FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Using SensorThings API to enable a multi-platform IoT environment

José Alexandre Barreira Santos Teixeira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Correia Lopes

Co-supervisor: Artur Rocha

13th July 2018

© José Alexandre Barreira Santos Teixeira, 2018

Using SensorThings API to enable a multi-platform IoT environment

José Alexandre Barreira Santos Teixeira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Carla Teixeira Lopes

External Examiner: Prof. Alexandre Valente de Sousa

Supervisor: Prof. João Correia Lopes

13th July 2018

Resumo

Recentemente, existe um esforço por parte de investigadores numa nova utilização da Internet, denominada por Internet of Things (IoT). Este conceito baseia-se na premissa que dispositivos eletrónicos podem ser acedidos remotamente pela Internet.

IoT representa uma próxima evolução da Internet, dando um contributo significativo na sua capacidade de colecionar, analizar e distribuir dados que poderão posteriormente ser transformados em informação e conhecimento. Dadas as suas aplicações, a IoT torna-se muito importante.

Uma das aplicações mais comuns para a IoT são as *smart cities* cujo objetivo é fazer um melhor uso de recursos públicos, aumentando a qualidade dos serviços oferecidos aos cidadãos, ao mesmo tempo reduzindo os custos operacionais das administrações públicas. No entanto, existem vários outros campos onde este paradigma se torna útil como em automação doméstica, automação industrial, saúde, gestão inteligente de energia e *smart grids*, gestão de trânsito, entre outras aplicações.

Devido à existência recente de IoT, os produtores de dispositivos conectados à Internet e fornecedores de serviços IoT estão a definir os seus próprios protocolos baseados nas aplicações a serem desenvolvidas. Isto por sua vez fará com que os ambientes IoT ganhem um grau de heterogeneidade em termos de capacidades de *hardware* e protocolos de comunicação.

Uma forma de mitigar a heterogeneidade seria optar por escolher convenções nestes dispositivos IoT de forma a permitir a interoperabilidade. Através da utilização de convenções para protocolos de comunicação nestes dispositivos, não só estes dispotivos tornam-se auto-descritivos e interoperáveis, como também dá origem a que novas aplicações possam ser desenvolvidas tendo por base uma interface subjacente comum.

O objetivo do trabalho descrito nesta dissertação é o desenvolvimento de uma aplicação que toma por base um *standard* — o SensorThings API — para permitir a criação de um ambiente IoT multi-plataforma. O SensorThings API é um *standard* proposto pelo Open Geospatial Consortium (OGC), desenhado especificamente para ambientes IoT tendo em consideração os recursos restritos dos mesmos.

Este projeto envolve o desenvolvimento de uma aplicação que permite agregar informação, dados dos dispositivos e interoperar com outras aplicações de catalogação via *standards* comuns. Desta forma, esta aplicação terá que aceder aos dados que estão espalhados entre os seus nós IoT constituintes. Os dados que são gerados são o resultado da actividade de sensores e que poderão ser uma multitude de propriedades físicas, de acordo com as próprias capacidades do sensor.

O resultado final do projeto pode constituir uma prova de conceito importante em como a utilização de *standards* comuns podem concretizar a possibilidade de diversas entidades participarem em formas colaborativas de partilhar dados e de simultaneamente usar informação de outras fontes para produzir análises mais complexas.

Abstract

Recently, researchers are focusing on a new use of the Internet called the Internet of Things (IoT). This concept is based on the premise that electronic devices can be remotely accessed over the Internet.

IoT represents an evolution of the Internet, taking a huge leap in its ability to gather, analyze, and distribute data that can in turn be transformed into information and knowledge. Due to its applications, IoT becomes immensely important.

One of the most common applications for IoT are smart cities whose final aim is to make a better use of the public resources, increasing the quality of the services offered to the citizens, while reducing the operational costs of the public administrations. However, there are many other fields where this concept is most useful such as home automation, industrial automation, healthcare, intelligent energy management and smart grids, traffic management and many others.

As the IoT is in a very early stage, manufacturers of Internet-connected devices and IoT web service providers are defining their own proprietary protocols based on their targeted applications. This in turn will make the IoT environment heterogeneous in terms of hardware capabilities and communication protocols.

To solve this heterogeneity one possible solution would be opting for open standards in these IoT devices thus enabling the interoperability between each other. By hosting open standard communication protocols on these devices, the devices become self-describable and interoperable and new applications can be developed with a common underlying interface.

