ALEXANDRU IOAN CUZA UNIVERSITY OF IAȘI

FACULTY OF COMPUTER SCIENCE

BACHELOR'S THESIS

Semantic Web API Specification

submitted by

Ioana Bîrsan

Term: *July, 2019*

Scientific coordinator

Conf. Dr. Sabin-Corneliu Buraga

ALEXANDRU IOAN CUZA UNIVERSITY OF IAȘI
FACULTY OF COMPUTER SCIENCE

Semantic Web API Specification

Ioana Bîrsan

Term: *July, 2019*

Scientific coordinator
Conf. Dr. Sabin-Corneliu Buraga

Declarație privind originalitatea și respectarea drepturilor de autor

Prin prezenta declar că Lucrarea de licență cu titlul "Semantic Web API Specification" este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele sau cursiv și dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- codul sursă, imagini etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor și dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Absolvent *Ioana Bîrsan*

Iași, *Iulie 2019*

_____

(semnătură în original)

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul "Semantic Web API Specification", codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea Alexandru Ioan Cuza din Iași să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent *Ioana Bîrsan*

Iași, *Iulie 2019*

_____

(semnătură în original)

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The Web is becoming *a large repository of open data, available for rich exploratory querying, machine processing such as generating visualizations, and for combining multiple data sources* [14]. The Semantic Web (Web 3.0) has been designed as a WWW extension that allows computing tools to search, combine and process content that is based on the meaning it has for us.

Thesis Statement *Can we augment the existing Web API Specifications with Semantic support in order to derive both machine and human readable content in a better, more comprehensive way?*

Thesis Contribution *Added support for Semantic Augmentation of OpenAPI specification within Swagger Editor and Swagger UI tools.*

The thesis is structured as follows. In chapter 1 we provide the introduction. Chapter 2 introduces the concept of Web services, provides an overview of the SOA architectural style, and describes the architecture of Web services along with its identified types, properties and steps involved in offering and consuming it. Chapter 3 describes the concept of Web APIs, how they emerged and what directions they are headed, it exposes several existing standards for defining an API, with particular emphasis on OpenAPI Specification, and provides recommendations regarding the design and documentation of an Web API. Chapter 4 introduces the foundations of Semantic Web technologies: Resource Description Framework (RDF), RDF Schema (RDFS), OWL 2 Web Ontology Language (OWL 2), and SPARQL Protocol and RDF Query Language (SPARQL). Chapter 5 describes the process of augmenting the OpenAPI specification with semantics, it provides an overview of the technologies used for development and the open source projects that we have contributed to in order to achieve this goal, it further proposes a set of extensions that can be used to augment the OpenAPI specification with knowledge, and presents the functionality added to the Swagger Editor and Swagger UI projects. Chapter 6 presents

a case study in order to demonstrate the functionality to the OpenAPI Specification tools that were augmented with Semantic support. In chapter 7 we conclude.

Chapter 2

Web Services

Contents

Introduction

The chapter introduces the concept of Web services. Section 2.1 provides a brief overview of the Service-Oriented Architecture (SOA) architectural style, it also describes the roles and operations involved. Section 2.2 describes the architecture of Web services and provides details regarding the involved technologies. Section 2.3 details the main steps involved in offering and consuming a Web service. Section 2.4 offers information about the main properties of Web services.

2.1 Service-Oriented Architecture (SOA)

The Organization for the Advancement of Structured Information Standards (OASIS) [1] defines SOA as *a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains*[16].

According to The Open Group[2] in [7], *SOA is an architectural style that supports service orientation and is a paradigm for business and IT. This architectural style is for designing systems in terms of services available at an interface and the outcomes of services.*

[1] https://www.oasis-open.org/
[2] https://www.opengroup.org/

At this point it is important to highlight the concept of a service. According to [9] a service *is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities.* We can say that a service represents a set of activities which have specified outcomes, is self-contained, modular and is a "black box" to consumers of the service.

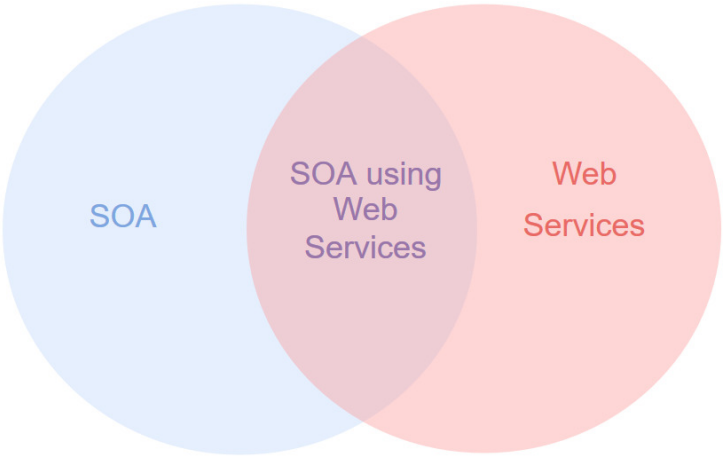

Figure 2.1: Relationship of SOA and Web Services (from [1]).

Web services can be used to implement SOA-based systems, but we need to be aware that this is not the only method. SOA can be implemented using any other service-based technology (e.g. middleware services such as Enterprise Service Bus (ESB), etc.) [1].

Roles

According to [6], we can identify the following roles in a Service-Oriented Architecture:

- **Service consumer**: can be an application, a software module, or another service; it initiates the enquiry of the service in the registry, binds to the service over a transport, andconsumes the functionality provided by the service according to the exposed interface/contract.
- **Service provider**: is a network-addressable entity that accepts and executes requests from consumers; it publishes its services and interface contract to the service registry so that the service consumer can discover and access the service.
- **Service registry**: is the enabler for service discovery; it contains a repository of available services and allows for the lookup of service provider interfaces to interested service consumers.

Operations

According to [6], the operations identified in a Service-Oriented Architecture are:

- **Publish**: a service description must be published so that it can be discovered and invoked by a service consumer.
- **Find**: a service consumer finds a service by querying the service registry for a service that meets its criteria.
- **Bind** and **invoke**: after retrieving the service description, the service consumer invokes the service according to the information in the service description.

Figure 2.3 provides a visual depiction of the roles and operations previously presented, as well as how these interact with each other.

2.2 Web Service Architecture

According to [3], we can define a Web service as *a software system designed to support interoperable machine-to-machine interaction over a network.*

The architecture of a Web service is comprised from the following elements, according to [3]: concepts, relationships and models.

A concept *is expected to have some correspondence with any realizations of the architecture* (e.g. *message concept identifies a class of object*).

The relationships *denote associations between concepts* (e.g. *a message has a message sender*).

A model is *a coherent subset of the architecture that typically revolves around a particular aspect of the overall architecture.*

- The **Message Oriented Model** *focuses on messages, message structure, message transport and so on — without particular reference as to the reasons for the messages, nor to their significance.*
- The **Service Oriented Model** elaborates *the concept of a service's owner — which has a real world responsibility for the service.*
- The **Resource Oriented Model** refers to the *resources that exist and have owners.*
- The **Policy Model** *focuses on policies that are put in place, or established, by those who have responsibility for the resource.*

As mentioned in [19], the Web service architecture is based on open technologies such as: eXtensible Markup Language (XML), Simple Object Access Protocol (SOAP), Universal Description, Discovery and Integration (UDDI), and Web Services Description Language (WSDL).

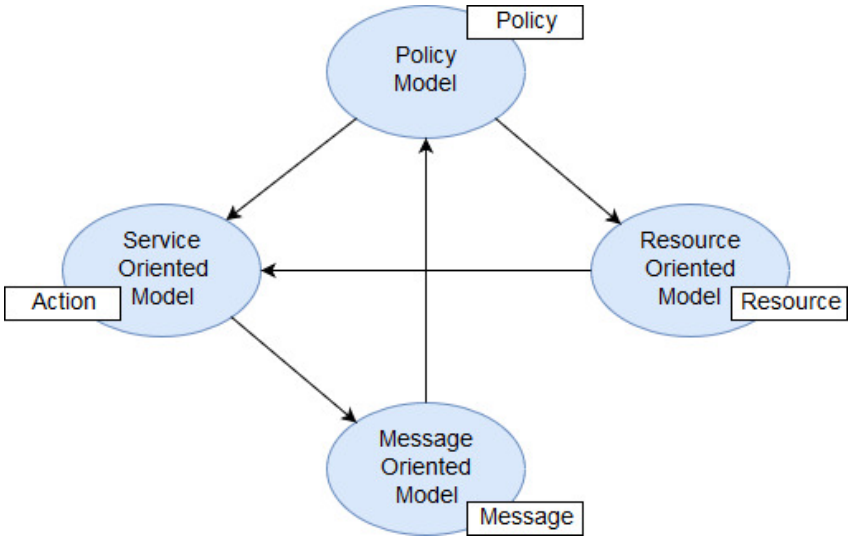

Figure 2.2: Meta Model of the architecture (from [3]).

According to [19], the involved *specifications are completely independent of programming language, operating system, and hardware to promote loose coupling between* the service consumer and provider.

2.3 Providing and Consuming a Web Service

If a consumer entity does not already know what service it wishes to engage with, then it must discover the appropriate provider entity for that particular task.

As mentioned by Douglas Barry, in [1], the following steps can be identified in providing and consuming a service:

1. A service provider describes its service using WSDL, which will then be published to a services registry.
2. A service consumer searches the services registry to locate a service.
3. Part of the WSDL offered by the service provider is passed to the service consumer. This informs the service consumer about the requests and responses types for the service provider.
4. The service consumer uses the WSDL to invoke the service provider.
5. The service provider provides the expected response to the service consumer.

In terms of steps 4 and 5, we observe that the request and subsequent response connections are defined in some way that is understandable to both the service consumer and the service provider.

According to W3C[3] in [3], we can summarize these steps in the following way: both the consumer and provider entities *become known to each other, agree on the semantics of the desired interaction, and exchange messages on behalf of their owners.*

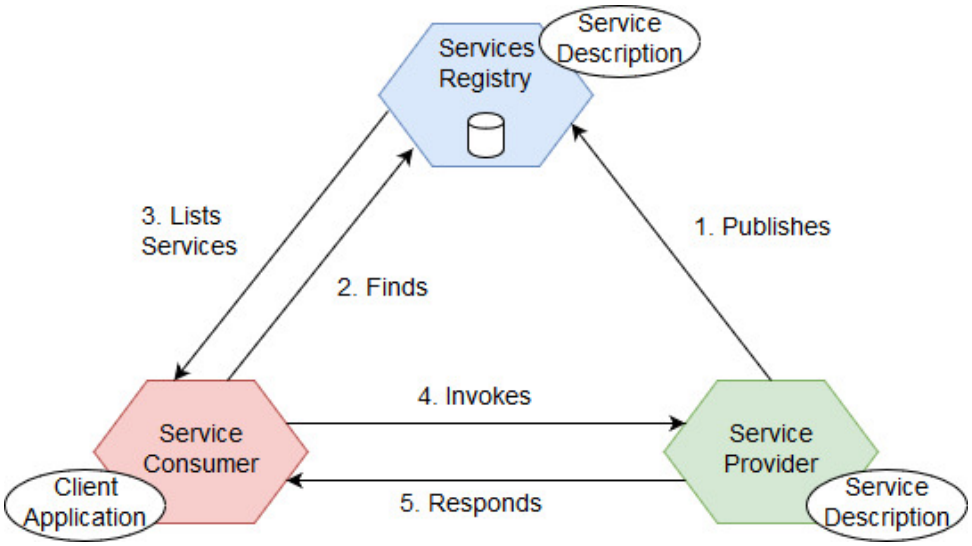

Figure 2.3: Overview of engaging a Web Service(adapted from [6]).

2.4 Properties

The main properties of a Web service identified by ITSO (part of IBM Global Technical Support[4]) as specified in [6] are:

- **self-contained**: on the client side, no additional software is required.
- **self-describing**: a WSDL file provides all of the necessary information to implement a Web service (as a provider) or to invoke a Web service (as a requester).
- can be **published, located, invoked** across the Web: with the help of standard technologies (for a better understanding see figure 2.3).
- **composable** and **modular**: simple Web services can be aggregated into more complex ones, can be chained together to perform higher-level business functions.
- **language independent** and **interoperable**: the interaction between a service provider and a service consumer is designed to be completely platform and language independent. Because the service provider and the service requester have no idea what platforms or languages the other is using, interoperability is a given.
- **inherently open** and **standards-based**: a large part of the Web service technology has been built using open-source projects, as mentioned in section 2.2.

[3] https://www.w3.org/
[4] https://www.ibm.com/support/home/

- **loosely coupled**: the service consumer only needs to know the external behavior of the service, not the details of its implementation.
- **programmatically accessible**
- **dynamic**: dynamic e-business can become a reality using Web services because their description and discovery can be automated (with UDDI and WSDL).

Summary

This chapter examind the open technologies on which Web services are based and how Web services are well suited for implementing a Service-Oriented Architecture. We exposed their defining properties: self-describing and modular applications that expose business logic as services that can be published, discovered, and accessed over the Internet.

Chapter 3

API Specification

Contents

Introduction

This chapter describes the concept of Web API. Section 3.1 provides a brief overview regarding the history of Web APIs, how they emerged and what directions they are headed. Section 3.2 exposes several existing standards that can be used for defining an API, with particular emphasis on OpenAPI Specification. Section 3.3 provides recommendations regarding how an API should be designed, built and documented.

3.1 Definitions, History, and Future Directions

According to Wikipedia[1], an application programming interface (API) is defined as *a set of routines, protocols, and tools for building software applications.*

The conceptual underpinnings of an Application Programming Interface are not a novelty, as Martin Bartlett points out:

[1] https://en.wikipedia.org/wiki/Application_programming_interface

The concept of an API pre-dates even the advent of personal computing, let alone the Web, by a very long time! The principal of a well documented set of publicly addressable "entry points" that allow an application to interact with another system has been an essential part of software development since the earliest days of utility data processing. However, the advent of distributed systems, and then the web itself, has seen the importance and utility of these same basic concepts increase dramatically.

