**Technical University of Crete**
School of Electrical & Computer Engineering

# DIPLOMA THESIS

Vasileios Papadopoulos

## Flow-Based Programming support with OpenAPI in the Web of Things

**Supervisor:** Euripides G.M. Petrakis, Professor, TUC

### Examination Committee

| | | |
|---|---|---|
| Euripides Petrakis<br>Professor | Stelios Sotiriadis<br>Associate Professor | Georgios Chalkiadakis<br>Associate Professor |

**Abstract**

In the Internet of Things (IoT) people, devices and other physical objects are connected to a common network, providing services and exchanging data with one another. However, the IoT brings in complexities that stem from the fact that different protocols and standards coexist in the network, as the number of the connected Things increases. The Web of Things, or WoT, aims to extend the Internet of Things by eliminating the difficulty in communication and management of all different objects. In this paradigm, each resource exposes its functionalities as a Web service, therefore making Things a part of the web. Using standards such as REST, HTTP and URIs, the WoT concept attempts to deal with the fragmentation of technologies in the IoT and facilitate the interoperability between Things. In this work a system is proposed called Mashup of OpenAPI Nodes (MoON). MoON suggests a way of overcoming the difficulties in communication and compatibility utilizing OpenAPI. Using the OpenAPI Specification (OAS), Things in a WoT or IoT architecture can be described in detail and become readable and accessible by both humans and other Things. Due to the fact that every Thing is described according to the OAS, a uniform representation for all Things is achieved. Furthermore, in order to combine Things in a way that allows application composition, an efficient platform that will utilize the OpenAPI descriptions is necessary. MoON includes a mashup service for a Web of Things environment. Mashup is defined as a way to compose a new service from existing services. In this architecture, every object is described using OpenAPI. The result is an application that consists of a set of objects. A mechanism is introduced to generate OpenAPI descriptions for the produced applications, containing information about all Things that comprise them. The application creation process is based on Flow Based Programming tools, such as Node-RED. The generated OpenAPI Description of the applications adds additional flexibility in the application creation process, since all services, things and applications are uniformly represented using OAS and, additionally, they can be combined in any meaningful way to form new applications. This description can, also, be an input to a more complex application of a bigger scale. For instance, a smart neighbourhood's OpenAPI can be a part of a smart city application. The process is facilitated by incorporating the visual representation and code generation tool of OpenAPI with those of a flow programming engine (i.e Node Red).

## Keywords

IoT, WoT, Mashup, OpenAPI, Flow Based Programming, Thing Description, Service

Oriented Architecture, OpenAPI Generator, Node-RED, REST

## Περίληψη

Στο Διαδίκτυο των Πραγμάτων (ΔτΠ) οι άνθρωποι, οι συσκευές και άλλα αντικείμενα, είναι συνδεδεμένα σε ένα κοινό δίκτυο, παρέχοντας υπηρεσίες και ανταλλάσσοντας δεδομένα μεταξύ τους. Το ΔτΠ, επιφέρει πολυπλοκότητα, η οποία προκύπτει από τα διάφορα πρωτόκολλα και τους τρόπους επικοινωνίας, καθώς ο αριθμός των συνδεδεμένων συσκευών αυξάνεται. Ο Ιστός των Πραγμάτων (ΙτΠ), επεκτείνει το Διαδίκτυο των Πραγμάτων, εκμηδενίζοντας τις δυσκολίες όσον αφορά την επικοινωνία των συσκευών και τη διαχείριση των διαφόρων αντικειμένων. Σε αυτό το μοντέλο, οι πόροι αποκαλύπτουν τις ιδιότητές τους ως υπηρεσίες ιστού, ενσωματώνοντας, έτσι, τα Πράγματα στον ιστό. Χρησιμοποιώντας τεχνολογίες όπως REST, HTTP και URIs, ο Ιστός των Πραγμάτων στοχεύει στην αντιμετώπιση της ετερογένειας του Διαδικτύου των Πραγμάτων και στην διευκόλυνση της διαλειτουργικότητας. Η συγκεκριμένη εργασία, προτείνει ένα σύστημα που ονομάζεται Mashup of OpenAPI Nodes (MoON). Το MoON παρουσιάζει έναν τρόπο να ξεπεραστούν οι δυσκολίες που αφορούν την επικοινωνία και τη συμβατότητα αξιοποιώντας το OpenAPI. Χρησιμοποιώντας το OpenAPI Specification (OAS), τα πράγματα μίας αρχιτεκτονικής ΔτΠ ή ΙτΠ, μπορούν να περιγραφούν λεπτομερώς και εξασφαλίζεται η κατανόηση και η αλληλεπίδραση τόσο από ανθρώπους, όσο και από άλλα Πράγματα. Εξαιτίας του γεγονότος ότι κάθε Πράγμα περιγράφεται σύμφωνα με το OAS , μπορεί να επιτευχθεί ομοιομορφία κατά την αναπαράστασή τους. Προκύπτει, ακόμα, η ανάγκη για μία αποτελεσματική πλατφόρμα, στην οποία αξιοποιούνται οι περιγραφές OpenAPI για το συνδυασμό Πραγμάτων με τέτοιο τρόπο ώστε να συνθέτονται εφαρμογές. Στην παρούσα εργασία, παρουσιάζεται μια αρχιτεκτονική ενός συστήματος σύνθεσης εφαρμογών στο περιβάλλον του Ιστού των Πραγμάτων. Ως σύνθεση εφαρμογών ορίζεται η διαδικασία κατά την οποία ο συνδυασμός υπαρχόντων αντικειμένων ή υπηρεσιών (Πραγμάτων), έχει ως αποτέλεσμα τη δημιουργία νέων. Στη συγκεκριμμένη αρχιτεκτονική, κάθε Πράγμα συνοδεύεται απο την αντίστοιχη περιγραφή OpenAPI. Το αποτέλεσμα της διαδικασίας αυτής είναι μια εφαρμογή η οποία αποτελείται από ένα σύνολο αντικειμένων. Επιπρόσθετα, παρουσιάζεται ένας μηχανισμός, με τον ποίο μπορεί να παραχθεί περιγραφή OpenAPI για κάθε νέα εφαρμογή που συντίθεται και περιέχει πληροφορία που αφορά όλα τα Πράγματα που την αποτελούν. Η σύνθεση εφαρμογών βασίζεται σε εργαλεία Προγραμματισμού Ροής, όπως το Node-RED. Η παραγόμενη περιγραφή OpenAPI των εφαρμογών έχει στόχο την αύξηση της ευελιξίας δημιουργίας τους και την κατανόηση της λειτουργεικότητας τους, μέσω των διαφόρων εργαλείων που παρέχει το OpenAPI. Η περιγραφή αυτή μπορεί, επίσης, να χρησιμοποιηθεί ως είσοδος και να αποτελέσει ένα μέρος μίας πιο σύνθετης εφαρμογής μεγαλύτερης κλίμακας. Για παράδειγμα, η περιγραφή OpenAPI μίας έξυπνης γειτονιάς μπορεί να είναι μέρος μίας εφαρμογής που αφορά την έξυπνη πόλη.

**Λέξεις Κλειδιά.**
Διαδίκτυο των Πραγμάτων, Ιστός των Πραγμάτων, Σύνθεση Εφαρμογών, OpenAPI, Προγραμματισμός με χρήση ροών, Περιγραφή Πραγμάτων, Υπηρεσιοκεντρική Αρχιτεκτονική, Παραγωγή OpenAPI, Node-RED, REST

# Acknowledgements

Foremost, I would like to express my sincere gratitude to my thesis supervisor, Professor Euripides Petrakis, for his continuous support of my thesis. His constant guidance helped me at every stage of this work. Moreover, I wish to show my appreciation to all the laboratory members for the support and insight that I received. Last but not least, I am thankful to my family and friends for their unfailing moral support and continuous encouragement.

# Contents

# 1 Introduction

## 1.1 Scope of Thesis

### 1.1.1 Internet of Things & Web of Things

The Internet of Things is a network of connected devices, software and other technologies. Within this network, these objects can collect, transfer and exchange data, communicate with each other and form new services. These services can find application in many aspects of everyday life such as health, security, agriculture, transportation etc. IoT applications that are, already, widely used, include smart lightbulbs that can be switched on using a smartphone, smart sensors, smart thermostats and biometric measuring smart watches with the ability to instantly inform a hospital in case of an emergency. Moreover, by minimizing human intervention, IoT applications reduce the cost and time needed to provide such applications. Figure 1 presents a typical IoT architecture divided into layers. The bottom layer includes smart devices, such as sensors and actuators. Fogs and gateways, also known as the network layer, are responsible for transmitting the data collected by the devices, by connecting them to other objects, servers and network devices. Additionally, a certain level of data processing can be done in this level, for applications where speed is the most desired parameter. The data processing required by the IoT solution takes place, for the most part, in cloud-hosted applications. Moreover, this layer holds enough storage for extensive quantities of data and it can provide monitoring and analytics. A key challenge that IoT systems face, is the heterogeneity between the communication protocols of the connected devices, the data models for exchanging payloads, and security requirements. Also, IoT applications are usually developed for specific use cases, therefore it is difficult to extend and maintain them long-term.

The World Wide Web Consortium (W3C) [20] is an international community where organizations cooperate with developers and the public to develop Web standards. As proposed by the W3C, the Web of Things (WoT) concept, while preserving and complementing existing IoT standards and solutions, strives for enabling interoperability across IoT platforms and application domains. The Web of Things (WoT) Architecture [19] is a recommendation that is based on modular building blocks that

work together. These building blocks are the Thing Description, Binding Templates, Scripting API and Security and Privacy Guidelines. This work focuses on the Thing Description, making an attempt to implement it and improve it using OpenAPI. Objects that are connected in the Web (Things) must have a Thing Description (TD). The Thing Description is a fundamental building block of the WoT Architecture and provides a data format for describing the metadata and network-facing interfaces of Things. Thing Descriptions, by default, are encoded in a JSON format that also allows JSON-LD processing. A TD instance has five main components. The first is textual metadata about the Thing itself. The second, is a set of Interaction Affordances that indicate how to interact with the Thing. The third main component is schemas for the data exchanged with the Thing for machine-understandability. The fourth is Security Definitions to provide metadata about the security mechanisms that must be used for interactions. Finally the fifth component is Web links to express any formal or informal relation to other Things or documents on the Web.

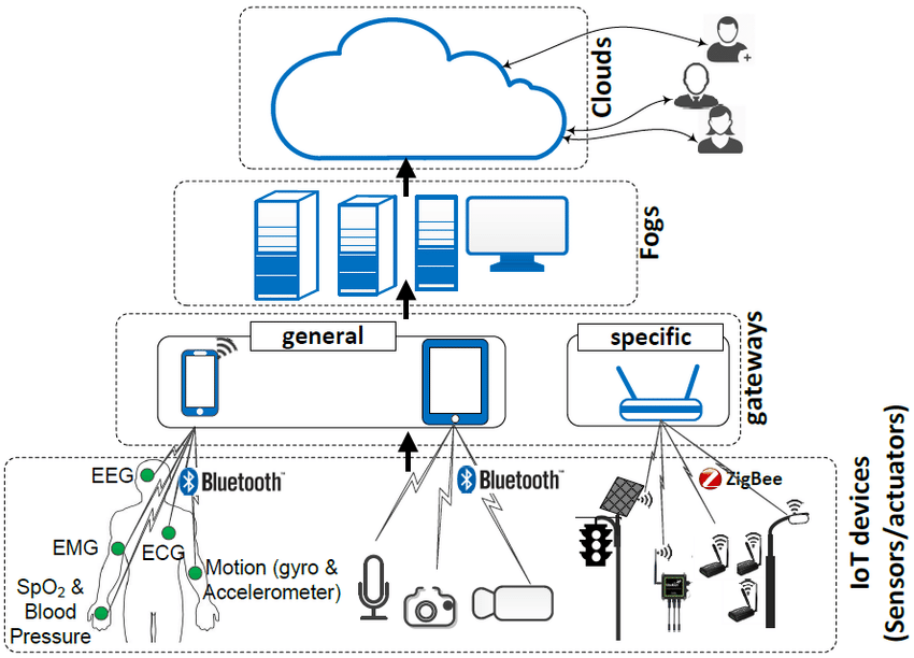



Figure 1: Layers of Internet of Things architecture

### 1.1.2 Problem Definition

As IoT systems become more prevalent, the need to standardize the technology landscape of Things arises. Typically, different vendors and manufacturers develop their own unique systems, which are usually incompatible with others, applying diversities such as data formats, APIs, communication protocols and security requirements. What is more, the process of re-using Things in new applications and handling their APIs can become extremely complex as there can be lack of documentation. One way of countering the aforementioned heterogeneity, is treating all Things as services. This idea, enables Things to be discoverable and reusable, while also allowing the developers to interact with them in a standard, predefined way. In addition to that, there is a necessity to provide an efficient platform to combine those services effectively as well as lessen the entry barrier to application creation in regards to required knowledge for developers.

## 1.2 Proposed Solution

This work proposes a system called Mashup of OpenAPI Nodes (MoON), which is a mashup system for IoT application generation by combining existing resources. Mashup is the process of creating a new service by combining existing services (i.e. Things). Flow Based Programming, is a programming paradigm that facilitates the composition of new applications without requiring specific knowledge, as most of the coding takes place in the background. In addition to that, while an OpenAPI definition is capable of fully describing a Web Thing, OpenAPI Specification (OAS) suggests a description format for REST APIs which fulfills the purpose of unifying the representation of Things as REST services. Things described by an OpenAPI document, follow an OpenAPI Thing Template [22] to achieve uniformity of representation among all the different services. Additionally, there is the ability to generate OpenAPI Descriptions for both Things and Applications, which allows for understandability using the visual representation that SwaggerHub provides, as well as consistency in the way that the services operate and are interacted with.

