Technical University of Crete
School of Electrical and Computer Engineering
Intelligent Systems Laboratory

Diploma Thesis

Web proxy service for the Web of Things

Isidoros Paterakis

Submitted in part fulfilment of the requirements for the degree of
Integrated Master of Engineering in Electrical and Computer Engineering at the
Technical University of Crete, December 2022

Abstract

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. WoT is a relatively new concept. There is no universal application layer protocol to enable Things and services to communicate. 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 should 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. 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. The focus of this work is on designing and implementing a Web Proxy service (Nexus) for exposing Things / Applications on the Web that decouples the Thing's functionality and its management. Nexus is a RESTful Service-Oriented Architecture that can be deployed in the Cloud. OpenAPI is a universal language that can be used to accurately and completely describe Things / Applications in a universal way that is understandable by both humans and machines. In this proposed architecture, we consider that all Things / Applications are described by an OpenAPI description.

Acknowledgements

I would like to express my gratitude to my thesis advisor Euripides G. M. Petrakis.

All this project could not be done without the help of Aimilios Tzavaras.

I must express my profound gratitude to my parents.

Finally, I must thank all my friends.

Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

Nowadays [19], 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 [8] 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 [16] 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 [12] defines an abstract architecture and sets the requirements for interacting with Things in the Web using the REST architectural style [7].

The emergence of REST generated new difficulties in the representation of hypermedia driven APIs [1]. In fact, hypermedia-driven APIs are consistent with the idea of the dynamic discovery of resources at runtime (referred to as HATEOAS [9]), 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[1] 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.

The focus of this work is on designing and implementing a Web Proxy service for exposing Things on the Web. A Web Thing Proxy service must be based on the principles of the WoT initiative. 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

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

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.

OpenAPI (formerly known as Swagger) 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[2] (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 [15] [14] has proposed that OpenAPI service descriptions can be semantically 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.

1.3 Proposed Solution

The focus of this work is on designing and implementing a Web Proxy service (Nexus) for exposing Things / Applications on the Web that decouples the Thing's functionality and its management. OpenAPI is a universal language that, according to [19], can be used to accurately and completely describe Things / Applications in a universal way that is understandable by

[2]https://openapi.tools/

both humans and machines. In this proposed architecture, we consider that all Things / Applications are described by an OpenAPI description. Nexus can be deployed in either the Cloud or a Gateway. Nexus's API is RESTful. Nexus provides a platform for Infrastructure Owners to store their Things as services, expose them to the web and manage them with the tools provided to them. It is also a platform for Application Developers to store their Applications, search for exposed Things that interest them, subscribe to them and use them in their Applications. Finally, it allows anyone to register as a Normal User and subscribe to any application. Nexus contains a generator for OpenAPI descriptions of Things, a server for storing and querying OpenAPI descriptions of Things / Applications and a publish-subscribe system for Things / Applications. It also contains a tool for managing Things / Applications according to the OpenAPI description format. Finally, it contains a database API for storing and querying historic values of Things / Applications.

1.4 Contributions of the Work

Nexus achieves the following:

- The must important thing Nexus achieves is that it decouples the Thing's functionality from its management.
- It allows a single Thing to be used in many different Applications as multiple Application developers can subscribe to it.
- It supports all kinds of Things / Applications provided that they have an OpenAPI description and their input data is in JSON format. Even if an Infrastructure Owner does not have an OpenAPI description of the Thing, the OpenAPI Generator can be used to create one with a simple user input.
- Many Things are stored and Application Developers can make queries about them. They can choose any Thing they want to subscribe to for a fee and use it in any number of their Applications.

- It can be deployed in either Cloud or a Gateway. The data it receives are from http requests where the payload is in JSON format. That means that the data can be handled easily without taking into account the Thing's transmission protocol or peculiarities.
- The system's design can support a business model by assigning different role to each user.
- Nexus is secure by design. It contains many security mechanisms that protect the system services, Things / Applications stored and the user data.

1.5 Thesis Outline

This thesis has been organized into six chapters. This section outlines the description of each chapter:

- In Chapter 2, we provide a brief introduction to basic concepts and technologies which are used throughout the Thesis.
- In Chapter 3, we analyze the system requirements and the system design.
- In Chapter 4, we present the system's API and how it is used
- In Chapter 5, we analyze the system's performance.
- In Chapter 6, we expose our final thoughts with future work possibilities.

Chapter 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 in order to create User Interfaces (UIs) for Things.

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) [3] is the semantic extension of WoT that allows Things to become machine discoverable on the Web using Semantic Web tools such as SPARQL.

The concept of WoT has received considerable attention from IoT vendors and from many investigators over the past few years. The WoT Working group[1] is an ongoing effort to create standards-track specifications and test suites. For example, the Thing Description specification of W3C (Working Draft) [10] 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[2].

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.

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).

[1]https://www.w3.org/WoT/wg/
[2]https://www.w3.org./WoT/IG/wiki/Implementations

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)"[3]. 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 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[4] 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[5] such as CoRE Link Format[6], OMA LWM2M[7], and OCF[8], whereas form is a new concept. Besides W3C WoT, the concept of forms is introduced by the Constrained RESTful Application Language (CoRAL)[9] defined by the IETF.

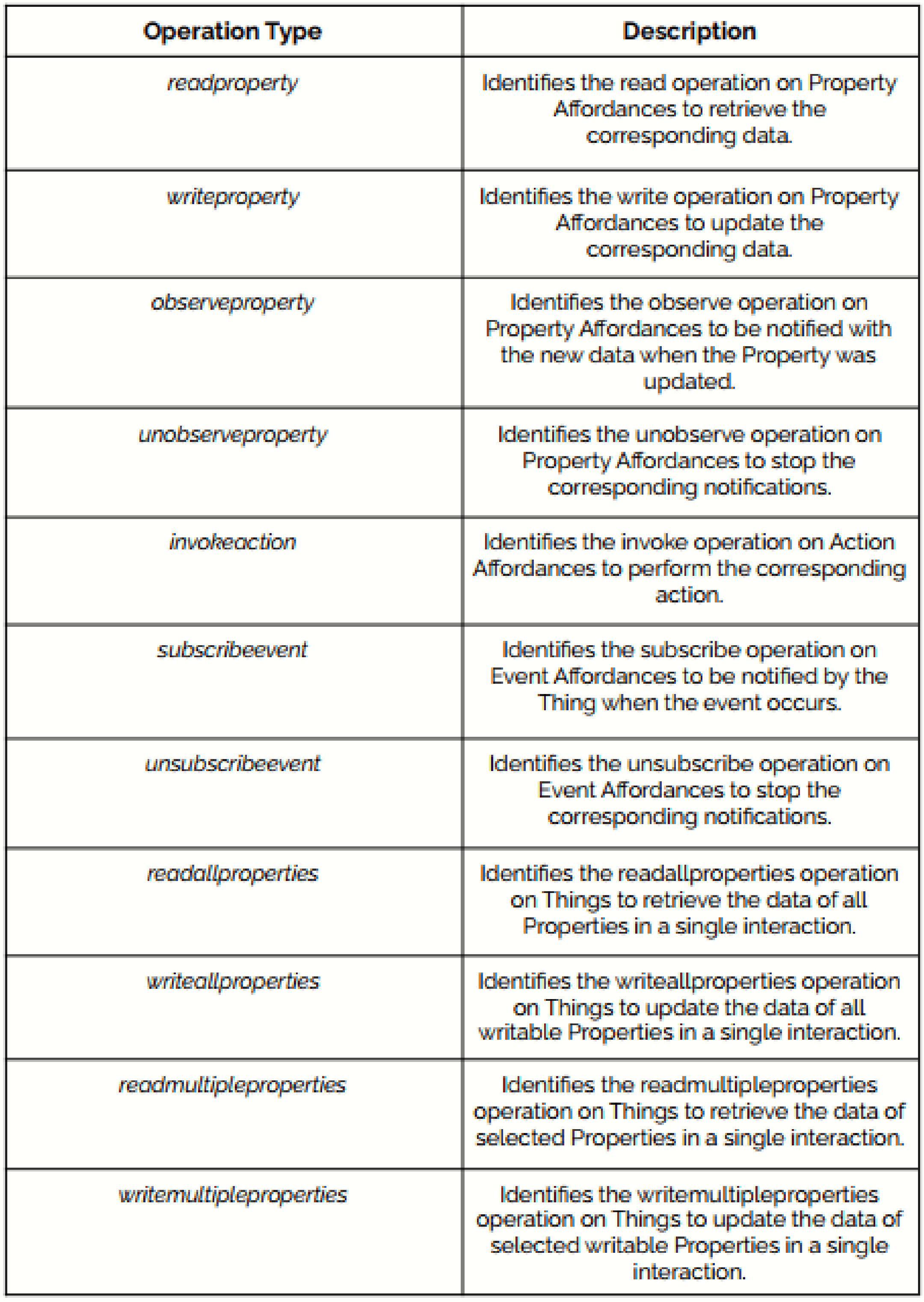
[3]https://www.w3.org/TR/wot-architecture/#terminology
[4]https://httpwg.org/specs/rfc8288.html
[5]https://www.w3.org/TR/wot-architecture/#dfn-iot-platform
[6]https://datatracker.ietf.org/doc/html/rfc6690
[7]http://openmobilealliance.org/release/LightweightM2M/V1_1-20180710-A/OMA-TS-LightweightM2M_Core-V1_1-20180710-A.pdf
[8]https://openconnectivity.org/developers/specifications/
[9]https://datatracker.ietf.org/doc/html/draft-hartke-t2trg-coral

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.

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 Figure 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[10] 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.

[10]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.

Figure 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 Working Draft document [10].

The Web of Things (WoT) Thing Description document[11] 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;

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

it is not used to describe a particular Thing instance.

2.3 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. 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.

Some of the advantages of SOA Architectures are the following:

- Reusability: In SOA, an application is created with the usage of many autonomous parts. Therefore, services can be reused in multiple applications despite their connections with other services.
- Easy service Maintenance: Provided a service is an autonomous entity, it can be updated without taking into account other services. That means that big and complex applications can be easily managed in cases of updates, upgrades and maintenance.

- Reliability: Applications based on SOA are more reliable because smaller and independent services can be tested and fixed much easier than big chunks of code.
- Scalability and Availability: Many instances of a service can be deployed at the same time in different servers. That increases the scalability and availability of the service.
- Flexibility: SOA makes the development of a complex product easier by implementing many different products from different providers ignoring the platform and background technology.

2.4 RESTful Web Services

Web services technology was initially built on existing standards such as Extensible Markup Language (XML) [4], Simple Object Access Protocol (SOAP) [17], Web Services Description Language (WSDL) [5] and Universal Description Discovery and Integration (UDDI) [6]. XML [4] 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 [7] 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 [13], 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) [9]. It is worth mentioning that Fielding himself highlighted this fact in a blog post[12] and explained that a service is considered RESTful only if all REST principles are met. The term Hypermedia API [1] has emerged to describe services that are implemented incorporating the hypermedia constraint.

Some of the REST principles are the following:

- Stateless: Every HTTP request contains all the necessary information for its execution. That means that neither the client nor the server have to store a previous state to satisfy the request.

- Resources are always handled by the URI: URI is the unique identifier of every resource in a REST architecture. Getting information about a resource, changing its content, deleting it or executing a related action are only accessible with the URI.

- Data transfer with JSON and XML: Most REST architectures tend to use either of those data representation formats for data exchange between clients and servers. Between many different data formats, the integrated confirmation properties of XML and the flexibility of JSON helped them become the most popular formats in RESTful communication. JavaScript Object Notation (JSON) is a lightweight data exchange format that is easy to read and write for a human and also easy to parse and be generated by machines. Those properties make JSON ideal for data exchange.

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

2.5 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.

2.6 SOA and Cloud Computing

SOA (Section 2.3) and Cloud Computing (Section 2.5) 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.

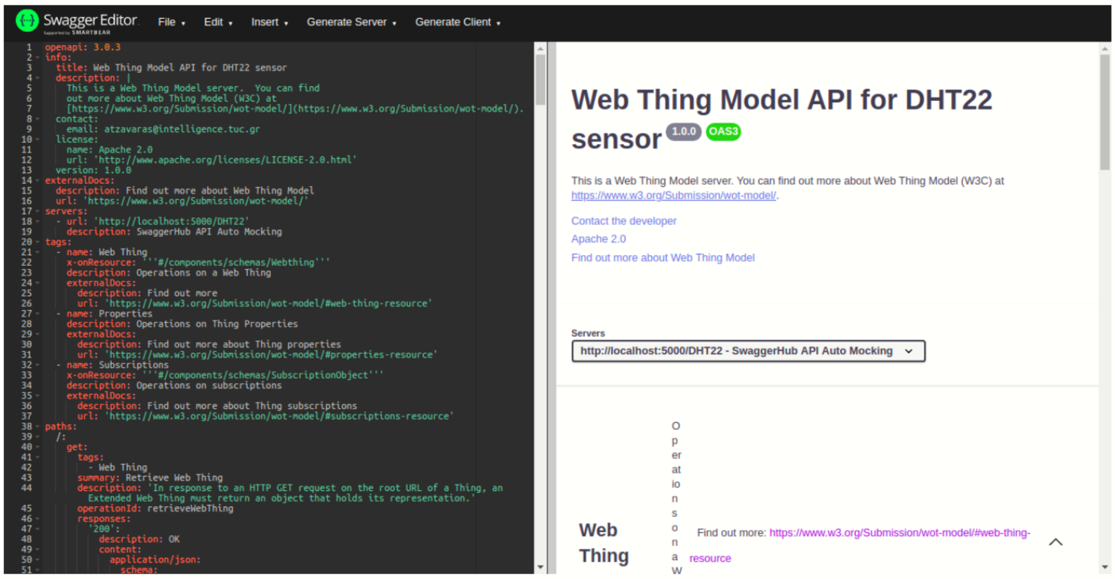
2.7 WoT and Cloud Computing

WoT (Section 2.1) and Cloud Computing (Section 2.5) 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 OpenAPI Specification

The OpenAPI Specification (OAS) [18], formerly known as Swagger, is probably the most heavily adopted approach for the description of RESTful services (Section 2.4). 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[13] 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).