—Martin Bartlett, from [15]

There are many different types of APIs: for operating systems, applications, websites, etc. Our main focus will be Web APIs. A Web API is a unique type of interface where the functionality is provided and can be accessed using the Internet and Web-specific protocols.

As for the evolution of Web APIs Specification Standards, and the main actors who have put their mark on this evolution, Kin Lane offers details in [15]. According to him, the pioneer of Web APIs Documentation Standards is the Wordnik [2] organization. In 2011 they created Swagger, a JSON format for describing the Wordnik API, and then built Swagger UI, an automatically generated, interactive documentation for the API. Their main reason for developing Swagger was to keep their documentation up to date. Through this process, they also set the stage for a new way to define and document our APIs.

Later on that year, API management provider Mashery[3] would copy Wordnik's approach, and launch their I/O Docs. Slowly the concept of API design would expand, with tech giants like Google establishing their own approach, which they called Google Discovery. By March 2013, Apiary[4] had launched their own API definition format called API Blueprint. By the end of 2013, we'd see another API definition language emerge, this time from enterprise technology provider Mulesoft[5], called RAML.

According to [13], APIs have reached their breakout moment for three reasons: process maturity, self-service, and technological maturity.

Wendell Santos, editor at ProgrammableWeb[6], has made an analysis of growth in Web APIs since 2015 until 2018. He mentions that the ProgrammableWeb directory passed the 19,000−API mark in January of 2018, with an average of more than 2,000 APIs added per year since 2014.

According to the analysis made by SmartBear[7], in 2019, it seems that the main challenges API teams want to solve in the years to follow are: standardization, versioning, and composability/multi-purpose reuse [22].

[2] https://developer.wordnik.com
[3] https://www.tibco.com/products/api-management
[4] https://apiary.io/
[5] https://www.mulesoft.com/
[6] https://www.programmableweb.com/
[7] https://smartbear.com/

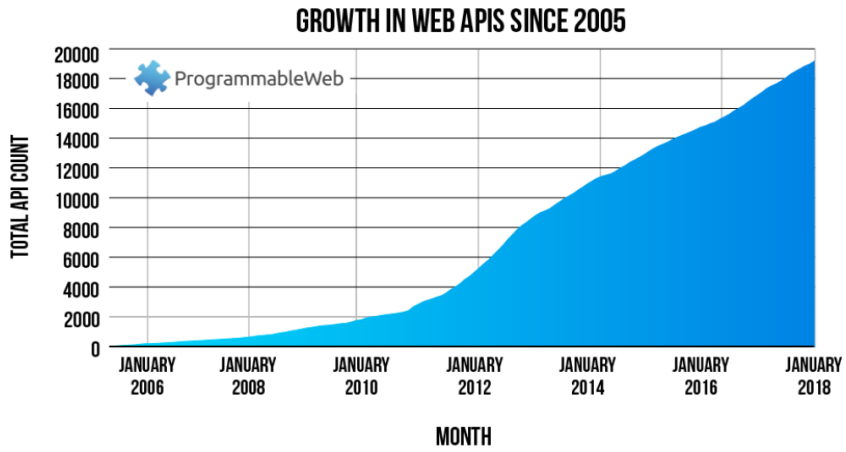

Figure 3.1: The growth over time of the ProgrammableWeb API directory (from https://www.programmableweb.com/).

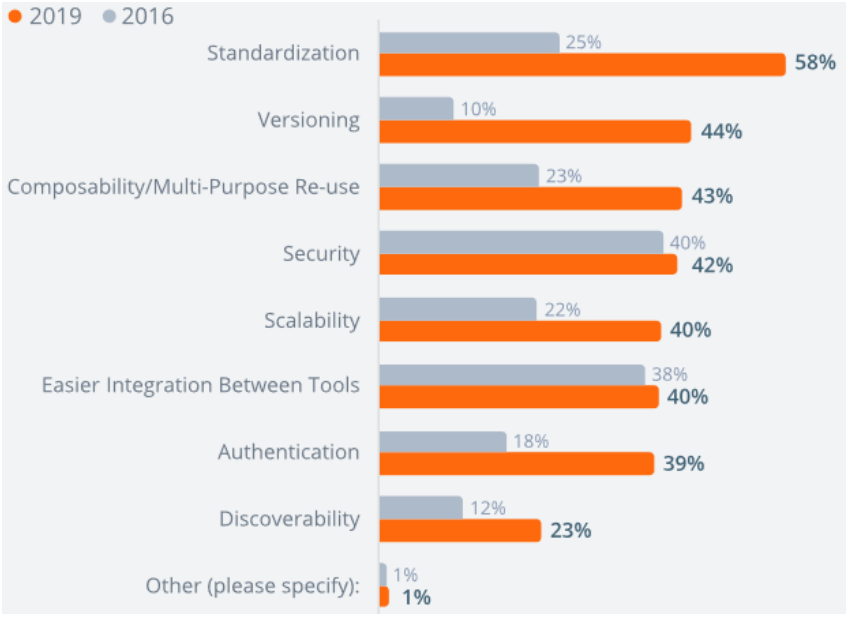

Figure 3.2: The current API challanges providers want to solve (from [22]).

In [14], the authors highlighted four directions in which the Web APIs are heading:

- **APIs as the basis of User Interfaces**: nowdays, the functionality is built as a set of APIs, and then the website is created as a client to those APIs.
- **Hypertext State Control**: Web APIs are *moving towards RESTfulness*. However, there is one part of REST *that is not commonly reflected in Web APIs: hypertext as the engine of application state* (HATEOAS). The main benefits of this approach are: reliability, evolution, and reuse.
- **Web API Descriptions**: *machine-readable standard API/interface description is one of the best aspects of the Web Services approach.* However, in the RESTful space, there has been pushback against description languages because, in theory, the Web already has mechanisms for description.
- **Automation**: *the Web is becoming a large repository of open data, available for rich exploratory querying, machine processing such as generating visualizations, and for combining multiple data sources [...] Services have been looking at how to represent service descriptions in RDF while working with the lightweight service description approaches common with Web APIs.*

3.2 API Specification Standards

We'll detail some of the most used formats for defining an API, with emphasis on OpenAPI Specification.

3.2.1 RESTful API Modeling Language: RAML

RAML[8] is built on broadly-used standards such as YAML and JSON and is a non-proprietary, vendor-neutral open specification. The RAML specification provides mechanisms for defining "practically-RESTful" APIs, creating client/server source code, and comprehensively documenting the APIs for users.

3.2.2 API Blueprint

API Blueprint[9] is a documentation oriented API description language. According to [30], an API Blueprint document *is a plain text Markdown document describing a Web API in whole or in part. The document is structured into logical sections. Each section has its distinctive meaning, content and position in the document.*

[8] https://raml.org/
[9] https://apiblueprint.org/

3.2.3 Web Application Description Language: WADL

According to W3C, in [17], *WADL is designed to provide a machine process-able description of such HTTP-based Web applications.* WADL is the REST equivalent of SOAP's Web Services Description Language (WSDL), which can also be used to describe REST web services.

In [20], WADL is seen as a less appealing alternative solution to those presentend in this section because *is incredibly time consuming to create descriptions with, and the linking methodology leaves much to be desired.*

3.2.4 OpenAPI Specification: OAS

The OpenAPI Specification (previously Swagger) is a community-driven open specification within the OpenAPI Initiative[10], a Linux Foundation Collaborative Project.

As mentioned in [23], the OpenAPI Specification (OAS) defines *a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection.*

Of the many API documentation and specification formats, OpenAPI Specification is certainly one of the most popular as we can see in figure 3.3.

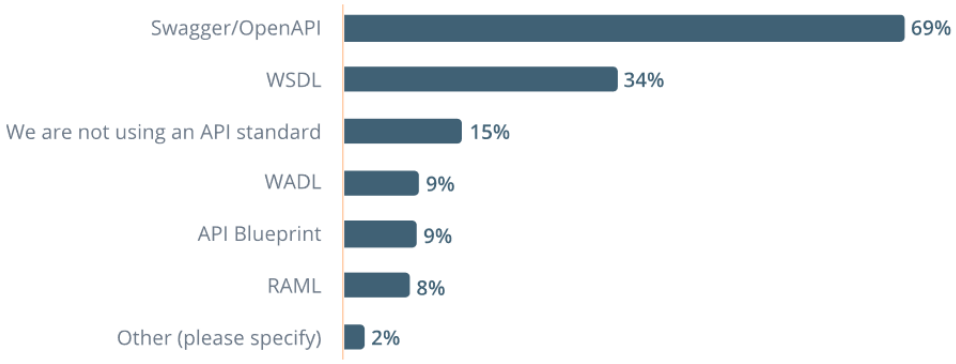

Figure 3.3: Adoption rates of common standards for defining APIs (from [22]).

The main advantages of OpenAPI, according to [20], are: is heavily adopted, has a large community of users and supporters, and greater support for multiple languages. It is important to note, however, that it lacks advanced constructs for metadata.

Keshav Vasudevan, Product Manager at SmartBear, describes in [28] the general outline of an OAS defined API using OAS 3.0:

- **info**: *contains the metadata associated with the API's contract* (e.g. title, version, and description of the API).

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

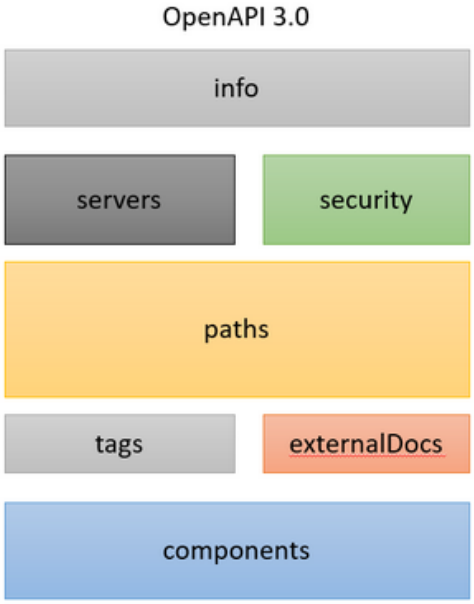

Figure 3.4: Sections in an API contract designed using the OAS 3.0 (from [28]).

- **servers**: the Server object provides information regarding the location of API servers.
- **security**: *supports various authentication and authorization schemes to mitigate unknown, unregistered users from accessing the API* (e.g. OAuth2, HTTP authentication schemes).
- **paths**: there are four types of parameters you can specify using the OAS 3.0: path parameters, query parameters, header parameters, and cookie parameters.
- **external docs**: *OAS 3.0 allows you to reference external documentation.*
- **tags**: they *allow consumers of the API to better segment and identify what they want use the API for.*

In listing 3.1 we provide an API contract example in yaml format, created using Swagger Editor[11] and OAS 2.0, realised for a project within the Web Technologies[12] course.

Listing 3.1: TREX API OpenAPI Specification.

```
1 swagger: "2.0"
2 info:
3   description: "Trex description"
4   version: "1.0.0"
5   title: "Trex"
6 host: localhost
7 basePath: /trex/api
```

[11] https://editor.swagger.io/
[12] https://profs.info.uaic.ro/~busaco/teach/courses/web/

```
tags:
- name: "books"
  description: "Search for books"
schemes:
- "http"
paths:
  /books:
    get:
      tags:
      - "books"
      summary: "Search for books"
      description: "Search for books"
      operationId: "getBooks"
      consumes:
      - "*/*"
      produces:
      - "application/json"
      parameters:
        - in: "query"
          name: "terms"
          type: string
          required: true
          description: "The keywords to search for"
        - in: "query"
          name: "language"
          type: string
          description: "The language of the books"
        - in: "query"
          name: "minimumRating"
          type: number
          description: "The minimum rating of the books"
        - in: "query"
          name: "from"
          type: number
          description: "The starting year of the books"
        - in: "query"
          name: "to"
          type: number
          description: "The final year of the books"
      responses:
        200:
          description: "successful operation"
          schema:
             $ref: "#/definitions/BooksResponse"
        400:
          description: "Invalid status value"
        405:
```

```
55            description: "Invalid input"
56  definitions:
57      Resource:
58          type: "object"
59          properties:
60              type:
61                  type: "string"
62              title:
63                  type: "string"
64              description:
65                  type: "string"
66              authors:
67                  type: "array"
68                  items:
69                      type: "object"
70              image:
71                  type: "string"
72              tags:
73                  type: "array"
74                  items:
75                      type: "object"
76              rating:
77                  type: "number"
78              date:
79                  type: "string"
80              language:
81                  type: "string"
82              url:
83                  type: "string"
84      BooksResponse:
85          type: "object"
86          properties:
87              totalItems:
88                  type: "number"
89              books:
90                  type: "array"
91                  items:
92                      $ref: "#/definitions/Resource"
93  externalDocs:
94    description: "Find out more about Swagger"
95    url: "http://swagger.io"
```

In [5], in the chapter inspired by Arnaud Lauret's session at the 2016 Platform Summit, there are some important features that are found within OpenAPI:

- **Proper design and approach**: *Any good engineer can tell you that structures don't fail because of the last brick, they fail because of the first. If you start with a*

shaky foundation, or bad bedrock, you're going to collapse, and the same is true of an API. There are three methods in which the specification generation is accomplished: Codegen, automatic generation, and manually.

- **Complete documentation and description**: is essential to be able to easily and efficiently describe an API in a simple, well structured document. *SwaggerUI, an element of OpenAPI, is supremely powerful and capable when it comes to generating descriptive documentation.*
- **Rapid testing and iteration**: during development is essential to have the means to actively test and modify API content in an enclosed environment. In this way *endpoints can be tested against any permutation, resource access can be tested in a multitude of environments, and even the API can be tested in interactions with other APIs. The OpenAPI Specification utilizes a declarative resource specification, and as part of this, allows for easy exploration of the API without access to the server code or an understanding of the server implementation. This is all done via that magic implementation called the SwaggerUI. SwaggerUI reads the API description that is stated as part of the OpenAPI Specification, and renders it as a web-page.*
- **Machine and human readability and translation**: *by functioning as a specification which describes the API, the OpenAPI Specification can then derive both machine and human readable content in a better, more complete way.*

3.3 Best Practices for API Design

According to [13], API publishers should be thinking about what users want from their business assets and how they can provide access to jumpstart the API economy.