## 1.3 Contributions

A Mashup Service Oriented Architecture (SOA) approach is proposed that utilizes OpenAPI descriptions and streamlines the process of creating new applications from

existing Things or applications. One of its components, is a mechanism that generates OpenAPI descriptions for Things. Things that have an OpenAPI description can be treated in a common manner, as RESTful Web services. Another core component is a Flow Based Programming tool that uses these OpenAPI descriptions to compose new applications. Last but not least, an OpenAPI generator for applications is introduced, that creates descriptions, containing information about all Things that comprise them. Using their OpenAPI descriptions, both Things and applications can serve as a part of a more complex application of a larger scale. For instance, an application of a Smart Home's energy consumption can function as a part of a Smart City's energy application. The descriptions of both Things and applications are extended with x-properties to add extra fields such as Geo-location in order to provide more functionality to the users in terms of discoverability. Using MoON, a variety of applications of different scales can be created ranging from a device-to-device level up to a Smart City scenario. The OpenAPI generator for applications that is proposed, has the advantage of not requiring any knowledge regarding OpenAPI from the user, as the input to this mechanism is provided directly from Node-RED.

## 1.4 Thesis Outline

The remainder of the thesis is organized as follows.

In chapter two, background work and related technologies to this work are presented.

In chapter three, the architecture and the services that comprise the system are analysed.

In chapter four, Use Case Scenarios of the system as well as example applications are provided.

In chapter five, conclusions are presented and future work is suggested.

# 2 Background and Related Work

## 2.1 Infrastructure and Tools

### 2.1.1 Web of Things & Web of Things Architecture

The Web of Things (WoT) paradigm aims to unify the world of interconnected devices over the Web. WoT suggests that Things should expose their identity and properties on the Web so that, they can be discovered by Web search engines and be reused in applications. Following this concept, Things become part of the web and are able to communicate with each other over existing Web protocols, such as HTTP, HTTPS and Websockets. Things can be represented using data-interchange formats (JSON, XML, etc.), while their functionality is implemented using the REST architectural style.

The Web of Things Architecture is a recommendation by W3C and proposes an abstract architecture for the WoT. The WoT Architecture defines a model to describe a consumer's interaction with Things and it is based on modular building blocks that work together. These building blocks are the Thing Description, Binding Templates, Scripting API and Security and Privacy Guidelines. Thing Description provides a machine-readable data format for describing a Thing's properties and it is used to expose a Thing's metadata on the Web, so that other Things or clients can interact with it. Binding Templates aim to define network interfaces in Things for IoT protocols and ecosystems. Scripting API is an optional block that enables the implementation of the application logic of a Thing. Finally, the Security and Privacy Guidelines provide guidelines for the secure implementation and configuration of Things. Being abstract, this architecture recommendation is not specific to any particular application or communication protocols and it does not describe a specific implementation.

### 2.1.2 RESTful web services

Representational State Transfer (REST) is an architectural style for developing web services. This architectural style specifies a set of constraints and the services that satisfy these constraints are called "RESTful". RESTful web services are advantageous in terms of performance, scalability, and modifiability which ensues optimized functionality of the services on the Web. REST is based on client/server architecture while the data and functionality are considered resources. Resources can be either static or dynamic and are accessed using Uniform Resource Identifiers (URIs), typically links on the Web. To interact with resources, a set of simple, well-defined operations can be used through a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol. The supported HTTP operations are GET, to retrieve the current state of a resource, POST and PUT, to create and update a resource and finally the DELETE method to delete a resource. In the architecture proposed in this Thesis, all Things are considered RESTful Web services, hence resources and can be acted upon by using the aforementioned operations.

### 2.1.3 OpenAPI

OpenAPI Specification (OAS) [13] defines a standard, language-agnostic specification to describe REST APIs. It is a widely adopted standard 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. OAS also allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. Moreover, a valid OpenAPI document can be used by documentation generation tools, code generation tools, testing tools and more [15]. In this implementation SwaggerHub is used (Figure 3). It is an editor that provides instant visualization of an OpenAPI document able to run locally or online and expose the APIs publicly. Figure 3 is a visual representation of a REST API with a title (SmartHome), description, information about the developer and the servers, as well as various endpoints (GET, POST) grouped by tags (Web Thing, Properties, Actions).

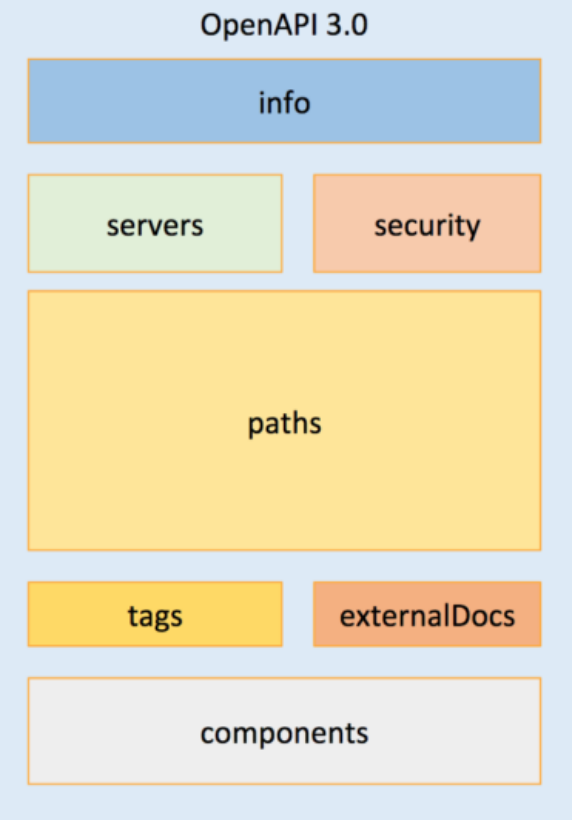



Figure 2: OAS 3.0 Structure

The basic structure of an OpenAPI document (Figure 2) consists of the following parts (objects). Firstly, the version of the OpenAPI specification, 3.0 in this case, is stated. The Info object contains metadata about the API such as the title and the description, Terms of Service, a license, contact information of the provider and the version of the service. The Servers object provides information on where the API servers are located.In this object, one or multiple host servers can be defined with a URL alongside a description for each server. The Components object holds a set of reusable objects for different aspects of the OAS. They can then be referenced in other objects or they can be linked to each other. The Security object defines security schemes for the services. These can be HTTP authentication, API keys in headers, query string or cookies, OAuth2 and OpenID Connect. These security schemes are declared in the Components unit of reusable objects. The Paths object comprises the available paths for the service's endpoints and the operations of the API. The path is appended to a URL in the Servers object in order to construct

the full URL. The Tag object is composed by additional metadata that are used by the specification, with the purpose of grouping operations. Last but not least, in the externalDocs object, additional external documentation about the service can be provided.

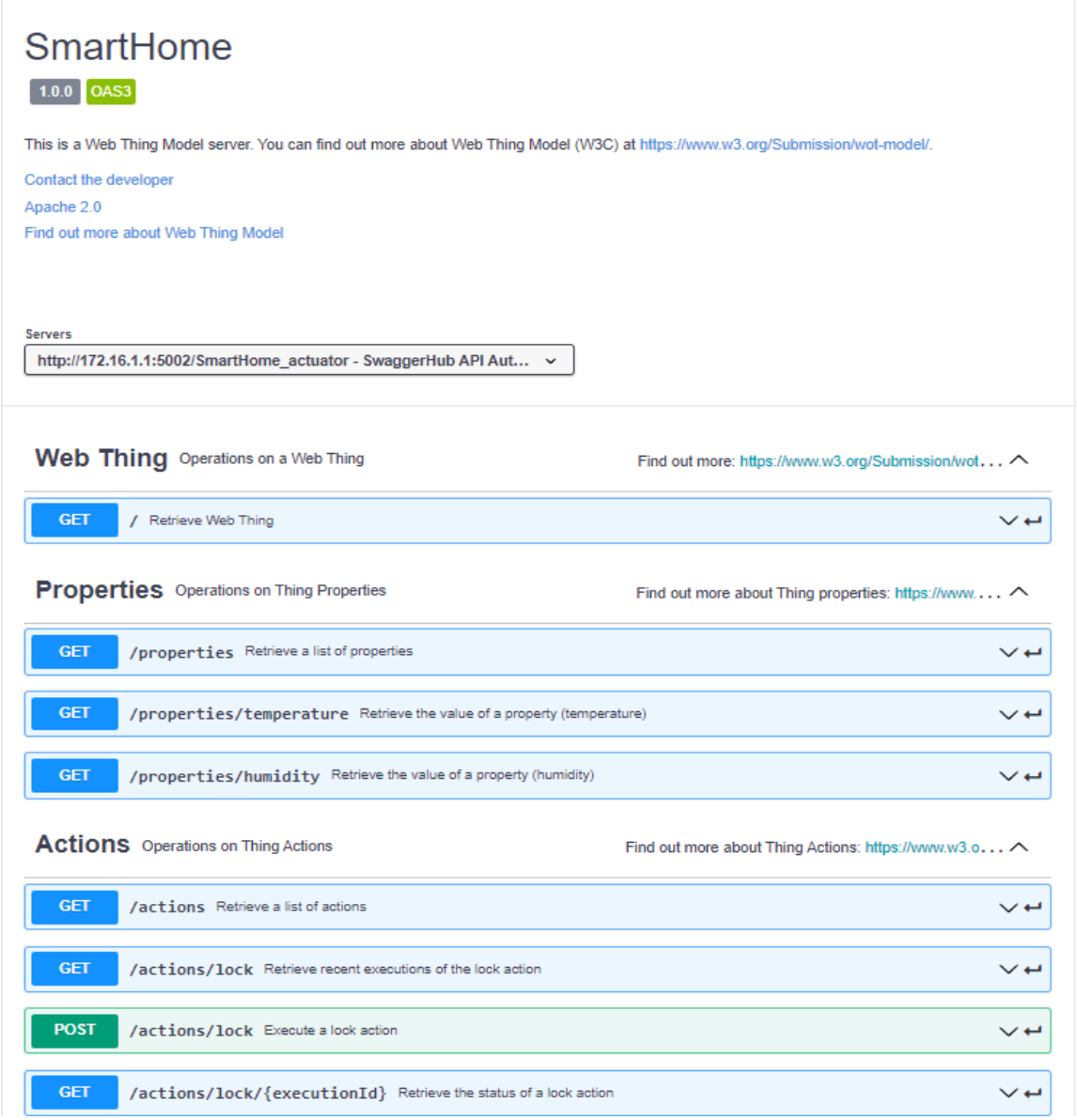



Figure 3: Visual Representation of an API in SwaggerHub

### 2.1.4 Flow Based Programming

IoT mashups compose a new service from existing services, therefore enabling users to connect devices, manage data and create personalized applications in various domains like healthcare, agriculture and home automation. For instance, in a smart city context, sensors placed in buildings and streets can collect measurements and monitor the air quality, thus making the cities friendlier to residents and visitors.

Flow-Based Programming (FBP) is a programming paradigm that defines a graphical way of creating computer applications with flows, i.e., a network of "black box" processes with well-defined ports. By connecting the output of a component with the input of another, as shown in Figure 4, data is exchanged and processed and therefore a variety of applications can be developed. A black box may implement a device or service that constitutes an application, in which way mashups are achieved. Tools that follow the FBP paradigm streamline the application creation process by providing a visual user interface, as well as reducing the technical knowledge requirements, as most of the coding takes place in the background.

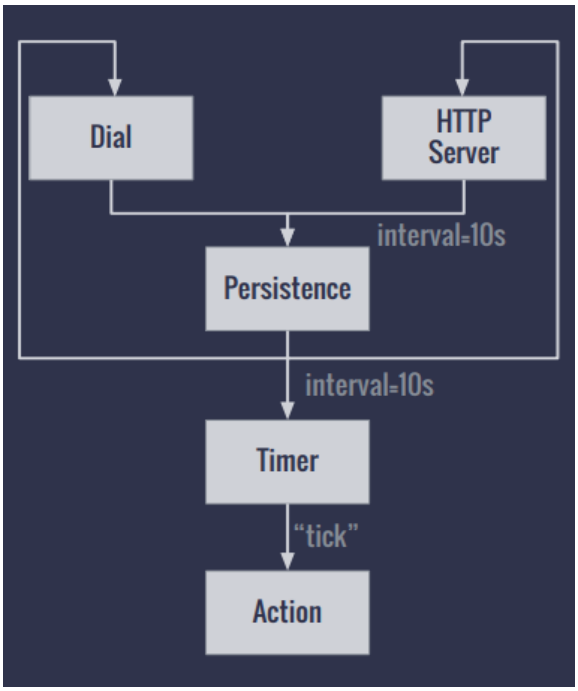



Figure 4: Example of a FBP flow

### 2.1.5 Node-RED

There are several runtime environments that approach IoT systems in a FBP manner. For instance, Flowhub [4] and NoFlo [9], CppFBP [2], WotKit [1]. The one that gained particular interest in the IoT and is used in this work, is the Node-RED [8] platform.

