OpenAPI Elements* the query searches into, has to be determined first in order to be able to apply types on SELECT and FROM tokens. The tokens extracted from the FROM sub-expression, are tokens of type Service, Request, Response and generally all the types that conform to *OpenAPI Elements*. The function responsible for converting the FROM content is the *tokenize_from*, as explained in the figure 9.



```
function tokenize_from(from_string)
- Split from_string into tokens
- Insert the tokens into from_list
- For every token in from_list
  - Check if the type of the token Service
  - Check if the type of the token Request
  - Check if the type of the token Response
  - Check if the type of the token QueryParam
  - Check if the type of the token PathParam
  - Check if the type of the token CookieParam
  - Check if the type of the token Header
  - Check if the type of the token ApiKeyAuth
  - Check if the type of the token OAuth2Auth
  - Check if the type of the token OpenIdConnectAuth
  - Check if the type of the token HttpAuth
  - Create a dedicated pair in the form of Token(token, type)
  - Append the Token(token, type) in a list
- Return a list of  Token(token, type) pairs
```

Figure 9. Function for FROM content tokenization

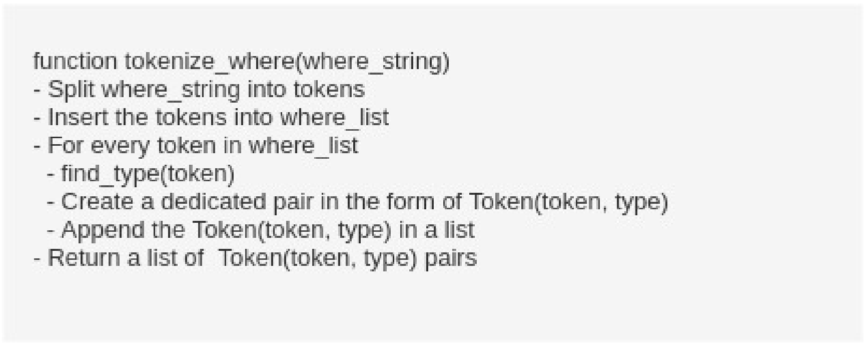
First, the *tokenize_from* function splits the FROM content (from_string argument) multiple times in order to convert it into tokens. After splitting process, the tokens produced are inserted in a list. The line following, loops over *from_list* in order to determine the type of the token. Then, the tokens characterized by type, are inserted in a list in the form of Token(token, type) pairs. Once the *from_list* is returned, the algorithm processes the SELECT and FROM sub-expressions.



```
function tokenize_select(select_string)
 - Split select_string into tokens
 - Insert the tokens into select_list
 - For every token in select_list
   - find_type(token)
   - Create a dedicated pair in the form of Token(token, type)
   - Append the Token(token, type) in a list
 - Return a list of  Token(token, type) pairs
```

Figure 10. Function for SELECT content tokenization

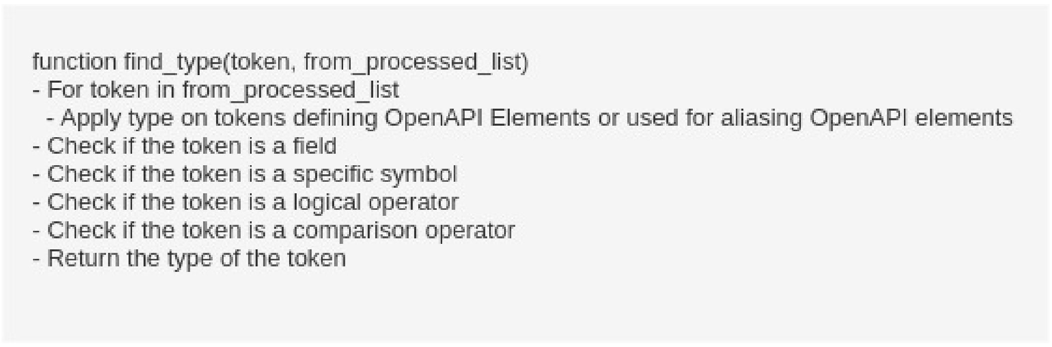
The function in the above figure 10, explains the SELECT content tokenization procedure. First, the *select_string* is split on whitespaces and then, the tokens are inserted in a list. For each token in the list, the function *find_token* is called. This function is responsible for applying a type on the token, as explained in the figure 12. After that, the token and the type applied are represented by a pair like Token(AND, LOGICAL_AND). The pairs are inserted in a list, which is the list returned by the function. The same steps are followed when splitting the WHERE content, as presented below:



```
function tokenize_where(where_string)
- Split where_string into tokens
- Insert the tokens into where_list
- For every token in where_list
  - find_type(token)
  - Create a dedicated pair in the form of Token(token, type)
  - Append the Token(token, type) in a list
- Return a list of  Token(token, type) pairs
```

Figure 11. Function for WHERE content tokenization

The *find_type* function characterizes a token with a type. First, checks if the tokens presented in SELECT and WHERE clauses exist in from_processed_list. This list contains the Token(token, type) pairs after converting the FROM content. The rest of the tokens can be a field, a symbol, a logical operator or a comparison operator. Finally, the function returns the type of the token, as explained in the figure 12:



```
function find_type(token, from_processed_list)
- For token in from_processed_list
  - Apply type on tokens defining OpenAPI Elements or used for aliasing OpenAPI elements
- Check if the token is a field
- Check if the token is a specific symbol
- Check if the token is a logical operator
- Check if the token is a comparison operator
- Return the type of the token
```

Figure 12. Function for finding the type of the token

Summarizing, this splitting strategy facilitates the type application procedure on SELECT and WHERE tokens. So, it is important a token/type distinction for the next step of interpretation process which is actually the query translation.

### 5.3.1 Dealing with case sensitivity

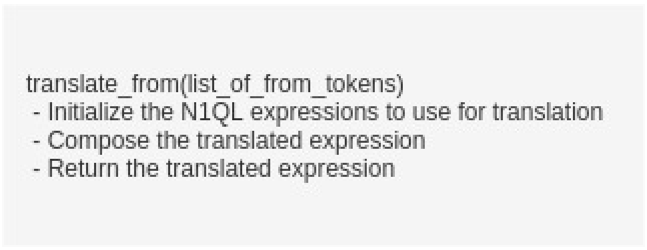
When composing a query, the main clauses and the operators are case insensitive. This means that the OpenAPI QL system recognizes either as an upper case, a lower case or a mixed case string for this type of tokens. However, when querying a property value, the input retains as it is entered by the user. For example, the name of a parameter can include mixed case letters. This name must correspond to the parameter name passed through an API. For this reason, it is more desirable to consider it as a non-case sensitive term.

## 5.4 Translation Algorithm

Finding the most appropriate translation of the OpenAPI QL queries, was been a difficult problem to solve. To deal with it, we propose a strategy that recognizes sequences of tokens. The implemented translator takes as input the first token of every sub-query. If a token with a specific type is found, then it moves to the next token. When the algorithm detects a specific sequence of tokens in the current list, converts it into a valid N1QL expression. The table 23 summarizes all possible translations in N1QL.

The sub-query containing the *OpenAPI Elements* to extract the information from, is translated first. The reason for that is that the input query has to know where to search for. Although, the defined *OpenAPI Elements* in a FROM expression can be numerous, in N1QL, these elements correspond to the bucket containing the OpenAPI services. The translation following is the SELECT sub-query, which is a simple process as long as the user can query OpenAPI definitions in two ways, either by id or content. Finally, the last conversion from OpenAPI QL to N1QL is the one referring to conditions. This part of the translator implementation is the most challenging part of our work.

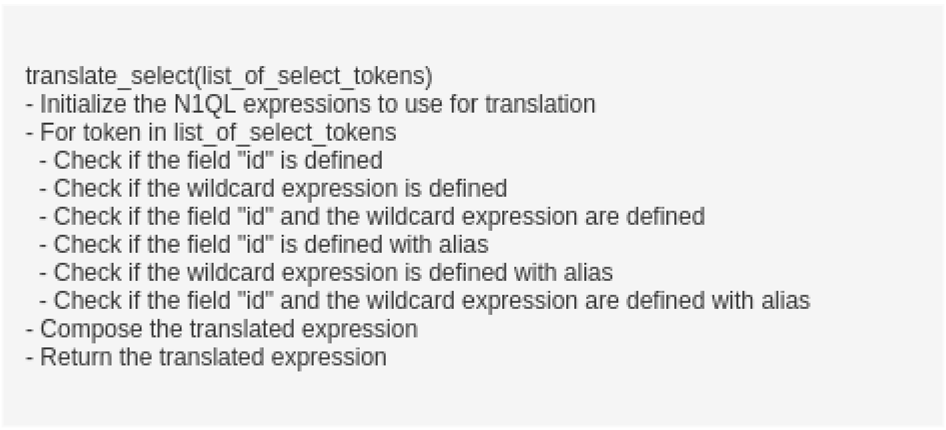
The function responsible for translating the OpenAPI QL *FROM* content, is depicted in figure 13.



```
translate_from(list_of_from_tokens)
 - Initialize the N1QL expressions to use for translation
 - Compose the translated expression
 - Return the translated expression
```

Figure 13. Function for translating FROM sub-expression

First, the strings containing the N1QL expressions are initialized. The action following is the composition of the final FROM expression in N1QL. Finally, the translated expression is returned. The translation following, is the one that refers to SELECT content, as the picture 14.