Asked about the best technical practices from the Tumblr API[13], Derek Gottfrid, Director of Product at Tumblr, offered this answer: *We want our API to be easy to learn without documentation – the way we lay out our URIs, you should just be able to drop into a command live* [13].

We will present some of the recommendations that should be taken into consideration when designing and building an API, as suggested in [13]:

- **Differentiated API**: the API developer should take into consideration the following questions: Why should a developer use a particular API? How is it different from other APIs that are available on the market? Why should they use it?

  A particular API can be differentiated through any of the following means: providing data that is unique, more comprehensive or accurate than that of competitors, by offering better support, by designing an API that is more reliable or faster than alternatives, or even by offering more data traffic for free.

[13] https://www.programmableweb.com/api/tumblr

- **APIs that are easy to try and use**: if the users of a particular API can't make almost immediate progress, they'll go elsewhere, so the key to success is to remove all barriers of entry in using the API.

  There are a few ways to do this. *For private APIs, it is important to demonstrate real value in running a system* (e.g. the API could offer *much more flexible access to content, better overall performance*). *For public APIs, one approach is to offer some level of free access: many developers won't even consider using a particular API if there are only paid options* because *they may not know yet if the API meets their needs.*

- **API understandability**: an example of an intuitively designed API is the Facebook Graph API[14].

  Kin Lane, developer evangelist at Mimeo, gives the following advices: *Don't try to make it too complex. Focus on the core building blocks for an API. Do one thing, do it really well, and bundle it with simple documentation and code samples.*

- **Avoidance of cumbersome approaches**: this is especially important in the area of security. *Many APIs offer custom or complex security schemes that require developers to learn them from scratch.* Instead, the API developers should think about using standardised specifications such as OAuth or other commonly understood security schemes.

- **Less is more**: the API developer should *start with the absolute minimum amount of functionality, and then expand it slowly over time, as feedback is collected*; the reason for doing this is because the API might evolve in a different direction than what was anticipated.

Summary

This chapter examined how Web APIs are about delivering valuable, meaningful, scalable and distributed resources across the World Wide Web. It also explored the emerging formats that use JSON, Markdown, and YAML to define an API, providing a simple, machine readable blueprint, that API providers and consumers can use a single truth for all API interactions. It explained why OpenAPI Specification provides perhaps one of the strongest foundations upon which one can build an API: not by solely providing tools or approaches, but by providing a proper design approach.

[14] https://developers.facebook.com/docs/graph-api/

Chapter 4

Knowledge Modeling and Representation

Contents

Introduction

The goal of this chapter is to introduce the foundations of Semantic Web technologies. Section 4.1 provides details on how the Semantic Web evolved. Section 4.2 introduces the Resource Description Framework (RDF). Section 4.3 describes RDF Schema, a semantic extension of RDF. Section 4.4 introduces OWL 2, a powerful modeling language for the Semantic Web. Section 4.5 offers a short description of SPARQL, along with query examples.

4.1 Web of Data

From the beginning, the development of Semantic Web technologies has been strongly correlated with the World Wide Web. Not surprisingly, as the WWW designer, Sir Tim Berners-Lee[1], has also introduced the Semantic Web concept and encouraged further research in this area in [2]. In 2001, Sir Tim Berners-Lee, along with two other researchers,

[1] https://en.wikipedia.org/wiki/Tim_Berners-Lee

Ora Lassila and James Hendler, wanted to give the world a new Web called the Semantic Web, in which content would be meaningful to software programs (they would have semantics), allowing programs to interact with the Web the same way that people do [27].

The Semantic Web has been designed as a WWW extension that allows computing tools to search, combine and process content that is based on the meaning it has for us. Because we do not yet have access to human-like artificial intelligence, the only way we can achieve this goal is to specify the semantics of Web resources in a format that can be processed by the computer [12].

According to [29], *the Semantic Web is a Web of Data – of dates and titles and part numbers and chemical properties and any other data one might conceive of. The collection of Semantic Web technologies (e.g. RDF, OWL,SPARQL) provides an environment where application can query that data, draw inferences using vocabularies, etc.*

For the Web of Data to become a reality, it is important to have the huge amount of content on the Web available in a standard format. Also, relationships among data should be made available. *This collection of interrelated datasets on the Web can also be referred to as Linked Data* [29]. To achieve and create Linked Data, technologies should be available in a common format (RDF), so that conversions or on-the-fly access to databases can be made easily.

W3C[2] offers a wealth of technologies (e.g. RDF, RDFS, OWL, SPARQL) to get access to the data, on which we will provide more details in the following sections.

4.2 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a framework for expressing information about resources. Resources can be anything, including documents, people, physical objects, and abstract concepts [21].

Some of the features that make RDF useful are: adds machine-readable information to Web pages, enriches a dataset by linking it to third-party datasets, builds aggregations of data around specific topics, and provides a standards-compliant way for exchanging data between databases [21].

RDF allows us to make assertions or statements about resources in a simple format with the following structure:

<subject> <predicate> <object>

Through such a statement, the relationship between two resources is expressed. The two related resources are represented by the **subject** and the **object**, and the nature of the relationship is denoted by the **predicate**. The link is unidirectional (from subject to

[2] https://www.w3.org/

object) and is called **property**. *Because RDF statements consist of three elements they are called triples* [21].

Table 4.1 offers a set of triple examples in pseudocode regarding the Earth.

Table 4.1: Triple examples in pseudocode regarding Earth.

Subject	**Property**	**Object**
Earth	is a	Planet
Earth	has age	4.5 billion years
Ioana	lives on	Earth
Solar System	includes	Earth

The triples can also be viewed as a graph in which the subjects and objects are represented as nodes, while the predicates form the arcs. Figure 4.1 shows the graph resulting from the triples mentioned in table 4.1.

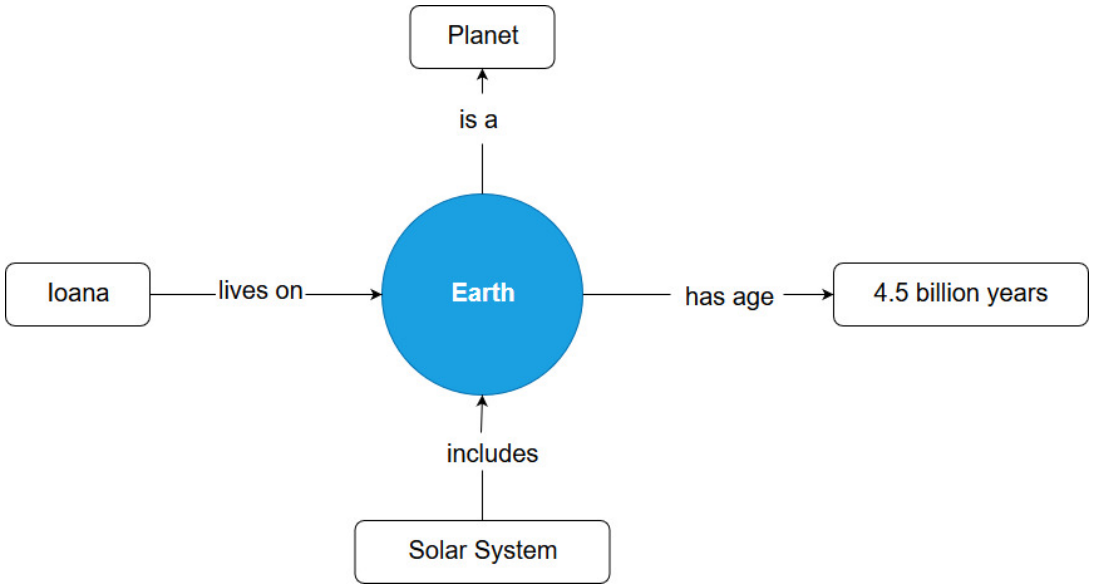

Figure 4.1: The RDF Graph of the Earth example.

While this ways of representing information about resources is easy to read for humans, it is clearly not suitable for processing it in computer systems. To translate this information into a format that can be processed by machines, there exist a number of different serialization formats for writing down RDF graphs: Turtle family of RDF languages, JSON-LD, RDFa, RDF/XML etc. What is important to remember is that different ways of rewriting the same graph lead to the same triplets, so they are logically equivalent.

The graph from figure 4.1 is represented in Turtle syntax in listing 4.1. Protégé[3] was used to define the Earth ontology.

[3] https://protege.stanford.edu/

Listing 4.1: Example using turtle syntax generated using Protégé 5.5.0.

```
@prefix : <http://www.semanticweb.org/ami/ontologies/earth-ontology#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix dbp: <http://dbpedia.org/property/> .
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://www.semanticweb.org/ami/ontologies/earth-ontology> .

<http://www.semanticweb.org/ami/ontologies/earth-ontology> rdf:type owl:Ontology
    .

dbo:meanRadius rdf:type owl:AnnotationProperty .
dbp:materials rdf:type owl:AnnotationProperty .

:hasAge rdf:type owl:ObjectProperty ;
        rdfs:domain dbo:Agent ;
        rdfs:range dbr:Planet .

:includes rdf:type owl:ObjectProperty .
:livesOn rdf:type owl:ObjectProperty .

dbo:Agent rdf:type owl:Class .
dbr:Planet rdf:type owl:Class .

dbr:Earth rdf:type owl:NamedIndividual ,
                   dbr:Planet ;
                   dbo:meanRadius 6371 ;
                   dbp:materials dbr:Aluminium ,
                                 dbr:Iron ,
                                 dbr:Magnesium ,
                                 dbr:Nickel ;
                   :hasAge "4.5 billion years"@en .

dbr:Metal rdf:type owl:NamedIndividual .
dbr:Planet rdf:type owl:NamedIndividual .

dbr:Solar_System rdf:type owl:NamedIndividual ;
                 :includes dbr:Earth .

:Ioana rdf:type owl:NamedIndividual ;
       :livesOn dbr:Earth .
```

4.3 RDF Schema (RDFS)

As we mentioned in the previous section, the *RDF data model provides a way to make statements about resources*. But, it *does not make any assumptions about what resources stand for*. To solve this problem, *RDF is typically used in combination with vocabularies or other conventions that provide semantic information about these resources* [21].

RDF Schema provides a data-modelling vocabulary for RDF data, [it] is a semantic extension of RDF. It provides mechanisms for describing groups of related resources and the relationships between these resources[...]The RDF Schema class and property system is similar to the type systems of object-oriented programming languages. RDF Schema differs from many such systems in that instead of defining a class in terms of the properties its instances may have, [it] describes properties in terms of the classes of resource to which they apply [4].

Table 4.2 summarizes the main modeling constructs provided by RDF Schema.

Construct	Syntactic form	Description
Class (a class)	**C** `rdf:type rdfs:Class`	**C** (a resource) is an RDF class
Property (a class)	**P** `rdf:type rdf:Property`	**P** (a resource) is an RDF property
type (a property)	**I** `rdf:type` **C**	**I** (a resource) is an instance of **C** (a class)
subClassOf (a property)	**C1** `rdfs:subClassOf` **C2**	**C1** (a class) is a subclass of **C2** (a class)
subPropertyOf (a property)	**P1** `rdfs:subPropertyOf` **P2**	**P1** (a property) is a sub-property of **P2** (a property)
domain (a property)	**P** `rdfs:domain` **C**	domain of **P** (a property) is **C** (a class)
range (a property)	**P** `rdfs:range` **C**	range of **P** (a property) is **C** (a class)

Figure 4.2: RDF Schema Constructs (from [21]).

4.4 OWL 2 Web Ontology Language (OWL 2)

OWL 2 Web Ontology Language (OWL) is a *Semantic Web language designed to represent rich and complex knowledge about things, groups of things, and relations between things[...]OWL 2 is a language for expressing ontologies. An ontology is a set of precise descriptive statements about some part of the world (usually referred to as the domain of interest or the subject matter of the ontology)* [11].

According to [11], to precisely describe a domain of interest, it is helpful to come up with a set of terms called vocabulary to eliminate ambiguities that may exist on the terms used in the different data sets. An ontology contains two categories of knowledge:

- **terminological knowledge**: provides the general context (vocabulary together with interrelation information).
- **assertional knowledge**: provides the concrete objects from our domain of interest.

4.5 SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL *is a set of specifications that provide languages and protocols to query and manipulate RDF graph content on the Web or in an RDF store* [8].

The results of SELECT queries consist of mappings from variables to RDF terms. To serialize these results, SPARQL supports four exchange formats: XML, JSON, CSV, and TSV.

SPARQL contains capabilities for querying graph patterns. It also supports aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries based on RDF graph [10]. Following, listings 4.2 and 4.3 offers some SPARQL query examples on the RDF graph for the Earth example.