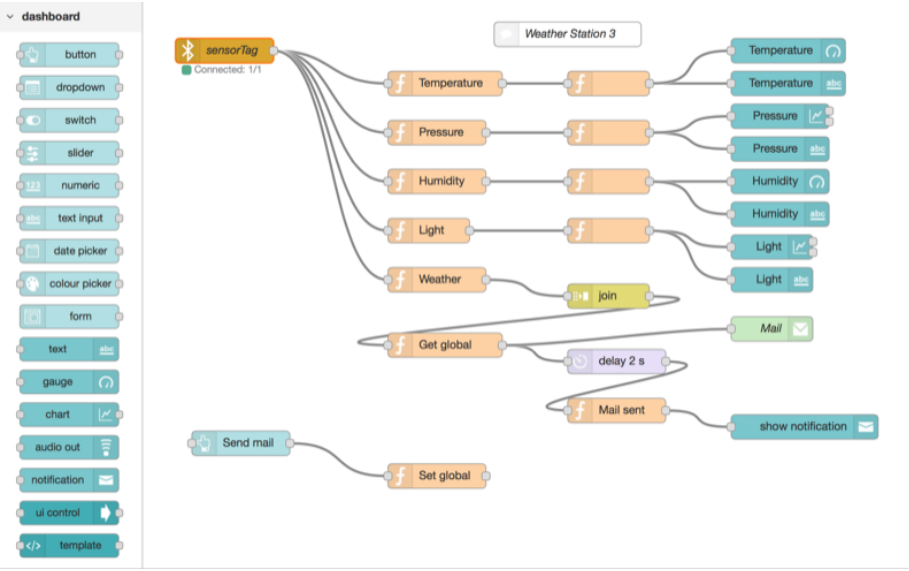



Figure 5: Example of a data flow in Node-RED

Node-RED is built on Node.js which makes it ideal to run at the edge of the network on low-cost hardware such as the Raspberry Pi as well as in the cloud. In the graphical interface, the "black box" components (nodes), can be wired together to create a new application. These nodes may contain functionality that ranges from logical operators to complete JavaScript functions. Furthermore, the code that determines the functionality of each node, is predefined and runs in the background which enables developers to create applications without requiring expertise in this particular field. After a flow (i.e. application) is formed, it can be exported in JSON file format that can also be imported into Node-RED in order to deploy the application. It is important to note that Node-RED does not export executable code, but only the JSON file that can be imported and executed in Node-RED. Node-RED's output will be discussed in detail in section 3.4.1

Figure 5 displays a sample of a flow in Node-RED. On the left side (dashboard), there is a list of the various available nodes, while on the right side some selected nodes are wired together and a programming flow is formed. In this particular example of a weather station, there is a sensor that collects data, the data gets processed in function nodes and some mails and notifications are sent according to an event.

### 2.1.6 Extra Nodes

Apart from the nodes that Node-RED provides by default, developers have the ability to contribute custom nodes which are called extra nodes. These extra nodes have more specific functionality than the basic ones and can be shared among the users in places like npm and GitHub. In this work, two extra nodes were used, the first one being openapi-red and the second one being mongodb.

#### 2.1.6.1 openapi-red Extra Node

Openapi-red [14] is an extra node, that once added to Node-RED it allows to interact with APIs that have an OpenAPI description of version 3 (OAS 3.0) and above, which are currently the latest versions. In the current architecture, every Thing (device or service) that is connected with the system is treated as a REST API and an OpenAPI description is provided. That makes it possible to interact with the Things through the Node-RED graphical environment and wire them with other nodes, such as other Things, timestamps and HTTP requests. Using this node and with the practicality that OpenAPI provides, more specifically the grouping of operations with tags and metadata, explained in section 2.1.3, it is possible for a user to get all the interactions that are available with a particular Thing according to their tags, set various parameters (e.g. JSON request bodies) and handle the outputs by connecting them to other nodes to get processed further. Figure 6 presents an example of a Smart Door's operations grouped by the "Actions" tag, while in Figure 7, an example of a JSON payload (i.e. request body) to execute a lock action is shown.

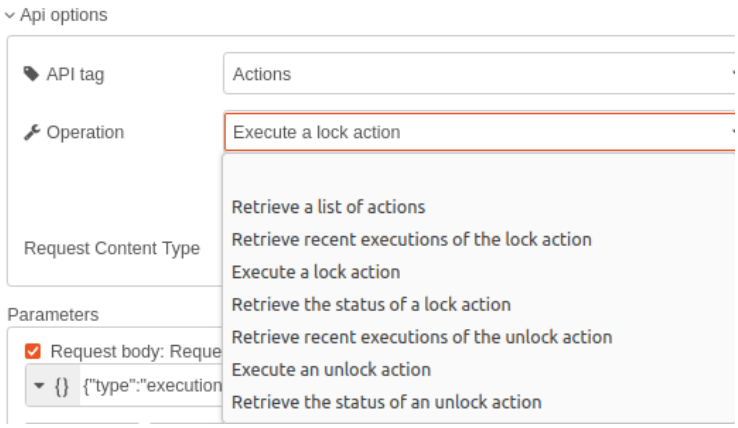



Figure 6: Example of a Smart Door Actuator in an openapi-red node

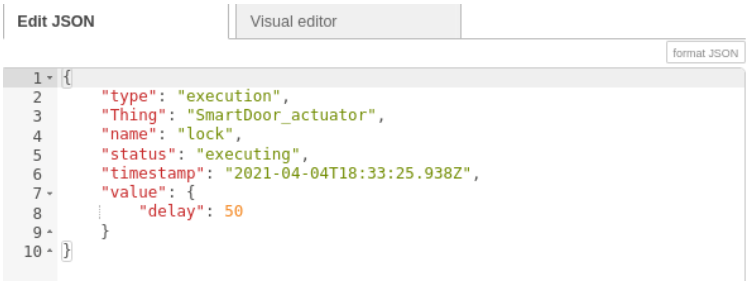



Figure 7: Example of a JSON payload to execute a lock action on a Smart Door

#### 2.1.6.2 MongoDB Extra Node

MongoDB [6] is a non-relational (NoSQL), open source database. Contrary to relational databases, MongoDB is a Dynamic Schema Document-Oriented Database that stores data in JSON-like documents. This is advantageous due to the fact that data structures are not required to have a strict format. Therefore, the same database can be used to store OpenAPI documents, sensor data and subscriptions, as befitted the architecture.

The MongoDB extra node [7] serves the purpose of connecting to a running MongoDB server through Node-RED's graphical environment. This is particularly useful

as Things and services require continuous communication with the database. For instance, Things that support subscriptions need to publish their data in the context broker's database, as well as subscribers need to read data from that database. Another example is sensors that store their measurements and, consequently, actuators that read those measurements to decide upon an action.

### 2.1.7 Publish/Subscribe Service

Publish/Subscribe messaging is an asynchronous service-to-service communication method where publishers submit data to a topic which then gets received by all subscribers of that topic. A Context Broker is responsible of handling the topics, along with the delivery of the messages to the relevant units. Using this model of communication, event-driven architectures can be achieved, as well as loose coupling between components resulting in a better performance, reliability and scalability.

The Context Broker of choice for this specific work is Orion Context Broker [16] and it is part of the FIWARE [3] platform. Figure 8 presents Orion's architecture. It is an NGSIv2 server implementation to manage context information and its availability. Orion Context Broker comes with its own MongoDB as a database to store the published data.

NGSIv2 is the API exported by Orion Context Broker, used for the integration between FIWARE platform components and by applications to update or consume context information. The core components that comprise the NGSI model are Entities, Attributes and Metadata. An entity may represent any physical or logical object, such as a device or a room. Every entity, obligatorily, has an identifier (id) and a type (e.g. sensor). What is more, entities may have one or more Attributes. Attributes describe the entity they belong to. For instance, "current_temperature" and "average_temperature" could be attributes of the entity "room". Name, type and value are the mandatory properties that define an Attribute. Attributes may also have one or more metadata objects. Metadata serves the purpose of describing the value of the attribute they are referring to. For example, metadata with the value "Celsius" can describe an attribute "current_temperature". In like manner as attributes, metadata have name, type and value fields.

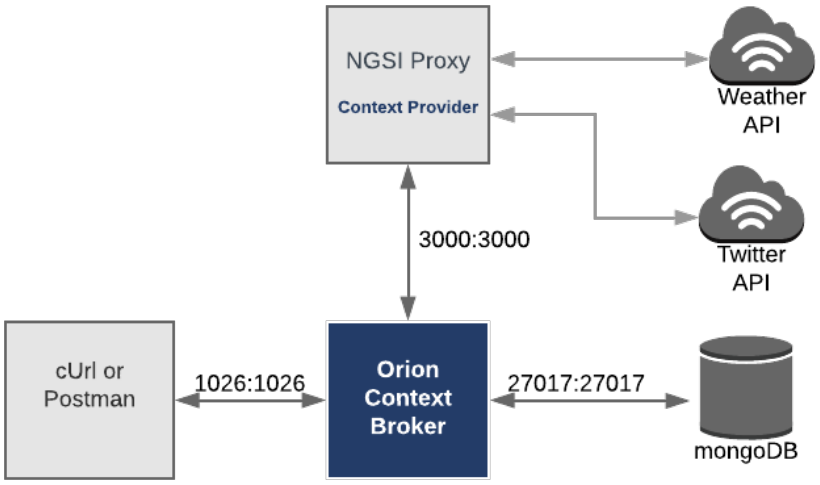



Figure 8: Orion Context Broker Architecture

### 2.1.8 Service-Oriented Architecture

Service-Oriented Architecture (SOA), defines a way to make software components reusable and interoperable, through a communication protocol over a network. Services use common interface standards and an architectural pattern so they can be rapidly incorporated into new applications. Each independent service in this architecture is responsible to execute a complete, discrete functionality which ensures loose coupling and reduces the dependencies between services and applications. By following the SOA model in this work, the services developed can be integrated into other IoT or WoT systems with minimal or no modifications.

### 2.1.9 OpenAPI Thing Template

OpenAPI Thing Template [22] seeks to provide an alternative to W3C's Thing Description that was presented in section 1.1.1. The central idea of this work is to map Thing Description properties to OpenAPI properties. The template is a valid

OpenAPI document, and consequently implements a JSON or YAML data format that is common to all things. Thing properties are described in this document, as well as actions and subscriptions in the case of Things that support those. Similar to W3C's TD, the OpenAPI Thing Description is composed of four building blocks, its resources.

The first resource, the Thing resource, provides an abstract description of a Web Thing. This can be achieved with an HTTP GET request to the root URL of the Thing. The second block, the Properties resource, defines the properties of a Thing, such as the measurements, or the state of a device. After issuing an HTTP GET request to the corresponding URL, a JSON array describing the Thing properties is returned. Following the Properties resource, the Actions resource defines the allowed actions on a Thing. This resource enables the execution of actions by issuing an HTTP POST request, the retrieval of all available actions of a Thing and all recent executions of a specific action, as well as an operation to retrieve the status of an action execution using its execution identifier with an HTTP GET request. The last building block is the Subscriptions Resource, which describes subscriptions to Web Things. This resource employs an operation to create and store new subscriptions with an HTTP POST request, an operation to retrieve a list of subscriptions made to a specific Thing or Web Thing resource, and an operation to retrieve a specific subscription using its subscription identifier using an HTTP GET request and an operation to delete a subscription using its identifier by issuing an HTTP DELETE request.

### 2.1.10 OpenAPI Thing Generator

A mechanism to generate an OpenAPI Description for a Web Thing was introduced in this work [22] [12]. In Figure 9 the flowchart of this mechanism is presented. The input consists of a JSON payload with the user settings (e.g. security settings) and the Thing characteristics that will be instantiated to the template. Furthermore, additional information that characterize the Thing and its functionality, such as available properties and actions, are declared by the user. This mechanism can be applied to any Thing, as long as its functionality can be exposed using REST, and the output is the OpenAPI description of the Thing in JSON or YAML format.

Firstly, Info, Security, Servers and Schema objects are created and appended in the OpenAPI Thing template. If the user has specified external documentation, an

External Documentation object will also be created. Next, the Thing's description payload, if provided by the user, is appended under the Webthing model object, as a Schema object, and it contains information about the device and its features. Following that, the security schemes (e.g. HTTP Authentication, OAuth2.0, OpenID Connect) are defined in the Security Requirement object. Then, the process reads a list of available servers as an array of Server objects. For each supported property or action that is defined in the input (e.g. a state property or a lock action), the corresponding paths are appended to the service description, along with a relative standard tag (e.g. Actions Tag).