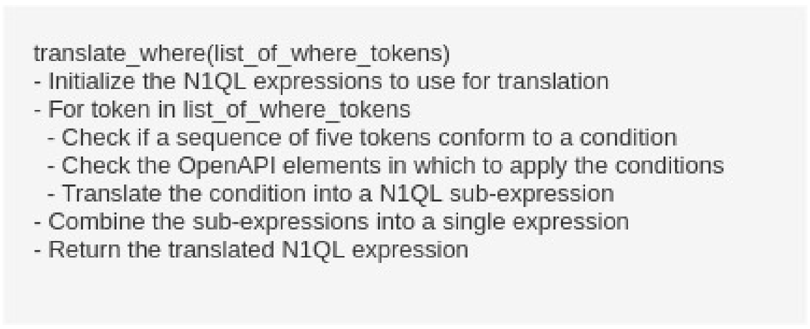


```
translate_select(list_of_select_tokens)
- Initialize the N1QL expressions to use for translation
- For token in list_of_select_tokens
  - Check if the field "id" is defined
  - Check if the wildcard expression is defined
  - Check if the field "id" and the wildcard expression are defined
  - Check if the field "id" is defined with alias
  - Check if the wildcard expression is defined with alias
  - Check if the field "id" and the wildcard expression are defined with alias
- Compose the translated expression
- Return the translated expression
```

Figure 14. Function for translating FROM sub-expression

The *SELECT* clause content is automatically translated after FROM sub-expression. The function translate_select is responsible for converting sequence of tokens into valid N1QL expressions. First, the expressions used for translating the sequences are defined. Next, each token is checked in order to be determined in which SELECT case corresponds, as the table 4 shows. Then, each sequence recognized by the algorithm, is converted in a N1QL sub-expression. Finally, the sub-expressions are combined into a string, which is returned from the function.

To explain further the mapping between the two languages, we need to define the data that OpenAPI QL requests to be returned as results within SELECT clause. OpenAPI QL supports the filtering of the results set, either by id, either by content or both. These fields correspond to OpenAPI fields within JSON objects and metadata files. More specific, the OpenAPI QL ***id*** field is mapped to the ***id*** metadata field in N1QL. It is can be queried by the use of N1QL's META() function. As about querying by content, the use of the wildcard character (*) is the answer for both languages. In OpenAPI QL, the wildcard selects all services in a database. In the same way N1QL selects all the OpenAPI documents considered as services, in a bucket. The table 23 summarizes the translation of both cases. The final translation is applied on WHERE clause content by the *translate_where*, which is explained below.



```
translate_where(list_of_where_tokens)
- Initialize the N1QL expressions to use for translation
- For token in list_of_where_tokens
  - Check if a sequence of five tokens conform to a condition
  - Check the OpenAPI elements in which to apply the conditions
  - Translate the condition into a N1QL sub-expression
- Combine the sub-expressions into a single expression
- Return the translated N1QL expression
```

Figure 15. Function for translation of the WHERE sub-expression

In case of conditions within a WHERE statement, an OpenAPI QL query defines multiple expressions combined with conjunctive operators. The translator module detects the conditions enabling the WHERE clause translation, by calling the *translate_where* function. The translation starts with initializing the N1QL expressions. Next, each individual condition is categorized into a group. Each group is the OpenAPI Element the condition refers to. After identifying the elements to be queried, the following action is the mapping between OpenAPI QL objects or properties to OpenAPI elements, as the table 1 shows. Then, the condition is converted into the corresponding N1QL expression. Each translated involves *ANY* predicate that tests the existence of these elements within definitions. If the  predicate expression is true, then the condition is tested over OpenAPI services. The final query produced by our mechanism is a composition of these sub-expressions in N1QL. Finally, the complete WHERE clause expression is returned by the function.

## 5.5 Equivalence to N1QL

Syntactically, OpenAPI QL expressions are similar to SQL expressions with the most important difference between them to be the data selection. To show proof of concept, we translate OpenAPI queries into a comprehensive and declarative query language such is N1QL, which is widely known as the *SQL for JSON*. The translation strategy we followed starts from the expressions defined in FROM clause.

OpenAPI QL queries are based on matching documents from a database, according clauses (SELECT-FROM-WHERE) that specify criteria. The mapping strategy followed in our implementation is divided into three stages. More specifically, the query mapping between OpenAPI QL and N1QL is based on the translation of FROM, SELECT and WHERE expressions individually. Thus, the first translation is applied on the elements that OpenAPI QL searches into, which are actually the FROM clause content. The second one is about mapping the fields within SELECT clause, and the last one refers to the condition conversion. After processing the each sub-expression, the three translated parts are combined in a single N1QL expression. In the following will show the equivalence between OpenAPI QL and N1QL expressions focusing on conditions because are the most elaborate part of translation.

**Translation of elements following FROM clause:** No-SQL databases such is Couchbase, stores JSON data in documents. These documents are parts of a collection belonging to a specific bucket. This means that N1QL's FROM clause declares the bucket or buckets of data in which to search. On the other hand, OpenAPI QL is a language for querying specific parts of a description. Hence, considering that OpenAPI QL searches over OpenAPI Elements and N1QL queries over buckets of data, the mapping between the two languages is the following:

So, independent from how many elements we want to be queried, the translation of the expressions following the FROM clause, is always the same and corresponds to the bucket containing the OpenAPI descriptions. In our case, the descriptions are stored in OASBucket.

**Translation of expressions following SELECT clause:** In OpenAPI QL, SELECT clause consists of expressions that can be easily translated to N1QL query. The "id" field appearing in OpenAPI QL

| Clauses | OpenAPI QL | N1QL |
|---|---|---|
| SELECT | . . . AS . . . | . . . AS . . . |
| FROM | OpenAPI_Element_1 AS s1,<br>OpenAPI_Element_2 AS s2,<br>OpenAPI_Element_3 AS s3, . . . | 'OASBucket m' |

Table 9. Query Mapping 1

expressions, contains the id key of the services retrieved after the query executes. In the same way, the N1QL's "id" field holds the id key of the documents returned as results. Thus, the "id" field is directly translated, as presented in the following table:

| Clauses | OpenAPI QL | N1QL |
|---|---|---|
| SELECT | Service.id AS id | 'META(OASBucket).id' AS id |
|  | Service . * | OASBucket . * |

Table 10. Query Mapping 2

As about querying the content of a service, the wildcard (*) operator has common use in both languages. In OpenAPI QL the wildcard returns all the records from the data source. This is mapped to N1QL's wildcard expression which retrieves all the services from OASBucket.

**Translation of conditions:** As we already mentioned, both languages specify conditions in WHERE clause. Each OpenAPI QL condition is translated into a N1QL expression containing *ANY* predicate. These expressions test a boolean condition over the elements or attributes of an object, or objects. In case of deeply nested elements and embedded arrays, these expressions can be defined nested to each other in order for the query to get access to the target data at any depth in the document. The basic syntax of a N1QL expression including *ANY* predicate, is the following:

| ANY Syntax |
|---|
| ANY var1 ( IN — WITHIN ) expr1<br>[ , var2 ( IN — WITHIN ) expr2 ]*<br>SATISFIES condition END |

Table 11. Basic ANY predicate Syntax

The table 11 shows the basic syntax of the translated condition in N1QL. The var(i) are actually testing variables over records and the expr(i) are strings representing the objects or arrays that *ANY* predicate expressions search into. In the following we will discuss a translation example.