Listing 4.2: SPARQL query that finds all persons that live on Earth.

```
1  PREFIX : <http://www.semanticweb.org/ami/ontologies/earth-ontology#>
2  PREFIX dbo: <http://dbpedia.org/ontology/>
3  PREFIX dbp: <http://dbpedia.org/property/>
4  PREFIX dbr: <http://dbpedia.org/resource/>
5  PREFIX owl: <http://www.w3.org/2002/07/owl#>
6  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
7  PREFIX xml: <http://www.w3.org/XML/1998/namespace>
8  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
9  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
10
11 SELECT ?earthling
12 WHERE { ?earthling :livesOn dbr:Earth . }
```

Listing 4.3: SPARQL query that finds all the materials from which Earth is made.

```
1  PREFIX : <http://www.semanticweb.org/ami/ontologies/earth-ontology#>
2  PREFIX dbo: <http://dbpedia.org/ontology/>
3  PREFIX dbp: <http://dbpedia.org/property/>
4  PREFIX dbr: <http://dbpedia.org/resource/>
5  PREFIX owl: <http://www.w3.org/2002/07/owl#>
6  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
7  PREFIX xml: <http://www.w3.org/XML/1998/namespace>
8  PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
9  PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
10
11 SELECT ?material
12 WHERE { dbr:Earth dbp:materials ?material . }
```

Summary

This chapter examined the main components of the Semantic Web. It introduced Resource Description Framework (RDF), a framework for describing general relationships between objects of interest. We discussed how RDF is extended with RDF Schema (RDFS), allowing us to add expressive features that go beyond the description of simple data. It also introduced OWL 2, a standard for the modeling of ontologies. Using SPARQL, consumers of the Web of Data can extract possibly complex information (e.g. existing resource, references, and their relationships).

Chapter 5

Augmenting OpenAPI Specification With Knowledge

Contents

Introduction

This chapter describes the process of augmenting the OpenAPI specification with semantics. Section 5.1 provides an overview of the technologies used for development and the open source projects that we have contributed to in order to achieve this goal. Section 5.2 proposes a set of extensions that can be used to augment the OpenAPI specification with knowledge and provides details regarding their intended use. Section 5.3 presents the functionality added to the Swagger Editor project along with the relevant code snippets. Section 5.4 exposes the functionality added to the Swagger UI project along with the relevant code snippets. Section 5.5 discusses future directions that can further improve the process of Web Api Semantic augmentation.

5.1 Technologies

In order to augment the OpenAPI specification with knowledge we have used the following technologies for development:

1. **Node.js**[1]: *is a JavaScript runtime built on Chrome's V8 JavaScript engine.* The version used is 10.15.3.

2. **Npm**[2]: is a package manager for the JavaScript programming language that consists of three distinct components: the website, the CLI , and the registry. The version used is 6.4.1.

3. **React**[3]: *is a JavaScript library for building user interfaces.* Its main features are the following: one-way data binding with props, stateful component Virtual DOM, lifecycle methods (e.g. shouldComponentUpdate, componentDidMount, componentWillUnmount, render, etc.), and JSX, an extension to the JavaScript language syntax.

As for the projects on which we have contributed, they are **Swagger Editor**[4] and **Swagger UI**[5], which were forked from their repositories on GitHub.

According to SmartBear[6] in [24], *The Swagger Editor is an open source editor to design, define and document RESTful APIs in the Swagger Specification.* The Swagger Editor software version augmented with Semantic support is available at: https://github.com/ioanabirsan/swagger-editor/.

Also, SmartBear offers in [25] the following description for Swagger UI: *Swagger UI is a tool that allows anyone to visualize and interact with the API's resources without having any of the implementation logic in place. It's automatically generated from your OpenAPI Specification, with the visual documentation making it easy for back end implementation and client side consumption.* The Swagger UI software version augmented with Semantic support is available at: https://github.com/ioanabirsan/swagger-ui/.

5.2 OpenAPI Extension Support

According to [26], *extensions are custom properties that start with x-* (e.g. x-<custom-property-name>). *They can be used to describe extra functionality that is not covered by the standard OpenAPI Specification.*

We have added support for the following extentions with the purpose to augment the OpenApi specification with Semantic support: **x-same-as** and **x-rdf-type**. The custom properties have been added both at schema and properties level in the case of OAS3, respectively at definitions and properties level for OAS2. The values supported by both extensions are of type string and represent concepts defined in the Schema.org[7] vocabulary:

[1] https://nodejs.org/en/
[2] https://www.npmjs.com/
[3] https://reactjs.org/
[4] https://swagger.io/tools/swagger-editor/
[5] https://swagger.io/tools/swagger-ui/
[6] https://smartbear.com/
[7] http://schema.org/

- The schemas/definitions-level supported value is equivalent to rdfs:Class;
- The properties-level supported value is equivalent to rdf:Property.

Listings 5.1 and 5.2 provide an example of augmenting the OpenAPI specification with Semantic support using the **x-same-as** extension.

Listing 5.1: Example of expansion of OAS3 with semantics.

```
components:
  schemas:
    Order:
      type: object
      x-same-as: 'http://schema.org/Order'
      properties:
        id:
          type: integer
          x-same-as: 'http://schema.org/identifier'
          format: int64
```

Listing 5.2: Example of expansion of OAS2 with semantics.

```
definitions:
  Order:
    type: object
    x-same-as: 'http://schema.org/Order'
    properties:
      id:
        type: integer
        x-same-as: 'http://schema.org/identifier'
        format: int64
```

5.3 Extending Swagger Editor With Semantic Support

The following features have been added to Swagger Editor:

- Suggestions and autocomplete support for the newly added extensions;
- Contextual suggestions and autocomplete support for concepts retrieved from *Schema.org*, depending on the location where the extension is used (i.e. at the Schema/-Model or Property level);
- Ability to convert and save an OpenAPI definition in Turtle (TTL)[8] format.

The entire implementation that augments Swagger Editor with Semantic support can be consulted at: https://github.com/ioanabirsan/swagger-editor/.

The structure of the Swagger Editor project is as follows:

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

```
...
src
    plugins
        ast
        editor
            components
            editor-helpers
            editor-plugins
        editor-autosuggest
        editor-autosuggest-keywords
        editor-autosuggest-oas3-keywords
        editor-autosuggest-refs
        editor-autosuggest-snippets
        editor-metadata
        json-schema-validator
        jump-to-path
        local-storage
        performance
        split-pane-mode
        validate-base
        validate-semantic
    standalone
        styles
        topbar
        topbar-insert
        topbar-menu-edit-convert
        topbar-menu-file-import
...

```

The relevant modules that can be identified in the src folder within Swagger Editor project are the following: *plugins* and *standalone*. The *plugins* module contains submodules responsible for providing suggestions, support for autocompletions, validations, etc. The *standalone* submodules are primarily responsible for the layout and functionality of the menu bar.

The support added for the above-mentioned features to Swagger Editor in order to extend it with Semantic support has been included in the following submodules: *editor-helpers*, *editor-autosuggest-keywords*, *editor-autosuggest-oas3-keywords*, and *topbar*.

The rest of the section provides a schematic overview of the work that was carried in order to implement the support for these features in Swagger Editor; we will also present key code snippets that detail the most important parts of the implementation.

Listing 5.3 presents the class responsible for implementing the functionality of extracting the identifier for both classes and properties from Schema.org. The class identifier is further used to provide suggestion and autocomplete support for the proposed extensions

at schemas/definitions levels of the OpenAPI Specification, while the property identifier is used for the same purpose at the properties level of the OpenAPI Specification.

The *schema-org-extractor.js* file that contains the SchemaOrgExtractor class was added within the Swagger Editor project in the *editor-helpers* submodule.

Listing 5.3: Code from SchemaOrgExtractor class.

```
export default class SchemaOrgExtractor {
 static getClasses () {
   let concepts = schemaOrg["@graph"]
   return concepts.filter(concept => concept["@type"] === "rdfs:Class").map(
    concept => concept["@id"])
 }

 static getProperties () {
   let concepts = schemaOrg["@graph"]
   return concepts.filter(concept => concept["@type"] === "rdf:Property").map(
    concept => concept["@id"])
 }
}
```

Listing 5.4 presents the key business logic added to provide autocomplete support for the **x-same-as** and **x-rdf-type** extensions and their accepted values for OpenAPI specification version 2 (OAS2). SchemaOrgExtractor class methods are called for each extension depending on the level at which it was placed.

The *keyword-map.js* file in which we added this functionality is found within the Swagger Editor project in the *editor-autosuggest-keywords* submodule.

Listing 5.4: Code snippet from keyword-map.js.

```
var schemaOrgClasses = SchemaOrgExtractor.getClasses()
var schemaOrgProperties = SchemaOrgExtractor.getProperties()

var properties = {
 ...
 "x-same-as": schemaOrgProperties,
 "x-rdf-type": schemaOrgProperties,
 get properties () {
   return {
     ".": this
   }
 }
}


var schema = {
 ...
```

```
 "x-same-as": schemaOrgClasses,
 "x-rdf-type": schemaOrgClasses,
 get properties () {
   return {
     ".": properties
   }
 }
}
```

Similarly, listing 5.5 presents the key business logic added to provide autocomplete support for the **x-same-as** and **x-rdf-type** extensions and their accepted values for OpenAPI specification version 3 (OAS3).

The *oas3-objects.js* file in which we added this functionality is found within the Swagger Editor project in the *editor-autosuggest-oas3-keywords* submodule.

Listing 5.5: Code snippet from oas3-objects.js.

```
var schemaOrgClasses = SchemaOrgExtractor.getClasses()
var schemaOrgProperties = SchemaOrgExtractor.getProperties()

export const Properties = {
 ...
 "x-same-as": schemaOrgProperties,
 "x-rdf-type": schemaOrgProperties,
 get properties () {
   return {
     ".": this,
   }
 }
}

export const Schema = {
 ...
 "x-same-as": schemaOrgClasses,
 "x-rdf-type": schemaOrgClasses,
 get properties () {
   return {
     ".": Properties,
   }
 }
}
```

Listing 5.6 presents the class responsible for implementing the functionality of converting and saving the OpenAPI definition from JSON/YAML format to Turtle format.

The *TTL.js* file that contains the code for the TTL class was added within the Swagger Editor project in the *editor-helpers* submodule.

Listing 5.6: Code from TTL class

```
1  import YAML from "@kyleshockey/js-yaml"
2
3  let rdfKeywords = {"x-rdf-type": "rdf:type", "x-same-as": "owl:sameAs"}
4
5  export default class TTL {
6   static convertToTurtle (jsonOrYaml, isOAS3) {
7     // JSON or YAML String -> JS object
8     let jsContent = YAML.safeLoad(jsonOrYaml)
9     let ontologyNamespace = jsContent["info"]["title"] || "swagger-schema"
10    ontologyNamespace = ontologyNamespace.replace(/ /g,"_")
11    return `${TTL.getOntologyHeader(ontologyNamespace)} \n\n${TTL.jsonToTurtle(
      jsContent, isOAS3)}`
12  }
13
14  static getOntologyHeader (ontologyNamespace) {
15    let header =
16      `@prefix : <http://www.semanticweb.org/ontologies/${ontologyNamespace}/> .\
      n` +
17      "@prefix dc: <http://purl.org/dc/elements/1.1/> .\n" +
18      "@prefix gr: <http://purl.org/goodrelations/v1#> .\n" +
19      "@prefix dbo: <http://dbpedia.org/ontology/> .\n" +
20      "@prefix dbp: <http://dbpedia.org/property/> .\n" +
21      "@prefix dbr: <http://dbpedia.org/resource/> .\n" +
22      "@prefix owl: <http://www.w3.org/2002/07/owl#> .\n" +
23      "@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .\n" +
24      "@prefix xml: <http://www.w3.org/XML/1998/namespace> .\n" +
25      "@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .\n" +
26      "@prefix muto: <http://purl.org/muto/core#> .\n" +
27      "@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .\n" +
28      "@prefix schema: <http://schema.org/> .\n" +
29      "@prefix sioc: <http://rdfs.org/sioc/ns#> .\n" +
30      "@prefix yago: <http://dbpedia.org/class/yago/> .\n" +
31      "@prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#> .\n" +
32      `@base <http://www.semanticweb.org/ontologies/${ontologyNamespace}/> .`
33
34    return header
35  }
36
37  static jsonToTurtle (swaggerSchema, isOAS3) {
38    let rdfTriplesInTurtleSyntax = ""
39    let definitions
40
41    if (isOAS3) {
42      definitions = swaggerSchema["components"]["schemas"]
43    } else {
```

```
44      definitions = swaggerSchema["definitions"]
45    }
46
47    for (let definitionKey in definitions) {
48      let definition = definitions[definitionKey]
49      for (let rdfKey in rdfKeywords) {
50        if (rdfKey in definition) {
51          let rdfKeyword = rdfKeywords[rdfKey]
52          let rdfTriple = `:${definitionKey} ${rdfKeyword} <${definition[rdfKey]}
      > . \n`
53          rdfTriplesInTurtleSyntax += rdfTriple
54        }
55      }
56
57      let properties = definition["properties"]
58      for (let propertyKey in properties) {
59        let property = properties[propertyKey]
60        for (let rdfKey in rdfKeywords) {
61          if (rdfKey in property) {
62            let rdfKeyword = rdfKeywords[rdfKey]
63            let rdfTriple = `:${propertyKey} ${rdfKeyword} <${property[rdfKey]}>
      . \n`
64            let rdfRangeTriple = `:${propertyKey} rdfs:range :${definitionKey} .
      \n`
65            rdfTriplesInTurtleSyntax += rdfTriple
66            rdfTriplesInTurtleSyntax += rdfRangeTriple
67          }
68        }
69      }
70
71      rdfTriplesInTurtleSyntax += `\n`
72    }
73
74    return rdfTriplesInTurtleSyntax
75  }
76 }
```

In listing 5.7, we have the code snippet that calls the TTL class functionality. The *topbar.jsx* file in which we added this functionality is found within the Swagger Editor project in the *topbar* submodule.

Listing 5.7: Code snippet from topbar.jsx

```
1 saveAsTurtle = () => {
2  let editorContent = this.props.specSelectors.specStr()
3  let fileName = this.getFileName()
4
5  if(this.hasParserErrors()) {
```

```
    // we can't recover from a parser error in save as JSON
    // because we are always parsing so we can beautify
    return alert("Save as Turtle is not currently possible because Swagger-Editor
      wasn't able to parse your API definiton.")
  }

  let isOas3 = this.props.specSelectors.isOAS3()
  let rdfTriplesInTurtleSyntax = TTL.convertToTurtle(editorContent, isOas3)
  this.downloadFile(rdfTriplesInTurtleSyntax, `${fileName}.ttl`)
}
```

5.4 Extending Swagger UI With Semantic Support

We have added support for the following features to Swagger UI in order to augment it with Semantic support:

- Add Structured Data to the generated OpenAPI definition documentation using Microdata[9];
- Retrieve the existing description on Schema.org for each class and property that has been augmented with semantics using the **x-same-as** and **x-rdf-type** extensions and include it in the generated OpenAPI definition documentation;
- Associate the elements from the OpenAPI definition with concepts from the Schema.org vocabulary based on the usage of the previously defined extensions.