The main goal of the work presented in this dissertation is to develop an application that leverages from an open standard — the SensorThings API — to allow the creation of a multi-platform IoT environment. The SensorThings API is a standard proposed by the Open Geospatial Consortium (OGC), designed specifically for an IoT environment taking into consideration the constrained resources of the IoT devices.

This project encompasses the development of an application that is able to aggregate information, device data and interoperate with other cataloging applications via common standards. Therefore, this application has to seamlessly access the data that is scattered between the IoT constituent nodes. The data that is generated is the result of the sensors activity and can be a multitude of physical properties according to the sensors own capabilities.

This project's success could be an important proof of concept on the usage of open standards to enable for distinct organizations to participate in a collaborative way to share their data, and also by using data from other sources in order to produce higher-value outputs.

Acknowledgments

Firstly, I would like to thank my supervisors, João Correia Lopes and Artur Rocha for all the guidance and orientation during all the phases of this work.

To the colleagues and friends I had the chance to meet over these past five years, undoubtedly they had an impact at some point and it is something I will hold dearly.

To my family, I express my solemn gratitude for the unfailing support through all the stages of my life, this one included.

To all the aforementioned, my sincerest thanks.

José Alexandre Teixeira

“Man’s real life is happy, chiefly because he is ever expecting that it soon will be so.”

Edgar Allan Poe

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Problem Definition . . . 2
1.3 Motivation and Goals . . . 3
1.4 Structure . . . 4

2 State of the Art **5**
2.1 Communication Protocols . . . 5
2.1.1 ZigBee . . . 6
2.1.2 Z-Wave . . . 6
2.1.3 6LoWPAN . . . 6
2.1.4 MQTT . . . 7
2.1.5 Constrained Application Protocol . . . 7
2.2 Communication Architectures and Platforms . . . 8
2.2.1 SensorThings API . . . 8
2.2.2 WebThings API . . . 13
2.2.3 Comparison between SensorThings API and Web Thing API . . . 14
2.2.4 oneM2M . . . 15
2.2.5 AIOTI High-Level Architecture . . . 16
2.2.6 RIOT-OS . . . 17
2.3 Semantic Interoperability in the Internet of Things . . . 17
2.4 SensorThings API Server Implementations . . . 19
2.4.1 FROST . . . 19
2.4.2 SensorUp SensorThings . . . 19
2.4.3 GOST . . . 19
2.4.4 Mozilla . . . 19
2.4.5 CGI Kinota Big Data . . . 20
2.5 Applications using SensorThings API . . . 20
2.5.1 SensorThings Admin Dashboard . . . 20
2.5.2 SensorThings Map . . . 21
2.5.3 Air quality monitoring after wildfires . . . 21
2.6 Conclusions . . . 22

3 Problem Statement and Solution Proposal **25**
3.1 Current Issues . . . 25
3.2 User Stories . . . 26
3.3 Solution Proposal . . . 28
3.3.1 CoralTools Backend Application . . . 29

3.3.2 SensorThings API endpoints . . . 31
3.3.3 Web Application . . . 32
3.3.4 OSGeoLive . . . 32
3.4 Conclusions . . . 33

4 Device Management **35**
4.1 Introduction . . . 35
4.2 Endpoint collection . . . 35
4.2.1 Collecting *Thing* metadata . . . 36
4.2.2 Catalog . . . 36
4.3 Control Panel . . . 36
4.3.1 Management of SensorThing API entities . . . 37
4.3.2 Management of Authorization tokens . . . 39
4.4 Conclusions . . . 39

5 Metadata Cataloguing Implementation **41**
5.1 Introduction . . . 41
5.2 Overview . . . 42
5.3 Structuring the metadata model . . . 42
5.4 Generating the Contents field . . . 43
5.5 Architecture . . . 45
5.6 Conclusions . . . 45

6 Dashboard Implementation **49**
6.1 Datastream selection . . . 49
6.2 Heatmaps . . . 49
6.3 Clustered View . . . 52
6.4 Historical Locations . . . 53
6.5 Datastreams presented in charts . . . 56
6.6 Datastream availability . . . 57
6.7 Conclusions . . . 58

7 Evaluation **65**
7.1 Management of the SensorThings API entities . . . 65
7.1.1 Creating a new *Thing* entity . . . 65
7.2 Using pycsw and GeoNetwork for metadata publishing . . . 68
7.2.1 Comparison of a Sensor Observation Service *GetCapabilities* . . . 68
7.2.2 Harvesting a SensorThings API endpoint . . . 70
7.2.3 Using GeoNetwork . . . 71
7.3 Interoperability of the SensorThings API . . . 71
7.4 Creating an IoT platform . . . 72
7.4.1 Setting up the IoT platform . . . 72
7.4.2 Querying the Observation Data . . . 73
7.5 Conclusions . . . 73