[13]https://yaml.org



```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

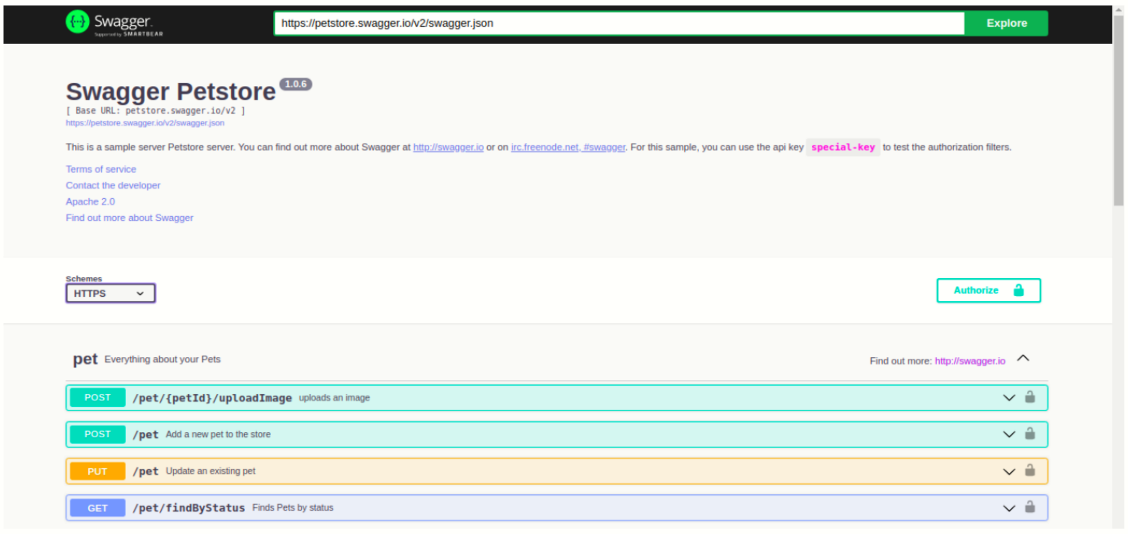
This is a Web Thing Model server. You can find out more about Web Thing Model (W3C) at https://www.w3.org/Submission/wot-model/.

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 2.2: Swagger Editor preview



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
Authorize

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 2.3: Swagger UI preview

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[14] is an open source Web-based editor for designing, defining and documenting RESTful services (Figure 2.2). It provides instant visualization and interaction with the API while still defining

[14]https://github.swagger-api/swagger-editor

it. The Swagger Codegen[15] 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[16] is an open-source HTML5-based user interface to visually render documentation for an OpenAPI service description (Figure 2.3).)

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.

2.9 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 [14] analyzed the reasons that cause ambiguities in OpenAPI service descriptions, taking into consideration version 3.0 of the specification. For example, the same OpenAPI

[15]https://github.com/swagger-api/swagger-codegen
[16]

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, [14] 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 discovery, and reasoning tools for detecting inconsistencies in service descriptions. The ontology proposed in [14] 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 [11], which describes OpenAPI objects and validates Schema descriptions against the ontology.

Figure 2.4 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.

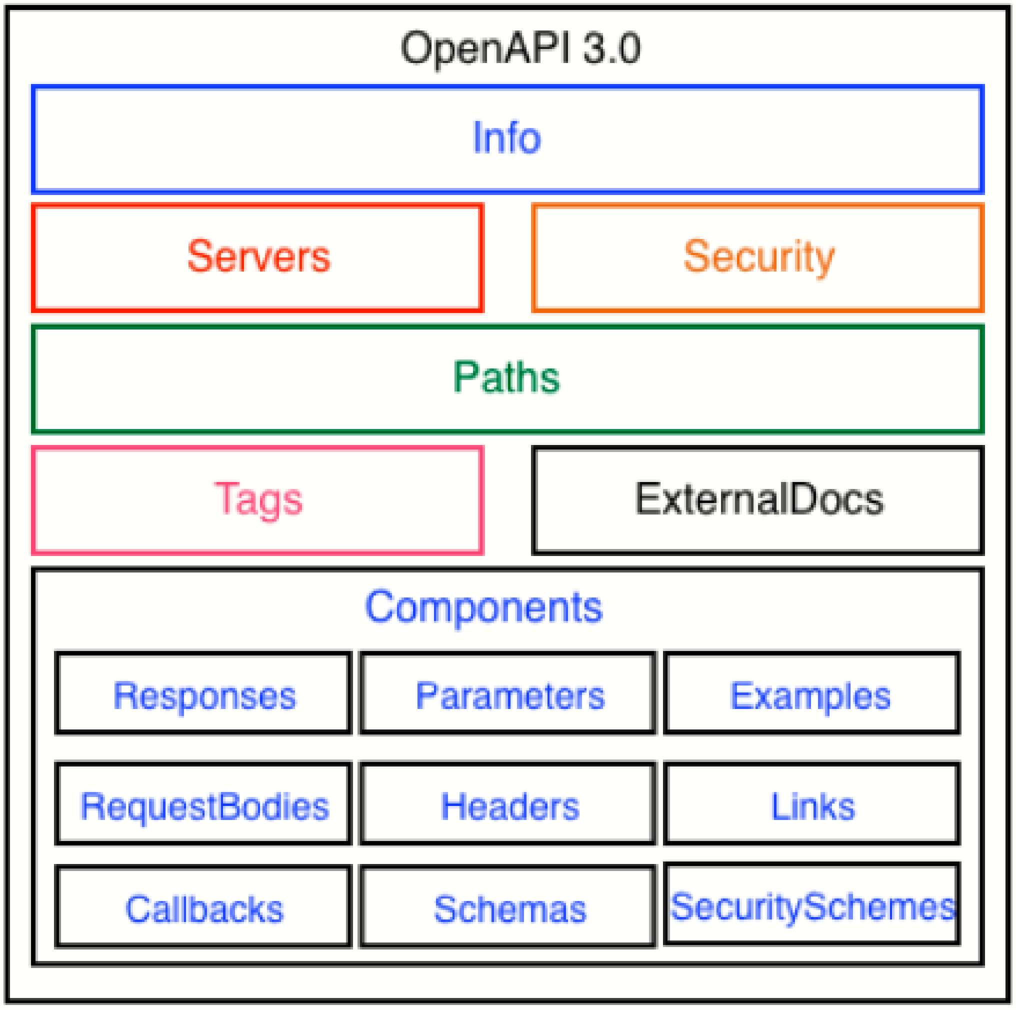
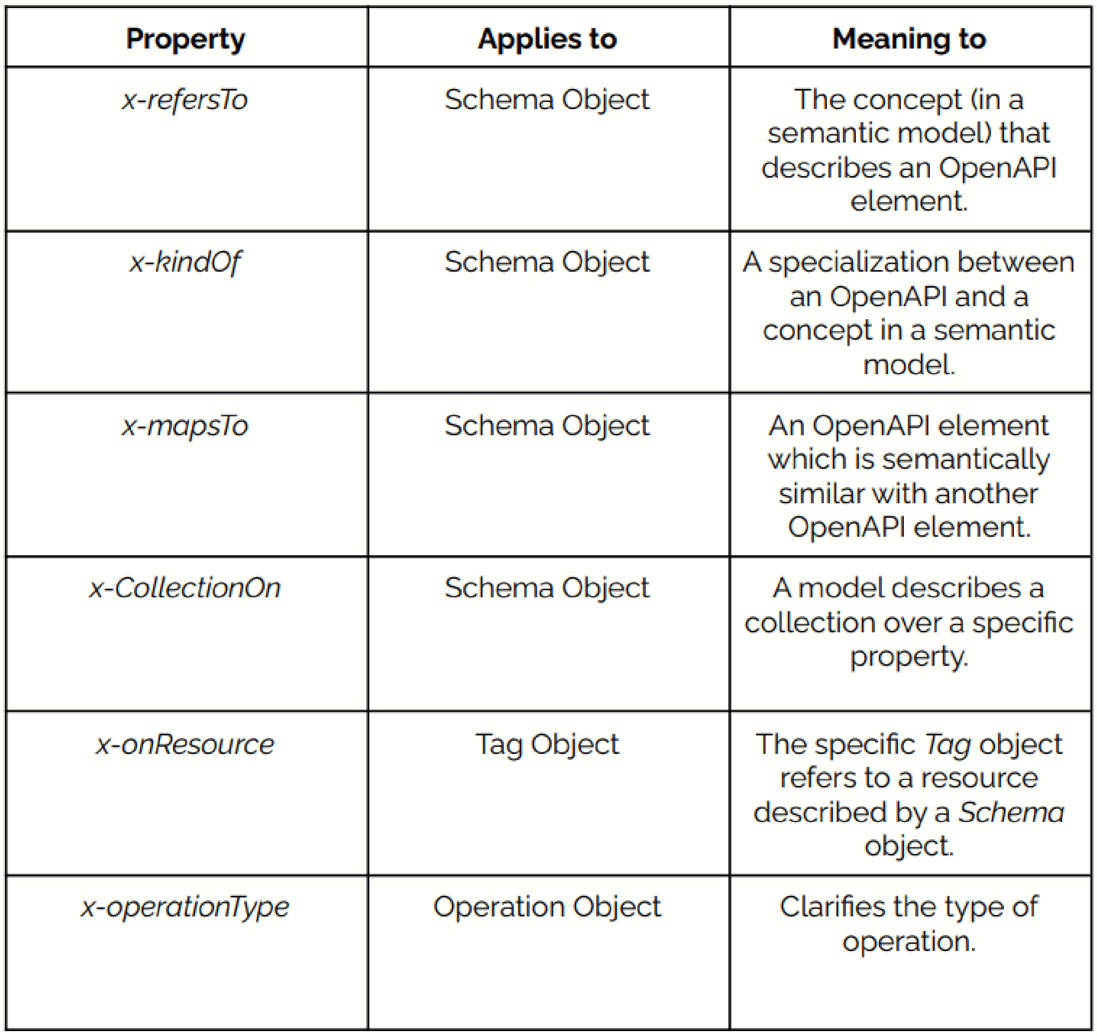


Figure 2.4: OpenAPI document structure

The description includes a Paths object that holds all the available service paths (i.e. end-points) 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.

The Schemas object describes the request and response messages based on JSON Schema[17]. 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 [20]. 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 for clarifying the meaning of operations (x-operationType). Figure 2.5 illustrates all the extension properties for semantic annotations proposed in Semantic OpenAPI.

[17]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.

Figure 2.5: OpenAPI extension properties for semantic annotations

2.10 OpenAPI Thing Generator

The flow-chart of Figure 2.6 summarizes the mechanism that generates the OpenAPI description of a Thing from user input. The input comprises: a) the standard OpenAPI Thing

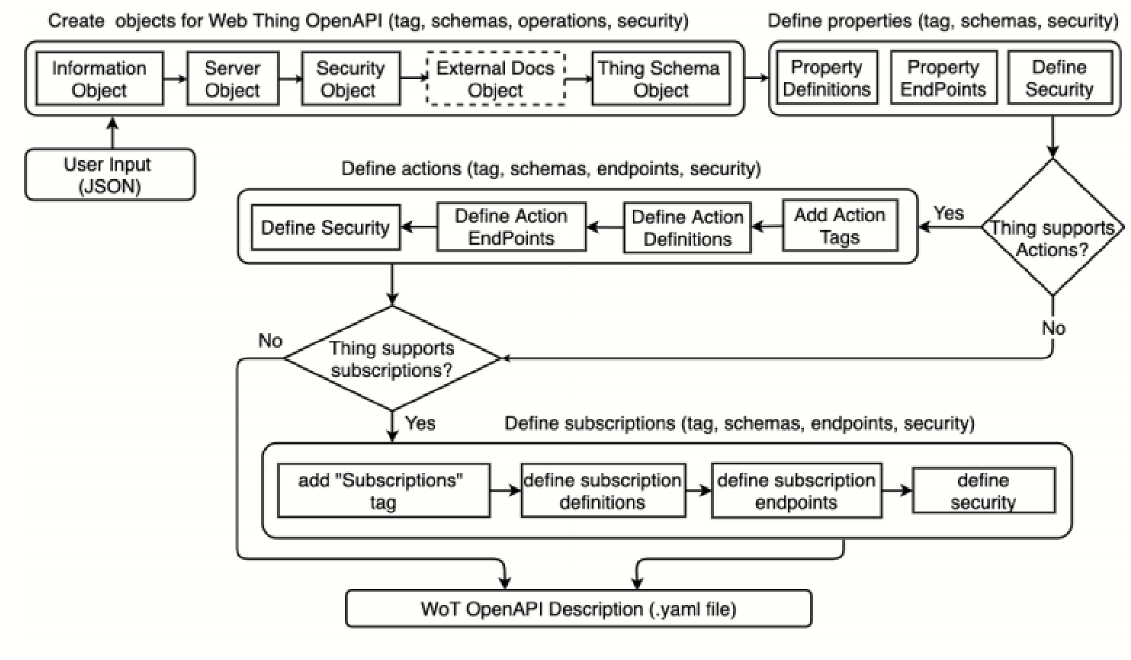

Create objects for Web Thing OpenAPI (tag, schemas, operations, security)
Information Object
Server Object
Security Object
External Docs Object
Thing Schema Object
Define properties (tag, schemas, security)
Property Definitions
Property EndPoints
Define Security
User Input (JSON)
Define actions (tag, 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 2.6: OpenAPI Generator Flowchart

Description template of Figure 2.7 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[18] 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.

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. This payload

[18]https://github.com/Emiltzav/wot

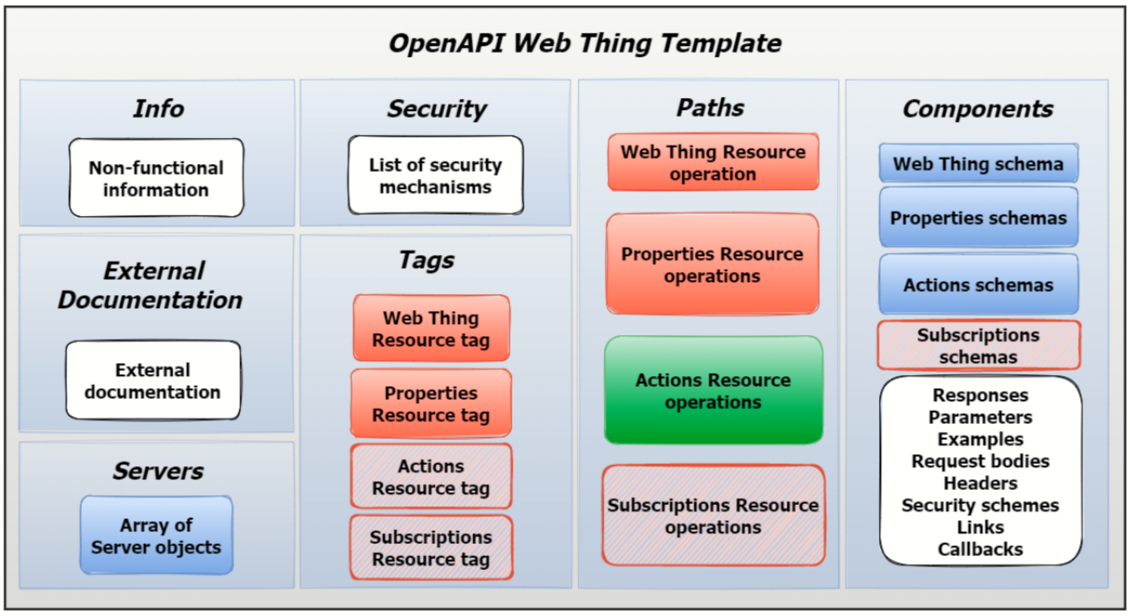


Figure 2.7: OpenAPI Web Thing Template

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 (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 2.1 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.