According to [18], *Microdata provides a simple mechanism to label content in a document, so it can be processed as a set of items described by name-value pairs. Microdata can be used to provide machine-parseable information about content that is processed by tools to improve accessibility.*

The entire implementation that augments Swagger UI with Semantic support can be consulted at: https://github.com/ioanabirsan/swagger-ui/.

The structure of the Swagger UI project is as follows:

```
    ...
    src
        core
            components
            containers
            plugins
            presets
        img
        plugins
        standalone
```

[9] https://developer.mozilla.org/en-US/docs/Web/HTML/Microdata

```
11          style
12      ...
13
```

The relevant module that can be identified in the src folder within Swagger UI project is the *core* module. It consists of four submodules: *components*, *containers*, *plugins* and *presets*. The support added for the above-mentioned features to Swagger UI in order to augment it with Semantic support has been incorporated within *components* submodule as it will be presented in the remainder of this section.

In listing 5.8, the code snippets with the added attributes specific to the Microdata model are presented, respectively itemscope, itemtype and itemprop. This functionality was added to Models and ObjectModel classes that are found within the Swagger UI project in the *components* submodule.

Listing 5.8: Code snippet showing added attributes from the Microdata model.

```
1 // from models.jsx
2 const extensionClass = schema.get("x-same-as") || schema.get("x-rdf-type") ||
      null
3 ...
4 <div itemScope itemType={extensionClass} id={ `model-${name}` } className="model
      -container" key={ `models-section-${name}` }>
5
6 // from object-model.jsx
7 let extensionProperty = this.getProperty(properties, key)
8 ...
9 <tr style={tableRowStyle} itemProp={extensionProperty} key={key} className={
      isDeprecated && 'deprecated'}>
```

The ObjectModel class is responsible for rendering the models defined in the created API. In the *componentWillMount()* method, a request is made to the URL where the schema.jsonld file resource is found, which contains the definition of the core vocabulary. If this request fails, then a local version of the Schema.org vocabulary is referenced. The *getConceptDescriptionFromSchemaOrg()* method is responsible for extracting the description of the concepts for which the defined extensions (i.e., **x-same-as**, **x-rdf-type**) was used. In the *render()* method, the necessary elements were added to display additional information about the models.

The above mentioned functionality that was added to ObjectModel class is found within the Swagger UI project in the *components* submodule. and is presented in the listing 5.9.

Listing 5.9: Code snippet that contains the core Model rendering logic augmented with semantic constructs.

```
1 import schemaOrg from './schema-org'
2 import Parser from 'html-react-parser'
3
```

```
let rdfKeywords = {"x-rdf-type": "rdf:type", "x-same-as": "owl:sameAs"}
...
export default class ObjectModel extends Component {
  ...
  constructor (props) {
    super(props)
        this.state = {
                schemaOrg: schemaOrg
    }
  }

  getConceptDescriptionFromSchemaOrg (id) {
    if (!id) { return null }
    let concept = this.getConceptFromSchemaOrg(id)
    if (!concept) {return null}

    return this.getDescription(concept)
  }

  getConceptFromSchemaOrg (id) {
    let schemaOrg = this.getSchemaOrg()
    return schemaOrg['@graph'].find(concept => concept['@id'] === id)
  }

  getSchemaOrg () {
    return this.state.schemaOrg
  }

  getDescription (concept) {
    return concept['rdfs:comment']
  }

  getPropertyFromSchemaOrg (properties, propertyName) {
    for (let [key, value] of properties.entries()) {
      if (key === propertyName) {
        for (let [innerKey, innerValue] of value.entries()) {
          if (innerKey in rdfKeywords) {
            return innerValue
          }
        }
      }
    }
    return null
  }

  componentWillMount () {
    fetch('https://schema.org/version/3.7/schema.jsonld')
```

```
51        .then(response => response.json())
52        .then(schema => this.setState({schemaOrg: schema}))
53        .catch(() => this.setState({schemaOrg: schemaOrg}))
54    }
55
56    render () {
57      ...
58      let extensionClass = schema.get('x-same-as') || schema.get('x-rdf-type') ||
        null
59      let conceptDescription = this.getConceptDescriptionFromSchemaOrg(
        extensionClass)
60      ...
61      <p>
62        {title}
63        {extensionClass && <a href={extensionClass} target={'_blank'}>&lt;{
        extensionClass}&gt;</a>}
64      </p>
65      <p>
66        <a style={descriptionStyle} target={'_blank'} href={extensionClass}>
67          {conceptDescription}
68        </a>
69      </p>
70      ...
71      let extensionProperty = this.getPropertyFromSchemaOrg(properties, key)
72      let propertyDescription
73      if (this.getConceptDescriptionFromSchemaOrg(extensionProperty)) {
74        propertyDescription = Parser(this.getConceptDescriptionFromSchemaOrg(
        extensionProperty))
75      } else {
76        propertyDescription = this.getConceptDescriptionFromSchemaOrg(
        extensionProperty)
77      }
78      ...
79      <td style={propertyStyle}>
80        <a href={extensionProperty} target="_blank">
81          {extensionProperty && <span>&lt;{extensionProperty}&gt;</span>}
82        </a>
83      </td>
84      <td style={propertyStyle}>
85        <a style={descriptionStyle} target="_blank" href={extensionProperty}>
86          {propertyDescription}
87        </a>
88      </td>
89      ...
90    }
```

5.5 Future Directions

The augmentation of OpenAPI specification with knowledge defined in Schema.org as it was proposed in this thesis is a foundational step in this direction. Further improvements can be made in this direction.

The OpenAPI specification may be extended to incorporate other ontologies (e.g. DBpedia[10], Disease Ontology[11], Dublin Core[12], etc.).

A step in the improvement process is that once we have the augmented specification, it can be discovered by automated tools.

Another direction of improvement is to publish the resulting ontology to a triplestore (e.g. AllegroGraph[13], Amazon Neptune[14], Stardog[15], etc.) and then to create a mechanism, using SPARQL, that provides API recommendations immersed directly in the Swagger UI tool.

A final direction of improvement would be the creation of a mechanism by which Swagger Editor can self-detect what concepts are defined in the created API and automatically create the associations or provide the user with suggestion from which he can choose the most appropriate one.

Summary

This chapter described the process of extending OpenAPI specification with semantics, along with the semantic support added to the Swagger Editor and Swagger UI open source tools. The chapter concluded with remarks regarding future directions the augmentation of OpenAPI specification might have.

[10] http://dbpedia.org/ontology/
[11] http://www.disease-ontology.org/
[12] http://www.dublincore.org/specifications/dublin-core/
[13] https://franz.com/agraph/allegrograph/
[14] https://aws.amazon.com/neptune/
[15] https://www.stardog.com/

Chapter 6

Case study: Taxi Service

In order to demonstrate the functionality to the OpenAPI Specification tools that were augmented with Semantic support, we have created an OpenAPI Definition that represents the functionality provided by a Taxi Service. The OpenAPI definition can be consulted in appendix A.

The OpenApi Taxi Service Definition was created using Swagger Editor and made use of the Semantic support functionality implemented as described in chapter 5, namely to provide suggestions of concepts from Schema.org. Figure 6.1 depicts the case in which the editor offers suggestions of concepts from Schema.org at the Schemas level for the **x-same-as** extension.

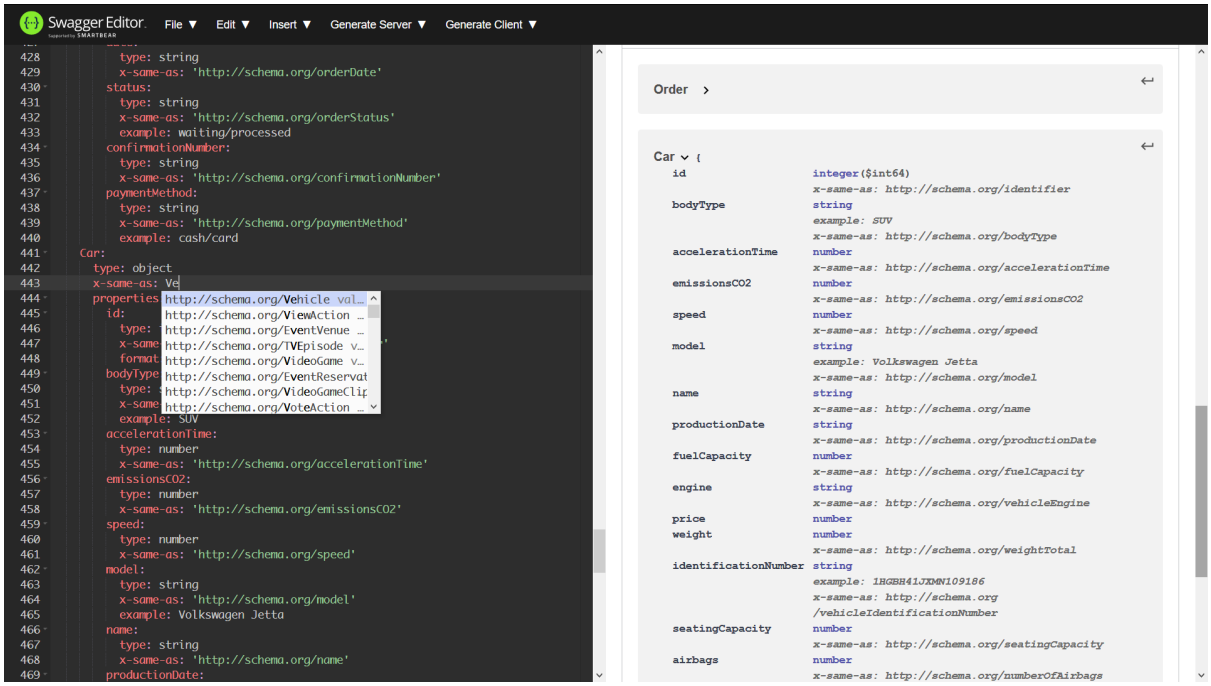

Figure 6.1: Autocomplete support offered by Swagger Editor for x-same-as extension.

During the creation of the Taxi Service OpenAPI Defition we have augmented several elements from it with related concepts from Schema.org:

- **Order** with <http://schema.org/Order>
- **Car** with <http://schema.org/Vehicle>
- **TaxiService** with <http://schema.org/TaxiService>
- **Customer** with <http://schema.org/Person>

We have also made references to related concepts from Schema.org to properties pertaining to the schemas used by the operations defined in the API:

- Order
  - **number** with <http://schema.org/orderNumber>
  - **status** with <http://schema.org/orderStatus>
  - **paymentMethod** with <http://schema.org/paymentMethod>
  - etc.
- Car
  - **bodyType** with <http://schema.org/bodyType>
  - **emissionsCO2** with <http://schema.org/emissionsCO2>
  - **productionDate** with <http://schema.org/productionDate>
  - **seatingCapacity** with <http://schema.org/seatingCapacity>
  - **fuelType** with <http://schema.org/fuelType>
  - etc.
- etc.

Once the creation of the Taxi Service definition has been completed, we can select the **File** → **Convert and save as Turtle** option, as depicted in figure 6.2.

The contents of the generated file will consist of RDF triples of the form <*subject*> <*predicate*> <*object*>. An excerpt from the result of the Turtle conversion of the Taxi Service OpenAPI definition can be consulted in listing 6.1. The entire result of the conversion to Turtle format can be consulted in Appendix B.

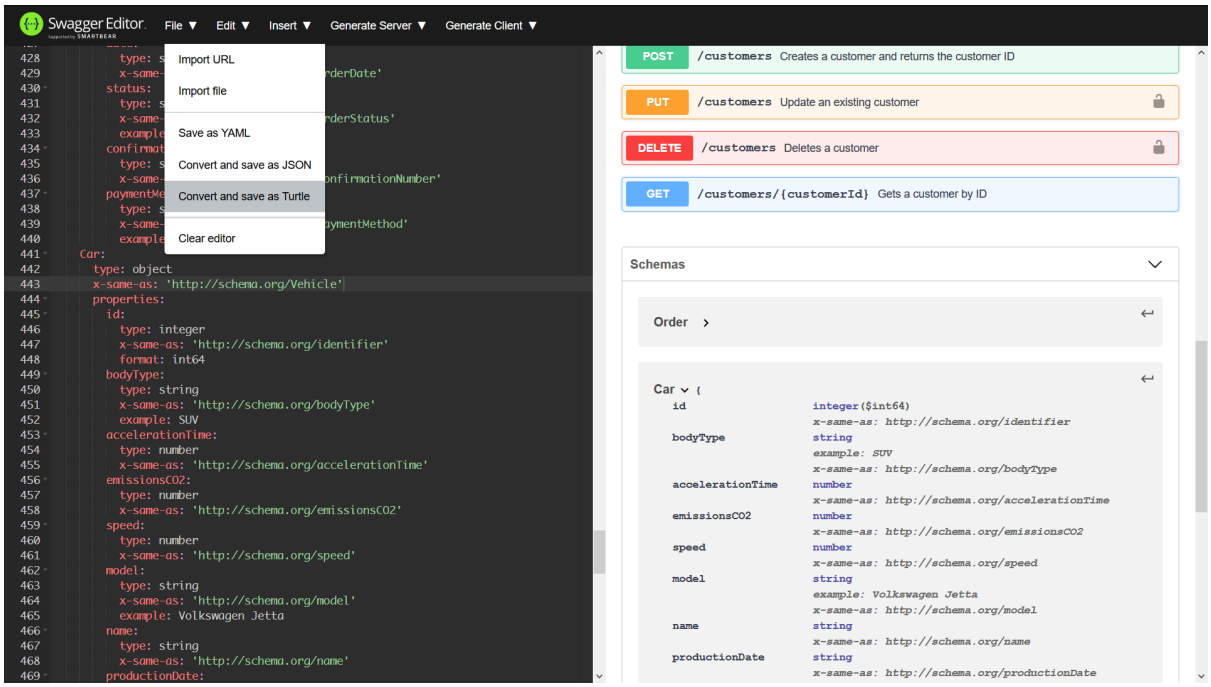

Figure 6.2: **Convert and save as Turtle** option added to Swagger Editor.