**Example 1:** The following query asks for a Request executed via GET HTTP method with a specific body format:

| Clauses | OpenAPI QL | N1QL |
|---|---|---|
| SELECT | s.id as sid , s.* | 'META(m).id' AS id , m . * |
| FROM | Service s, Request r | 'OASBucket' AS m |
| WHERE | s.request = r | ANY v WITHIN<br>OBJECT_VALUES(m.'paths')[*]<br>< **optional_method_declaration** ><br>SATISFIES<br>OBJECT_VALUES(m.'paths')[*]<br>IS NOT MISSING<br>AND<br>< **nested_expression_1** > END |
| | AND | <This operator does not connect<br>actual conditions in a N1QL query> |
| | r.method = "get" | **optional_method_declaration** : "get" |
| | AND | AND |
| | r.media_type = "application/json" | **nested_expression_1** :<br>ANY y IN<br>OBJECT_NAMES(v.'requestBody'.'content')<br>SATISFIES<br>y = "application/json" END |

Table 12. Query Mapping 1

The table 12 shows the mapping between OpenAPI QL and N1QL expressions. The condition in the form of [OpenAPI_Element_1_Alias(.)OpenAPI_Element_2 = OpenAPI_Element_2_Alias] expresses the relation between two OpenAPI elements. Thus, the expression < s.request = req > expresses the relation between a *Service* and a *Request*. These relations has to be defined before any element definition from a lower level of hierarchy, as the diagram 4 illustrates. Moreover, this expression is translated into the equivalent expressions in N1QL using *ANY* predicate. N1QL expresses the same condition with the help of JSON paths. In fact, N1QL determines the path to the *Request* definition using dot (.) notation. Considering that OpenAPI defines a request within Paths Object, the path to this object is [ OASBucket.'paths'[*] ]. After that, N1QL includes this path in a boolean expression in order to get access to this object and search for other properties.

The condition < r.method = ”get” >, tests if the request method is GET. Considering that OpenAPI defines the HTTP method with a verb, N1QL has to locate the ”get” method declaration within Paths Object. For this reason, N1QL uses the path [ OASBucket.’paths’.[ * ].’get’ ] declared in an *ANY* predicate expression. In order to avoid looping over all possible operations within paths, and then looping again over GET request definitions, we choose to add the ”get” verb in the existing expression which searches for Paths Object. As a result, the following expression filters data for both, a *Request* and a GET HTTP method:

```
ANY v WITHIN
OBJECT_VALUES(m.‘paths‘)[*]. ’get’
SATISFIES
OBJECT_VALUES(m.‘paths‘)[*]. ’get’ IS NOT MISSING END
```

The condition < r.media_type = ”application/json” > is also translated into an expression using ANY. N1QL specifies the path to the media type of a request. Then, the translated expression in N1QL, is an *ANY* predicate expression including the path to the media type. The difference is that this expression refers to a field, nested in Operation Object. This means that the media type of a request is declared under method definition, according to OpenAPI. Hence, the translated condition relating to media type, has to be embedded within the translated expression relating to method. With combining the translated conditions nested to each other, we avoid false result matching among other request definitions in the same OpenAPI file.

**Example 2:** The following query asks for a Request with a specific authentication scheme:

| Clauses | OpenAPI QL | N1QL |
|---|---|---|
| SELECT | s.id as sid , s.* | 'META(m).id' AS id , m . * |
| FROM | Service s, Request r, OAuth2Auth oa | 'OASBucket' AS m<br>LEFT OUTER UNNEST<br>OBJECT_NAMES(m.components.securitySchemes)<br>AS security_names |
| WHERE | s.request = r | ANY v WITHIN<br>OBJECT_VALUES(m.'paths')[*]<br>< **optional_method_declaration** ><br>SATISFIES<br>OBJECT_VALUES(m.'paths')[*]<br>IS NOT MISSING<br>AND<br>< **nested_expression_1** > END |
|  | AND | <This operator does not connect<br>actual conditions in a N1QL query> |
|  | req.oauth2auth = oa | **nested_expression_1** :<br>ANY x IN v.'security'<br>SATISFIES<br>OBJECT_NAMES(x)<br>AND ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names<br>AND<br>(ANY c WITHIN m.'components'<br>SATISFIES<br>c.[y].'type' = "oauth2"<br>AND<br>< **nested_expression_2** > END ) END |
|  | AND | AND |
|  | oa.flow = "implicit" | **nested_expression_2** :<br>ANY p IN OBJECT_NAMES(c.[y].'flows')<br>SATISFIES<br>p = "implicit"<br>END |

Table 13. Query Mapping 2

The table 13 shows how a query searching for a request authentication scheme, is expressed in N1QL. The first condition defining the relation between a Service and a Request, is mapped to N1QL's expression querying over Paths Object, as discussed in the example 12. Next, the < req.oauth2auth = oa > condition checks if OAuth 2.0 security is the authentication scheme for a Request. However, the translation in N1QL is a rather complicated expression because the APIs' security schemes are defined within Paths and Components Object. Thus, N1QL has to check if a specific security name exists in both locations. The following sub-expression holds the security names defined within Paths Object:

```
ANY x IN v.‘security‘
SATISFIES
OBJECT_NAMES(x)
AND
(ANY y IN OBJECT_NAMES(x) SATISFIES y = security names END)
END
```

Then, N1QL has to check if the possible security names match any OAuth 2.0 security declaration within Components Object. The expression following tests if a security name defined in Paths Object, matches a security scheme of type "oauth2" within Components Object:

```
ANY c WITHIN m.‘components‘
SATISFIES
c.[y].‘type‘ = "oauth2" END
```

The next expression searches for an OAuth 2.0 scope with value "implicit". The conversion in N1QL is simple as long as the previous expression confirmed the existence of OAuth 2.0 security scheme in a Request. N1QL accesses this scheme within Components Object using the following path:

```
[ 'OASBucket'.'components'.'securitySchemes'. <name of OAuth 2.0 security scheme >.'flows' ]
```

After that, N1QL uses this path into an *ANY* range expression which tests if the OAuth 2.0 Flow Object includes the field "implicit".

Chapter 6

# Examples and Results

To discover services in a database filled with OpenAPI definitions, we might use a system capable of querying simple properties such is the method indicating a request. In this thesis, we present a query language that addresses this challenge by exploiting relations between OAS objects and properties. Therefore, we declare OpenAPI elements and a set of rules for our language in order to filter the OAS definitions over one or multiple conditions. The OpenAPI QL conditions are combined with ***AND*** and ***OR*** operators. The mentioned conjunctive operators provide multiple comparisons in order for the desired data to be retrieved.

In this section we present OPENAPI QL expressions executed over 30MB of data. The data in our case is a bucket containing 100 OAS services. The services consist of an average of 300 lines of JSON. Each service can be inserted to the system via a simple user interface.

## 6.1 OpenAPI QL Web Application

The OpenAPI QL API is a very simple user interface. Our goal was to create a system consisting of the basic mechanisms reaching the main purposes of our work. The system first of all provides a basic login mechanism so that the user can access the application as figure 16 illustrates. After the success login, the user has to options, either to upload a JSON description on Couchbase database, either to execute query on the existing descriptions in the same database. A user may upload an OpenAPI description in JSON format with a specific filename. As about query execution, the user environment includes a query box where the user can insert an OpenAPI QL query and then execute it. When the query mechanism retrieves the data from the database properly, the user is able to see the results in the results box.

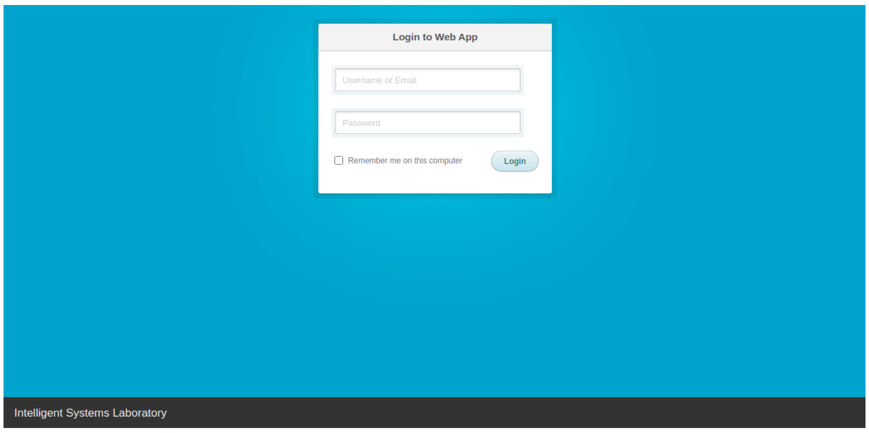


Figure 16. User Interface Login

After the success login, the user has to options, either to upload a JSON description on Couchbase database, either to execute query on the existing descriptions in the same database. The two mechanisms are presented on a user menu as 17 depicts.