```
{
"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"
}
}
}
```

Listing 2.1: User input for a smart door OpenAPI description

Chapter 3

System Requirements and Design

3.1 Use Case

Consider a WoT Proxy architecture. The 4 types of users it supports are Administrators, Infrastructure Owners, Application Developers and Normal Users.

Administrators are responsible for managing the users and setting up their access rights. Infrastructure Owners register and manage their Things. Application Developers subscribe to Things and use them to create Applications. Finally, Normal Users just subscribe to applications. Each user can access different parts of the system. Infrastructure Owners and Application Developers are subclasses of Normal Users and inherit their properties.

3.2 Functional Requirements

The functional requirements are the functions the system must implement. Those requirements are different for each user category. The functional requirements for each user role are presented below.

Normal Users:

- SignUp: Users provide their personal information to register in the system.
- GetAccessToken: Users log in to the system with their email and password.
- MakeApplicationSubscription: Users select an application they want and subscribe to it.
- UpdateSubscription: Users update one of their subscriptions.
- DeleteSubscription: Users delete one of their subscriptions.
- GetSubscription: Users get information about one of their subscriptions.
- GetAllUserSubscriptions: Users get information about all subscriptions they have made.
- GetAvailableApplications: Users get all applications they can subscribe to.
- MakeOpenAPIDescriptionQuery: Users make a query on all stored OpenAPI descriptions.
- GetSubscriptionStatus: Users get the status of one of their subscriptions.
- DisableSubscription: Users disable one of their subscriptions.
- EnableSubscription: Users enable one of their subscriptions.

Infrastructure Owners:

- GetAllEntitySubscriptions: Users get all subscriptions made to a Thing they own.
- CreateOpenAPIDescription: Users use the OpenAPIGenerator to make an OpenAPI description of a Thing
- EnableHistoricDataCollection: Users allow the collection of historic data of one of their Things with STH-Comet or Cygnus.
- StoreOpenAPIDescription: Users store the OpenAPIDescription of one of their Things.
- CreateAccessRights: Users create the rights to access one of their Things.
- UseManagementService: Users use WTMs to manage their Things.

Application Developers:

- GetAllEntitySubscriptions: Users get all subscriptions made to an Application they own.
- MakeThingSubscription: Users make a subscription to an available Thing.
- EnableHistoricDataCollection: Users allow the collection of historic data of one of their Applications with STH-Comet or Cygnus.
- MakeHistoricDataQuery: Users make a query on the historic data of one of the Things they have a subscription on.
- GetAvailableThings: Users get all Things they can subscribe to.
- StoreOpenAPIDescription: Users store the OpenAPIDescription of one of their Applications.
- CreateAccessRights: Users create the rights to access one of their Things.
- UseManagementService: Users use WTMs to manage their Things.

Administrators:

- GetUserIdFromUsername: Users get the Id of a user from their username.
- UpdateUser: Users update a user.
- DeleteUser: Users delete a user.
- GetAllUsers: Users get all registered users.
- GetAdminToken: Users get the special admin token.
- GetUserInfo: Users get the personal information of a user.
- GetOrganizationId: Users get the Id of a user role.
- CheckOrganizationMembership: Users check whether a user has a specific role.

- GetLogs: Users get the system logs.
- DeleteLogs: Users delete the system logs.
- GetAccessRights: Users get the access rights of users.
- UpdateAccessRights: Users update the access rights of users.
- DeleteAccessRights: Users delete the access rights of users.

3.3 Non-Functional Requirements

The non-functional requirements are not mandatory for the system to perform its basic functionality. However, better non-functional requirements result in a better final product especially in commercial applications. The non-functional requirements are presented below.

- Performance: The response time of the system. All functions of the system are executed in real time (bellow 300 ms).
- Scalability: It is important that when new users, Things or Applications are added, the rest of the system functions are not affected.
- Security: It is important that users can log in to the system safely and their private information is protected. Also it is important that services are protected by preventing unauthorized accesses to them. With the usage of OAuth2, Service Proxy and Access Control the above are satisfied in this system.
- Usability: How easily the system can be used. The system has a detailed documentation, all services are RESTful and the payload is in either JSON format or is plain text. The above make the system easy to use by any user.
- Uptime: In web applications it is important that the system is online and can accept user requests almost always. Therefore, it is important that the system downtime is as low as possible. Downtime can happen because of maintenance, error on the cloud provider's

side or changes on the actual code. The above are satisfied because Nexus is a SOA. SOA makes maintenance and scalability easier and there is no single point of failure.

3.4 Class Diagram

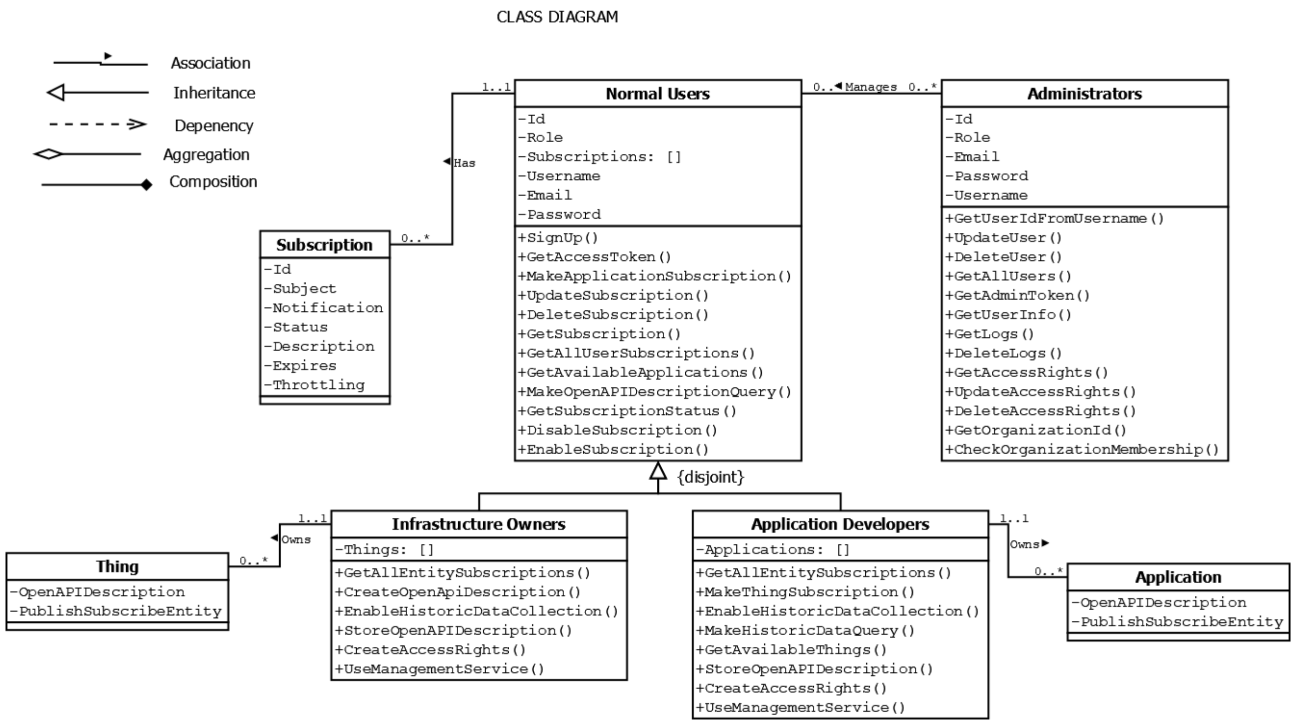


Figure 3.1: Class Diagram

The system's functional requirements can be displayed in a UML Class Diagram. The system's Class Diagram is shown in Figure 3.1. The attributes of each class are shown with the minus(-) sign and the operations with the plus(+) sign. It is shown that each user has an Id, a Role, a Username, an Email, a Password and finally a list of Subscriptions. Each subscription has an Id, a Subject (target entity), a Notification, A Status, an Expiration date and potentially a Throttling field. Infrastructure Owners and Application Developers are an extension of Normal Users. Infrastructure Owners have a list of Things they own and Application Developers have a list of Applications they own. Each Thing / Application has its own OpenAPI

Description and its own Publish-Subscribe entity. The arrows between each class are explain their relation in regards of type and cardinality.

3.5 Activity Diagrams

In the Figures 3.2, 3.3, 3.4 and 3.5, four Activity Diagrams, and their explanations, for four of the system's basic functionalities are shown.

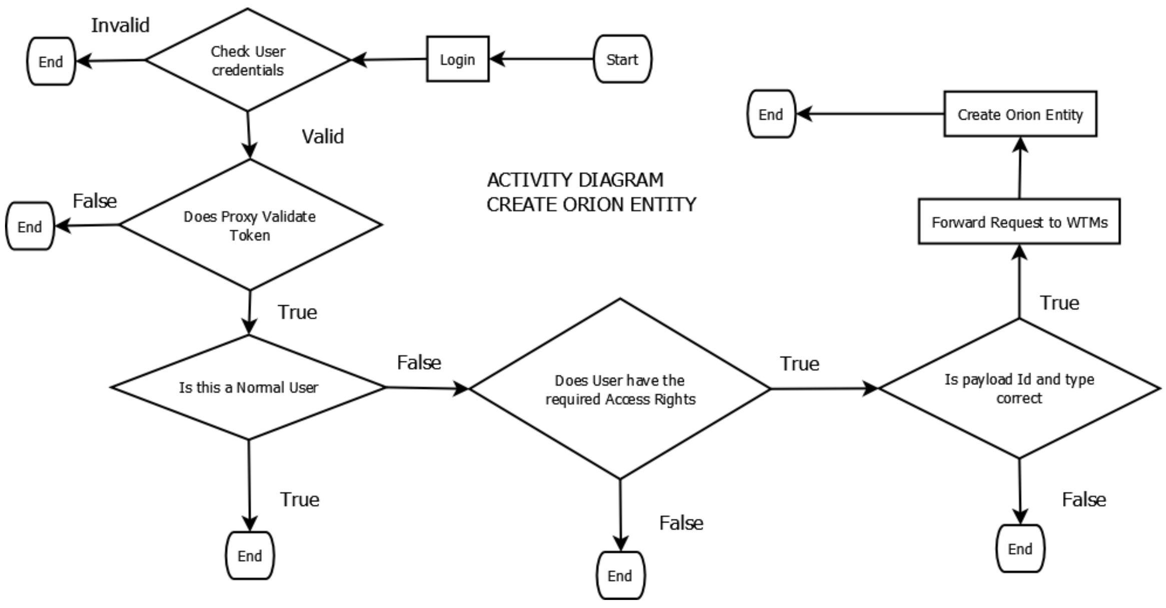


Figure 3.2: Activity Diagram 1: Create Orion Entity

1. The user logs in the system with the email and password.
2. The system checks whether the provided credentials are correct or not. If the credentials are correct, then the user is granted a token that is used to make the request to the system's application logic service. If the credentials are wrong, then the user is not granted an access token.
3. The system proxy checks if the token used to access the service is valid or not. If the token is valid, then the request is forwarded to the service. If the token is invalid, then the request is denied.

4. The system checks whether the user is a Normal User or not. If the user is a Normal User, then the request is denied. If the user is not a Normal User (which means the user is either an Infrastructure Owner or an Application Developer), then the check passes.

5. The system checks whether the user has the required Access Rights to create that entity. If the user does not have them, then the request is denied. If the user has them, then the check passes.

6. The system checks if the request payload id and type are correct. If they are not, then the request is denied. If they are correct then the request is forwarded to WTMs.

7. WTMs executes the related code functionality.

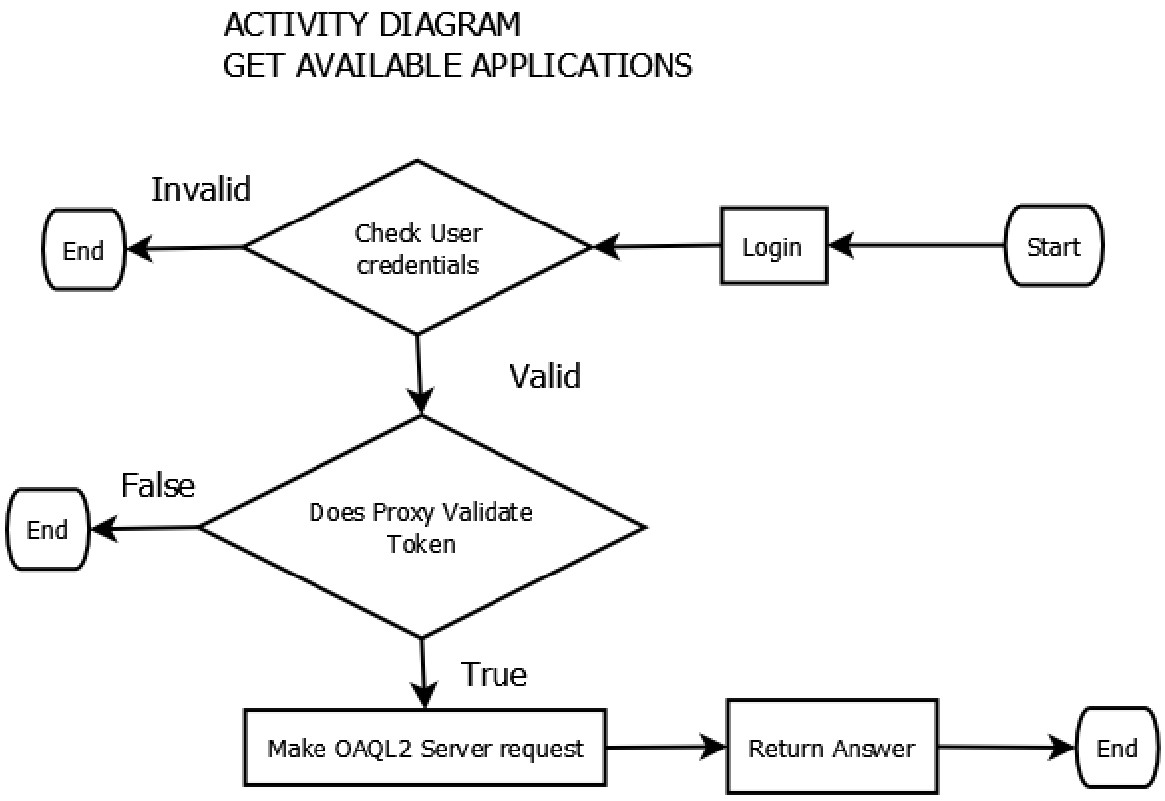


Figure 3.3: Activity Diagram 2: Get Available Applications

1. The user logs in the system with the email and password.

2. The system checks whether the provided credentials are correct or not. If the credentials are correct, then the user is granted a token that is used to make the request to the system's application logic service. If the credentials are wrong, then the user is not granted an access token.

3. The system proxy checks if the token used to access the service is valid or not. If the token is valid, then the request is forwarded to the service. If the token is invalid, then the request is denied.

4. A fixed request is made to the OAQL2 Server.

5. The OAQL2 Server answer is returned.

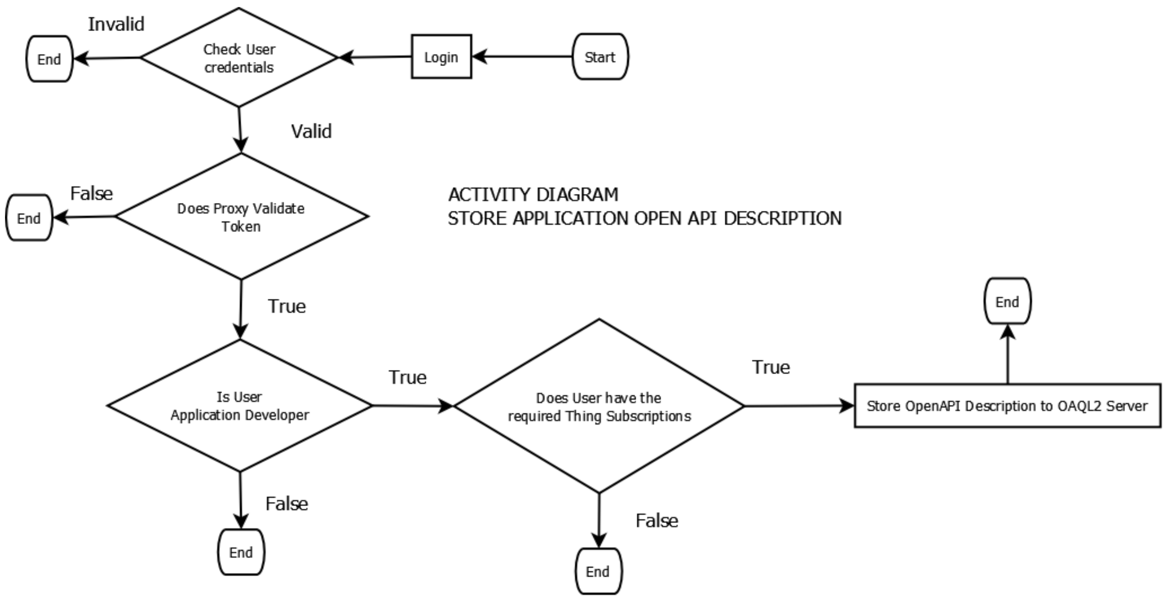

Start
Login
Check User credentials
Invalid
End
Valid
Does Proxy Validate Token
False
End
ACTIVITY DIAGRAM
STORE APPLICATION OPEN API DESCRIPTION
True
Is User Application Developer
True
False
End
Does User have the required Thing Subscriptions
True
False
End
Store OpenAPI Description to OAQL2 Server
End

Figure 3.4: Activity Diagram 3: Store Application OpenAPI Description

1. The user logs in the system with the email and password.

2. The system checks whether the provided credentials are correct or not. If the credentials are correct, then the user is granted a token that is used to make the request to the system's application logic service. If the credentials are wrong, then the user is not granted an access token.

3. The system proxy checks if the token used to access the service is valid or not. If the token is valid, then the request is forwarded to the service. If the token is invalid, then the request is denied.

4. The system checks whether the user is an Application Developer or not. If the user is not an Application Developer, then the request is denied. If the user is an Application Developer, then the check passes.

5. The system checks whether the user has the required subscriptions to the Things the Application uses. If the user does not have them, then the request is denied. If the user has them, then the check passes.

6. The OpenAPI description of the Application is stored to the OAQL2 Server.

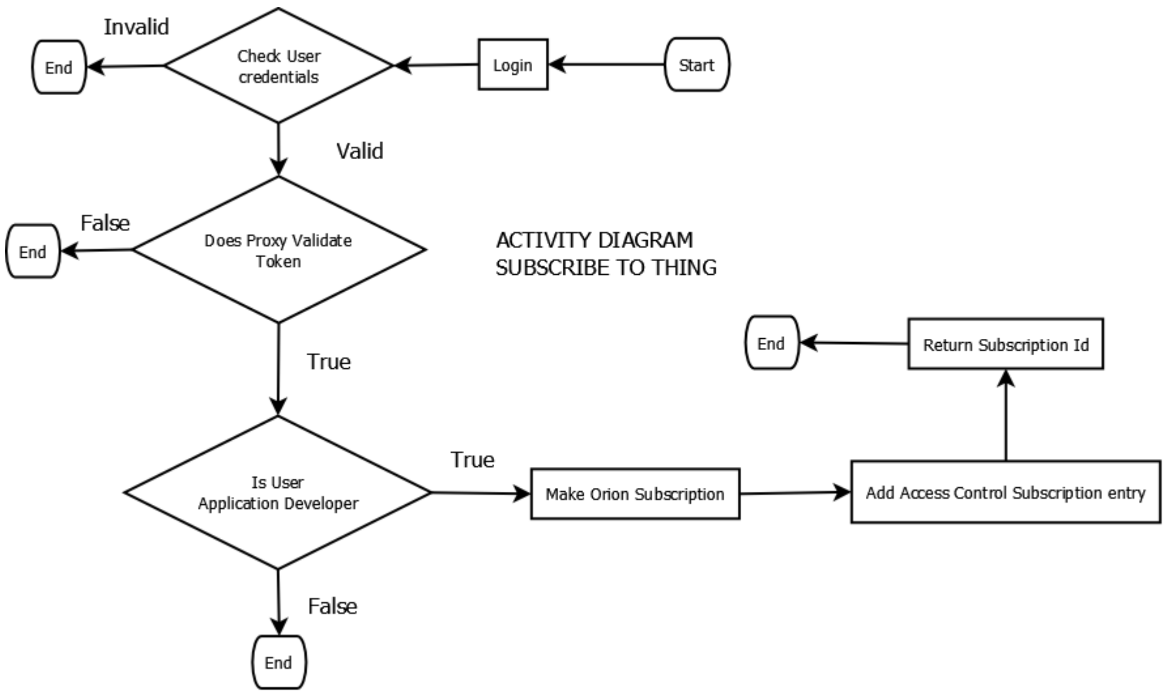

Invalid
End
Check User credentials
Login
Start
Valid
False
End
Does Proxy Validate Token
ACTIVITY DIAGRAM SUBSCRIBE TO THING
End
Return Subscription Id
True
True
Is User Application Developer
Make Orion Subscription
Add Access Control Subscription entry
False
End

Figure 3.5: Activity Diagram 4: Subscribe To Thing

1. The user logs in the system with the email and password.

2. The system checks whether the provided credentials are correct or not. If the credentials are correct, then the user is granted a token that is used to make the request to the

system's application logic service. If the credentials are wrong, then the user is not granted an access token.

3. The system proxy checks if the token used to access the service is valid or not. If the token is valid, then the request is forwarded to the service. If the token is invalid, then the request is denied.
4. The system checks whether the user is an Application Developer or not. If the user is not an Application Developer, then the request is denied. If the user is an Application Developer, then the check passes.
5. The subscription on the Orion Context Broker is made.
6. The Access Control Subscription entry is added.
7. The Orion Context Broker subscription Id is returned.

3.6 Architecture

The system architecture is shown in Figure 3.6. Consider that each service is protected by Wilma Pep Proxy. To avoid making the figure too crowded, the service proxies are not shown.

The arrows show the RESTful communication between the system's services. Each service is represented with different colour. Shapes with the same colour mean that the services are linked. The services are explained below.

- Grey: Application Logic Service, Frontend, Authentication.
- Red: User Management Service, Keyrock, Users Database.
- Light Blue: Access Control, Access Control Database.
- Green: Publish-Subscribe Service (Orion Context Broker), Publish-Subscribe Database.

- Blue: OAQL2 Server, OpenAPI Descriptions Database.
- Yellow: History Data.
- Light Green: Web Thing Model Service.
- Blue Grey: OpenAPIGenerator.

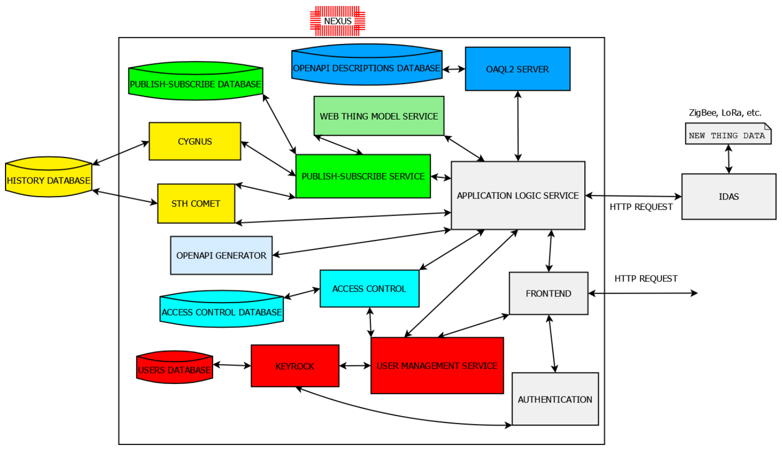


Figure 3.6: The Complete Architecture

3.6.1 Application Logic