Listing 6.1: Taxi Service ontology resulting from knowledge augmentation.

```
@prefix : <http://www.semanticweb.org/ontologies/Taxi_Service/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .
@prefix gr: <http://purl.org/goodrelations/v1#> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@base <http://www.semanticweb.org/ontologies/Taxi_Service/> .
...

:Order owl:sameAs <http://schema.org/Order> .
:status owl:sameAs <http://schema.org/orderStatus> .
:status rdfs:range :Order .
:confirmationNumber owl:sameAs <http://schema.org/confirmationNumber> .
:confirmationNumber rdfs:range :Order .
:paymentMethod owl:sameAs <http://schema.org/paymentMethod> .
:paymentMethod rdfs:range :Order .
...

:Car owl:sameAs <Vehicle> .
:seatingCapacity owl:sameAs <http://schema.org/seatingCapacity> .
:seatingCapacity rdfs:range :Car .
:airbags owl:sameAs <http://schema.org/numberOfAirbags> .
:airbags rdfs:range :Car .
```

```
:meetsEmissionStandard owl:sameAs <http://schema.org/meetsEmissionStandard> .
:meetsEmissionStandard rdfs:range :Car .
:fuelType owl:sameAs <http://schema.org/fuelType> .
:fuelType rdfs:range :Car .
:fuelConsumption owl:sameAs <http://schema.org/fuelConsumption> .
:fuelConsumption rdfs:range :Car .
...

:TaxiService owl:sameAs <http://schema.org/TaxiService> .
:hoursAvailable owl:sameAs <http://schema.org/hoursAvailable> .
:hoursAvailable rdfs:range :TaxiService .
...
```

Further, we can upload the contents of the Turtle file into WebVOWL[1] to be able to visualize the resulting Taxi Service Ontology. Figure 6.3 provides an interactive visualization of the Taxi Service Ontology.

With the help of the semantic support added to Swagger UI, we can visualize in the Schemas/Models section the references to Schema.org for each concept for which the extensions were used. We can also observe the descriptions retrieved from Schema.org and can access the content on Schema.org for further details for each concept of interest. The descriptions were dynamically retrieved from Schema.org, where available. The previously mentioned functionalities can be observed in figure 6.4.

The Google Structured Data Testing Tool[2] allows us to validate our code to make sure that structured data is understood by search engines when crawling a page. After introducing the content of the HTML document that was augmented with microdata by the Swagger UI tool, the tool detected 4 concepts (i.e. *Order*, *Vehicle*, *TaxiService*, *Person*), as can be observed in figure 6.5.

Figure 6.6 displays all the structured data detected by Google when the TaxiService concept is selected and expanded.

[1] http://vowl.visualdataweb.org/webvowl.html
[2] https://search.google.com/structured-data/testing-tool

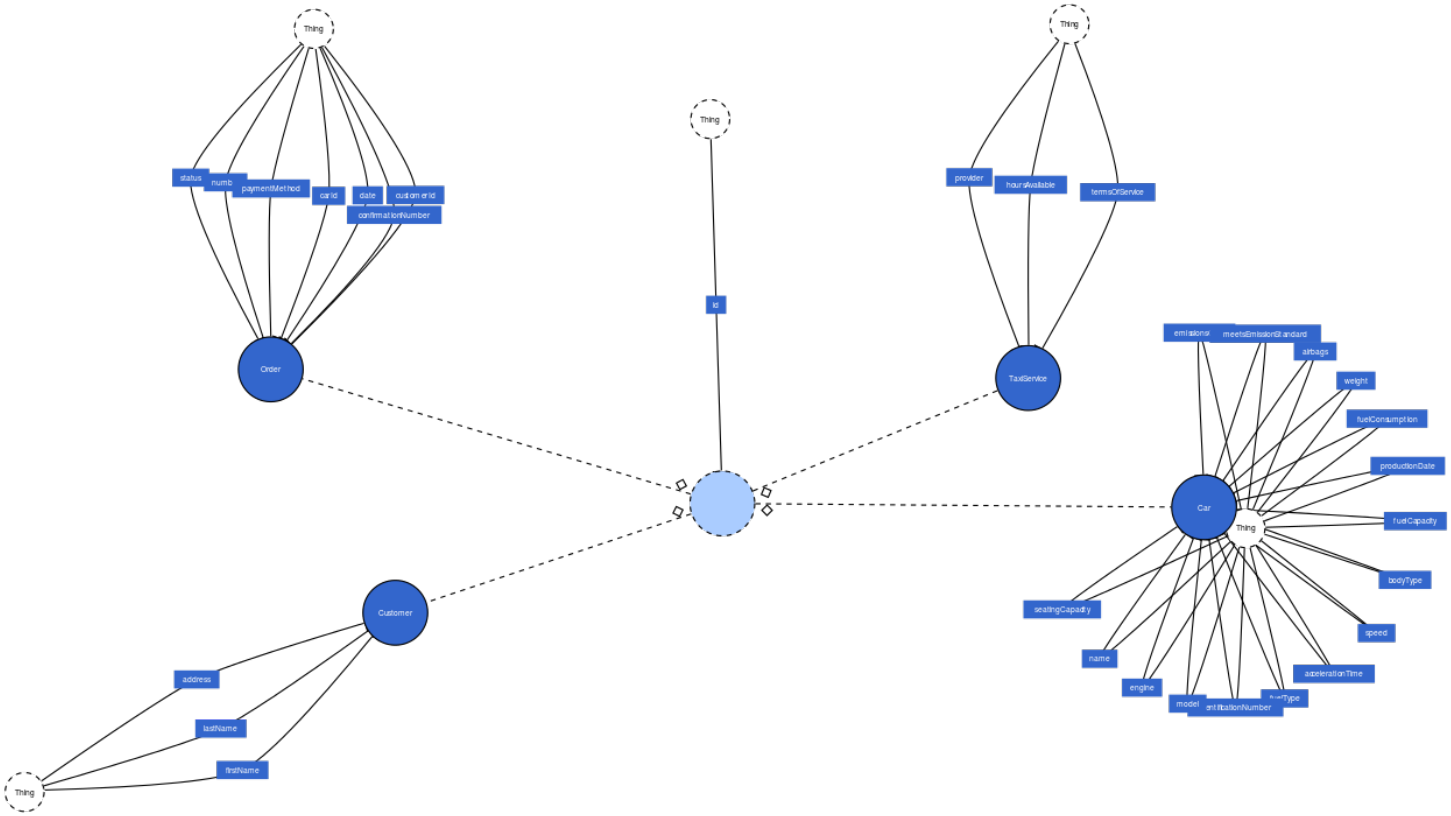

Figure 6.3: Interactive visualization of the Taxi Service Ontology using WebVOWL.

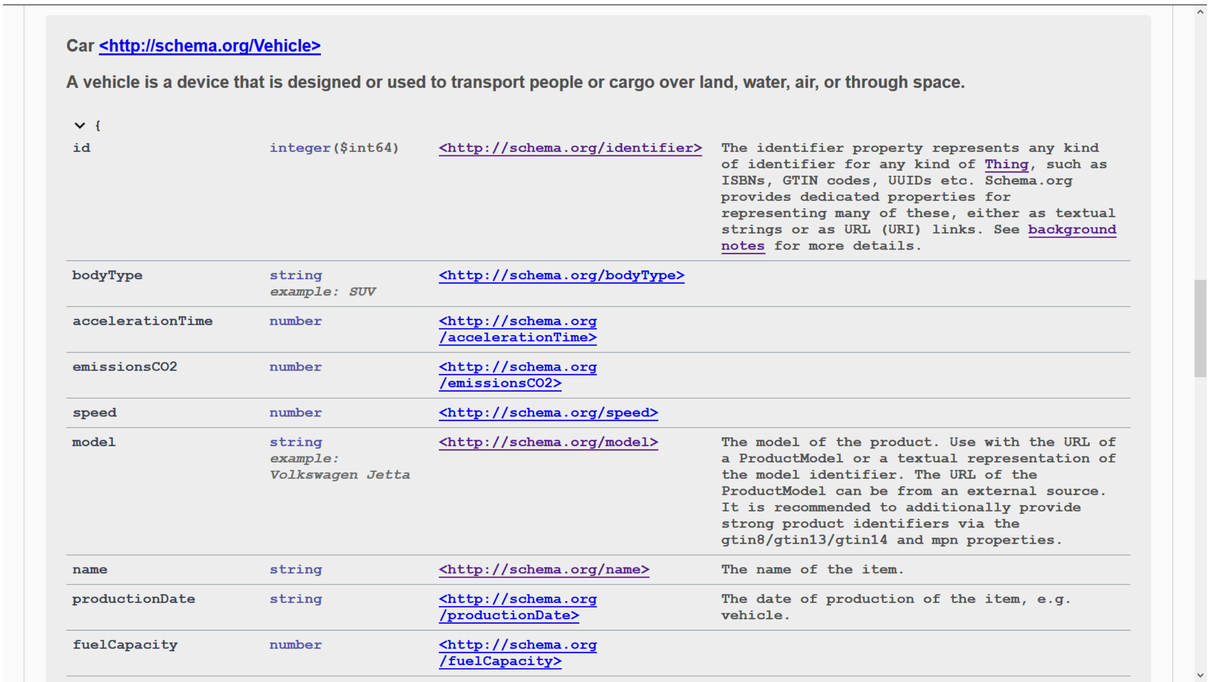

Figure 6.4: Display of the semantic details in Swagger UI.

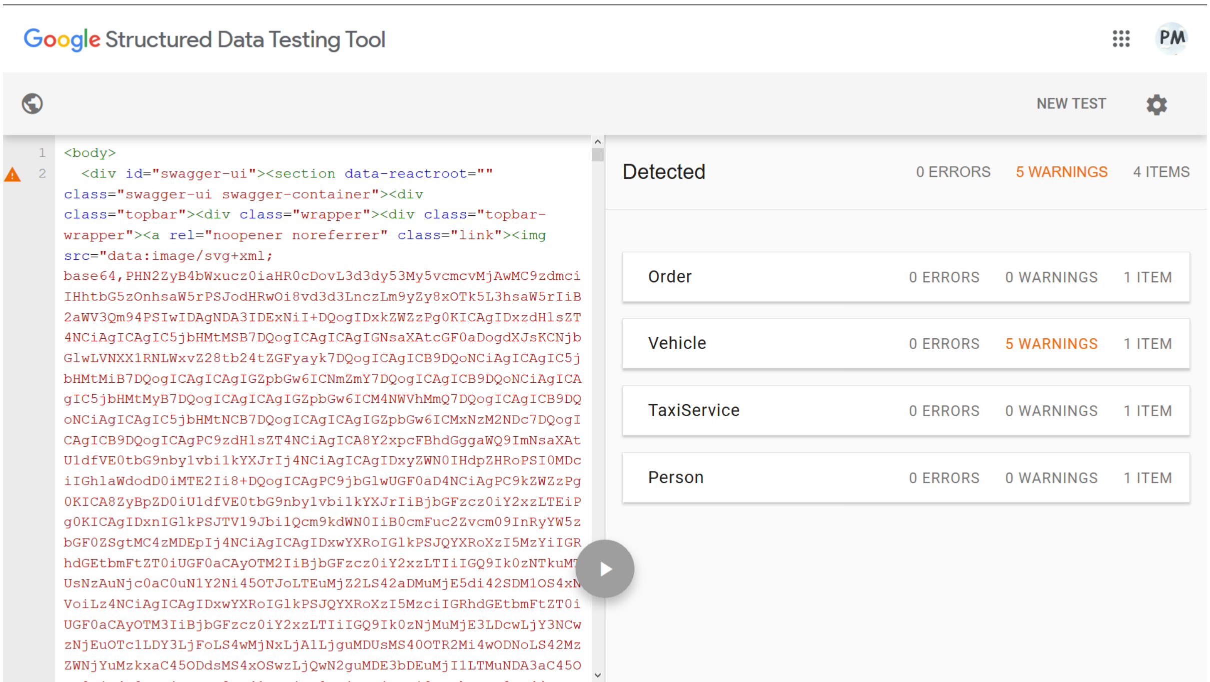

Figure 6.5: Structured data detected by Google Structured Data Testing Tool.