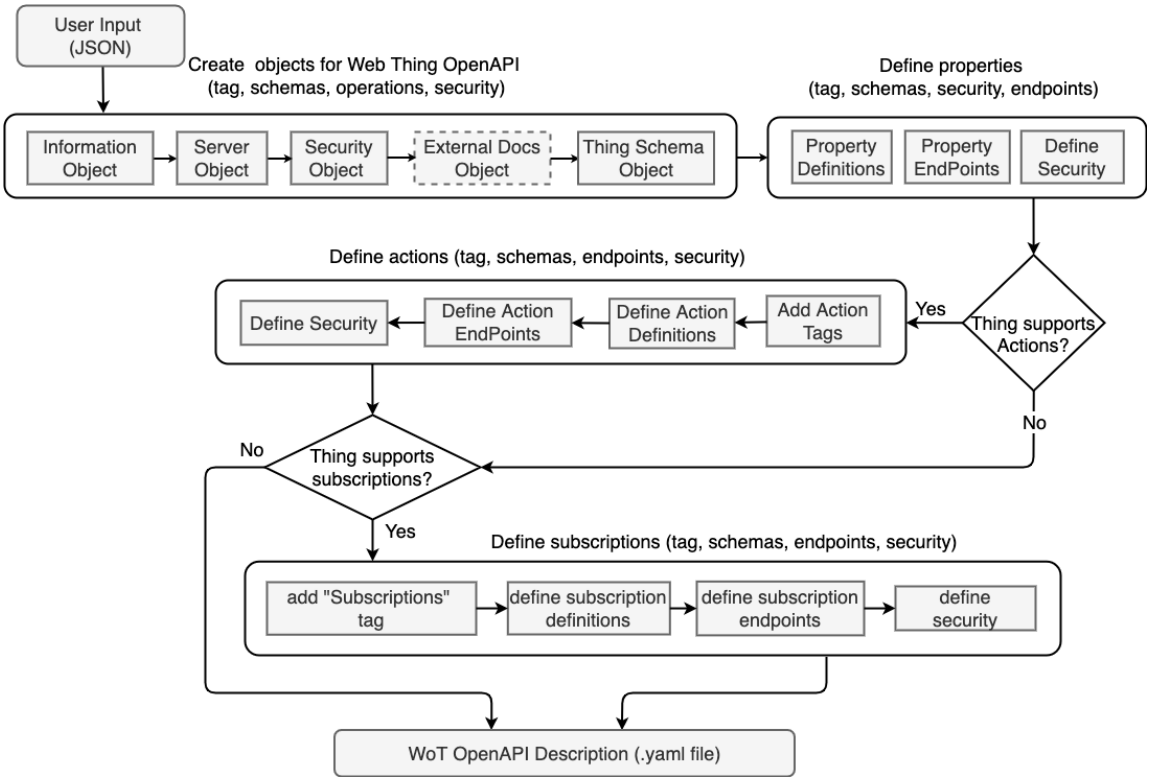



Figure 9: OpenAPI Thing Description Generator flowchart

### 2.1.11 Web Thing Model service (WTMs)

The Web Thing Model service (WTMs) [21] is an autonomous RESTful service in Python Flask and implements some of the WoT Architecture operations on Things using HTTP. It is designed to support operations for retrieving and updating Thing descriptions and their properties, as well as all Thing model operations. Moreover, it implements functions that support actions (i.e. through an actuator), retrieval of action executions and functions related to subscriptions on Web Thing resources. Essentially WTMs implements the REST API that allows consumers (i.e. users or Things) to interact with all Things that are connected to the system, since they all follow the OpenAPI Thing Template discussed in section 2.1.9.

## 2.2 Related Work

### 2.2.1 loT Mashups with the WoTKit

In this paper [1], the development of an IoT mashup platform, called the Web of Things Toolkit (WoTKit) is discussed. WotKit is a Java web application that aims to address key requirements for loT mashup developers in one system. The data model consists of sensors with fields describing either a sensor or actuator. The communication between components is implemented with Active MQ, a standard Java Messaging Service and allows for aggregating data from a variety of sensors, and passing simple control messages to actuators. Also, while certain applications are provided as core system facilities, users have the option to develop their own. In this architecture, Thing integration within an application is implemented by means of gateways, which are web clients implemented with simple scripts that register discovered sensors, collect data from the sensors they serve, and push data into the system. A flow-based programming environment based on Node-RED is provided with the toolkit where modules are connected to form pipes that process data and create new systems.

### 2.2.2 IoT Mashup as a Service: Cloud-based Mashup Service for the Internet of Things

This work [5] proposes a mashup service model, called IoT mashup as a service (IoTMaaS), based on cloud computing paradigm. The three core blocks that comprise this application are thing model, software model, and computation resource model. Also, a cloud platform on which IoTMaaS is served, is proposed. Sensors and actuators that are connected to the system, have exposed functionalities, that become sensing and actuation services and, additionally, an identifier for each thing is created. Thing model consists of an identifier, device profile, services it provides, relationship with other Things, and configuration methods. Software is defined as an assembly description of software components. Things serve the role of data sources and consumers, while software plays the role of processing instructions. Computation resource includes available CPU, RAM, storage etc. To counter the heterogeneity of platforms and protocols, it is assumed that manufacturers provide a thing service driver (TSD) component which is a representation of things in the software component model to provide the programming interface to other software of IoTMaaS. In order to create an application, a professional service planner (SP) creates templates which are then chosen by end users. Users can customize the template by selecting things and configuring their parameters. Next, the preconfigured virtual machines are launched, according to the instantiation request, which includes the IoTMaaS template, customization parameters, and thing identifiers. Then, a request is sent to the service deployer and an instance of the IoTMaaS is composed. Finally, the deployment command is sent from the service deployer and the instance begins to run, connecting to things and retrieving the data, thus providing the new service.

### 2.2.3 Webifying Heterogenous Internet of Things Devices

This paper [10] approaches the problem of enabling application development using devices with different protocols by introducing the WoTDL2API (Web of Things Description Language to API) tool. This tool employs an ontology and is capable of generating and deploying RESTful APIs for devices that are instances of this ontology using OpenAPI. Web of Things Description Language (WoTDL) ontology, is an extension of existing WoT models, able to describe IoT/WoT devices and environments. Initially, all the Things are described with the WoTDL ontology. The process begins by identifying the available devices and their capabilities for

the desired IoT scenario, which is defined by the developer, followed by a toolchain that will create an API for the devices, by transforming their WoTDL model to an OpenAPI description. Using the OpenAPI model, the OpenAPI Generator [11] is utilized to produce the REST API code, which is then deployed, and the API is run. The central *"WoTDL Hub"* is responsible for handling HTTP requests by managing the physical IoT devices at runtime. A developer can choose a set of HTTP endpoints that correspond to WoT devices to create applications "regardless of the hardware type and its communication technology".

### 2.2.4 Comparison

Comparing these works, it is apparent that each has advantages and disadvantages. WoTKit offers a full featured IoT platform with an intuitive visual Flow-Based Programming tool, giving the ability to users to develop their own applications, however it lacks a uniform representation of the devices. To compensate for that, gateways gather all device data and push it into the system. IoTMaas comes with the most complex design as it also proposes a Cloud structure on which it is executed. This system provides uniformity of representation regarding Things, as well as allows for custom applications. A notable drawback of this system is that Things are not described in a standard way, but it is assumed that manufacturers provide a Thing Service Driver (TSD) for each device. Additionally, the process of combining devices to form applications seems convoluted, as users that have the Service Planner (SP) role need to create templates that contain the application's configuration (devices, parameters, resources to allocate) in a format that is tailored specifically to this work. WoTDL2API uses OpenAPI to represent things and the users can develop applications using HTTP and REST. Nevertheless, a platform to create applications by combining existing services is missing, as developers create applications by manually choosing from "the set of HTTP endpoints provided for the WoT devices".

| *System* | **WoTKit** | **IoTMaaS** | **WoTDL2API** | **MoON** |
|---|---|---|---|---|
| *Representation* | - | TSD | OpenAPI | OpenAPI |
| *Uniformity* | No | Yes | Yes | Yes |
| *Communication* | Active MQ | Internet Exchange Client (ICE) | HTTP-REST | HTTP-REST |
| *Custom Applications* | Yes | Yes | Yes | Yes |
| *Planning* | Node-RED | IoTMaaS Template | - | Node-RED |
| *Flow-Based* | Yes | No | No | Yes |

# 3 System Architecture

## 3.1 Database

The database system used in this architecture is MongoDB. The database consists of four collections, "things", "applications", "entities" and "subscriptions". Figure 10 presents the database's structure along with the collections and the documents in each collection. The "things" collection contains OpenAPI descriptions of things, the "applications" collection contains OpenAPI descriptions of applications, the "entities" and "subscriptions" are two collections provided by the Orion Context Broker, in order to store measurements, subscriptions, device state, action executions etc.

When a new Thing is registered through user's input, it is stored in the "things" collection of the database. For each new document (i.e. Thing's OpenAPI) inserted in the collection, MongoDB automatically creates a unique identifier "_id" field. After each document's insertion, a URL is created with the identifier being the last part of this URL (e.g. http://172.16.1.1:5001/things/6238e9b5e3447e4e4d13006e, where 6238e9b5e3447e4e4d13006e is the value of the "_id" field). As a consequence of the identifier being unique, all the URLs are also unique. The URL exposes the contents of the document, which in this case is the OpenAPI Thing Description of the Thing it is referring to and will later be used in Node-RED.

The "applications" collection is responsible for storing the OpenAPI Descriptions of applications generated through the Application OpenAPI Generator (section 3.5). Similar to the Thing Descriptions, MongoDB automatically creates a unique identifier "_id" field for each new application stored. Following that, for every document in the collection a unique URL is created, where the last part of this URL is the value of the identifier. This URL exposes the OpenAPI document of the application it describes and it will be used as an input of a node (i.e. openapi-red node) in Node-RED if the developer wants to use the application as a Thing.

The "entities" and "subscriptions" collections refer to the data coming from the Context Broker. The "entities" collection stores information about NGSI entities. Each document in the collection corresponds to an entity, which can be properties (e.g. temperature, humidity, pressure), device's internal state, action executions,

recent actions or a list of available actions. The "subscriptions" collection contains subscriptions to various entities.

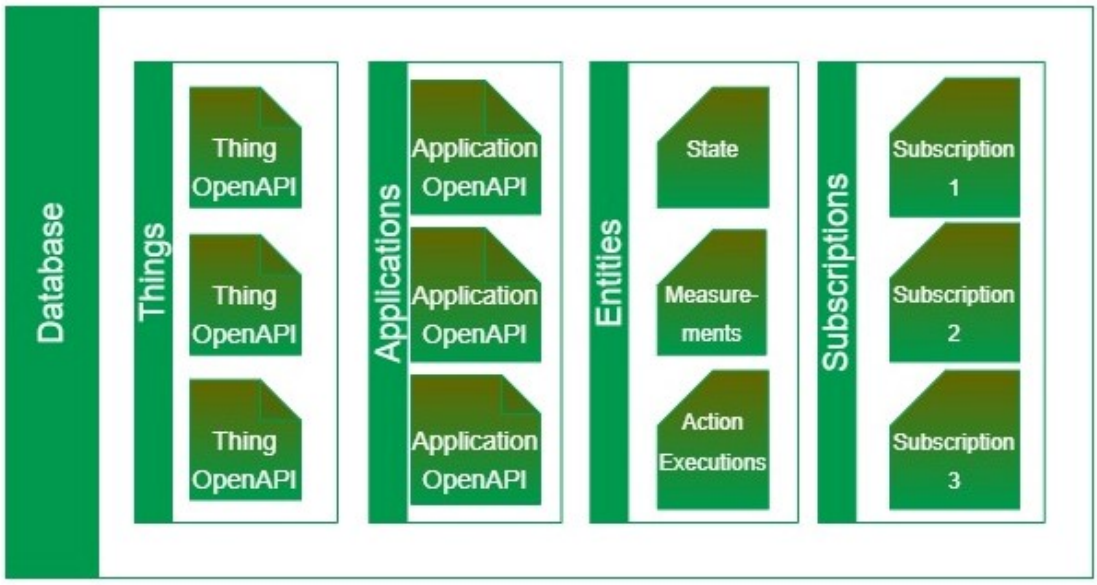



Figure 10: The structure of MongoDB Database with Collections and Documents

## 3.2 MoON's Subsystems

In this section the proposed architecture will be described, along with the components that comprise it. The three options (subsystems) that are available to the user are the following: Insert a new Thing in the system, create or execute applications in Node-RED and, finally, generate OpenAPI Descriptions for the applications.

1. **Insert a new Thing in the system**
   To insert a new Thing into the system (Figure 11a), the user's input is a JSON file, that can either be a valid OpenAPI document, or a file that follows the OpenAPI Generator constraints. In both cases, the output is the OpenAPI description of a Thing, which will be stored in MongoDB.

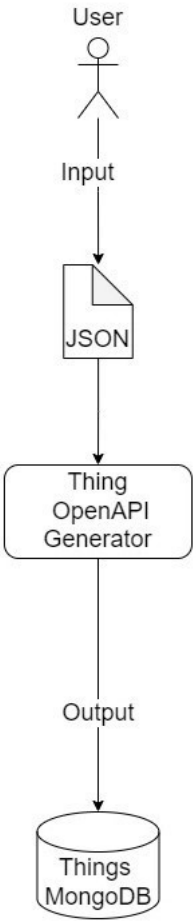
2. **Create or execute applications in Node-RED**
   "Node-RED" is the core block of the mashup service where users can create,

import or export applications (Figure 11b). To create an application, the user creates a flow of connected items, such as sensors, relational operators and computing functions, in the form of black boxes (i.e. nodes). After an application is created, it can be exported as a JSON file, which can, then, be given as user input to import it back into Node-RED in order to execute the application.

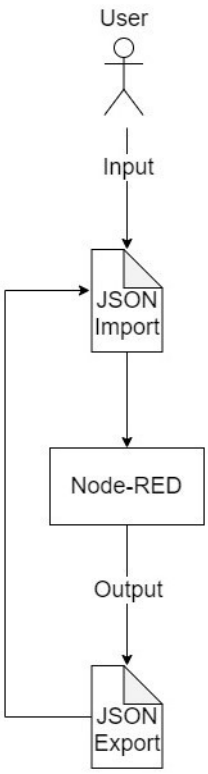
3. **Generate OpenAPI Descriptions for applications**
   User's input to the Applications OpenAPI Generator (Figure 11c) comprises a title and a short description for the application, in addition to Node-RED's JSON export (i.e. flow of the application described in a JSON file). Following that, the applications' OpenAPI Generator will produce an OpenAPI description for the application and store it in MongoDB.