Application Logic serves as the main service that connects all the system services. It implements the system's functional requirements. Users, through the system's Frontend service, make a request to the Authentication service to get an access token (or register if they are not registered yet). After they get a token, they can freely make their requests to the Application Logic endpoints. The Application Logic port is exposed to the outside world and through that requests to the other system services are made. Depending on the endpoint of the service, different access control checks are made. The service is protected by a proxy so all requests made must contain, on the request headers, the OAuth2 token that was previously generated. All access control checks made to the endpoints are explained in Table 3.6.1. It is important to note that on the UseManagementService endpoint, the validity of the payload is also checked. The related parts of the architecture are shown in Figure 3.7.

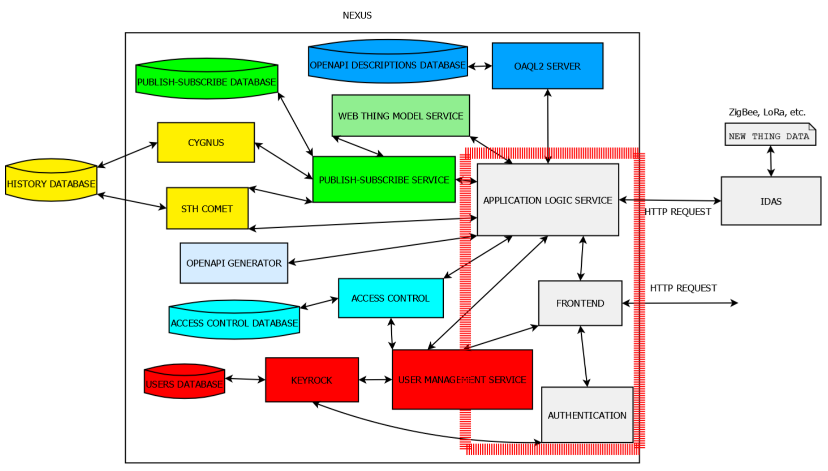


Figure 3.7: Application Logic

Table 3.1: Application Logic Access Control Checks

Endpoint	Role	Access Rights	Subscriptions
CreateAccessRights	YES	NO	NO
CreateOpenAPIDescription	YES	NO	NO
DeleteSubscription	NO	NO	YES
DisableSubscription	NO	NO	YES
EnableHistoricDataCollection	YES	YES	NO
EnableSubscription	NO	NO	YES
GetAllEntitySubscriptions	YES	YES	NO
GetAllUserSubscriptions	NO	NO	YES
GetAvailableApplications	NO	NO	NO
GetAvailableThings	YES	NO	NO
GetSubscription	NO	NO	YES
GetSubscriptionStatus	NO	NO	YES
MakeApplicationSubscription	NO	NO	NO
MakeHistoricDataQuery	YES	NO	YES
MakeOpenAPIDescriptionQuery	NO	NO	NO
MakeThingSubscription	YES	NO	NO
StoreOpenAPIDescription	YES	YES	NO
UpdateSubscription	NO	NO	YES
UseManagementService	YES	YES	NO

3.6.2 User IDM

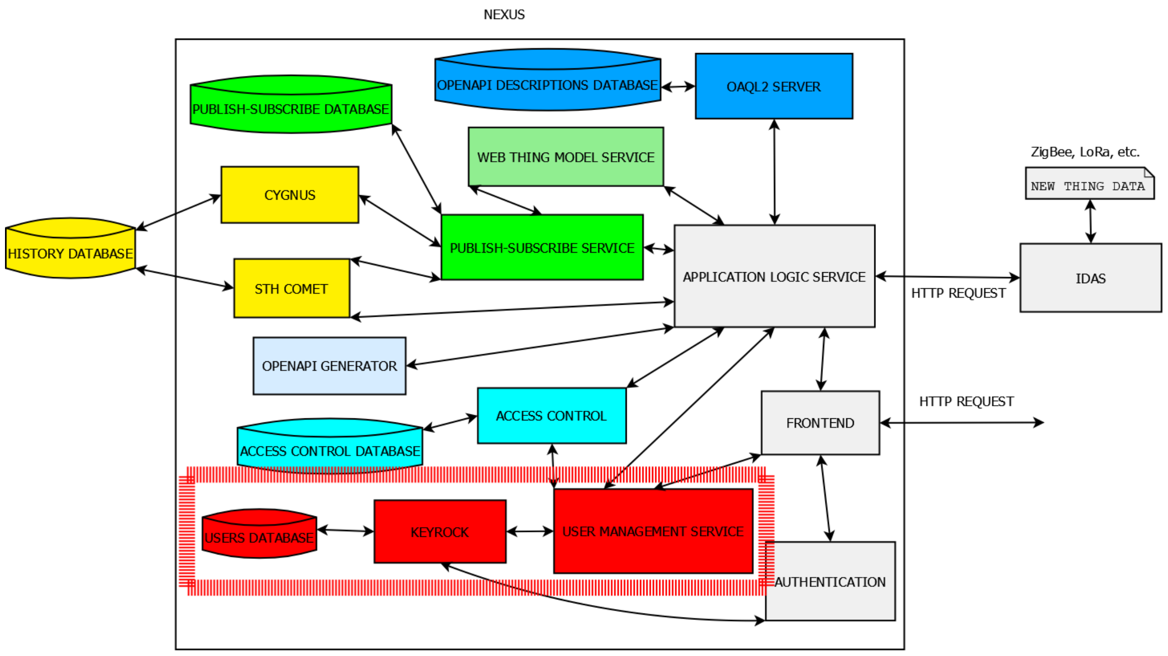


Figure 3.8: User IDM

These services are the gate of the system. They are responsible for user registration and authentication. During registration, users provide their personal information, including username, email, password and role. To generate a token for the user, the user must provide to the system the personal email and password. If these credentials are valid, Keyrock [1] generates an OAuth2 [2] token, that encodes the user's identity. OAuth2 token is a unique identifier, that expires after a time period, that acts as the user's identity inside the system. A new token is generated every time a new session is initiated. The generated token is used in most services to check the user's Id, role and access rights. Registered users and their roles are saved in Keyrock's SQL database. Administrators, through the User Management service, can manage the system's users. They can get their personal information, update or delete them. The related parts of the architecture are shown in Figure 3.8.

[1]https://keyrock.doc.apiary.io/#reference/keyrock-api/role
[2]https://oauth.net/2/

3.6.3 Access Control

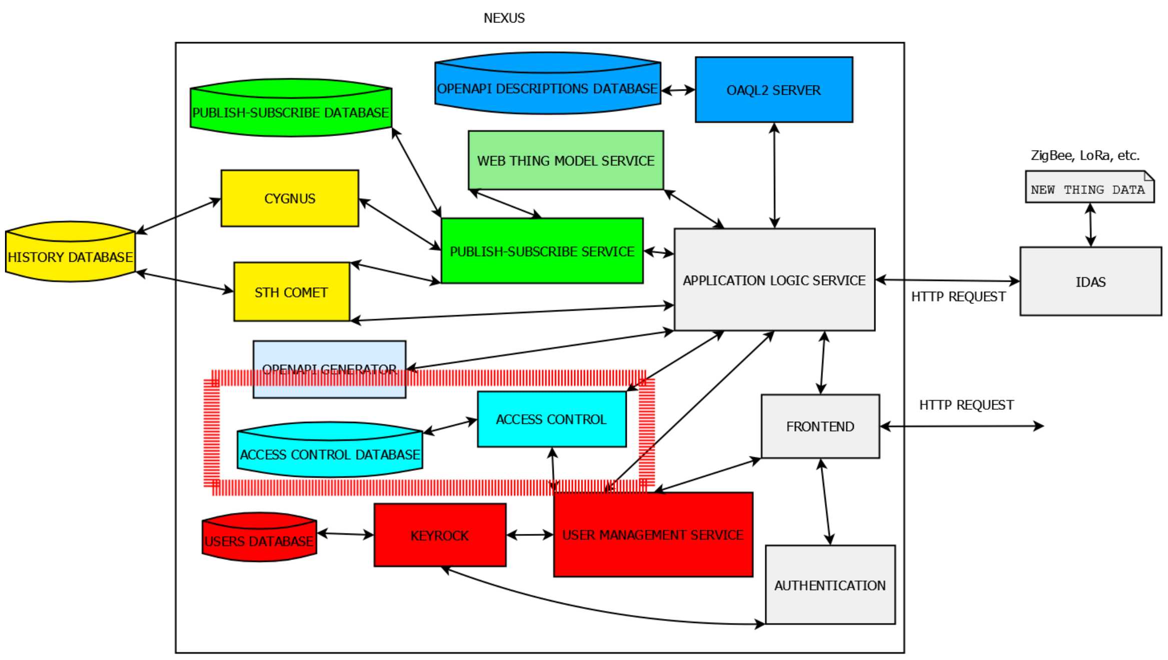


Figure 3.9: Access Control

Access Control services acts as a secondary security mechanism for the system. It holds logs of all unauthorized access attempts. Specifically, each log entry contains the id of the user that attempted to make an unauthorized request, the attempted request, the date of the request attempt and a description explaining the reason the request that unauthorized. Possible reason are the user did not have the required role to make the request, did not have the required access rights to access the Thing / Application or did not have the required subscriptions to the Things used in the Application that was attempted to store. Administrators can review the system's logs and delete them. Access Control also holds all the users' access rights. Each access rights entry contains the Id of the user that owns the Thing / Application, the Thing's / Application's URI and the confirmation status. By default, the user access rights are not confirmed and only administrators can confirm them. Administrators can view the users' access rights, update and delete them. Finally, Access Control, holds the subscriptions each user has made. Each entry contains the Id of the user that made the subscription, the Id of the Orion Entity the subscription was made on and the Id of the Orion subscription. Every time a user attempts to access a subscription, Access Control is responsible for confirming that the subscription is indeed made by that user. Also, during the storing of an OpenAPI description of an application, a check must be made to confirm that the user that attempts to store the OpenAPI description has made the required subscriptions to the Things used in the Application. Access Control provides the answer to that. The related parts of the architecture are shown in Figure 3.9.

3.6.4 Policy Enforcement Point Proxy

Services are protected by a security mechanism. This is especially important if the services is offering a public interface. This security mechanism is realised by means of Policy Enforcement Proxy (PEP) [3] service. Each service is protected by a separate PEP service. It is a responsibility of this service to approve or reject a request to the protected service. Each user request is forwarded to Application Logic service which dispatches the request to the appropriate service. The request comes with a token in its header. The PEP service will check if the token is valid by sending a request to User IDM service. If the token is valid (and the session is active), the request is approved and it is forwarded to the protected service. At this step the user's role is not taken into account in the decision to forward or reject the request. The user's role is checked inside the Application Logic code execution. Most services do not accept requests by users. Most services are accepting requests only through Application Logic. Those services are still protected by a separate PEP service. Those services are protected by a security key, referred to as master key. In this case, PEP service stores the master key. Only requests with the correct key in their header can access the protected service. In Figure 3.10 is shown how PEP Proxy works with the master key in the request header. It works exactly the same with the OAuth2 token in the header instead of the master key. PEP Proxy is mandatory only for services whose port is exposed to the outside world, like the Application Logic service.

[3]https://fiware-pep-proxy.readthedocs.io/en/latest

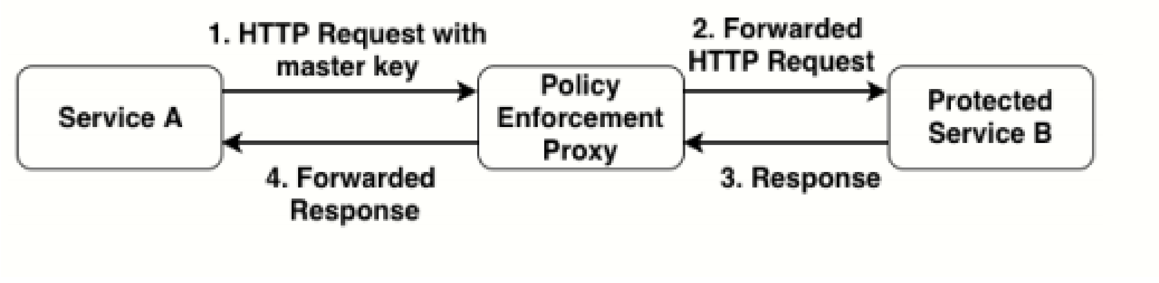


Figure 3.10: PEP Proxy

3.6.5 Publish-Subscribe

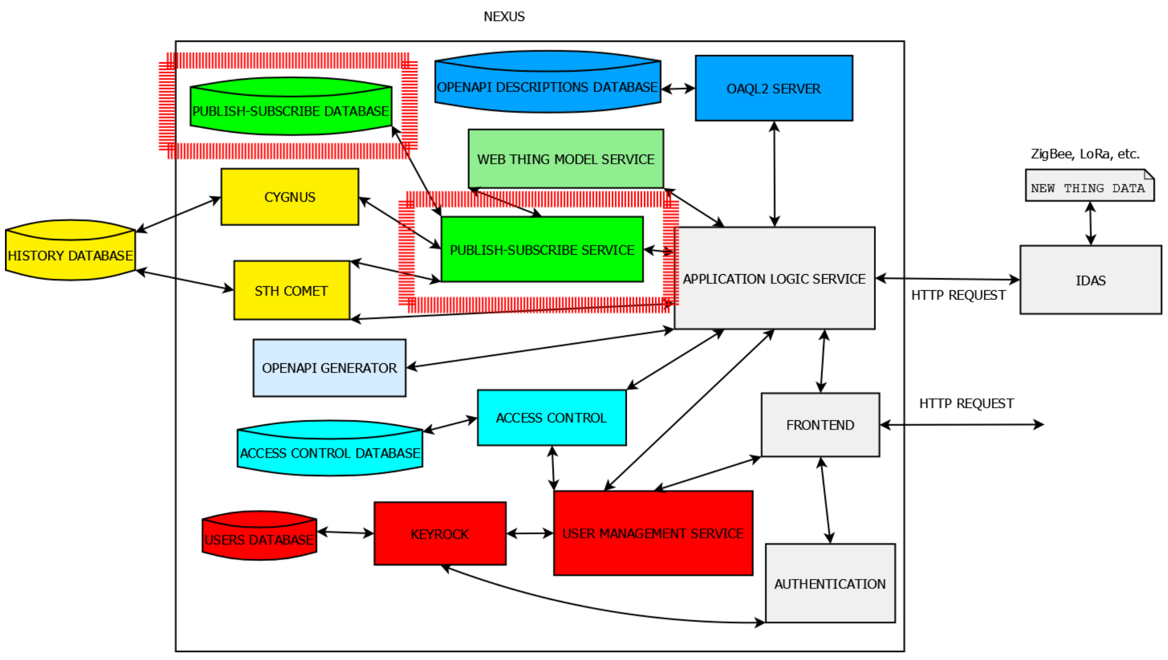


Figure 3.11: Publish Subscribe

Things and Applications stored in the system can publish information to Publish-Subscribe service (Orion Context Broker [4]). After a Thing / Application receives new information, a notification is sent to the users that made a subscription to it. Orion Context Broker holds only the most recent values of all registered entities. All Things / Applications stored in the system have an entity in Orion Context Broker that users can subscribe to. The Infrastructure Owner of the Thing and the Application Developer of the Application register the entity on

[4]https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Orion+Context+Broker

Orion Context Broker with the key-value pairs they want. However, there are some constraints. The Id of the entity must match the URI of the Thing / Application and the type of the entity must either be Thing or Application for Infrastructure Owners' Things and Application Developers' Applications respectively. Normal Users can only subscribe to Applications and subscriptions to Things can only be made by Application Developers. A typical Orion Context Broker entity contains at least the Id, a name, a description and the type. Users can make as many subscriptions as they want. Each subscription has an Id, a description, a subject, an expiration date, a status and a notification. Users can choose what the condition of the notification is. Users can update their subscriptions, disable / enable them and delete them. The related parts of the architecture are shown in Figure 3.11.

3.6.6 OAQL2 Server

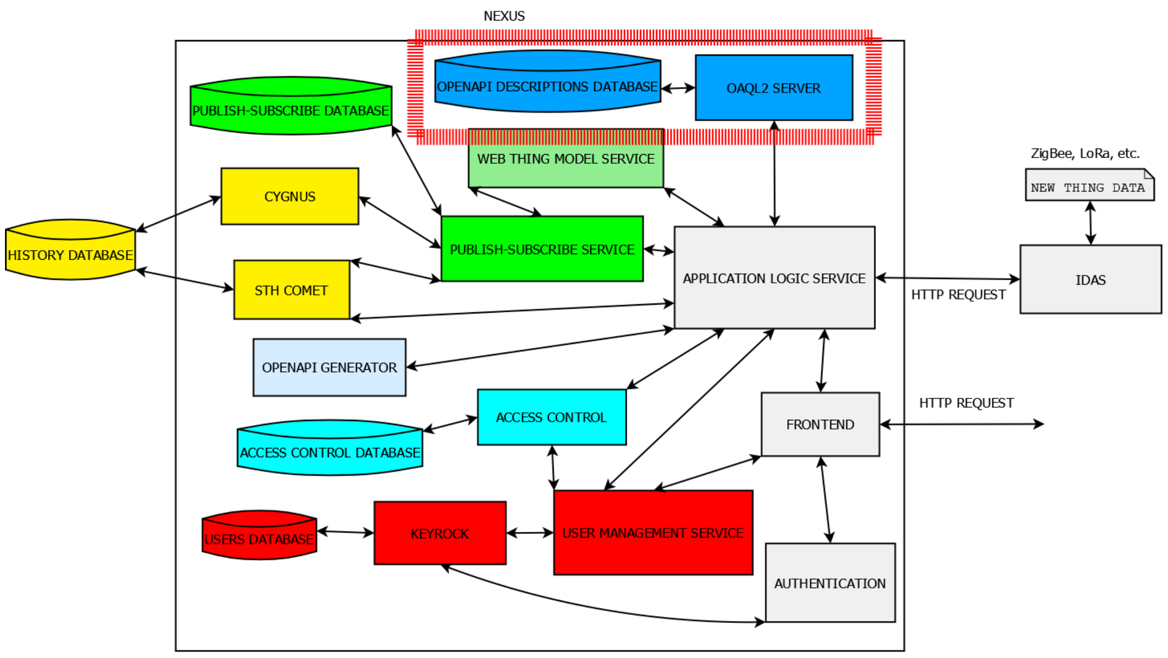


Figure 3.12: OAQL2 Server

OAQL2 Server [2] is responsible for storing the OpenAPI descriptions of the Things / Applications registered in the system and for performing queries about them. OpenAPI Query

Language 2 is a language for querying OpenAPI documents. OAQL2 has a syntax similar to SQL and supports querying most of the fields in an OpenAPI document as well as the special "x-" fields. OpenAPI descriptions, inside the info object, must contain the fields "x-type", "x-id" and "x-devicesUsed" (for Applications only). "x-type" must have either the value Thing or Application and "x-id" must have the URI of the Thing / Application. In cases of Applications, the field "x-devicesUsed" must be an array with the URIs of all the Things that it uses. The related parts of the architecture are shown in Figure 3.12.

3.6.7 History Data

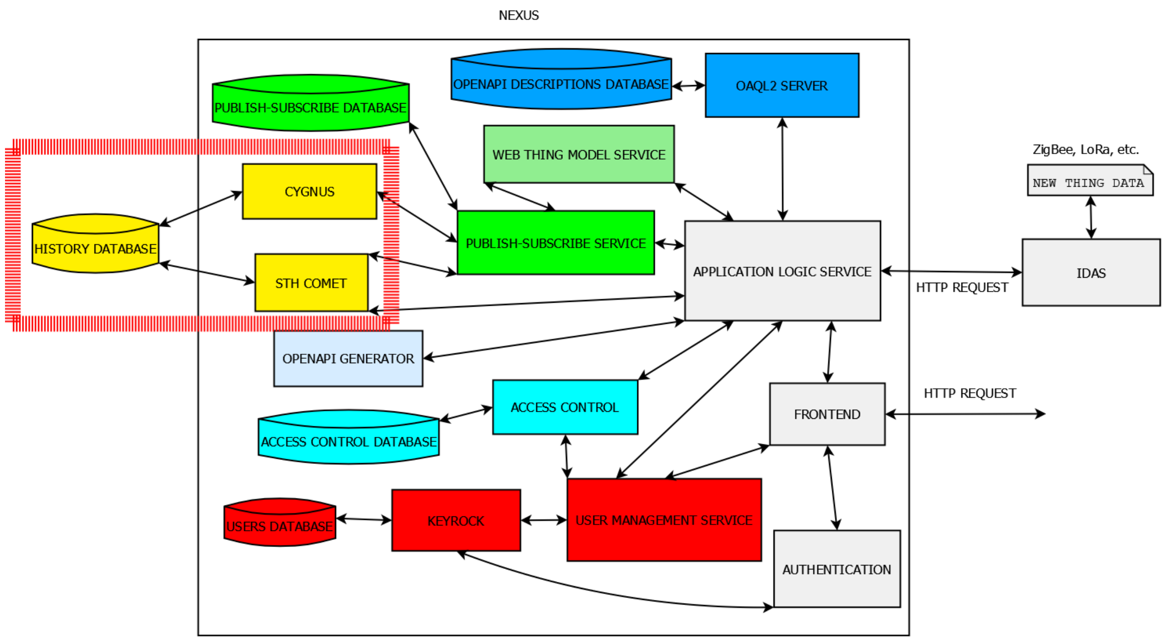


Figure 3.13: Historic Data

This services collects data flows (history values) from Orion Context Broker. The time series created from the history of data are stored in a MongoDB [5] as either raw (unprocessed) values as received from Things / Applications or aggregated (processed) values. More specifically, maximum, minimum and average values over predefined time intervals (e.g. every hour, day,

[5]https://www.mongodb.com

week etc.) are stored. This service is implemented using Cygnus [6] and STH-Comet [7]. By default no historic data is collected. Infrastructure Owners must enable the collection of historic data for their Things using either Cygnus or STH-Comet. Application Developers can make historic data queries, on Things they have subscriptions on, with the STH-Comet API. The related parts of the architecture are shown in Figure 3.13.

3.6.8 WTMs

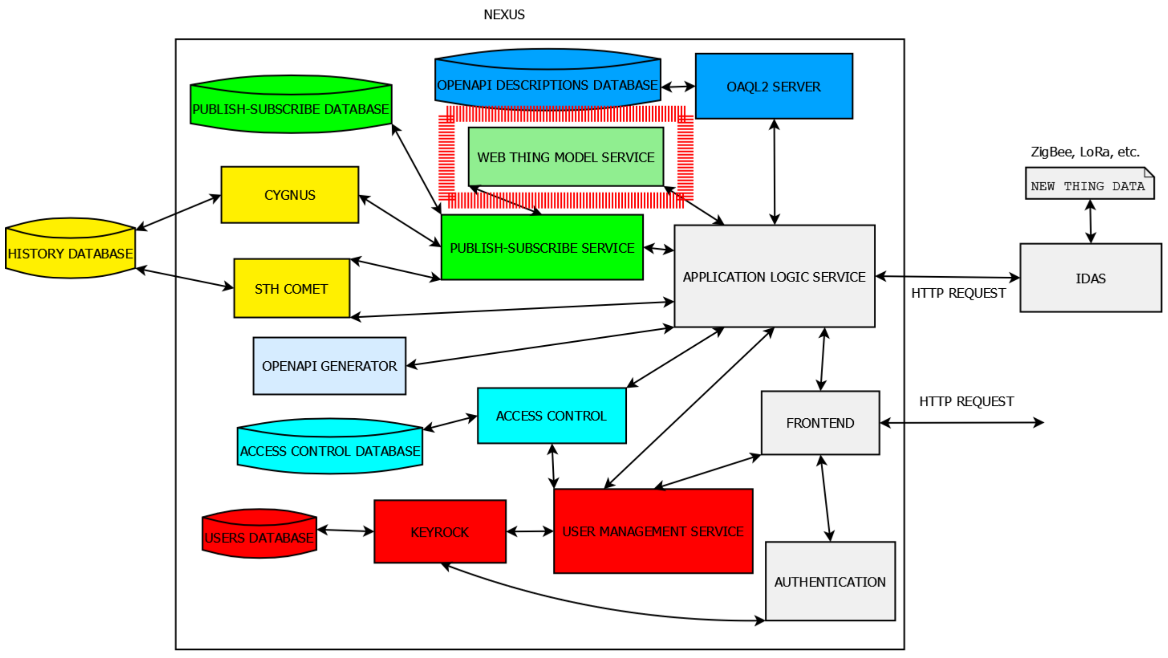


Figure 3.14: WTMs

WTMs is a tool for managing Things / Applications. Infrastructure Owners / Application developers can make requests like they would if their Thing / Application was stored in the web as a web service. WTMs uses Orion Context Broker for its implementation its MongoDB as a database. The related parts of the architecture are shown in Figure 3.14. Specifically, WTMs supports the following functions:

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

1. Register a Web Thing / Application.
2. Get a Web Thing / Application.
3. Update a Web Thing / Application.
4. Delete a Web Thing / Application.
5. Register a Web Thing's / Application's property.
6. Get a Web Thing's / Application's properties.
7. Get a Web Thing's / Application's specific property.
8. Update a Web Thing's / Application's specific property.
9. Delete a Web Thing's / Application's specific property.
10. Register a Web Thing's / Application's actions.
11. Get a Web Thing's / Application's actions.
12. Get a Web Thing's / Application's recent action executions.
13. Execute a Web Thing's / Application's action.
14. Get a Web Thing's / Application's action execution.
15. Delete a Web Thing's / Application's actions.
16. Get a Web Thing's / Application's specific subscription.
17. Delete a Web Thing's / Application's specific subscription.

3.6.9 OpenAPI Generator

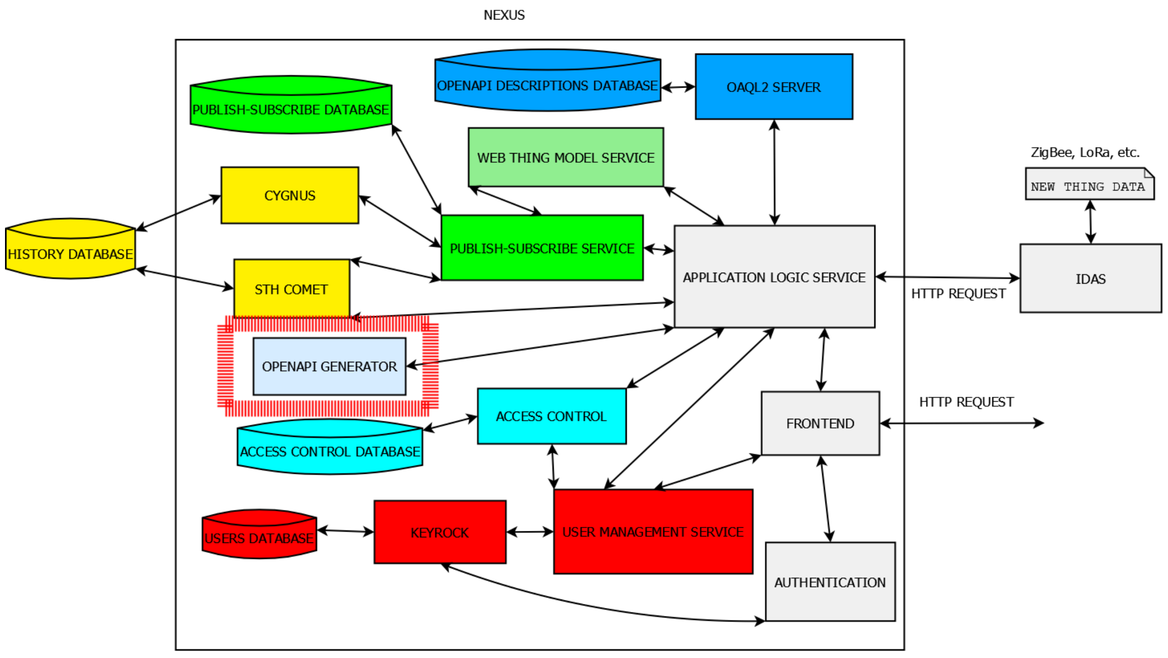


Figure 3.15: OpenAPI Generator

OpenAPI Generator is a mechanism that generates the OpenAPI description of a Thing from user input. The user input is a payload in JSON with the user settings (e.g. security settings). The user specifies also 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, in JSON format, of the Thing. The related parts of the architecture are shown in Figure 3.15. The mechanism flowchart is shown in Figure 2.6. OpenAPI Generator is explained in detail in Chapter 2.10.

Chapter 4

System Implementation

Below you can see the tables with the complete API of the system. All HTTP requests on the system are with the POST method. All payloads are in the form of "Content-type". All requests require the "Authorization": "Bearer 'user token'" header if the request was made with JavaScript or the "X-auth-token": 'user token' header if the request was made with cURL (except for the Authentication service). In the Payload Example columns you can see listings with payload examples for each endpoint. In the related listings consider that organization is the user's role.

4.1 System's Authentication API

In Table 4.1 you can see the system's Authentication API.

Table 4.1: System's Authentication API

Endpoint	Payload Type	Payload Description	Payload Example
GetAccessToken	application/json	The email and password	4.1
SignUp	application/json	The personal user information	4.2