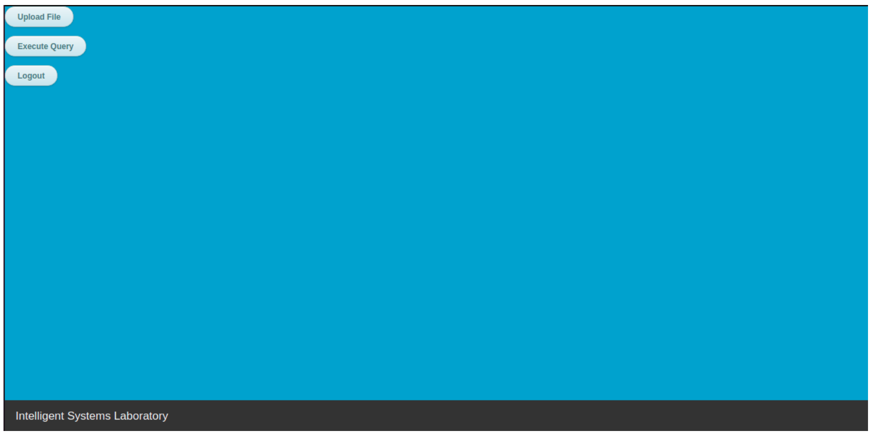
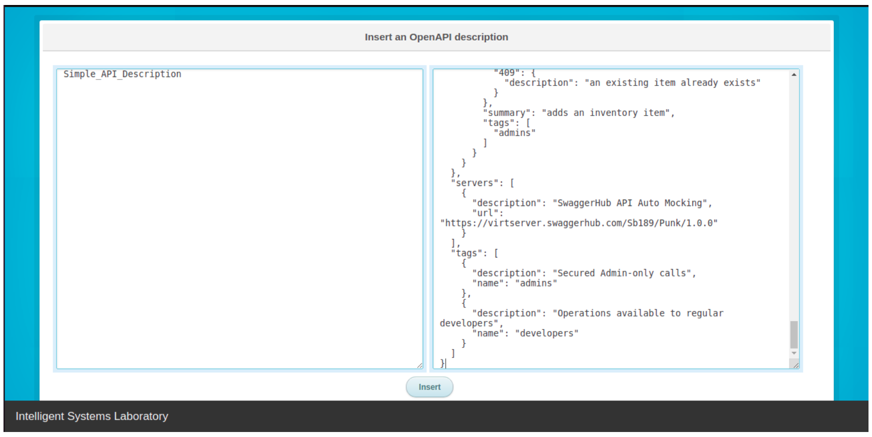


Figure 17. Selection Menu

As the menu presents, the user has the opportunity to insert a description in the database as well as to perform OpenAPI QL queries. If the user makes a choice from the menu, then the system automatically redirects to the uploading page or querying page, as the figures and show respectively.

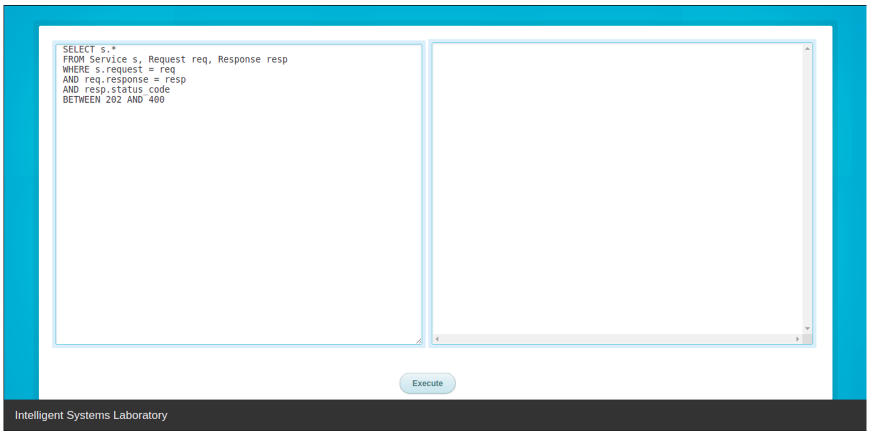


```
Simple_API_Description
```

```
            "409": {
              "description": "an existing item already exists"
            }
          },
          "summary": "adds an inventory item",
          "tags": [
            "admins"
          ]
        }
      }
    },
    "servers": [
      {
        "description": "SwaggerHub API Auto Mocking",
        "url":
"https://virtserver.swaggerhub.com/Sb189/Punk/1.0.0"
      }
    ],
    "tags": [
      {
        "description": "Secured Admin-only calls",
        "name": "admins"
      },
      {
        "description": "Operations available to regular
developers",
        "name": "developers"
      }
    ]
}
```

Figure 18. Uploading Description

When navigating to *Insert an OpenAPI description*, two sections are displayed. The box at the left side of the page is the input for the title of the description inserted in the database. The box placed on the right side of the page, is the description input in JSON format. This way the user is able to enter the name of the file as well as the content of the OpenAPI description that desires, and perform OpenAPI QL queries on them.

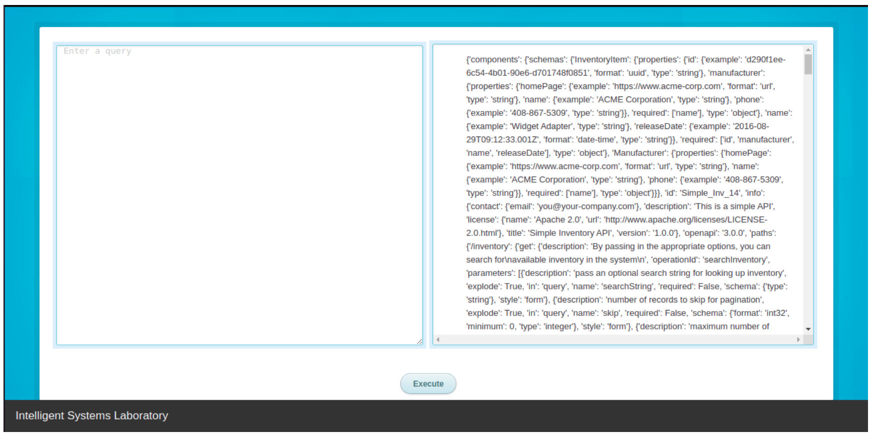


```
SELECT s.*
FROM Service s, Request req, Response resp
WHERE s.request = req
AND req.response = resp
AND resp.status_code
BETWEEN 202 AND 400
```

Figure 19. Query Execution

Next, in the *Query Execution* page, the user is able to write an OpenAPI QL query. After entering the desirable query, the user can use the submit button to see the results collected from the database. The syntax of the query was discussed in a previous section and the grammar is available in the figure ?? as well as a list of examples following in the 6.2 section.

Once the query is submitted, next to the OpenAPI QL query text area, another text area next to the previous, will show the results in JSON format, as the following picture illustrates:



```
{'components': {'schemas': {'InventoryItem': {'properties': {'id': {'example': 'd290f1ee-
6c54-4b01-90e6-d701748f0851', 'format': 'uuid', 'type': 'string'}, 'manufacturer':
{'properties': {'homePage': {'example': 'https://www.acme-corp.com', 'format': 'url',
'type': 'string'}, 'name': {'example': 'ACME Corporation', 'type': 'string'}, 'phone':
{'example': '408-867-5309', 'type': 'string'}}, 'required': ['name'], 'type': 'object'}, 'name':
{'example': 'Widget Adapter', 'type': 'string'}, 'releaseDate': {'example': '2016-08-
29T09:12:33.001Z', 'format': 'date-time', 'type': 'string'}}, 'required': ['id', 'manufacturer',
'name', 'releaseDate'], 'type': 'object'}, 'Manufacturer': {'properties': {'homePage':
{'example': 'https://www.acme-corp.com', 'format': 'url', 'type': 'string'}, 'name':
{'example': 'ACME Corporation', 'type': 'string'}, 'phone': {'example': '408-867-5309',
'type': 'string'}}, 'required': ['name'], 'type': 'object'}}}, 'id': 'Simple_Inv_14', 'info':
{'contact': {'email': 'you@your-company.com'}, 'description': 'This is a simple API',
'license': {'name': 'Apache 2.0', 'url': 'http://www.apache.org/licenses/LICENSE-
2.0.html'}, 'title': 'Simple Inventory API', 'version': '1.0.0'}, 'openapi': '3.0.0', 'paths':
{'/inventory': {'get': {'description': 'By passing in the appropriate options, you can
search for\navailable inventory in the system\n', 'operationId': 'searchInventory',
'parameters': [{'description': 'pass an optional search string for looking up inventory',
'explode': True, 'in': 'query', 'name': 'searchString', 'required': False, 'schema': {'type':
'string'}, 'style': 'form'}, {'description': 'number of records to skip for pagination',
'explode': True, 'in': 'query', 'name': 'skip', 'required': False, 'schema': {'format': 'int32',
'minimum': 0, 'type': 'integer'}, 'style': 'form'}, {'description': 'maximum number of
```

Figure 20. Query Results

## 6.2 Examples

In this section we present a list of examples of the mechanism described before. The translated queries derive from a mapping between OPENAPI QL and N1QL expressions. The main clauses such are SELECT, FROM and WHERE, are directly mapped from our language to N1QL. However, their content is translated separately, emphasizing on WHERE clause conditions. The examples illustrated below, present a service discovery, depending on the number of properties queried. Each query category includes a certain number of conditions that refer to specific objects. Each condition complies with a required OpenAPI property, although the OpenAPI object may be optional.

