SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
TECHNICAL UNIVERSITY OF CRETE

Thing Descriptions for the Semantic Web of Things

Aimilios Tzavaras

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science.

Chania, April 2022

Abstract

The Web of Things (WoT) initiative aims at unifying the world of interconnected devices over the Internet. The Web of Things (WoT) Architecture is a recommendation of W3C that suggests a model for integrating Things (e.g. devices) in the Web. In addition to W3C, OpenAPI specification provides a method for documenting RESTful services, so that a client can comprehend their purpose and use them in applications. This work applies the OpenAPI service description framework to Web objects (i.e. Things). As a result, OpenAPI descriptions of Web Things provide complete documentation of the services exposed by Things and their capabilities. The resulting descriptions can be converted to ontologies in order to allow a machine to better understand the inherent meaning of Thing descriptions and interact with them. Then, Thing descriptions exposed on the Web can be easily discovered, queried by Semantic Web query languages (e.g. SPARQL) and checked by reasoners (e.g. Pellet). The approach is compared to the WoT Thing Description (TD) information model of W3C in terms of completeness of each representation. This work also proposes an implementation of a proxy service for the Web of Things based on the principles suggested by the WoT Architecture model of W3C. The implementation is compared against existing WoT implementations selected from the Web based on the requirements defined by the W3C WoT Architecture.

Acknowledgements

This thesis would not have been possible without the help of several people who contributed in the preparation and completion of this study.

I would really like to express my sincere appreciation to my advisor, Professor Euripides G. M. Petrakis, for his continuous guidance, advice and support through all the stages of this thesis. I am extremely grateful for being given the opportunity to work on this very interesting field of research.

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

I would also like to thank Associate Professor Michail G. Lagoudakis and Associate Professor Vasileios Samoladas for their constructive comments and participation in the evaluation committee.

I would also like to thank all the members of the Intelligence Lab - especially Konstantinos Tsakos and Fotios Bouraimis - for their excellent communication and generous support.

Most of all, I would like to thank my family and my partner, Inesa, for their enormous help, understanding and support.

Contents

List of Figures

List of Tables

1

Introduction

Nowadays, devices have become part of people's daily lives in a variety of fields, such as healthcare, transportation, agriculture, education, environmental purposes, monitoring, physical exercise and many other application domains. Smart cities, smart buildings and smart factories are based on Internet of Things (IoT) devices and companies constantly develop IoT applications. Today, there are more than 20 billion interconnected devices in the world and the number is still growing rapidly.

The use of Web technologies is now being applied for the development of services and applications in the IoT field. Application Programming Interfaces (APIs) have now dominated the Web. Research has led to the conclusion that the interconnection of devices can be facilitated using existing Web technologies. The Web of Things (WoT) initiative [1] is an evolved version of the Internet of Things that aims at unifying the world of interconnected devices over the Internet. The term Thing may refer to any device: a temperature or proximity sensor, a window actuator, a coffee machine, a smart TV, a Wi-Fi connected garage door or a smart car. WoT suggests that each Thing should be published on the Web, thus advertising its identity and properties. As a result, a Thing can be discovered by Web search engines and reused in different applications.

Cloud Computing [2] allows access to unlimited computing resources that could be managed effectively. Individuals and organizations can make use of scalable IT infrastructures at lower costs while processing power can be accessed based on demand and budget allowance. These advantages make Cloud Computing an ideal application development environment for the IoT and the WoT.

WoT and cloud computing are complementary technologies. Cloud services can be built around IoT devices based on the principles of WoT. As the number and diversity of cloud providers and IoT devices are increasing, the need for standardizing technologies that publish WoT applications and services to developers is becoming of crucial importance for their adoption and market success. In this context, Web services exposed by IoT devices should be properly described and documented, so that any authorized client (i.e. user or service) can use them.

1.1 Background and Motivation

WoT is a relatively new concept. There is no universal application layer protocol to enable Things and services to communicate. Devices may implement any protocol from a wide range of application-specific protocols (e.g. Bluetooth, MQTT, ZigBee, LoRa, etc.). WoT suggests that communication should be protocol-independent. In fact, IoT protocols should be translated to a common Web protocol (e.g. HTTP) to enable communication; in this way, implementations do not depend on particular IoT protocols. Existing standards (such as HTTP and REST APIs) should be adopted to implement the integration of Things (e.g. devices) within the Web. The fact that Web technologies are now very popular in the world of programmers facilitates the development of new frameworks and tools for WoT.

The Web of Things (WoT) Architecture recommendation of W3C [3] defines an abstract architecture and sets the requirements for interacting with Things in the Web using the REST architectural style [4]. The WoT Thing Description (TD) of W3C is a JSON template representation of Thing properties (e.g. purpose, data types and operations). TDs are used to expose Thing metadata on the Web, so that other Things or clients (i.e. services or users) can interact with them. In fact, TDs are gradually becoming popular; the effort is supported by a set of developer tools and a list of candidate implementations[1]. The WoT Architecture suggests that TDs should be extended with semantics to enhance their information content and make them machine-understandable. The resulting representation is JSON-LD [5], which is actually equivalent to an ontology. The motivation for using ontologies to describe Things is that they are closer to the way machines analyse and comprehend the content of TDs. It is, therefore, easier for a machine to discover similarities in TDs and search the Web for Things with the desirable properties (e.g. using ontology query languages, such as SPARQL [6]) or, apply semantic inference in order to detect services with inconsistencies (e.g. using ontology reasoners, such as Pellet [7]). Enabling automatic synthesis of Things in applications could be possible as well.

Nowadays, thousands of APIs are provided publicly (without any cost). For instance, 17 million developers use the Postman[2] API Platform to build, test, debug and monitor APIs, and thousands of these APIs are available on Postman's Public API Network. ProgrammableWeb[3], the most well-known Web service directory, has registered more than 24000 public services. Alongside, research has focused on how to efficiently describe any aspect of a service (functional or non-functional) over the years [8]. UDDI (Universal Description Discovery and Integration) [9], WSDL (Web Service Description Language) [10] were introduced as a first approach towards the syntactic description of services. SAWSDL [11], OWL-S [12] and other approaches were proposed as an effort to enrich the existing service descriptions with semantics based on Semantic Web technologies. However, many of these approaches became obsolete, as new technologies and architectural styles (REST [4]) emerged. The demand for better service descriptions and consequently effective discovery led to new research efforts, such as

[1] https://www.w3.org/WoT/developers/
[2] https://www.postman.com/
[3] http://www.programmableweb.com/

WADL (Web Application Description Language) [13], Hydra [14], OpenAPI Specification (OAS) [15].

The emergence of REST generated new difficulties in the representation of hypermedia-driven APIs [16]. In fact, hypermedia-driven APIs are consistent with the idea of the dynamic discovery of resources at runtime (referred to as HATEOAS), which is actually a constraint of the REST architectural style. According to the HATEOAS principle, the interaction of clients with REST applications should be driven by hypermedia. Hypermedia controls are used to guide clients on what resources they can retrieve and on what operations (i.e. requests) they may perform. In other words, they can provide clients with the information of available state transitions in an application and show clients how they can perform these transitions. Applications can drive clients to a desired outcome, so they are named as hypermedia-driven applications or APIs. For example, hypermedia controls can be located in the headers of an HTTP request or response or inside a JSON payload in the form of links. These links can instruct clients on how to retrieve additional resources and also inform them on how these resources are related to the original ones (e.g. an additional resource could be documentation for the original resource).

To increase the adoption of services for WoT by software developers and enterprises, these must be accompanied by appropriate service descriptions. Services need to be exposed using API specifications and thus introduce themselves to users or other services in order to be able to use them. In other words, services should make their APIs and functionality public and accessible to others. Likewise, the functionality of devices and thus their exposed services must be properly defined and documented in detail, in order to be useful to users and services. Developers and cloud providers usually describe their services in plain text (i.e. code documentation). However, service definitions should no longer be provided in plain text format, but in a format that is understandable by both humans and machines. In addition, web services need to be described in a way that eliminates ambiguities, so that they can be uniquely identified by users or machines. As long as devices and their services are accurately defined, they can be discoverable and easy to use when published on the Web. Consequently, the need for efficient and accurate service description and discovery for Things seems to be a significant challenge for the WoT research area.

1.2 Problem Definition

The interconnection of Things is commonly supported by sensor-specific protocols (e.g. Bluetooth, Zigbee, etc.) rather than by HTTP directly. The Web of Things approach by W3C and other investigators suggests that the interconnection of Things does not depend on peculiarities of IoT protocols that would require an extra layer of complexity in an implementation. Ideally, the Web of Things approach requires that Things receive (each one) an IPv6 address and have a Web server installed. However, this is not always possible, especially for resource-constrained devices. Although lightweight Web servers[4]

[4] https://www.linux.com/news/which-light-weight-open-source-web-server-right-you/

can be embedded in small devices, IoT devices usually feature limited resources and the solution is not optimal in terms of autonomy and cost. A workaround to this problem is to deploy a Web proxy on a server (or on a gateway) that keeps the virtual image of each Thing (e.g. a JSON representation). Web proxy implements a directory (e.g. a database) with all Things (i.e. instances, their types, descriptions and services supported). Things become part of the Web and can be accessed via their Web Proxy (i.e. they can be published, consumed, aggregated, updated and searched for). Web services exposed by Things can then be discovered by users or other services. Therefore, Thing descriptions become an important component of any architecture intended for the WoT, so that devices and their APIs become discoverable. Moreover, the Semantic Web of Things (SWoT) [21] is the semantic extension of WoT that allows Things to become machine discoverable on the Web using Semantic Web tools, such as SPARQL.

The focus of this work is: a) on designing and implementing a Web Proxy service for exposing Things on the Web, and b) on defining Web Things and their functionality, by providing complete documentation of the services exposed by Things. A Web Thing Proxy service must be based on the principles of the WoT initiative and more specifically on the requirements of the relevant W3C specifications (i.e. Web of Things Architecture, Web Thing Model, etc). The proposed approach for defining the functionality of Things should be universal (i.e. applicable to any Thing); it should describe all the operations offered by a Thing regardless of its physical or other characteristics. The detailed description of these services allows the implementation of efficient and accurate service discovery mechanisms for Things and their functionality. Provided that devices themselves can be considered as Web services, they need to be described in a way that eliminates ambiguities and provides descriptions that are both uniquely defined and discoverable. This would allow users and machines to know all the service operations they can perform on a device and how to interact with it. Therefore, a description language is required that would allow for both syntactic and semantic description of services exposed by Things. Existing description languages and approaches intended for describing device functionality are reviewed and their features and characteristics are analysed in order to determine their suitability for the description of Things and their services.

OpenAPI (formerly known as Swagger) [15] suggests a description format for REST APIs. It is a mature framework providing both, human and machine-readable descriptions of Web services. It can be enriched with text descriptions, so that users can easily discover and understand the service and interact with it. Given an OpenAPI service description, a client can easily understand and discover the functionality of a Thing and how to interact with it with minimum implementation logic. OpenAPI provides the needed information about service endpoints, service operations, the exchanged message formats and the conditions which need to be fulfilled before invoking the service. Finally, OpenAPI is supported by a complete tools palette[5] (e.g. it provides tools for interactive documentation and client SDK generation).

OpenAPI is mainly focused on human-readable service descriptions. An OpenAPI service description needs to be formally defined and its content be semantically enriched in order for a machine to understand the meaning of the description. Semantic OpenAPI [17] has proposed that OpenAPI service descriptions can be semantically

[5] https://openapi.tools/

annotated by associating OpenAPI entities to entities of an ontology (e.g. domain ontology). This work utilizes the Semantic OpenAPI approach, so it can be adopted for the description of devices and their exposed REST APIs. It proposes a particular OpenAPI Thing template approach for describing any device as a service along with a mechanism for generating OpenAPI Thing descriptions based on user input given in JSON format. It is an alternative to the TD of the W3C Web Architecture and offers a more informative and elaborate mechanism for the description of Things exposing their functionality in the Web as RESTful services. However, both representations share common features and serve the same purpose (i.e. discovering Things in WoT) which are reviewed in this work.

The reasons that cause ambiguities in OpenAPI descriptions were analysed in [17]. For example, similar to Thing Description, the same property may appear with different names within the same OpenAPI document or, its meaning may not be defined at all. [17] showed that, in order to eliminate ambiguities, each ambiguous property must be semantically annotated and mapped to a semantic model (e.g. a vocabulary or ontology). This work suggests that it is plausible to transform semantically enriched OpenAPI descriptions to ontologies, as this enables application of state-of-the-art querying languages (e.g. SPARQL) for service discovery and of reasoning tools (e.g. Pellet) for detecting inconsistencies and inferred relationships in service descriptions. OpenAPI ontology [17, 18] incorporates features of Hydra [14] for modeling service operations and SHACL [19] for validating Schema descriptions against the ontology.

1.3 Proposed Solution

This work proposes an implementation of a proxy service for WoT and also proposes the OpenAPI Specification[6] for the description of Things in the WoT. It follows some of the basic requirements of the WoT Architecture, which is a W3C recommendation, but it does not adopt TDs for the description of Things and the interaction with them. Although the Web Thing Model submission of W3C is not a recommendation, it laid the foundations for our implementation of a Thing description approach and a REST API implementation for the WoT. This work builds on the REST API (i.e. endpoints, payloads, etc) proposed in the Web Thing Model submission and adopts the OpenAPI Specification as the main description language for devices and their exposed services. OpenAPI specification provides a method for documenting RESTful services so that a user or another service can comprehend their purpose and reuse them in applications. This work applies the OpenAPI service description framework to Web objects (i.e. Things) using a common description template. As a result, OpenAPI descriptions of Web Things provide complete documentation of the services exposed by Things and of their capabilities. The resulting descriptions can be converted to an ontology to allow a machine to better understand the inherent meaning of Thing descriptions and interact with them. Then, Thing descriptions exposed on the Web can be easily discovered, queried by Semantic Web query languages (e.g. SPARQL) and checked by reasoners (e.g. Pellet) for

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

consistency or, for inferencing hidden properties. The approach is compared to the WoT Thing Description (TD) model of W3C in terms of completeness of the representation.

1.4 Contributions of the Work

The major contributions of this thesis can be summarized as follows:

- It proposes an implementation of a Web Thing Proxy service that exposes Things on the Web. This is a novel implementation based on the WoT Architecture of W3C and builds on the REST API proposed by the Web Thing Model of W3C; it implements all the model's operations on Things using HTTP. It is compared against existing implementations from the Web.
- It presents a comprehensive review of existing approaches used for service description. This review also includes related technologies such as SOA, cloud and semantic Web technologies. A critical analysis of each technology is presented in this review, by highlighting the characteristics that restrict the adoption or limit the usefulness of each approach.
- It proposes that the Semantic OpenAPI Specification of [17] should be used for the description of Things and of their functionality. It is an extension to the OpenAPI specification and it eliminates any ambiguities in service descriptions by semantically enriching them. Therefore, it provides human and machine readable service definitions.
- It demonstrates how Semantic OpenAPI can be actually applied to describe Things and their functionality without ambiguities. A specific template description for Things is proposed based on the idea of mapping Thing properties to OpenAPI properties.
- It introduces a mechanism for generating OpenAPI Thing Descriptions from user input. A user that is aware of the device characteristics can provide all the necessary information for the device and its functionality, including the endpoints, the HTTP methods, the data schemas (e.g. request body and response body schemas) required for the service operations, etc. Therefore, all this information can be included in the OpenAPI definition of the Thing, enriched with semantic annotations that further describe concepts such as sensor, actuator, temperature, etc.
- It proposes that OpenAPI Thing descriptions can be transformed to ontologies by adopting a mechanism provided from previous work. Thus, descriptions can take advantage of Semantic Web query languages for service discovery and reasoners for consistency or for inferencing hidden properties. A machine will be allowed to better understand the inherent meaning of Thing descriptions and interact with them. Thing descriptions exposed on the Web can be easily discovered by both users and machines, thus realizing the vision of SWoT. Some indicative SPARQL query examples on the generated ontologies and their results are also demonstrated.

- It presents a critical comparison between the W3C Thing Description (TD) approach and the OpenAPI Thing description approach proposed in this work, based on their capability to fully describe the functionality of a Thing. The two approaches are compared in terms of completeness of the representation.

1.5 Thesis Outline

Chapter 2 provides a brief introduction to basic concepts and technologies which are used throughout the thesis. In addition, it summarizes and presents the most common approaches for the description of Cloud services, including approaches used for describing Things and their functionality. Chapter 3 presents a Web Proxy implementation for the WoT and a Web service implementation of the Web Thing Model's REST API. This service implementation is compared with existing WoT implementations in Chapter 3 based on the requirements of the Web Thing Model; W3C presented this model prior to introducing the WoT Architecture. Chapter 4 proposes a detailed solution for Thing descriptions. Firstly, it identifies the reasons that led us to the adoption of the OpenAPI Specification as well as the Semantic OpenAPI Specification. It demonstrates how our approach can be applied to real-world devices by providing Thing description examples for specific devices. Then, it introduces the common OpenAPI Thing template that applies to all Things and their functions, regardless of their characteristics. Moreover, it describes a mechanism for automatically generating Thing descriptions based on the input given by a user in JSON format. Chapter 5 reviews the W3C Thing Description approach and our proposed solution, and compares the two approaches in detail, taking into consideration the JSON descriptions and the ontologies provided by every approach. Chapter 6 compares the WoT implementations described in Chapter 3 based on the requirements of the WoT Architecture of W3C (i.e. a recommendation), thus reconsidering the results of Chapter 3. Finally, conclusions and issues for future work are discussed in Chapter 7.

2

Background and Related Work

2.1 Web of Things (WoT)

The Web of Things (WoT) concept aims at integrating objects (Things) within the Web so that they can become part of the Web and communicate with each other (and also with clients). Devices used in everyday life (such as smartphones, cars, coffee machines, washing machines, humidity sensors, etc) should communicate with the Web using existing Web protocols rather than application-specific protocols. For instance, common Web protocols (e.g. HTTP, HTTPS, Websockets, etc.) can be used for the communication of Things with applications, while data-interchange formats (JSON, XML, etc.) can be used for the representation of Things (i.e. of their functionality, identity, purpose and of data they provide). Even simple technologies like HTML (Hypertext Markup Language) could be used for the representation of Things in a webpage, for example, to create User Interfaces (UIs) for Things [20].

The actual functionality that Things offer (i.e. by means of Web services) can be implemented using the REST architectural style; each Thing may expose a REST API that implements the operations supported by the Thing. Alongside, WoT may utilize additional useful technologies for the interaction with Things such as API security mechanisms (e.g. Basic authentication, API key authentication, OAuth2.0 protocol) for authentication and authorization, mechanisms for service composition and synthesis of Things in applications (e.g. Node-RED), etc. To be scalable, WoT implementations can be deployed in the cloud. WoT leverages Cloud computing which is capable of providing IoT solutions that may involve thousands to millions of devices.

Things may use any protocol (e.g. ZigBee, Wi-Fi, Bluetooth, 6LoWPAN, 3/4/5G, NFC) to communicate. HTTP (HyperText Transfer Protocol) is an application layer protocol that is widely used to support RESTful communication of services in the cloud and over TCP. Due to its high overhead (i.e. high power consumption, header size and complexity of handshakes), HTTP is not suitable for the IoT and resource-constrained devices that exchange small amounts of information and are not connected to a sustainable power source. It implements a request-reply communication where the server responds to the requests of a client. This is good for communication between services but not for devices that send information to a server without a prior request. CoAP is a lightweight protocol over UDP. It is similar to HTTP (e.g. with a similar command set) but for resource-constrained environments. In the following, we assume that communication in the Web of Things is realized using an HTTP protocol following the basic assumption of the Web of Things and W3C.

As long as Things get connected to a network, it is plausible to assume that Things also connect to a protocol translation service whose role is to convey Thing related data (i.e. identifier, description and payload) to the application using HTTP and JSON. Communication of Things and services in WoT relies on common IoT protocols. Things can become part of the Web and be accessible via a Web Proxy. The operations that Things may support can be regarded as Web services that can be advertised, discovered and used by clients (users or services) that search for them on the Web.

The Semantic Web of Things (SWoT) [21] is the semantic extension of WoT that suggests the design of interoperable IoT services on the Web using Semantic Web technologies. SWoT allows Things to become machine discoverable on the Web using Semantic Web tools such as SPARQL. It also enables applications to share content and services beyond their limits and to develop new applications as a composition of existing ones. A Semantic WoT architecture [41] (from a previous work) built upon principles of SOA (Service-Oriented Architecture) design and deployed on the cloud is described in Section 3.6.

The concept of WoT has received considerable attention from IoT vendors and from many investigators over the past few years. The WoT Working group[7] is an ongoing effort to create standards-track specifications and test suites. The Thing Description specification of W3C (recommendation) [22] defines how Things and their functionality can be represented using JSON Thing Descriptions (TD information model). The results of the W3C WoT research effort are summarized by the WoT Architecture which suggests a list of possible operations to be supported by a WoT implementation[8].

2.2 Web of Things (WoT) Architecture

The Web of Things (WoT) Architecture recommendation of W3C proposes an abstract architecture for W3C WoT. The document includes terminology and use cases (i.e. different application domains for WoT) and sets the requirements for the interaction with Things in the Web using RESTful APIs. The recommendation does not bind to any application and it does not depend on specific communication protocols. In addition, it does not describe a specific implementation or mechanism, but an abstract architecture approach for the Web of Things.

[7] https://www.w3.org/WoT/wg/
[8] https://www.w3.org/WoT/IG/wiki/Implementations

The WoT Architecture defines an interaction model that describes the interaction of a consumer (i.e. client) with Things. Things may offer particular Interaction Affordances (i.e. metadata showing how a client can interact with Things) such as Web links, Properties, Actions and Events. Properties are used to define the state that Things expose (e.g. humidity value). Actions are used to describe the functions that Things may perform (e.g. a smart window that opens and closes). Events are used to represent the transition of the Thing's state (e.g. the state property of a smart window turning to *open*). Interaction Affordances can be described using JSON Thing Descriptions (TDs).

The term *Events* is used in WoT Architecture to represent Thing state transitions. An *Event* is defined by the recommendation as "*An interaction affordance that describes an event source, which asynchronously pushes event data to Consumers (e.g. overheating alerts)*"[9]. In other words, an event source sends event data from the Thing to the subscribed clients. Events are closely related to subscriptions. The WoT Architecture of W3C defines operations for subscribing and unsubscribing to events (e.g. overheating of a device), as highlighted in Table 2.1. That is, clients can only subscribe or unsubscribe to an event and receive asynchronous notifications (alerts) when the event occurs. There are no other operations related to events. A subscription is the result of subscribing to a specific event related to a Thing. A client could subscribe, for example, with a Webhook callback URI.

The WoT Architecture also proposes the use of hypermedia controls for the interaction of clients with Things. Two kinds of hypermedia controls are used in the W3C WoT: Web *links*[10] and Web *forms*. Web links are referred to as *"the well-established control to navigate the Web"*. That is, they provide navigation affordances that allow clients to discover linked resources. For example, a link may provide a link target attribute and a link relation type to relate a Thing with another resource that is represented by a hyperlink. Web forms are referred to as "*a more powerful control to enable any kind of operation*". That is, they allow clients to perform particular operations that may even change the state of a Thing (e.g. turn on a device) and not just navigate to discover resources. The recommendation highlights that web links are already used in other IoT standards and IoT platforms[11] such as CoRE Link Format[12], OMA LWM2M[13], and OCF[14], whereas form is a new concept. Besides W3C WoT, the concept of forms is introduced by the Constrained RESTful Application Language (CoRAL)[15] defined by the IETF.

Links can be followed by both users and machines. A link may include (at least) the URI of a resource (i.e. target resource) which can be followed to fetch the representation of a resource. The recommendation highlights that Web links are used in the WoT to discover Things and also to express relations to other Web documents. Hypermedia controls such as links are discovered during the interaction of the Web client with a server. A link comprises: a) a link context, b) a relation type, c) a link target, and d) optionally target attributes.

[9] https://www.w3.org/TR/wot-architecture/#terminology
[10] https://httpwg.org/specs/rfc8288.html
[11] https://www.w3.org/TR/wot-architecture/#dfn-iot-platform
[12] https://datatracker.ietf.org/doc/html/rfc6690
[13]http://openmobilealliance.org/release/LightweightM2M/V1_1-20180710-A/OMA-TS-LightweightM2M_Core-V1_1-20180710-A.pdf
[14] https://openconnectivity.org/developer/specifications/
[15] https://datatracker.ietf.org/doc/html/draft-hartke-t2trg-coral

Forms allow Web clients to perform specific operations to manipulate the state of a Thing. Clients are instructed on how to perform these operations by sending a proper request to their submission target. The recommendation states that "*W3C WoT defines forms as new hypermedia control*". A form comprises: a) a form context, b) an operation type, c) a submission target, d) a request method, and e) optionally form fields. In other words, Web forms in the WoT are used to perform operations on Things.

The recommendation also defines a number of well-known operation types for the Web of Things. Web forms can specify how these operations can be performed. The operation types are presented in Table 2.1. The operations are related to Thing properties (e.g. an operation to read a property or an operation to update a property), to Thing actions (i.e. an operation to invoke an action) and to events related to Things (e.g. an operation to subscribe to an event). In relation to Thing properties, a client may also "*observe*" a specific property of a Thing. As a result, whenever a Thing property is updated, a client can be notified of the new value(s) of the property. To stop these notifications, a client can simply "*unobserve*" the selected (i.e. observed) property. The *observe* and *unobserve* operations refer to the CoAP protocol[16] that allows a client to register to a resource and thus get notified of its changes over time. More specifically, a CoAP client (called *observer*), which is interested in the state of a resource at any given time, can send a modified CoAP GET request to register to the resource and create an observation relationship between the client and the server resource. After the registration, the client can get notifications over a period of time.

[16] https://tools.ietf.org/id/draft-ietf-core-observe-01.html

Operation Type	Description
readproperty	Identifies the read operation on Property Affordances to retrieve the corresponding data.
writeproperty	Identifies the write operation on Property Affordances to update the corresponding data.
observeproperty	Identifies the observe operation on Property Affordances to be notified with the new data when the Property was updated.
unobserveproperty	Identifies the unobserve operation on Property Affordances to stop the corresponding notifications.
invokeaction	Identifies the invoke operation on Action Affordances to perform the corresponding action.
subscribeevent	Identifies the subscribe operation on Event Affordances to be notified by the Thing when the event occurs.
unsubscribeevent	Identifies the unsubscribe operation on Event Affordances to stop the corresponding notifications.
readallproperties	Identifies the readallproperties operation on Things to retrieve the data of all Properties in a single interaction.
writeallproperties	Identifies the writeallproperties operation on Things to update the data of all writable Properties in a single interaction.
readmultipleproperties	Identifies the readmultipleproperties operation on Things to retrieve the data of selected Properties in a single interaction.
writemultipleproperties	Identifies the writemultipleproperties operation on Things to update the data of selected writable Properties in a single interaction.

Table 2.1: Well-known Operation Types for the Web of Things

W3C TD is a central building block of the WoT Architecture. It is used to define the functions as well as the interfaces of devices. TD can provide the entry point for discovering services as well as resources related to a Thing. In other words, TD exposes Thing metadata on the Web. The WoT Architecture also suggests that TDs are hosted in a directory service (on a gateway or the cloud) which actually provides a Web interface for registering and searching for Things. The architectural aspects of a Thing (i.e. Behavior, Interaction Affordances, Data Schemas, Security Configuration, Protocol Bindings) are also included in the recommendation. A detailed specification of the Thing Description is given in the Thing Description recommendation [22].

The Web of Things (WoT) Thing Description document[17] is a recommendation of W3C that describes the model and the representation of Things using TDs. It includes terminology about TDs and presents the TD information model in detail. The document describes the TD representation and serialization format and demonstrates how Things can be represented by Thing Descriptions using examples. TD is a short and abstract description of a Thing, including its functions and interfaces. The JSON representation of a TD can be enriched with semantic annotations to become machine-understandable. TD's JSON serialization format can be enhanced with a context field (@context) for converting the JSON format to JSON-LD.

The WoT TD Working Draft includes Class definitions for the vocabularies used in the TD information model for semantic annotations and defines temporary namespaces for the vocabularies. The document defines core vocabulary classes that represent basic concepts such as Thing, interaction affordances, properties, actions, events, etc. It also defines the vocabulary classes used to describe data schemas, API security mechanisms and hypermedia controls. Moreover, the document defines the Thing Model, which is the information model of a Thing. It is used as a general template description for a type of Things that have common properties; it is not used to describe a particular Thing instance.

Figure 2.1 illustrates the structure of a TD. It is an abstract description that mainly describes the Interactions, Data Schemes, Security Configuration and Protocol Binding of a Web Thing. More specifically, a TD includes the Thing's name, its unique identifier, its security requirements, a title, an optional human-readable description, and all the interactions supported by the Thing.

[17] https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/

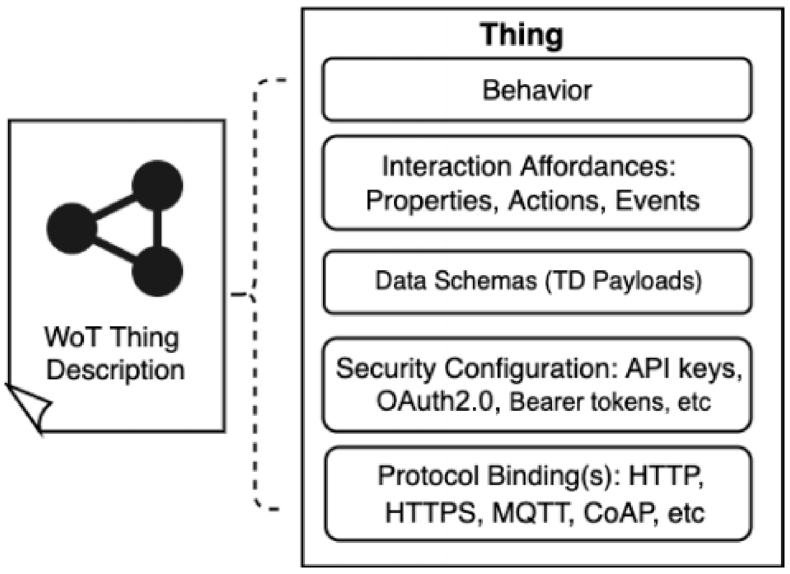

WoT Thing Description
Thing
Behavior
Interaction Affordances: Properties, Actions, Events
Data Schemas (TD Payloads)
Security Configuration: API keys, OAuth2.0, Bearer tokens, etc
Protocol Binding(s): HTTP, HTTPS, MQTT, CoAP, etc

Figure 2.1: Thing Description (TD) document structure

Listing 2.1 is an example TD for a smart door IoT device that contains (a) a *@context* attribute which extends the definition with additional vocabulary terms (e.g. using schema.org), (b) the identifier of the device, (c) an indicative title, (d) the security configuration of the service (i.e. Basic Authentication in this example), (e) interactions supported by the smart door; the state property, the lock and unlock actions, the door open event (i.e. the state property of the door turning to open) and, (f) the forms field that describes how each interaction (i.e. operation) can be performed; it specifies the protocol that should be used (i.e. HTTPS) and the operation endpoint. The endpoint to retrieve the last state value of the smart door is specified in the *Properties* object (i.e. in the forms array). The protocols and the endpoints used to execute the lock and the unlock actions are specified by an *Actions* object; the protocol, the endpoint and the subprotocol (i.e. the exact mechanism used for asynchronous notifications) for subscribing to the open event of the smart door are specified by an *Events* object.

The W3C Thing Description specification explains that the HTTP GET method in the *Properties* object is not stated explicitly, as it "*is one of the default assumptions defined by this document*". Similarly, the HTTP POST method used in the *Actions* object is omitted, as it is "*a default assumption for invoking Actions*". The WoT Thing Description is also discussed in Section 5.1 of the present work.

Listing 2.1: Thing Description for a smart door device

```
{
    "@context": "http://www.w3.org/ns/td",
    "id": "urn:dev:ops:32473-WoTSmartDoor-1234",
    "title": "MySmartDoor",
    "securityDefinitions": {
        "basic_sc": {"scheme": "basic", "in": "header"}
    },
    "security": "basic_sc",
    "properties": {
        "state": {
            "type": "string",
            "forms": [{"href": "https://mysmartdoor.example.com/state"}]
        }
    },
    "actions": {
        "lock": {
            "forms": [{"href": "https://mysmartdoor.example.com/lock"}]
        },
        "unlock": {
            "forms": [{"href": "https://mysmartdoor.example.com/unlock"}]
        }
    },
    "events":{
        "opening":{
            "description": "Smart door opens",
            "data": {"type": "string"},
            "forms": [{
                "href": "https://mysmartdoor.example.com/open",
                "subprotocol": "longpoll"
            }]
        }
    }
}
```

2.3 Web Thing Model (W3C submission)

Prior to presenting the WoT Architecture recommendation, W3C had introduced the Web Thing Model [23]. Although not a standard, the Web Thing Model was more concrete than the WoT Architecture and specified the requirements that software or hardware vendors (who create products for the Web of Things) should meet. It was an early attempt to define a Thing description format for the Web of Things. The Web Thing Model defined (a) the information model for Thing descriptions based on JSON and, (b) a set of operations (i.e. services) together with their corresponding RESTful interface for accessing Things on the Web. Things become JSON objects representing their identity (i.e. a URI), properties, functionality (e.g. actions that Things may execute) and state information. They expose a RESTful interface on the Web in order to facilitate their interaction with other Things and services over the Web. Things are published on the Web (i.e. expose their identity, properties and functionality) so that they can be accessed by other services and be used in Web applications. However, the Web Thing Model was not based on Thing Descriptions (TDs) as defined by the WoT Architecture.

More specifically, the Web Thing Model describes the requirements, a set of operations and a specific REST API that should be followed by WoT implementations. The model initially defines some basic requirements (e.g. "*A Web Thing MUST at least be an HTTP/1.1 server*") and proposes conformance rules and terminology. It also defines an information model for Things (i.e. the structure of JSON payloads for every operation of a Thing) and proposes a RESTful API based on specific resources related to Things (e.g. Properties Resource, Actions Resource) and particular operations offered by Things. The suggested operations support interaction (i.e. create, retrieve, update, delete operations) with the resources related to Things. The specification suggests that semantic extensions should be adopted so that detailed vocabularies (e.g. *schema.org*) further describe Things and their features; it also describes how these extensions should be implemented. More specifically, semantic extensions should be realized by referencing a JSON-LD context in the HTTP Link Header using the *http://www.w3.org/ns/json-ld#context* link relation, as defined in the JSON-LD specification.

A Thing is identified by its resources, namely, a Web Thing Resource, a Model Resource, a Properties Resource, an Actions Resource (as long as the Thing supports actions), a Things Resource and a Subscriptions Resource. The *Web Thing Resource* is "*the root resource of the Web Thing Model*". It is used to provide a short and abstract description of a Web Thing in JSON format. This description can be updated by the client. The *Model Resource* defines the values of Things properties as well as the values of actions that can be performed on Things. The model description can also be updated by the client. The *Properties Resource* defines the properties of a Thing in general (e.g. pressure, temperature) and describes measurements related to a specific Thing property (e.g. temperature values provided by a temperature sensor) or the internal state of a Thing (e.g. the state of a smart door). The *Actions Resource* is used to define all the allowed actions on a Thing and describe the action executions of a Thing, such as execution commands (e.g. a command sent by a client to a window actuator to open). The specification introduces the *Subscriptions Resource* suggesting that clients (i.e. users or services) can subscribe to the properties and/or actions of Things, to be notified of new values or values changes they are interested in. For instance, users and services can subscribe to the humidity property of a specific humidity sensor and get notified of any changes of this information (i.e. new humidity value). Additionally, the Web Thing Model utilizes the *Things Resource* for the description of specific operations on Things such as registration of new Things to an application (i.e. by sending the JSON representation of the Thing) or retrieval of a list of registered (e.g. to a proxy) Things. Notice that the Things Resource is different from the Web Thing Resource.

Based on the Web Thing Model, the data exchange format for Web Thing resources is based on JSON. The exact formatting depends on the particular services of the proxy service (e.g. NGSI[18]). The Web Thing Model introduces a list of operations which (in part or in full) can be offered by a Thing. Concerning the Web Thing Resource, the model allows an operation to retrieve or update the description of a Thing in JSON. Similarly, for the Thing Model (or for Properties Resource), the Web Thing model allows operations to retrieve or update the properties or actions (their values or state information respectively) on a Thing. Concerning the Actions Resource, the model allows

[18] https://www.fiware.org/2016/06/08/fiware-ngsi-version-2-release-candidate/

an operation to retrieve the actions that a Thing may perform and, in addition, an operation for sending a command to a Thing to execute an action and operations to retrieve past action executions. In relation to Things Resource, the model may allow an operation that registers a Thing or an operation that retrieves all registered Things. Finally, in relation to Subscriptions Resource, the model allows an operation that creates a new subscription to a Web Thing resource or, an operation that retrieves or updates the information of an existing subscription. In the following, the set of operations suggested by the Web Thing Model are presented, grouped by resource:

A. *Web Thing Resource*

1) <u>Retrieve a Web Thing</u>: Retrieves an abstract JSON description of a Thing. It is realized by issuing an HTTP GET request to the root URL of a Thing. The root URL of a Thing is its IP address and default port that follows the IP (e.g. http://34.122.93.207:5001/MySmartDoor in the case of a smart door device).

2) <u>Update a Web Thing</u>: Updates the JSON description of a Thing. It requires sending an HTTP PUT request to the root URL of a Thing, containing a JSON object in the request body.

B. *Model Resource*

3) <u>Retrieve the model of a Thing</u>: Retrieves the model description of a Thing by issuing an HTTP GET request to the /model endpoint of the root URL of a Thing.

4) <u>Update the model of a Thing</u>: Updates the model of a Thing (i.e. updates attributes of the Thing model description). It is realized by issuing an HTTP PUT request to the /model endpoint of the root URL of a Thing. The new attribute values can be included in the request body.

C. *Properties Resource*

5) <u>Retrieve a list of properties</u>: Retrieves a list of the Thing's properties. It requires sending an HTTP GET request to the /properties endpoint of the root URL of a Thing.