```
{
"name": "admin@test.com",
"password": "1234"
}
```

Listing 4.1: Example payload of GetAccessToken endpoint

The payload contains the user's credentials(email and password).

```
{
"organization": "InfrastructureOwners",
"name": {
    "username": "testusername",
    "email": "testemail@t.com",
    "password": "testpassword"
}
}
```

Listing 4.2: Example payload of SignUp endpoint

The organization refers to the user's desired role and the JSON value of the name key contains the user's personal information.

4.2 System's Application Logic API

In Tables 4.2 and 4.2 you can see the system's Application Logic API. In the Roles column, consider that IO means Infrastructure Owners, AD means Application Developers, NU means Normal Users and ALL means all user roles. For the UseManagementService endpoint, all different functions are shown in a separate table.

Table 4.2: System's Application Logic API Part 1

Endpoint	Roles	Payload Type
CreateAccessRights	IO / AD	text/plain
CreateOpenAPIDescription	IO	application/json
DeleteSubscription	ALL	text/plain
DisableSubscription	ALL	text/plain
EnableHistoricDataCollection	IO / AD	application/json
EnableSubscription	ALL	text/plain
GetAllEntitySubscriptions	IO /AD	text/plain
GetAllUserSubscriptions	ALL	text/plain
GetAvailableApplications	ALL	text/plain
GetAvailableThings	AD	text/plain
GetSubscription	ALL	text/plain
GetSubscriptionStatus	ALL	text/plain
MakeApplicationSubscription	ALL	application/json
MakeHistoricDataQuery	AD	text/plain
MakeOpenAPIDescriptionQuery	ALL	text/plain
MakeThingSubscription	AD	application/json
StoreOpenAPIDescription	IO / AD	application/json
UpdateSubscription	ALL	application/json
UseManagementService	IO / AD	application/json