8 Conclusions and Future Work **77**
8.1 Summary . . . 77
8.2 Future Work . . . 78

References **81**

List of Figures

2.1 SensorThings API Data Model . . . 9
2.2 Web Things API Data Model . . . 14
2.3 oneM2M Functional Architecture . . . 16
2.4 Chain of requests for observation retrieval on a SOS platform . . . 18
2.5 GOST Dashboard . . . 20
2.6 SensorThings Admin Dashboard . . . 21
2.7 SensorThings Map . . . 22
2.8 Animated map based on a timeseries evolution of Saint Albert air quality monitoring 23

3.1 Deployment diagram for a Raspberry Pi, the server application and the web application 30

4.1 Catalog view on the web application . . . 37

5.1 *Thing* entity table . . . 43
5.2 Application Deployment Diagram . . . 46

6.1 Interface for selecting datastreams . . . 50
6.2 Interface for the selected datastreams . . . 52
6.3 Heatmap . . . 59
6.4 Map representation of the clustered View for 4 different datastreams . . . 60
6.5 Table containing the information of the result values for a point represented in the map . . . 61
6.6 Table containing the information of the result values for a point represented in the map . . . 61
6.7 Representation of the Historical Locations of two distinct *Things* . . . 62
6.8 Representation of the Historical Locations of three Endpoints and multiple *Things* 62
6.9 Interface for the selection of *Datastreams* and time span . . . 63
6.11 Interface for the selection of *Datastreams* and their corresponding gaps . . . 64

7.1 Form with the *properties* field containing metadata about the service. . . . 66
7.2 Table containing all the *Thing* entities of an endpoint address. . . . 67
7.3 Form with the *properties* field containing metadata about the service . . . 75
7.4 Values of the *Datastream* Wind Meters/second of the **Mercury** endpoint . . . 76

List of Tables

3.1 User Stories for the CoralTools Application 27

5.1 GetCapabilities fields 43
5.2 JSON structure 44
5.3 Minimum parts of a single *Datastream* in the Contents field 45

Acronyms

API	Application Programming Interface
AUV	Autonomous Underwater Vehicle
CoAP	Constrained Application Protocol
CTD	Conductivity-Temperature-Depth
CSW	Cataloging Service for the Web
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
KML	Keyhole Markup Language
MQTT	Message Queue Telemetry Transport
REST	Representational State Transfer
SDK	Software Development Kit
SOAP	Simple Object Access Protocol
OGC	Open Geospatial Consortium
OWS	OGC Web Services
ROV	Remotely Operated Vehicle
SOS	Sensor Observation Service
STA	SensorThings API
UART	Universal Asynchronous Receiver
USB	Universal Serial Bus
Wi-Fi	Wireless Internet
W3C	World Wide Web Consortium
WMS	Web Map Service
WSN	Wireless Sensor Networks
WWW	*World Wide Web*
XML	Extensible Markup Language

Chapter 1

Introduction

From 2003 to 2010, in a population of approximately 6.3 billion people, the number of devices connected to the Internet increased from 500 million to 12.5 billion mainly due to the explosive growth of smartphones and tablet PCs. Following this trend, it is expected by the year 2020 there will be even a larger number of devices connected to the Internet, estimated to be around the 25 billion [9].

1.1 Context

The Internet of Things (IoT) is a recent paradigm based on the principle of daily life objects to be equipped with sensors and networking capabilities. Thus enabling these objects to communicate under a specific protocol resulting in becoming part of the Internet. This concept leads to the evolution of the Internet into a more intricate and complex network [1].

By enabling many different devices to partake in this globalist network, the IoT will inevitably lead to the need of the development of applications. These applications in turn will make use of the enormous amount of data generated by these devices to provide better and innovative ways to share such information to citizens, companies, educational institutions or scientific institutions [19]. The scope of the IoT is immense and can be applied to many areas such as home [39] and industrial automation, healthcare systems [5], energy management and smart grids, environmental monitoring and many others. Therefore the Internet of Things can connect devices and facilities in different networks to provide efficient and secure services.

However, for this vision to materialize, the Internet of Things requires one important feature. We need to assume that various networks should coexist, and the interoperability among these networks is important for the sharing of information and their supported applications, thus interconnection is a critical architecture issue in IoT [17]. Based on the features of IoT, interconnection is a critical architecture issue, strictly speaking, these systems or applications are not “Internet of Things”, but the “Net of Things”, or can even be considered as “Net of Devices”. Thus, this paradigm should

englobe all things in large-scale networks, where various networks ought to coexist and interact with each other via middleware interfaces [38] [21].