In the next examples we show some query cases in our knowledge base. We present OPENAPI QL queries that apply one or more conditions over OAS services. First we are going to discuss examples that fulfil a few conditions. After that we will focus on testing queries based on the number of conjunctive operators combining such conditions.

### 6.2.1 Example 1

In this example we seek the Request Body of a Service with a specific format for the body of the data. The Service element represents the OAS definition instance. In *WHERE* clause, the first condition defined by order, checks the existence of a request declaration within a Service. The second one examines if this request has a specific media type. Both conditions must be true in order for the results to be retrieved.

| **Query Example 1** |
|---|
| SELECT s.id AS service_id<br>FROM Service s, Request req<br>WHERE s.request = req<br>AND<br>req.media_type = “application/json” |
| **Results** |
| {’service_id’: ’Artist_API’}<br>{’service_id’: ’Simple_API_Description’}<br>{’service_id’: ’Simple_Inventory_API’}<br>{’service_id’: ’Petstore’}<br>...<br>Number of documents: 81<br>Execution time: — 0.26708412170410156 seconds — |

Table 14. Media Type condition example

### 6.2.2 Example 2

In this example we demonstrate the relations between a Service, a Request, a Response and Schema. We present a multiple condition query which shows that the Request Element can have one or more Response Elements. Each Response Element has a schema describing the body of the data, according to OAS 3.0. Finally the schema case that OPENAPI QL supports, is a property with a specific name. So, this query asks for the property ”company_id” defined underneath a response schema.

| **Query Example 2** |
|---|
| SELECT s.id AS service_id<br>FROM Service s, Request req, Response resp, Schema sc<br>WHERE s.request = req<br>AND req.response = resp<br>AND resp.schema = sc<br>AND sc.property = “company_id” |
| **Results** |
| {’service_id’: ’Artist_API’}<br>{’service_id’: ’Simple_API_Description’}<br>{’service_id’: ’Simple_Inventory_API’}<br>{’service_id’: ’Petstore’}<br>...<br>Number of documents: 34<br>Execution time: — 0.4804239273071289 seconds — |

Table 15. Schema condition example

### 6.2.3 Example 3

Here we seek to find a response with particular status code. More specific, this example demonstrates the use of the ***BETWEEN*** operator asking for a number in a range of discrete values. The *WHERE* clause consists of three conditions with the first and the second representing the relations between a Service, Request and a Response. The last one, specifies whether a value in a range or not.

| **Example 3** |
|---|
| SELECT s.id AS service_id<br>FROM Service s, Request req, Response resp<br>WHERE s.request = req<br>AND req.response = resp<br>AND resp.status_code BETWEEN 202 AND 400 |
| {’service_id’: ’Artist_API’}<br>{’service_id’: ’Simple_API_Description’}<br>{’service_id’: ’Simple_Inventory_API’}<br>{’service_id’: ’Petstore’}<br>...<br>Number of documents: 58<br>Execution time: — 0.3751966953277588 seconds — |

Table 16. Status code example with the use of BETWEEN operator

### 6.2.4 Example 4

The following *SELECT* statement asks for request execution from an API. The HTTP method we are seeking and indicates the desired action is the verb POST. Next, the query asks for the request data to be a schema including the property "email". If each one of the conditions is true, then static results are retrieved.

**Example 4**

```
SELECT s.id AS service_id
FROM Service s, Request req, Schema sc
WHERE s.request = req
AND req.method = "post"
AND req.schema = sc
AND sc.property = "email"
```

```
{'service_id': 'Artist_API'}
{'service_id': 'Simple_API_Description'}
{'service_id': 'Simple_Inventory_API'}
{'service_id': 'Petstore'}
...
Number of documents: 11
Execution time: — 0.20876860618591309 seconds —
```

Table 17. HTTP method and schema condition example

### 6.2.5 Example 5

Here we are searching for an authentication scheme and a parameter while a request is executing. What is important about this query is the security scheme that protect APIs. OpenAPI 3.0 allows the definition of the OAuth2, Api keys, cookies and OpenIdConnect schemes. In this example, we ask for an action that uses HTTP method GET for execution, passes a query parameter through the API and an API protection scheme with a certain scope.

| **Example 5** |
|---|
| SELECT s.id AS service_id<br>FROM Service s, Request req, QueryParam qp, OAuth2Auth oa<br>WHERE s.request = req<br>AND req.method = ”get”<br>AND req.queryparam = qp<br>AND qp.name = ”skip”<br>OR req.oauth2auth = oa<br>AND oa.scope = ”write” |
| {’service_id’: ’Artist_API’}<br>{’service_id’: ’Simple_API_Description’}<br>{’service_id’: ’Simple_Inventory_API’}<br>{’service_id’: ’Petstore’}<br>...<br>Number of documents: 2<br>Execution time: — 0.47533433723449707 seconds — |

Table 18. HTTP method, query parameter and OAuth 2.0 security condition example

### 6.2.6 Example 6

This example includes request, parameter, and header definitions. Here we query for a Service with three characteristics. The first is the use of GET method to request data. The second one is the media type of the request, to be a string, which, in this case, is "application/json". The last one refer to a header named "X-Rate-Limit". If the logical operations between the conditions are true, then the query returns the id of the matched descriptions.

| **Example 6** |
|---|
| SELECT s.id AS service_id<br>FROM Service s, Request req, QueryParam qp,<br>Header hed<br>WHERE s.request = req<br>AND req.method = "get"<br>AND req.media_type = "application/json"<br>AND req.queryparam = qp<br>AND qp.name = "skip"<br>OR *req.header = hed*<br>AND hed.name = "X-Rate-Limit" |
| {'service_id': 'Artist_API'}<br>{'service_id': 'Simple_API_Description'}<br>{'service_id': 'Simple_Inventory_API'}<br>{'service_id': 'Petstore'}<br>...<br>Number of documents: 2<br>Execution time: — 0.52033433723449707 seconds — |

Table 19. HTTP method, media type and header condition example

## 6.3 Results

To discover services within OAS 3.0 definition catalogues we need to develop tools that manipulate data and retrieve the desired results. Our approach requires that the services are inserted in Couchbase Database after being processed by *Flattening Reference* module. With queries expressed in OPENAPI QL as input text in our system, the translator is automatically enabled to convert them into valid N1QL expressions. After that, the produced queries are executed over OAS data. The database includes 150 services collected from SwaggerHub and Amazon. Each service has the average of 400 lines. The following tables illustrate OPENAPI QL queries. The criteria distinguishing the queries are the number of conditions applied and the increasing complexity by the use of multiple conjunctive operators.

### 6.3.1 Run-time Performance

In this section we discuss the performance of several OpenAPI Queries, executed in our Web Application. The run time efficiency of an OpenAPI QL query depends on the data volume and how complicated is the query executed on them. Consequently, the larger the dataset is, the longer it will take for the answer to return.

Executing a number of OpenAPI queries on our database containing several descriptions, we concluded that the execution time of a query basically depends on three factors [1]. The first factor is about the indexes created and used in this database. The second factor is the size of the dataset. Finally, the third factor is about the query predicates that dictate the number of document keys and hence the documents fetched and processed.