6) <u>Retrieve the value of a property</u>: Retrieves the current value of a property. It is realized by issuing an HTTP GET request to the /properties endpoint of the root URL of a Thing followed by the specific Thing property name as a path parameter (e.g. rootURL/properties/state for the state of a smart window or rootURL/properties/temperature for a temperature sensor).

7) <u>Update a specific property</u>: Updates the current value of a property. It is realized by issuing an HTTP PUT request to the /properties endpoint of the root URL of a Thing followed by the name of the property. The new property value (or values) is included in the request body.

8) Update multiple properties at once: Updates the values of multiple properties. It is realized by sending an HTTP PUT request to the /properties path of the root URL of a Thing followed by a path name. This operation manages to update the values of multiple properties of a specific Thing (e.g. the temperature and the humidity of DHT22 sensor) using a single HTTP request. An array of values is included in the request body.

D. Actions Resource

9) Retrieve a list of actions: Retrieves a list of the Thing's actions. It is realized by sending an HTTP GET request to the /actions endpoint of the root URL of a Thing. It is meant to return an array of descriptions for the actions that the Thing may perform (e.g. open and close for a smart window device).

10) Retrieve recent executions of a specific action: Retrieves all recent executions of a specific action of the Thing. It is realized by sending an HTTP GET request to the /actions endpoint of the root URL of a Thing followed by the specific action name as a path parameter. For example, a GET rootURL/actions/{actionName} (e.g. rootURL/actions/open) will return an array of the recent executions of a specific action on a device (e.g. opening the smart window), including information about the status of the action execution and a timestamp.

11) Execute an action: Executes a specific action by sending a command to a Thing. It is realized by issuing an HTTP POST request to the /actions endpoint of the root URL of a Thing followed by an action name in a path parameter (e.g. POST rootURL/actions/on to turn on a smart air conditioner).

12) Retrieve the status of an action: Retrieves the status of a specific action execution using its identifier as a path parameter. It requires sending an HTTP GET request to the /actions endpoint of the root URL of a Thing followed by the name of the action and the execution identifier as a path parameter (e.g. GET rootURL/actions/open/142).

E. Things Resource

13) Retrieve a list of Web Things: Retrieves a list of Things proxied by a Web Thing (i.e. they are registered to the proxy). It is realized by sending an HTTP GET request to the /things endpoint of the root URL of a Thing.

14) Add a Web Thing to a gateway: Registers a new Thing (i.e. device) in a specific infrastructure (i.e. and in a proxy). It requires sending an HTTP POST request to the /things endpoint of the root URL of a Thing. The request body must contain the JSON Thing description of the new device.

F. Subscriptions Resource

15) Create a subscription: Creates a new subscription, so that a client may subscribe to a specific resource of a Thing. That is, a user or service may subscribe to specific Thing properties or actions. According to the Web Thing Model, subscriptions are ideally supported using custom callbacks (i.e. Webhooks) which are naturally supported by the Websocket protocol.

16) Retrieve a list of subscriptions: Retrieves a list of subscriptions made to a specific resource of a Thing. It requires sending an HTTP GET request to the /subscriptions endpoint of the root URL of a Thing.

17) Retrieve information about a specific subscription: Retrieves a specific subscription made to a resource of a Thing using its subscription identifier *(Subscription-ID)*. It requires sending an HTTP GET request on the /subscriptions endpoint of the root URL of a Thing followed by the subscription identifier as a path parameter (e.g. GET rootURL/subscriptions/5a72eb4d883af1b95ac9f710).

18) Delete a subscription: Deletes a specific subscription made to a resource of a Thing using its subscription identifier. It issues an HTTP DELETE request to the /subscriptions endpoint of the root URL of a Thing followed by the subscription identifier as a path parameter.

2.4 Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) is an architecture style that intends to enhance the efficiency, agility, and productivity of an enterprise by designing, developing, deploying and managing systems, based on service orientation [24]. Service orientation is a design paradigm that suggests that all functional components of a system are viewed as services that communicate with each other through well defined interfaces by message passing. A service is essentially a well-defined, self-contained and independent (i.e. of other services) function. That is, a service does not need to be aware of the technical details of another service to interact with it. This is achieved through the implementation of a strictly defined interface that can perform the necessary actions to enable the transmission of data between services, thus facilitating communication. The invoker of a service actually needs to be aware of its interface only and not its implementation. SOA, as an architectural style, does not impose a specific technology for the communication of services. With the emergence of machine communication protocols such as HTTP and representation formats such as XML, RDF and JSON, SOA is becoming the most common approach for building distributed systems (i.e. communicating systems in general) in terms of communicating services.

2.5 REST-based services

Web services technology was initially built on existing standards such as Extensible Markup Language (XML) [25], Simple Object Access Protocol (SOAP) [26], Web Services Description Language (WSDL) [10] and Universal Description Discovery and Integration (UDDI) [9]. XML [25] was selected, due to its popularity at the time, as the main data format for machine to machine communication. Fielding suggested a different flavor of Web Services, introducing REpresentational State Transfer (REST) architectural style [4] in 2000. REST defines a set of architectural principles, based on which Web services are designed to focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. The primary abstraction of information in REST is a resource. A resource is anything important enough to be referenced as a thing in itself, such as a document or image, a collection of other resources, a non-virtual object (e.g. a cat). Resources can either be static (i.e. like a book) or dynamic, like a news report (i.e. it always changes, but still it is a resource). REST uses a resource identifier (URI) to identify the particular resource involved in an interaction between components. REST has gained massive adoption, including Cloud Services, compared to other approaches (e.g. SOAP, WSDL). REST-based services are actually simpler to express, faster to process and make efficient use of bandwidth, as they don't require additional parsing for messages and are much less verbose than SOAP-based services. In contrast to SOAP-based services, REST-based services are designed to be *stateless* and also enable *caching* that improves performance and scalability. Moreover, REST-based services may support multiple data formats (e.g. XML and JSON), whereas SOAP-based services are only limited to the use of XML. Nevertheless, the term *REST* has been misused as most Web services that claim to be RESTful (i.e. REST APIs) are actually not. Although in most cases services are based on the REST architecture, they often violate the hypermedia constraint (HATEOAS). It is worth mentioning that Fielding himself highlighted this fact in a blog post[19] and explained that a service is considered RESTFul only if all REST principles are met. The term *Hypermedia API* [27] has emerged to describe services that are implemented incorporating the *hypermedia* constraint.

2.6 Hypermedia-driven APIs

In contrast to first and second generation Web APIs, which are not truly RESTful, Hypermedia-driven APIs (i.e. third generation services) [16] manage to support hypermedia because of the serialization formats they rely on, such as JSON-LD. Hence, hypermedia-driven APIs are actually RESTful APIs. First generation SOAP-based Web

[19] https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

Services do not use hypermedia at all. Second generation services use hypermedia but most of these services do not actually manage to support them, because they depend on formats that do not have built-in support for hyperlinks. Third generation Web APIs leverage Linked Data [28] technologies to become hypermedia-driven and thus truly RESTful services; consequently, they have all the benefits of RESTful services, such as scalability, maintainability, and evolvability. To be more specific, detailed vocabularies (e.g. Hydra core vocabulary [14]) are used along with JSON-LD, to enable the creation of hypermedia-driven APIs.

2.7 Cloud Computing

Cloud Computing allows the allocation of computing resources (e.g. servers, applications, etc) using the Internet. Many users (i.e. individuals and organizations) can have access to the same service or infrastructure at the same time, using the ability of Cloud computing to allow the consumption of their resources on a large scale. In fact, resources in Cloud computing are available on-demand. Therefore, users are allowed to use them based on their needs and be charged exclusively for that use. In addition, the scalability of a Cloud infrastructure makes it easier to serve the demands of the ever-increasing amount of users and applications.

2.7.1 SOA and Cloud Computing

SOA (Section 2.5) and Cloud Computing (Section 2.8) are technologies that can exist separately, since neither depends on the other. However, an integrated architecture with these two solutions can offer many advantages in terms of cost, speed of development, management, and ease of maintenance. Cloud computing provides computational resources, such as software and hardware, for the delivery and deployment of scalable applications and services. However, it doesn't impose any particular method for the efficient use and management of the services that it offers. SOA intends to fill this gap by providing guidelines, principles, and techniques for the development of applications and services, and strictly defines the architecture of service-oriented systems. Cloud services are typically API or service-driven, and thus service-oriented. Therefore, Cloud providers organize their services in directories or service registries to enable discovery of services that best fit the needs of customers as well as reuse and better management of services.

2.7.2 WoT and Cloud Computing

WoT (Section 2.1) and Cloud Computing (Section 2.8) are also separate technologies but they can be complementary as well. Cloud services can expose the functionality of IoT devices, while following the requirements of WoT. For example, a client can communicate and interact with a Thing through the cloud using HTTP. Therefore, WoT benefits from Cloud computing and its features (e.g. scalability), and exposes Things on a large scale, based on the effective and efficient management of resources. Although IoT solutions may incorporate thousands to millions of devices, cloud allows users to take advantage of scalable IT infrastructures (at lower costs) to expose Things or interact with them as consumers. In other words, WoT utilizes existing Web technologies to allow interaction with any IoT device; Cloud computing facilitates and improves this interaction. For instance, consumers can purchase Web services that expose real-world devices (e.g. temperature sensors, smart home actuators, etc) and can rapidly be scaled up or down, depending on their user requirements.

2.8 Semantic Web and Linked Data

Semantic Web

The Semantic Web represents an extension of the existing Web. It offers tools that allow information to be offered not only in the form of natural language documents, but also as machine-readable data. Thus, machine to machine communication is enabled in addition to human to machine communication that is supported by the current Web. In Semantic Web, query languages can be leveraged so that vocabularies and data can be accessed.

The foundation of the Semantic Web is built by the Resource Description Framework (RDF) standard [29], which is a family of specifications developed by W3C. It was originally designed as a metadata model, but it is now used as a general method for conceptual description or for modeling the information of Web resources. In order to achieve the above objectives, RDF adopts a variety of syntax notations and data serialization formats. The RDF standard is also used in knowledge management applications. Resources can be anything, including documents, people, physical objects (e.g. a table), and abstract concepts. A resource is identified by an International Resource Identifier (IRI). IRIs are global identifiers, so that an IRI can be re-used to identify the same thing. An RDF statement expresses a relationship between two resources, in the form of a triple (Figure 2.2). The Subject and the Object represent the two resources being related, the predicate represents the nature of their relationship. Multiple triples build a graph, and multiple graphs build a dataset.

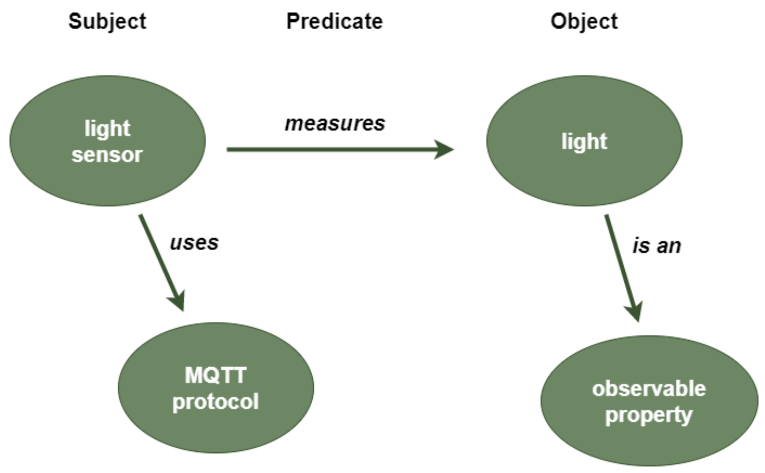


Figure 2.2: RDF data model

RDF Schema (RDFS) [30] provides a data-modeling vocabulary for RDF data. It offers mechanisms for describing classes, class hierarchies, data types, or properties similar to object-oriented programming languages. Unlike RDFS, the Web Ontology Language (OWL) [31] is a family of knowledge representation languages that offer increased expressiveness for describing classes and properties. Among others, OWL allows the definition of relations between classes (e.g. disjointness), equality, restrictions over properties (e.g. cardinality restrictions) and partial order and equivalence relations between properties (e.g. transitive, symmetric properties).

Finally, the Semantic Web offers the ability to query RDF data. SPARQL [6], a recursive acronym for SPARQL Protocol and RDF Query Language, is the W3C recommendation not only for querying and manipulating RDF data but also a protocol to invoke such queries over HTTP and a number of result formats (XML, JSON, CSV).

Linked Data

Data on the Web needs to be structured, machine-readable and also connected with other data. Sir Tim Berners-Lee[20] introduced the term Linked Data [28] in 2006. He described how RDF data should be published on the Web and fulfilled part of the Semantic Web vision. In fact, Linked Data benefits from RDF and from standard Web technologies (e.g. HTTP and URIs) and allows the creation of relations between structured data from different sources. It relates data with URIs so they can be accessed and queried. According to [32], Linked Data refers to data published on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it links to other external data sets, and it can in turn be linked to from external data sets. Berners-Lee also introduced a set of principles[21] for successfully creating and publishing Linked Data.

Although Semantic Web uses ontologies (i.e. sets of RDF triples) to define data and create relations between them, it is not capable of making data machine-readable

[20] https://www.w3.org/People/Berners-Lee/
[21] https://www.w3.org/DesignIssues/LinkedData.html

without using Linked Data. Ontologies provide definitions for various concepts, but data on the Web need to be linked (i.e. associated) to ontology and vocabulary terms. Linked Data manages to map data with semantic models and thus define their meaning. Therefore, data becomes machine-readable and machine-discoverable; machines can search for desired properties using Semantic Web tools (e.g. query languages such as SPARQL). Moreover, data can be represented using JSON-LD, which is in fact JSON format enriched with Linked Data. Therefore, data can be very simply represented using JSON (i.e. probably the most common representation format for REST APIs) and associated to semantic models using Linked Data.

2.9 Interface Description Languages

Service descriptions (or contracts) are a fundamental part of SOA. They define and expose the purpose and functionality of services, so that users or services are able to discover and use them. Moreover, they should describe all the necessary information of a service such as operations, service endpoints, request and response bodies (i.e. schemas), etc. They also set the conditions that have to be satisfied before the services can be executed (e.g. API authentication rules).

Cloud environments provide text documentation for services, so consumers have to browse and read service instructions in plain text format, and then determine if the services actually meet their needs. However, text descriptions are mostly human-readable; machines cannot understand and interpret their meaning. Services in cloud environments should be formally defined in such a way that is understandable by both humans and machines.

In the following, we will discuss the most remarkable approaches that suggest interface description languages for Cloud services intending to define Web services; SOAP-based and RESTful services in particular, since they are the most popular types of Web services today. In this work, we highlight service definition approaches that have been used in order to describe RESTful Web services.

2.9.1 WSDL and SAWSDL

The Web Service Description Language (WSDL) [10] is a service description language based on XML notation. It is used in order to describe the functionality of Web services and show how service requests can be issued by a client. WSDL actually forms the basis for the original Web Services technology platform (Section 2.6). The current version of the WSDL specification is 2.0; it is a W3C recommendation. However, version 1.1 [33] is still used. The structure of a WSDL document is available on the Web[22]. WSDL 2.0 introduced some changes in document structure. WSDL 1.1 described SOAP-based services. WSDL 2.0 introduced a protocol binding for HTTP in order to support the description of RESTful

[22] https://en.wikipedia.org/wiki/Web_Services_Description_Language

services. However, WSDL describes services at a syntactic level. W3C introduced the Semantic Annotations for WSDL and XML Schema (SAWSDL) [12] in a technical recommendation in 2007. SAWSDL extends WSDL using a mechanism that maps service interfaces and message schemas with semantics. Thus, semantics can be added to various parts of a WSDL document (e.g.inputs, outputs, interfaces and operations). More specifically, SAWSDL suggests three extensibility attributes to WSDL and XML Schema elements, so that semantics to semantic models (i.e. vocabularies or ontologies) can be added in a service description.

SAWSDL is criticized for its weaknesses and the fact that it provides only the syntactic information of a service and not any formal semantics [34]. Regarding WSDL, although it is capable of describing both SOAP-based and REST-based services, it is not widely adopted by developers. It is preferred mainly for the description of traditional SOAP-based services. In addition, most tools are offered only for WSDL 1.1; this leads to poor adoption of WSDL 2.0 by developers.

2.9.2 WADL

The Web Application Description Language (WADL) [13] is an interface description language based on XML. It is an alternative to WSDL and it is used to describe HTTP-based Web APIs. WADL is not restricted to XML payloads (i.e. it can also handle JSON and other formats). In fact, WADL is intended to describe RESTful services (Section 2.6). It models the resources provided by the service and also the relationships between them.

WADL has received a lot of criticism for the fact that it is very similar to WSDL and only provides limited support for the description of service resources; it actually offers only syntactic description of the service. It has not managed to attract significant adoption among developers despite the efforts for standardization. An important drawback of WADL is the lack of a mechanism for the semantic annotation of service descriptions.

2.9.3 OpenAPI Specification, RAML, API Blueprint, AsyncAPI

The REST architecture style has gained massive adoption over the years. As a result, several interface description languages have been proposed and they aim at providing efficient and accurate descriptions for RESTful services. OpenAPI Specification, RAML, and API Blueprint are the most commonly used approaches for describing RESTful services. They follow an approach similar to WADL (Section 2.9.2), meaning that everything is bound to the URLs for accessing resources; this contradicts REST's hypermedia constraint that calls for the dynamic discovery of resources at runtime

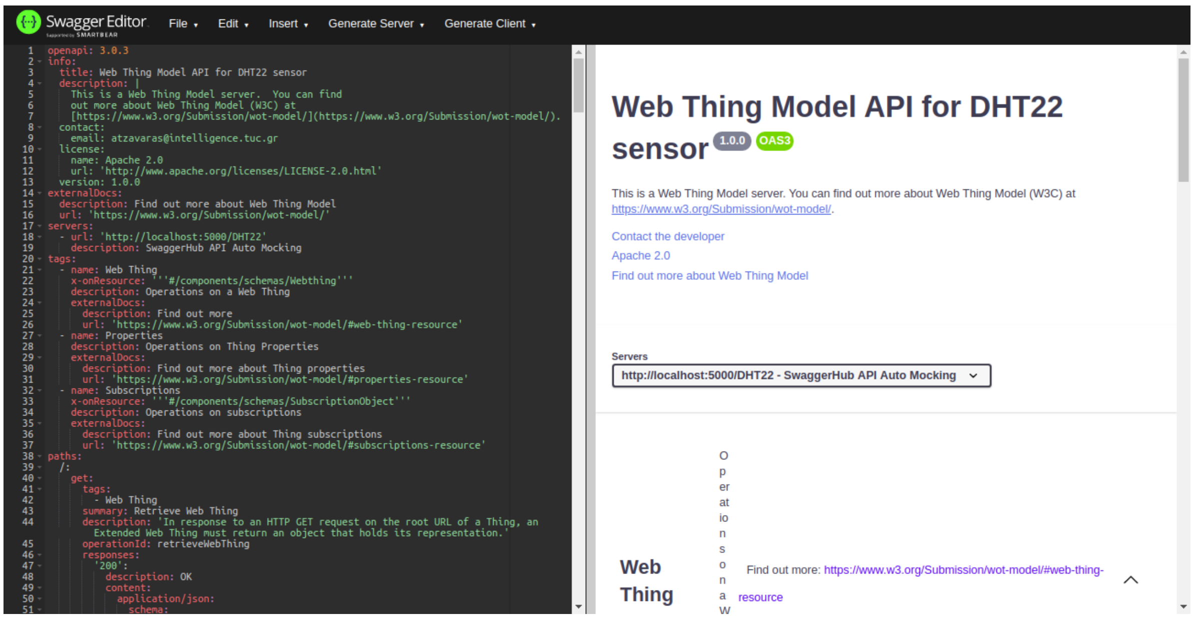
(HATEOAS). However, their adoption is mainly due to the large tooling support they offer covering the whole API lifecycle from design to sharing. Moreover, a new, protocol-agnostic service description approach, AsyncAPI, aims at describing event-driven architectures.

OpenAPI Specification

The OpenAPI Specification (OAS) [15], formerly known as Swagger, is probably the most heavily adopted approach for the description of RESTful services (Section 2.6). OpenAPI suggests a description format for REST APIs. It is an open-source, language-agnostic specification, through which a consumer can understand and use a service with minimum implementation logic. Service descriptions are offered in either JSON or YAML[23] format, which can be produced and served statically, or be generated dynamically from the application. This allows the design and implementation of APIs to follow either a top-down (i.e. the service description is initially created and then the service is implemented) or bottom-up approach (i.e. the service description is generated from the service implementation). A comprehensive analysis of the specification is presented in Chapter 3.

OpenAPI is a simple, yet complete and powerful framework, supported by a large set of tools for designing, building and documenting RESTful services. The Swagger Editor[24] is an open-source Web-based editor for designing, defining and documenting RESTful services (Figure 2.3). It provides instant visualization and interaction with the API while still defining it. The Swagger Codegen[25] is an open-source code generator to build server code and client SDKs directly from an OpenAPI service description in almost any programming language and framework (PHP, Java, NodeJS). Swagger UI[26] is an open-source HTML5-based user interface to visually render documentation for an OpenAPI service description (Figure 2.4).

[23] https://yaml.org/
[24] https://github.com/swagger-api/swagger-editor
[25] https://github.com/swagger-api/swagger-codegen
[26]



```yaml
openapi: 3.0.3
info:
  title: Web Thing Model API for DHT22 sensor
  description: |
    This is a Web Thing Model server.  You can find
    out more about Web Thing Model (W3C) at
    [https://www.w3.org/Submission/wot-model/](https://www.w3.org/Submission/wot-model/).
  contact:
    email: atzavaras@intelligence.tuc.gr
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
  version: 1.0.0
externalDocs:
  description: Find out more about Web Thing Model
  url: 'https://www.w3.org/Submission/wot-model/'
servers:
  - url: 'http://localhost:5000/DHT22'
    description: SwaggerHub API Auto Mocking
tags:
  - name: Web Thing
    x-onResource: '''#/components/schemas/Webthing'''
    description: Operations on a Web Thing
    externalDocs:
      description: Find out more
      url: 'https://www.w3.org/Submission/wot-model/#web-thing-resource'
  - name: Properties
    description: Operations on Thing Properties
    externalDocs:
      description: Find out more about Thing properties
      url: 'https://www.w3.org/Submission/wot-model/#properties-resource'
  - name: Subscriptions
    x-onResource: '''#/components/schemas/SubscriptionObject'''
    description: Operations on subscriptions
    externalDocs:
      description: Find out more about Thing subscriptions
      url: 'https://www.w3.org/Submission/wot-model/#subscriptions-resource'
paths:
  /:
    get:
      tags:
        - Web Thing
      summary: Retrieve Web Thing
      description: 'In response to an HTTP GET request on the root URL of a Thing, an
        Extended Web Thing must return an object that holds its representation.'
      operationId: retrieveWebThing
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
```

Web Thing Model API for DHT22 sensor 1.0.0 OAS3

This is a Web Thing Model server. You can find out more about Web Thing Model (W3C) at https://www.w3.org/Submission/wot-model/.

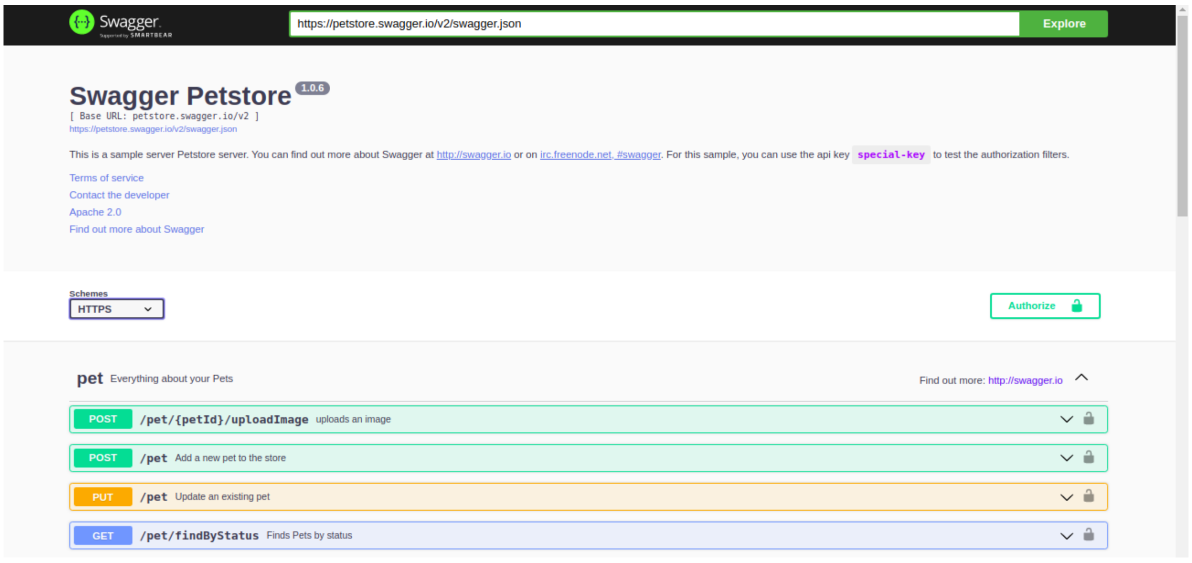
Contact the developer
Apache 2.0
Find out more about Web Thing Model

Servers
http://localhost:5000/DHT22 - SwaggerHub API Auto Mocking

Web Thing Operations on a W

Find out more: https://www.w3.org/Submission/wot-model/#web-thing-resource

Figure 2.3: Swagger Editor preview



https://petstore.swagger.io/v2/swagger.json

Swagger Petstore 1.0.6
[Base URL: petstore.swagger.io/v2]
https://petstore.swagger.io/v2/swagger.json

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service
Contact the developer
Apache 2.0
Find out more about Swagger

Schemes
HTTPS

pet Everything about your Pets — Find out more: http://swagger.io

POST /pet/{petId}/uploadImage uploads an image
POST /pet Add a new pet to the store
PUT /pet Update an existing pet
GET /pet/findByStatus Finds Pets by status

Figure 2.4: Swagger UI preview

OpenAPI is a widely adopted industry standard. It is endorsed by Linux Foundation and supported by large software vendors like Google, Microsoft, IBM, Oracle and many others. OpenAPI format is based on JSON (or YAML) and comprises a large set of properties for composing service descriptions. OpenAPI 3.0 is the first major update of the specification released in 2017. Version 3.1 (as of February 2021) provides full JSON Schema support (i.e. all keywords of JSON Schema vocabulary can be used in OpenAPI 3.1) while being fully compatible with version 3.0. OpenAPI can be enriched with text descriptions so that users can easily discover and understand the service and interact with it.

RAML

The RESTful API Modeling Language (RAML) [35] is a simple yet powerful language for describing "practically"-RESTful APIs. It is based on YAML format and it is capable of supporting RESTful services (Section 2.6). However, it can also define services that don't strictly follow the principles of REST (e.g. SOAP-based services). Contrary to OpenAPI Specification, RAML is only a top-down specification, meaning that the API is firstly designed and then the rest of the system is implemented. RAML is also supported by a number of native and third-party tools that facilitate the management of the whole API lifecycle from design to sharing. API Designer[27] is a web-based API development tool that allows API providers to design their API quickly, efficiently, and consistently and also share the design. It consists of a RAML editor side-by-side with an embedded RAML console (API Console). The API Console[28] provides live interactive documentation that lets users try out an API in real time. Unlike OpenAPI Specification, client code generation from RAML API documents is mainly provided by third-party commercial tools.

API Blueprint

The API Blueprint [36] suggests a different approach compared to OpenAPI Specification and RAML. It is a documentation-oriented description language based on a set of semantic assumptions laid on top of the Markdown syntax [37]. Unlike OpenAPI Specification and RAML, API Blueprint doesn't impose a specific style for the description of a service. A service provider is free to describe the functionality of his service in any way he prefers. For example, a service may be described by only providing examples for the various request and response messages without including any data type definitions (XML Schema, JSON Schema) that specify the structure of request and response messages. Due to the nature of the API Blueprint specification, there is limited tool support. The most common tool around API Blueprint is the Apiary platform[29]. It provides a collaborative design editor, interactive documentation, and other tools to improve user experience and interaction with a described Web API. The main drawback of API Blueprint is that it lacks tools for code generation.

AsyncAPI

Event-driven architectures are powerful and they can provide distributed, loosely-coupled and also fault-tolerant services that cannot be described by the existing service description approaches. AsyncAPI specification[30] is a relatively new, open-source approach that intends to solve the gap in the description of event-driven services. As

[27] https://www.mulesoft.com/platform/api/anypoint-designer
[28] https://github.com/mulesoft/api-console
[29] http://apiary.io/
[30] https://www.asyncapi.com/docs/specifications/v2.0.0

highlighted by its authors[31], AsyncAPI started as an adaptation of the OpenAPI Specification. As a result, they intended to make it as compatible as possible to help users that may use both frameworks. Although OpenAPI 3 enables the definition of service callbacks, it does not actually address several asynchronous communication use cases. For example, using asynchronous communication, if a device is currently unavailable (i.e. inaccessible), data can be stored and sent when the device becomes accessible again. Therefore, AsyncAPI is essentially an alternative approach to OpenAPI and RAML, but, to the best of our knowledge, it has not received considerable attention compared to OpenAPI. Likewise OpenAPI, AsyncAPI utilizes JSON and YAML format for the description of services. It is also an agnostic-protocol framework; there are AsyncAPI protocol bindings for common protocols such as MQTT, CoAP, Apache Kafka and Websockets. OpenAPI depends on REST and HTTP. This is a major advantage of the Async approach. In fact, AsyncAPI is entirely based on message-centric API interaction, which is actually found in IoT and other platforms alike. Therefore, AsyncAPI allows the description of various devices that use IoT data protocols (e.g. MQTT) and of their functionality. Moreover, AsyncAPI provides documentation tools and also code generation tools.

2.10 Semantic OpenAPI

WSDL and WADL could not be satisfying for the description of Cloud services. Despite being a W3C recommendation, WSDL has not been adopted widely by developers; they considered it complex and with not enough tooling support. Moreover, WSDL is preferred for the description of SOAP-based services, thus not leveraging the interoperability of the REST architectural style. WADL, on the other hand, was meant to enable the description of RESTful services. However, the approach was not adopted widely by developers either.

In this context, the OpenAPI framework, which is an industry standard, could be a very interesting and powerful solution for the description of RESTful services. However, Semantic OpenAPI [17] analysed the reasons that cause ambiguities in OpenAPI service descriptions, taking into consideration version 3.0 of the specification. For example, the same OpenAPI property may appear with different names within the same service document or, its meaning may not be defined at all in service definition. The authors suggest that OpenAPI properties should be semantically enriched, "*by associating OpenAPI entities to entities of a domain ontology*". In other words, they showed that, in order to eliminate ambiguities, each ambiguous property must be semantically annotated and mapped to a semantic model (e.g. a semantic vocabulary or an ontology).

Semantic OpenAPI introduces some extra properties (i.e. extension properties) to annotate existing OpenAPI properties. Therefore, the meaning of OpenAPI entities (e.g. an operation or a schema) can be defined and thus not be vague. In addition, [17] suggests that it is plausible to transform semantically annotated OpenAPI descriptions to ontologies. This allows the application of query languages (e.g. SPARQL) for service

[31] https://www.asyncapi.com/docs/getting-started/coming-from-openapi

discovery, and reasoning tools for detecting inconsistencies in service descriptions. The ontology proposed in [17, 18] incorporates features of Hydra to model service operations along with models not foreseen in Hydra (e.g. security features, header, constraints). Classes along with constraints on class properties are described using SHACL [19], which describes OpenAPI objects and validates Schema descriptions against the ontology.

Figure 2.5 illustrates the structure of an OpenAPI service document. It comprises many parts (objects). Each object specifies a list of properties that can be objects as well. Objects and properties defined under the Components unit of an OpenAPI document can be reused by other objects or they can be linked to each other (e.g. using keyword *$ref*). However, these links are not always explicitly expressed. For example, there can be properties with the same name, but with no reference to one another or an external model. The Info object provides non-functional information such as the names of the service and the service provider, license information and terms of the service. The Server object provides information about where the API servers are located. Servers can be defined for different operations (locally declared servers override global servers). The service description contains an Info object with some non-functional information for the service, an External Documentation object and Tag objects, which are used to group operations by resources or any other qualifier. For instance, in our work, a Web Thing tag, a Properties tag, an Actions tag and a Subscriptions tag are used to group properties by type or resource.

The description includes a Paths object that holds all the available service paths (i.e. endpoints) and their operations, which may also specify Parameter objects. The Paths object provides information about expressing HTTP requests to the service and about the responses of the service. It describes the supported HTTP methods (e.g. GET, PUT, POST, etc.) and defines the relative paths of the service endpoints (which are appended to a server URL to construct the full URLs of the operations). The Responses object describes the responses of an operation, its message content and the HTTP headers that a response may contain. The Parameters object describes parameters that operations use (i.e. path, query, header and cookie parameters). The Components object lists reusable objects. That includes (among others) definitions of schemas, responses, headers, parameters and security schemes. The Security object lists the security schemes of the service. The specification supports HTTP authentication, API keys, OAuth2 common flows or grants (i.e. ways of retrieving an access token) and OpenID Connect.

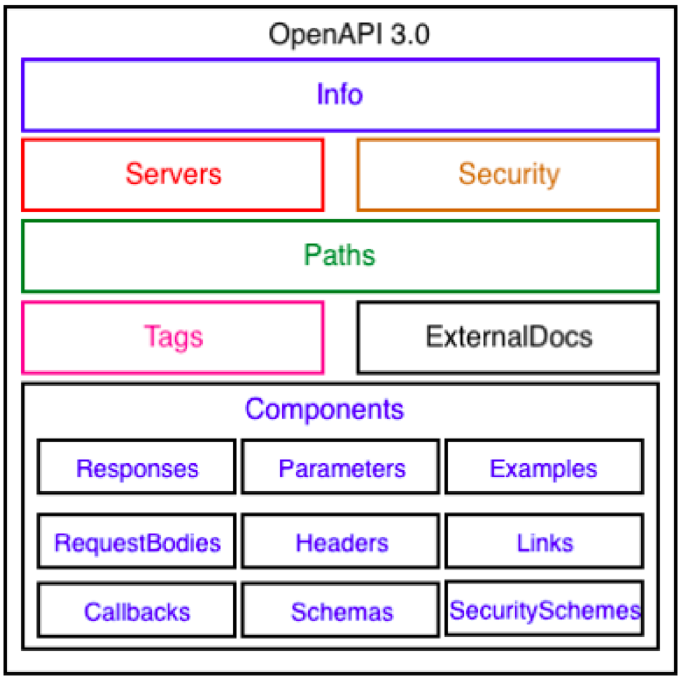


Figure 2.5: OpenAPI document structure

The Schemas object describes the request and response messages based on JSON Schema[32]. A Schema object can be a primitive (string, integer), an array or a model or an XML data type and may also have properties of its own accord (i.e. externalDocs). New data types can be defined as a composition or specialization of existing ones using properties allOf, oneOf, anyOf and not. Schema properties do not have semantic meaning and, consequently, their meaning can be vague. In addition, there can be Schema properties with different names that share the same meaning. A human might easily resolve ambiguities either by the element names or by the description that may be provided but a machine cannot. The problem is solved by associating each Schema object with a semantic model [17]. OpenAPI properties are semantically annotated and associated with entities of a semantic model using the *x-refersTo* extension property. The *x-kindOf* extension property defines a specialization between an OpenAPI property and a semantic model (e.g. a class). The *x-mapsTo* extension property denotes that a Schema property is semantically equivalent to another property in the same document. Additional extension properties are defined to clarify the meaning of the members in a collection of objects (*x-collectionOn*), for grouping Schema objects by type (*x-onResource*) and clarifying the meaning of operations (*x-operationType*). Table 2.2 illustrates all the extension properties for semantic annotations proposed in Semantic OpenAPI.

[32] https://json-schema.org/

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

Table 2.2: OpenAPI extension properties for semantic annotations

2.11 Ontologies and vocabularies

Service description languages normally offer only syntactic descriptions of service APIs. However, such descriptions are insufficient to enable the automation of tasks such as service discovery and composition. Earlier as well as more recent research suggests describing services also semantically in order to solve this problem. In the following, we will discuss the most remarkable approaches that suggest ontologies and vocabularies intending to define Web services as well as WoT concepts and, more specifically, interactions of clients with Things. We will focus on ontology approaches that intend to describe terms related to WoT.

2.11.1 Semantic Sensor Network Ontology (SSN)

The Semantic Sensor Network Ontology (SSN)[33] is defined as *"an ontology for describing sensors and their observations, the involved procedures, the studied features of interest, the samples used to do so, and the observed properties, as well as actuators"*. The term "actuator" refers to any device component that is able to cause any actuation (i.e. motion). The term "feature of interest" refers to the thing that is examined (e.g. measured, calculated, sampled, etc) in a specific use case (e.g. the tree is the feature of interest when measuring the height of a tree). SSN comprises a lightweight ontology called SOSA (Sensor, Observation, Sample, and Actuator) [38] with basic classes (e.g. *Observation*, *Sensor*) and properties (e.g. *observes*). W3C actually refers to SOSA as a "lightweight but self-contained" core ontology of SSN. The two ontologies together are capable of describing different types of use cases and applications. For instance, they may be used to describe WoT concepts, industrial infrastructures or even satellite imagery. The main advantage of SSN ontology over SOSA is that it has added expressiveness to SOSA and thus it can describe a larger number of concepts more precisely. As far as WoT is concerned, SSN and SOSA can be proved very useful for the description of Things (e.g. sensors and actuators) and their related concepts (e.g. sensor measurements), thus enabling the discovery of Things and of their functionality.

Figure 2.6 illustrates a conceptual view of the SSN and the SOSA ontology that originates from the SSN ontology specification[34]. It presents the basic modules of the two ontologies and shows that SOSA ontology can act as a lightweight core for SSN, as highlighted in the specification.