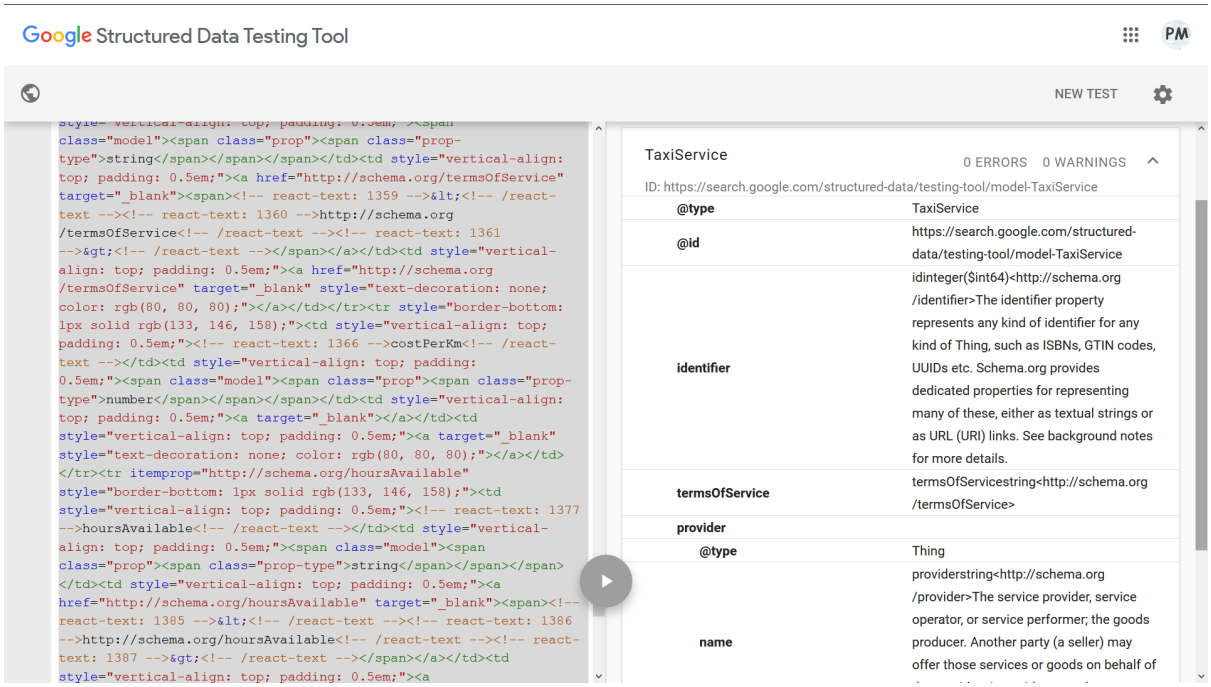

Figure 6.6: Structured data for TaxiService item detected by Google Structured Data Testing Tool.

Chapter 7

Conclusions

Thesis Contribution

Added support for Semantic Augmentation of OpenAPI specification within Swagger Editor and Swagger UI tools.

In order to augment OpenAPI specification with Semantic support, we have extended the functionlity of the Swagger Editor and Swagger UI open source tools.

The following features have been added to the *Swagger Editor* tool: the extension of the OpenAPI specification with custom properties: *x-same-as* and *x-rdf-type*, suggestions and autocomplete support for the newly added extensions, contextual suggestions and autocomplete support for concepts retrieved from *Schema.org*, and the ability to convert and save an OpenAPI definition in *Turtle* format.

As for the *Swagger UI* tool, the upcoming features have been added: the incorporation of *Structured Data* in the generated OpenAPI definition documentation using *Microdata*, the retrieval of existing description from *Schema.org* for each element that has been augmented with semantics and its inclusion in the generated OpenAPI definition documentation, and the association of the elements from OpenAPI definition with concepts from *Schema.org vocabulary*.

Future Directions

The thesis provided a foundational layer that enables Semantic support and augmentation of OpenAPI Specification conformant API definitions. The foundational capabilities presented in this thesis can further be extended or improved in the following directions.

The Semantic support currently added by the work in this thesis is limited to using concepts defined only in the Schema.org vocabulary. The implementation would benefit

by enabling the user to incorporate/refer concepts from other ontologies (e.g. DBpedia[1], Disease Ontology[2], Dublin Core[3], etc.).

The next layer of improvement would consist in the addition of a mechanism through which the ontology resulted from the Semantic Augmentation is automatically published in a triplestore (e.g. AllegroGraph[4], Amazon Neptune[5], Stardog[6], etc.); this database will entail itself to provide API recommendations, through the usage of SPARQL queries, immersed directly in the Swagger UI tool. This system can enable API recommendation support based on several similarity criteria.

A final direction of improvement would be the creation of a mechanism by which Swagger Editor can self-detect what concepts are defined in the created API and automatically create the associations or provide the user with suggestion from which he can choose the most appropriate one.

[1] http://dbpedia.org/ontology/
[2] http://www.disease-ontology.org/
[3] http://www.dublincore.org/specifications/dublin-core/
[4] https://franz.com/agraph/allegrograph/
[5] https://aws.amazon.com/neptune/
[6] https://www.stardog.com/

Appendix A

Taxi Service OpenAPI Definition

This appendix presents the definition of an Web API that conforms to the OpenAPI Specification. The API models the functionality provided by a Taxi Service and it was developed by the author of the thesis as a case study for demonstrating the Semantic support functionalities that were implemented by the same author in the Swagger Editor and Swagger UI Open Source tools.

Listing A.1: Taxi Service OpenAPI definition using YAML format.

```
openapi: 3.0.0
info:
  version: 0.0.0
  title: Taxi Service
  description: This is a sample server for a taxi service.
  termsOfService: 'http://example.com/terms/'
tags:
  - name: order
    description: Operations about orders
  - name: car
    description: Operations about cars
  - name: taxi service
    description: Operations about taxi services
  - name: customer
    description: Operations about customers
paths:
  /orders:
    post:
      tags:
        - order
      summary: Creates an order and returns the order ID
      operationId: createOrder
      requestBody:
        required: true
```

```
25          description: A JSON object that contains the order number and item.
26          content:
27            application/json:
28              schema:
29                $ref: '#/components/schemas/Order'
30        responses:
31          '201':
32            description: Created
33            content:
34              application/json:
35                schema:
36                  type: object
37                  properties:
38                    id:
39                      type: integer
40                      format: int64
41                      description: ID of the created order.
42            links:
43              GetOrderById:
44                operationId: getOrder
45                parameters:
46                  orderId: '$response.body#/id'
47                description: >
48                  The `id` value returned in the response can be used as the
49                  `orderId` parameter in `GET /orders/{orderId}`.
50      put:
51        tags:
52          - order
53        summary: Update an existing order
54        operationId: updateOrder
55        requestBody:
56          required: true
57          description: A JSON object that contains the order number and item.
58          content:
59            application/json:
60              schema:
61                $ref: '#/components/schemas/Order'
62        responses:
63          '400':
64            description: Invalid ID supplied
65          '404':
66            description: Order not found
67          '405':
68            description: Validation exception
69        security:
70          - taxiservice_auth:
71              - 'write:orders'
```

```
72              - 'read:orders'
73      delete:
74        tags:
75          - order
76        summary: Deletes an order
77        operationId: deleteOrder
78        parameters:
79          - name: orderId
80            in: path
81            required: true
82            description: Order id to delete
83            schema:
84              type: integer
85              format: int64
86        responses:
87          '400':
88            description: Invalid ID supplied
89          '404':
90            description: Order not found
91        security:
92          - taxiservice_auth:
93              - 'write:orders'
94              - 'read:orders'
95    '/orders/{orderId}':
96      get:
97        tags:
98          - order
99        summary: Gets an order by ID
100       operationId: getOrderById
101       parameters:
102         - in: path
103           name: orderId
104           required: true
105           schema:
106             type: integer
107             format: int64
108       responses:
109         '200':
110           description: An Order object
111           content:
112             application/json:
113               schema:
114                 $ref: '#/components/schemas/Order'
115   /cars:
116     post:
117       tags:
118         - car
```

```
119       summary: Creates a car and returns the car ID
120       operationId: createCar
121       requestBody:
122         required: true
123         description: A JSON object that contains the car.
124         content:
125           application/json:
126             schema:
127               $ref: '#/components/schemas/Car'
128       responses:
129         '201':
130           description: Created
131           content:
132             application/json:
133               schema:
134                 type: object
135                 properties:
136                   id:
137                     type: integer
138                     format: int64
139                     description: ID of the created car.
140           links:
141             GetCarById:
142               operationId: getCar
143               parameters:
144                 carId: '$response.body#/id'
145               description: >
146                 The `id` value returned in the response can be used as the
147                 `carId` parameter in `GET /cars/{carId}`.
148     put:
149       tags:
150         - car
151       summary: Update an existing car
152       operationId: updateCar
153       requestBody:
154         required: true
155         description: A JSON object that contains the car.
156         content:
157           application/json:
158             schema:
159               $ref: '#/components/schemas/Car'
160       responses:
161         '400':
162           description: Invalid ID supplied
163         '404':
164           description: Car not found
165         '405':
```

```
166           description: Validation exception
167       security:
168         - taxiservice_auth:
169             - 'write:cars'
170             - 'read:cars'
171     delete:
172       tags:
173         - car
174       summary: Deletes a car
175       operationId: deleteCar
176       parameters:
177         - name: carId
178           in: path
179           required: true
180           description: Car id to delete
181           schema:
182             type: integer
183             format: int64
184       responses:
185         '400':
186           description: Invalid ID supplied
187         '404':
188           description: Car not found
189       security:
190         - taxiservice_auth:
191             - 'write:cars'
192             - 'read:cars'
193   '/cars/{carId}':
194     get:
195       tags:
196         - car
197       summary: Gets a car by ID
198       operationId: getCar
199       parameters:
200         - in: path
201           name: carId
202           required: true
203           schema:
204             type: integer
205             format: int64
206       responses:
207         '200':
208           description: A Car object
209           content:
210             application/json:
211               schema:
212                 $ref: '#/components/schemas/Car'
```

```
213   /taxiServices:
214     post:
215       tags:
216         - taxi service
217       summary: Creates a taxi service and returns the taxi service ID
218       operationId: createTaxiService
219       requestBody:
220         required: true
221         description: A JSON object that contains the taxi service.
222         content:
223           application/json:
224             schema:
225               $ref: '#/components/schemas/TaxiService'
226       responses:
227         '201':
228           description: Created
229           content:
230             application/json:
231               schema:
232                 type: object
233                 properties:
234                   id:
235                     type: integer
236                     format: int64
237                     description: ID of the created taxi service.
238           links:
239             GetTaxiServiceById:
240               operationId: getTaxiService
241               parameters:
242                 carId: '$response.body#/id'
243               description: >
244                 The `id` value returned in the response can be used as the
245                 `taxiServiceId` parameter in `GET /taxis/{taxiServiceId}`.
246     put:
247       tags:
248         - taxi service
249       summary: Update an existing taxi service
250       operationId: updateTaxiService
251       requestBody:
252         required: true
253         description: A JSON object that contains the taxi service.
254         content:
255           application/json:
256             schema:
257               $ref: '#/components/schemas/TaxiService'
258       responses:
259         '400':
```

```
260           description: Invalid ID supplied
261         '404':
262           description: TaxiService not found
263         '405':
264           description: Validation exception
265       security:
266         - taxiservice_auth:
267             - 'write:taxiServices'
268             - 'read:taxiServices'
269     delete:
270       tags:
271         - taxi service
272       summary: Deletes a taxi service
273       operationId: deleteTaxiService
274       parameters:
275         - name: taxiServiceId
276           in: path
277           required: true
278           description: TaxiService id to delete
279           schema:
280             type: integer
281             format: int64
282       responses:
283         '400':
284           description: Invalid ID supplied
285         '404':
286           description: TaxiService not found
287       security:
288         - taxiservice_auth:
289             - 'write:taxiServices'
290             - 'read:taxiServices'
291   '/taxiServices/{taxiServiceId}':
292     get:
293       tags:
294         - taxi service
295       summary: Gets a taxi service by ID
296       operationId: getTaxiService
297       parameters:
298         - in: path
299           name: taxiServiceId
300           required: true
301           schema:
302             type: integer
303             format: int64
304       responses:
305         '200':
306           description: A TaxiService object
```

```
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/TaxiService'
  /customers:
    post:
      tags:
        - customer
      summary: Creates a customer and returns the customer ID
      operationId: createDCustomer
      requestBody:
        required: true
        description: A JSON object that contains the customer.
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Customer'
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema:
                type: object
                properties:
                  id:
                    type: integer
                    format: int64
                    description: ID of the created customer.
          links:
            GetCustomerById:
              operationId: getCustomer
              parameters:
                customerId: '$response.body#/id'
              description: >
                The `id` value returned in the response can be used as the
                `customerId` parameter in `GET /customers/{customerId}`.
    put:
      tags:
        - customer
      summary: Update an existing customer
      operationId: updateCustomer
      requestBody:
        required: true
        description: A JSON object that contains the customer.
        content:
          application/json:
```

```
            schema:
              $ref: '#/components/schemas/Customer'
      responses:
        '400':
          description: Invalid ID supplied
        '404':
          description: Customer not found
        '405':
          description: Validation exception
      security:
        - taxiservice_auth:
            - 'write:customers'
            - 'read:customers'
    delete:
      tags:
        - customer
      summary: Deletes a customer
      operationId: deleteCustomer
      parameters:
        - name: customerId
          in: path
          required: true
          description: Customer id to delete
          schema:
            type: integer
            format: int64
      responses:
        '400':
          description: Invalid ID supplied
        '404':
          description: Customer not found
      security:
        - taxiservice_auth:
            - 'write:customers'
            - 'read:customers'
  '/customers/{customerId}':
    get:
      tags:
        - customer
      summary: Gets a customer by ID
      operationId: getCustomer
      parameters:
        - in: path
          name: customerId
          required: true
          schema:
            type: integer
```

```
            format: int64
      responses:
        '200':
          description: A Customer object
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Customer'
components:
  schemas:
    Order:
      type: 'object'
      x-same-as: 'http://schema.org/Order'
      properties:
        id:
          type: 'integer'
          x-same-as: 'http://schema.org/identifier'
          format: int64
        carId:
          type: 'integer'
          x-same-as: 'http://schema.org/identifier'
          format: int64
        customerId:
          type: 'integer'
          x-same-as: 'http://schema.org/identifier'
          format: int64
        number:
          type: 'string'
          x-same-as: 'http://schema.org/orderNumber'
          format: int64
        date:
          type: 'string'
          x-same-as: 'http://schema.org/orderDate'
        status:
          type: 'string'
          x-same-as: 'http://schema.org/orderStatus'
          example: 'waiting/processed'
        confirmationNumber:
          type: 'string'
          x-same-as: 'http://schema.org/confirmationNumber'
        paymentMethod:
          type: 'string'
          x-same-as: 'http://schema.org/paymentMethod'
          example: 'cash/card'
    Car:
      type: object
      x-same-as: 'http://schema.org/Vehicle'
```

```
448       properties:
449         id:
450           type: 'integer'
451           x-same-as: 'http://schema.org/identifier'
452           format: 'int64'
453         bodyType:
454           type: 'string'
455           x-same-as: 'http://schema.org/bodyType'
456           example: 'SUV'
457         accelerationTime:
458           type: 'number'
459           x-same-as: 'http://schema.org/accelerationTime'
460         emissionsCO2:
461           type: 'number'
462           x-same-as: 'http://schema.org/emissionsCO2'
463         speed:
464           type: 'number'
465           x-same-as: 'http://schema.org/speed'
466         model:
467           type: 'string'
468           x-same-as: 'http://schema.org/model'
469           example: 'Volkswagen Jetta'
470         name:
471           type: 'string'
472           x-same-as: 'http://schema.org/name'
473         productionDate:
474           type: 'string'
475           x-same-as: 'http://schema.org/productionDate'
476         fuelCapacity:
477           type: 'number'
478           x-same-as: 'http://schema.org/fuelCapacity'
479         engine:
480           type: 'string'
481           x-same-as: 'http://schema.org/vehicleEngine'
482         price:
483           type: 'number'
484         weight:
485           type: 'number'
486           x-same-as: 'http://schema.org/weightTotal'
487         identificationNumber:
488           type: 'string'
489           x-same-as: 'http://schema.org/vehicleIdentificationNumber'
490           example: '1HGBH41JXMN109186'
491         seatingCapacity:
492           type: 'number'
493           x-same-as: 'http://schema.org/seatingCapacity'
494         airbags:
```

```
          type: 'number'
          x-same-as: 'http://schema.org/numberOfAirbags'
        meetsEmissionStandard:
          type: 'string'
          x-same-as: 'http://schema.org/meetsEmissionStandard'
        fuelType:
          type: 'string'
          x-same-as: 'http://schema.org/fuelType'
        fuelConsumption:
          type: 'number'
          x-same-as: 'http://schema.org/fuelConsumption'
    TaxiService:
      type: 'object'
      x-same-as: 'http://schema.org/TaxiService'
      properties:
        id:
          type: 'integer'
          x-same-as: 'http://schema.org/identifier'
          format: int64
        provider:
          type: 'string'
          x-same-as: 'http://schema.org/provider'
        rating:
          type: 'number'
        reviews:
          type: 'array'
          items:
            type: 'string'
        cars:
          type: 'array'
          items:
            $ref: '#/components/schemas/Car'
        orders:
          type: 'array'
          items:
            $ref: '#/components/schemas/Order'
        termsOfService:
          type: 'string'
          x-same-as: 'http://schema.org/termsOfService'
        costPerKm:
          type: 'number'
        hoursAvailable:
          type: 'string'
          x-same-as: 'http://schema.org/hoursAvailable'
        phoneNumber:
          type: 'string'
    Customer:
```

```
542       type: 'object'
543       x-same-as: 'http://schema.org/Person'
544       properties:
545         id:
546           type: 'integer'
547           x-same-as: 'http://schema.org/identifier'
548           format: int64
549         firstName:
550           type: 'string'
551           x-same-as: 'http://schema.org/name'
552         lastName:
553           type: 'string'
554           x-same-as: 'http://schema.org/name'
555         address:
556           type: 'string'
557           x-same-as: 'http://schema.org/address'
558 externalDocs:
559   description: Find out more about Swagger
560   url: 'http://swagger.io'
```