To support this solution it needs to exist a generalized network infrastructure that integrates various networks. And by taking advantage of such structure all IoT-based systems and applications can interact and expose their services through the efficient sharing of their resources, namely information resources or network resources.

Applying to a broad context such as smart cities, if a generalized network infrastructure is implemented and is able to cover all regions in a city, applications of different scopes — smart grid, smart transportation, smart healthcare — can share their individual network infrastructures to enable data collection and information delivery [21]. According to this perspective everything that is inter-connected in the network can be used in any way possible because all the different applications can interact with each other and makes the task of sharing the resources more effective.

The project CoralTools aims to implement tools allowing scientific researchers to create, manage and interact with independent platforms of sensors, for instance, autonomous underwater vehicles (AUV) or remotely operated vehicles (ROV) related with oceanic research and exploration. These platforms can be composed by a set of devices with networking and processing capabilities thus allowing for the storage of the data that was gathered by the sensors communicating with them.

Specifically for the context of oceanic exploration, it is expected for data to be mostly generated by AUVs. The data can be generated through several sensors, namely a conductivity-temperature-depth sensor (CTD), a pH sensor or a fluorometer. Although the CoralTools project aims towards oceanic research, it is possible to explore further possible use cases. Due to the broad scope of this project it may be applied to other fields such as in citizenship science initiatives.

1.2 Problem Definition

One of the main requirements for this project is to build an interface for researchers associated to this project to access the information collected by sensors that are physically scattered and also, to provide a way to seamlessly share this data into a common point of access, taking into consideration other problems that may arise in the context of information systems, such as availability of network and quality of data.

However, by analyzing the possibility of having a wide plethora of different fields, it will become clear that there will be challenges that will arise, one of them being the identification of feasible solutions that are capable of satisfying the requirements of several possible application scenarios. This challenge inevitably led to the proliferation of different proposals for the same goal, which in turn make these very same proposals incompatible with each other. In the perspective of a IoT system this is inconceivable simply because it breaks down the possibility to create an environment where all the devices are able to intercommunicate, despite the underlying complexity of the system beneath each device. The adoption of the IoT paradigm is hindered by the lack of a clear and widely accepted model that can fulfill the needs of interoperability.

The SensorThings API is able to store the collected data from sensor activity and make it accessible through a RESTful API however, there is not a mechanism that can further allow a more in-depth analysis of such data. For most users it is important to add more significant ways to organize and present the information. If the purpose of the system is to measure the temperature in multiple locations it is not good enough to present the data as they are directly collected by the sensors. It is important to take into consideration that it is possible to rearrange the information in other way to produce higher-level outputs such as the creation of heatmaps.

For the fact of the SensorThings API being a newly proposed standard, it lacks some of the core functionalities that were previously implemented in the Sensor Observation Service [4]. The SOS has several core operations that must be provided by each implementation. One of the core operations is the *GetCapabilities* that allows to query a service for a description of the service interface and the available sensor data. This is one really important feature that enables for metadata content to be available. The lack of such information will most likely translate to further difficulties to interpret the source of the various devices.

1.3 Motivation and Goals

From the analysis of the problems described, it becomes clear that there is the need to build a system to support the gathering and storage of the devices that belong to it. This work intends to be a proof of concept, therefore it is expected to have a scaled down IoT structure at the start which progressively will evolve in size.

One of the main goals is to build a system that is generic enough so it is possible to apply it to other applications that relate to other scopes. As such it can be either applied for Health applications such as health monitoring in the context of a network of hospitals or even to other subject such as environmental applications. Considering the fact that there is an increasing interest in the topic of environment and climate perception, citizens may partake in collective initiatives on environmental monitoring and are interested in getting more information and data.

Having all of the data stored in these devices in itself does not fulfill all the requirements neither it does solve all the problems presented. In order to create a system where seamlessly all the devices seem to be connected, there is the need to build a common structure that contains information about all the other IoT devices. Ideally this application is supposed to work as a bridge between the devices that expose the SensorThings API and the end users. For a more complete approach on the solution, the application needs to have some functionalities that can be used by Catalog Services implementations, such as GeoNetwork [20].The catalog is mainly used to manage spatially referenced resources, providing powerful metadata editing and search functions. Thus meeting the requirement of having such common structure responsible for organizing the different devices that compose the whole ecosystem.