[33] https://www.w3.org/TR/vocab-ssn/
[34] https://www.w3.org/TR/vocab-ssn/#intro

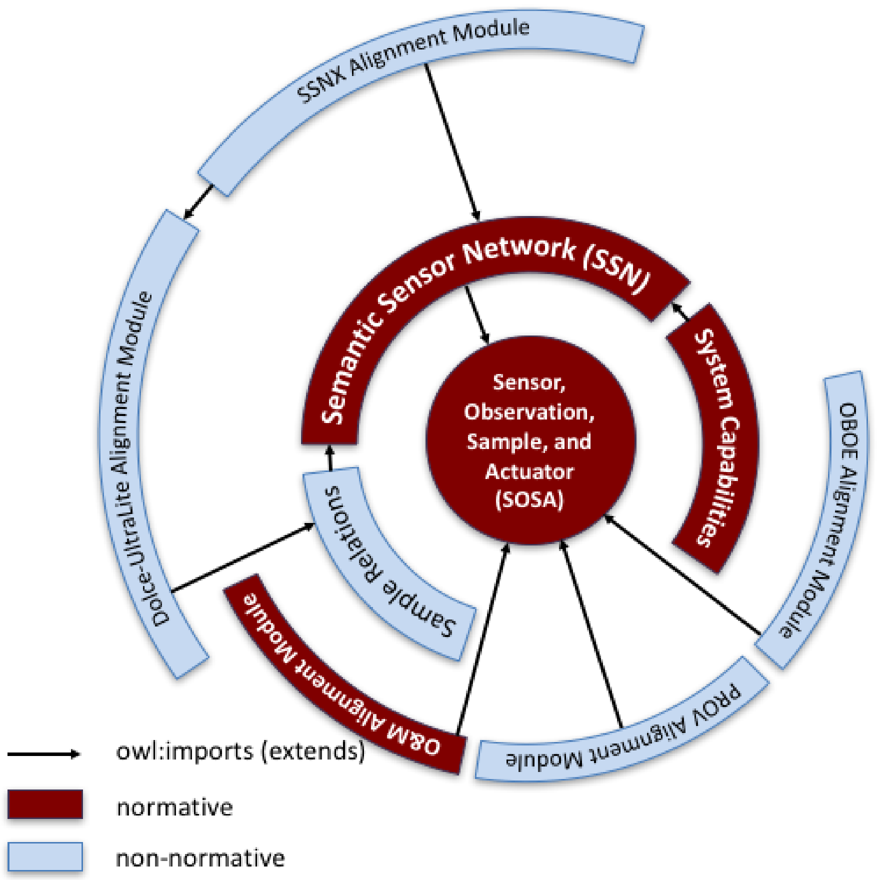


Figure 2.6: SOSA and SSN ontology modules

2.11.2 Thing Description (TD) Ontology

Thing Description (TD) Ontology[35] (W3C draft) is an ontology used to axiomatize the TD information model. According to its specification, the TD ontology is an alternative to the JSON representation format of TDs, but it can also be used to describe information related to Things (e.g. information in TDs) and also to provide alignments with other ontologies related to WoT (e.g. SOSA ontology). The ontology defines an *ActionAffordance* class, an *EventAffordance* class, an *InteractionAffordance* class, an *OperationType* class (i.e. it lists well-known operation types which are necessary in order to implement the WoT interaction model), a *PropertyAffordance* class and a *Thing* class. It also includes a number of object properties, datatype properties and named individuals. Moreover, the ontology imports another ontology named Hypermedia Controls Ontology[36] (Section 2.11.5) which is used to describe links and forms (i.e. referred to as hypermedia controls) used on the Web. The ontology describes Things and their interaction affordances such as properties, actions and events. For instance, a JSON-LD

[35] https://www.w3.org/2019/wot/td
[36] https://www.w3.org/2019/wot/hypermedia

TD document can benefit from the ontology by using the context attribute (i.e. *@context*). This attribute actually extends the definition of the Thing with additional ontology or vocabulary terms. For that purpose, the TD of a particular device needs to include the IRI namespace of the TD ontology (i.e. *https://www.w3.org/2019/wot/td*), as also illustrated in Listing 2.1. The document also presents alignments of the ontology with other WoT-related vocabularies (i.e. *SOSA*, *schema.org*) and some usage examples of the ontology. For instance, it demonstrates how a specific TD instance sample (i.e. JSON-LD format) could be represented by the TD ontology in the form of RDF triples[37]. The *Actions* object (i.e. field *actions*) of the TD, for example, can be represented by the *hasActionAffordance* object property of the TD ontology (i.e. using the IRI *https://www.w3.org/2019/wot/td#hasActionAffordance*).

Figure 1[38] in the WoT Thing Description specification illustrates a UML diagram that represents the TD core vocabulary of the TD Information Model (i.e. TD ontology). The figure is automatically generated from the ontology definition, as highlighted in the specification.

2.11.3 Web of Things (WoT) Security Ontology

Web of Things (WoT) Security Ontology is a W3C Working Draft and it is intended to define API security mechanisms. For instance, it includes a *SecurityScheme* class that defines security schemes in general, an *APIKeySecurityScheme* class for API key authentication, a *BasicSecurityScheme* class for Basic Authentication and an *OAuth2SecurityScheme* class for OAuth2.0 protocol, among others. The ontology also includes a number of object properties and datatype properties. In addition, the document presents a usage example of the ontology. Therefore, this ontology is capable of describing the API security mechanisms (i.e. used for Things and their operations) included in Thing Descriptions proposed by W3C. The *JSON-LD Context Usage* section[39] of the Thing Description W3C Draft indicates that the "*basic*" JSON key of TDs is mapped to the IRI that represents the *BasicSecurityScheme* class of the WoT Security ontology (i.e. *https://www.w3.org/2019/wot/security#BasicSecurityScheme*) in a specific JSON-LD file (among other mappings). This file is provided as a resource in the namespace identified by the IRI *https://www.w3.org/ns/td*. Consequently, the term "*basic*" can be used directly in TDs (i.e. instead of the mapped IRI) in order to declare how clients can perform a specific operation (i.e. using Basic Authentication). Based on the Thing Description draft specification, "*all vocabulary terms referenced in TD Information Model are serialized as (compact) JSON strings in a TD document*".

Figure 3[40] in the WoT Thing Description specification illustrates a UML diagram that represents the WoT Security vocabulary of the TD Information Model (i.e. WoT Security ontology). The figure is automatically generated from the ontology definition, as highlighted in the specification.

[37] https://www.w3.org/2019/wot/td#thing-description-json-ld-1-1-instance-to-rdf-dataset
[38] https://www.w3.org/TR/wot-thing-description11/#overview
[39] https://www.w3.org/TR/wot-thing-description11/#json-ld-ctx-usage
[40] https://www.w3.org/TR/wot-thing-description11/#overview

2.11.4 Data Schema Vocabulary

Data Schema Vocabulary (or JSON Schema in RDF) is a W3C Working Draft[41] that presents an RDF vocabulary for JSON schema definitions. It actually builds on the main terms defined by JSON schema[42] (i.e. a vocabulary for the annotation and also the validation of JSON documents) to represent schema definitions in RDF format. It provides namespace IRIs for JSON schema keywords and simple axioms. According to the vocabulary specification, various examples[43] have been introduced to show how the vocabulary can be used (e.g. for the annotation of schemas with JSON-LD metadata or for embedding schema annotations inside RDF graphs).

More specifically, the vocabulary includes an *ArraySchema*, a *BooleanSchema*, a *DataSchema*, an *IntegerSchema*, a *NullSchema*, a *NumberSchema*, an *ObjectSchema* and a *StringSchema* class. For example, the *ArraySchema* class defines the metadata describing data of type *array* and the *NumberSchema* class defines the metadata describing data of type *number*. This is indicated by the value *"array"* or the value *"number"* assigned to the *type* attribute in *DataSchema* instances in TDs. The vocabulary also includes a number of object properties (e.g. *allOf*, *anyOf*) and datatype properties (e.g. *contentEncoding*, *contentMediaType*) for the description of data schemas. Therefore, this vocabulary is meant to describe the data schemas that represent Things and their interactions (e.g. properties, actions, events).

2.11.5 Hypermedia Controls Ontology

Hypermedia Controls Ontology is also a W3C Working Draft[44]. It is defined as *"an ontology for Web links and forms, the main hypermedia controls in use on the Web"*. Regarding forms, the document states that they are request templates which can be exposed by servers to clients, so that clients fill them in with client-specific information and send them back to servers. Moreover, according to the specification, "*forms are similar in spirit to operation descriptions as defined by the Open API Specification [openapis][45] or by the Hydra RDF vocabulary [hydra][46]*". Thus, Web forms proposed by W3C are similar but not the same with the operation descriptions of the OpenAPI Specification. The ontology includes an *ExpectedResponse* class that defines the primary response of a service request, an *AdditionalExpectedResponse* class that defines the additional response of a service request, a *Form* class for Web forms and a *Link class* for Web links. The ontology also includes a number of object properties (e.g. the *hasTarget* object property which

[41] https://www.w3.org/2019/wot/json-schema
[42] https://json-schema.org/specification.html
[43] https://www.w3.org/2019/wot/json-schema#usage-examples
[44] https://www.w3.org/2019/wot/hypermedia
[45] https://www.w3.org/2019/wot/hypermedia#bib-openapis
[46] https://www.w3.org/2019/wot/hypermedia#bib-hydra

refers to the URI, protocol and method used to perform an operation) and datatype properties (e.g. *forSubProtocol* that refers to the exact mechanism of a protocol used to receive asynchronous event notifications from a Thing).

More specifically, the ontology describes the concepts of Web links and Web forms defined in the WoT Architecture. In addition, the specification presents some alignments of the ontology with the Hydra RDF vocabulary[47]. According to the specification, there is alignment between links in the Hypermedia Controls Ontology (*htcl:Link*) and links in Hydra (*hydra:Link*), which is a vocabulary intended to describe hypermedia-driven Web APIs (Section 2.11.5). In addition, the specification indicates that there is alignment between forms in the Hypermedia Controls Ontology (*htcl:Form*) and operations in Hydra (*hydra:Operation*). However, the specification of the ontology notes that there is "*a close match*" between the corresponding classes (i.e. Hypermedia Controls Ontology *Link* class with Hydra *Link* class, Hypermedia Controls Ontology *Form* class with Hydra *Operation* class) and that they are not equivalent. Hydra classes are described in Section 2.11.5 and they are illustrated in Figure 2.7. It highlights that the alignments should not be "*understood as formal semantic equivalences but rather as hints to Hydra users*".

Therefore, this ontology is capable of describing the hypermedia controls included in the TD documents of W3C. Properties of TDs can be mapped to terms (e.g. object properties) of the Hypermedia Controls ontology using the ontology's namespace IRI (i.e *https://www.w3.org/2019/wot/hypermedia*), as noted in the W3C TD specification[48]. The *JSON-LD Context Usage* section[49] of the Thing Description Draft specification indicates that the "*href*" JSON key of TDs, for example, is mapped to *https://www.w3.org/2019/wot/hypermedia#hasTarget* (i.e. the *hasTarget* object property of the ontology) in a specific JSON-LD file. The file is provided as a resource in the namespace identified by the IRI *https://www.w3.org/ns/td*. Consequently, the "*href*" can be used directly in TDs (i.e. instead of the mapped IRI) in order to declare how clients can perform a specific operation.

Furthermore, the ontology specification presents some usage examples of the ontology. For instance, it demonstrates a JSON-LD context that could be used in a TD to describe hypermedia controls[50] using the ontology. This example includes terms that may map TD properties to terms of the ontology, such as object properties (e.g. *hasRelationType*[51]) and data properties (e.g. *forSubProtocol*[52]). The *forSubProtocol* data property, for example, can describe the exact mechanism that should be used for a given protocol (i.e. when there are multiple mechanism options) to succeed a particular interaction with a Thing (e.g. long polling subprotocol to subscribe to the overheating event of a temperature sensor using the HTTP protocol).

Figure 4[53] in the WoT Thing Description specification illustrates a UML diagram that represents the Hypermedia controls vocabulary of the TD Information Model (i.e.

[47] https://www.w3.org/2019/wot/hypermedia#alignments
[48] https://www.w3.org/TR/wot-thing-description11/#namespaces
[49] https://www.w3.org/TR/wot-thing-description11/#json-ld-ctx-usage
[50] https://www.w3.org/2019/wot/hypermedia#example-json-ld-context-for-hypermedia-controls
[51] https://www.w3.org/2019/wot/hypermedia#hasRelationType
[52] https://www.w3.org/2019/wot/hypermedia#forSubProtocol
[53] https://www.w3.org/TR/wot-thing-description11/#overview

Hypermedia Controls Ontology). The figure is automatically generated from the ontology definition, as highlighted in the specification.

2.11.6 Hydra Core Vocabulary

JSON-LD provides a generic serialization format and also a shared vocabulary - understood by both the server exposing the API and the client consuming it. Hydra provides a minimal vocabulary for the description of hypermedia-driven Web APIs. This vocabulary defines a number of concepts (e.g. hypermedia controls) in RDF Schema that allow machines to understand how to interact with a Web API. The main essence of Hydra is to provide servers with a vocabulary that will allow the advertisement of valid state transitions to clients. That is, response messages from the server should contain enough information that a client can use in order to discover all the available resources and actions that are actually needed, so as to construct new HTTP requests to achieve a specific goal. Therefore, Hydra allows the creation of smarter clients.

Figure 2.7 illustrates a conceptual view of the Hydra Core Vocabulary. The primary class of the vocabulary is *ApiDocumentation*. According to the specification, it allows the server to define the main entry point (*EntryPoint*) and document all the operations (*Operation*) as well as the entities (*Class*) and their properties (*Property*) it supports.

The *Resource* class informs the client that a resource is dereferenceable. This means that, as long as an IRI is accessed, the representation of a resource can be retrieved. Finally, according to the specification of Hydra, the *Link* class defines properties that represent dereferenceable links.

There are cases that the interaction with a service requires links that cannot be created by a server. For example, in order to query a service, a link may contain parameters that must be filled by a client at runtime. In Hydra, such cases are described by the *IriTemplate* class. An *IriTemplate* consists of a template that describes an IRI template[54] and a number of *mappings*. An IriTemplateMapping maps a variable in the IRI template to a property. Listing 2.2 demonstrates an example of an IriTemplate description, where the variable "*lastname*" maps to the property "*givenName*" from *Schema.org* vocabulary. With this information, a client

[54] https://tools.ietf.org/html/rfc6570

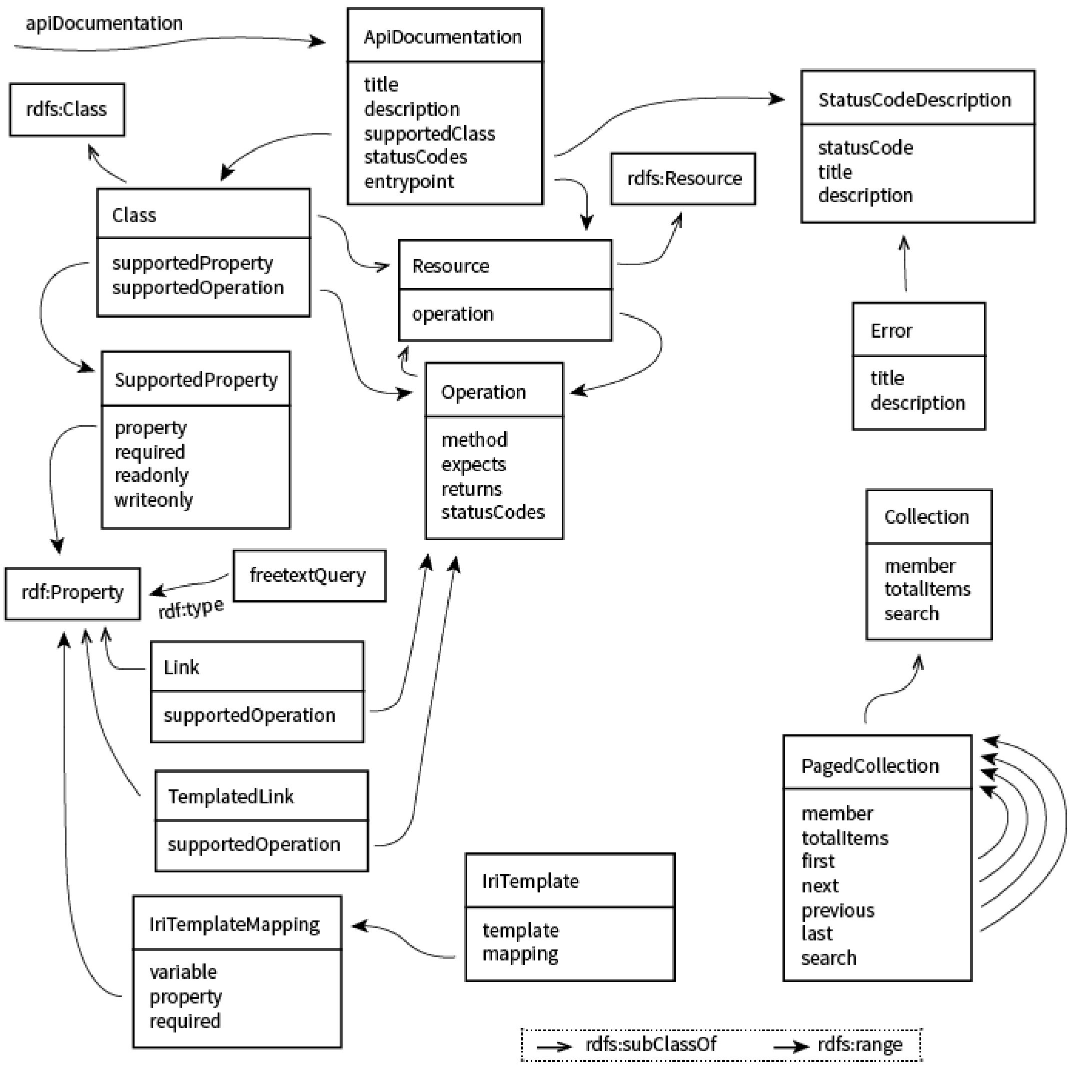


Figure 2.7: The Hydra Core Vocabulary

may understand the meaning of variables and generate a complete IRI.

Listing 2.2: Description of an IriTemplate in Hydra

```
{
    "@context" : "http://www.w3.org/ns/hydra/context.jsonld",
    "@type" : "IriTemplate",
    "template" : "http://api.example.com/users{?lastname}",
    "mapping" : [
        {
            "@type" : "IriTemplateMapping",
            "variable" : "lastname",
            "property" : "schema.org/givenName",
            "required" : true
        }
    ]
}
```

The *Operation* class represents the information that is necessary so that a client may send valid HTTP requests to the server. The *method* property is used to specify the HTTP method, while the *expects* and *returns* properties define the expected data in request and response messages. In addition, the statusCode property specifies a StatusCodeDescription that provides a developer with information regarding what to expect when invoking an operation.

The *Class* class is an interesting feature of the Hydra vocabulary. It extends a class definition by providing the *supportedProperties* that belong to the class. This is important as in RDF there is not any mechanism informing which properties belong to a class and also enabling properties from other vocabularies to be reused directly. A *SupportedProperty* defines the property that is used and specifies whether it is required, readonly or writeonly.

In a Hydra-driven Web API, the service description may be discovered automatically by a client if the API provider marks its responses with an HTTP Link Header to direct a client to the corresponding API document. This allows the dynamic discovery of API descriptions at runtime. Moreover, due to the use of RDF's unique identifiers, parts of the API descriptions can be shared and reused improving interoperability of services.

To enable the creation of hypermedia-driven Web APIs, the Hydra core vocabulary is used along with JSON-LD, which manages to make data self-descriptive. JSON-LD [5] is a lightweight format used to represent Linked Data in JSON. The design of JSON-LD allows existing JSON to be interpreted as Linked Data with minimal changes. Furthermore, JSON-LD is 100% compatible with JSON. As a result, existing JSON parsers and libraries can be reused. In fact, JSON-LD provides hypermedia controls (i.e. links to other resources) and Hydra can describe them. Therefore, the combination of Hydra and JSON-LD enables the runtime discovery of resources. This allows the implementation of completely generic clients (e.g. API consoles or client libraries)[55].

Hydra is currently endorsed by W3C and a community group is working to extend Hydra and provide tools and guidelines for designing and creating Hydra-driven Web APIs. Therefore, Hydra is a promising effort that could contribute to the evolution of RESTful APIs. Nevertheless, it is under development and there is not an official W3C recommendation (yet). This is a discouraging factor for considering Hydra the most suitable approach for the description of Cloud services. We should note, however, that Hydra can be very useful for modeling service operations; once a recommended specification comes out, we expect Hydra to be much more popular.

2.11.7 Schema.org Vocabulary

Schema.org is a semantic, shared vocabulary for structured data on the Web; in fact, it is a collection of vocabularies. It includes a number of concepts such as person, organization, product, place, action, movie and book. It can be used with different

[55] http://www.hydra-cg.com/#tooling

encodings such as JSON-LD, microdata[56] and RDFa[57]. Schema.org is founded by Google, Microsoft, Yahoo and Yandex. It launched in 2011 and it is constantly updated since then. Schema.org can be referred to using semantic annotations (e.g. JSON-LD *@context* in a TD or *x-properties* in Semantic OpenAPI) and define concepts for an application domain. For instance, the Web Thing Model specification [23] includes an indicative example that shows the usage of the *schema.org* detailed vocabulary (see Section 2.4).

2.11.8 OpenAPI Ontology

The OpenAPI Ontology (or OpenAPI 3.0 ontology) [17, 18, 39], which is illustrated in Figure 2.8, captures all information of an OpenAPI description of [18]. Hydra core vocabulary is at the heart of the ontology. More specifically, the OpenAPI ontology utilizes Hydra in order to model service operations, and also SHACL [19] in order to describe OpenAPI objects. Therefore, the OpenAPI ontology supports the efficient representation and dynamic discovery of hypermedia-driven APIs on the Web. A specific algorithm, which is used to map service descriptions to the OpenAPI ontology, is available as a Web Application[58] for testing. In fact, the idea of using ontologies is not new and existing ontologies fit well the needs of remote procedure call technologies such as SOAP [45]. However, the emergence of REST, generated new difficulties in the representation of hypermedia-driven APIs (such as REST) that call for the dynamic discovery of resources at run-time (referred to as HATEOAS). This feature is not supported by known service ontologies (such as OWL-S [12] for SOAP services). In the following, we will discuss the OpenAPI ontology briefly, as it is presented in detail in [39].

[56] https://en.wikipedia.org/wiki/Microdata_(HTML)
[57] https://rdfa.info/
[58] http://www.intelligence.tuc.gr/semantic-open-api/

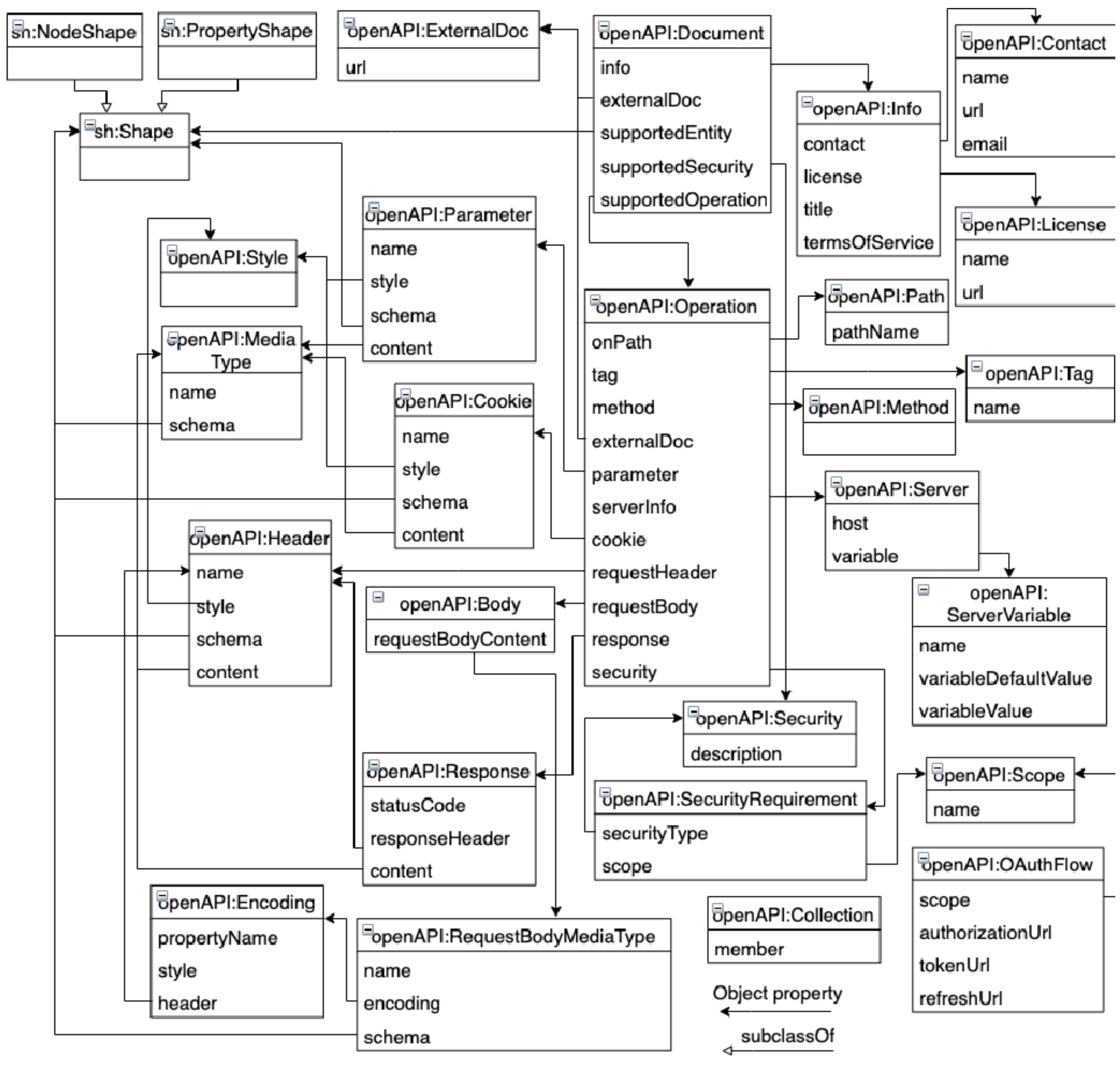


Figure 2.8: OpenAPI 3.0 ontology

The *Document* class of the ontology provides general information about the service (i.e. in the Info class) and specifies the service paths, the entities and the security schemes it supports. The *Path* class represents (relative) service paths (in *pathName* property). The *Operation* class provides information for issuing HTTP requests. Request bodies are represented by the *Body* class, while responses are declared in the *Response* class specifying the status code and the data returned. The entire range of HTTP responses is represented. The *MediaType* class describes the representation format (the most common being JSON and XML) of a request or response body data. The *Operation* class refers to a security scheme in the *SecurityRequirement* class.

Figure 2.9 illustrates the security schemes supported by OpenAPI. The *Security* class includes security schemes as sub-classes. The *OAuth2* class has different flows (grants) as sub-classes. If the security scheme is of type *OAuth2* or *OpenID Connect*, then scope names are defined as properties.

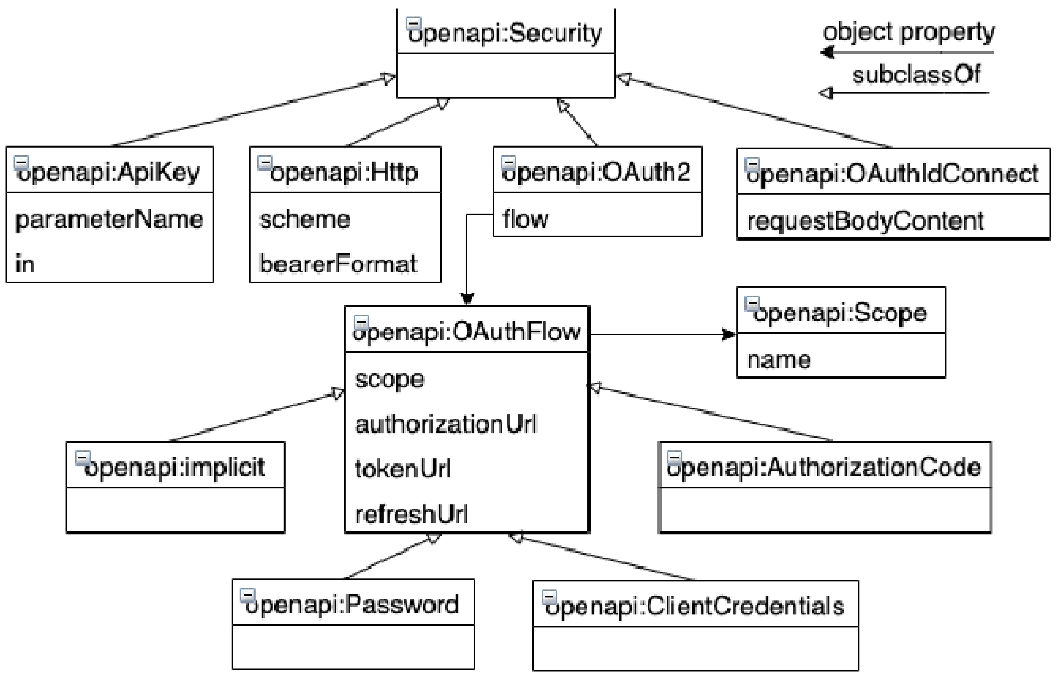


Figure 2.9: OpenAPI 3.0 security class

Schema objects are expressed as classes, object properties and data properties using SHACL, which is an RDF vocabulary that can be used in order to define classes along with constraints on their properties. It provides built-in types of constraints (e.g. cardinality: *minCount*, *maxCount*) and is expressed by the *Shape* class. That is, a Schema object is mapped to the *Shape* class of the ontology, which is distinguished into the *NodeShape* class and the *PropertyShape* class. The *NodeShape* class defines the properties of a class and specifies whether a class may contain additional properties (additionalProperties) of a specific type. Additionally, it represents operations related to a class (*supportedOperation*), which come from *x-onResource* extension property. Class *PropertyShape* represents the properties of a class, their data type and restrictions (e.g. a maximum value for a numeric property) and indicates whether the supported property is required or read-only. Table 2 in [39] shows how Schema object properties are mapped to properties of the SHACL vocabulary. Moreover, [39] demonstrates how the Person model (Listing 1.1 in [39]) is represented in the OpenAPI ontology. The model contains references to the *www.schema.org* vocabulary using the *x-refersTo* extension property.

The OpenAPI ontology also represents collections through the *Collection* class by specifying the members (*member*) of a collection. For example, Listing 1.6 in [39] demonstrates the *PersonCollection* of Listing 1.3 representation in the OpenAPI ontology. A *PersonCollection* class is defined in the OpenAPI ontology as a subclass of the Collection class. This example is further described in [39].

OpenAPI parameters are represented as separate classes for every parameter type in the OpenAPI ontology. The *Header* class contains all the definitions of header parameters that are used in HTTP requests and responses. The *Cookie* class defines the cookies that are sent through HTTP requests and responses. In addition, the *Parameter* class defines all parameters that are attached to the operation's URL. The class is further organized in the *PathParameter* and the *Query* classes that refer to the corresponding path and query parameters of the specification.

A request or response body (defined using the *content* property) is used to send and receive data via the REST API respectively (a response also includes a response code, e.g. 200, 400, etc.). Media type is a representation format of request or response body data in different formats - the most common are JSON, XML, text and images. They are typically defined in the *Paths* object; however, reusable bodies can also be defined in the *Components* object. Each media type includes a *Schema* property, defining the data type of the message body. Request and response bodies are represented as properties of the *Operation* class. In particular, request and response bodies are defined as classes and their media type is also defined in this way. The *Encoding* class defines keywords denoting serialization rules for media types with primitive properties (e.g. *contentType* for nested arrays or JSON).

3

Web of Things implementations

The Web Thing Model submission of W3C is not a recommendation; the WoT Architecture specification is. The Web Thing Model is an earlier work of W3C, but it laid the foundations for our implementation of a REST API that allows the interaction of clients with Things and our implementation of a Thing description approach for the WoT (i.e. an alternative approach to the TD of W3C).

In this section, we introduce and discuss Web Thing Model service (WTMs), our implementation of the Web Thing Model according to the requirements of the model. Existing Web of Things implementations are also reviewed and compared with WTMs considering the requirements of the Web Thing Model. The three candidate WoT implementations (namely, *Thingweb.node-wot*, *Webofthings.js* and *WTMs*) are compared based on their capacity to support the entire set of operations of the Web Thing Model specification. In fact, Thingweb.node-wot is the *only* implementation that adopts TDs and is fully compatible with the *W3C TD information model* proposed in the WoT Architecture. Webofthings.js and WTMs were based on the Web Thing Model REST API; although they follow the basic requirements of the WoT Architecture, they are more compliant with the OpenAPI Thing template approach described in Chapter 4. Moreover, in Chapter 6, we review and compare the implementations again based on the WoT Architecture of W3C which is a recommendation.

The comparison and the discussion followed, revealed that WTMs is the only implementation that fully complies with the Web Thing Model specification. It is realized as a RESTful Web service that communicates (i.e. with other services or clients) over HTTP in the cloud (and the Web) and implements the requirements, and all operations suggested by the Web Thing Model.

The performance of WTMs is evaluated in a smart city scenario [40]. The results of this evaluation reveal that WTMs is capable of responding in real-time under stress (i.e. with thousands of requests out of which many are executed in parallel). Finally, to convey Thing specific functionality and information (i.e. Thing identifier, descriptions and payloads) from the Web to an IoT application, WTMs is implemented as a proxy service of iSWoT [41], a prototype IoT Service Oriented Architecture (SOA) deployed on Google Cloud Platform (GCP). Cloud is the ideal environment for IoT applications deployment due to reasons related to its affordability (no up-front investment, low operation costs), scalability, easy maintenance and accessibility. Therefore, we show how WTMs can be integrated within a Service Oriented Architecture (SOA) for the IoT to support full-fledged WoT functionality.

3.1 Thingweb node-wot

Thingweb node-wot[59] is an implementation of the WoT Scripting API[60] on Node.js[61] and enables the implementation of Thing operations using a JavaScript API similar to the Web browser APIs. Thingweb is based on the W3C Architecture and it is fully compatible with the W3C TD information model, as it adopts TDs to represent Things and thus allow clients to interact with them. In other words, it provides an API Interface that allows scripts to interact with Things using Web protocols such as HTTP, HTTPS, CoAP, MQTT, Websockets. Not all WoT operations are supported by all protocols. Thingweb supports most of the operation types of the WoT Architecture listed in Table 2.1. These are discussed in Section 3.4, where the implementation is compared with WTMs.

The Thingweb node-wot repository in Github includes short code documentation[62] of the operations (i.e. not detailed API documentation) which are now supported in Thingweb. The repository also lists the protocol bindings that are currently supported by the implementation. Not all of these protocols are fully supported. At the time of writing, Thingweb provides only server-side support for the Websocket protocol and only client-side support for some industrial protocols (e.g. OPC-UA). In addition, Thingweb supports protocol binding for Cloud Firestore[63] (i.e. hosted in Google Cloud), which is a popular, flexible and also scalable NoSQL database (i.e. it stores documents) provided by Google. It is a database intended for mobile and web applications. Firestore is capable of subscribing to written data; the Thingweb Firestore Binding utilizes this feature to enable communication between Things and clients. As noted in the documentation of Thingweb, this binding offers a storage location (i.e. a Firestore DocumentReference[64]) in Firestore for data related to the properties, actions and events of a Thing. After data is written to a Firestore Reference, the communication of a Thing with Clients that subscribe to the Thing's data is enabled. Therefore, the exposed Thing Descriptions are stored in Firestore.

The list of protocol bindings currently supported by Thingweb is the following:

- HTTP

[59] https://github.com/eclipse/thingweb.node-wot
[60] https://www.w3.org/TR/wot-scripting-api/
[61] https://nodejs.org/en/
[62] https://github.com/eclipse/thingweb.node-wot/blob/master/API.md
[63] https://firebase.google.com/docs/firestore
[64] https://firebase.google.com/docs/reference/js/v8/firebase.firestore.DocumentReference

- HTTPS
- CoAP
- CoAPS
- MQTT
- Firestore
- Websocket (server-side only)
- OPC-UA (client-side only)
- NETCONF (client-side only)
- Modbus (client-side only)
- M-Bus (client-side)

Finally, according to Thingweb documentation in Github, more protocols can be simply added by implementing particular programming interfaces (i.e. specified in the documentation) for each of these protocols.

3.2 Webofthings.js

Webofthings.js[65] is a lightweight and extensible implementation of WoT operations in Node.js for HTTP and Websockets protocols. Similar to WTMs, it is a typical implementation of the WoT operations suggested in the original WoT resource [1]. Therefore, it is based on the Web Thing Model submission; it is fully compatible with the Web Thing Model proposed representation of Things and thus the OpenAPI Thing template approach. It is not compatible with the W3C TD information model, as it does not adopt W3C TDs for the representation and the interaction with Things. According to the creators of Webofthings.js, it is a “server and gateway reference implementation of the W3C Web Thing Model, written in Node.js and tailored for embedded systems”. They intend to provide an implementation that allows Thing to interact together and also with a Web application. However, according to the description of Webofthings.js in Github, it is just a way to experiment with WoT and the basis of the Building the Web of Things book [1]. It provides a WoT server that can run on any platform that supports Node.js and especially on several embedded devices. For example, it can be deployed on a Raspberry Pi and provide Web access to devices such as sensors and actuators.

3.3 Web Thing Model service (WTMs)

In the following, we describe a Web Thing Proxy that allows the interaction of clients (i.e. users or services) with Things on the Web and proposes a particular Web service (i.e. a REST API) that implements particular operations. Web Thing Model service (WTMs) was based on the Web Thing Model submission of W3C. Therefore, it does not adopt the WoT Thing Description (TD) format proposed by W3C which is, according to the WoT

[65] https://github.com/webofthings/webofthings.js/

Architecture, "*the central building block of W3C WoT*". However, the Web Thing Proxy is compatible with the WoT Architecture, as it follows its common principles such as flexibility (i.e. it supports a wide variety of physical device configurations), scalability (i.e. for IoT solutions with thousands to millions of devices) and also interoperability (i.e across device and cloud manufacturers). Moreover, WTMs enables the interaction of clients with Things using a RESTful API, which is one of the basic requirements of the WoT Architecture. The compliance of WTMs with the WoT Architecture is further discussed in Chapter 6.