```
{
"openapi": "3.0.0",
"info": {
"title": "CustomerAPI",
"x-type": "Thing",
"x-id": "CustomerAPI",
"description": "This␣is␣an␣API␣for␣HPlusSport␣customers",
"contact": {
"email": "you@hplussport.com" },
"license": {
"name": "Apache␣2.0",
"url": "http://www.apache.org/licenses/LICENSE-2.0.html" },
"version": "2.0.0" },
"servers": [
{ "url": "https://virtserver.swaggerhub.com/andersonardila/
customer/2.0.0",
"description": "SwaggerHub␣API␣Auto␣Mocking" } ],
"tags": [
{ "name": "Thing",
"description": "This␣is␣a␣Web␣Thing" },
{ "name": "customer",
"description": "Customer␣related␣calls" } ]
}
```

Listing 4.3: Example payload of CreateOpenAPIDescription endpoint

The JSON user input contains the details that are specific for each Thing and not universal, like the info and servers object. The info field must contain the x-type and x-id fields so that the OpenAPI description created will be compliant with the rest of the Architecture.

Table 4.3: System's Application Logic API Part 2

Endpoint	Payload Description	Payload Example
CreateAccessRights	The Id of the Thing / Application	'ThingId'
CreateOpenAPIDescription	The user settings	4.3
DeleteSubscription	The Id of the subscription	'SubscriptionId'
DisableSubscription	The Id of the subscription	'SubscriptionId'
EnableHistoricDataCollection	The subscription payload	4.4
EnableSubscription	The Id of the subscription	'SubscriptionId'
GetAllEntitySubscriptions	The Id of the entity	'ThingId'
GetAllUserSubscriptions	Nothing	–
GetAvailableApplications	Nothing	–
GetAvailableThings	Nothing	–
GetSubscription	The Id of the subscription	'SubscriptionId'
GetSubscriptionStatus	The Id of the subscription	'SubscriptionId'
MakeApplicationSubscription	The subscription payload	4.5
MakeHistoricDataQuery	The url query	4.6
MakeOpenAPIDescriptionQuery	The query	4.7
MakeThingSubscription	The subscription payload	4.8
StoreOpenAPIDescription	The OpenAPI description	4.9
UpdateSubscription	The updated subscription	4.10
UseManagementService	The request details	4.11