1.4 Structure

The remainder of this document has the following chapters:

- Chapter 2, “State of the Art” (p. 5), provides a literature review covering topics relevant to the dissertation such as communication protocols, mechanisms and fields of application.
- Chapter 3, “Problem Statement and Solution Proposal” (p. 25), includes the solution as well as the introduction to the architecture of the solution.
- Chapter 4 “Device Management” (p. 35), contains the implementation details of the first component of the solution, the Device Management.
- Chapter 5, “Metadata Cataloguing Implementation” (p. 41), refers to the implementation details of the proposed solution to incorporate metadata within the SensorThings API resource-based model.
- Chapter 6, “Dashboard Implementation” (p. 49), describes the development of the Visualization Tools component.
- Chapter 7, “Evaluation” (p. 65), discusses the results obtained with the use of the application, as well as the end-result of the proposed metadata model referred in Chapter 5.
- Chapter 8, “Conclusions and Future Work” (p. 77), intends to provide an overview over the entire work as well as refer to the future of the project.

Chapter 2

State of the Art

This chapter details the context of this dissertation in the form of its state of the art. The first section, does an overall overview of the most recent technologies that are used in the context of the Internet of Things. The second section, refers to the analysis of the open standards for the Internet of Things, namely the SensorThings API — that is the adopted specification for this solution and is thoroughly analyzed by its key components and features. The other standard is the Web Things API. Afterwards a comparison is made between both specifications and the analysis of the advantages and disadvantages that each specification yields. Moreover several other solutions are briefly referenced. The third section, is an overview of the Sensor Observation Service and its similarities with the SensorThings API. This section evaluates the advantages that the Sensor Observation Service has in terms of the way it exposes the metadata of the services and devices and its impact in terms of what it can enhance this project's solution. The last two sections, addresses some use cases of the SensorThings API that are applied and what can be further explored in terms of application development.

2.1 Communication Protocols

Communication is one of the main elements of IoT because there is the need to exchange information between devices [28]. With the increased use of Sensor Networks and applications applied to the most diverse environments, the need for different protocols is ever growing. Protocols that rely on wired communications are still used to connect devices since they are more reliable, secure and have higher data transfer rates [34].

For wired technologies the most common are the Universal Asynchronous Receiver and Transmitter (UART). A UART is usually an individual integrated circuit (IC) used for serial communications over a computer or peripheral device serial port [24]. They appear more commonly in the form of USB or other serial ports such as the RS232, RS485.

Wireless communication protocols in other hand are lacking in reliability over its wired counterparts. The most commonly known technologies are WiFi, Bluetooth, ZigBee but also as well as new technologies such as 6LoWPAN [7]

Taking into consideration the different applicable communication technologies for the context of the proposed solutions, it is recommended to choose a wireless communication protocol as it enables for a more flexible deployment of the IoT devices.

2.1.1 ZigBee

Introduced in 2002, ZigBee uses the IEEE 802.15.4 protocol as a base. Created for low-rate wireless private areas networks (LR-WPAN) it is one of the most used communication protocols for IoT due to its low consumption, low data rate, low cost and high message throughput. It can also provide high reliability, security, with both encryption and authentication services, works with different platforms and can handle up to 65000 nodes [7].

Advantages of ZigBee [13]:

- Low power consumption, allowing battery-powered devices to use it
- One coordinator can control a numerous amount of slaves.
- Self-organizing network capabilities
- Secure, with 128-bit AES encryption.

Disadvantages of ZigBee [13]:

- 127 bytes per message, resulting in a low data transmission varying from 20kbit/s in 868 MHz bands to 250 kbit/s in 2.4 GHZ bands
- Does not interoperate with other network protocols and lacks Internet Protocol support

2.1.2 Z-Wave

Z-wave is a short-term wireless communication technology with the advantages of low cost, low energy consumption and at the same time having great reliability [29]. The main purpose of Z-wave is providing reliable transmission between control units and one or more end-devices. However, no more than 232 nodes can be included in a Z-wave network, moreover all nodes would be controlled by the controller and have routing capability [29] [33].

2.1.3 6LoWPAN

Low-power wireless personal area networks (LoWPAN) are composed by a large number of low-cost devices connected via wireless communications [33]. In comparison with other types of networks, LoWPAN has a number of advantages (small packet sizes, low power, low bandwidth). As an enhancement, 6LoWPAN protocol was designed by combining IPv6 and LoWPAN. In 6LoWPAN, IPv6 packets can be transmitted over IEEE 802.15.4 networks. Because of the low cost and low energy consumption, 6LoWPAN is suitable to IoT, in which a large number of low

cost devices are included. 6LoWPAN have several advantages, including a great connectivity and compatibility with legacy architectures, low-energy consumption and ad-hoc self-organization.