3.3.1 Web Thing Proxy

In Section 2.1, we highlighted that the interconnection of Things to an application is commonly supported by device-specific protocols rather than by HTTP directly. Thus, we assume that Things connect to a protocol translation service whose role is to convey Thing-related data to the application using HTTP and JSON. This service runs on a gateway but it can be deployed within a proxy in the cloud (or server) equally well. IDAS backend IoT management[66] is a reference implementation of this service. It is the only service that is affected by the property of Things (e.g. a sensor) to use a specific protocol. Following IDAS, Things register to a Publication and Subscription service [42] to make their information available to users or other services based on subscriptions. Each time a Thing registers to Web Thing Proxy, a new entity is created in the Publication and Subscription service. Each time new data about a Thing becomes available, its corresponding component in the Publication and Subscription service is updated and a notification is sent to all subscribed entities (i.e. users or services). It works in conjunction with any Publication and Subscription service such as ORION Context Broker[67], Scorpio[68] or RabbitMQ[69]. The database of the service (i.e. a MongoDB) is used to store all data about Things (i.e. descriptions, properties, measurements, actions, action executions and subscriptions to Things). These data can be retrieved, updated or deleted by the Web Thing Model service. Notice that, Context Broker holds only the most recent information. History (past) data is forwarded to a history database (not part of Web Thing proxy). Fig. 3.1 illustrates the most essential parts (i.e. services) of the Web Thing proxy and its API interface.

66 https://fimac.m-iti.org/5a.php
67 https://fiware-orion.readthedocs.io/en/master/
68 https://github.com/ScorpioBroker/ScorpioBroker
69 https://www.rabbitmq.com/

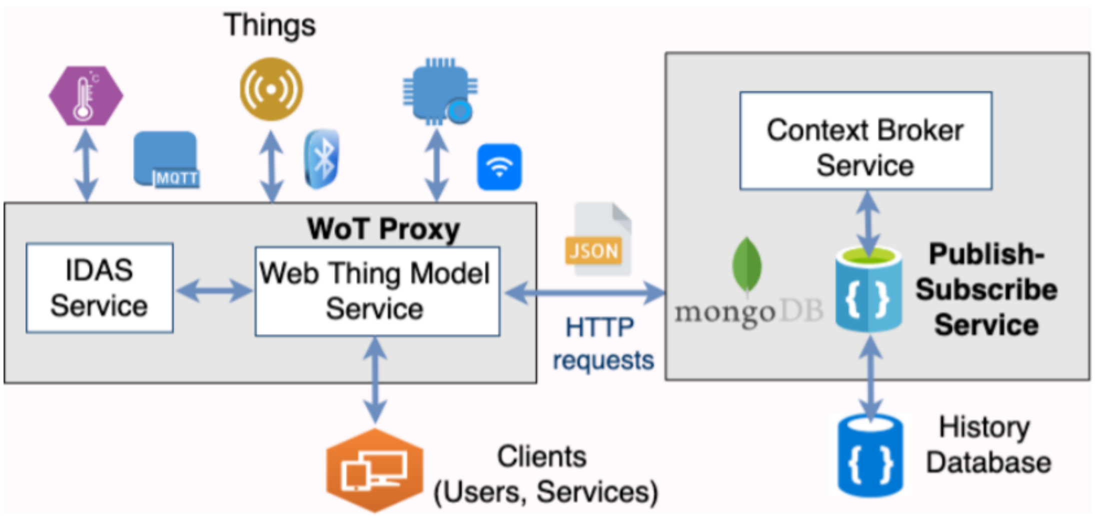


Figure 3.1: Web Thing Proxy service

According to the Web Thing Model specification, a Thing is identified by its resources, namely, a Web Thing Resource, a Model Resource, a Properties Resource, an Actions Resource (as long as the Thing supports actions), a Things Resource and a Subscriptions Resource. The Web Thing Resource is the root resource of the Web Thing model and it is used to provide a short and abstract description of a Web Thing. The Model Resource is meant to describe the values of Things properties as well as the actions that can be performed on Things. The Thing model is a more detailed description and it may also include helpful links (e.g. a link to a documentation file). The Properties Resource defines the properties of a Thing in general (e.g. pressure, humidity) and describes measurements related to a Thing property. (e.g. temperature values provided by a temperature sensor) or the internal state of a Thing (e.g. the state of a smart door). The Actions Resource defines the allowed actions on a Thing, such as execution commands (e.g. a command sent by a client to a window actuator to open). The Things Resource is different from the Web Thing resource and it is used to describe specific operations on Things such as the registration of new Things to an application. Finally, the Subscriptions Resource is used to describe subscriptions to Web Things (especially to their actions and properties). For example, users and services can subscribe to the humidity property of a specific sensor and get notified of any changes of this information (i.e. new humidity value).

JSON is the data exchange format for Web Things, as already highlighted in this work. The exact formatting, however, depends on the particular services of the proxy service (e.g. NGSI in the case of IDAS). The Web Thing Model introduces a list of operations that (in part or in full) can be offered by a Thing. Concerning Web Thing Resource, the model may allow an operation that retrieves or updates the description of a Thing in JSON. Similarly, for Thing Model (or for Properties Resource), the Web Thing model may allow operations that retrieve or update the properties or actions (their values or state information respectively) on a Thing. Concerning Actions Resource, the model may allow an operation that retrieves the actions that a Thing may perform and, in addition, an operation for sending a command to a Thing to execute an action and operations to retrieve past action executions. Concerning Things Resource, the model may allow an operation that registers a Thing or an operation that retrieves all registered Things. Finally, in relation to Subscriptions Resource, the model may allow an operation that creates a new subscription to a Web Thing resource or, an operation that retrieves or updates the information of an existing subscription.

3.3.2 WTMs implementation

Web Thing Model service (WTMs) is an autonomous RESTful service in Python Flask and implements some of the WoT Architecture operations on Things using HTTP. It does not adopt W3C TDs for the representation and the interaction with Things. In fact, WTMs is a Web service designed to support Thing operations and uses the same JSON payloads, API endpoints and response codes as described in the W3C Web Thing Model submission of W3C. Specifically, WTMs supports all operations for retrieving and updating Thing descriptions and their properties, as well as all Thing model operations. It implements functions that send a command to a Thing (i.e. an actuator) to execute or retrieve actions and action executions, as well as functions that create, retrieve and delete subscriptions on Web Thing resources. In the following discussion, a hypothetical Smart Door (i.e. an actuator device) is used in most examples. It provides information on its state (i.e. open, closed or locked) and can receive an action command to open, close, lock or unlock. The DHT22 sensor[70] is also mentioned in some examples. In the following, WTMs operations are grouped by resource. The majority of these operations (i.e. 14 out of 18) are used in the OpenAPI Thing template proposed in Section 3, so they have already been described to some extent.

A. Web Thing Resource

The first operation retrieves a Thing and is realized by issuing an HTTP GET request to the root URL of a Thing. The root URL of a Thing is its IP address and default port that follows the IP (e.g. *http://34.122.93.207:5001/MySmartDoor* in the case of the smart door). To retrieve the TD, an HTTP GET request is sent to the Context Broker service, where the TD is stored. The JSON representation of the Thing contains its identifier (e.g. MySmartDoor) and possibly a characteristic name, a description, the date when the Thing entity was created and any other information given by the Thing owner. A Thing can be registered by its owner, who is responsible for setting the Thing description. A 200 OK response code is also returned as long as the operation is successful, similar to any other retrieval operation of the model. The function of updating a Thing description requires sending an HTTP PUT request to the root URL of a Thing, containing a JSON object in the request body. This object may contain new values for any of the Thing's attributes (except the identifier which cannot be updated). The operation is realized by sending an HTTP PATCH request to the Context Broker to update the existing Thing

[70] https://www.adafruit.com/product/385

description by changing its attributes. A 204 NO CONTENT response code is returned if the operation is successful.

B. Model Resource

The first operation intends to retrieve the model of a Thing by issuing an HTTP GET request to the */model* endpoint of the root URL of a Thing. The request is forwarded to Context Broker service where the description of the Thing model is stored (i.e. using */v2/entities/MySmartDoor?type=Model* instead of retrieving the entity that stores the TD by using */v2/entities/MySmartDoor?type=Thing*). The JSON representation returned contains the identifier of the Thing along with any attribute which describes the model of the Thing (i.e. its properties, actions, etc.). Compared to TD, it is a more detailed description of the Thing.

The second operation intends to update the model of a Thing (i.e. update attributes of the Thing model description), by issuing an HTTP PUT request to the */model* endpoint of the root URL of a Thing. The new attribute values are included in the request body (except the identifier attribute that cannot be updated). The operation is realized by forwarding an HTTP PATCH request to the Context Broker service. A 204 NO CONTENT response is returned if the operation is successful.

C. Properties Resource

The first operation intends to retrieve a list of properties. It is realized by issuing a GET request to the */properties* endpoint of the root URL of a Thing. For example, to retrieve the properties of the smart door, an HTTP GET request is sent to the properties endpoint of the root URL of the smart door. A JSON array describing the Thing properties is returned. The array contains a JSON object for each specific property of the Thing. This object includes a short description of the property (e.g. identifier, name and any other attribute set by the Thing owner) and the current (i.e. last) measurement value of the Thing (e.g. last humidity value of DHT22) or the internal state of the Thing (e.g. the state of the door). So this operation returns a conceptual description as well as the last measured or state value for each property of the Thing. In the case of a smart door, only information about the state property of the Thing will be returned; in the case of DHT22, both temperature and humidity information will be included in the response JSON array.

The second operation on Properties Resource intends to retrieve the current value of a property. It requires sending an HTTP GET request to the */properties* endpoint of the root URL of a Thing followed by the specific Thing property name as a path parameter (e.g. *rootURL/properties/state* for the smart door actuator or *rootURL/properties/temperature* for the DHT22 sensor). The Context Broker service allows storing only the most recent measurement values of a sensor in the case of DHT22 or, the last observed state of the smart door. The implementation can be modified to return the most recent (e.g. the 10 latest) values. Past measurements can be also stored in a history database (as shown in Figure 3.1).

An update operation on the Properties Resource (e.g. update a specific property) requires sending an HTTP PUT request to the */properties* endpoint of the root URL of a Thing followed by the name of the property. The new property value(s) and the new timestamp are included in the request body. For example, an HTTP PUT request is used to update the state property of the smart door (e.g. by changing its value to open). This is a property update operation that is different from an action (e.g. open the door) and not an action execution command to open the door. In response, WTMs forwards an HTTP PUT request to the Context Broker service to update the existing property values and the property timestamp. A 204 NO CONTENT response and a header containing the property URL path are returned if the operation is successful.

The last operation on Properties Resource, updates multiple properties at once, by sending an HTTP PUT request to the */properties* path of the root URL of a Thing followed by a path name. This operation is used to update the values of multiple properties of a specific Thing (e.g. the temperature and the humidity of DHT22) using a single HTTP request. The request body contains a JSON array with the new value and the new timestamp of each property. In turn, WTMs issues an HTTP POST request to the Context Broker service to update the existing property values. The service utilizes the batch update operation of the Context Broker, which allows to create or update several entities with a single request. A 204 NO CONTENT response and a header containing the */properties* endpoint of the root URL of the Thing are returned provided that the operation is successful.

D. Actions Resource

The first operation retrieves a list of actions by issuing an HTTP GET request to the */actions* endpoint of the root URL of a Thing. This operation is meant to return an array of descriptions for the actions that the Thing may perform (e.g. open, close, lock, unlock for the smart door). In turn, the WTMs issues an HTTP GET request to the Context Broker to retrieve the entity that holds this information in the form of a JSON array (i.e. containing an identifier and a name for each possible action). This information can be set by the Thing owner.

The second operation intends to retrieve all recent executions of a specific action and issues an HTTP GET request to the */actions* endpoint of the root URL of a Thing followed by the specific action name as a path parameter. For example, a GET rootURL/actions/{actionName} (e.g. *rootURL/actions/lock*) will return an array of the recent executions of a specific action on the smart door (e.g. locking the smart door device), including information about the status of the action execution and a timestamp. This information can be retrieved by issuing an HTTP GET request to the Context Broker service. A JSON array is returned containing a separate object for each action execution. A time delay value might have been sent in the request body of specific action execution, to schedule the execution of the action at a later time. The value attribute can be omitted in commands for opening, closing, locking and unlocking the smart door.

The next operation is meant to execute an action. An HTTP POST request is sent to the */actions* endpoint of the root URL of a Thing followed by an action name in a path parameter (e.g. POST *rootURL/actions/lock* to lock the smart door). WTMs generates a unique identifier for each execution of an action which is stored in the Context Broker and can be used to retrieve the action. In fact, a random integer number is generated and its value is used to identify the action. The last operation retrieves the status of an action using its identifier as a path parameter. An HTTP GET request is sent to the */actions* endpoint of the root URL of a Thing followed by the name of the action and the execution identifier as a path parameter (e.g. GET *rootURL/actions/lock/156*). The operation is forwarded to the Context Broker where the specific action execution is stored.

E. Things Resource

The first operation retrieves the list of Things registered to the proxy. It requires sending an HTTP GET request to the */things* endpoint of the root URL of a Thing. WTMs forwards an HTTP GET request to the Context Broker service to retrieve all the entities of type Thing (i.e. GET *v2/entities?type=Thing*). A JSON array with the description of Things registered to the proxy is returned.

The second operation registers a new device in a specific infrastructure (and the proxy). The operation requires sending an HTTP POST request to the */things* endpoint of the root URL of a Thing. The request body must contain the TD of the new Thing. Following this request, WTMs issues an HTTP POST request with the new TD to the Context Broker. A 204 NO CONTENT response and a header containing the root URL of the Thing are returned as long as the operation is successful.

F. Subscriptions Resource

These are operations for handling subscriptions to Things. The first operation creates a new subscription, so a user or service may subscribe to a specific Web Thing resource. A client (i.e. user or service) may subscribe to specific Thing properties or actions. According to the Web Thing model, subscriptions are ideally supported using custom callbacks (i.e. Webhooks) which are naturally supported by Websocket protocol. In WTMs in particular, subscriptions are also realized via Webhooks and specifically using HTTP and the subscription mechanism of Context Broker service. An HTTP POST request is required to create a new subscription. More specifically, WTMs issues an HTTP POST request to Context Broker (i.e. to its */v2/subscriptions* endpoint) to store the new subscription. The subscription request body is set by the subscriber (client or service). A response header containing the subscription identifier is returned as long as the operation is successful. In addition, an HTTP PATCH request is also sent to the database of Context Broker to store the new subscription identifier as a separate entity. A 200 OK response is then returned as long as the whole operation is successful. In the following, as a result of successful subscription operation, the subscribed user or service gets notified (i.e. receiving asynchronous notifications) on changes of Thing's state information (e.g. new temperature value). The subscription identifier is a 24-digit hexadecimal number, generated by the subscription mechanism of the Context Broker service.

The second operation retrieves a list of subscriptions made to a specific Thing or Web Thing resource. The operation issues an HTTP GET request to the */subscriptions* endpoint of the root URL of a Thing. WTMs in particular, issues an HTTP GET request to the Context Broker service where the subscriptions are stored. A JSON array containing all these subscriptions is returned. According to the Web Thing model, all stored subscriptions are retrieved (not only the ones made to a specific Web Thing resource). However, WTMs enables retrieval (in a JSON array) of the subscriptions that refer to a specific resource (i.e. Thing). For example, to retrieve the subscriptions made to a registered smart door, an HTTP GET request is issued (i.e. GET *http://34.122.93.207:5001/subscriptions*).

The next operation is meant to retrieve a particular subscription (i.e. to a resource of a Thing) using its subscription identifier as a path parameter. The operation requires sending an HTTP GET request on the */subscriptions* endpoint of the root URL of a Thing followed by the subscription identifier as a path parameter (e.g. GET *rootURL/subscriptions/5a82be4d093af1b95ac0f730*). Following this, WTMs forwards the request to Context Broker service (i.e. to */v2/subscriptions/5a82be4d093af1b95ac0f730* service endpoint) where the specific subscription is stored. A JSON representation of the subscription is returned in the response.

The last operation deletes a subscription using its subscription identifier. The operation issues an HTTP DELETE request to the */subscriptions* endpoint of the root URL of a Thing followed by the subscription identifier as a path parameter. WTMs in particular, issues an HTTP DELETE request to the Context Broker service where the specific subscription is stored (i.e. DELETE */v2/subscriptions/identifier*). The subscription is removed and a 200 OK response header is returned.

3.4 Comparing WoT implementations

Thingweb node-wot and Webofthings.js are representative open-source implementations that are compared with WTMs, in terms of completeness: to what extent an implementation supports all operations foreseen by Web Thing Model specification (rather than those suggested by WoT). There is a lot of ambiguity as to what operations and protocols should be supported in WoT and, existing implementations differ significantly in both operations and protocols supported. The relevant comparison would open a broader discussion about the equivalence of operations in different contexts. As noted in Sections 3.1, 3.2 and 3.3, Thingweb adopts TDs and is compatible with the *W3C TD information model*, while Webofthings.js and WTMs are based on the Web Thing Model REST API so they are more compatible with the OpenAPI Thing template approach. That is, the latter two implementations do not use W3C TDs at all and they follow the REST API (i.e. endpoints, operations, payloads, etc) proposed in the Web Thing Model.

Similar to the Thingweb node-wot, some functions of Web Thing Model have not been implemented in Webofthings.js. For instance, the operation for updating a Thing description and the operation for updating multiple Thing properties at the same time using a single HTTP PUT request are not supported. As mentioned in Section 3.3.2, users or services should be able to subscribe to Web Thing resources (e.g. properties, actions) and get notified of any new values or value changes. Webofthings.js supports the creation of subscriptions using the Websocket protocol. However, the rest of the subscription operations have not been implemented (i.e. would need a context broker to implement such operations on subscriptions). Similar to Thingweb.node-wot, operations on subscriptions (e.g. the retrieval of a list of subscriptions or of information about a specific subscription and the deletion of a subscription) are not supported. In contrast to both frameworks, WTMs aims to fully implement the Web Thing Model and attempts to realize the functionality of all services as described in the model. Table 3.1 summarizes the results of this comparison.

Operation	**Web Thing Model service (WTMs)**	**Thingweb node-wot**	**Webofthings.js**
Retrieve a Web Thing	√	√	√
Update a Web Thing	√		
Retrieve the model of a Thing	√		√
Update the model of a Thing	√		
Retrieve a list of properties	√	√	√
Retrieve the value of a property	√	√	√
Update a specific property	√	√	√
Update multiple properties at once	√	√	
Retrieve a list of actions	√	√	√
Retrieve recent executions of an action	√		√
Execute an action	√	√	√
Retrieve the status of an action	√	√	√
Retrieve a list of Web Things	√	√	
Add a Web Thing to a gateway	√	√	
Create a subscription	√	√	√
Retrieve a list of subscriptions	√		
Retrieve information of a subscription	√		
Delete a subscription	√		

Table 3.1: Comparison of WoT reference implementations

Compared to existing implementations, WTMs is complete (i.e. it implements all Web Thing Model model operations) while being more flexible in certain cases; it allows WoT operations to address certain Thing properties rather than handling the Thing Descriptions as a unit (i.e. as Thingweb does).

3.5 WTMs Performance

WTMs is deployed using Docker in a medium flavor (2 vCPUs, 4,096Gb RAM and 20Gb SSD disk drive capacity) Virtual Machine (VM) in the Google Cloud Platform (GCP). Individual services (i.e. all WoT Model services, Orion Context Broker service and a MongoDB) are deployed within a Docker Engine as containers. Experimental results are taken using simulated (but realistic) data obtained from software simulating the operation of physical devices at a home and in a city environment. This allowed shifting the emphasis of the work from device-specific functionality (e.g. IoT transmission protocols and vendor-specific device functionality) to the actual performance of WTMs. To generate a large data set, the software produces pseudo-random measurements in the same value range and form as a real sensor.

WTMs is tested in a smart city scenario with 1,000 actuator Things that provide observations of two properties (e.g. temperature and pressure). Each Thing is capable of executing actions and receiving subscriptions. Context Broker service stores 7,000 data entities in total comprising information about Things and hypothetical users. More specifically, the MongoDB of the Context Broker services comprises 1,000 Web Thing descriptions, 1,000 Thing model descriptions, 2,000 Thing properties (1,000 for each of the two observable properties), 1,000 possible Thing actions, 1,000 Web Thing action executions and 1,000 subscriptions to Thing resources.

Apache Benchmark (AB)[71] is used to stress WTMs with 1,000 simultaneous requests (for each operation) for varying values of concurrency representing operations executing in parallel. Table 3.2 summarizes the performance (i.e. response time) of the most representative WoT service requests for three values of concurrency. All measurements of time are averages of over 1,000 requests. In all cases, the performance of WoT requests improves with the simultaneous execution of requests (i.e. the Apache HTTP server switches to multitasking) reaching their lowest values for concurrency=100. Even with concurrency=200, the average execution time per request is close to real-time in most cases. All requests are forwarded to the MongoDB service of Context Broker service (in fact MongoDB is part of the service). The request that updates multiple properties (two properties in our case) is slower than a simple read (i.e. GET) request.

[71] https://httpd.apache.org/docs/2.4/programs/ab.html

Service request	**Response Time (ms), c=1**	**Response Time (ms), c=100**	**Response Time (ms), c=200**
Retrieve a list of properties	41.84	10.59	10.77
Update the model of a Thing	44.70	13.53	13.65
Update a specific property	47.88	16.73	17.10
Update multiple properties at once	59.62	28.43	28.75
Retrieve a list of Web Things	53.37	20.47	20.81
Retrieve a list of actions	43.07	8.74	8.84
Retrieve recent executions of an action	41.37	9.46	9.59
Retrieve a list of subscriptions	51.00	18.73	29.47
Execute an action	45.54	16.47	17.85

Table 3.2: Performance of WTMs

3.6 Integrating WTMs in a SOA architecture

iSWoT [41] is an example architecture that incorporates WTMs proxy service. Building upon principles of SOA design, iSWoT is implemented as a composition of RESTful microservices communicating over HTTP in the cloud. iSWoT is a Semantic WoT architecture integrating the following desirable characteristics: (a) it is highly configurable and modular and supports generation of fully customizable applications by reusing services and devices; (b) leveraging flow-based programming combined with SWRL (i.e. a rule-based language for ontologies in OWL), new applications can be generated with the aid of user-friendly interfaces; (c) all services are reusable, implement fundamental functionality and offer a public interface allowing secure connections with other services (even third party ones); (d) it is also interoperable and expandable (i.e. services can be added or removed) while being secure by design: all services are protected by an OAuth2.0 mechanism. Access to services is granted only to authorized users (or authorized services) based on user roles and access policies.

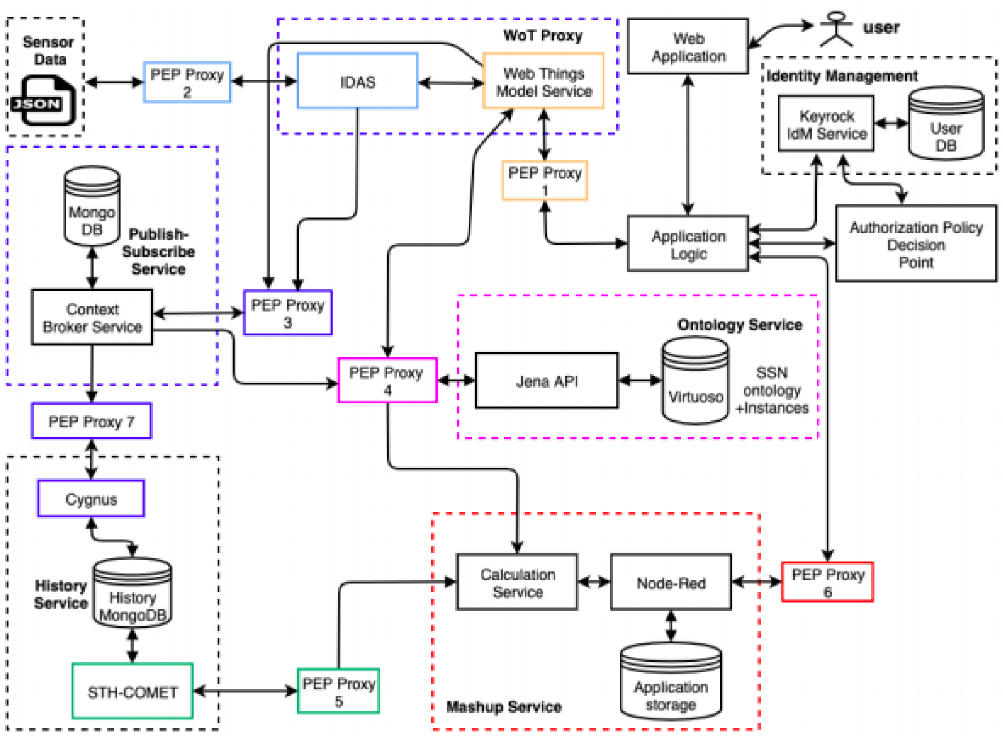


Figure 3.2: iSWoT architecture

Figure 3.2 illustrates iSWoT's architecture. Besides WoT proxy, the following groups of services are identified:

a) *Publication and Subscription:* Things registered to WoT Proxy can publish information to this service. It receives measurements from devices registered to IDAS service and makes this information available to other services and users based on subscriptions. Each time a new sensor registers to WoT Proxy, a new entity is created in the service (and also to the ontology). Each time a Thing value (e.g. sensor measurement) becomes available, this component is updated and a notification is sent to entities subscribed to the Thing. The service holds the most recent values from all registered sensors in a MongoDB database. History (past) measurements are forwarded to the History database.

b) *History database*: Collects data flows from Publication and Subscription service. The time series created from the history of data is stored in a MongoDB as (a) raw (unprocessed) values as received from devices and, (b) aggregated (processed) values. More specifically, maximum, minimum and average values over predefined time intervals (e.g. every hour, day, week etc.) are stored. This service is implemented using Cygnus[72]. The History database is implemented using MongoDB. STH-Comet[73] implements a query interface for MongoDB.

c) *Ontology*: It is the knowledge base component of iSWoT that handles IoT general-purpose and application-specific knowledge. The term Thing stands either for physical entities (e.g. Web devices) or for virtual entities (i.e. definitions of Thing

[72] https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Cygnus
[73] https://fiware-sth-comet.readthedocs.io/en/latest/index.html

categories). Physical entities are related (i.e. are instances) of virtual entities in SOSA ontology [38]. iSWoT supports seven sensor types (i.e. atmospheric pressure, temperature, humidity, luminosity, precipitation, human presence at home and wind speed). All are defined as subclasses of SOSA class *Sensor*.

d) *Application Mashup*: This service facilitates the development of new applications by re-using information residing in the History database and the ontology. The service is realized with the aid of Node-Red[74], an open-source flow-based programming tool for the IoT. Applications created are stored as a JSON entity in Application storage (i.e. a MongoDB database).

e) *Application logic*: Application logic orchestrates, controls and executes services running in the cloud. When a request is received (from a user or service), it is dispatched to the appropriate service. User requests are issued on the Web interface. First, a user logs in to iSWoT with a login name and password. The user is assigned a role (by the cloud administrator) and receives a token encoding his or her access rights (i.e. the authorization to access iSWoT services). This is the responsibility of the User identification and authorization service. Each time application logic dispatches the request to another service, the token is attached to the header of the request. It is the responsibility of the security mechanism to approve (or reject) the request. In iSWoT, all public services are protected by a security mechanism. iSWoT users can access the system using a Web interface. Application owners can issue requests for available devices and subscribe to selected devices; customers can issue queries to select applications available for subscriptions.

f) *Security*: Implements access control services based on user roles and access policies. Initially, users register to iSWoT to receive a login name, a password and a role (i.e. customer, application owner, or infrastructure owner) encoding user's access rights. This is the responsibility of the cloud administrator. Once a user is logged-in, she/he is assigned an OAuth2 token encoding her or his identity. The token remains active during a session. A session is initialized at login and remains active during a time interval which is also specified in advance. User respective access rights are described by means of XACML[75] (i.e. a vendor-neutral declarative access control policy language based on XML). Keyrock identity manager[76] is an implementation of this service. For each user, an XACML file is stored in Authorization Policy Decision Point (PDP)[77] service; user profile information is stored in the User database.

Services offering a public interface are protected by a security mechanism (i.e. they do not expose their interface to the Web without protection). This security mechanism is realized by means of the Policy Enforcement Proxy (PEP)[78] service. Each public service is protected by a separate PEP service. It is the responsibility of this service to approve or reject a request to the protected service. Each user request is forwarded to Application logic which dispatches the request to the appropriate service.

[74] https://nodered.org/
[75] https://fiware-tutorials.readthedocs.io/en/latest/administrating-xacml/index.html
[76] https://keyrock.docs.apiary.io/#reference
[77] https://authzforce-ce-fiware.readthedocs.io/en/release-5.1.2/
[78] https://fiware-pep-proxy.readthedocs.io/en/latest/

4

OpenAPI Thing Descriptions for the Web of Things

In Chapter 2, we reviewed the most common and noteworthy approaches used for the syntactic and semantic description of Web services. The W3C effort on the Web of Things aims at giving solutions for the description of Things and the interaction with them. Most cloud providers and developers, however, do not usually adopt these approaches to describe the functionality of Things. Although some providers or potential users may use some of these approaches (e.g. the W3C TD approach), services exposed by devices are not described using a commonly approved and adopted approach. The documentation of devices and their services is usually based on plain-text descriptions which may be proprietary (i.e. not open-source) and not consistent with the documentation of other providers and manufacturers.

In the following sections, we present our approach that builds upon the OpenAPI and specifically the Semantic OpenAPI Specification for the effective and efficient description of Things exposed as Cloud services. Although the Web Thing Model of W3C is not a recommendation, it proposes a simple REST API that allows a client (i.e. user or service) to easily interact with Things. Our approach builds on this REST API to introduce a common OpenAPI template of specific resources and operations (i.e. a specialization of OpenAPI) that applies to all Things, regardless of their features. Although this work does not adopt the TDs of W3C to allow the interaction with Things, it follows the common principles[79] of the WoT Architecture recommendation. For example, our approach enables the interaction of clients with Things on the web architecture using a RESTful API and it can allow the mutual interworking of different eco-systems using web technology. It provides a Thing description approach that shows clients how to use Things (e.g. read information about them, invoke actions on them, subscribe to them) with minimum implementation logic, using a number of specific, well-defined operations. In addition, our work is compliant with the principles of flexibility, compatibility, scalability and interoperability in the WoT.

[79] https://www.w3.org/TR/wot-architecture/#sec-requirements-principles

We discuss the reasons that led us to the adoption of the OpenAPI Specification as well as the adoption of semantically enriched OpenAPI definitions. Moreover, we illustrate how these descriptions can be used to define a generic OpenAPI Thing description template and how this template can be instantiated to the description of specific Things. We propose a mechanism for automatically generating OpenAPI Thing descriptions (i.e. JSON or YAML files) based on specific input provided by a user. We advocate that service descriptions be transformed into ontologies using the mechanism provided in [39] to take advantage of Semantic Web tools (e.g reasoners and query languages) for service discovery. Finally, we demonstrate how service discovery can be realized by providing specific SPARQL query examples on the ontology (i.e. generated from an OpenAPI Thing description) and their results.

4.1 Semantic OpenAPI Thing Descriptions

Aiming to choose a description language suitable for Things and their operations, we had to examine many aspects that would affect our decision. Considering devices as Web services, we should choose an approach that would properly describe them in detail. More specifically, since most Cloud services are provided as Web services based on the REST architectural model, we needed a description language for RESTful services. Besides, we should decide on an approach that fits well with HTTP. Devices do not use HTTP but different, energy-efficient protocols such as CoAP, MQTT, etc. Therefore, we aim to describe the functionality offered by Things as long as they get connected to the cloud or to a gateway, as this is the basic assumption of the Web of Things. HTTP is a common Web protocol that enables communication on the Web, allowing us to avoid IoT protocols - their complexity and peculiarities. We should also consider that we need a simple description language based on JSON that could be easily adopted by users with different background knowledge. Last but not least, the description language should be accompanied by useful tools that facilitate the interaction of users with a service.

In this context, we adopt the OpenAPI Specification as the description language for Things, as well as the semantic annotations proposed by the Semantic OpenAPI Specification [17]. OpenAPI is particularly well-suited for the description of RESTful services and as such, Things are handled as REST. This might be a disadvantage when Things have properties which are not supported by REST (e.g. security properties such as security levels in BLE protocol, encryption for the Zigbee protocol, etc.). OpenAPI is a simple but mature framework providing both human and machine-readable service descriptions, which are detailed and can be understood and discovered by humans and machines. Besides being the current industry's standard for the description of REST services, the selection of OpenAPI was based primarily on the popularity of the specification. The capabilities and tools it offers for facilitating user interaction with a service (Section 2.9.3) were also taken into consideration. OpenAPI's active community is constantly working to improve its tooling support as well as the specification itself. In addition, the OpenAPI Initiative is powered by large companies such as Google, Microsoft, IBM, Oracle and many others, attempting to standardize a description mechanism for RESTful services. By adopting the Semantic OpenAPI Specification approach, OpenAPI properties can be semantically annotated, so that their meaning is not vague. This solution enables service discovery by users and machines. Therefore, we consider that semantically enriched OpenAPI service descriptions are suitable for defining Things and the operations they support.

In Section 2.10 we described the objects provided by the OpenAPI Specification for the description of services. We advocate that general-purpose OpenAPI objects and properties for REST services can be specialized for Web Things. The services that Things may expose can then be described as RESTful Web services, by taking advantage of all the capabilities of the OpenAPI specification (version 3.0).

More specifically, the Info object can be used to provide non-functional information for a Thing and its exposed Web service. The ExternalDocumentation object provides additional metadata for the device API. The Server object can define the location of the API servers used for a specific device. The Paths object can be used to define all the available service paths for the Thing as well as their operations. The Components object should hold reusable objects for the whole service document. Moreover, the Security Requirements object can list the security schemes (i.e. defined under the Components object) used to protect the API of a device. All security schemes can be used to protect the service exposed by a Thing. The security schemes supported by the OpenAPI specification have already been highlighted in Section 2.10. The Schemas object describes the request and response payloads based on JSON Schema. Therefore, the Schemas object should be used to define the request and response payloads used in Thing's API operations. Examples are used to define reusable example values (e.g. for schema attributes). Parameters are used to specify all operation parameters. Finally, links in OpenAPI are "*somewhat similar to hypermedia*"[80] and they can support link functionality for Things, and callbacks are asynchronous requests, so they can support subscription functionality for Things.

The semantic extensions proposed by Semantic OpenAPI (i.e. discussed in Section 2.10) can be used to describe Things and their functions without ambiguities. That is, OpenAPI external properties (i.e. *x-properties)*, which have been presented in Table 2.1, can be adopted to semantically annotate the OpenAPI descriptions of Things. Therefore, if the meaning of Thing properties is ambiguous, they can be mapped to equivalent properties in semantic models (i.e. vocabularies or ontologies). More specifically, each Schema object can be associated with a semantic model using the *x-refersTo* extension property. The *x-kindOf* extension property may be used to define a specialization between an OpenAPI property of a Thing and a semantic model (e.g. a class). The *x-mapsTo* extension property may also be used to denote that a Schema property in a Thing's service document is semantically equivalent with another property in the same document. In addition, extension properties can be used in an OpenAPI Thing description to clarify the meaning of the members in a collection of objects (*x-collectionOn*), for grouping Schema objects by type (*x-onResource*), and also for clarifying the meaning of the Thing's operations (*x-operationType*).

In general, detailed vocabularies (e.g. *www.schema.org*) can be referred to using x-properties and thus define Things for an application domain. The semantic meaning of the service is captured by the OpenAPI ontology [17, 18]. Moreover, we assume that a Thing may also support subscriptions. A subscription is the result of subscribing to a

[80] https://swagger.io/docs/specification/links/

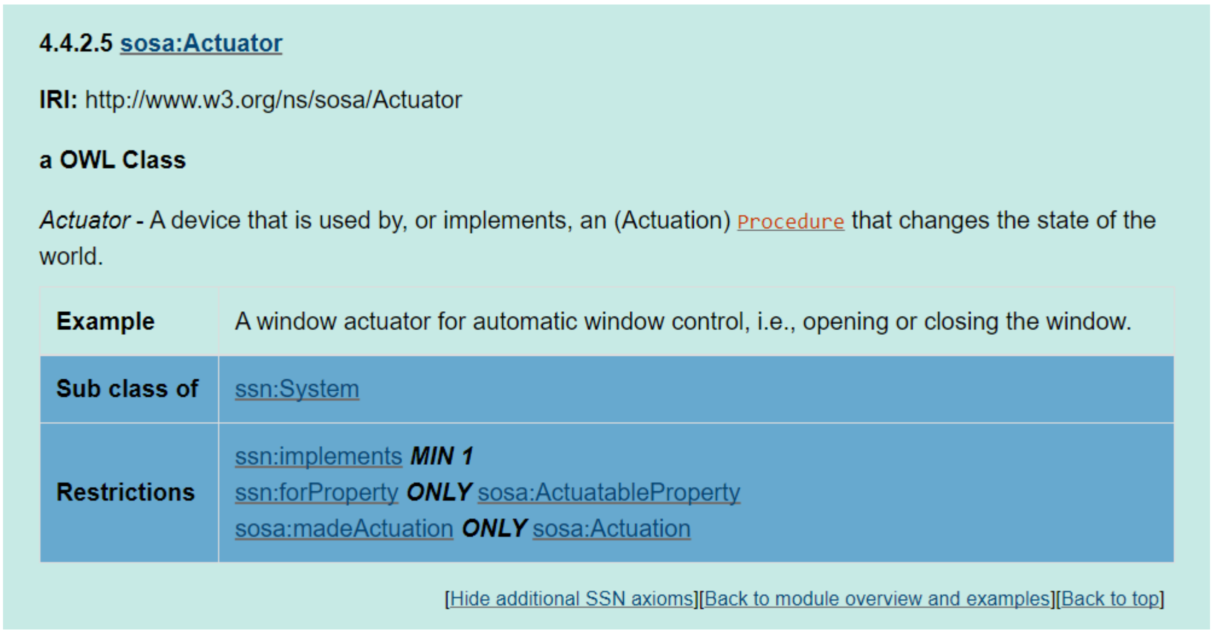
specific resource of a Thing (e.g. a particular property or action) to get notified of changes of the Thing's state information (e.g. new temperature value). The subscriptions are stored in a storage structure so that they can be retrieved by a subscription identifier (TD supports subscriptions to events that might not be stored).