The first factor comes as a result from the database indexing [2]. In this work we applied a primary index on our database. A primary index contain a set of keys for a collection which allows the query engine to access all the documents, then do the filtering, joining, aggregation operations on them holding the uniqueness of the document key. The index used in our case is available below:

| |
|---|
| CREATE PRIMARY INDEX '#primary' ON 'OASBucket' |

Table 20. Primary Index applied on OASBucket

[1] https://dzone.com/articles/new-performance-tricks-with-n1ql
[2] https://docs.couchbase.com/server/5.5/performance/indexing-and-query-perf.html

The use of a primary index for full bucket scans (primary scans) is desirable when the query does not have any filters (predicates) or when no other index or access path can be used. OpenAPI QL translated expressions have filters but not indexes or access paths, because indexing is a difficult process to be applied on OpenAPI descriptions. This happens because N1QL's indexes, besides primary, require to define an access path for each type of query in order to filter the data in a more efficient way. In our case the access paths are not clearly defined in OpenAPI descriptions, because N1QL is not able to now the exact path of the information needed to be retrieved. The following example illustrates the access path of a response schema in OpenAPI:

| |
|---|
| 'responses'.'200'.'content'.'text/plain'.'schema' |

Table 21. Access Path of a Response Schema in OpenAPI description

The elements of the access path surrounded by "{ }" are field names that completely unknown to the user. So, when querying a response schema the possible access paths for this information can be numerous. Considering also the fact that a primary index fetches all the documents in the bucket and then apply the filter, the query processing is very expensive [3]. This means that affects the execution time of N1QL translated queries and hence the performance of OpenAPI QL queries because it causes a delay.

The second factor does not affect the execution time in a several way because the dataset containing the OpenAPI descriptions is small. OASBucket, includes 100-150 OpenAPI descriptions in JSON format. Although a description can be a large file itself, the expected execution time for a small dataset is low.

Finally, the third factor is the determinant of query's execution time. With the term predicate we refer to a condition expression that evaluates to a boolean value. Considering the fact that every OpenAPI QL query translates the content of SELECT and FROM clauses in the same way for all the queries, the WHERE content affects the actual query's execution time because it filters the data over specific conditions.

In this thesis, the translated expressions in N1QL contain one or multiple predicates within WHERE clause, as presented in detail in table A.1. The ANY predicate [4] uses the IN and WITHIN operators

[3] https://dzone.com/articles/primary-uses-for-couchbase-primary-index

[4] https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/collectionops.html#collection-op-any

to range through the collection of OpenAPI descriptions. More specific, the functionality of the IN operator is that specifies the search depth to include only the current level of an array and not to include any child or descendant arrays. On the other hand, WITHIN operator defines the search depth to include the current level of an array and all of its child and descendant arrays. As a result, N1QL expressions containing WITHIN operator tend to cause a delay on the execution time because they loop in a rather larger depth. The table below categorizes the queries according to the complexity of the conditions when querying a specific field.

Table 22. Summarizing execution times

| **OpenAPI QL** | **Average Time (ms)** | **Average Filtering Time (ms)** | **Target Element** | **Operator** |
|---|---|---|---|---|
| Table 24 | 48.63 | 25.25 | Operation | AND |
| Table 25 | 393.73 | 289.94 | Response | AND |
| Table 26 | 458.28 | 360.83 | Request | AND |
| Table 27 | 465.46 | 393.34 | Parameter | AND |
| Table 28 | 476.27 | 443.15 | Security Scheme | AND |
| Table 29 | 492.42 | 471.24 | Header | AND |

The above table contains the average execution times of six categories of OpenAPI QL queries. The tables are categorized by the increasing complexity of their conditions. Firstly, the queries of Table 24 search for a specific HTTP method in a request. The expression for querying the property ”method” is available in table A.1. Observing the execution times we conclude that querying by the property defining an operation, the results are fetched fast. This comes as a result from the fact that the Operation Object, according to OpenAPI, starts with the HTTP method definition and encapsulates all the other properties we are searching for. So, if we filter the data with a method, we automatically reduce the accesses in other operations.

Continuing, the Table 25 contains statements querying properties within a Response Object, such is a status code or a schema. Each property is translated in an equal N1QL expression which defines a ANY predicates with the use of IN or WITHIN operators. The query asking for the property ”media_type”, is translated into an equal N1QL expression in which the predicate uses the operator IN. This means that when the condition matches the field in the document, the loop stops due to ANY. Contrastingly, the query scanning for the property "status_code", defines the ANY predicate using WITHIN operator, which means that if the status code is not found in the current level of JSON hierarchy, the query does not terminates due to ANY. As a result, the query asking for status codes in a range, is much slower. For the same reason, the queries asking for a specific property in a response schema, have similar execution times with the status code property.

In Table 26 we present the queries searching for a request property, such is a media type or a schema. The property defining the media type of a request, is located in a higher JSON level from the response media type. So, we expected the request media type query, to be faster. However, is much slower because the Response Object is a required object in every defined operation in Paths Object, according to OpenAPI. On the other hand, the request body -defining both media type and schema- is an optional object. Because of that, the query is more rare to locate a request body in an OpenAPI description than a response. If a query searches for a response, the query stops when it finds the first response matching the criteria in the file, due to ANY. If ANY predicate searches for an optional field, the query continues filtering the data until it finds it. As a result, the queries of Table 26 are slower than the response queries, although they have common the "media_type" and "schema" properties.

Next, the Table 27 contains queries scanning for a specific parameter. The possible parameters in an OpenAPI description are of type "path", "cookie", "query" and "header", distinction based on their location. Although there are four types of parameters, the "header" type is not included on OpenAPI QL parameters because we considered them as headers in the same OpenAPI Element with response headers.In OpenAPI 3.0, parameters are defined in the parameters section of an operation or path, as an array. In OpenAPI QL the a parameter of type "path" is considered as an PathParam Element. In the same way, parameters of type "query" and "cookie" considered as QueryParam and CookieParam respectively. Each parameter in an OpenAPI description is scanned by an ANY predicate in N1QL which loops over the current depth of the parameter's array, until it finds the parameter meeting the criteria. Considering the fact that OpenAPI treats the parameters as optional, the execution time of a parameter is the predicate's cost, as summarized in Table 22.

In Table 28, the queries presented are about searching for specific authentication and authorization schemes in OpenAPI descriptions. Security schemes are defined in two parts of the OpenAPI document. The definition within the Operation Object contains only the name of the security scheme applied. The Components Object provides also some additional information for each kind of security, under "securitySchemes" key. Consequently, the query has to check first, if the name of the security defined in an operation is equal to the name defined in Components Object. Secondly, it has to loop over the information of the corresponding security scheme in order to locate the one matches the condition. This process uses many predicates -and nested ones- that scan two parts of the same document simultaneously. For this reason, these queries have more costly filters in WHERE clause, and hence larger execution times than all the above queries.

The last table, 29, is about the headers described in OpenAPI QL. According to OpenAPI, a header can be found in two places in a document, within Responses Object and Parameters Object. So, when querying a header the predicate has to check if a specific header exists in both places. For this reason, the one predicate examines if a header exists in responses, using WITHIN operator. The other checks if a parameters is of type header, using IN operator to filter the parameters array. The two predicates are joined with OR operator, which means that both conditions filter data, independently from each other. So, observing the results of such queries, we concluded that the queries searching for headers, are the most complicated expressions in N1QL and hence they have the higher execution times.

Summarizing, in the collected results we observed that the queries searching required OpenAPI elements, such is the Response Object, tend to be quicker than the search of an optional element, such is a Parameter Object. For example the Responses Object appears in all the operations defined in OpenAPI descriptions. On the other hand, a parameter, a header or a request body are elements that OpenAPI defines optionally. If the document field is present in the documents, the condition matched early, it will end due to ANY. If the field is not present or rarely appears in the document, the conditions needs to continue till it finds it or terminate without result. This means that the type of the OpenAPI field queried, can also affect the results.

Chapter 7

# Conclusions and Future Work

In this work, we introduced OPENAPI QL, a query language for querying services within OAS descriptions. The proposed query language provides a an SQL-like syntax with a powerful set of operators for querying simple API characteristics. Our approach is based on OpenAPI Query model responsible for handling semi-structured OAS data in JSON format. With instantiating OpenAPI Elements and properties based on the classic OpenAPI model and REST architectural constraints, we succeed to design a high-level query language for querying API information in OpenAPI descriptions.

To show proof of concept, all language features are implemented in full and supported by a software system for OPENAPI QL translation and processing consisting of a query translator and an API for query execution. The OPENAPI QL QL system is described in chapter 5.

## 7.1 Conclusions

The contributions of this work are the following:

- The language is capable of querying information in OAS descriptions than any other query language.
- OPENAPI QL supports SQL-like syntax, as well as the basic operators such are logical operators used in condition statements, or *AS* operator used for element aliasing.
- OpenAPI Query model handles semi-structured in JSON format. Apart from the deeply nested structure of OAS descriptions, it also deals with the relations between OAS objects and properties.
- OPENAPI QL is fully compliant with REST principles and architectural constraints. A user has to be familiar with REST in order to use OPENAPI QL, but not necessarily with the OpenAPI Specification 3.0.