6LoWPAN would have many advantages compared to using non-standard protocols such as ZigBee or Z-Wire. First, gateways are not necessary to translate messages between different non-standard protocols if all Wireless Sensor Networks (WSN) use the standard Internet Protocol. Flexibility is improved as new applications would not require modifications for specialized protocols within the WSN. Other advantages include rapid connectivity and compatibility with pre-existing architectures, plug-and-play installation of WSNs, and the rapid development of applications, as well as the possibility of integrating things with existing Web services that use IP [6].

To make 6LoWPAN a reality though, it would require having every node have to agree on the same protocol. Many of the current WSN protocols have limited support for being able to operate through standard interfaces [36]. Thus, complications arise for forming a WSN of different protocols that communicate to the wider Internet.

2.1.4 MQTT

Message Queue Telemetry Transport (MQTT) is a simple and lightweight protocol, and supports the network with low bandwidth and high latency. This technology can be implemented in various platforms to connect things in IoT into the Internet, therefore using MQTT as a messaging protocol between the sensing devices and the servers [21].

There exists three kind of actors in publish subscribe architecture publisher, subscriber and broker. The publisher is responsible to send the message identified by a specific subject to the broker which in turn will forward the message to everyone that is subscribed to that particular topic. The subscriber does not need to know from whom the message was originated and the publisher does not need to know to whom the message is sent [3].

This kind of architecture is suitable for the IoT since it can provide a more data oriented protocol which can reduce the burden of a constrained device for exchanging messages. However, since both publisher and subscriber do not know each other, an authentication method is required to validate the sender and receiver nodes [3].

2.1.5 Constrained Application Protocol

Constrained Application Protocol (CoAP) is a messaging protocol based on REST. Most IoT devices are resource constrained, be it in terms of storage, processing or networking. The HTTP sometimes may not be used due to its overheads. To overcome the issue, CoAP was proposed to modify some HTTP functions thus being more lightweight, therefore suitable for IoT environments. CoAP is the application layer protocol in the 6LoWPAN protocol stack, and aims to enable resource constrained devices to achieve RESTful interactions [10].

2.2 Communication Architectures and Platforms

For IoT, there are many proposed architectures all with the premise of them being essential for the whole system to work. Physical IoT devices can communicate using messaging queue systems like MQTT or through the HTTP protocol [8].

However, without having a common structure or protocol defined for these structures, each individual IoT system would be using a solution that could fit its purpose but it would lead to a constraint in terms of interoperability between different implementations. Therefore, having a common communication platform among these systems makes the development of applications simpler.

2.2.1 SensorThings API

In an attempt to solve the problem of interoperability, the Open Geospatial Consortium (OGC) developed a specification for an Internet of Things protocol. The OGC SensorThings API provides an open, geospatial-enabled and unified way to interconnect the IoT devices, data and applications over the Web [26]. At a higher level the API provides two core functionalities. The first core functionality is the Sensing part. The Sensing profile provides a standard way to manage and retrieve observations and metadata from heterogeneous IoT sensor systems.

The SensorThings API simplifies and accelerates the development of IoT applications [22]. Developers can use this open standard to connect to various IoT devices and build applications without worrying about the daunting heterogeneous protocols of the different IoT devices, gateways and services. Moreover the device manufacturers can also use this API to be embedded within various IoT hardware and software platforms, so that the various IoT devices can connect to other OGC standard-compliant servers around the world [26].

2.2.1.1 SensorThings API Data Model

As previously stated, the Sensing part allows IoT devices and applications to perform the basic CRUD operations (Create, Read, Update and Delete) in IoT data and metadata in a SensorThings service. The data model is designed based on the ISO/OGC Observation and Measurement (O&M) model [26]. The model is centered around *observations* and their *results* whose value is an estimate of a property of the observation target (see Figure 2.1). An *observation* instance is classified by the time it has occurred, the *resultTime* and the *phenomenonTime*, its corresponding *FeatureOfInterest*, and the *Datastream* is has been associated to. Additionally the concept of *Things* are also modeled in the SensorThings API following the ITU-T definition: “an object of the physical world (physical things) or the information world (virtual things) that is capable of being identified and integrated into communication networks” [14].