Listing 4.1 shows how *x-refersTo* is used to semantically associate the Actuator type of the smart door to the SOSA ontology [38]. The URI *http://www.w3.org/ns/sosa/Actuator* is used for that purpose. The *x-kindOf* extension property is used to semantically annotate the Thing properties (i.e. id, name) with concepts in *www.schema.org* vocabulary.

Listing 4.1: Webthing Schema in OpenAPI Thing Description for a smart door device

```
Webthing:
  required:
    - id
    - name
    - type
  type: object
  x-refersTo: 'http://www.w3.org/ns/sosa/Actuator'
  properties:
    id:
      type: string
      default: SmartDoor
      x-kindOf: 'http://schema.org/identifier'
    name:
      type: string
      example: IoTSmartDoor
      x-kindOf: 'http://schema.org/name'
    description:
      type: string
      example: 'A Smart Door is an electronic door which can be sent
commands to be locked or unlocked remotely. It can also report on its current
state (OPEN, CLOSED or LOCKED).'
      x-refersTo: 'http://schema.org/description'
    createdAt:
      type: string
      format: date-time
    updatedAt:
      type: string
      format: date-time
    tags:
      type: array
      items:
        type: string
        example: smart door
  xml:
    name: Webthing
```

Figure 4.1 illustrates the description of the Actuator class provided in the SOSA ontology specification[81], which is identified by the respective URI. This URI is used to semantically annotate and map the Actuator type of the smart door to the SOSA ontology, thus specifying its meaning.


4.4.2.5 sosa:Actuator

IRI: http://www.w3.org/ns/sosa/Actuator

a OWL Class

Actuator - A device that is used by, or implements, an (Actuation) Procedure that changes the state of the world.

Example	A window actuator for automatic window control, i.e., opening or closing the window.
Sub class of	ssn:System
Restrictions	ssn:implements ***MIN 1*** ssn:forProperty ***ONLY*** sosa:ActuatableProperty sosa:madeActuation ***ONLY*** sosa:Actuation

[Hide additional SSN axioms][Back to module overview and examples][Back to top]

Figure 4.1: Actuator class in SOSA ontology

Listing 4.2 describes a delete operation on a subscription using its subscription identifier. A human may refer to the description of the operation to understand its intended purpose, but a machine needs additional information which is provided by the *x-operationType* extension property. The value of the property is a URL pointing to the concept (i.e. in a semantic model) that semantically describes the operation type. The Action type of the *www.schema.org* vocabulary provides a detailed hierarchy of Action sub-types that can be used by the property.

Listing 4.2: Semantic annotation of a smart door's delete operation

```
'/subscriptions/{subscriptionID}':
    delete:
      tags:
        - Subscriptions
      summary: Delete a subscription
      description: In response to an HTTP DELETE request on the destination
URL of a subscriptions an Extended Web Thing must either reject  the request
with an appropriate status code or remove (unsubscribe) the subscription and
return a 200 OK status code.
```

[81] https://www.w3.org/TR/vocab-ssn/#SOSAActuator

```
      operationId: deleteSubscription
      x-operationType: 'https://schema.org/DeleteAction'
      parameters:
        - name: subscriptionID
          in: path
          description: The id of the specific subscription
          required: true
          style: simple
          explode: true
          schema:
            type: string
            example: 5fd23faccde6be05da68bcfb
      responses:
        '200':
          description: OK
        '404':
          description: Not found
```

4.2 OpenAPI Thing Description examples

In the following, we will present two real-world device examples (i.e. a DHT22 sensor and a smart door actuator) which are described using OpenAPI Thing descriptions. In Section 4.3, we will introduce the general OpenAPI Web Thing template that applies to all devices.

4.2.1 Smart door actuator

The smart door actuator device exposes information about its current state (i.e. open, closed, locked), includes a lock-unlock actuator, and provides the possible resources for this (i.e. Thing, Properties, Actions and Subscriptions resources). The OpenAPI document of the device describes all subscription operations: the operation for subscribing to a smart door resource, the subscription retrieval operations and the subscription delete operation.

The smart door description is enriched with semantic annotations using extension properties. That is, it includes *x-properties* to annotate OpenAPI properties and map them to semantic models such as the SOSA ontology or the *www.schema.org* vocabulary. For instance, Listing 4.1 describes how *x-refersTo* is used to semantically associate the Actuator type of the smart door to the SOSA ontology. The *x-operationType* property in Listing 4.2 states that the type of the HTTP DELETE operation is clarified by *https://schema.org/DeleteAction*. The *x-kindOf* extension property is used in most cases to semantically annotate the schemas and the schema properties in the Components object of the document.

Regarding OpenAPI objects, the OpenAPI Thing description of the smart door contains:

- An Info field (i.e. information for the device API).
- An External Documentation field.
- A Servers field (i.e. where the Thing's API servers are located).
- A Tags field that adds metadata to the tags used (i.e. by the Operation objects) to represent the Thing's resources. These Tag objects are described in Listing 4.3. Each Tag object represents a resource of the Thing.
- A Paths field that describes the relative paths for the service endpoints exposed for the smart door. The Paths object also includes other objects such as Operation objects, Response objects, Parameter objects, etc. The operation of retrieving the abstract description of the Thing is described in Listing 4.9. The operation of creating a subscription (i.e. subscribing) to a resource of the smart door is presented in Listing 4.12, while the operation of deleting a subscription is presented in Listing 4.2. Listing 4.10 and Listing 4.11 also illustrate operations included in the description of the smart door.
- A Components field (object) that specifies the schemas (i.e. Schema objects) used to describe request bodies, response bodies, etc. For instance, Listing 4.1 describes the schema of the abstract payload used to represent the device. The Schema object for the action execution payloads of the smart door is described in Listing 4.4.

Listing 4.3: Tag objects in the smart door OpenAPI description

```
tags:
  - name: Web Thing
    x-onResource: '''#/components/schemas/Webthing'''
    description: Operations on a Web Thing
    externalDocs:
      description: Find out more
      url: 'https://www.w3.org/Submission/wot-model/#web-thing-resource'
  - name: Properties
    description: Operations on Thing properties
    externalDocs:
      description: Find out more about Thing properties
      url: 'https://www.w3.org/Submission/wot-model/#properties-resource'
  - name: Actions
    description: Operations on Thing Actions
    externalDocs:
      description: Find out more about Thing Actions
      url: 'https://www.w3.org/Submission/wot-model/#actions-resource'
  - name: Subscriptions
    x-onResource: '''#/components/schemas/SubscriptionObject'''
    description: Operations on subscriptions
    externalDocs:
      description: Find out more about subscriptions
      url: 'https://www.w3.org/Submission/wot-model/#things-resource'
```

The description contains schemas describing the response payloads returned by the operations on the Properties and the Actions resources, and all the payloads of the Subscription operations. The schemas for the subscription operations are predefined in the template. Additional schemas (i.e. request and response payloads) can be defined by the user in the JSON input; in fact, the user is allowed to include in the input (and thus define) the most schemas of the description. The schemas are appended in Thing's description as Schema objects; these objects are included in the components field that holds various reusable schemas. For example, the user may define the general schema that describes the smart door (i.e. returned in the first operation) or the Schema that exposes the value of the smart door's property (i.e. the current state of the device) or the Schema that describes an action execution command of the client; to lock or unlock the smart door.

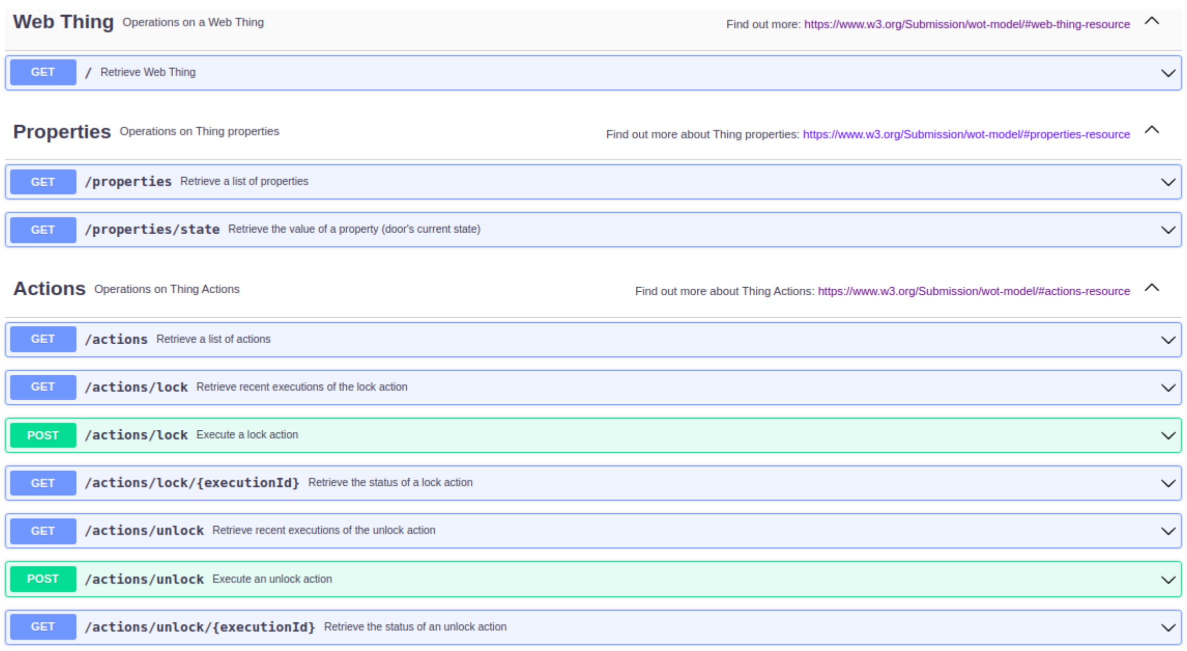
Listing 4.4 describes the schema used for the action executions of the smart door (i.e. lock, unlock). The schema includes an *id* property (string) to identify the action execution using a unique identifier (i.e. number), a status property (string) to declare the status of the particular action execution and a timestamp property (string, date-time format) for the action execution. The schema is semantically annotated with a URI that maps the schema with the *Actuation* class[82] of the SOSA ontology to declare that the particular schema represents an actuation.

Listing 4.4: Schema object for the action execution payloads of the smart door

```
ActionExecution:
  required:
    - id
    - status
    - timestamp
  x-kindOf: 'http://www.w3.org/ns/sosa/Actuation'
  type: object
  properties:
    id:
      type: string
      example: '223'
    status:
      type: string
      example: completed
    timestamp:
      type: string
      format: date-time
  xml:
    name: ActionExecution
```

[82] https://www.w3.org/TR/vocab-ssn/#SOSAActuation

Figures 4.2 and 4.3 demonstrate the endpoints and the descriptions for the smart door actuator OpenAPI description, as depicted in Swagger UI. In contrast to the DHT22 sensor example (i.e. presented next), which only supports properties and subscriptions, the smart door example supports both properties and actions, as well as subscriptions. Therefore, it includes all the possible operations provided by the OpenAPI Thing template.



Web Thing Operations on a Web Thing — Find out more: https://www.w3.org/Submission/wot-model/#web-thing-resource

GET `/` Retrieve Web Thing

Properties Operations on Thing properties — Find out more about Thing properties: https://www.w3.org/Submission/wot-model/#properties-resource

GET `/properties` Retrieve a list of properties

GET `/properties/state` Retrieve the value of a property (door's current state)

Actions Operations on Thing Actions — Find out more about Thing Actions: https://www.w3.org/Submission/wot-model/#actions-resource

GET `/actions` Retrieve a list of actions

GET `/actions/lock` Retrieve recent executions of the lock action

POST `/actions/lock` Execute a lock action

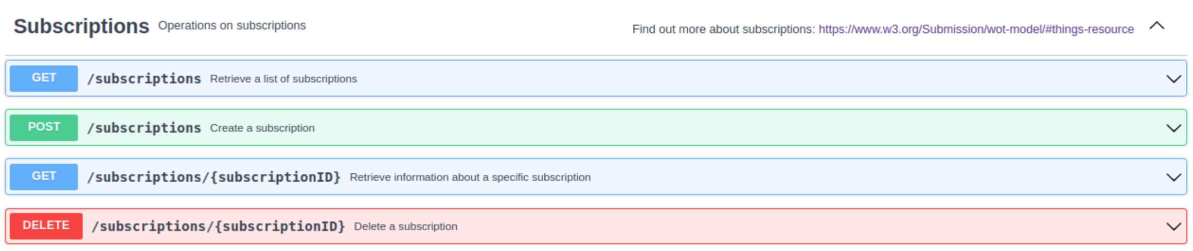
GET `/actions/lock/{executionId}` Retrieve the status of a lock action

GET `/actions/unlock` Retrieve recent executions of the unlock action

POST `/actions/unlock` Execute an unlock action

GET `/actions/unlock/{executionId}` Retrieve the status of an unlock action

Figure 4.2: Smart door service endpoints (1)



Subscriptions Operations on subscriptions — Find out more about subscriptions: https://www.w3.org/Submission/wot-model/#things-resource

GET `/subscriptions` Retrieve a list of subscriptions

POST `/subscriptions` Create a subscription

GET `/subscriptions/{subscriptionID}` Retrieve information about a specific subscription

DELETE `/subscriptions/{subscriptionID}` Delete a subscription

Figure 4.3: Smart door service endpoints (2)

The whole OpenAPI Thing description of the smart door (in YAML format) is listed in the Appendix. It is a very indicative example of our approach, as it contains all the possible operations proposed in the OpenAPI Thing template (i.e. all the operations related to properties and actions, and also subscriptions).

4.2.2 DHT22 sensor

The second real-world device example proposed in this work is a DHT22 sensor that measures temperature and humidity. This device exposes information about temperature and humidity properties, their measurement values and provides the possible resources for this (i.e. Thing, Properties and Subscriptions resources). The OpenAPI document of the sensor describes all subscription operations: the operation for subscribing to a DHT22 sensor resource, the subscription retrieval operations and the subscription delete operation.

Similar to the OpenAPI smart door description, the OpenAPI Thing description of the DHT22 sensor contains:

- An Info field.
- An External Documentation field.
- A Servers field.
- A Tag field that adds metadata to the tags used in the description.
- A Paths field that describes the relative paths for the service endpoints exposed for the DHT22 sensor. The Paths object also includes other objects such as Operation objects, Response objects, Parameter objects, etc. Listing 4.9 describes the operation of retrieving the sensor's abstract description. Listing 4.10 illustrates the operation of retrieving all the properties of the sensor (i.e. temperature and humidity). The operation of creating a subscription (i.e. subscribing) to a resource of the DHT22 sensor is presented in Listing 4.12, while the operation of deleting a subscription is presented in Listing 4.2.
- A Components field (object) that specifies the schemas (i.e. Schema objects) used to describe request bodies, response bodies, etc. Similarly to the smart door example, Listing 4.5 describes the general Web Thing schema that represents the device.

The DHT22 sensor description includes semantic annotations as well. In fact, several semantic annotations have been used in the DHT22 sensor description to describe specific concepts or characteristics of the device. For example, they define concepts such as observation, temperature property and humidity property and device characteristics such as sensor sensitivity, sensor accuracy, sensor precision and sensor frequency. OpenAPI objects and properties of the description (e.g. schemas) are mapped to classes (i.e. OWL classes) of the SOSA or the SSN ontology. Indicative examples of such semantic annotations are described in Listings 4.5 and 4.6.

Listing 4.5 is the Webthing schema (i.e. it describes the payload returned in the response of the first operation) that provides an abstract description of the DHT22 sensor, similar to the schema for the smart door in Listing 4.1. It is semantically enriched using the *x-refersTo* and the *x-kindOf* extension property. For instance, the *x-refersTo* annotation is used to map the Webthing schema with the *Sensor* class of the SOSA ontology (i.e. *http://www.w3.org/ns/sosa/Sensor*). Any device owner is free to define the schema that best fits the described device and add the semantic annotations that best describe the schema and its properties.

Listing 4.5: Webthing Schema in the OpenAPI Thing Description of a DHT22 sensor

```
Webthing:
  required:
    - id
    - name
  type: object
  x-refersTo: 'http://www.w3.org/ns/sosa/Sensor'
  properties:
    id:
      type: string
      default: DHT22
      x-kindOf: 'http://schema.org/identifier'
    name:
      type: string
      example: DHT22/AM2302
      x-kindOf: 'http://schema.org/name'
    description:
      type: string
      example: 'The DHT-22, also named as AM2302, is a digital-output
relative humidity and temperature sensor. It uses a capacitive humidity
sensor and a thermistor to measure the surrounding air, and spits out a
digital signal on the data pin.'
      x-refersTo: 'http://schema.org/description'
    createdAt:
      type: string
      format: date-time
    updatedAt:
      type: string
      format: date-time
    tags:
      type: array
      items:
        type: string
        example: temperature sensor
  xml:
    name: Webthing
```

Listing 4.6 is the schema used to represent a specific temperature measurement of the DHT22 sensor. The schema includes a *temp* property (i.e. represents the temperature measurement value, e.g. 25) and a *timestamp* property (i.e. represents the measurement timestamp). The *x-kindOf* property is used to map the schema with the *Observation* class of the SOSA ontology (i.e. *http://www.w3.org/ns/sosa/Observation*). The *x-kindOf* property is also used to map the *temp* property of the schema with the *hasSimpleResult* OWL datatype property[83] of the SOSA ontology (i.e. represents the simple value of an observation or an actuation). In addition, the same extension property is used to map the *timestamp* property of the schema with the *resultTime* OWL datatype property[84] of the SOSA ontology. This datatype property represents the instant of time when an observation or actuation or sampling activity was completed. The humidity measurement schema is described similarly to the temperature measurement schema.

Listing 4.6: Schema object for the temperature measurement payload of the DHT22 sensor

```
TempMeasurement:
  type: object
  x-kindOf: 'http://www.w3.org/ns/sosa/Observation'
  properties:
    temp:
      type: integer
      example: 25
      x-kindOf: 'http://www.w3.org/ns/sosa/hasSimpleResult'
    timestamp:
      type: string
      format: date-time
      x-kindOf: 'http://www.w3.org/ns/sosa/resultTime'
```

Listing 4.7 describes the schema for the response payload of the operation that retrieves all properties of the DHT22 sensor. The device supports two properties, so the response JSON array that includes both properties (i.e. temperature and humidity) is implemented using polymorphism (i.e. using the property *anyOf*). The OpenAPI specification supports the combination of model definitions using the properties *allOf*, *oneOf* and *anyOf* of JSON Schema. The corresponding schema for the smart door properties includes only the schema of the state property in the array (i.e. the *anyOf* property is used as well), as the device supports only this property.

Listing 4.7: Schema object for the properties response payload of the DHT22 sensor

```
PropertiesResponse:
  anyOf:
    - $ref: '#/components/schemas/TempProperty'
    - $ref: '#/components/schemas/HumProperty'
  xml:
    name: PropertiesResponse
```

The schema of the temperature property referenced in Listing 4.7 is described in Listing 4.8. The schema describes the temperature property of the sensor and thus some basic characteristics of the sensor, by including information in the form of schema

[83] https://www.w3.org/TR/vocab-ssn/#SOSAhasSimpleResult
[84] https://www.w3.org/TR/vocab-ssn/#SOSAresultTime

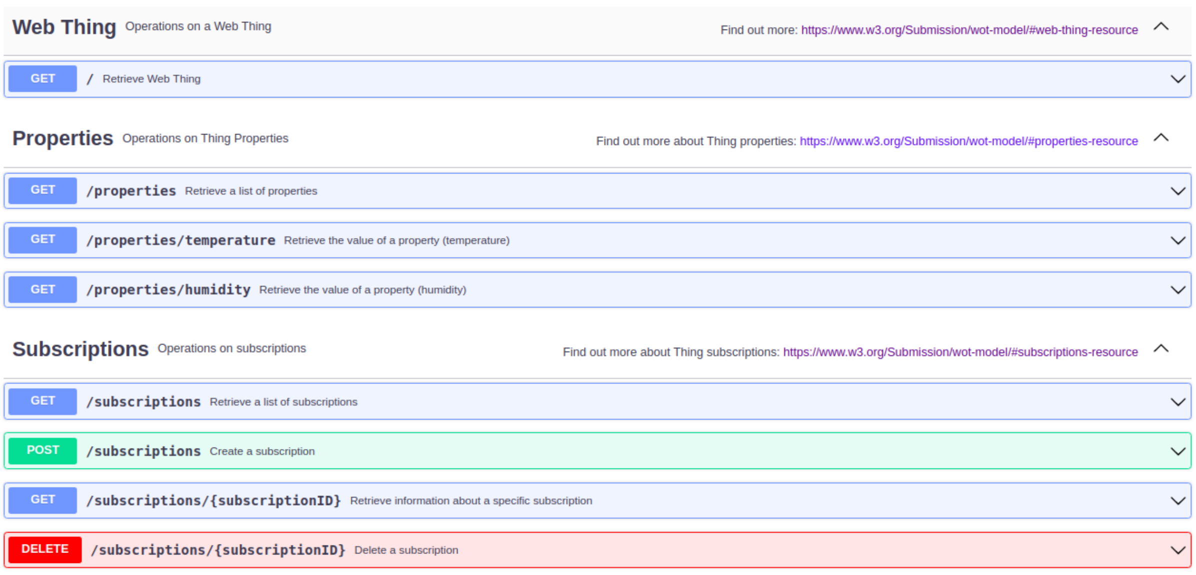
properties. The schema includes an id and a name of the property, recent measurement values (i.e. using the schema of Listing 4.6), the range of temperature value (i.e. measurement unit, minimum and maximum value), the sensor accuracy, the sensor sensitivity, the sensor precision and the sensor frequency. The schema is semantically annotated and mapped to the *SystemCapability* class of the SSN ontology using the *x-kindOf* extension property to describe the temperature property concept. In addition, the schema properties, as well as their properties, are mapped to SSN classes or to concepts of the schema.org vocabulary (i.e. using the *x-kindOf* or the x-*refersTo* property). Taking advantage of the SSN ontology, we are able to directly map several concepts of the DHT22 sensor (e.g. sensor precision, sensitivity and accuracy) to ontology classes that specify their exact meaning.

Listing 4.8: Schema object for the temperature property payload of the DHT22 sensor

```
TempProperty:
  type: object
  required:
    - id
    - values
  x-kindOf: 'http://www.w3.org/ns/ssn/systems/SystemCapability'
  properties:
    id:
      type: string
      default: DHT22_temperature
      x-kindOf: 'http://schema.org/identifier'
    name:
      type: string
      example: Temperature
      x-kindOf: 'http://schema.org/name'
    values:
      $ref: '#/components/schemas/TempMeasurement'
    range:
      type: object
      x-kindOf: 'http://www.w3.org/ns/ssn/systems/Condition'
      properties:
        unit:
          type: string
          default: DegreeCelsius
          x-refersTo: 'https://schema.org/unitText'
        minValue:
          type: number
          format: float
          default: -40
          x-refersTo: 'https://schema.org/value'
        maxValue:
          type: number
          format: float
          default: 80
          x-refersTo: 'https://schema.org/value'
    sensorAccuracy:
      type: object
      x-refersTo: 'http://www.w3.org/ns/ssn/systems/Accuracy'
      properties:
        range:
          type: object
          x-kindOf: 'http://www.w3.org/ns/ssn/systems/Condition'
          properties:
            unit:
              type: string
              default: DegreeCelsius
              x-refersTo: 'https://schema.org/unitText'
            minValue:
              type: number
              format: float
              default: -0.5
              x-refersTo: 'https://schema.org/value'
            maxValue:
              type: number
              format: float
              default: 0.5
              x-refersTo: 'https://schema.org/value'
    sensorSensitivity:
      type: object
      x-refersTo: 'http://www.w3.org/ns/ssn/systems/Sensitivity'
      properties:
        unit:
          type: string
          default: DegreeCelsius
          x-refersTo: 'https://schema.org/unitText'
        value:
          type: number
          format: float
          default: 0.1
          x-refersTo: 'https://schema.org/value'
    sensorPrecision:
      type: object
      x-refersTo: 'http://www.w3.org/ns/ssn/systems/Precision'
      properties:
        range:
          type: object
          x-kindOf: 'http://www.w3.org/ns/ssn/systems/Condition'
```

```yaml
            properties:
              unit:
                type: string
                default: DegreeCelsius
                x-refersTo: 'https://schema.org/unitText'
              minValue:
                type: number
                format: float
                default: 0.2
                x-refersTo: 'https://schema.org/value'
              maxValue:
                type: number
                format: float
                default: 0.2
                x-refersTo: 'https://schema.org/value'
        sensorFrequency:
          type: object
          x-refersTo: 'http://www.w3.org/ns/ssn/systems/Frequency'
          properties:
            unit:
              type: string
              default: sec
              x-refersTo: 'https://schema.org/unitText'
            period:
              type: integer
              format: int32
              default: 2
              x-refersTo: 'https://schema.org/value'
      xml:
        name: TempProperty
```

The whole OpenAPI Thing description of the DHT22 sensor (in YAML format) is listed in the Appendix. Figure 4.4 demonstrates all the service endpoints of the DHT22 sensor OpenAPI description proposed in this work.


Web Thing Operations on a Web Thing Find out more: https://www.w3.org/Submission/wot-model/#web-thing-resource

GET / Retrieve Web Thing

Properties Operations on Thing Properties Find out more about Thing properties: https://www.w3.org/Submission/wot-model/#properties-resource

GET /properties Retrieve a list of properties

GET /properties/temperature Retrieve the value of a property (temperature)

GET /properties/humidity Retrieve the value of a property (humidity)

Subscriptions Operations on subscriptions Find out more about Thing subscriptions: https://www.w3.org/Submission/wot-model/#subscriptions-resource

GET /subscriptions Retrieve a list of subscriptions

POST /subscriptions Create a subscription

GET /subscriptions/{subscriptionID} Retrieve information about a specific subscription

DELETE /subscriptions/{subscriptionID} Delete a subscription

Figure 4.4: DHT22 sensor service endpoints

4.3 OpenAPI Web Thing template

The OpenAPI Web Thing template employs a JSON (or YAML) description format which is common to all Things. It is a valid OpenAPI document that can be handled by all known OpenAPI tools (e.g. Swagger editor, code generator etc.). All Things expose properties (e.g. the humidity of a sensor or the state of an actuator) and, depending on their type, they may also support actions (e.g. a smart window that opens and closes). If supported by an implementation, a Thing can also support subscriptions to specific events. Subscriptions can describe any number of HTTP requests that may arrive in response to an earlier HTTP request. For example, clients (i.e. users or services) can subscribe to events using a publish/subscribe pattern implemented using the WebSocket protocol, to get notified on changes of Things state information (e.g. new temperature value).

The building blocks of the OpenAPI Thing template are its resources. Each resource of a Thing supports operations. The *Thing resource* provides an operation for retrieving the abstract description of a device (e.g. device characteristics) in JSON format. The *Properties resource* supports operations for each property and, the *Actions resource* supports operations for each action that a Thing can perform. In relation to the *Subscriptions resource*, the OpenAPI Web Thing template provides particular operations that refer to subscriptions. For example, there is an operation for subscribing to specific events (e.g. the change of a property value or the execution of a specific action).

Thing resources and their supported operations in the OpenAPI Web Thing template are described below.

a) Thing resource: It is an abstract description of a Web Thing. The operation retrieves the description of the Thing by issuing an HTTP GET request to the root URL of the Thing, meaning its IP address and the default port that follows the IP (e.g. *http://34.122.93.207:5001/MySmartDoor*). The operation (i.e. GET, PUT, POST, etc.) and its corresponding path are specified in the Paths object of the OpenAPI description. The Schema object used to describe a Thing payload (i.e. returned in the response body) is included in the Components object. The Thing contains an identifier named as id (string), an indicative name (string), a description of the device (string), the date it was created or updated or registered to an application (date-time string) and tags for all devices. This operation, which is standard for all Things, is illustrated in Listing 4.9. As noted above, the Paths object represents the particular endpoint and operation. A tag is used to define the resource related to the operation (i.e. Web Thing). A Response object, including a reference (i.e. $ref) to a Schema object (i.e. Webthing), is used for the description of the response message. The Schema object essentially defines the abstract payload that briefly describes a device. Indicative examples of the Webthing schema have been provided in Listings 4.1 and 4.5.

Listing 4.9: Thing Resource operation example

```
paths:
  /:
    get:
      tags:
        - Web Thing
      summary: Retrieve Web Thing
      description: 'In response to an HTTP GET request on the root URL of a
Thing, an Extended Web Thing must return an object that holds its
representation.'
      operationId: retrieveWebThing
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Webthing'
        '404':
          description: Not found
```

b) Properties resource: It defines the properties of a Thing (e.g. temperature, humidity) and describes measurements of properties (e.g. temperature and humidity values for DHT22 sensor) or, the internal state of a Thing (e.g. the current state of the smart door). This resource applies to all devices.

The first operation retrieves a list of properties. It is realized by issuing an HTTP GET request to the /properties endpoint of the root URL of the device. The endpoint (i.e. /properties) and operation (i.e. GET, PUT, POST, etc.) are specified in the Paths object. A JSON array describing the Thing properties is returned in response to this operation. The array contains a JSON object for each property of the Thing. This object includes a short description of the property (e.g. identifier, name and any other attribute set by the Thing owner) and the current (i.e. last) measurement value of the Thing (e.g. last humidity value of DHT22) or, the internal state of the Thing (e.g. the state of a smart door). This operation also returns the description of each property. In the case of a sensor that observes more than one property (e.g. pressure, temperature), information about properties is included in a response array. This operation, which is standard for all Things, is described in Listing 4.10. The Schema object that describes a property of the device is defined in the Components object. If the Thing accepts one or more from a list of alternative properties (like the DHT22 sensor), the response JSON array is implemented using polymorphism (i.e. using the property *anyOf*) like in example of Listing 4.7. OpenAPI supports the combination of model definitions, as already highlighted. Similar to the example in Listing 4.9, a tag is used to define the resource related to the operation (i.e. Properties).

The second operation on Properties Resource retrieves the current value (or the most recent values) of a property. It requires sending an HTTP GET request to the */properties* endpoint of the root URL of the Thing followed by the specific Thing property name as a path parameter (e.g. *rootURL/properties/state* for a smart door). In case there is more than one state value, an array of JSON objects is returned in the response body. The particular endpoint (i.e. */properties/state*) and operation are also specified in the Paths object of the service description. The Schema object that describes the payload (i.e. temperature or humidity) is defined in the Components object.

Listing 4.10: Properties Resource operation example

```yaml
/properties:
  get:
    tags:
      - Properties
    summary: Retrieve a list of properties
    description: 'In response to an HTTP GET request on the destination URL
of a properties link, an Extended Web Thing must return an array of Property
that the initial resource contains.'
    operationId: retrieveWebThingProperties
    responses:
      '200':
        description: OK
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/PropertiesResponse'
      '400':
        description: Invalid ID supplied
      '404':
        description: Not found
```

c) Actions resource: It defines the allowed actions on a Thing, such as execution commands (e.g. a command sent by a client to a window actuator to open). In all cases below, the Schema object that describes the execution information of a specific action is returned in the response body and it is defined in the Components object.

The first operation retrieves a list of actions by issuing an HTTP GET request to the */actions* endpoint of the root URL of the Thing. This operation returns an array of descriptions for the actions that the Thing may perform (e.g. locking and unlocking for a smart door). The response is in the form of a JSON array which contains an identifier and a name for each possible action. The particular endpoint (i.e. */actions*) and operation are specified in the Paths object. An action identifier and an action name are properties of the corresponding Schema object.

The second operation retrieves all recent executions of a specific action and issues an HTTP GET request to the */actions* endpoint of the root URL of the Thing followed by the specific action name as a path parameter. For example, a GET *rootURL/actions/{actionName}* (e.g. GET *rootURL/actions/lock*) returns a JSON array of the recent executions of an action including information about the status of the action execution and a timestamp. A time delay value might have been sent in the request body to schedule the execution of the action at a later time. This operation is described in Listing 4.11. The operation is standard for all Things that support Actions. However, the schema used to describe the action execution (i.e. named as *ActionExecution*) is not standard, so a different schema can be defined for each Thing. That is, a device owner can choose the schema that best describes the actions performed by the device (e.g. a time delay of the action). Similar to the previous examples, a tag is used to define the resource related to the operation (i.e. Actions).

The next operation executes an action. An HTTP POST request is sent to the */actions* endpoint of the root URL of the Thing followed by an action name as a path parameter (e.g. POST *rootURL/actions/lock* to lock a smart door). The service may generate a unique identifier for each execution of an action which can be stored in a database and then be used to retrieve the action. That is, a unique integer number can be generated and its value can be used to identify the action. The operation is specified in the */actions/{actionName}* path (e.g. */actions/lock*) in the Paths object of the OpenAPI description.

The last operation retrieves the status of an action using its execution identifier (i.e. *executionId*): an HTTP GET request is sent to the */actions* endpoint of the root URL of a Thing followed by the name of the action and the execution identifier as a path parameter (e.g. GET *rootURL/actions/lock/156*). The endpoint for the particular action (i.e. */actions/lock/{executionId}*) and the respective operations are specified in the Paths object of the service description. The execution information is the same as those used in the second operation.

Listing 4.11: Actions Resource operation example

```
/actions/unlock:
  get:
    tags:
      - Actions
    summary: Retrieve recent executions of the unlock action
    description: 'In response to an HTTP GET request on an Action URL, an
Extended Web Thing must return an array that lists the recent executions of a
specific Action.'
    operationId: retrieveRecentExecutionsOfUnlockAction
    responses:
      '200':
        description: successful operation
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/ActionExecution'
      '404':
        description: Not found
```

d) Subscriptions resource: It describes subscriptions to Web Things (e.g. to their actions and properties). For example, users and services can subscribe to the humidity property of a specific sensor to get notified of changes in humidity.

The first operation creates a new subscription (i.e. a user or service may subscribe to a specific Web Thing resource). Subscriptions are ideally supported using custom callbacks (i.e. Webhooks) which are naturally supported by the Websocket protocol. An HTTP POST request is required to create and store a new subscription. The particular endpoint (i.e. */subscriptions*) and operation are specified in the Paths object. This operation is described in Listing 4.12. Although the subscription information is defined in the request body by the subscriber (i.e. client or service), the Schema object that describes the payload of the subscription is standard for all Things that support subscriptions. This schema can be a reusable object which is defined under the Components object. The object contains an indicative name, a description, the subscription type (e.g. *webhook*), the callback URL, an object containing (i.e. as object properties) the type and the name of the resource to which the subscription is made, the expiration date of the subscription and, a throttling parameter which is used to specify a minimum inter-notification arrival time for the subscription. A response header containing the subscription identifier is returned as long as the operation is successful (i.e. a 200 OK response is returned).

Listing 4.12: OpenAPI description of a subscription to a resource

```
/subscriptions:
  post:
    tags:
      - Subscriptions
    summary: Create a subscription
    description: An Extended Web Thing should support subscriptions for its
resources.
    operationId: createSubscription
    x-operationType: 'https://schema.org/CreateAction'
    requestBody:
      description: Create a new subscription
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/SubscriptionRequestBody'
      required: true
    responses:
      '200':
        description: OK
      '404':
        description: Not found
```

The second operation retrieves a list of subscriptions made to a specific Thing or Web Thing resource. The operation issues an HTTP GET request to the */subscriptions* endpoint of the root URL of the Thing. A JSON array containing the subscriptions made to a specific Web Thing resource is returned. For example, to retrieve the subscriptions made to the smart door, an HTTP GET request is issued (i.e. GET *http://34.122.93.207:5001/subscriptions*). The operation is specified in the */subscriptions* path in the Paths object. The Schema object describing the payload for each subscription included in the JSON array is defined in the Components object.

The next operation retrieves a subscription using its subscription identifier. The operation requires sending an HTTP GET request on the */subscriptions* endpoint of the root URL of the Thing followed by the subscription identifier as a path parameter (e.g. GET *rootURL/subscriptions/5a82bg4d893bf1b95ae0f730*). A JSON representation of the subscription is returned in the response. This operation is described in Listing 4.13. The particular endpoint (i.e. */subscriptions/{subscriptionID}*) and operation are included in the Paths object of the smart door. The subscription identifier (string) is defined as a Path parameter under the Paths object. The Schema object that describes the subscription (i.e. in the response body) is standard for all Things that support subscriptions, so it is predefined in the template. It is also a reusable object which is defined in the Components object. The subscription identifier (subscriptionID), which is defined using a Parameter object, is mapped to the *id* property of the SubscriptionObject schema (i.e. defined in the Components object) using an *x-mapsTo* extension property. Therefore, the subscriptionID parameter defined in this operation is semantically equivalent with the specific Schema property. As in the previous examples, a tag is used to define the resource related to the operation (i.e. Subscriptions).

The last operation deletes a subscription (using its subscription identifier) by issuing an HTTP DELETE request to the */subscriptions* endpoint of the root URL of the Thing followed by the subscription identifier as a path parameter. If successful, the subscription is removed and a 200 OK response header is returned. The operation is included in the relative path (i.e. /subscriptions/{subscriptionID}) in the Paths object of the service description. The subscription identifier (string) is defined as a Path parameter in the Paths object.