## 7.2 Future Work

The following are important issues for future work:

- Extend the syntax of our language with adding more operators improving the language utilities. Right now the language is only capable of querying only some features referring to a specific part of the OAS description.
- Dealing with schema objects is an important issue for future work. OpenAPI 3.0 uses an extended subset of JSON Schema Specification, but OPENAPI QL supports one specific case of a schema which is the one including the ”properties” keyword.
- Creating the right indexes on the Couchbase database because is critical for the query performance. Indexes will be used to reduce the number of data scans that have to be made in order to find the specific record.

# References

[1] *A Guide to What's New in OpenAPI 3.0*. https://swagger.io/blog/news/whats-new-in-openapi-3-0/.

[2] *About: OpenAPI Specification*. https://dbpedia.org/page/OpenAPI_Specification.

[3] *Collection Operators*. https://docs.couchbase.com/server/current/n1ql/n1ql-language-reference/collectionops.html.

[4] *Document size and query performance*. https://forums.couchbase.com/t/document-size-and-query-performance/153.

[5] *Documenting Your Existing APIs: API Documentation Made Easy with OpenAPI & Swagger*. https://swagger.io/resources/articles/documenting-apis-with-swagger/.

[6] *Indexing JSON Documents and Query Performance*. https://docs.couchbase.com/server/5.5/performance/indexing-and-query-perf.html.

[7] Tom Johnson. *Introduction to the OpenAPI specification*. https://idratherbewriting.com/learnapidoc/pubapis_openapi_intro.html.

[8] Aikaterini Karavasileiou. "An ontology for describing OpenAPI version 3 services in the cloud". In: (2019).

[9] Ilya Katsov. *NOSQL DATA MODELING TECHNIQUES*. https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/.

[10] Daniel Kirkdorffer. *OpenAPI spec and generated reference docs*. https://idratherbewriting.com/learnapidoc/restapispecifications.html.

[11] Zell Liew. *Understanding And Using REST APIs*. https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/.

[12] *Main Uses for Couchbase Primary Index*. https://dzone.com/articles/primary-uses-for-couchbase-primary-index.

[13] Nikolaos Mainas. "Semantically Enriched API Descriptions for Improv- ing Service Discovery in Cloud Environments". In: (2017).

[14] Ravi Mayuram. *N1QL brings SQL to NoSQL databases*. https://www.infoworld.com/article/2977045/n1ql-brings-sql-to-nosql-databases.html.

[15] *N1QL Language Reference*. https://docs.couchbase.com/server/6.5/n1ql/n1ql-language-reference/index.html.

[16] *OpenAPI Specification Version 3.0.3*. https://swagger.io/specification/.

[17] *Powerful NoSQL queries and performance at scale*. https://www.couchbase.com/products/n1ql.

[18] *Primary Uses for Couchbase Primary Index*. https://blog.couchbase.com/primary-uses-for-couchbase-primary-index/.

[19] *REST Architectural Constraints*. https://restfulapi.net/rest-architectural-constraints/.

[20] Thilini Shanika. *Open API 3.0 vs Swagger 2.0*. https://medium.com/@tgtshanika/open-api-3-0-vs-swagger-2-0-94a80f121022.

[21] *The Couchbase Data Model*. https://docs.couchbase.com/server/current/learn/data/document-data-model.html.

[22] *The Latest Performance Tricks in N1QL*. https://dzone.com/articles/new-performance-tricks-with-n1ql.

[23] *Tokens*. https://www.ibm.com/docs/en/rdfi/9.6.0?topic=le-tokens.

[24] Keshav Vasudevan. *What is API Documentation, and Why It Matters?* https://swagger.io/blog/api-documentation/what-is-api-documentation-and-why-it-matters/.

[25] *What are some alternatives to SwaggerHub?* https://stackshare.io/swaggerhub/alternatives.

[26] *Why Couchbase?* https://docs.couchbase.com/server/current/introduction/why-couchbase.html.

Appendix A

# OpenAPI QL Appendix

## A.1 OPENAPI QL to N1QL Mapping Table

| **OPENAPI QL** | **N1QL** |
|---|---|
| Service | OASBucket m |
| Service.id | META(m).id |
| Service.* | m.* |
| Service.Request | ANY v WITHIN OBJECT_VALUES(m.‘paths‘)[*] SATISFIES OBJECT_VALUES(m.‘paths‘)[*] IS NOT MISSING END |
| Request | OASBucket m |
| Request.method | ANY v WITHIN OBJECT_VALUES(m.‘paths‘)[*].<method_string> SATISFIES OBJECT_VALUES(m.‘paths‘)[*] .<method_string> IS NOT MISSING END |
| Request.media_type | ANY y IN OBJECT_NAMES(v.‘requestBody‘.‘content‘) SATISFIES y = <request_media_type_string> END |
| Request.schema | ANY x WITHIN v.‘requestBody‘ SATISFIES x AND ANY t IN OBJECT_NAMES(x.‘schema‘.‘properties‘) SATISFIES t = <request_schema_string> END END |

| **OPENAPI QL** | **N1QL** |
|---|---|
| Request.Response | ANY s IN OBJECT_NAMES(v.‘responses‘)<br>SATISFIES<br>OBJECT_NAMES(v.‘responses‘)<br>IS NOT MISSING END |
| Response | OASBucket m |
| Response.status_code | ANY s IN OBJECT_NAMES(v.‘responses‘)<br>SATISFIES<br>TO_NUMBER(s) <status_code_string> END END |
| Response.media_type | ANY x WITHIN v.‘responses‘<br>SATISFIES x<br>AND<br>ANY y IN OBJECT_NAMES(x.‘content‘)<br>SATISFIES<br>y = <response_media_type_string> END END |
| Response.schema | ANY x WITHIN v.‘responses‘<br>SATISFIES x<br>AND<br>ANY t IN OBJECT_NAMES(x.‘schema‘.‘properties‘)<br>SATISFIES t = <response_schema_string><br>END END |
| Request.QueryParam | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”query” END |
| QueryParam | OASBucket m |
| QueryParam.name | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”query”<br>AND<br>p.‘name‘ = <parameter_name> END |
| QueryParam.required | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”query”<br>AND p.‘true‘ = <true or false> END |

| **OPENAPI QL** | **N1QL** |
|---|---|
| Request.PathParam | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”path” END |
| PathParam | OASBucket m |
| PathParam.name | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”path”<br>AND p.‘name‘ = <parameter_name> END |
| PathParam.required | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”path”<br>AND p.‘true‘ = <true or false> END |
| Request.Header | (ANY t within v.‘responses‘<br>SATISFIES v.‘headers‘ IS NOT MISSING END)<br>OR<br>(ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”header” END) |
| Header | OASBucket m |
| Header.name | ANY t within v.‘headers‘<br>SATISFIES<br>(ANY h WITHIN OBJECT_NAMES(v.‘headers‘)<br>SATISFIES h = <header_name> END) END)<br>OR<br>(ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”header”<br>AND p.‘name‘ = <header_name> END |
| Request.CookieParam | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”cookie” END |
| CookieParam | OASBucket m |
| CookieParam.name | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”cookie”<br>AND p.‘name‘ = <parameter_name> END |

| **OPENAPI QL** | **N1QL** |
|---|---|
| CookieParam. required | ANY p IN v.‘parameters‘<br>SATISFIES p.‘in‘ = ”cookie”<br>AND p.‘true‘ = <true or false> END |
| Request.Oauth2Auth | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x)<br>AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”oauth2” END)<br>END END |
| Oauth2Auth | OASBucket m |
| Oauth2Auth.flow | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x) AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names<br>AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”oauth2”<br>AND<br>( ANY k WITHIN c.[y].‘flows‘<br>SATISFIES k.‘scopes‘ AND<br>( ANY i IN OBJECT_NAMES(k.‘scopes‘)<br>SATISFIES i = <flow_string> END ) END)<br>END) END END |

| **OPENAPI QL** | **N1QL** |
|---|---|
| Oauth2Auth.scope | ANY x IN v.‘security‘ SATISFIES<br>OBJECT_NAMES(x)<br>AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”oauth2”<br>AND<br>( ANY p IN OBJECT_NAMES(c.[y].‘flows‘)<br>SATISFIES p = <flow_string> END )<br>END) END END |
| Request.OpenIdConnectAuth | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x) AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”openIdConnect”<br>END) END END |
| OpenIdConnectAuth | OASBucket m |