To provide geospatial information, there is also the *Locations* that contain geographical data about the *Things*. Additionally a *Thing* may change locations from time to time, therefore old locations will generate *HistoricalLocations* entities. A *Datastream* is a collection of *Observations*.

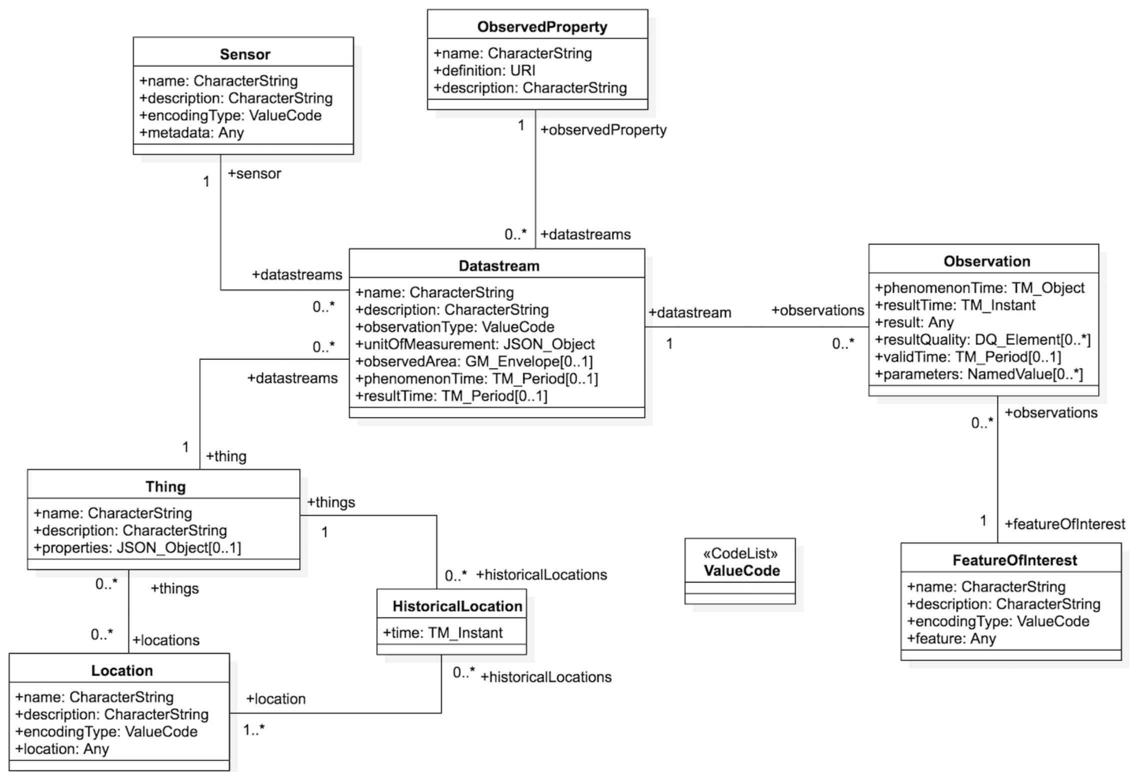


Figure 2.1: SensorThings API Data Model [26]

The *Datastreams* are also grouped by their corresponding *ObservedProperty* and *Sensor*. An *Observation* is an event performed by a *Sensor* that produces a *result* whose value is an estimate of an *ObservedProperty* of the *FeatureOfInterest* [26].

2.2.1.2 SensorThings API Sensing Entities

In this section it will be thoroughly explained the properties in each entity type and the direct relation to the other of the entity types.

Thing A *Thing* is an object of the physical world or the information world that is capable of being identified and integrated into communication networks[14].

```
1 {
2   "@iot.id": 1,
3   "@iot.selfLink": "http://example.org/v1.0/Things(1)",
4   "Locations@iot.navigationLink":  "Things(1)/Locations",
5   "Datastreams@iot.navigationLink": "Things(1)/Datastreams",
6   "HistoricalLocations@iot.navigationLink": "Things(1)/HistoricalLocations",
7   "name": "Oven",
8   "description": "This thing is an oven.",
9   "properties": { "owner": "Noah Liang", "color": "Black"}
```

```
10 }
```

Listing 2.1: Example of a Thing Entity

Location The *Location* entity describes the physical location of the *Thing*. Usually the *Location* entity is defined as being always the last known location of the *Thing*.


```
1 {
2   "@iot.id": 1,
3   "@iot.selfLink": "http://example.org/v1.0/Locations(1)",
4   "Things@iot.navigationLink": "Locations(1)/Things",
5   "HistoricalLocations@iot.navigationLink": "Locations(1)/HistoricalLocations",
6   "encodingType": "application/vnd.geo+json",
7   "name": "CCIT",
8   "description": "Calgary Center for Innvative Technologies",
9     "location": {
10    "type": "Feature",
11    "geometry":{
12      "type": "Point",
13      "coordinates": [-114.06,51.05]
14    }
15 }
```