Listing 4.13: Subscriptions Resource retrieval operation example

```
'/subscriptions/{subscriptionID}':
  get:
    tags:
      - Subscriptions
    summary: Retrieve information about a specific subscription
    description: 'In response to an HTTP GET request on a Subscription URL,
an Extended Web Thing must return a JSON representation of the subscription.
The JSON representation should be the same as the one returned for that
subscription in ''Retrieve a list of subscriptions''.'
    operationId: retrieveInfoAboutSubscription
    parameters:
      - name: subscriptionID
        in: path
        description: The id of the specific subscription
        required: true
        style: simple
        explode: true
        x-mapsTo: '#/components/schemas/SubscriptionObject.id'
        schema:
          type: string
          example: 5fd23faccde6be05da68bcfb
    responses:
      '200':
        description: OK
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/SubscriptionObject'
      '404':
        description: Not found
```

Figure 4.5 illustrates the general document structure of the OpenAPI Web Thing template. The template is common to all devices. This means that OpenAPI Thing descriptions of different devices can be formed based on this template and their supported features (e.g. properties, actions, subscriptions). The template structure is divided into boxes that represent the main sections (i.e. components) of an OpenAPI description which are illustrated in Figure 2.5 and described in Section 2. For example, the *Info* section describes non-functional information for the device API, the *Paths* section describes all the operations of a Thing and the *Components* section holds reusable objects (i.e. schemas, responses, callbacks, links, etc.). The template structure is also grouped into five categories of information (i.e. using five different colors) based on the conditions that determine whether and in which cases they will be included in a Thing description (or not). In the following, the five categories of information are specified:

a) First category: This category includes *mandatory* information (i.e. it must be included) that applies to all Thing descriptions. More specifically, the Web Thing Resource (i.e. the Web Thing Resource tag and the operation for retrieving the Web Thing description) applies to all Things - regardless of what features they support - and cannot be changed**.** The same applies to the Properties resource; the Properties Resource tag is the same for all Things, while the operations depend on the properties supported by each Thing (e.g. pressure, humidity). That is, the Properties Resource operation types are always the same, but they refer to different properties. This category of information is illustrated using the *dark red color* in the template figure.

b) Second category: This category includes *mandatory* information that only applies to descriptions of Things that provide actions or to descriptions of Things that may provide subscriptions (or to Things that provide both of them). In particular, the Actions Resource tag and the Subscription Resource tag are always the same (i.e. default values predefined in the template), but they only apply to devices that provide actions and subscriptions respectively. The same applies to the Subscriptions Resource operations and the schemas related to subscription operations. There are standard subscription operations and schemas which only apply to device APIs that support subscriptions. This category of information is illustrated using the *light red color* in the figure.

c) Third category: This category includes *mandatory* information that only applies to descriptions of Things that provide actions and includes particular operation types that apply to all actuators. The Actions Resource operations types (e.g. execute an action, retrieve the status of an action) are standard and they are predefined in the OpenAPI Web Thing template. However, the particular action operations depend on the physical actions supported by each Thing (e.g. lock, unlock, move, turn on/off). This category of information is illustrated using the *green color* in the figure.

d) Fourth category: This category includes *mandatory* information that can either be defined by the owner of a device or be set to the default value predefined in the template. For example, the Web Thing schema that generally describes a device and all schemas related to properties and actions can be set by the device owner or be set to the default schema of the template for Web Things. In other words, the payload attributes of devices, properties and actions in OpenAPI Thing descriptions can be set by device owners and be included in the Components objects of Thing descriptions. If not set by them, all these schemas (or some of them) can be set to the default values of schemas predefined in the OpenAPI Web Thing template and be included in the Components section. The default schemas, however, might be different for every device, as they depend on the names of properties and actions of each Thing. In addition, the API servers defined in the Servers section (i.e. as an array of Server objects) are either defined by a device owner or they are set to the default server value (i.e. "/"). According to the OpenAPI Specification, "*if the servers property is not provided, or is an empty array, the default value would be a Server Object with a url value of /*". This category of information is illustrated using the *blue color* in the figure.

e) Fifth category: This category includes mainly *optional* information that can only be defined by a device owner (i.e. there are no default values in the template). In particular, the Info section, the External Documentation section and the Security section can only be defined by the device owners. However, only the Info Section is mandatory. Furthermore, any responses, parameters, examples, request bodies, headers, security schemes, links and callbacks used in the REST API of a device can only be defined by the device owner and be included under the Components object of the OpenAPI document or not be defined at all. This category of information is illustrated using the *white color* in the figure.

The OpenAPI Web Thing template, therefore, is consistent with the two OpenAPI Thing description examples described above. It benefits from OpenAPI and describes several features of device APIs (e.g. operations, schemas, security, links, callbacks); not all of them are included in the DHT22 sensor and smart door description examples provided above.

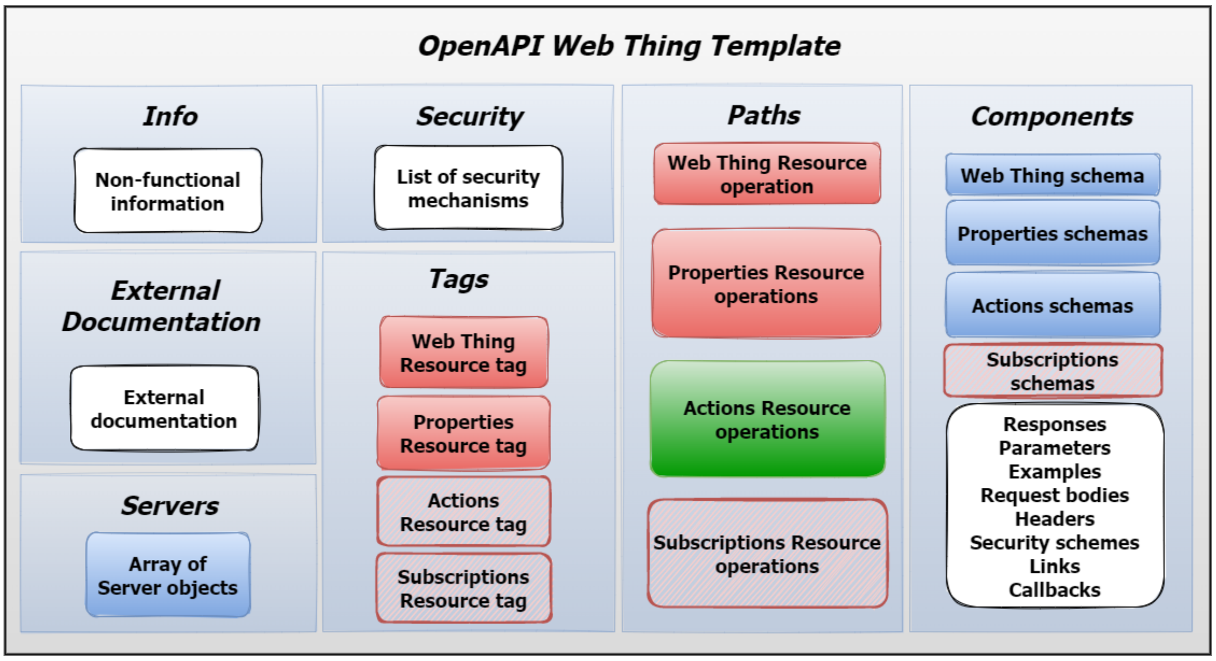


Figure 4.5: OpenAPI Web Thing Template structure

4.4 A mechanism for generating OpenAPI Thing Descriptions

The flow-chart of Figure 4.6 summarizes the mechanism that generates the OpenAPI description of a Thing from user input. The input comprises: a) the standard OpenAPI Thing Description template of Section 3.3.2 that applies to all Things and, b) a payload in JSON with the user settings (e.g. security settings) and the Thing characteristics that will be instantiated to the template. The user specifies the necessary information that characterizes the device and the functionality it supports (e.g. the properties it provides, the actions it performs, etc.). The output of this mechanism is the OpenAPI description of the Thing (in YAML or JSON format). The mechanism is a RESTful service itself which is implemented in Python Flask and is available on Github[85] for download and testing. It applies to any device as long as its functionality can be exposed using REST. As a use case, the complete OpenAPI Thing descriptions for a smart door and a DHT22 sensor device (along with their corresponding JSON files given as input to the mechanism), can be found in the same Github address.

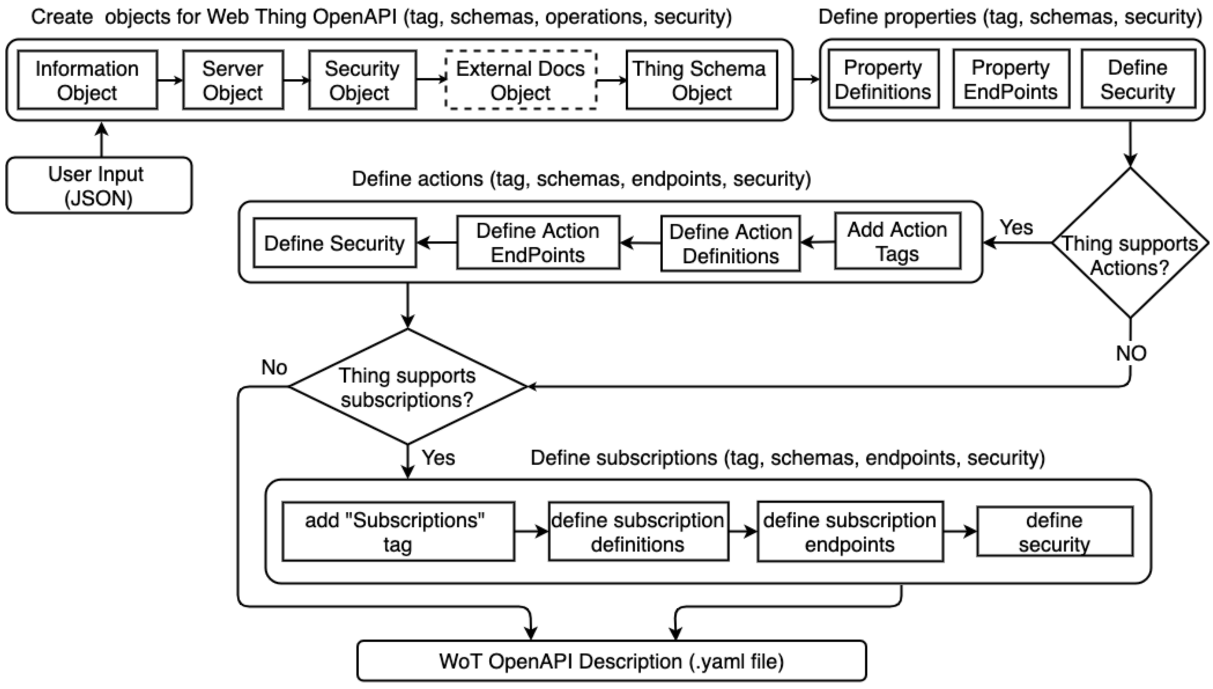

Create objects for Web Thing OpenAPI (tag, schemas, operations, security)
Information Object
Server Object
Security Object
External Docs Object
Thing Schema Object
Define properties (tag, schemas, security)
Property Definitions
Property EndPoints
Define Security
User Input (JSON)
Define actions (tag, schemas, endpoints, security)
Define Security
Define Action EndPoints
Define Action Definitions
Add Action Tags
Yes
Thing supports Actions?
NO
No
Thing supports subscriptions?
Yes
Define subscriptions (tag, schemas, endpoints, security)
add "Subscriptions" tag
define subscription definitions
define subscription endpoints
define security
WoT OpenAPI Description (.yaml file)

Figure 4.6: Generating an OpenAPI Thing Description

Initially, the process creates the OpenAPI objects for the Web Thing description: Info, Security, Servers, Schema and (optionally) External document objects are created and appended in the OpenAPI Thing template. As long as the user has set external documentation information, the process creates an External Documentation object. Next, the process appends the Thing's description payload (as a Schema object) under the Webthing model object (as in Listing 4.1). This payload describes the device and its features. Basic payload attributes (e.g. identifier, name, description, etc.) are mandatory. Available, Security Requirement objects are set next (e.g. HTTP Authentication, OAuth2.0, OpenID Connect). The process reads a list of available servers as an array of Server objects. The values of all OpenAPI objects are defined and instantiated to the respective objects in the next stages.

Schemas, parameters, paths (i.e. endpoints), operations and security information are defined. For example, apart from the */properties* path which is standard for all Things, a new path is appended to the service description for each particular property of the device. If the device supports actions, the mechanism appends a standard Action Tag. Input regarding the Actions resource and their security settings is provided. For example, the */actions* path (standard for all Things that perform actions) is appended to the Paths object. All relative action execution operations and their response payload models

[85] https://github.com/Emiltzav/wot_openapi_generator

(Schemas) are also defined in the input. If the user wishes to set a request body for the action execution operations (i.e. the commands to lock or unlock the smart door), this can be specified in the input as well. If the device supports subscriptions, a Subscriptions Tag is added to subscription objects (i.e. paths, operations, schemas, etc.) along with the security settings related to the Subscription resource. Subscription paths, operations and Schemas are predefined in the OpenAPI Thing Description template (i.e. they are the same for all Things).

Listing 4.14 is a short but indicative example of the user input that can be provided for the mechanism in JSON format for a typical smart door device description. This example includes only some essential information specified by the user for the device. For example, the device type (i.e. actuator), the properties and the actions it supports, the information that subscriptions can be supported, the Webthing schema (i.e. abstract description of the device) and also some non-functional information about the service are included. For the sake of brevity, the rest of the schemas that could also be defined by a user have been omitted. However, the user is allowed to specify many more schemas for the description (e.g. the payload of an action execution request or an action execution retrieval response payload), by properly including them in the input of the mechanism. If the user does not define them (i.e. as in the example below), default schemas, which are predefined by the OpenAPI Thing template, are appended to the service description. The smart door OpenAPI description example provided in the Appendix includes several default schemas of the template such as the schema used to return a list of actions supported by the device (i.e. realized by sending an HTTP GET request to the */actions* endpoint). All default schemas are appended to the OpenAPI Thing description, in case they have not been specified by the user.

Listing 4.14: User input for a smart door OpenAPI description

```
{
    "info": {
      "title": "A Smart Door device OpenAPI Thing description",
      "description": "An OpenAPI Thing description for a smart door device
that exposes its current state and supports lock and unlock actions (client
commands).",
      "contact": {
        "email": "atzavaras@isc.tuc.gr"
      },
      "license": {
        "name": "Example license",
        "url": "http://www.example.com/licenses/LICENSE-2.0.html"
      }
    },
    "externalDocs": {
      "description": "Find out more about the smart door actuator",
      "url": "https://www.example.com/actuators/smart-door"
    },
    "servers": [
      {
        "url": "http://localhost:5000/smart-door",
        "description": "A testing server"
      }
    ],
    "type_of_thing": "actuator",
    "supported_properties": [ "state" ],
    "supported_actions": [ "lock", "unlock" ],
    "sub_support": "yes",
    "webthing_schema": {
      "required": [
        "id",
        "name",
        "type"
      ],
      "type": "object",
      "x-refersTo": "http://www.w3.org/ns/sosa/Actuator",
      "properties": {
        "id": {
          "type": "string",
          "default": "SmartDoor",
          "x-kindOf": "http://schema.org/identifier"
        },
        "name": {
          "type": "string",
          "example": "IoTSmartDoor",
          "x-kindOf": "http://schema.org/name"
        },
        "description": {
          "type": "string",
          "example": "A Smart Door is an electronic door which can be sent
commands to be locked or unlocked remotely. It can also report on its current
state (OPEN, CLOSED or LOCKED).",
          "x-refersTo": "http://schema.org/description"
        },
        "createdAt": {
          "type": "string",
          "format": "date-time"
        },
        "updatedAt": {
          "type": "string",
          "format": "date-time"
        },
        "tags": {
          "type": "array",
          "items": {
              "type": "string",
              "example": "smart door"
          }
        }
      },
      "xml": {
        "name": "Webthing"
      }
    }
}
```

4.5 Ontology translation process

Aiming to facilitate the search for Things on the Web, OpenAPI Thing Descriptions are translated to an ontology ([17], [18]). As highlighted in [39], the instantiation of OpenAPI descriptions and services is a rather complicated process. The reasons can be many. For example, the same property may appear many times in the same document (with the same or different meaning), with different scopes (i.e. local property declarations overshadow global ones) or, it can be nested inside other properties. Furthermore, a *Schema* object can be defined as an extension or a composition of existing models (e.g. using the *allOf* property) which add extra complexity to the instantiation process. The algorithm proposed by the authors of [39] handles all these issues and it is presented in [39].

More specifically, Algorithm 1 in [39] scans the OpenAPI document of a service and instantiates *OpenAPI* objects to classes of the ontology. In particular, after uploading the ontology in the memory, the algorithm will scan the *OpenAPI* file to extract *info, servers, securitySchemes, securityRequirements, tags* and *paths* objects. These objects will become individuals of their corresponding classes.

The *OpenAPI* object is mapped to class *Document*. There may exist more than one appearances of servers or securityRequirements in an *OpenAPI* document. Property servers declare server information which applies across the description (global servers). This will be overwritten by server information defined in Path or Operations objects. Similarly, Security property declares security requirements. Security requirements declared by an operation will also override global declaration of security requirements. Property Tags contain the Tag objects for operations which are grouped. Through the *x-onResource* property Tag objects can associate operations with Schema objects.

Algorithm 3 in [39] illustrates the instantiation of *Tag* objects in the ontology and how *x-onResource* relations are handled. In Algorithm 4, Tag names and their associated Shapes are kept in a Map structure (*tagShapeMap*) that will be used when instantiating Operation objects. *tagShapeMap* defines a mapping (key, value) between a string and an individual. An entry to the *tagShapeMap* will contain the tag name and the corresponding

Shape individual. This will take place if an x-onResource annotation has been defined on a tag object (i.e. describing a Schema object). The Schema object will be converted to an individual of the Shape class and will be added inside the corresponding mapping (e.g. (tag.getName(), SchemaInd)).

Algorithm 2 shows how *Path* objects are converted to individuals of class Path. This is where *Operation* individuals are created with their respective properties (i.e. tag, security, parameter, server). This is done after creating individuals for Server and Parameter objects that may be declared in a Path. Server objects declared in *Path* (as stated in OpenAPI Specification) will override global declaration of *Server* Objects.

Algorithm 4 shows how the individuals of class *Operation* are created. When the *x-operationType* annotation is used, the *Operation* individual is also considered as an individual of the provided Semantic entity. The structural elements of the operation (e.g. *responseObject, requestBody, security and tag)* become properties of the *Operation* individual (e.g. *operationInd property:tag tagInd* is an example triple). Property method defines the type of an operation (get, put, etc.) which are already defined in the OpenAPI ontology. Parameter objects can be any of type Path, Query Header or Cookie and are instantiated to the corresponding classes (i.e. PathParameter, Query, Header and Cookie). Then, parameter individuals are associated to the Operation individual using the respective properties. Finally, property onPath associates an Operation individual with a Path individual.

The OpenAPI Specification allows the combination and extension of model definitions using the allOf property. However, in order to support polymorphism, the *discriminator* property is used to determine which *Schema* definition validates the structure of the model. A model defined using the *allOf* property becomes a subclass of the model it extends. In the example of Listing 1.1 in [39], classes *Male* and *Female* become sub-classes of Person. An OpenAPI service description may have *Schema* objects sharing common properties. Instead of describing these properties for each Schema repeatedly, these *Schema* objects are described as a composition of common properties and schema-specific properties. If these Schema objects are not related to a semantic entity, we suggest annotating Schema objects with the property *x-refersTo: none*, and the algorithm should not attempt to relate these models with any semantic entity.

The ontology translation process has been incorporated into an application in the Web[86]. The main functionality of the application supports uploading OpenAPI descriptions of Things (in YAML or JSON) and their instantiation to the ontology. As a result, the ontology can be downloaded in TTL (turtle) format, can be searched using SPARQL or checked using Pellet. For example, the ontology resulting from the DHT22 sensor OpenAPI description is available (in TTL format)[87] in the ontologies section of the Web application described above.

[86] http://www.intelligence.tuc.gr/semantic-open-api/
[87] https://www.intelligence.tuc.gr/semantic-open-api/file/DHT22_OpenAPI_14-03-2022_10-42-11.ttl

4.6 Service discovery using the OpenAPI ontology

Listing 4.8 is the OpenAPI description of an action (i.e. resulting from the client's command) that creates a subscription to a DHT22 or a smart door resource. The *x-operationType* property states that the type of the HTTP POST operation is clarified by *https://schema.org/CreateAction*.

The SPARQL query of Listing 4.15 on the ontology is used to search for operations (i.e. included in Thing descriptions) that create a specific result. The query would respond with the names of all operations (on any Thing) with the type defined by *https://schema.org/CreateAction*. The particular action type is defined as "*The act of deliberately creating/producing/generating/building a result out of the agent*" in *www.schema.org*.

The response would include subscriptions (and other actions that create a result) to the smart door or DHT22. In this example, the query responds with the name of an operation (i.e. *createSubscription*) of the DHT22 sensor and its text description and summary, as illustrated at the bottom of Listing 4.15.

Listing 4.15: SPARQL query to retrieve Things that support create Actions

```
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api#>
SELECT ?graph ?operName ?operDescription ?operSummary WHERE  {
GRAPH ?graph {?operation a <https://schema.org/CreateAction> .
?operation openapi:name ?operName .
?operation openapi:description ?operDescription .
?operation openapi:summary ?operSummary .} }

Answer:
graph ==> http://example/DHT22_OpenAPI
operName ==> createSubscription
operDescription ==> An Extended Web Thing should support
subscriptions for the specific resource (DHT22).
operSummary ==> Create a subscription
```

Similarly, the SPARQL query of Listing 4.16 on the ontology is used to search for operations (i.e. included in Thing descriptions) that delete an entity. The query would respond with the names of all operations (on any Thing) with the type defined by *https://schema.org/DeleteAction*. The particular action type is defined as "*The act of editing a recipient by removing one of its objects*" in *www.schema.org*. The response would include actions (i.e. related to Things) that delete an entity (e.g. a subscription). In this example, the query responds with the name of an operation (i.e. *deleteSubscription*) of the DHT22 sensor and its text description, as illustrated in the bottom of Listing 4.16.

Listing 4.16: SPARQL query to retrieve Things that support delete Actions

```
PREFIX openapi: <http://www.intelligence.tuc.gr/ns/open-api#>
SELECT ?graph ?operName ?operDescription ?operSummary WHERE  {
GRAPH ?graph {?operation a <https://schema.org/DeleteAction> .
?operation openapi:name ?operName .
?operation openapi:description ?operDescription .
?operation openapi:summary ?operSummary .} }

Answer:
graph ==> http://example/DHT22_OpenAPI
operName ==> deleteSubscription
operDescription ==> In response to an HTTP DELETE request on the destination
URL of a subscriptions an Extended Web Thing must either reject  the request
with an appropriate status code or remove (unsubscribe) the subscription and
return a 200 OK status code.
operSummary ==> Delete a subscription
```

The ontologies resulting from the translation of OpenAPI Thing descriptions allow the discovery of operations related to Things. Furthermore, they could enable service synthesis (i.e. composition) using Things and their functionality, thus allowing the design and implementation of new WoT mashup applications.

5

Comparing W3C TDs with OpenAPI Thing Descriptions

In the following sections, we discuss the W3C Thing Description approach and we compare it to the OpenAPI Thing description approach: a) in terms of completeness of the JSON descriptions, and b) in terms of completeness of the corresponding ontologies. We analyze how each approach supports (or not) key aspects related to Things and their functionality, such as operations and interactions (e.g. properties, actions, events), protocol bindings, hypermedia controls and security requirements. We also present the advantages and disadvantages of every approach. Regarding the ontology aspect, we compare the ontologies and vocabularies proposed by W3C with the OpenAPI ontology and we analyze to what extent they can express the functionality of Things. Furthermore, we identify and compare the two approaches in relation to the hypermedia links support they offer.

5.1 W3C Thing Descriptions

Thing Description (TD) is a primary component of the WoT Architecture of W3C. It enables the discovery of services and resources related to a Thing. According to the WoT Architecture, TDs can be hosted in a directory service that provides a Web interface for registering and searching for Things. Thingweb node-wot[88] is an implementation of TD along with an implementation of Thing operations using a JavaScript API similar to the Web browser APIs. It provides an API Interface that allows scripts to interact with Things using Web protocols such as HTTP, HTTPS, CoAP, MQTT and Websockets. Thingweb node-wot implementation will be further discussed in Section 5.1.

[88] https://github.com/eclipse/thingweb.node-wot

The document structure of a TD has already been presented (Figure 2.1) and discussed in Section 2. It describes the Interactions, Data Schemes, Security Configuration and Protocol Binding of a Web Thing. It includes important information such as the Thing's name and identifier, its security requirements and the interactions (i.e. properties, actions, events) supported by the Thing. Similar to OpenAPI, TD uses a JSON serialization format. In addition, TD may be enhanced with a context field (i.e *@context*) for converting the JSON format to JSON-LD, which is practically equivalent to an ontology. TD leverages the context mechanism of JSON-LD to semantically annotate the information it provides. More specifically, it benefits from the independent vocabularies provided for the TD Information Model; the *core* TD Vocabulary, the *Data Schema* Vocabulary, the *WoT Security* Vocabulary and the *Hypermedia Controls* Vocabulary. Therefore, a TD instance can be enriched with additional vocabulary terms, utilizing these vocabularies and the TD Context Extension mechanism. An indicative TD example for an actuator device has already been presented in Section 2.2.

TD documents support Protocol Bindings for HTTP and HTTPS, and they also support the Webhook mechanism (i.e. HTTP callbacks). According to the Thing Description specification[89], a Thing can also implement extra Protocol Bindings (apart from HTTP); their number is not restricted. More specifically, the Thing Description specification highlights that "*other Protocol Bindings (e.g., for CoAP, MQTT, or OPC UA) are intended to be standardized in separate documents such as a protocol Vocabulary similar to HTTP Vocabulary in RDF 1.0*[90] *or specifications including Default Value*[91] *definitions*".

The TD approach also supports several common authentication schemes. More specifically, TD documents support Basic Authentication, Bearer Token Authentication, API key Authentication, OAuth 2.0 common flows, Digest Access Authentication and also Pre-Shared key authentication (PSK)[92]. A combination of security schemes can also be defined in the TD security definition (*ComboSecurityScheme*), as long as the security schemes can be combined. Apparently, when no authentication or other mechanism is required to access the resources of a Thing, this is also declared in the TD security definition (*NoSecurityScheme*).

According to the WoT Architecture, W3C WoT utilizes two kinds of hypermedia controls: Web *links* and Web *forms* (see Section 2.2). In this context, TD documents adopt a HATEOAS approach to support the description of hypermedia controls for Things. They describe Web links and Web forms related to Things and their interactions.

According to the TD specification, the *links* field in TDs "*provides Web links to arbitrary resources that relate to the specified Thing Description*", as highlighted in the TD specification[93]. W3C suggests that link relations in a TD "*can be used to describe relations such as to other Things (e.g., a Switch Thing controls a Lamp Thing), to a specific kind of Thing Models (e.g., a Thing Description is an instance of a specific Thing Model), or to further documentations information (e.g., device manual of a Thing)*". Moreover, the TD specification recommends the reuse of existing and established Link Relation definitions from IANA[94].

[89] https://www.w3.org/TR/wot-thing-description11/#other-protocol-bindings
[90] https://www.w3.org/TR/wot-thing-description11/#bib-http-in-rdf10
[91] https://www.w3.org/TR/wot-thing-description11/#dfn-default-value
[92] https://datatracker.ietf.org/doc/html/rfc4279
[93] https://www.w3.org/TR/wot-thing-description/#thing
[94] https://www.iana.org/assignments/link-relations/link-relations.xhtml

Listing 5.2 (i.e. originates from the TD specification[95]) illustrates how links can be used in a Thing Model definition of the W3D TD approach. In fact, the figure shows a device class model for a *Smart Lamp Control* that can be used as a template for the creation of specific TD instances. This Thing Model extends the definition of the *Basic On/Off TM* (Thing Model) illustrated in Listing 5.1 (i.e. originates from the TD specification[96]) with a dim property (i.e. for dimming). The *links* field in the definition is used to declare that the Smart Lamp Control Thing Model definition extends the Basic On/Off TM. For this purpose, the *rel* (i.e. relation) field is set with the value "*tm:extends*".

Listing 5.1: Basic On/Off Thing Model Definition (TD approach)

```
{
    "@context": ["http://www.w3.org/ns/td"],
    "@type" : "tm:ThingModel",
    "title": "Basic On/Off Thing Model",
    "properties": {
        "onOff": {
            "type": "boolean"
        }
    }
}
```

Listing 5.2: Links usage in a Thing Model (TD approach)

```
{
    "@context": ["http://www.w3.org/ns/td"],
    "@type" : "tm:ThingModel",
    "title": "Smart Lamp Control with Dimming",
    "links" : [{
        "rel": "tm:extends",
        "href": "http://example.com/BasicOnOffTM",
        "type": "application/td+json"
     }],
    "properties" : {
        "dim" : {
            "type": "integer",
            "minimum": 0,
            "maximum": 100
        }
   }
}
```

For example, the *item* link relation from IANA declares that the target IRI points to a resource that is a member of the collection represented by the context IRI. In the

[95] https://www.w3.org/TR/wot-thing-description11/#td-model-example-smart-lamp-control
[96] https://www.w3.org/TR/wot-thing-description11/#td-model-example-basic-on-off

example of Listing 5.3, which describes an electric drive that includes two motors, the item link relation declares that the target IRIs point to the resources (i.e. the two motors) that are members of the electric drive collection.

Listing 5.3: An item link relation example (TD approach)

```
...
{
    "links": [
        {
            "rel": "item",
            "href": "coaps://motor1.example.com",
            "type": "application/td+json"
        },
        {
            "rel": "item",
            "href": "coaps://motor2.example.com",
            "type": "application/td+json"
        }
    ]
}
...
```

Among other uses, Web linking can be used in TDs to point to the developer documentation of a Thing. In that case, the value "*service-doc*" can be used in the *rel* field, as demonstrated in Listing 5.4 (i.e. also originates from the TD specification[97]).

Listing 5.4: Link to developer documentation (TD approach)

```
...
"links": [{
    "rel": "service-doc",
    "href": "https://example.com/howTo",
    "type": "application/pdf"
}]
...
```

Web forms are included in the Properties, Actions and Events objects of TDs to inform clients how to interact with the corresponding properties, actions and events of devices. They illustrate the protocols (e.g. HTTP, HTTPS, CoAP) and the endpoints used to perform particular operations (e.g. to turn on a smart air-conditioner). Listing 5.5 describes an example of Web forms used in a specific TD. This example illustrates the *Properties* object that can be used in the TD of a DHT22 sensor. This object includes information for both properties supported by the sensor (i.e. temperature and humidity). The *forms* fields included in the Properties object (i.e. one for each property) show the

[97] https://www.w3.org/TR/wot-thing-description11/#example-26-link-to-developer-documentation

protocol (i.e. HTTPS) and the endpoints that a client can use to retrieve the current temperature value and the current humidity value of a DHT22 sensor. Another example of Web forms is described in Listing 2.1 where a *forms* field is used in the *Properties*, the *Actions* and the *Events* object of the smart door TD to illustrate how the client can interact with the Thing's provided affordances (e.g. door's state).

Listing 5.5: Example of Web forms in a DHT22 sensor TD

```
...
"properties": {
    "temperature": {
        "type": "number",
        "forms": [{"href": "https://dht22sensor.com/temperature"}]
    },
    "humidity": {
        "type": "number",
        "forms": [{"href": "https://dht22sensor.com/humidity"}]
    }
}
...
```

Webhooks (or HTTP callbacks) are requests sent in response to specific events, so they are used in asynchronous APIs. Listing 5.6 describes a Webhook Event example that originates from the TD specification[98]. The example illustrates how callbacks can be defined in a W3C TD. It assumes that there is a Thing (*WebhookThing*) that provides an Event affordance named *temperature*. This affordance pushes the most recent temperature value to the client using a Webhook mechanism; the Thing issues HTTP POST requests to a callback URI provided by the client. The example demonstrates how the callback URI can be specified using a write-only parameter (*callbackURL*) in the *subscription* member (i.e. field) of the temperature event affordance. In addition, a *subscriptionID* (read-only parameter), which is returned in the subscription response, is defined in the TD. The specification assumes that the Thing will periodically send POST requests to the defined callback URI including a payload defined by the *data* member of the temperature event affordance. The client is also allowed to unsubscribe by submitting the *unsubscribeevent* form (i.e. included in the *forms* field of the TD); an HTTP DELETE request is utilized for this purpose.

The specification notes that this process can be further automated, by utilizing the TD Context Extension mechanism and thus including proper semantic annotations. The client could also unsubscribe using the *cancellation* member of the temperature event and combine this with an *unsubscribeevent* form, which describes an HTTP POST request used to unsubscribe. The *uriVariables* member of the temperature event affordance informs the client to include the *subscriptionID* string in the URI (i.e. as a URI parameter) to cancel a specific subscription.

Listing 5.6: Webhook Event example in a TD

[98] https://www.w3.org/TR/wot-thing-description/#webhook-example-serialization

```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "id": "urn:dev:ops:32473-Thing-1234",
    "title": "WebhookThing",
      "description":  "Webhook-based  Event  with  subscription  and  unsubscribe
form.",
    "securityDefinitions": {"nosec_sc": {"scheme": "nosec"}},
    "security": ["nosec_sc"],
    "events": {
        "temperature": {
            "description": "Provides periodic temperature value updates.",
            "subscription": {
                "type": "object",
                "properties": {
                    "callbackURL": {
                        "type": "string",
                        "format": "uri",
                          "description": "Callback URL provided by subscriber
for Webhook notifications.",
                        "writeOnly": true
                    },
                    "subscriptionID": {
                        "type": "string",
                                 "description":  "Unique  subscription  ID  for
cancellation provided by WebhookThing.",
                        "readOnly": true
                    }
                }
            },
            "data": {
                "type": "number",
                 "description": "Latest temperature value that is sent to the
callback URL."
            },
            "cancellation": {
                "type": "object",
                "properties": {
                    "subscriptionID": {
                        "type": "integer",
                           "description": "Required subscription ID to cancel
subscription.",
                        "writeOnly": true
                    }
                }
            },
            "uriVariables": {
                "subscriptionID": { "type": "string" }
            },
            "forms": [
                {
                    "op": "subscribeevent",
                     "href":"http://localhost:8080/events/temp/subscribe",
                    "contentType": "application/json",
                    "htv:methodName": "POST"
                },
                {
                    "op": "unsubscribeevent",
                     "href":"http://localhost:8080/events/temp/{subscriptionID}",
                    "htv:methodName": "DELETE"
                }
            ]
        }
    }
}
```

5.2 Comparison of the TD and OpenAPI approaches

In the following, the TD of the WoT Architecture of W3C and the OpenAPI approach are compared based on their capacity to describe Web Things and their functionality (i.e. services exposed by Things). In Section 5.2.1, the comparison is based on the JSON representation format of the two approaches, while in Section 5.2.2. we discuss the capabilities of the corresponding ontologies.

5.2.1 Comparison based on JSON descriptions

TD and OpenAPI present remarkable similarities in terms of the JSON representations they provide. The particular properties and actions supported by Things are defined in both approaches, while the operations, specific endpoints and protocols used to interact with Things are provided by both TD and OpenAPI descriptions. The TD approach supports events related to Things (e.g. overheating of a lamp Thing) and also operations for subscribing and unsubscribing to particular events. The OpenAPI approach also represents subscriptions to specific events. The comparison of the two approaches is based on specific criteria such as the document specificity (Section 5.2.1.1), the protocol support (Section 5.2.1.2), the security support (Section 5.2.1.3), the hypermedia controls support (Section 5.2.1.4), the events and subscriptions support (Section 5.2.1.5), the data schema support (Section 5.2.1.6), the support of semantic extensions (Section 5.2.1.7) and finally the support of some general aspects (Section 5.2.1.8). The comparison results and conclusions are summarized in Section 5.2.1.9.

5.2.1.1 Document specificity

Compared to the OpenAPI approach, TD is a more abstract description of a Thing. It is, in fact, a much shorter document. An OpenAPI document describes every OpenAPI object and property in detail, in contrast to a TD. Listing 2.1, for example, shows how the lock operation that might be performed on a smart door Thing (i.e. to lock the door) can be briefly described by a TD using the *Actions* object and the *forms* field. The HTTP method of the operation (i.e. POST) is omitted in the TD approach, as it is a default assumption for the Actions object (see Section 2.2). On the contrary, the same operation can be described in more detail in OpenAPI, as illustrated in Listing 5.7. The type of HTTP method is declared as in every OpenAPI operation. A tag is used to declare the Actions resource in this approach, while in the TD approach the Actions object declares that. In the OpenAPI example, there is also an operation summary, a description, an operationId and the possible responses along with their descriptions.

Listing 5.7: The lock action execution of the smart door in OpenAPI

```
"/actions/lock": {
    "post": {
        "tags": [
            "Actions"
        ],
        "summary": "Execute a lock action",
        "description": "In response to an HTTP POST request on an Action URL
with valid parameters as request body (or no request body), an Extended Web
Thing must either reject the request with the appropriate status code or
queue a task to run the action. The action may not run immediately.",
        "operationId": "executeLockAction",
        "responses": {
            "204": {
                "description": "NO RESPONSE"
            },
            "404": {
                "description": "Not found"
            }
        }
    }
}
```

Although both approaches adopt a JSON serialization format, only TD uses the context mechanism of JSON-LD and converts the JSON format to JSON-LD. Therefore, TD is practically equivalent to an ontology and utilizes the context mechanism to semantically annotate the provided information. OpenAPI resorts to JSON or YAML and does not support JSON-LD which is more powerful.

5.2.1.2 Protocol support

Both TD and OpenAPI support the HTTP(s) protocol and they also support the Webhooks mechanism. TD documents may also refer to extra IoT protocols (e.g. CoAP, MQTT), while OpenAPI only supports HTTP(S). As already highlighted in Section 5.1, such (extra) protocols can be simply integrated into the TD by the usage of the TD Context Extension mechanism[99] provided by JSON-LD. For example, the TD of a sensor may specify an MQTT Protocol Binding. This feature is not supported in OpenAPI.