| **OPENAPI QL** | **N1QL** |
|---|---|
| OpenIdConAuth.scope | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x)<br>AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”openIdConnect”<br>AND<br>(ANY h WITHIN v.‘security‘<br>SATISFIES h.[y]<br>AND ( ANY t WITHIN h.[y]<br>SATISFIES t = <scope_string><br>END ) END )<br>END) END END |
| OpenIdConAuth.openIdConnectUrl | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x)<br>AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”openIdConnect”<br>AND<br>’c.[y].‘openIdConnectUrl‘ = <url_string><br>END) END END |

| **OPENAPI QL** | **N1QL** |
|---|---|
| Request.ApiKeyAuth | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x)<br>AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”apiKey”<br>END) END END |
| ApiKeyAuth | OASBucket m |
| ApiKeyAuth.name | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x)<br>AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”apiKey”<br>AND<br>c.[y].‘name‘ = <name_string><br>END) END END |
| ApiKeyAuth.in | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x)<br>AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names AND<br>(ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”apiKey”<br>AND<br>c.[y].‘in‘ = <location_string><br>END) END END |

| **OPENAPI QL** | **N1QL** |
|---|---|
| Request.HttpAuth | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x)<br>AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names<br>AND (ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”http” END) END END |
| HttpAuth | OASBucket m |
| HttpAuth.scheme | ANY x IN v.‘security‘<br>SATISFIES OBJECT_NAMES(x) AND<br>ANY y IN OBJECT_NAMES(x)<br>SATISFIES y = security_names<br>AND (ANY c WITHIN m.‘components‘<br>SATISFIES c.[y].‘type‘ = ”http”<br>c.[y].‘scheme‘ = <scheme_string><br>END) END END |

Table 23. Translation of OPENAPI QL expressions in N1QL

## A.2 Results

Table 24. Querying HTTP Method

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_id$<br>FROM Service s, Request req<br>WHERE s.request = req<br>AND req.method = ”get” | 121 | 66.31ms | 0.033826290 |
| SELECT s.id AS service$_id$<br>FROM Service s, Request req<br>WHERE s.request = req<br>AND req.method = ”post” | 103 | 53.21ms | 0.033589441 |

**Table 24 continued from previous page**

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_id$ FROM Service s, Request req WHERE s.request = req AND req.method = ”put” | 13 | 36.7ms | 0.0024782846 |
| SELECT s.id AS service$_id$ FROM Service s, Request req WHERE s.request = req AND req.method = ”delete” | 1 | 36.9ms | 0.017565836 |
| SELECT s.id AS service$_id$ FROM Service s, Request req WHERE s.request = req AND req.method = ”patch” | 1 | 50.1ms | 0.04250844 |
| SELECT s.id AS service$_id$ FROM Service s, Request req WHERE s.request = req AND req.method = ”trace” | 1 | 55.1ms | 0.028044024 |
| SELECT s.id AS service$_id$ FROM Service s, Request req WHERE s.request = req AND req.method = ”options” | 1 | 42.1ms | 0.017670918 |

Table 25. Querying Responses

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_{i}d$ FROM Service s, Request req, Response resp WHERE s.request = req AND req.response = resp AND resp.media$_{t}ype = ”application/json”$ | 137 | 136.10ms | 0.101518311 |
| SELECT s.id AS service$_{i}d$ FROM Service s, Request req, Response resp WHERE s.request = req AND req.response = resp AND resp.status$_{c}ode$ BETWEEN 202 AND 400 | 4 | 433.33ms | 0.388512566 |
| SELECT s.id AS service$_{i}d$ FROM Service s, Request req, Response resp WHERE s.request = req AND req.response = resp AND resp.media$_{t}ype = ”application/json”$ AND resp.status$_{c}ode$ BETWEEN 201 and 400 | 2 | 433.10ms | 0.369252514 |
| SELECT s.id AS service$_{i}d$ FROM Service s, Request req, Response resp, Schema sc WHERE s.request = req AND req.response = resp AND resp.schema = sc AND sc.property = ”company$_{i}d$” | 34 | 443.21ms | 0.45185561 |

**Table 25 continued from previous page**

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_id$<br>FROM Service s, Request req,<br>Response resp<br>WHERE s.request = req<br>AND req.response = resp<br>AND resp.media$_type = "application/json$"<br>AND resp.status$_code$<br>BETWEEN 202 AND 400 | 4 | 439.67ms | 0.396330521 |
| SELECT s.id AS service$_id$<br>FROM Service s, Request req,<br>Response resp, Schema sc<br>WHERE s.request = req<br>AND req.response = resp<br>AND resp.status$_code$<br>BETWEEN 201 AND 404<br>AND resp.schema = sc<br>AND sc.property = "company$_id$" | 2 | 476.89ms | 0.45661855 |

TABLE 26. Querying Requests

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_id$<br>FROM Service s, Request req<br>WHERE s.request = req<br>AND req.media$_type = "application/json"$ | 81 | 284.33ms | 0.284516349 |
| SELECT s.id AS service$_id$<br>FROM Service s, Request req, Schema sc<br>WHERE s.request = req<br>AND req.schema = sc<br>AND sc.property = "address" | 23 | 293.60ms | 0.288943502 |
| SELECT s.id AS service$_id$<br>FROM Service s, Request req, Schema sc<br>WHERE s.request = req<br>AND req.media$_type = "application/json"$<br>AND req.schema = sc<br>AND sc.property = "address" | 23 | 309.88MS | 0.29494226500000004 |

Table 27. Querying Parameters

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_id$ FROM Service s, Request req, QueryParam qp WHERE s.request = req AND req.queryparam = qp AND qp.name = "limit" | 56 | 422.17ms | 0.400491969 |
| SELECT s.id AS service$_id$ FROM Service s, Request req, QueryParam qp, HTTPAuth ht WHERE s.request = req AND req.httpauth = ht AND ht.scheme = "bearer" AND req.queryparam = qp AND qp.name = "limit" | 5 | 460.51ms | 0.401898931 |
| SELECT s.id AS service$_id$ FROM Service s, Request req, PathParam pp WHERE s.request = req AND req.pathparam = pp AND pp.name = "id" | 11 | 455.06ms | 0.406151247 |
| SELECT s.id AS service$_id$ FROM Service s, Request req, Response resp, Schema sc WHERE s.request = req AND req.media$_type = "application/json$" AND req.response = resp AND resp.status$_code$ BETWEEN 201 AND 404 AND resp.schema = sc AND sc.property = "company$_id$" | 2 | 490.25ms | 0.411585896 |

**Table 27 continued from previous page**

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_id$<br>FROM Service s, Request req,<br>CookieParam co<br>WHERE s.request = req<br>AND req.cookieparam = co<br>AND co.name = ”skip” | 4 | 491.11ms | 0.347215235 |

Table 28. Querying Security Schemes

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_id$ FROM Service s, Request req, ApiKeyAuth ap WHERE s.request = req AND req.apikeyauth = ap AND ap.name = ”api$_key$” | 11 | 429.58ms | 0.412461024 |
| SELECT s.id AS service$_id$ FROM Service s, Request req, OAuth2Auth oa WHERE s.request = req AND req.oauth2auth = oa AND oa.scope = ”write” | 4 | 492.58ms | 0.478035119 |
| SELECT s.id AS service$_id$ FROM Service s, Request req, OpenIdConnectAuth op WHERE s.request = req AND req.openidconnectauth = op AND op.scope = ”pets$_read$” | 4 | 473.53ms | 0.451437119 |
| SELECT s.id AS service$_id$ FROM Service s, Request req, OAuth2Auth oa WHERE s.request = req AND req.oauth2auth = oa AND oa.flow = ”implicit” | 4 | 461.50ms | 0.4436699269 |
| SELECT s.id AS service$_id$ FROM Service s, Request req, HTTPAuth ht WHERE s.request = req AND req.httpauth = ht AND ht.scheme = ”bearer” | 2 | 444.18ms | 0.439482525 |

Table 29. Querying Headers

| OPENAPI QL Query | Documents | Execution time | Condition cost (s) |
|---|---|---|---|
| SELECT s.id AS service$_id$<br>FROM Service s, Request req,<br>Header hed<br>WHERE s.request = req<br>AND req.header = hed<br>AND hed.name = ”X-Rate-Limit” | 22 | 459.79ms | 0.402258521 |
| SELECT s.id AS service$_id$<br>FROM Service s, Request req,<br>Header hed, PathParam pa<br>WHERE s.request = req<br>AND req.media$_type = ”application/json$”<br>AND req.pathparam = pa<br>AND pa.required = true<br>AND req.header = hed<br>AND hed.name = ”X-Rate-Limit” | 17 | 525.06ms | 0.475243555 |