```
{
"description": "A␣historic␣subscription␣on␣STH−Comet",
"subject": {
"entities": [
{ "idPattern": "{ThingId}" } ] },
"notification": {
"httpCustom": {
"url": "http://sth−comet:8666/notify",
"headers": {
"fiware−service": "{ThingId}",
"fiware−servicepath": "/{ThingId}/" } },
"attrsFormat": "legacy" }
}
```

Listing 4.4: Example payload of EnableHistoricDataCollection endpoint

The historic subscription JSON payload contains a description that explains the purpose of the subscription, a subject which contains the target Thing that the historic subscription is made on and a notification which contains the historic data API endpoint. The notification should be httpCustom, the url should be the one shown in the listing and the headers must contain fiware-service and fiware-servicepath with the followed values so that the historic values will be stored in target specific sub-directory in the history database.

```
{
"description": "Application␣subscription",
"subject": {
"entities" [
{ "id": "CustomerAPIApplication",
"type": "Application" } ],
"condition": {
"attrs": [
"employees" ] } },
"notification": {
"http": {
"url": "http://myserver:80/GetSubscriptionNotification" },
"attrs": [
"employees" ] },
"expires": "2030-01-1T14:00:00.00Z",
"throttling": 3
}
```

Listing 4.5: Example payload of MakeApplicationSubscription endpoint

The subscription JSON payload contains a description that explains the purpose of the subscription, a subject which contains the target Application that the subscription is made on and a notification which contains the endpoint that the user wants the notifications to be sent to. It also contains an expiration date and potentially throttling. It is important that the entities section contains only 1 Id and it is of type Application.

```
'http://sth-cometproxy:1035/STH/v1/contextEntities/type/Thing/
id/CustomerAPI/attributes/employees?hlimit=20'
```

Listing 4.6: Example payload of MakeHistoricDataQuery endpoint

The query is made through the historic data API endpoint url. The url should be like the one in the listing. After the endpoint it should include /type followed by the entity's type,

/id followed by the entity Id and then finally the /attributes section that contains the query. Fiware's STH-Comet website API [1] contains a detailed explanation of the queries.

```
'SELECT * FROM Service s WHERE s.x-id = "{ThingId}" '
```

Listing 4.7: Example payload of MakeOpenAPIDescriptionQuery endpoint

The payload is just the query. All the queries that can be made are explained in [2] in great detail.

```
{
"description": "Thing subscription",
"subject": {
"entities" [
{ "id": "CustomerAPI",
"type": "Thing" } ],
"condition": {
"attrs": [
"employees" ] } },
"notification": {
"http": {
"url": "http://myserver:80/GetSubscriptionNotification" },
"attrs": [
"employees" ] },
"expires": "2030-01-1T14:00:00.00Z",
"throttling": 3
}
```

Listing 4.8: Example payload of MakeThingSubscription endpoint

[1]https://fiware-sth-comet.readthedocs.io/en/latest/index.html

The subscription JSON payload contains a description that explains the purpose of the subscription, a subject which contains the target Thing that the subscription is made on and a notification which contains the endpoint that the user wants the notifications to be sent to. It also contains an expiration date and potentially throttling. It is important that the entities section contains only 1 Id and it is of type Thing.

```
{
"id": "{ThingId}",
"openapidescription": {OpenAPI Description in JSON format}
}
```

Listing 4.9: Example payload of StoreOpenAPIDescription endpoint

The JSON payload contains the Id of the Thing / Application and its OpenAPI Description.

```
{
"subid": "{SubscriptionId}",
"payload": {
"description": "Updated␣Description" }
}
```

Listing 4.10: Example payload of UpdateSubscription endpoint

The JSON payload contains the Id of the subscription to be updated and the payload that contains the attributes of the subscription to be updated (ex. the description).

```
{
"url": "{ThingId}/",
"method": "GET",
"containspayload": "false"
}
```

Listing 4.11: Example payload of UseManagementService endpoint

The JSON payload contains the http request parameters for the WTMs. Specifically it contains the url endpoint of the Thing / Application, the method and the payload for non-GET methods. The WTMs payloads are explained in great detail in Tables 4.4 and 4.4

4.3 System's Administrators API

In Tables 4.3 and 4.3 you can see the system's Administrator API. All requests have payload in application/json format. For all User Management service requests, the xtoken generated by the admin, must be included. For the Access Control queries, the options value can determine whether all entries are affected or a specific one that satisfies the values of the other fields.

```
{
"name": "admin@test.com",
"password": "1234"
```

Table 4.4: System's Administrator API Part 1

Endpoint	Service	Payload Description
GetAdminToken	User Management	The email and password
GetAllUsers	User Management	The admin token only
GetUserInfo	User Management	The user's Id
GetUserIdFromUsername	User Management	The user's username
UpdateUser	User Management	The user's id and updates
DeleteUser	User Management	The user's id
GetOrganizationId	User Management	The role's name
CheckOrganizationMembership	User Management	The user's and role's Ids
GetLogs	Access Control	The query and the query options
DeleteLogs	Access Control	The query and the query options
GetAccessRights	Access Control	The query and the query options
UpdateAccessRights	Access Control	The query
DeleteAccessRights	Access Control	The query and the query options

Table 4.5: System's Administrator API Part 2

Endpoint	Payload Example
GetAdminToken	4.12
GetAllUsers	4.13
GetUserInfo	4.14
GetUserIdFromUsername	4.15
UpdateUser	4.16
DeleteUser	4.17
GetOrganizationId	4.18
CheckOrganizationMembership	4.19
GetLogs	4.20
DeleteLogs	4.21
GetAccessRights	4.22
UpdateAccessRights	4.23
DeleteAccessRights	4.24

```
}
```

Listing 4.12: Example payload of GetAdminToken endpoint

The JSON payload contains the Administrator's credentials used for acquiring an xtoken.

```
{
"xtoken": "{xtoken}"
}
```

Listing 4.13: Example payload of GetAllUsers endpoint

The JSON payload contains the Administrator's xtoken.

```
{
"xtoken": "{xtoken}",
"id": "{UserId}"
}
```

Listing 4.14: Example payload of GetUserInfo endpoint

The JSON payload contains the Administrator's xtoken and the Id of the user whose personal information the Administrator wants to get.

```
{
"xtoken": "{xtoken}",
"username": "{Username}"
}
```

Listing 4.15: Example payload of GetUserIdFromUsername endpoint

The JSON payload contains the Administrator's xtoken and the username of the user whose Id the Administrator wants to find.

```
{
"xtoken": "{xtoken}",
"userid": "{UserId}",
"user": {
"username": "newusername",
"email": "newemail@t.com",
"password": "newpassword" }
}
```

Listing 4.16: Example payload of UpdateUser endpoint

The JSON payload contains the Administrator's xtoken, the Id of the user that is to be updated and a user object that contains the updated user information.

```
{
"xtoken": "{xtoken}",
"userid": "{UserId}"
}
```

Listing 4.17: Example payload of DeleteUser endpoint

The JSON payload contains the Administrator's xtoken and the Id of the user to be deleted.

```
{
"xtoken": "{xtoken}",
"orgname": "InfrastructureOwners"
}
```

Listing 4.18: Example payload of GetOrganizationId endpoint

The JSON payload contains the Administrator's xtoken and the name of the users' role.

```
{
"xtoken": "{xtoken}",
"orgid": "{RoleId}",
"userid": "{UserId}"
}
```

Listing 4.19: Example payload of CheckOrganizationMembership endpoint

The JSON payload contains the Administrator's xtoken, the Id of the users' role and the Id of the user whose role is checked.

```
{
"userid": "{UserId}",
"request": "MakeThingSubscription",
"date": "14/10/2022",
"authorized": "false",
"description": "User␣did␣not␣have␣the␣required␣role...",
"options": "specific"
}
```

Listing 4.20: Example payload of GetLogs endpoint

The JSON payload contains the values that the query should satisfy and the options that state whether the query should return a specific entry or all entries.

```
{
"userid": "{UserId}",
"request": "MakeThingSubscription",
"date": "14/10/2022",
"authorized": "false",
"description": "User did not have the required role ...",
"options": "all"
}
```

Listing 4.21: Example payload of DeleteLogs endpoint

The JSON payload contains the values that the query should satisfy and the options that state whether the query should delete a specific entry or all entries.

```
{
"userid": "{UserId}",
"entityid": "{EntityId}",
"confirmed": "false",
"options": "all"
}
```

Listing 4.22: Example payload of GetAccessRights endpoint

The JSON payload contains the values that the query should satisfy and the options that state whether the query should return a specific entry or all entries.

```
{
"userid": "{UserId}",
"entityid": "{EntityId}",
"confirmed": "false",
"newconfirmed": "true"
}
```

Listing 4.23: Example payload of UpdateAccessRights endpoint

The JSON payload contains the values that the query should satisfy and the updated value for the attribute confirmed.

```
{
"userid": "{UserId}",
"entityid": "{EntityId}",
"confirmed": "false",
"options": "specific"
}
```

Listing 4.24: Example payload of DeleteAccessRights endpoint

The JSON payload contains the values that the query should satisfy and the options that state whether the query should delete a specific entry or all entries.

4.4 System's UseManagementService API endpoint

In Tables 4.4 and 4.4 you can see examples for the UseManagementService endpoint. The payload must always be in JSON format and contain the following key values pairs:

- "url": Mandatory.
- "method": Mandatory with one of the following values (POST, GET, PUT, DELETE).
- "containspayload": Mandatory with one of the following values (true, false).
- "payload: Optional. Only if the "containspayload" value is true.

Table 4.6: System's UseManagementService API endpoint Part 1

url	method	containspayload
{EntityId}/	POST	true
{EntityId}/	GET	false
{EntityId}/	PUT	true
{EntityId}/	DELETE	false
{EntityId}/properties	POST	true
{EntityId}/properties	GET	false
{EntityId}/properties/{PropertyId}	GET	false
{EntityId}/properties/{PropertyId}	PUT	true
{EntityId}/properties/{PropertyId}	DELETE	false
{EntityId}/actions	POST	true
{EntityId}/actions	GET	false
{EntityId}/actions/{ActionId}	GET	false
{EntityId}/actions/{ActionId}	POST	true
{EntityId}/actions/{ActionExecutionId}	GET	false
{EntityId}/actions	DELETE	false
{EntityId}/subscriptions/{SubscriptionId}	GET	false
{EntityId}/subscriptions/{SubscriptionId}	DELETE	false

```
{
"url": "{EntityId}/",
"method": "POST",
"containspayload": "true",
"payload": {
"id": "CustomerAPI",
"type": "Thing",
"name": "Customer␣API",
"description": "This␣is␣an␣API␣for␣HPlusSport␣customers",
"createdAt": { "type": "string", "format": "date−time" },
"updatedAt": { "type": "string", "format": "date−time" },
"employees": 30,
"customers": 120
}
}
```

Listing 4.25: Example payload of CreateEntity

The JSON payload contains the url of the entity to create, the method of the request and a payload that contains the details of the entity, including its name, type, Id and description.

Table 4.7: System's UseManagementService API endpoint Part 2

url	Full Payload
{EntityId}/	4.25
{EntityId}/	4.26
{EntityId}/	4.27
{EntityId}/	4.28
{EntityId}/properties	4.29
{EntityId}/properties	4.30
{EntityId}/properties/{PropertyId}	4.31
{EntityId}/properties/{PropertyId}	4.32
{EntityId}/properties/{PropertyId}	4.33
{EntityId}/actions	4.34
{EntityId}/actions	4.35
{EntityId}/actions/{ActionId}	4.36
{EntityId}/actions/{ActionId}	4.37
{EntityId}/actions/{ActionExecutionId}	4.38
{EntityId}/actions	4.39
{EntityId}/subscriptions/{SubscriptionId}	4.40
{EntityId}/subscriptions/{SubscriptionId}	4.41

```
{
"url": "{EntityId}/",
"method": "GET",
"containspayload": "false"
}
```

Listing 4.26: Example payload of GetEntity

The JSON payload contains the url of the entity to get and the method of the request.