An OpenAPI description cannot be extended using semantic annotations to support extra protocols, in contrast to the W3C TD. OpenAPI intends to support the REST architectural style and especially the HTTP(S) protocol, so it is not designed to support additional protocols. Therefore, OpenAPI cannot be extended to support more transfer protocols like TD does. This is also highlighted by the members of the OpenAPI Initiative[100]. Even if we conventionally extended OpenAPI to support extra protocols such as CoAP (i.e. by using semantic extensions), this would not be compatible with the large set of tools provided for OpenAPI. The above comparison is summarized in Table 5.1.

Application Protocol	**W3C TD approach**	**OpenAPI TD approach**
HTTPS / HTTPS	*Supported*	*Supported*
Webhooks	*Supported*	*Supported*
CoAP	*Supported*	*Not supported*
MQTT	*Supported*	*Not supported*
Other protocols (e.g. Modbus, OPC UA)	*Supported*	*Not supported*

Table 5.1: Application protocols supported by the OpenAPI approach and the W3C TD approach

[99] https://www.w3.org/TR/wot-thing-description11/#sec-context-extensions
[100] https://github.com/OAI/OpenAPI-Specification/issues/777

5.2.1.3 Security support

Both TDs and OpenAPI describe the security requirements of services exposed by Things. The HTTP security schemes, vocabulary, and syntax in the WoT Architecture of W3C share many similarities with OpenAPI v3.0.1. More specifically, both TD and OpenAPI support security configuration for different HTTP authentication schemes (i.e. Basic Authentication, Digest Access Authentication, Bearer Token Authentication), for API key authentication (i.e. in headers, query strings or cookies), and OAuth 2.0 common flows. However, only the TD approach supports Pre-Shared key authentication (PSK)[101], which uses pre-shared keys such as TLS-PSK. PSK authentication is a method used for client authentication on wireless networks; it is used for WPA and WPA2 encryption. On the other hand, only the OpenAPI approach supports OpenID Connect authentication[102], which extends the OAuth2.0 protocol, by adding an identity layer on top of it. Similar to the *ComboSecurityScheme* that can be used in the TD security definition, the combination of security requirements is also possible in OpenAPI (using logical OR and AND operators[103]) to describe REST APIs that support multiple authentication types. In addition, when no authentication or other mechanism is required to access the resources of a Thing, this can be declared both in the TD security definition (*NoSecurityScheme*) and in the OpenAPI security requirements. In OpenAPI, an empty array is used in the security field to declare that there is no security scheme for a particular API; in that case, the API is not protected. This comparison is summarized in Table 5.2.

Security scheme	**W3C TD approach**	**OpenAPI TD approach**
Basic Authentication	*Supported*	*Supported*
Bearer Authentication	*Supported*	*Supported*
API Key Authentication	*Supported*	*Supported*
OAuth2.0	*Supported*	*Supported*
OpenID Connect	*Not supported*	*Supported*
Digest Authentication	*Supported*	*Supported*
PSK Authentication	*Supported*	*Not supported*
Combo	*Supported*	*Supported*
NoSecurity	*Supported*	*Supported*

Table 5.2: Security schemes supported by the OpenAPI approach and the W3C TD approach

[101] https://datatracker.ietf.org/doc/html/rfc4279
[102] https://openid.net/connect/
[103] https://swagger.io/docs/specification/authentication/

5.2.1.4 Hypermedia controls support

OpenAPI is a promising technology towards understanding and constructing Web services that meet the HATEOAS requirement of the REST architectural style. A new feature is introduced in the latest OpenAPI v3.0 (along with *Callbacks*) referred to as *Links.* This is an attempt to incorporate HATEOAS functionality in the specification. In fact, the OpenAPI Specification notes that, although links in OpenAPI are "*somewhat similar to hypermedia*[104]", they do not require the presence of link information in the actual responses. OpenAPI links are not directly related to HATEOAS. In fact, a truly RESTful service sends itself to a client the information on how to send the next requests according to the requested resources. OpenAPI links differ from HATEOAS, as they do not come from the service; they can just be present in the service description. In addition, anyone that describes a service can provide whatever links he/she wants for the service. Therefore, OpenAPI links do not typically meet the HATEOAS requirement of REST. However, if OpenAPI links are defined properly for a service and there is a suitable client that can use them (i.e. it is not mandatory for a client to implement them), then we could claim that OpenAPI links are a naive or initial HATEOAS approach.

In OpenAPI service descriptions, links are defined in the service response section to allow values returned by a service call to be used as input for the next call. More specifically, Links are defined as Link objects in responses (i.e. Response objects) of API operations. Links can be reusable objects in an OpenAPI document so they can be defined in the Components object of the description. Listing 5.8 provides a complete example of a link in an OpenAPI Thing description, similar to the example in the OpenAPI guide[105]. Although the OpenAPI link feature has not been included in the examples of Chapter 4, Listing 5.8 describes how links could be easily supported in an OpenAPI Thing description. Among others, the service document of the described device defines the "Create a subscription" and "Retrieve information about a specific subscription" operations which are proposed by the OpenAPI Web Thing template. The result of "Create a subscription" is used as an input to "Retrieve information about a specific subscription". More specifically, the *subscriptionID* parameter that is returned in the response of the "Create a subscription" operation is used as a path parameter in the operation that retrieves the information of a subscription based on its *subscriptionID*. In this example, the *links* section is defined under the Response Object of the first operation and it describes a link named *GetSubscriptionBySubscriptionID*. However, the link could also be defined under the Components object. The *subscriptionID* parameter and the OpenAPI *operationId* property of the operation that retrieves the subscription are highlighted with bold letters in the Listing.

Listing 5.8: Link example in an OpenAPI Thing description

```
paths:
```

[104] https://smartbear.com/learn/api-design/what-is-hypermedia/
[105] https://swagger.io/docs/specification/links/

```yaml
  /subscriptions:
    post:
      tags:
        - Subscriptions
      summary: Create a subscription
      description: An Extended Web Thing should support subscriptions for the
specific resource (DHT22).
      operationId: createSubscription
      x-operationType: 'https://schema.org/CreateAction'
      requestBody:
        description: Create a new subscription
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/SubscriptionRequestBody'
          required: true
      responses:
        '201':
          description: Created
          content:
            application/json:
              schema:
                type: object
                properties:
                  SubscriptionID:
                    type: string
                    description: ID of the created subscription
          example: 5fd23faccde6be05da68bcfb
          # ------------------------------------------------------
          # Links
          # ------------------------------------------------------
          links:
             GetSubscriptionBySubscriptionID: # <--- arbitrary link name
               operationId: retrieveInfoAboutSubscription
               parameters:
                 SubscriptionID: '$response.body#/SubscriptionID'
                   description: > The `SubscriptionID` value returned in the
response can be used as the `SubscriptionID` parameter in `GET
/subscriptions/{subscriptionID}`.
          # ------------------------------------------------------
        '404':
          description: Not found
  '/subscriptions/{subscriptionID}':
    get:
      tags:
        - Subscriptions
      summary: Retrieve information about a specific subscription
      description: 'In response to an HTTP GET request on a Subscription URL,
an Extended Web Thing must return a JSON representation of the subscription.
The JSON representation should be the same as the one returned for that
subscription in ''Retrieve a list of subscriptions''.'
      operationId: retrieveInfoAboutSubscription
      parameters:
        - name: subscriptionID
          in: path
          description: The id of the specific subscription
          required: true
          style: simple
          explode: true
          schema:
            x-mapsTo: '#/components/schemas/SubscriptionObject.id'
            type: string
            example: 5fd23faccde6be05da68bcfb
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/SubscriptionObject'
        '404':
          description: Not found

components:
  schemas:
    SubscriptionObject:
      allOf:
        - type: object
          required:
            - id
            - type
            - resource
            - description
            - callbackUrl
          properties:
            id:
              type: string
              example: 5fc978fc96cc26a4e202c3d6
        - $ref: '#/components/schemas/SubscriptionRequestBody'
      xml:
        name: SubscriptionObject
```

Therefore, the OpenAPI Thing description approach can be used to represent links contained in HTTP response messages and thus provides a useful mechanism that allows traversing between the operations of Things (e.g. subscription or action operations). The OpenAPI Thing description approach can also describe operations that clients may perform on Things. According to W3C, OpenAPI operations are similar to *forms* as defined in the WoT Architecture. Hence, the OpenAPI approach supports hypermedia controls that refer to Things and their functionality; *links* and *operations* can all be described in an OpenAPI document.

The TD approach proposes a HATEOAS approach for the support of hypermedia controls, as described in Sections 2.3 and 5.1. According to the WoT Architecture, hypermedia controls can be provided in a TD using links and forms. Although a TD describes links, it does not refer to links in the same way as in OpenAPI. That is, links in a TD represent a broader sense than in OpenAPI; it has been described in Section 5.1. Link examples in TDs have already been presented in Section 5.1 (Listings 5.1, 5.2, 5.3 and 5.4). TDs use forms to show clients how to perform specific operations offered by Things (e.g. to retrieve information about a property, to invoke an action, to subscribe to an event, etc). Therefore, forms are somehow similar to operations in OpenAPI, as indicated by W3C. TD form examples have been illustrated in the Properties, Actions and Events objects of the smart door TD in Listing 2.1, in the Properties object of a DHT22 sensor presented in Listing 5.5 (i.e. shows how to retrieve the temperature property and the humidity property of the sensor) and in the Events object of a TD presented in Listing 5.6 (i.e. shows how to subscribe and unsubscribe to the temperature event of a Thing).

Overall, both approaches support hypermedia controls such as links (i.e. in the OpenAPI and the TD approach) and forms (i.e. in the TD approach) which are similar to operations of the OpenAPI approach. Links are used in both approaches, but they represent a different concept in every approach. Moreover, the WoT Architecture proposes Web forms as a hypermedia control to describe the operations that clients may perform on Things; TDs use forms to describe these operations. On the other hand, forms do not exist in the OpenAPI approach; instead, there are service operations which are described by the *Paths* object of OpenAPI.

5.2.1.5 Events and Subscriptions support

TDs can describe events related to Things (e.g. when the humidity value measured by a sensor reaches a specific threshold). They can also describe operations of subscribing and unsubscribing to such events. They do not support any operation for retrieving information about existing subscriptions. Listing 2.1 illustrates how the TD of a smart door actuator defines the event of the door opening using the *Events* object in the description. This *Event* object includes a *forms* field that describes the protocol, the endpoint and the sub-protocol (e.g. the exact mechanism used for asynchronous notifications) for subscribing to the open event of the smart door, so that a client can be notified when the door opens. Moreover, Listing 5.5. illustrates how callbacks can be defined in a TD and uses forms to describe how a client can subscribe and unsubscribe to the temperature event affordance of the Thing.

OpenAPI Thing description approach can also define certain events that trigger callbacks. The approach takes advantage of the OpenAPI specification that supports callbacks in response to certain events. In fact, callbacks are a new feature introduced in the latest OpenAPI v3.0 for defining asynchronous APIs or Webhooks; they are used to define the requests that the described service will send to another service in response to certain events. The OpenAPI Thing description approach emphasizes on describing subscription operations (e.g. the operation for creating a subscription) using Callbacks or Webhooks properties added to a *Paths* object (at the same level as parameters, responses, etc.). In fact, the OpenAPI Web Thing template defines particular operations for creating a subscription (i.e. subscribing) to certain events, for retrieving subscriptions (i.e. all existing subscriptions to a Thing or a specific subscription using its subscription identifier) and also for deleting a specific subscription (i.e. unsubscribing) using its subscription identifier. Therefore, OpenAPI callbacks have not actually been used in the OpenAPI Thing description approach, but they could easily be supported.

More specifically, callbacks in OpenAPI are defined as Callback objects and they are included in Operation objects. The OpenAPI guide provides an indicative example of callbacks in OpenAPI[106]. Similar to links, callbacks can act as reusable objects of an OpenAPI service definition, by being included in the Components object of the document. Although the OpenAPI callback feature has not been included in the examples of Chapter 4, Listing 5.9 describes how callbacks could be simply supported in an OpenAPI Thing description. In this example, a callback is defined for the "Create a subscription" operation of a DHT22 sensor. The request body schema defined under the Components object includes the required *callbackUrl* property. In this example, the *callbacks* section that describes the callback is defined under the Operation Object of the particular operation. However, callbacks can also be defined under the Components object (i.e. they can be reusable).

Listing 5.9: Callback example in an OpenAPI Thing description

```
paths:
  /subscriptions:
    post:
      tags:
        - Subscriptions
      summary: Create a subscription to temperature
      description: A DHT22 sensor should support subscriptions for events
related to temperature value.
      operationId: createSubscription
      x-operationType: 'https://schema.org/CreateAction'
      requestBody:
         description: Create a new subscription
         content:
           application/json:
```

[106] https://swagger.io/docs/specification/callbacks/

```
            schema:
              $ref: '#/components/schemas/SubscriptionRequestBody'
          required: true
      responses:
        …
      callbacks:  # Callback definition
        highTemperatureValueEvent:  # Event name
          '{$request.body#/callbackUrl}':  # The callback URL
            post:
              requestBody:   # Contents of the callback message
                required: true
                content:
                  application/json:
                    schema:
                      type: object
                      properties:
                        message:
                          type: string
                          example: Very high temperature value!
                      required:
                        - message
              responses:   # Expected responses to the callback message
                '200':
                  description: Your server returns this code if it accepts
the callback

components:
  schemas:
    SubscriptionRequestBody:
  type: object
  required:
    - description
    - type
    - callbackUrl
    - resource
  properties:
    name:
      type: string
      example: My subscription for DHT22 temperature
    description:
      type: string
      example: A subscription to get info about DHT22 temperature
    type:
      type: string
      example: webhook (callback)
    callbackUrl:  # Callback URL
      type: string
      format: uri
      example: 'http://172.16.1.5:5000/accumulate'
    resource:
      type: object
      properties:
        type:
          type: string
          example: property
        name:
          type: string
          example: temperature
    expires:
      type: string
      format: date-time
    throttling:
      type: integer
      format: int32
      example: 5
  xml:
    name: SubscriptionRequestBody
```

Therefore, both approaches actually represent events and operations for subscribing and unsubscribing to specific events (e.g. when receiving a new pressure measurement). Both approaches can also describe callbacks that result from events related to Things (e.g. a new temperature value measured from a sensor). However, only the OpenAPI Thing description approach provides operations that retrieve information about existing subscriptions.

5.2.1.6 Data schemas support

Both approaches describe the JSON data schemas used for the representation of Things and their interactions. They can both define input and output data types (e.g. the input or output data schemas for actions). The TD approach describes data schemas using the *"data"* object in a JSON TD in conjunction with the Data Schema Vocabulary (Section 2.11.4). Therefore, the TD approach can describe the input data schema of an action, the output data schema of an event, the input data schema of a subscription, etc. For example, the *"data"* member in the TD of Listing 5.6 is used to describe the input data schema of the request sent to the callback URL in response to a temperature event. The data schema includes the temperature value of the Thing (i.e. a number). The value of the *type* attribute in the data member is *"number"*. This data schema type is defined by the *NumberSchema* class in the Data Schema Vocabulary of W3C. On the other hand, the

OpenAPI approach uses the *Schema* object to describe data schemas for Things and their affordances. For example, it can define the output data schema of a Web Thing, the input data schema of a subscription, the input data schema of an action, the output data schema of all properties, etc. The Schema objects are defined under the Components object of an OpenAPI Thing description. Indicative examples of schemas in OpenAPI Thing descriptions have been illustrated in Listings 4.1, 4.4, 4.5 and 4.6.

5.2.1.7 Semantic extensions

Both TD and OpenAPI make services offered by Things machine-understandable and discoverable by Semantic Web tools. In the OpenAPI approach, the respective descriptions are translated to ontologies, while TDs are very close to ontologies because of their representation format. In essence, TD adopts JSON-LD format which is more powerful than JSON or YAML and it is equivalent to an ontology. So in contrast to the OpenAPI TD approach, where Thing descriptions are translated to ontologies, W3C TDs are directly represented in JSON-LD format. TDs are not converted to ontologies like in the OpenAPI TD approach. In other words, the ontologies and vocabularies described in W3C specifications (e.g. TD Ontology, WoT Security Ontology, etc), do not result from the translation of JSON TDs. Instead, they have been proposed by W3C to support the TD information model by further describing Things and their related concepts such as Things interactions and API security schemes. So the ontologies described by W3C are rather complementary in the process of describing Things and their functions. More specifically, the context extension mechanism of JSON-LD is used to extend TD documents with concepts from the semantic models of W3C. However, the proposed vocabularies and ontologies may also act as alternatives to the JSON representation format used for TDs, as highlighted in the TD Ontology specification.

In contrast to the W3C TD approach that benefits from JSON-LD format, the OpenAPI TD approach is limited to plain JSON representation format. In fact, the OpenAPI TD approach provides semantic extensions using the extension properties (i.e. x-properties) of [17]. Therefore, both approaches utilize semantic annotations to map specific properties (e.g. the concept of action or actuator) to semantic models. However, the serialization format of TD documents (i.e. JSON-LD) is certainly more powerful than plain JSON format, as it serializes Linked Data. JSON-LD allows the representation of knowledge about Things in a machine-understandable way. OpenAPI Thing descriptions may provide semantic annotations, but they have to be translated to an ontology so that semantics can be leveraged by machines. Then, the representation of a Thing and its functions can benefit from Semantic Web tools (e.g. reasoners, query languages) enabling service discovery and service orchestration. Otherwise, the semantic annotations of OpenAPI descriptions can only be useful to humans.

5.2.1.8 General aspects

As highlighted above, TD is a much shorter description than an OpenAPI document, which might be helpful in some cases (e.g. in terms of memory). An OpenAPI service description defines all service request and response bodies (schemas), operations, HTTP status codes, headers, parameters (e.g. path parameters), security requirements, non-functional information, etc. OpenAPI also defines services in a way that eliminates ambiguities and provides Web Thing service descriptions that are uniquely defined and discoverable, (i.e. using semantic annotations). Documentation generation tools can use an OpenAPI description to display the API, code generation tools to generate servers and clients in various programming languages, testing tools, etc. On the contrary, the TD approach does not provide so many tools; to the best of our knowledge, there are no documentation generation tools and code generation tools for the TD approach.

Both approaches include the names of the Thing interaction affordances (e.g. properties, actions) and describe the functionality (service) of the Thing. They do not provide the values of these interactions though. A user can only find out which properties and actions the Thing supports since they are indirectly indicated in the descriptions (in the form of paths, schemas, operations, etc). The values of these properties (e.g. a humidity measurement value) and actions (e.g. the status of a lock action execution) are provided by the corresponding service.

Overall, OpenAPI is an alternative to the TD of the W3C Architecture in terms of describing an IoT device and especially its exposed Web API in the form of a JSON (or YAML) description. Both approaches are capable of describing Things and their affordances. However, we consider that TD is a more abstract description. OpenAPI is detailed and complete; it fully describes the functionality of a device and provides all the information a client needs to use the services it provides (e.g. paths, parameters, response and request bodies, headers and even example values) and not just interact with the device, as TD does. In addition, OpenAPI provides a wide variety of tools compared to the TD approach, including documentation generation tools, code generation tools and developer (or testing) tools. Finally, we should note that OpenAPI is a widely adopted industry standard endorsed by Linux Foundation and supported by large software vendors, whereas the TD is (still) a W3C recommendation.

5.2.1.9 Comparison conclusions

The results of the above comparison are summarized in Table 5.3.

Comparison criteria	**W3C TD approach**	**OpenAPI TD approach**
Document specificity	Much shorter, abstract. JSON-LD format, more powerful.	Larger, detailed, complete for service description. JSON/YAML format.
Protocol Support	HTTP(S), CoAP(S), MQTT and other protocols. Not limited.	Only HTTP(S). Limited.
Security Support	Basic Authentication, Bearer Authentication, API Key Authentication, OAuth 2.0, Digest Authentication, PSK Authentication, Combo, No Security.	Basic Authentication, Bearer Authentication, API Key Authentication, OAuth 2.0, Digest Authentication, OpenID Connect, Combo, No Security.
Hypermedia Controls Support	Links and Forms (WoT Architecture)	OpenAPI Links, operations instead of forms
Data Schemas Support	Uses the Data Schema Vocabulary.	Uses OpenAPI Schema Objects.
Semantic Extensions Support	Uses the @context mechanism of JSON-LD. Not necessary to translate TDs to ontologies.	Semantic annotations using x-properties. Descriptions have to be translated to ontologies.
General Aspects	Short, abstract. No documentation - code generation tools. Still a W3C recommendation.	Detailed, complete. Documentation - code generation tools provided. Industry standard.

Table 5.3: Comparison results based on the JSON descriptions of the two approaches

5.2.2 Comparison based on ontologies

In general, there is no direct correlation between the OpenAPI ontology and the ontologies used by the Thing Description approach. Although the ontologies may describe similar concepts related to services, this is done in a completely different way and there are not many direct similarities. In fact, the OpenAPI TD approach proposes the OpenAPI ontology which is used to capture all information of an OpenAPI description. Therefore, the ontology itself does not support concepts related to the Web of Things such as Thing, properties, actions, WoT operation types, etc. In fact, the OpenAPI ontology is based on the Hydra core vocabulary, as described in Section 2.11.8. However, the OpenAPI Thing description approach benefits from semantic annotations proposed by the Semantic OpenAPI to describe WoT concepts along with other concepts such as id, name, description, etc. It maps these concepts to terms in the SSN ontology, in the SOSA ontology and also in schema.org. The RDF triples in the translated OpenAPI ontology (i.e. resulting from a Thing description) may include URIs from these semantic models. Therefore, these concepts are also described in the resulting ontology. On the contrary, the TD approach proposes: a) an ontology to describe WoT concepts (i.e. the TD ontology described in Section 2.11.2), b) an ontology to describe API security for Things (i.e. the WoT Security Ontology described in Section 2.11.3), c) an ontology to describe data schemas used to represent Things (i.e. the Data Schema Vocabulary described in Section 2.11.4), and d) an ontology to describe hypermedia controls such as links (i.e. the Hypermedia Controls Ontology described in Section 2.11.5).

The comparison of the two approaches is based on specific criteria such as the WoT concepts support (Section 5.2.2.1), the security support (Section 5.2.2.2), the hypermedia controls support (Section 5.2.2.3), the events and subscriptions support (Section 5.2.2.4), the protocol support (Section 5.2.2.5) and the data schema support (Section 5.2.2.6). The comparison results and conclusions are summarized in Section 5.2.2.7.

5.2.2.1 WoT concepts support

The TD approach of W3C proposes the W3C TD Ontology (Section 2.11.2) for defining WoT concepts which are proposed by the WoT Architecture such as Thing, interaction affordance, property affordance, action affordance, operation type. Thus, the ontology defines classes to represent a Thing, a property affordance, an event affordance, etc. It also defines specific object properties, datatype properties and named individuals, as described in Section 2.11.2. This ontology also presents some alignments with the SOSA ontology and the schema.org vocabulary (see Section 2.11.2). On the other hand, the OpenAPI ontology does not represent WoT concepts (e.g. Thing, properties, actions), as it was meant to describe API terms, according to the OpenAPI specification. However, the OpenAPI Thing description approach benefits from semantic annotations (i.e. in OpenAPI Thing descriptions) to semantic models such as the SOSA ontology and the schema.org vocabulary that describe this kind of concepts (e.g. see Listing 4.1).

The *Thing* class (W3C TD ontology) and the *Document* class (OpenAPI ontology) are top-level concepts in each ontology. Below these classes, other concepts are defined to describe a Web Thing in the TD ontology and an API in the OpenAPI ontology. The Thing Description specification of W3C defines some properties (i.e. vocabulary terms)[107] of the *Thing* class such as *@context*, *@type*, *id*, *title*, *properties*, *actions* and *events* which are also used in a TD to describe a Thing. The *support* property provides contact information for the TD maintainer just like the *openApi:contact* class of the OpenAPI ontology. The *version* property is the same with the *openApi:version* property (i.e. both properties describe the version of the corresponding description, either W3C TD or OpenAPI TD). The *links* property is similar to the *openApi:ExternalDoc* class that provides links for external documentation. The *securityDefinitions* property is similar to the *openApi:security* property, as it contains security definitions used in the W3C TD just like the OpenAPI ontology for the security definitions of an API. However, there is also a *security* property and there is no corresponding property in the OpenAPI ontology. We should note that in the OpenAPI ontology security requirements only exist in an Operation. In fact, when translating an OpenAPI description to an ontology, security requirements are clarified.

InteractionAffordance is an abstract concept proposed by the WoT Architecture to describe how a client can interact with Things. It is divided into *PropertyAffordances*, *ActionAffordances* and *EventAffordances*. We cannot directly relate these concepts to any concepts of the OpenAPI ontology, as they are not compliant with the logic of the OpenAPI Specification in general. In OpenAPI, everything is described as an Operation where the client retrieves information or performs some action or creates a subscription to a resource. On the contrary, in TDs, these operations are classified according to their types. More specifically, in case a client retrieves Thing property information, it is a *PropertyAffordance*. In case a client performs an action (e.g. turns on a lamp or moves a camera device), this is classified as *ActionAffordance*. Moreover, *EventAffordance* describes events related to Things and the mechanisms used to transmit asynchronous notifications from Things to consumers (i.e. clients). In OpenAPI 3.1, this is described using callbacks or webhooks. However, the description of callbacks and webhooks has not been yet implemented in the OpenAPI ontology.

5.2.2.2 Security support

Both approaches are able to describe the security mechanisms used for authentication and authorization of clients that interact with Things. The TD approach of W3C proposes the WoT Security Ontology for defining security schemes such as the API Key security scheme and the Basic Authentication security scheme, as described in Section 2.11.3. This ontology defines classes that represent security schemes and specific object properties (e.g. *allOf*, *authorization*, *OneOf*) and datatype properties (e.g. *flow*, *scopes*). Similarly, the OpenAPI Thing description approach benefits from the OpenAPI ontology and the *Security* class, in particular, to describe the API security definitions used in

[107] https://www.w3.org/TR/wot-thing-description11/#thing

OpenAPI Thing descriptions. The Security class of the ontology, that includes security schemes as subclasses, is already described in Section 2.11.8 and illustrated in Figure 2.8. Similar to the JSON description of the two approaches, the ontologies describe the same security schemes, except from the PSK security scheme that is only described in the WoT Security Ontology and the OpenID Connect security scheme that is only described in the OpenAPI ontology.

5.2.2.3 Hypermedia controls support

The TD approach of W3C proposes the Hypermedia Controls Ontology for defining hypermedia controls, as defined by the WoT Architecture; Web links and Web forms. This ontology defines classes to represent a link, a form, an expected response and an additional expected response, and specific object properties (e.g. *hasTarget*) and datatype properties (e.g. *forSubProtocol*), as described in Section 2.11.5. This ontology also presents some alignments with the Hydra core vocabulary (see Section 2.11.5).

On the other hand, the OpenAPI ontology does not (yet) represent hypermedia concepts as described in OpenAPI (i.e. links). In fact, there is no direct correlation of links described in the Hypermedia Controls Ontology with links described in the OpenAPI approach. Concerning forms, the OpenAPI ontology is based on Hydra and it represents service operations, which, according to the Hypermedia Controls Ontology specification, are "*similar in spirit*" to TD forms. Forms represent how the client can perform specific requests to perform with Things. The class *Form* of the TD ontology could be related to the *openApi:Operation* class of the OpenAPI ontology, but this would not be absolutely correct (i.e. the two concepts are not equivalent). Finally, links (along with callbacks) might be represented in the ontology in the near future; it is a work in progress by the authors of the OpenAPI ontology.

5.2.2.4 Events and subscriptions support

The TD approach of W3C proposes the TD ontology (Section 2.11.2) for describing EventAffordances and subscriptions to events. These are described using webhooks and callbacks in OpenAPI. However, there is not (yet) support for webhooks and callbacks in the OpenAPI ontology, as highlighted in Section 5.2.2.1. Although the concept of subscriptions is not defined in the OpenAPI ontology, subscription operations (i.e. to create, retrieve or delete subscriptions), are supported in OpenAPI Thing descriptions using the *Operation* object and they can be supported in the OpenAPI ontology as well using the *Operation* class.

5.2.2.5 Protocol support

Regarding protocols, we should note that the OpenAPI ontology can only describe APIs implemented using the HTTP(S) protocol, while the TD approach is meant to support additional protocols. W3C notes that the number of Protocol Bindings that can be implemented by a Thing is not restricted. Other protocol bindings (e.g. for CoAP, MQTT or OPC UA) are intended to be standardized in separate documents such as a protocol vocabulary (see Section 5.1).

5.2.2.6 Data schemas support

Both approaches use ontologies to describe the data schemas used for the representation of Things and their interactions (e.g. the input and output data schemas for actions). The TD approach proposes an RDF vocabulary (i.e. the Data Schema Vocabulary described in Section 2.11.4) for JSON data schema definitions. For the OpenAPI ontology, the corresponding mechanism is SHACL (Shapes Constraint Language) which is also a recommendation and is incorporated in the OpenAPI ontology. However, these mechanisms are totally different from each other and there is no correlation between them.

5.2.2.7 Comparison conclusions

The results of the above comparison are summarized in Table 5.4.

Comparison criteria	**W3C TD approach**	**OpenAPI TD approach**
WoT Concepts Support	Uses the TD Ontology (different from the OpenAPI ontology that describes API concepts).	The OpenAPI ontology does not define them. Semantic annotations to SOSA, SSN, schema.org are used.
Protocol Support	HTTP(S), CoAP(S), MQTT and other protocols. Not limited.	The OpenAPI ontology describes only HTTP(S). Limited.
Security Support	Uses the WoT Security Ontology to define: Basic Authentication, Bearer Authentication, API Key Authentication, OAuth 2.0, Digest Authentication, PSK	Uses the OpenAPI Ontology to define: Basic Authentication, Bearer Authentication, API Key Authentication, OAuth 2.0, Digest Authentication, OpenID

	Authentication, Combo, No Security.	Connect, Combo, No Security.
Hypermedia Controls Support	Links and Forms (WoT Architecture).	Links are not yet described in the OpenAPI Ontology.
Events - Subscriptions Support	The TD Ontology describes EventAffordances and subscriptions to events.	The concepts of events, subscriptions and OpenAPI callbacks are not defined in the OpenAPI Ontology. Subscription operations to events are described using the *Operation* class.
Data Schemas Support	Uses the Data Schema Vocabulary.	The OpenAPI Ontology uses SHACL.

Table 5.4: Comparison results based on the ontologies of the two approaches

Overall, there is no direct correlation between the ontologies of the two approaches. However, both approaches can describe the most important concepts related to Things using the provided ontologies and semantic annotations to external ontologies or vocabularies. Therefore, the OpenAPI Thing description approach, which uses the OpenAPI ontology and external semantic models to describe Things and their interactions, can be an alternative to the TD approach.

6

WoT Architecture implementations

6.1 Comparing WoT implementations

In this section, we review the WoT implementations presented in Chapter 3 according to the requirements of the WoT Architecture of W3C. The three candidate WoT implementations (namely, *Thingweb.node-wot, Webofthings.js and WTMs*) are compared based on their capacity to support the requirements of the recommendation.

The WoT Architecture recommendation presents some common principles for WoT architectures such as the interconnection of different eco-systems using Web standards, the adoption of RESTful APIs for the interaction with Things, the interoperability of architectures, the scalability, the compatibility, etc. Moreover, it states that the four building blocks[108] for W3C WoT are: a) the *WoT Thing Description* that is used to describe Things and their interactions, b) the *WoT Binding Templates* for IoT protocols, c) the *WoT Scripting API* (i.e. *optional* building block), and d) *WoT Security and Privacy guidelines*. In fact, the recommendation notes that the WoT TD is the central building block of W3C WoT. According to the WoT Architecture, "*the description metadata MUST be a WoT Thing Description (TD)*" in W3C WoT. In addition, it is noted in the specification that *at least one* TD representation must be available for a specific device to be a Thing. Therefore, WoT implementations that follow the WoT Architecture have to use TDs for the interaction with Things. The Interaction Model proposed in the specification (i.e. properties, actions and events) is also an important aspect of the WoT Architecture. Last but not least, the well-known operation types for WoT and the hypermedia controls (i.e. Web links and Web forms) that show clients how to interact with Things are also basic features of the W3C WoT.

[108] https://www.w3.org/TR/wot-architecture/#sec-building-blocks

Taking into account all these aspects, we can determine which of the candidate implementations are fully or more compliant with the recommendation of W3C. Although all implementations follow some requirements of the WoT Architecture, not all of them are fully compatible with the specification. In fact, only *Thingweb node-wot* is fully compliant with the specification, as it is the only implementation that adopts W3C Thing Descriptions to allow interaction with Things. In addition, only Thingweb supports all WoT operation types defined in the WoT Architecture (see Table 2.1). It also implements several protocol bindings and it can apply WoT security mechanisms for the authentication of clients (e.g. Basic authentication, Bearer Tokens). Furthermore, Thingweb implements the WoT Scripting API (i.e. an optional building block of W3C WoT), it may support hypermedia controls such as links and forms and it naturally follows the interaction model of the WoT Architecture. Thingweb also implements additional protocols such as CoAP, CoAPS and MQTT (see Section 3.1).

WTMs and Webofthings.js were originally based on the Web Thing Model submission of W3C, as highlighted in Chapter 3. Therefore, they do not use the WoT Thing Description (TD) format proposed by W3C, which is, according to the WoT Architecture, "*the central building block of W3C WoT*". However, these implementations also allow clients to interact with Things using RESTful APIs, they support the interaction affordance model of the recommendation (e.g. properties, actions, events) and hence they follow basic requirements of the recommendation. In addition, they may apply WoT security mechanisms (e.g. Basic Authentication) and they may support hypermedia controls such as Web links. They only implement some operation types of the WoT Architecture, but they could be extended to support more operation types. Finally, in relation to protocol support, WTMs only supports HTTP and the Webhooks mechanism, while Webofthings.js only supports HTTP(S) and the Websocket mechanism. The results of the comparison are summarized in Table 6.1.

WoT Architecture feature	**Thingweb node-wot**	**Web Thing Model service (WTMs)**	**Webofthings.js**
W3C Thing Description (TD)	*Supported*	*Not supported*	*Not supported*
Binding templates	*HTTP, HTTPS, CoAP, MQTT and other are supported*	*Only HTTP and Webhooks supported*	*Only HTTP, HTTPS and Websockets supported*
WoT Scripting API *(optional)*	*Supported*	*Not supported*	*Not supported*
WoT Security mechanisms	*Supported*	*Supported*	*Supported*
Interaction Model	*Supported*	*Supported*	*Supported*
Hypermedia controls	*Supported*	*Supported*	*Supported*
Operations types	*Supported*	*Some of them supported*	*Some of them supported*

Table 6.1: Comparison of reference implementations based on W3C Architecture requirements

Table 6.2 summarizes the results of the comparison of the implementations based on the operation types they support. In contrast to WTMs and Webofthings.js that only implement some of the WoT operation types, Thingweb node-wot supports all operations proposed in the WoT Architecture.

WoT operation type	**Thingweb node-wot**	**Web Thing Model service (WTMs)**	**Webofthings.js**
readproperty	*Supported*	*Supported*	*Supported*
writeproperty	*Supported*	*Supported*	*Supported*
observeproperty	*Supported*	*Not supported*	*Not supported*
unobserveproperty	*Supported*	*Not supported*	*Not supported*
invokeaction	*Supported*	*Supported*	*Supported*
subscribeevent	*Supported*	*Supported*	*Supported*
unsubscribeevent	*Supported*	*Supported*	*Not supported*
readallproperties	*Supported*	*Supported*	*Supported*
writeallproperties	*Supported*	*Not supported*	*Not supported*
readmultipleproperties	*Supported*	*Not supported*	*Not supported*
writemultipleproperties	*Supported*	*Supported*	*Not supported*

Table 6.2: Comparison of reference implementations based on the W3C Architecture operation types

Therefore, we conclude that only Thingweb node-wot could be considered a reference implementation of the WoT Architecture, while WTMs and Webofthings.js are not fully compliant with the recommendation, mainly because they do not use W3C Thing Descriptions.

7

Conclusions and Future Work

The Web of Things concept is gradually becoming popular, as it aims to unify the world of interconnected devices, by utilizing existing and common Web technologies. In the context of WoT Architecture (i.e. W3C recommendation) that defines an abstract architecture for the Web of Things, we aimed to discover and review existing approaches that intend to describe Web Things (e.g. humidity or precipitation sensors, smart door or window actuators, smart TVs, cameras, etc) and their functionality as Web services. The devices and the operations they may support can be exposed on the Web using REST and thus be considered as RESTful Web services. However, as the number of devices and Cloud services is constantly increasing, the need for efficient and accurate service discovery (i.e. especially for Things and their operations) has become a significant challenge. This is mainly due to the lack of formal service descriptions, as most Cloud providers describe their offerings in plain text. Therefore, the purpose of our work was to propose a description language for devices exposed as Cloud services, that both humans and machines could understand. This description language would allow the implementation of various tools, such as service discovery and service orchestration mechanisms. Devices and their functionality can then be human and machine understandable and also discoverable on the Web. Alongside, Thing Descriptions (TDs), which adopt the JSON format, are proposed by the WoT Architecture to describe Things and allow the interaction of clients with them.

7.1 Conclusions

During our research, we initially reviewed approaches that would efficiently describe any aspect of a service, both syntactically and semantically. However, we were mainly focused on approaches for describing RESTful services, as the majority of Cloud services are provided as Web services based on the REST architectural style. Unlike SOAP-based services that use the standard WSDL, there are many approaches for the description of

RESTful services (e.g. OpenAPI, RAML). In relation to existing approaches for the description of Things as RESTful services, we focused on analyzing the Thing Description (TD) of W3C Architecture, as it provides a JSON representation for Things and supports the interaction with Things in the Web using REST.

We also proposed and used the OpenAPI Specification, which is an industry standard, for the description of devices as Web services. The selection of OpenAPI was motivated by the popularity of the specification, its powerful tool support, and the active community. In addition, there was a series of events that also affected our decision. At the end of 2015, the OpenAPI Initiative was announced, founded by large organizations such as Google, Microsoft and IBM, in order to extend the OpenAPI Specification (formerly known as Swagger) and standardize a description mechanism for RESTful services. Openstack, in mid-2016, announced the adoption of the OpenAPI specification for the description of its offered services. Oracle, was the first organization that released an API catalogue of its offering Cloud services described in OpenAPI, while Microsoft preserves a Github repository[109] that includes the OpenAPI descriptions of Azure's Cloud services.