Appendix B

Taxi Service Ontology

This appendix presents a Taxi Service-related Ontology in Turtle format as a case study for demonstrating the Semantic support functionalities that were added by the author of the thesis in the Swagger Editor and Swagger UI Open Source tools. The Ontology resulted from converting and exporting the Taxi Service OpenAPI Definition presented in appendix A using the functionalities implemented by the author of the thesis in the Swagger Editor tool.

Listing B.1: Taxi Service ontology resulting from knowledge augmentation.

```
1  @prefix : <http://www.semanticweb.org/ontologies/Taxi_Service/> .
2  @prefix dc: <http://purl.org/dc/elements/1.1/> .
3  @prefix gr: <http://purl.org/goodrelations/v1#> .
4  @prefix dbo: <http://dbpedia.org/ontology/> .
5  @prefix dbp: <http://dbpedia.org/property/> .
6  @prefix dbr: <http://dbpedia.org/resource/> .
7  @prefix owl: <http://www.w3.org/2002/07/owl#> .
8  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
9  @prefix xml: <http://www.w3.org/XML/1998/namespace> .
10 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
11 @prefix muto: <http://purl.org/muto/core#> .
12 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
13 @prefix schema: <http://schema.org/> .
14 @prefix sioc: <http://rdfs.org/sioc/ns#> .
15 @prefix yago: <http://dbpedia.org/class/yago/> .
16 @prefix owl2xml: <http://www.w3.org/2006/12/owl2-xml#> .
17 @base <http://www.semanticweb.org/ontologies/Taxi_Service/> .
18
19 :Order owl:sameAs <http://schema.org/Order> .
20 :id owl:sameAs <http://schema.org/identifier> .
21 :id rdfs:range :Order .
22 :carId owl:sameAs <http://schema.org/identifier> .
23 :carId rdfs:range :Order .
```

```
:customerId owl:sameAs <http://schema.org/identifier> .
:customerId rdfs:range :Order .
:number owl:sameAs <http://schema.org/orderNumber> .
:number rdfs:range :Order .
:date owl:sameAs <http://schema.org/orderDate> .
:date rdfs:range :Order .
:status owl:sameAs <http://schema.org/orderStatus> .
:status rdfs:range :Order .
:confirmationNumber owl:sameAs <http://schema.org/confirmationNumber> .
:confirmationNumber rdfs:range :Order .
:paymentMethod owl:sameAs <http://schema.org/paymentMethod> .
:paymentMethod rdfs:range :Order .

:Car owl:sameAs <Ve> .
:id owl:sameAs <http://schema.org/identifier> .
:id rdfs:range :Car .
:bodyType owl:sameAs <http://schema.org/bodyType> .
:bodyType rdfs:range :Car .
:accelerationTime owl:sameAs <http://schema.org/accelerationTime> .
:accelerationTime rdfs:range :Car .
:emissionsCO2 owl:sameAs <http://schema.org/emissionsCO2> .
:emissionsCO2 rdfs:range :Car .
:speed owl:sameAs <http://schema.org/speed> .
:speed rdfs:range :Car .
:model owl:sameAs <http://schema.org/model> .
:model rdfs:range :Car .
:name owl:sameAs <http://schema.org/name> .
:name rdfs:range :Car .
:productionDate owl:sameAs <http://schema.org/productionDate> .
:productionDate rdfs:range :Car .
:fuelCapacity owl:sameAs <http://schema.org/fuelCapacity> .
:fuelCapacity rdfs:range :Car .
:engine owl:sameAs <http://schema.org/vehicleEngine> .
:engine rdfs:range :Car .
:weight owl:sameAs <http://schema.org/weightTotal> .
:weight rdfs:range :Car .
:identificationNumber owl:sameAs <http://schema.org/vehicleIdentificationNumber>
    .
:identificationNumber rdfs:range :Car .
:seatingCapacity owl:sameAs <http://schema.org/seatingCapacity> .
:seatingCapacity rdfs:range :Car .
:airbags owl:sameAs <http://schema.org/numberOfAirbags> .
:airbags rdfs:range :Car .
:meetsEmissionStandard owl:sameAs <http://schema.org/meetsEmissionStandard> .
:meetsEmissionStandard rdfs:range :Car .
:fuelType owl:sameAs <http://schema.org/fuelType> .
:fuelType rdfs:range :Car .
```

```
:fuelConsumption owl:sameAs <http://schema.org/fuelConsumption> .
:fuelConsumption rdfs:range :Car .

:TaxiService owl:sameAs <http://schema.org/TaxiService> .
:id owl:sameAs <http://schema.org/identifier> .
:id rdfs:range :TaxiService .
:provider owl:sameAs <http://schema.org/provider> .
:provider rdfs:range :TaxiService .
:termsOfService owl:sameAs <http://schema.org/termsOfService> .
:termsOfService rdfs:range :TaxiService .
:hoursAvailable owl:sameAs <http://schema.org/hoursAvailable> .
:hoursAvailable rdfs:range :TaxiService .

:Customer owl:sameAs <http://schema.org/Person> .
:id owl:sameAs <http://schema.org/identifier> .
:id rdfs:range :Customer .
:firstName owl:sameAs <name> .
:firstName rdfs:range :Customer .
:lastName owl:sameAs <http://schema.org/name> .
:lastName rdfs:range :Customer .
:address owl:sameAs <http://schema.org/address> .
:address rdfs:range :Customer .
```

Bibliography

[1] Douglas K. Barry. *Web Services, Service-Oriented Architectures, and Cloud Computing, Second Edition: The Savvy Manager's Guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2013.

[2] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web . *Scientific American*, page 4, 05 2001.

[3] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture . W3C Working Group Note, 11 February 2004, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/ . Latest version available at http://www.w3.org/TR/ws-arch/ .

[4] Dan Brickley and R.V. Guha. RDF Schema 1.1 . W3C Recommendation, 25 February 2014, http://www.w3.org/TR/2014/REC-rdf-schema-20140225/ . Latest version available at http://www.w3.org/TR/rdf-schema/ .

[5] Bill Doerrfeld, Kristopher Sandoval, Art Anthony, and Chris Wood. *API Design on the Scale of Decades*. Nordic APIs, 2016-2017.

[6] M Endrei, J Ang, Ali Arsanjani, S Chua, Philippe Comte, P Krogdahl, Min Luo, and T Newling. *IBM: Patterns: service-oriented architecture and web services*. 01 2004.

[7] International Organization for Standardization. Information technology — Reference Architecture for Service Oriented Architecture (SOA RA) . ISO/IEC 18384-1 Part 1: Terminology and concepts for SOA, 01 June 2016 .

[8] W3C SPARQL Working Group. SPARQL 1.1 Overview . W3C Recommendation, 21 March 2013, http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/ . Latest version available at http://www.w3.org/TR/sparql11-overview/ .

[9] Hugo Haas and Allen Brown. Web Services Glossary . W3C Working Group Note, 11 February 2004, http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/ . Latest version available at http://www.w3.org/TR/ws-gloss/ .

[10] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language . W3C Recommendation, 21 March 2013, http://www.w3.org/TR/2013/

REC-sparql11-query-20130321/ . Latest version available at http://www.w3.org/TR/sparql11-query/ .

[11] Pascal Hitzler, Markus Krötzsch, Bijan Parsia, Peter F. Patel-Schneider, and Sebastian Rudolph. OWL 2 Web Ontology Language Primer (Second Edition) . W3C Recommendation, 11 December 2012, http://www.w3.org/TR/2012/REC-owl2-primer-20121211/ . Latest version available at http://www.w3.org/TR/owl2-primer/ .

[12] Pascal Hitzler, Markus Kr ö tzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.

[13] Daniel Jacobson, Greg Brail, and Dan Woods. *APIs: A Strategy Guide*. O'Reilly Media, Inc., 2011.

[14] Jacek Kopecký, Paul Fremantle, and Rich Boakes. A history and future of web apis. *it - Information Technology*, 56:12, 01 2014.

[15] Kin Lane. API Providers Guide API Design . 3scale by Red Hat, July 2015, https://www.redhat.com/cms/managed-files/mi-3scale-api-provider-guide-api-design-ebook-f7865-201706-en.pdf.

[16] Ken Laskey, Peter Brown, Jeff A. Estefan, Francis McCabe, and Danny Thornton. Reference Architecture Foundation For Service Oriented Architecture Version 1.0 . Comittee Specification 01, 04 December 2012, http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html . Latest version available at http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra.html .

[17] Inc Mark Hadley — Sun Microsystems. Web Application Description Language . W3C Member Submission, 31 August 2009, http://www.w3.org/Submission/2009/SUBM-wadl-20090831/ . Latest version available at http://www.w3.org/Submission/wadl/ .

[18] Chaals McCathie Nevile, Dan Brickley, and Ian Hickson. HTML Microdata . W3C Recommendation, 26 April 2018, https://www.w3.org/TR/2018/WD-microdata-20180426/ . Latest version available at https://www.w3.org/TR/microdata/ .

[19] Chris Rayns, Mark Cocker, Regis David, and et al. *CICS and SOA Architecture and Integration Choices*. IBM Redbooks, seventh edition edition, March 2012.

[20] Kristopher Sandoval, 2016. Top Specification Formats for REST APIs, https://nordicapis.com/top-specification-formats-for-rest-apis/ .

[21] Guus Schreiber and Yves Raimond. RDF 1.1 Primer . W3C Working Group Note, 24 June 2014, http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/ . Latest version available at http://www.w3.org/TR/rdf11-primer/ .

[22] SmartBear. The State of API 2019 . https://smartbear.com/SmartBearBrand/media/pdf/SmartBear_State_of_API_2019.pdf.

[23] SmartBear Software, 2019. OpenAPI Specification Version 3.0.2, https://swagger.io/specification/ .

[24] SmartBear Software, 2019. Swagger Editor Documentation, https://swagger.io/docs/open-source-tools/swagger-editor/ .

[25] SmartBear Software, 2019. Swagger UI, https://swagger.io/tools/swagger-ui/ .

[26] SmartBear Software, 2019. OpenAPI Extensions, https://swagger.io/docs/specification/openapi-extensions/ .

[27] Sinclair Target. Whatever Happened to the Semantic Web? . 27 May 2018, https://twobithistory.org/2018/05/27/semantic-web.html .

[28] Keshav Vasudevan, 2018. OpenAPI-Driven API Design, https://swagger.io/blog/api-design/openapi-driven-api-design/ .

[29] W3C, 2015. Linked Data, https://www.w3.org/standards/semanticweb/data .

[30] z@apiary.io. API Blueprint Specification, Format 1A revision 9 . https://apiblueprint.org/documentation/specification.html .