The OpenAPI Descriptions of both things and applications have been extended with "x-properties". An "x-property" is a property that is not defined in the OpenAPI Specification and is added to serve a specific purpose in this architecture. The descriptions of Things have a location x-property, which lets the users discover services based on their location, while the application descriptions have been extended with an "x-devicesUsed" x-property which is an array of the services used in a specific application and enables their discovery based on this property. Figure 12 presents an example of an application that is implemented with a Motion Sensor and a Smart Lamp Actuator. The "x-devicesUsed" field contains an array of the devices that comprise this application and a user has the ability to filter applications based on this field, for example, search for applications implemented with motion sensors.

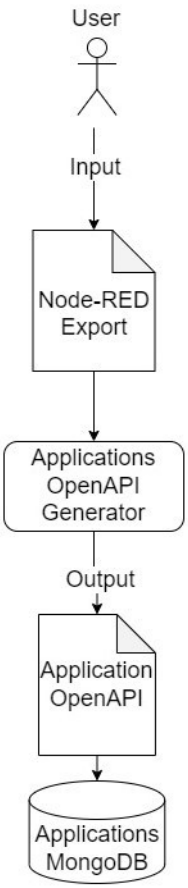
In the following sections, an analysis of the architecture's subsystems will be presented. More specifically, there will be focus on the application development process in Node-RED, the Node-RED output and the Applications OpenAPI Generator. Finally, the complete system's architecture incorporating these components will be presented.




(a) Insert a new Thing




(b) Create or execute applications in Node-RED




(c) Generate an application's OpenAPI description

Figure 11: The three options (subsystems) that are available in the system

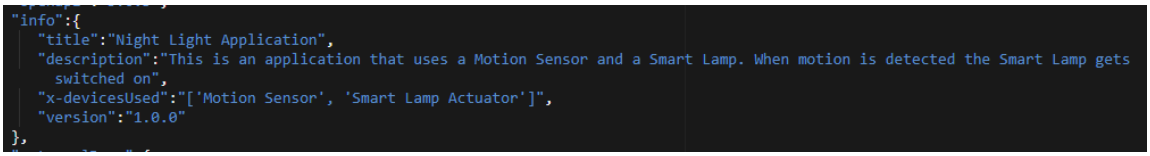



Figure 12: Example of an x-property ("x-devicesUsed") in an application's OpenAPI

## 3.3 Subsystem 1: Thing insertion

In order to insert a new Thing, the user provides as an input, either a valid OpenAPI description of a Thing, or a JSON file that follows the user input constraints of the OpenAPI Generator. If the user's input is an OpenAPI document, the OpenAPI Generator, will validate it and it will be stored in the MongoDB database. In case it is not an OpenAPI, it will be parsed by the OpenAPI Generator to produce an OpenAPI Thing Description. This JSON file must contain the Thing Description according to the Thing Template and extra characteristics of the Thing. Such characteristics are external documentation, supported actions, the type of the Thing, security settings or more necessary information to describe the service. After the OpenAPI description is generated, it will be stored in the MongoDB database. Each new Thing stored in the database is assigned with a unique URL that exposes its OpenAPI description.

## 3.4 Subsystem 2: Application Development & Execution

An application is the outcome of the combination (i.e. a flow) of nodes in Node-RED's visual programming interface. Nodes (i.e. black boxes) may implement devices, relational operators, computing functions, provide timestamps or time intervals and output messages for debugging. Additionally, there are function nodes that allow JavaScript code to be run and provide functionality that is not included in other basic nodes (e.g. a Random Number Generator). For applications that use Things such as sensors and actuators, the minimum requirement is that Node-RED communicates with the "things" collection of the database to get the Thing's OpenAPI descriptions, as well as with the "entities" collection in order to execute actions or retrieve properties (i.e. states, measurements) during run-time. For instance, an application with a sensor needs this collection to store temperature measurements, as well as read measurements to decide upon actions.

In this system, Things are implemented using their OpenAPI descriptions. A major advantage of describing Things with OpenAPI, is that they have a uniform representation. This stems from the fact that their OpenAPI descriptions follow the OpenAPI Thing Template, that is common for all Things. This allows for other components of applications to interact in a common way with all the Things that exist in the database and requires no additional knowledge from the developer when

it comes to the specific Thing's APIs, protocols or other dissimilarities.

In order to interact with a Thing, the OpenAPI description URL is given as an input in the openapi-red extra node. Figure 14 presents an example of a Thing node. The user has the choice of operations related to the selected tag. Tags group operations by type and are convenient for the user to find the desired one, choosing from a list of all the operations that the device offers. In this particular example, the Tag "Actions" is chosen and a list of available actions is shown. There is also a field where a JSON payload can be inputted which gets passed into the HTTP requests when they are issued, in case a request body is necessary. The openapi-red node, is used as an interface for the device's REST API that it is referring to, while the device's API itself is implemented using the Web Thing Model Service. In Figure 13, a JSON payload to execute a lock action on the Smart Door is shown. To perform actions that require data stored in the database, such as a device's state or certain measurements, the MongoDB extra node is used. The database's port, name and the collection's name are required as an input, while the queries are passed from other function nodes (JavaScript). In Figure 15, an example of a connection to a collection using this node is shown. In this example the node is connected to the "entities" collection of the database to get temperature measurements with a find operation. In this way, temperature measurements can be retrieved from the database and be passed to the next node to process them.

Node-RED is the core of the subsystem responsible for the application development process. The openapi-red nodes get OpenAPI descriptions of Things as an input and serve the purpose of connecting devices in the applications. These nodes combined with other basic or extra Node-RED nodes, such as relational operators and HTTP request nodes, are able to form applications. Some examples are shown in section 4. After an application is created, a JSON output can be exported from Node-RED. This exported file can either be imported back into Node-RED to execute the corresponding application, or used to generate an OpenAPI description for the application through the "Applications OpenAPI Generator" block. After the application's OpenAPI description is created, it can be used either as a visual representation for better understanding of the application's API (e.g. endpoints, request bodies) or it can be deemed to be a Thing and therefore become a new input for the "openapi-red" node.

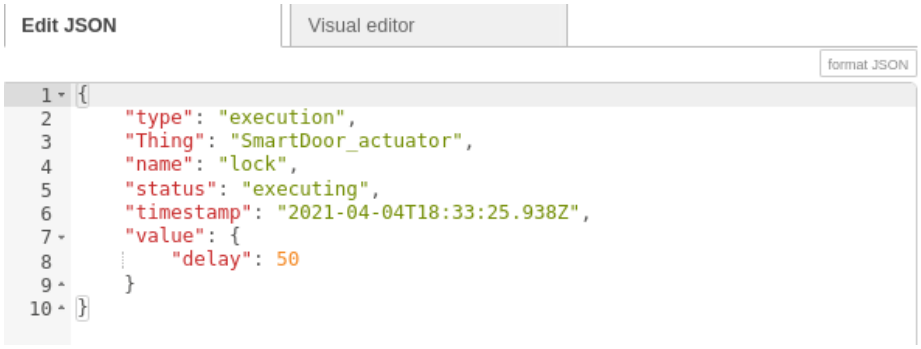



Figure 13: Example of a JSON payload to execute a lock action on a Smart Door

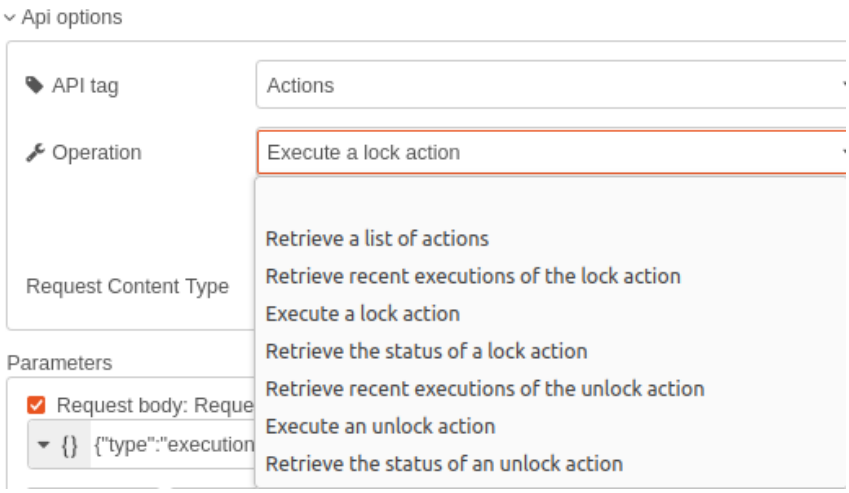



Figure 14: Example of a Smart Door Actuator in an openapi-red node

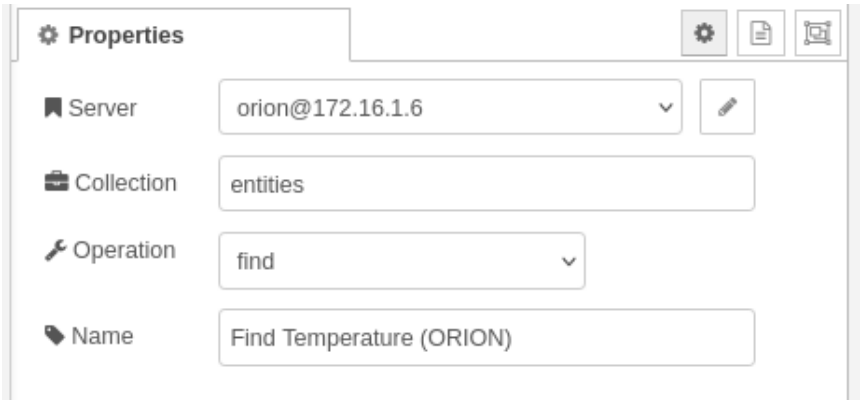



Figure 15: Example of MongoDB extra node's configuration to connect to a collection

### 3.4.1 Node-RED Output

After an application is created, it can be exported from Node-RED as a JSON file. The exported file comprises all information that was used in the application. It contains the entirety of the configuration settings for each different node, along with information about the wiring of the node, the payloads and functions. Moreover, parameters for the whole flow are included and, also, additional metadata, descriptions and comments from the developer. The output file is a key component for the Applications OpenAPI Generator that will be discussed in section 3.5. Node-RED's output file is not executable code; in order to deploy and execute an application, it has to be imported back into Node-RED.

Figure 16a presents an example of how a switch node is described in Node-RED's output file. This node functions as a relational operator ($>$ or $\leq$) and depending on the result, the output will be passed to a different node. The node has an identifier "id" and the type "switch". The most important part of this node is the "rules" field that contains the operations ($> 40$ and $\leq 40$) and the "wires" field that dictates the "id" of the next node that the output is passed to, depending on the comparison.

Figure 16b shows an example of another node that is a part of the application in the exported file. In this particular example, the node refers to a MongoDB instance that serves the purpose of finding the available devices in the database. It contains information about which collection it is connected to ("things"), which operation is

performed (“find”) and which node it is wired to (“a77023ba0c97f3fa”, which is the id of another node).


```
{
    "id": "0c6af71784cebca9",
    "type": "switch",
    "z": "f6f2187d.f17ca8",
    "name": "",
    "property": "payload.body.temp",
    "propertyType": "msg",
    "rules": [
        {
            "t": "gt",
            "v": "40",
            "vt": "str"
        },
        {
            "t": "lte",
            "v": "40",
            "vt": "str"
        }
    ],
    "checkall": "true",
    "repair": false,
    "outputs": 2,
    "x": 750,
    "y": 700,
    "wires": [
        [
            "1537361b86e71261"
        ],
        [
            "21eb06f8e6a98dbe"
        ]
    ]
},
```


(a) Switch Node


```
{
    "id": "82aa2752fd0ae44e",
    "type": "mongodb in",
    "z": "f6f2187d.f17ca8",
    "mongodb": "688d2c635bb0f177",
    "name": "",
    "collection": "things",
    "operation": "find",
    "x": 1080,
    "y": 200,
    "wires": [
        [
            "a77023ba0c97f3fa"
        ]
    ]
},
```


(b) MongoDB Node

Figure 16: Example nodes in Node-RED’s output file

## 3.5 Subsystem 3: Applications OpenAPI Generator

In this section, a mechanism is introduced to generate an OpenAPI description for applications created in Node-RED. The user needs to provide a title for the application and a short description. The description may contain information about what the application is capable of doing with the purpose of helping a user understand its functionality. In addition to the title and the description, the user provides the JSON file that was exported from Node-RED. A remarkable advantage of this mechanism, is that no technical knowledge is required by the user, as the input, aside from the textual metadata, is provided directly from Node-RED. What is more, describing applications with OpenAPI offers scalability and reusability, as these descriptions can be reused and be part of more complex applications of a larger scale. For example, applications of smart cars can be combined and comprise a new service to monitor the traffic, or a smart home's application can be reused in a smart city context.

Node-RED's export is a JSON file that contains the configuration settings, such as name, wiring and functionality, from each node used in the flow. As previously explained, the URL of the Things is an input to the openapi-red nodes and therefore it is included in the file as a configuration setting. Figure 17 presents the flow chart of the algorithm's functionality.

The process begins by parsing the JSON file (i.e. exported from Node-RED), in order to find the various URLs of Things used (step 1). Then, each URL found, will be split into parts and only the last part will be kept. As explained in section 3.1, the last part of the URL is the identifier ("_id" field) of the Thing in MongoDB. During the second step, using this identifier, the OpenAPI description of the corresponding Thing will be searched in the database (i.e. "things" collection in MongoDB).