```
{
"url": "{EntityId}/",
"method": "PUT",
"containspayload": "true",
"payload": {
"employees": {
"value": 60 }
}
}
```

Listing 4.27: Example payload of UpdateEntity

The JSON payload contains the url of the entity to update, the method of the request and a payload that contains the details of the entity to be updated.

```
{
"url": "{EntityId}/",
"method": "DELETE",
"containspayload": "false"
}
```

Listing 4.28: Example payload of DeleteEntity

The JSON payload contains the url of the entity to delete and the method of the request.

```
{
"url": "{EntityId}/properties",
"method": "POST",
"containspayload": "true",
"payload": {
{
```

```
"id": "CustomerAPI\_testproperty",
"type": "property",
"entity": "CustomerAPI",
"name": "TestProperty",
"property": "Test",
"values": {
"test": "25",
"timestamp": "2020−06−14T14:30:00.000Z" }
}
}
```

Listing 4.29: Example payload of CreateEntityProperty

The JSON payload contains the url of the entity whose properties are created, the method of the request and a payload that contains the details of the entity's properties entity, including its name, entity, type, Id and attributes.

```
{
"url": "{EntityId}/properties",
"method": "GET",
"containspayload": "false"
}
```

Listing 4.30: Example payload of GetEntityProperties

The JSON payload contains the url of the entity's properties to get and the method of the request.

```
{
```

```
"url": "{EntityId}/properties/{PropertyId}",
"method": "GET",
"containspayload": "false"
}
```

Listing 4.31: Example payload of GetEntityProperty

The JSON payload contains the url of the entity's specific property to get and the method of the request.

```
{
"url": "{EntityId}/properties/{PropertyId}",
"method": "PUT",
"containspayload": "true",
"payload": {
"value": 30 }
}
```

Listing 4.32: Example payload of UpdateEntityProperty

The JSON payload contains the url of the entity's specific property to update, the method of the request and a payload that contains the details of the entity's property to be updated.

```
{
"url": "{EntityId}/properties/{PropertyId}",
"method": "DELETE",
"containspayload": "false"
}
```

Listing 4.33: Example payload of DeleteEntityProperty

The JSON payload contains the url of the entity's property to delete and the method of the request.

```
{
"url": "{EntityId}/actions",
"method": "POST",
"containspayload": "true",
"payload": {
"id": "CustomerAPI\_actions",
"type": "actions",
"entity": "CustomerAPI",
"name": "TestActions",
"actions": [
{ "id": "testaction1",
"name": "testname1" },
{ "id": "testaction2",
"name": "testname2" } ]
}
}
```

Listing 4.34: Example payload of CreateEntityActions

The JSON payload contains the url of the entity whose actions are created, the method of the request and a payload that contains the details of the entity's actions, including its Id, type, name and specific actions.

```
{
"url": "{EntityId}/actions",
"method": "GET",
"containspayload": "false"
}
```

Listing 4.35: Example payload of GetEntityActions

The JSON payload contains the url of the entity's actions to get and the method of the request.

```
{
"url": "{EntityId}/actions/{ActionId}",
"method": "GET",
"containspayload": "false"
}
```

Listing 4.36: Example payload of GetEntityActionRecentExecutions

The JSON payload contains the url of the entity's recent action executions to get and the method of the request.

```
{
"url": "{EntityId}/actions/{ActionId}",
"method": "POST",
"containspayload": "true",
"payload": {
"id": "acex",
"type": "execution",
"action": "CustomerAPI",
"name": "testaction1"
}
}
```

Listing 4.37: Example payload of ExecuteEntityAction

The JSON payload contains the url of the entity's action to execute, the method of the request and a payload that contains the details of the entity's action execution, including its name, type, and action.

```
{
"url": "{EntityId}/actions/{ActionExecutionId}",
"method": "GET",
"containspayload": "false"
}
```

Listing 4.38: Example payload of GetEntityActionExecution

The JSON payload contains the url of the entity's action execution to get and the method of the request.

```
{
"url": "{EntityId}/actions",
"method": "DELETE",
"containspayload": "false"
}
```

Listing 4.39: Example payload of DeleteEntityActions

The JSON payload contains the url of the entity's actions to delete and the method of the request.

```
{
"url": "{EntityId}/subscriptions/{SubscriptionId}",
"method": "GET",
"containspayload": "false"
}
```

Listing 4.40: Example payload of GetEntitySubscription

The JSON payload contains the url of the entity's subscription to get and the method of the request.

```
{
"url": "{EntityId}/subscriptions/{SubscriptionId}",
"method": "DELETE",
"containspayload": "false"
}
```

Listing 4.41: Example payload of DeleteEntitySubscription

The JSON payload contains the url of the entity's subscription to delete and the method of the request.

Chapter 5

System Performance

The purpose of the experiments is to evaluate the system's run-time performance in normal workloads and under stress.

5.1 Specs

The experiments were performed on a single VM. The VM's specs are shown below.

- OS: Ubuntu 18.04
- CPU: 6 Cores 2.8GHz
- RAM: 8192 MB
- Storage: 25GB HDD

5.2 Experiments

The stress experiments were performed with the Locust service. There were 3 experiments with 3 different number of users making parallel requests (concurrency = number of users). Each experiment featured approximately 10000 system requests, split amongst the different endpoints. The first column is the API that received the request and the other 3 columns are the system's response times in milliseconds. There are 3 different MakeOpenAPIDescriptionQuery entries on the tables. This because each query was different and satisfied different amounts of returned data. MakeOpenAPIDescriptionQuery 1 returned approximately 40% of OpenAPI descriptions stored, MakeOpenAPIDescriptionQuery 2 returned approximately 25% of OpenAPI descriptions stored and MakeOpenAPIDescriptionQuery 3 returned a single OpenAPIDescription. The experiments are shown in the tables 5.2, 5.2 and 5.2.

Table 5.1: System's Performance with concurrency 50

API	Median	90%ile	Average
CreateOpenAPIDescription	11	12	11
UseManagementService (Create Entity)	87	150	96
MakeThingSubscription	16	22	18
UseManagementService (Get Entity)	8	10	8
MakeHistoricDataQuery	10	27	15
MakeOpenAPIDescriptionQuery 1	15	22	17
MakeOpenAPIDescriptionQuery 2	14	21	16
MakeOpenAPIDescriptionQuery 3	11	18	13
StoreOpenAPIDescription	68	120	77
UseManagementService (Update Entity)	81	110	87
Aggregated	14	74	25

Table 5.2: System's Performance with concurrency 100

API	Median	90%ile	Average
CreateOpenAPIDescription	10	15	12
UseManagementService (Create Entity)	74	120	83
MakeThingSubscription	15	25	19
UseManagementService (Get Entity)	7	11	9
MakeHistoricDataQuery	9	34	16
MakeOpenAPIDescriptionQuery 1	15	25	18
MakeOpenAPIDescriptionQuery 2	13	22	15
MakeOpenAPIDescriptionQuery 3	11	18	13
StoreOpenAPIDescription	65	91	68
UseManagementService (Update Entity)	74	120	82
Aggregated	13	67	24

Table 5.3: System's Performance with concurrency 150

API	Median	90%ile	Average
CreateOpenAPIDescription	12	19	14
UseManagementService (Create Entity)	120	430	479
MakeThingSubscription	23	60	81
UseManagementService (Get Entity)	10	21	14
MakeHistoricDataQuery	7	13	9
MakeOpenAPIDescriptionQuery 1	23	72	38
MakeOpenAPIDescriptionQuery 2	19	62	33
MakeOpenAPIDescriptionQuery 3	16	53	29
StoreOpenAPIDescription	110	300	319
UseManagementService (Update Entity)	120	310	355
Aggregated	19	110	78

5.3 Results

The experiments show that the system's performance is optimal in concurrency=100. It has worse performance with concurrency 50 and a lot worse performance with concurrency=150. The slowest endpoints are the UseManagementService (for entity creation and update) and StoreOpenAPIDescription. The UseManagementService endpoint for entity creation and update is slow because it makes requests on many services (Application Logic, Access Control, User Management, WTMs, Publish-Subscribe) unlike the CreateOpenAPIDescription and MakeHistoricDataQuery endpoints which are really fast. The StoreOpenAPIDescription endpoint is also slow because for an OpenAPIDescription to be stored, multiple requests on multiple collections in MongoDB must be made. Slow endpoints are not an issue provided that the requests made on them are rare. StoreOpenAPIDescription and UseManagementService for entity creation are generally rare requests so them being slow is not a problem. The most common requests are the GET requests and the query requests. The results on the MakeOpenAPIDescriptionQuery, MakeHistoricDataQuery and UseManagementService to GET an entity endpoints are great. Finally, the system responds in real time even under stress(concurrency=150) proving that it can handle big workloads.

Chapter 6

Conclusion

Nexus is a complete IoT architecture that can support all IoT functionalities Infrastructure Owners and Application Developers potentially need. Inside Nexus, Things / Applications and described with OpenAPI which is a universal language, and industry standard, that is understandable easily by both humans and machines. Nexus contains tools for searching for stored Things / Applications and for managing them. It is also secured by design since it contains security mechanisms like OAuth2, proxies and role specific endpoints. Finally, it is a SOA and has all the advantages explained in Chapter 2.3.

6.1 Future Work

One important security update for Nexus is replacing the HTTP protocol of the requests with HTTPs for outside connections. This will secure the requests by providing encryption on the data sent. Requests between system's services can still work with standard HTTP provided that the services are in the same Cloud platform.

One other potential change is to replace Orion Context Broker with a faster Publish-Subscribe system like RabbitMQ[1]. This means that WTMs should also be changed to work independently

[1] https://rabbitmq.com/

from Orion Context Broker and a new Historic Data API should be incorporated since STH-Comet and Cygnus also have their functionality tied with Orion Context Broker.

Like the current OpenAPI Generator for Things, an OpenAPI Generator for Applications can be added in the architecture.

OpenIdConnect is a mechanism for authentication that allows the users to register and log in the system by using the token they obtained from another system. This allows them to use the system effectively without needing to use their credentials to log in every time. The system is compatible with any server authentication service provided it supports OAuth2 and its users have matching roles with the Nexus. Incorporating OpenIdConnect in the system is a nice addition.

Bibliography

[1] M. Amundsen. *Building Hypermedia APIs with HTML5 and Node.* " O'Reilly Media, Inc.", 2011.

[2] I. Apostolakis. Simple querying service for OpenAPI descriptions with Semantic Web extensions. Diploma thesis, School of Electrical and Computer Engineering, Technical University of Crete (TUC), Chania, Crete, Greece, May 2022.

[3] T. Berners-Lee, J. Hendler, O. Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[4] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible markup language (xml) 1.1-w3c recommendation. *World Wide Web Consortium. http://www. w3. org/TR/xml11/*, 2006.

[5] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, June 2007. W3C Recommendation.

[6] L. Clement, A. Hately, C. von Riegen, T. Rogers, et al. Uddi version 3.0. 2, uddi spec technical committee draft. *OASIS UDDI Spec TC*, 2004.

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

[8] D. Guinard and V. Trifa. *Building the Web of Things.* Manning Publications Co., Greenwich, CT, USA, 2016.

[9] L. Gupta. HATEOAS Driven REST APIs, Oct. 2021.

[10] S. Kaebisch, T. Kamiya, M. McCool, V. Charpenay, and M. Kovatsch. Web of Things (WoT) Thing Description, Apr. 2020. W3C Recommendation.

[11] H. Knublauch and D. Kontokostas. Shapes Constraint Language (SHACL), July 2017.

[12] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, and K. Kajimoto. Web of Things (WoT) Architecture, Apr. 2020. W3C Recommendation.

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

[14] N. Mainas and E. Petrakis. SOAS 3.0: Semantically Enriched OpenAPI 3.0 Descriptions and Ontology for REST Services. In *IEEE Intern. Conf. on Semantic Computing (ICSC 2020)*, pages 207–210, San Diego, California, Feb. 2020.

[15] N. Mainas, E. Petrakis, and S. Sotiriadis. Semantically Enriched Open API Descriptions in the Cloud. In *IEEE Intern. Conf. on Software Engineering and Service Science (IEEE ICSESS)*, pages 66–69, Beijing, China, Nov. 2017.

[16] P. Mell and T. Grance. The nist definition of cloud computing. 2011.

[17] N. Mitra and Y. Lafon. SOAP Version 1.2 Part 0: Primer (Second Edition), Apr. 2007.

[18] RedHat. An Introduction to OpenAPI Specification, 8 2009.

[19] A. Tzavaras, N. Mainas, F. Bouraimis, and E. G. Petrakis. OpenAPI Thing Descriptions for the Web of Things. In *IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2021)*, Virtual, Nov. 2021.

[20] Web Thing Model, Aug. 2015. W3C member submission.