As demonstrated in this work, the OpenAPI Specification offers a human-friendly environment for discovering and using RESTful services. However, despite being machine-readable, the specification is not machine-understandable, thus limiting the availability of tools that facilitate machine tasks such as service discovery. The Semantic OpenAPI Specification [17] attempted to fill this gap, allowing the description of RESTful services both semantically and syntactically. Using the extension mechanism that OpenAPI offers, Semantic OpenAPI defines some additional properties that semantically enrich various parts of an OpenAPI service description, resolving any ambiguities that may exist in service descriptions and allowing machines to better understand the described services. In addition, the authors of Semantic OpenAPI developed an ontology allowing any OpenAPI service description to be transformed in, enabling the use of Semantic Web tools such as reasoners and query languages. Our proposed solution adopts the Semantic OpenAPI approach for the description of devices and their functions as RESTFul Web services. Therefore, Things can be described without ambiguities and be discoverable on the Web by leveraging Semantic Web query languages and reasoners.

The adoption of the whole OpenAPI Specification ecosystem in conjunction with our proposal can be substantially beneficial for both Cloud providers and individual users involved with IoT devices. In this work, we demonstrated several advantages of OpenAPI descriptions (enriched with semantic annotations) that facilitate the unambiguous description of Things. We also introduced a particular template (OpenAPI Web Thing template), which is a specialization of OpenAPI and can be applied to any real-world device. Web Things can be fully described using OpenAPI similar to the way RESTful services are described. Apart from the OpenAPI template, a mechanism for generating OpenAPI Thing descriptions from input given by a user or a service is proposed and it is available on Github for testing.

Our approach does not adopt the TDs of W3C to allow the interaction with Things. However, it is compliant with the common principles of the WoT Architecture such as the interaction of clients with Things on the Web using RESTful APIs and the support of the interaction affordance model of W3C (i.e properties, actions, events and also

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

subscriptions to events and navigation links). We compared our solution to the Thing Description (TD) of W3C Architecture in terms of completeness of JSON descriptions (i.e. TD and OpenAPI descriptions) as well as completeness of provided ontologies (i.e. OpenAPI ontology and the ontologies proposed by W3C). We showed that the approaches are similar and that the OpenAPI Thing Description approach can be an alternative to the W3C TD approach. It provides all the information that a client (i.e. user or service) needs in order to use the service that exposes the Thing's functionality and not just interact with it (as TD does).

We also discussed the Web Thing Model specification of W3C, which was introduced by W3C before the WoT Architecture. It attempted to set the requirements for the Web of Things and suggested a particular REST API for simply interacting with Things. This model laid the foundations for our implementation of a Thing description approach for the WoT. In fact, it led to the formation of the OpenAPI Web Thing template and, more specifically, of the REST API we used to describe the operations of Things.

In addition, Web Thing Model service (WTMs) was our proposed implementation of the Web Thing Model. We designed and realized a Web proxy - an actual WoT architecture - that implements a directory (e.g. a database) with all Things. Therefore, Things become part of the Web and can be accessed via their Web Proxy; they are discoverable. This Web Proxy follows the basic requirements of the WoT Architecture recommendation such as interoperability, flexibility and scalability. Compared to existing WoT implementations, WTMs is complete (i.e. implements all Web Thing Model model operations) while being more flexible in certain cases.

The Web Thing Model, however, is earlier than the WoT Architecture and it is not a recommendation. Therefore, we needed to review and compare the WoT implementations again, based on the requirements of the WoT Architecture. The comparison results showed that WTMs is not fully compatible with the WoT Architecture, as it does not adopt the TDs of W3C to represent Things. Instead, the Thingweb node-wot implementation (i.e. a reference implementation of the WoT Scripting API), which is based on TDs, is fully compliant with the WoT Architecture specification according to our review.

7.2 Future Work

In relation to future work, there are many issues worth considering further, but it will focus on Things discovery and composition. The work will resort to semantic descriptions of Things (i.e. ontologies) which can be derived from OpenAPI. A query language in the spirit of SOWL-QL [43] will be designed so that the user need not be familiar with the specifics of the Things representation. Regarding WTMs, HTTPS protocol will eventually replace HTTP as a secure solution for the transmission of confidential information. Moreover, incorporating trust evaluation mechanisms for dealing with IoT risks due to malicious behaviour of IoT [44] would be an important add-on to the WTMs implementation. Furthermore, the AsyncAPI framework could be used to solve the gap of OpenAPI in IoT protocols and asynchronous communication use cases of devices. AsyncAPI seems a powerful solution for message-centric API interaction which is usually found in IoT and other similar platforms. In fact, AsyncAPI is able to describe IoT data protocols (e.g. MQTT, CoAP) used for IoT devices; such protocols are not supported in OpenAPI. Finally, the OpenAPI ontology could be extended in order to support OpenAPI links and callbacks. As long as links are described, the ontology will be capable of describing hypermedia concepts included in Thing descriptions. Hydra is a detailed vocabulary that can contribute to the achievement of this goal. Response messages from servers should contain the essential information that a client (i.e. user or service) may use in order to discover all the available resources and actions of a Thing, so as to construct new HTTP requests to achieve a specific goal related to the Thing. Therefore, by providing hypermedia Web APIs for devices and by leveraging Hydra, smarter clients will be created for the interaction with Things.

Appendix

1. OpenAPI Thing description for a DHT22 sensor device in YAML format

```
openapi: 3.0.3
info:
  title: DHT22 sensor OpenAPI Thing description
  description: An OpenAPI Thing description for a DHT22 sensor device that
exposes its current temperature and humidity.
  The OpenAPI Web Thing template and thus the REST API of the device is
inspired by the Web Thing Model submission of W3C.
  Find out more about Web Thing Model (W3C) at
[https://www.w3.org/Submission/wot-model/](https://www.w3.org/Submission/wot-
model/).
  contact:
    email: atzavaras@isc.tuc.gr
  license:
    name: An example license
    url: 'http://www.example.com/licenses/LICENSE-2.0.html'
  version: 1.0.0
externalDocs:
  description: Find out more about the OpenAPI Thing Description approach
here.
  url:
'https://www.intelligence.tuc.gr/~petrakis/publications/OpenAPIWoT.pdf'
servers:
  - url: 'http://localhost:5000/DHT22'
    description: An example server for the Web service exposed for the
device.
tags:
  - name: Web Thing
```

```
    x-onResource: '''#/components/schemas/Webthing'''
    description: Operations on a Web Thing
    externalDocs:
      description: Find out more
      url: 'https://www.w3.org/Submission/wot-model/#web-thing-resource'
  - name: Properties
    description: Operations on Thing Properties
    externalDocs:
      description: Find out more about Thing properties
      url: 'https://www.w3.org/Submission/wot-model/#properties-resource'
  - name: Subscriptions
    x-onResource: '''#/components/schemas/SubscriptionObject'''
    description: Operations on subscriptions
    externalDocs:
      description: Find out more about Thing subscriptions
      url: 'https://www.w3.org/Submission/wot-model/#subscriptions-resource'
paths:
  /:
    get:
      tags:
        - Web Thing
      summary: Retrieve Web Thing
      description: 'In response to an HTTP GET request on the root URL of a
Thing, an Extended Web Thing must return an object that holds its
representation.'
      operationId: retrieveWebThing
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Webthing'
        '404':
          description: Not found
  /properties:
    get:
      tags:
        - Properties
      summary: Retrieve a list of properties
      description: 'In response to an HTTP GET request on the destination URL
of a properties link, an Extended Web Thing must return an array of Property
that the initial resource contains.'
      operationId: retrieveWebThingProperties
      responses:
```

```yaml
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/PropertiesResponse'
        '404':
          description: Not found
  /properties/temperature:
    get:
      tags:
        - Properties
      summary: Retrieve the value of a property (temperature)
      description: 'In response to an HTTP GET request on a Property URL, an
Extended Web Thing must return an array that lists recent values of that
Property.'
      operationId: retrieveTempProperty
      parameters:
        - $ref: '#/components/parameters/pageParam'
        - name: perPage
          description: Pagination second (per page) parameter
          in: query
          required: false
          schema:
            type: integer
      responses:
        '200':
          description: successful operation
          headers:
            Result-Count:
              schema:
                type: integer
                example: 562
              description: The Result-Count header contains the total number
of results.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/TempMeasurement'
        '404':
          description: Not found
  /properties/humidity:
    get:
      tags:
        - Properties
      summary: Retrieve the value of a property (humidity)
      description: 'In response to an HTTP GET request on a Property URL, an
Extended Web Thing must return an array that lists recent values of that
Property.'
      operationId: retrieveHumidProperty
      parameters:
        - $ref: '#/components/parameters/pageParam'
        - name: perPage
          description: Pagination second (per page) parameter
          in: query
          required: false
          schema:
            type: integer
      responses:
        '200':
          description: successful operation
          headers:
            Result-Count:
              schema:
                type: integer
                example: 562
              description: The Result-Count header contains the total number
of results.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/HMeasurement'
        '404':
          description: Not found
  /subscriptions:
    get:
      tags:
        - Subscriptions
      summary: Retrieve a list of subscriptions
      description: 'In response to an HTTP GET request on the destination URL
of a subscriptions link, an Extended Web Thing must return the array of
subscriptions to the underlying resource.'
      operationId: retrieveListOfSubscriptions
      parameters:
        - $ref: '#/components/parameters/pageParam'
        - name: perPage
          description: Pagination second (per page) parameter
          in: query
          required: false
          schema:
            type: integer
      responses:
        '200':
          description: OK
          headers:
            Result-Count:
              schema:
                type: integer
                example: 562
              description: The Result-Count header contains the total number
of results.
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/SubscriptionObject'
        '404':
          description: Not found
    post:
      tags:
        - Subscriptions
      summary: Create a subscription
      description: An Extended Web Thing should support subscriptions.
      operationId: createSubscription
      x-operationType: 'https://schema.org/CreateAction'
      requestBody:
        description: Create a new subscription
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/SubscriptionRequestBody'
        required: true
      responses:
        '200':
          description: OK
        '404':
          description: Not found
  '/subscriptions/{subscriptionID}':
    get:
      tags:
        - Subscriptions
      summary: Retrieve information about a specific subscription
      description: 'In response to an HTTP GET request on a Subscription URL,
an Extended Web Thing must return a JSON representation of the subscription.
The JSON representation should be the same as the one returned for that
subscription in ''Retrieve a list of subscriptions''.'
      operationId: retreiveInfoAboutSubscription
      parameters:
        - name: subscriptionID
          in: path
          description: The id of the specific subscription
          required: true
          style: simple
          explode: true
          schema:
            x-mapsTo: '#/components/schemas/SubscriptionObject.id'
            type: string
            example: 5fd23faccde6be05da68bcfb
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/SubscriptionObject'
        '404':
          description: Not found
    delete:
      tags:
        - Subscriptions
      summary: Delete a subscription
      description: In response to an HTTP DELETE request on the destination
URL of a subscriptions an Extended Web Thing must either reject  the request
with an appropriate status code or remove (unsubscribe) the subscription and
return a 200 OK status code.
      operationId: deleteSubscription
      x-operationType: 'https://schema.org/DeleteAction'
      parameters:
        - name: subscriptionID
          in: path
          description: The id of the specific subscription
          required: true
          style: simple
          explode: true
          schema:
            x-mapsTo: '#/components/schemas/SubscriptionObject.id'
            type: string
            example: 5fd23faccde6be05da68bcfb
      responses:
        '200':
          description: OK
        '404':
          description: Not found
components:
  schemas:
    Webthing:
      required:
        - id
        - name
      type: object
      x-refersTo: 'http://www.w3.org/ns/sosa/Sensor'
      properties:
        id:
          type: string
          default: DHT22
          x-kindOf: 'http://schema.org/identifier'
        name:
          type: string
          example: DHT22/AM2302
          x-kindOf: 'http://schema.org/name'
        description:
          type: string
          example: 'The DHT-22, also named as AM2302, is a digital-output
relative humidity and temperature sensor. It uses a capacitive humidity
sensor and a thermistor to measure the surrounding air, and spits out a
digital signal on the data pin.'
          x-refersTo: 'http://schema.org/description'
        createdAt:
          type: string
          format: date-time
        updatedAt:
          type: string
          format: date-time
        tags:
          type: array
          items:
            type: string
            example: temperature sensor
      xml:
        name: Webthing
    PropertiesResponse:
      anyOf:
        - $ref: '#/components/schemas/TempProperty'
        - $ref: '#/components/schemas/HumProperty'
      xml:
        name: PropertiesResponse
    TempMeasurement:
      type: object
      x-kindOf: 'http://www.w3.org/ns/sosa/Observation'
      properties:
        temp:
          type: integer
          example: 25
          x-kindOf: 'http://www.w3.org/ns/sosa/hasSimpleResult'
        timestamp:
          type: string
          format: date-time
          x-kindOf: 'http://www.w3.org/ns/sosa/resultTime'
    HMeasurement:
      type: object
      x-kindOf: 'http://www.w3.org/ns/sosa/Observation'
      properties:
        h:
          type: integer
          example: 21
          x-kindOf: 'http://www.w3.org/ns/sosa/hasSimpleResult'
        timestamp:
          type: string
          format: date-time
          x-kindOf: 'http://www.w3.org/ns/sosa/resultTime'
    TempProperty:
      type: object
      required:
        - id
        - values
      x-kindOf: 'http://www.w3.org/ns/ssn/systems/SystemCapability'
      properties:
        id:
          type: string
          default: DHT22_temperature
          x-kindOf: 'http://schema.org/identifier'
        name:
          type: string
          example: Temperature
          x-kindOf: 'http://schema.org/name'
        values:
          $ref: '#/components/schemas/TempMeasurement'
        range:
          type: object
          x-kindOf: 'http://www.w3.org/ns/ssn/systems/Condition'
          properties:
            unit:
              type: string
              default: DegreeCelsius
              x-refersTo: 'https://schema.org/unitText'
            minValue:
              type: number
              format: float
              default: -40
              x-refersTo: 'https://schema.org/value'
            maxValue:
              type: number
              format: float
              default: 80
              x-refersTo: 'https://schema.org/value'
        sensorAccuracy:
          type: object
          x-refersTo: 'http://www.w3.org/ns/ssn/systems/Accuracy'
          properties:
            range:
              type: object
              x-kindOf: 'http://www.w3.org/ns/ssn/systems/Condition'
              properties:
                unit:
                  type: string
                  default: DegreeCelsius
                  x-refersTo: 'https://schema.org/unitText'
                minValue:
                  type: number
                  format: float
                  default: -0.5
                  x-refersTo: 'https://schema.org/value'
                maxValue:
                  type: number
                  format: float
                  default: 0.5
                  x-refersTo: 'https://schema.org/value'
        sensorSensitivity:
          type: object
          x-refersTo: 'http://www.w3.org/ns/ssn/systems/Sensitivity'
          properties:
            unit:
              type: string
              default: DegreeCelsius
              x-refersTo: 'https://schema.org/unitText'
            value:
              type: number
              format: float
              default: 0.1
              x-refersTo: 'https://schema.org/value'
        sensorPrecision:
          type: object
          x-refersTo: 'http://www.w3.org/ns/ssn/systems/Precision'
          properties:
            range:
              type: object
              x-kindOf: 'http://www.w3.org/ns/ssn/systems/Condition'
              properties:
                unit:
                  type: string
                  default: DegreeCelsius
                  x-refersTo: 'https://schema.org/unitText'
                minValue:
                  type: number
                  format: float
                  default: 0.2
                  x-refersTo: 'https://schema.org/value'
                maxValue:
                  type: number
                  format: float
                  default: 0.2
                  x-refersTo: 'https://schema.org/value'
        sensorFrequency:
          type: object
          x-refersTo: 'http://www.w3.org/ns/ssn/systems/Frequency'
          properties:
            unit:
              type: string
              default: sec
              x-refersTo: 'https://schema.org/unitText'
            period:
              type: integer
              format: int32
```

```
            default: 2
            x-refersTo: 'https://schema.org/value'
  xml:
    name: TempProperty
HumProperty:
  type: object
  required:
    - id
    - values
  x-kindOf: 'http://www.w3.org/ns/ssn/systems/SystemCapability'
  properties:
    id:
      type: string
      default: DHT22_humidity
      x-kindOf: 'http://schema.org/identifier'
    name:
      type: string
      example: Humidity
      x-kindOf: 'http://schema.org/name'
    values:
      $ref: '#/components/schemas/HMeasurement'
    range:
      type: object
      x-kindOf: 'http://www.w3.org/ns/ssn/systems/Condition'
      properties:
        unit:
          type: string
          default: Percent
          x-refersTo: 'https://schema.org/unitText'
        minValue:
          type: number
          format: float
          default: 5
          x-refersTo: 'https://schema.org/value'
        maxValue:
          type: number
          format: float
          default: 85
          x-refersTo: 'https://schema.org/value'
  xml:
    name: HumProperty
SubscriptionRequestBody:
  type: object
  required:
    - description
```

```yaml
        - type
        - callbackUrl
        - resource
      properties:
        name:
          type: string
          example: My subscription for DHT22 temperature
        description:
          type: string
          example: A subscription to get info about DHT22 temperature
        type:
          type: string
          example: webhook (callback)
        callbackUrl:
          type: string
          example: 'http://172.16.1.5:5000/accumulate'
        resource:
          type: object
          properties:
            type:
              type: string
              example: property
            name:
              type: string
              example: temperature
        expires:
          type: string
          format: date-time
        throttling:
          type: integer
          format: int32
          example: 5
      xml:
        name: SubscriptionRequestBody
    SubscriptionObject:
      allOf:
        - type: object
          required:
            - id
            - type
            - resource
            - description
            - callbackUrl
          properties:
            id:
```

```
            type: string
            example: 5fc978fc96cc26a4e202c3d6
        - $ref: '#/components/schemas/SubscriptionRequestBody'
      xml:
        name: SubscriptionObject
  parameters:
    pageParam:
      name: page
      description: Pagination first (page) parameter
      in: query
      required: false
      schema:
        type: integer
```

2. OpenAPI Thing description for a smart door device in YAML format

```
openapi: 3.0.3
info:
  title: Smart Door device OpenAPI Thing description
  description: An OpenAPI Thing description for a smart door device that
exposes its current state and supports lock and unlock actions (client
commands).
 The OpenAPI Web Thing template and thus the REST API of the device is
inspired by the Web Thing Model submission of W3C. Find out more about Web
Thing Model (W3C) at
[https://www.w3.org/Submission/wot-model/](https://www.w3.org/Submission/wot-
model/).
  contact:
    email: atzavaras@isc.tuc.gr
  license:
    name: An example license
    url: 'http://www.example.com/licenses/LICENSE-2.0.html'
  version: 1.0.0
externalDocs:
  description: Find out more about the OpenAPI Thing Description approach
here.
  url:
'https://www.intelligence.tuc.gr/~petrakis/publications/OpenAPIWoT.pdf'
servers:
  - url: 'http://localhost:5000/SmartDoor'
```

```
    description: An example server for the Web service exposed for the
device.
tags:
  - name: Web Thing
    x-onResource: '''#/components/schemas/Webthing'''
    description: Operations on a Web Thing
    externalDocs:
      description: Find out more
      url: 'https://www.w3.org/Submission/wot-model/#web-thing-resource'
  - name: Properties
    description: Operations on Thing properties
    externalDocs:
      description: Find out more about Thing properties
      url: 'https://www.w3.org/Submission/wot-model/#properties-resource'
  - name: Actions
    description: Operations on Thing Actions
    externalDocs:
      description: Find out more about Thing Actions
      url: 'https://www.w3.org/Submission/wot-model/#actions-resource'
  - name: Subscriptions
    x-onResource: '''#/components/schemas/SubscriptionObject'''
    description: Operations on subscriptions
    externalDocs:
      description: Find out more about subscriptions
      url: 'https://www.w3.org/Submission/wot-model/#things-resource'
paths:
  /:
    get:
      tags:
        - Web Thing
      summary: Retrieve Web Thing
      description: 'In response to an HTTP GET request on the root URL of a
Thing, an Extended Web Thing must return an object that holds its
representation.'
      operationId: retrieveWebThing
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Webthing'
        '404':
          description: Not found
  /properties:
```

```yaml
    get:
      tags:
        - Properties
      summary: Retrieve a list of properties
      description: 'In response to an HTTP GET request on the destination URL
of a properties link, an Extended Web Thing must return an array of Property
that the initial resource contains.'
      operationId: retrieveWebThingProperties
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/PropertiesResponse'
        '400':
          description: Invalid ID supplied
        '404':
          description: Not found
  /properties/state:
    get:
      tags:
        - Properties
      summary: Retrieve the value of a property (door's current state)
      description: 'In response to an HTTP GET request on a Property URL, an
Extended Web Thing must return an array that lists recent values of that
Property.'
      operationId: retrieveProperty
      parameters:
        - $ref: '#/components/parameters/pageParam'
        - name: perPage
          description: Pagination second (per page) parameter
          in: query
          required: false
          schema:
            type: integer
      responses:
        '200':
          description: successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/State'
        '404':
          description: Not found
  /actions:
    get:
      tags:
        - Actions
      summary: Retrieve a list of actions
      operationId: retrieveWebThingActions
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/Action'
        '404':
          description: Not found
  /actions/lock:
    get:
      tags:
        - Actions
      summary: Retrieve recent executions of the lock action
      description: 'In response to an HTTP GET request on an Action URL, an
Extended Web Thing must return an array that lists the recent executions of a
specific Action.'
      operationId: retrieveRecentExecutionsOfLockAction
      responses:
        '200':
          description: successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/ActionExecution'
        '404':
          description: Not found
    post:
      tags:
        - Actions
      summary: Execute a lock action
      description: 'In response to an HTTP POST request on an Action URL with
valid parameters as request body, an Extended Web Thing must either reject
the request with the appropriate status code or queue a task to run the
action and return the status of that action in a 201 Created response. The
action may not run immediately. The Location HTTP header identifies the URL
to use to retrieve the most recent update on the action''s status.'
      operationId: executeLockAction
      responses:
        '204':
          description: NO RESPONSE
        '404':
          description: Not found
  '/actions/lock/{executionId}':
    get:
      tags:
        - Actions
      summary: Retrieve the status of a lock action
      description: 'In response to an HTTP GET request on the URL targeted by
the Location HTTP header returned in response to the request to execute an
action, an Extended Web Thing must return the status of this action or a 404
Not Found status code if the action''s status is no longer available.'
      parameters:
        - name: executionId
          in: path
          description: action execution of Webthing to return
          required: true
          style: simple
          explode: true
          schema:
            type: array
            items:
              type: integer
              format: int64
              default: 1
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ActionExecution'
        '404':
          description: Not found
  /actions/unlock:
    get:
      tags:
        - Actions
      summary: Retrieve recent executions of the unlock action
      description: 'In response to an HTTP GET request on an Action URL, an
Extended Web Thing must return an array that lists the recent executions of a
specific Action.'
      operationId: retrieveRecentExecutionsOfUnlockAction
      responses:
        '200':
          description: successful operation
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/ActionExecution'
        '404':
          description: Not found
    post:
      tags:
        - Actions
      summary: Execute an unlock action
      description: 'In response to an HTTP POST request on an Action URL with
valid parameters as request body, an Extended Web Thing must either reject
the request with the appropriate status code or queue a task to run the
action and return the status of that action in a 201 Created response. The
action may not run immediately. The Location HTTP header identifies the URL
to use to retrieve the most recent update on the action''s status.'
      operationId: executeUnlockAction
      responses:
        '204':
          description: NO RESPONSE
        '404':
          description: Not found
  '/actions/unlock/{executionId}':
    get:
      tags:
        - Actions
      summary: Retrieve the status of an unlock action
      description: 'In response to an HTTP GET request on the URL targeted by
the Location HTTP header returned in response to the request to execute an
action, an Extended Web Thing must return the status of this action or a 404
Not Found status code if the action''s status is no longer available.'
      parameters:
        - name: executionId
```

```
          in: path
          description: action execution of Webthing to return
          required: true
          style: simple
          explode: true
          schema:
            type: array
            items:
              type: integer
              format: int64
              default: 1
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/ActionExecution'
        '404':
          description: Not found
  /subscriptions:
    get:
      tags:
        - Subscriptions
      summary: Retrieve a list of subscriptions
      description: 'In response to an HTTP GET request on the destination URL
of a subscriptions link, an Extended Web Thing must return the array of
subscriptions to the underlying resource.'
      operationId: retrieveListOfSubscriptions
      parameters:
        - $ref: '#/components/parameters/pageParam'
        - name: perPage
          description: Pagination second (per page) parameter
          in: query
          required: false
          schema:
            type: integer
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
```

```
                $ref: '#/components/schemas/SubscriptionObject'
        '404':
          description: Not found
    post:
      tags:
        - Subscriptions
      summary: Create a subscription
      description: An Extended Web Thing should support subscriptions.
      operationId: createSubscription
      x-operationType: 'https://schema.org/CreateAction'
      requestBody:
        description: Create a new subscription
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/SubscriptionRequestBody'
        required: true
      responses:
        '200':
          description: OK
        '404':
          description: Not found
  '/subscriptions/{subscriptionID}':
    get:
      tags:
        - Subscriptions
      summary: Retrieve information about a specific subscription
      description: 'In response to an HTTP GET request on a Subscription URL,
an Extended Web Thing must return a JSON representation of the subscription.
The JSON representation should be the same as the one returned for that
subscription in ''Retrieve a list of subscriptions''.'
      operationId: retreiveInfoAboutSubscription
      parameters:
        - name: subscriptionID
          in: path
          description: The id of the specific subscription
          required: true
          style: simple
          explode: true
          x-mapsTo: '#/components/schemas/SubscriptionObject.id'
          schema:
            type: string
            example: 5fd23faccde6be05da68bcfb
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/SubscriptionObject'
        '404':
          description: Not found
    delete:
      tags:
        - Subscriptions
      summary: Delete a subscription
      description: In response to an HTTP DELETE request on the destination
URL of a subscriptions an Extended Web Thing must either reject  the request
with an appropriate status code or remove (unsubscribe) the subscription and
return a 200 OK status code.
      operationId: deleteSubscription
      x-operationType: 'https://schema.org/DeleteAction'
      parameters:
        - name: subscriptionID
          in: path
          description: The id of the specific subscription
          required: true
          style: simple
          explode: true
          schema:
            type: string
            example: 5fd23faccde6be05da68bcfb
      responses:
        '200':
          description: OK
        '404':
          description: Not found
components:
  schemas:
    Webthing:
      required:
        - id
        - name
        - type
      type: object
      x-refersTo: 'http://www.w3.org/ns/sosa/Actuator'
      properties:
        id:
          type: string
          default: SmartDoor
          x-kindOf: 'http://schema.org/identifier'
        name:
          type: string
          example: IoTSmartDoor
          x-kindOf: 'http://schema.org/name'
        description:
          type: string
          example: 'A Smart Door is an electronic door which can be sent
commands to be locked or unlocked remotely. It can also report on its current
state (OPEN, CLOSED or LOCKED).'
          x-refersTo: 'http://schema.org/description'
        createdAt:
          type: string
          format: date-time
        updatedAt:
          type: string
          format: date-time
        tags:
          type: array
          items:
            type: string
            example: smart door
      xml:
        name: Webthing
    PropertiesResponse:
      anyOf:
        - $ref: '#/components/schemas/StateProperty'
      xml:
        name: PropertiesResponse
    State:
      type: object
      x-kindOf: 'http://www.w3.org/ns/sosa/Observation'
      properties:
        state:
          type: string
          x-kindOf: 'http://www.w3.org/ns/sosa/hasSimpleResult'
          enum:
            - OPEN
            - CLOSED
            - LOCKED
        timestamp:
          type: string
          format: date-time
          x-kindOf: 'http://www.w3.org/ns/sosa/resultTime'
      xml:
        name: State
    StateProperty:
      required:
        - id
        - values
      x-kindOf: 'http://www.w3.org/ns/ssn/systems/SystemCapability'
      properties:
        id:
          type: string
          default: state
          x-kindOf: 'http://schema.org/identifier'
        name:
          type: string
          example: Smart door's current state
          x-kindOf: 'http://schema.org/name'
        values:
          $ref: '#/components/schemas/State'
      xml:
        name: StateProperty
    Action:
      required:
        - id
        - name
      type: object
      properties:
        id:
          type: string
          example: lock
        name:
          type: string
          example: Lock the Smart Door
      xml:
        name: Action
    ActionExecution:
      required:
        - id
        - status
        - timestamp
      x-kindOf: 'http://www.w3.org/ns/sosa/Actuation'
      type: object
      properties:
        id:
          type: string
          example: '223'
        status:
```

```yaml
          type: string
          example: completed
        timestamp:
          type: string
          format: date-time
      xml:
        name: ActionExecution
    SubscriptionRequestBody:
      required:
        - description
        - type
        - callbackUrl
        - resource
      type: object
      properties:
        name:
          type: string
          example: My subscription for SmartDoor's current state
        description:
          type: string
          example: A subscription to get info about SmartDoor's current state
        type:
          type: string
          example: webhook (callback)
        callbackUrl:
          type: string
          example: 'http://172.16.1.5:5000/accumulate'
        resource:
          type: object
          properties:
            type:
              type: string
              example: property
            name:
              type: string
              example: state
        expires:
          type: string
          format: date-time
        throttling:
          type: integer
          format: int32
          example: 5
      xml:
        name: SubscriptionRequestBody
    SubscriptionObject:
      allOf:
        - required:
            - id
            - type
            - resource
            - description
            - callbackUrl
          type: object
          properties:
            id:
              type: string
              example: 5fc978fc96cc26a4e202c3d6
        - $ref: '#/components/schemas/SubscriptionRequestBody'
      xml:
        name: SubscriptionObject
  parameters:
    pageParam:
      name: page
      description: Pagination first (page) parameter
      in: query
      required: false
      schema:
        type: integer
```

Bibliography

[1] D. Guinard and V. Trifa, Building the Web of Things. Greenwich, CT, USA: Manning Publications Co., 2016. [Online]. Available: https://webofthings.org/book/

[2] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[3] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, and K. Kajimoto, "Web of Things (WoT) Architecture," Apr. 2020, W3C Recommendation. [Online]. Available: https://www.w3.org/TR/wot-architecture/

[4] Roy Fielding. Fielding dissertation: Chapter 5: Representational state transfer (rest). *Recuperado el*, 8, 2000.

[5] "JSON-LD 1.1: A JSON-based Serialization for Linked Data," Jul. 2020, W3C Working Draft. [Online]. Available: https://www.w3.org/TR/json-ld11/

[6] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. SPARQL 1.1 query language. *W3C Recommendation*, Mar. 21, 2013. Available: https://www.w3.org/TR/sparql11-query/

[7] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A Practical OWL-DL Reasoner," Journal of Web Semantics, vol. 5, no. 2, pp. 51–53, 2007, 9th Intern. Conference on Ambient Systems, Networks and Technologies (ANT 2018). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570826807000169

[8] Hai Dong, Farookh Khadeer Hussain, and Elizabeth Chang. Semantic web service matchmakers: state of the art and challenges. Concurrency and Computation: Practice and Experience, 25(7):961–988, 2013.

[9] Luc Clement, Andrew Hately, Claus von Riegen, Tony Rogers, et al. Uddi version 3.0. 2, uddi spec technical committee draft. OASIS UDDI Spec TC, 2004.

[10] Roberto Chinnici, J Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core Language, w3c recommendation, june 2007, 2007.

[11] Joel Farrell and Holger Lausen. Semantic annotations for wsdl and xml schema. w3c recommendation, 28 august 2007. World Wide Web Consortium (W3C), Tech. Rep, 2007.

[12] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, P. Paolucci, B. Parsia, T. Payne, et al. OWL-S: Semantic markup for web services. w3c submission (2004), 2004.

[13] Marc J. Hadley. Web application description language (WADL). W3C member submission. World Wide Web Consortium, W3C (November 2006), 2009.

[14] Lanthaler, M., G¨utl, C.: A vocabulary for hypermedia-driven web apis. In: Workshop on Linked Data on the Web (LDOW 2013). Rio de Janeiro, Brazil (2013). URL http://www.markus-lanthaler.com/hydra/

[15] Open API Initiative (OAI). Open api specification. Technical report, Technical report, https://github.com/OAI/OpenAPI-Specification.

[16] Markus Lanthaler. Creating 3rd generation web apis with hydra. In *Proceedings of the 22nd International Conference on World Wide Web* , pages 35–38. ACM, 2013.

[17] Mainas, N., Petrakis, E.: Soas 3.0: Semantically enriched openapi 3.0 descriptions and ontology for rest services. In: IEEE Intern. Conf. on Semantic Computing (ICSC 2020), pp. 207–210. San Diego, California (2020).

[18] A. Karavisileiou, N. Mainas, and E. G. Petrakis, "Ontology for openapi rest services descriptions," in IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2020), 2020, pp. 25–40. [Online]. Available: https://ieeexplore.ieee.org/document/9288198

[19] H. Knublauch and D. Kontokostas, "Shapes Constraint Language (SHACL)," Jul. 2017. [Online]. Available: https://www.w3.org/TR/shacl/

[20] Jagni Dasa Horta Bezerra and Cidcley Teixeira de Souza. 2019. A model-based approach to generate reactive and customizable user interfaces for the web of things. In *Proceedings of the 25th Brazillian Symposium on Multimedia and the Web* (*WebMedia '19*). Association for Computing Machinery, New York, NY, USA, 57–60. DOI: https://doi.org/10.1145/3323503.3360631

[21] A. Rhayem, M. B. A. Mhiri, and F. Gargouri, "Semantic Web Technologies for the Internet of Things: Systematic Literature Review," Internet of Things, vol. 11, pp. 1–22, 9 2020. [Online]. Available: https://doi.org/10.1016/j.iot.2020.100206

[22] Sebastian Kaebisch, Takuki Kamiya, Michael McCool, Victor Charpenay, and Matthias Kovatsch. 2020. Web of Things (WoT) Thing Description, W3C Recommendation 9 April 2020. W3C Recommendation. World Wide Web Consortium (W3C). https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/

[23] “Web Thing Model,” Aug. 2015, w3C member submission. [Online]. Available: http://www.w3.org/Submission/wot-model/

[24] Thomas Erl. Soa: principles of service design, volume 1. Prentice Hall Upper Saddle River, 2008.

[25] Tim Bray, Jean Paoli, CM Sperberg-McQueen, Eve Maler, Franois Yergeau, and John Cowan. Extensible markup language (xml) 1.1- w3c recommendation. World Wide Web Consortium. https://www.w3.org/TR/2006/REC-xml11-20060816/.

[26] World Wide Web Consortium et al. W3c: Simple object access protocol, soap, version 1.2 part 0: Primer,(2003). Web site: http://www.w3.org/TR/soap12-part0.

[27] Mike Amundsen. Building Hypermedia APIs with HTML5 and Node. " O'Reilly Media, Inc.", 2011.

[28] Tim Berners-Lee. Linked data-design issues (2006). URL http://www. w3. org/DesignIssues/LinkedData. html, 2006.

[29] Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. W3C Recommendation, 25:1–8, 2014.

[30] Dan Brickley and R.V. Guha. Rdf schema 1.1. w3c recommendation (25 february 2014). World Wide Web Consortium, 2014.

[31] OWL Working Group et al. W.: Owl 2 web ontology language: Document overview. w3c recommendation (27 october 2009), 2012.

[32] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. Semantic Services, Interoperability and Web Applications: Emerging Concepts, pages 205–227, 2009.

[33] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1, w3c note, 2001, 2001.

[34] Matthias Klusch. Semantic web service description. In CASCOM: intelligent service coordination in the semantic web, pages 31–57. Springer, 2008.

[35] RAML Workgroup. Restful api modeling language (raml). Technical report, Technical report, https://github.com/raml-org/raml-spec.

[36] Apiary. Api blueprint. Technical report, https://github.com/apiaryio/api-blueprint.

[37] John Gruber. Daring fireball: Markdown syntax documentation, 2004.

[38] K. Janowicz, A. Haller, S. J.D. Cox, D. Le Phuoc, and M. Lefrançois. SOSA: A Lightweight Ontology for Sensors, Observations, Samples, and Actuators. Journal of Web Semantics, 56:1–10, May 2019.

[39] Aikaterini Karavisileiou, Nikolaos Mainas, Fotios Bouraimis, and Euripides G.M. Petrakis. Automated Ontology Instantiation of OpenAPI REST Service Descriptions. Future Information and Communication Conference (FICC 2021), Vancuver, Canada, April 29-30, 2021.

[40] G. Myrizakis and E. G. Petrakis, "iHome: Secure Smart Home Management in the Cloud and the Fog. IOS Press, Advanced in Parallel Computing, 2020, pp. 237–263. [Online]. Available: https://ebooks.iospress.nl/volumearticle/53830

[41] S. Botonakis, A. Tzavaras, and E. G. Petrakis, "iSWoT: Service Oriented Architecture in the Cloud for the Semantic Web of Things," in Advanced Information Networking and Applications (AINA 2020), Cham, Mar. 2020, pp. 1201–1214. [Online]. Available: https://doi.org/10.1007/978-3-030-44041-1_103

[42] M. O'Riordan, "Everything You Need To Know About Publish/Subscribe," online, 2021, the ABLY platform. [Online]. Available: https://ably.com/topic/pub-sub

[43] K. Stravoskoufos, E. Petrakis, N. Mainas, S. Batsakis, and V. Samoladas, "Sowl ql: Querying spatio-temporal ontologies in owl," Journal on Data Semantics, vol. 5, no. 4, pp. 249–269, Dec. 2016. [Online]. Available: https://doi.org/10.1007/s13740-016-0064-5

[44] T. Wang, S. Zhang, A. Liu, Z. A. Bhuiyan, and Q. Jin, "A Secure IoT Service Architecture with an Efficient Balance Dynamics Based on Cloud and Edge Computing," IEEE Internet of Things Journal, vol. 6, no. 4, pp. 4831–4843, 6 2019.

[45] World Wide Web Consortium et al. W3c: Simple object access protocol, soap, version 1.2 part 0: Primer,(2003). Web site: http://www.w3.org/TR/soap12-part0.