In order to create the various OpenAPI objects, Python's dictionaries are utilized. Using dictionaries, it is ensured that there will not be any duplicates in case a Thing is used more than once in the same application. In the third step, for each Thing found in the database using the identifier, a dictionary for each JSON object is created. The info object will be appended to an "info" dictionary along with the title and the description that the user initially provided. These will be the title and description of the final OpenAPI Document. Following that, the tags array object will be appended to a "tags" dictionary, the paths object in a "paths" dictionary and the components object in a "components" dictionary. Using this approach, each dictionary will contain all the information regarding the corresponding OpenAPI

object of all the devices. For example, the "info" object will contain the info objects of all the devices, the "paths" object will contain all the paths objects etc. Finally, the aforementioned dictionaries, are appended to a final dictionary (dictionary of dictionaries), thus containing all the OpenAPI objects and fields that each separate Thing comprised (step 4). The process ends in the fifth step by converting the final dictionary into a JSON file which is the output of the algorithm (i.e. the final OpenAPI description of the application).

When a new application description is generated, it is stored in the "applications" collection. A unique identifier "_id" is assigned to each new application document insertion by MongoDB, and a URL is created, where the last part of the URL is the value of the identifier. The URL exposes the contents of the document (i.e. the OpenAPI of the application) and after declaring the correct server URLs in the OpenAPI servers field, the document can be reused in new applications. Since the application consists of Things that follow the OpenAPI Thing Template, and have a common REST API (i.e. the WTM service), the application, as a whole, will also follow the template and be implemented with the WTM service REST API (Section 2.1.11).

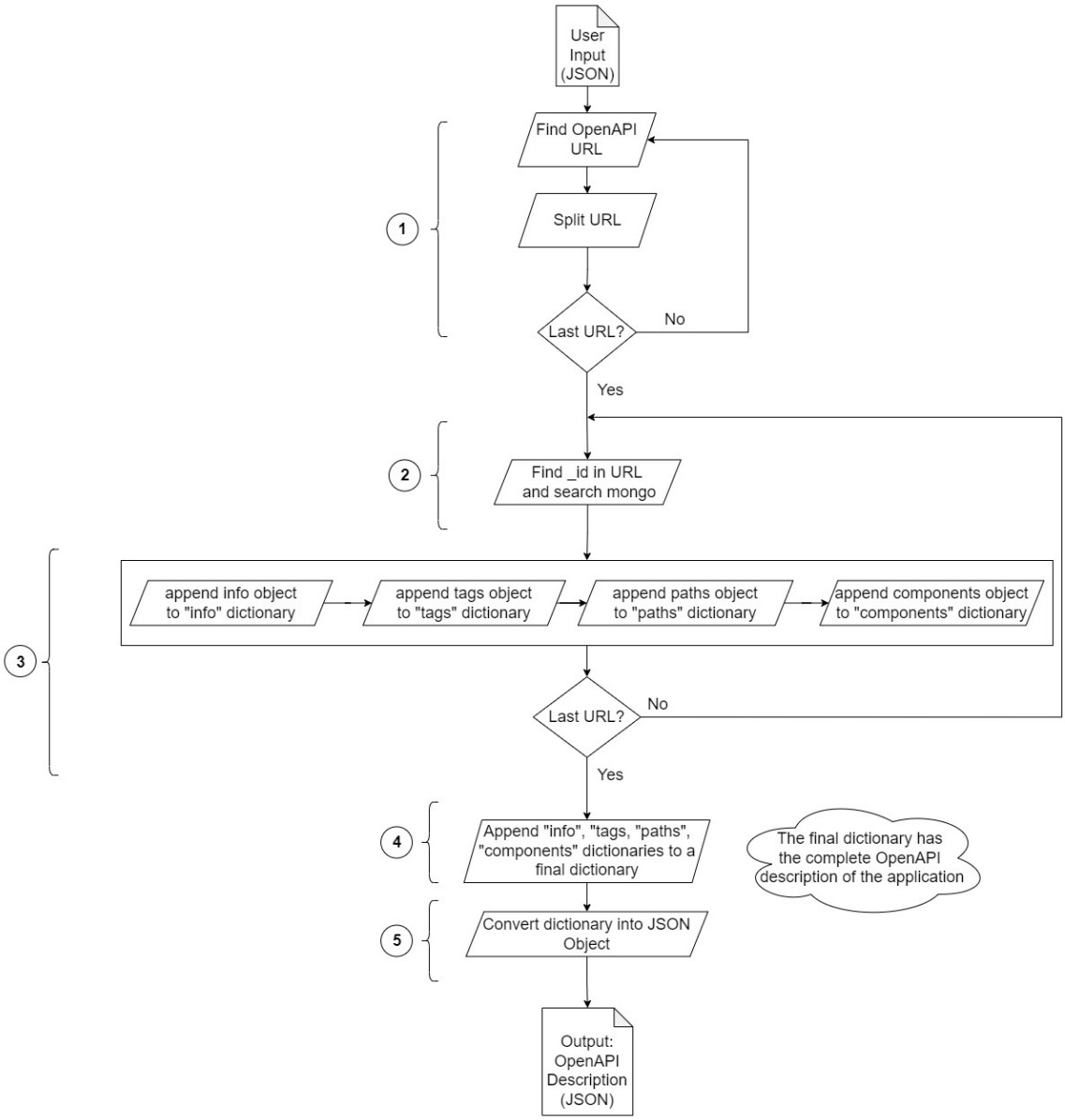



Figure 17: Generation process of OpenAPI description for an application

Figure 18 depicts the Info Object of the application's generated OpenAPI Description in SwaggerHub's UI. The title and description from the user input is included in this object. This example is an application that uses a Motion Sensor's and a Smart

Lamp's OpenAPI descriptions. When motion is detected through the sensor, the Smart Lamp gets switched on. Figure 19 presents the Paths Object. In this Object the properties, actions and subscriptions (Figure 20) of the Things are included. In this particular example, both the Motion Sensor and the Smart Lamp support properties and subscription operations, while only the Smart Lamp supports actions. In this figure, the available endpoints (e.g. /actions/lampOn) along with their requests (i.e. GET, POST) are shown. Last but not least, in Figure 21 the Components Object of all Things used in the application are shown, for instance the request body to switch on the Smart Lamp (i.e. LampOnRequestBody) or to retrieve the state of a device (i.e. State).

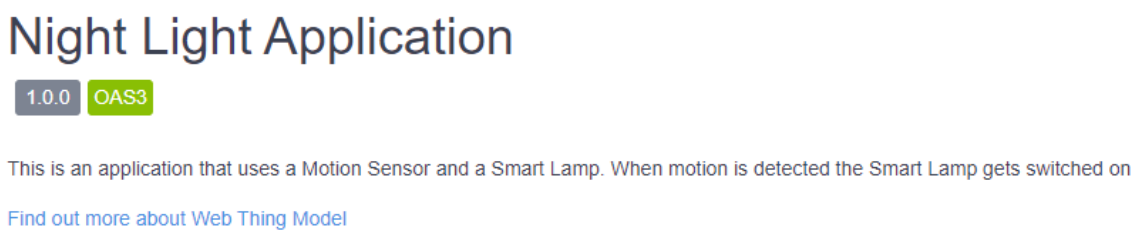



Figure 18: Example Info Object of a generated OpenAPI for an application


Figure 19: Example Paths Object of a generated OpenAPI for an application

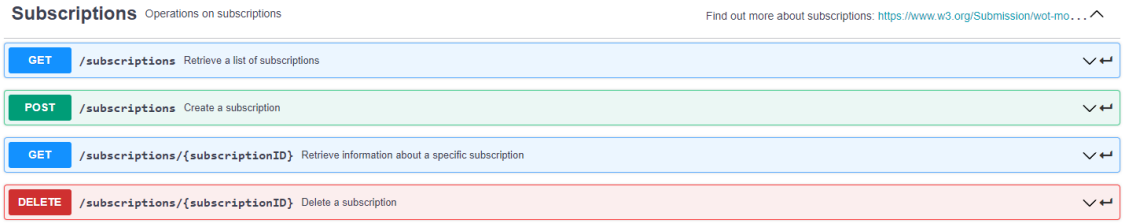



Figure 20: Example Subscriptions Object of a generated OpenAPI for an application


Figure 21: Example Components Object of a generated OpenAPI for an application

## 3.6 The complete System Architecture

In this section the complete system's architecture, that comprises all the aforementioned components will be discussed. As it was described in Section 3.2, there are three options (subsystems) given to the user, each consisting of different components and having different inputs and outputs. Thus, the final system, shown in Figure 22,

is a merge of those subsystems (i.e. Figure 11), enriched with the Context Broker and the REST API provided by the Web Thing Model Service. Figure 22 will be analysed further in this section.

The central idea is to provide a Service Oriented Architecture, where Things are described with OpenAPI as a way to expose their functionality and make them discoverable and reusable, as well as to give the ability to users to interact with them as RESTful services. The "openapi-red" extra node, provides a User Interface for OpenAPI Documents in Node-RED, which results in an efficient way of interacting with Things, due to the fact that operations, payload schemas and endpoints are conviniently visible, grouped by the relevant tags and described with textual metadata explaining each operation's functionality. For example Figure 14 presents an openapi-red node of a Smart Door Actuator. Selecting the tag "Actions" reveals all the available actions that this particular device supports.

Starting from the first available option, after a user inserts a new Thing's description, it will be stored in the MongoDB database's "things" collection. This OpenAPI description describes the Thing's REST API which is implemented with the WTM Service. When an HTTP request is issued towards a Thing, the REST API will communicate with the Context Broker and either store or retrieve data regarding this request.

In order to compose a new application, Node-RED's interface provides a variety of nodes, which are black boxes with various functionality. As explained in Section 3.4, the user creates a flow of these nodes in a way that achieves the goal of the scenario that was planned. To implement a Thing, the openapi-red node gets the Thing's OpenAPI description as an input, hence the other nodes are able to communicate with the Thing's REST API (i.e. WTM Service) through HTTP requests. When an application is created, it can be exported from Node-RED in a JSON file that was explained in section 3.4.1. This output can be imported back into Node-RED to execute the application, or become the input for the application's OpenAPI Generator, which is the user's third option.

To generate an application's OpenAPI description, the user needs to provide Node-RED's exported file as an input. After the description is produced, it will be stored in "applications" collection in MongoDB. The application's REST API is, also, implemented with the Web Thing Model Service, since its endpoints are identical with the Things it includes. Therefore, applications can be treated as Things and their

OpenAPI can be used in Node-RED to become a part of applications of a larger scale.

In this architecture, it is mandatory for Things to have an OpenAPI description, due to the fact that it is the way that Things are integrated in application flows and other components of the application can communicate with them. Apart from that, by describing Things with OpenAPI, it is ensured that they can be used without knowledge requirements about source code or documentation, as well as they can be reused to compose new applications. Regarding the system's infrastructure components, such as the MongoDB or the Context Broker, OpenAPI provides documentation and, additionally, it indicates how these components can be used in applications by exposing endpoints where HTTP requests can be issued towards.

In this system Node-RED, MongoDB, the Context Broker, the Things OpenAPI Generator and the WTM Service are existing technologies and they were described in Section 2.1 (Infrastructure and Tools). In the case of the Thing OpenAPI Generator, it was modified, so that it is able to accept a JSON file, instead of a JSON payload in an HTTP Request, as well as being able to accept a valid OpenAPI document in addition to the JSON file with the defined user input. This work proposes the design of the architecture and the utilization of OpenAPI to connect Things in Node-RED. Moreover, the Applications OpenAPI Generator was designed from the ground up to suit the needs of this system, both in terms of application description and the ability to reuse applications in Node-RED.

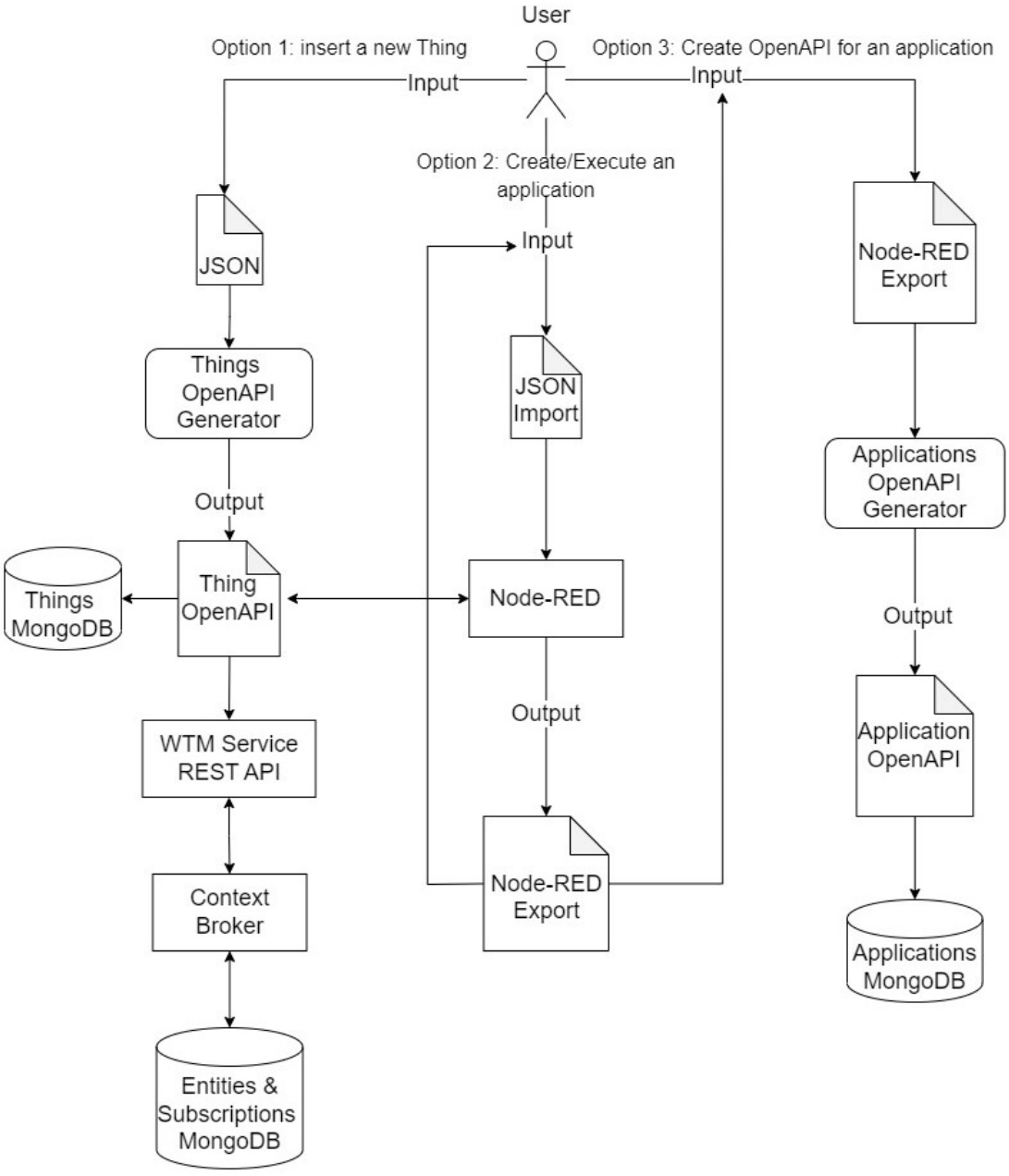



Figure 22: The complete System Architecture of MoON

# 4 Use Case Scenario & Example Applications

The mashup service can be used to create applications of various scales. In the following examples, the applications range from a device-to-device level up to a Smart City scenario. Users create the plan of the application's functionality in Node-RED's visual programming tool by connecting disparate items such as sensors, relational operators and computing functions, in the form of black boxes.

The Things used in these examples are a Smart Door Actuator, a Smart Lamp Actuator, an Air Conditioner, a Motion Sensor and a DHT22 Sensor that gets temperature and humidity measurements. OpenAPI descriptions of applications that include these devices, are composed of all of their Thing resources (i.e. properties, operations, endpoints etc.). Smart Homes are created using these descriptions and contain combinations of the aforementioned Things. It is, then, possible to use the Smart Homes as Things in an application regarding a Smart City.

The openapi-red node, which is the purple node in the following flows, is used as an interface for the other parts of the application to interact with the device's API. In order to form applications, the user is required to have basic knowledge of Node-RED's environment and JavaScript in order to form more complex functions whose functionality is not already provided by the basic nodes. The API implementation of the devices is handled through OpenAPI's interface that offers textual metadata, therefore the user does not need any technical knowledge in terms of understandability.

MoON provides all the required components to store Things, create and execute applications, and generate OpenAPI descriptions for applications. These components are the MongoDB database, the Context Broker, Node-RED, the REST API and the OpenAPI Generators for Things and applications. In this regard, it can function as a standalone mashup system. Apart from that, it can be integrated into any Internet of Things or Web of Things system and provide a Mashup environment, as long as the aforementioned components are either provided or integrated as well. Being a Service Oriented Architecture, it enables seamless integration with more complex systems in order to provide the additional functionality.

**Example 1**

The functionality of the first application is to switch on the Smart Lamp at 20:00 and then switch it off at 02:00. The values of the time range are provided by the user before run time. As shown in Figure 23, the starting point of the application is the inject node (light blue) that triggers the flow. The second node is a Time Range Node (orange). Using this node the next step is decided based on a time range (20:00 - 02:00 in this instance). The executeLampOnAction and executeLampOffAction (purple nodes) are nodes to execute an action on the device using their OpenAPI Description. With this node the Smart Lamp will be switched on or off depending on the time range output of the previous step. Finally, there is the msg.payload (green node) that outputs a message to the user for debugging purpose. Figure 24 shows an extension of this application with the addition of a Smart Door Actuator and an action to lock or unlock the Smart Door will be executed during the time range 24:00 - 09:00.

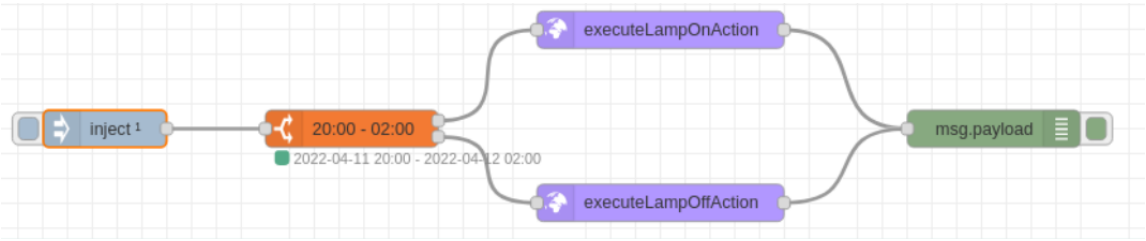



Figure 23: Application to switch ON/OFF the lights depending on time

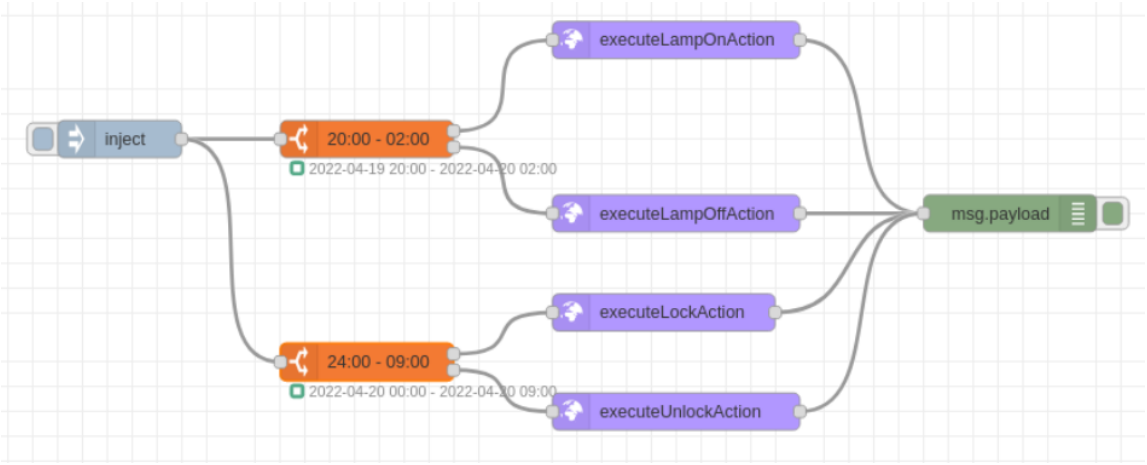



Figure 24: Application to switch ON/OFF the lights and lock/unlock the door depending on time

**Example 2**

The second application (Figure 25) has the purpose to switch on the Air Conditioner when the temperature surpasses a certain threshold, that is set by the user. The inject node (light blue) triggers the flow to start the application and the user sets an interval to repeat the process. The bottom row that comprises two HTTP requests nodes (yellow) and a function node (orange), simulates the functionality of the sensor, with the function node generating a random number as a temperature and then issuing an http request to the Context Broker. With the delay node (pink), the developer dictates the rate that the temperature will be observed. Following the delay, the retrieveTempProperty node (purple) retrieves the temperature measurement using the DHT22 Sensor's OpenAPI Description. Using a switch node (gold), a temperature threshold is set by the user and with a relational operator ($\geq$ or $<$) the next node will either switch OFF the AC by interacting with the Thing through the executeSwitchOffAction node or switch ON the AC through the executeSwitchOnAction (purple nodes).

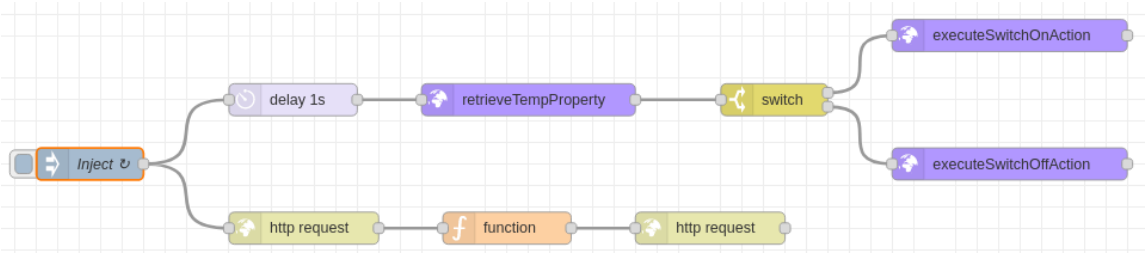



Figure 25: Application to switch ON/OFF the AC depending on a temperature threshold

### Example 3

In this example, a motion sensor is used that has two states, "IDLE" and "DETECTED", and depending on its state, whether motion is detected or not, a Smart Lamp gets switched on. Figure 26 presents the components that comprise this functionality. The third example operates in a similar way as the second. Initiating with a timestamp node (light blue), the application starts and a time interval is set by the developer. Subsequently, through the function node (orange) a random state is generated, either IDLE or DETECTED, and is then submitted to the Context Broker through an HTTP request (yellow node). Using the delay node (pink), the developer dictates the rate that the state will be observed after it is updated. With the retrieveProperty node (purple), the state of the Motion Sensor will be examined and passed to the switch node (gold). In this node, a relational operator ($state = IDLE$ or $state = DETECTED$) will dictate the next step and execute an action, either switch ON the Smart Lamp with the executeLampOnAction or switch OFF the Smart Lamp using the executeLampOffAction (purple nodes).

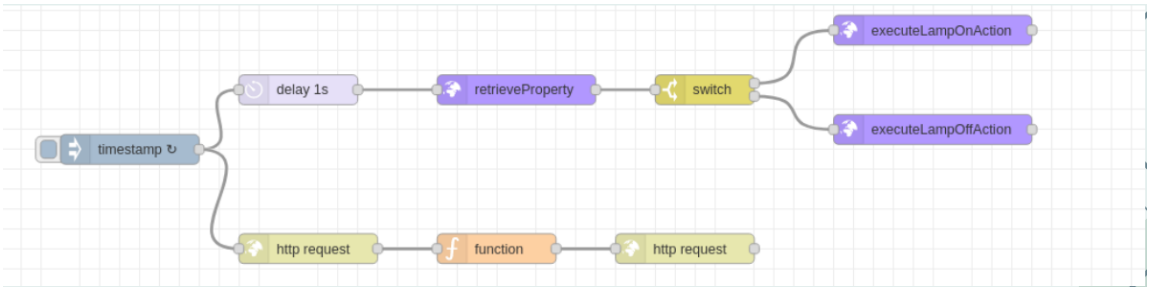



Figure 26: Application to switch ON/OFF the lights when motion is detected through the Motion Sensor

### Example 4

The fourth application presents a smart city example. In this application the OpenAPI of Smart Homes and DHT22 sensors were used. The goal of this example is to calculate the average energy consumption due to ACs being switched on. As shown in Figure 27, a city comprises four neighbourhoods which are consisted of four houses each.

Figure 28 focuses on one of the neighbourhoods. The application starts with an inject node (light blue) that retriggers the flow in an interval of ten seconds. The first HTTP request node (yellow) initializes the device's state, followed by a function node (orange) that initializes the counters. The counter variables hold the number of repetitions as well as a counter of how many ACs are switched on at any given time. The second row contains two HTTP requests (yellow nodes) and a function (orange node), simulating the functionality of the sensor and generating a random number as a temperature. Following the delay node (pink) that issues a delay of one second, the temperature is retrieved using the retrieveProperty node (purple) through the DHT22 Sensor's OpenAPI Description. Using a switch node (gold) the threshold is set by the user and with a relational operator ($\geq$ or $<$) the next node (i.e. yellow HTTP request node) will change the AC's state to ON or OFF by issuing and HTTP request to the Orion Context Broker, depending on the result of the switch node. If the AC gets switched on, the function node increments a global counter of switched on devices. The next step will either switch OFF the AC by interacting with the device through the executeSwitchOffAction node or switch ON the AC through the executeSwitchOnAction node (purple nodes) using their

OpenAPIs. This process is repeated for each neighbourhood of the Smart City and finally after a delay of seven seconds (pink node) a function node calculates the average number of devices switched on by dividing the counter with the number of repetitions. The flow ends with a message node (green) that outputs the average number of ACs that are switched on.

In this way, assuming an average consumption of 1KW per hour of a switched on device, the average energy consumption of the Smart City can be calculated. As presented in Figure 29, this Smart City had an average of 6.2 switched on devices over the course of 360 cycles of ten seconds each, which would result in an energy consumption of 6.2 KWh.

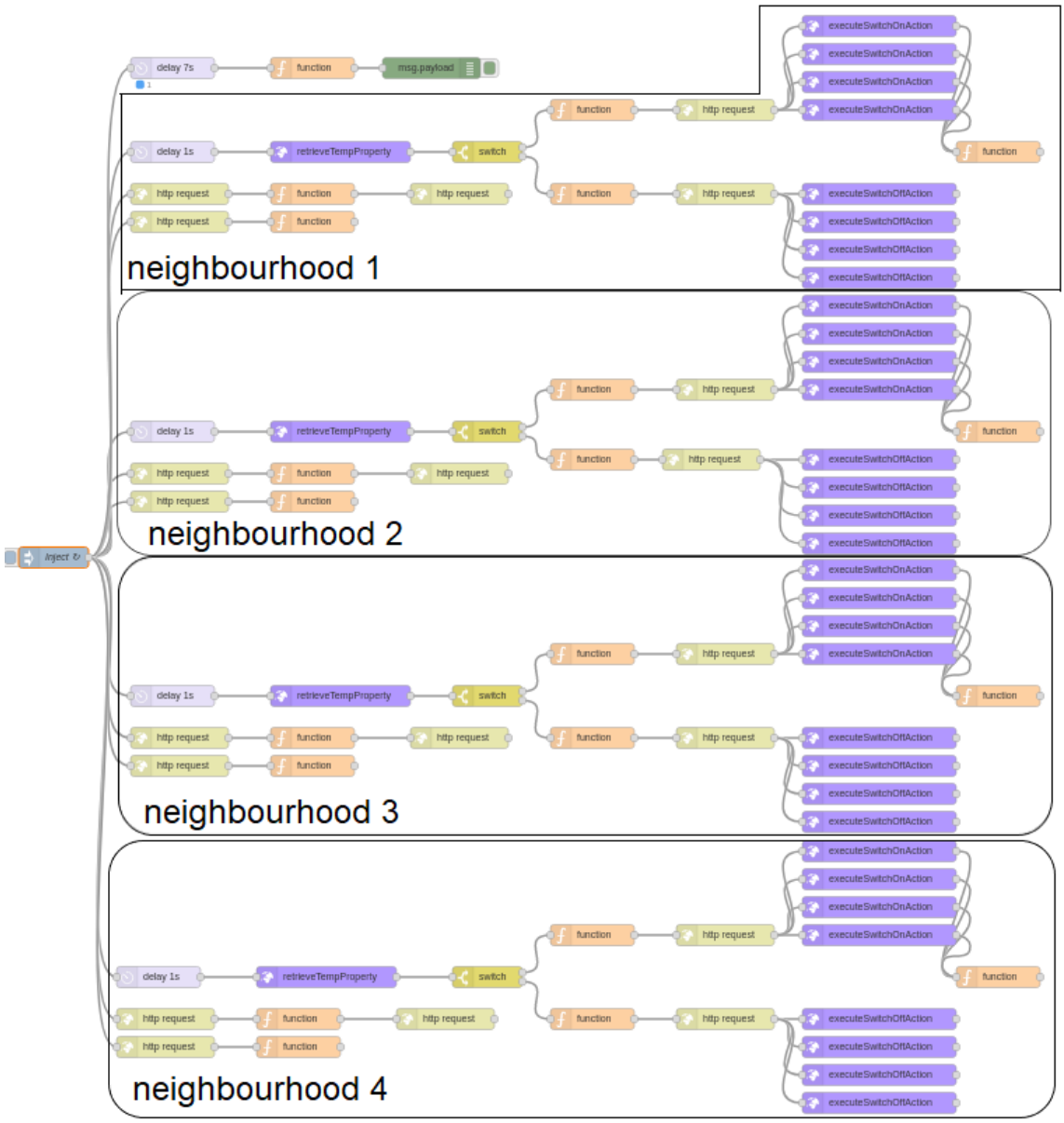



Figure 27: Application to calculate the average energy consumption by ACs in a Smart City

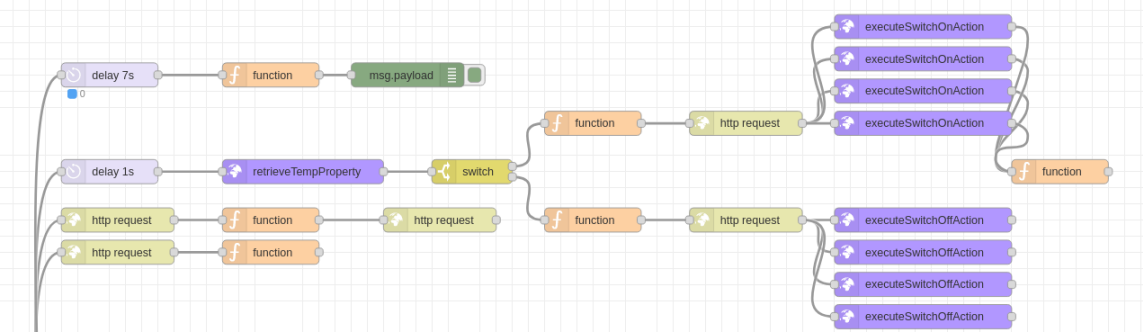



Figure 28: One neighbourhood of the Smart City application

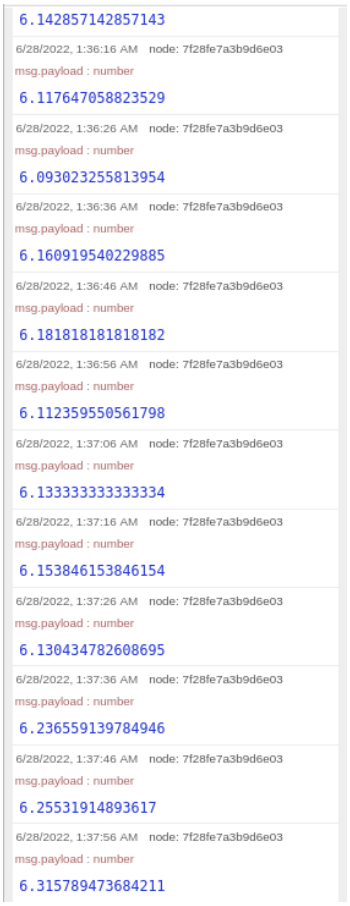



Figure 29: Average number of ACs switched on after one hour

## 4.1 Response Time Measurements

| Application | Response Time (ms) |
|---|---|
| Application to switch on the lights depending on time (Figure 23) | 18 ms |
| Application to switch on the lights and lock the door depending on time (Figure 24) | 32 ms |
| Application to switch on the AC depending on a temperature threshold (Figure 25) | 29 ms |
| Application to switch on the lights when motion is detected through the Motion Sensor (Figure 26) | 30 ms |
| Application to calculate the average energy consumption by ACs in a Smart City (Figure 27) | 298 ms |

The first application consists of only one device and therefore it has the lowest response time (18 ms). The next 3 flows issue requests to the API of 2 devices, thus they have similar response times. The third and fourth flows (i.e. Examples 2 and 3) are very similar in implementation as they use the same endpoints of the API. The first operation retrieves a property, while the second operation performs an action on the device and their similar response time is justified. The fifth flow (Example 4) uses a lot of devices, since there are sixteen Smart Homes that comprise the city which leads to a significantly longer response time. It should be noted that the aforementioned response times include some delays introduced by the machine they were measured on.

# 5 Conclusion & Future Work

## 5.1 Conclusion

Treating all Things as RESTful Web services to facilitate the interoperability in application scenarios made the communication among the Things easier and, consequently, streamlined the process of interacting with those services. A notable advantage of MoON, is that, being Service Oriented, it allows for not only scalability, but it also enables the users to modify the individual services while maintaining the core functionality.

Even though W3C attempted to propose a solution for the problems that stem from the IoT concept by recommending the description of Web Things using the W3C WoT Thing Description (TD), OpenAPI can maintain the core notion but also complement it further to provide detailed descriptions of any kind of Thing as a RESTful Web service. Utilizing this standard, that is also compliant with the industry, ensures uniformity among the representations of the services and creates a common way in which they are interacted with.

Flow-Based Programming tools are able to diminish the steep learning curve of the application creation process by lessening the knowledge requirements due to having a large part of the coding take place in the background. This particular work makes use of Node-RED, but can be compatible with more tools with negligible modifications. What is more, the graphical interface of the Flow-based programming tools provides a visual environment with a description of the application's logic, thus rendering them understandable by users of minimal experience. Extending Node-RED with OpenAPI specific nodes provides ease of use, both in a graphical sense, making the application more compact and avoiding using more nodes to achieve the same result, and in a practical sense, requiring less coding knowledge and instantly providing the desired endpoints.

This work is based on many existing tools and prior knowledge, however it presents a new concept that can find application in a variety of real world scenarios. The OpenAPI Thing Template, the idea of replacing W3C's Thing Descriptions with OpenAPI descriptions and the Flow Based Programming tools are progress made in the past, but combining all these ideas together, to make a Mashup system that utilizes OpenAPI to expose Things and make them Web Services, brings a new system with a lot of potential in IoT and WoT architectures.

## 5.2 Future Work

The implementation of this work meets the requirements set during the planning process of this assignment, and provides a scalable Mashup system where every service is accompanied by an OpenAPI description, as well as an OpenAPI generating tool to provide descriptions that allow for the understandability and reusability of the produced applications. Nevertheless, due to the fact that this thesis had a limited time frame there are some weak points that will be presented below along with a proposed solution as future work.

The first improvement that can be made is adding security services. An example of that is FIWARE's PEP Proxy - Wilma [17] that allows the access to system resources only by authenticated and authorized users. The addition of such services is of utmost importance in case this system is implemented in a real world application.

Another extension of MoON could be the implementation of the OpenAPI QL, a query language for querying OpenAPI service descriptions [18]. The basic idea behind OpenAPI QL approach is that the OpenAPI document is a description of a REST request with the corresponding responses. The addition of such a query language specifically for OpenAPI documents would make the discoverability of Things in the system easier and more detailed, as the queries are able to contain all the information that the user is searching for.

Last but not least, an improvement that would make MoON compatible with more IoT architectures, would be to make it possible to replace Node-RED with any other Flow-Based Programming tool. This would mainly require modifications on the way that the OpenAPI Generator for applications works in order to recognise the output files of more FBP tools.

# References


[1] Michael Blackstock and Rodger Lea. «IoT mashups with the WoTKit». In: *Proceedings of 2012 International Conference on the Internet of Things, IOT 2012* (Oct. 2012), pp. 159–166. DOI: 10.1109/IOT.2012.6402318.

[2] *C++ Implementation of Flow-Based Programming (FBP)*. URL: https://github.com/jpaulm/cppfbp.

[3] *FIWARE Platform*. URL: https://www.fiware.org/.

[4] *Flowhub IDE*. URL: https://flowhub.io/ide/.

[5] Janggwan Im, Seonghoon Kim, and Daeyoung Kim. «IoT Mashup as a Service: Cloud-Based Mashup Service for the Internet of Things». In: *2013 IEEE International Conference on Services Computing*. 2013, pp. 462–469. DOI: 10.1109/SCC.2013.68.

[6] *MongoDB Documentation*. URL: https://www.mongodb.com/docs/.

[7] *MongoDB Extra Node for Node-RED*. URL: https://www.npmjs.com/package/node-red-node-mongodb.

[8] *Node-RED Documentation*. URL: https://nodered.org/docs/.

[9] *NoFlo Documentation*. URL: https://noflojs.org/documentation/.

[10] Mahda Noura, Sebastian Heil, and Martin Gaedke. «Webifying Heterogenous Internet of Things Devices». In: Apr. 2019, pp. 509–513. ISBN: 978-3-030-19273-0. DOI: 10.1007/978-3-030-19274-7_36.

[11] *OpenAPI Generator*. URL: https://openapi-generator.tech/.

[12] *OpenAPI Generator mechanism*. URL: https://github.com/Emiltzav/wot_openapi_generator.

[13] *OpenAPI Specification*. URL: https://swagger.io/specification/.

[14] *openapi-red Extra Node for Node-RED*. URL: https://www.npmjs.com/package/openapi-red.

[15] *OpenAPI.Tools*. URL: https://openapi.tools.

[16] *Orion Context Broker Documentation*. URL: https://fiware-orion.readthedocs.io/en/master/.

[17] *PEP Proxy - Wilma*. URL: https://fiware-pep-proxy.readthedocs.io/en/latest/.

[18] Ioanna Stergiou Maria and Euripides G.M. Petrakis. "Searching in REST Service Catalogues with OpenAPI Descriptions". In: ().

[19] *The Web of Things architecture*. URL: https://www.w3.org/TR/wot-architecture/.

[20] *The World Wide Web Consortium (W3C)*. URL: https://www.w3.org/Consortium/.

[21] Aimilios Tzavaras. ≪Thing descriptions for the semantic Web of Things≫. MA thesis. School of Electrical and Computer Engineering, Technical University of Crete, Chania, Greece, 2022. DOI: https://doi.org/10.26233/heallink.tuc.92084.

[22] Aimilios Tzavaras, Nikolaos Mainas, Fotios Bouraimis, and Euripides G.M. Petrakis. "OpenAPI Thing Descriptions for the Web of Things". In: *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*. 2021, pp. 1384–1391. DOI: 10.1109/ICTAI52525.2021.00220.

# Appendices

## A Thing OpenAPI Documents

https://github.com/baspap54/Thesis-Appendix/tree/main/Appendix/OpenAPI/OpenAPI%20Documents

## B Node-RED Output

https://github.com/baspap54/Thesis-Appendix/tree/main/Appendix/Node-RED%20Output%20Examples

## C Application OpenAPI Documents

https://github.com/baspap54/Thesis-Appendix/tree/main/Appendix/OpenAPI/Application_OpenAPI

## D User Input for the OpenAPI Generator

https://github.com/baspap54/Thesis-Appendix/tree/main/Appendix/OpenAPI/JSON%20User%20Input