Listing 2.2: Example of a Location Entity

HistoricalLocation The *HistoricalLocation* refers to the times of the current and previous locations of the *Thing*.


```
1 { "value":
2   [ {
3       "@iot.id": 1,
4         "@iot.selfLink":"http://example.org/v1.0/HistoricalLocations(1)",
5     "Locations@iot.navigationLink": "HistoricalLocations(1)/Locations",
6     "Thing@iot.navigationLink": "HistoricalLocations(1)/Thing",
7     "time": "2015-01-25T12:00:00-07:00"
8   }, {
9     "@iot.id": 2,
10        "@iot.selfLink": "http://example.org/v1.0/HistoricalLocations(2)",
11        "Locations@iot.navigationLink": "HistoricalLocations(2)/Locations",
12        "Thing@iot.navigationLink": "HistoricalLocations(2)/Thing",
13    "time": "2015-01-25T13:00:00-07:00"
14  } ],
15  "@iot.nextLink":"http://example.org/v1.0/Things(1)/HistoricalLocations?$skip=2&
        $top =2"
16 }
```

Listing 2.3: Example of a HistoricalLocation Entity

Datastream A *Datastream* groups a collection of *Observations* measuring the same *ObservedProperty*.

```
1  {
2    "@iot.id": 1,
3      "@iot.selfLink": "http://example.org/v1.0/Datastreams(1)",
4      "Thing@iot.navigationLink": "HistoricalLocations(1)/Thing",
5      "Sensor@iot.navigationLink": "Datastreams(1)/Sensor",
6      "ObservedProperty@iot.navigationLink": "Datastreams(1)/ObservedProperty",
7      "Observations@iot.navigationLink": "Datastreams(1)/Observations",
8      "name": "oven temperature",
9      "description": "This is a datastream measuring the air temperature in an oven."
          ,
10     "unitOfMeasurement": { "name": "degree Celsius", "symbol": "C",
11   "definition": "http://unitsofmeasure.org/ucum.html#para-30" },
12   "observationType": "http://www.opengis.net/def/observationType/OGC- OM/2.0/
         OM_Measurement",
13     "observedArea": { "type": "Polygon",
14     "coordinates": [[[100,0],[101,0],[101,1],[100,1],[100,0]]]
15   },
16   "phenomenonTime": "2014-03-01T13:00:00Z/2015-05-11T15:30:00Z",
17     "resultTime": "2014-03-01T13:00:00Z/2015-05-11T15:30:00Z"
18 }
```

Listing 2.4: Example of a Datastream Entity

Sensor A *Sensor* is an instrument that observes a property or phenomenon with the goal of producing an estimate of the value of the property measured. However, in some cases the Sensor in this data model can also be seen as the Procedure[15].

```
1 {
2   "@iot.id": 1,
3   "@iot.selfLink": "http://example.org/v1.0/Sensors(1)",
4   "Datastreams@iot.navigationLink": "Sensors(1)/Datastreams",
5   "name": "TMP36",
6   "description": "TMP36 - Analog Temperature sensor",
7   "encodingType": "application/pdf",
8   "metadata": "http://example.org/TMP35_36_37.pdf"
9 }
```

Listing 2.5: Example of a Sensor Entity

ObservedProperty An *ObservedProperty* specifies the phenomenon of an *Observation*.

```
1 {
2   "@iot.id": 1,
3     "@iot.selfLink": "http://example.org/v1.0/ObservedProperties(1)",
4     "Datastreams@iot.navigationLink": "ObservedProperties(1)/Datastreams",
5   "description": "The dewpoint temperature is the temperature to which the air must
        be cooled, at constant pressure, for dew to form. As the grass and other
       objects near the ground cool to the dewpoint, some of the water vapor in the
       atmosphere condenses into liquid water on the objects.",
6   "name": "DewPoint Temperature",
7   "definition": "http://dbpedia.org/page/Dew_point"
8 }
```

Listing 2.6: Example of a ObservedProperty Entity

Observation An *Observation* is the act of measuring or otherwise determining the value of a property [15].

```
1 {
2   "@iot.id": 1,
3   "@iot.selfLink": "http://example.org/v1.0/Observations(1)",
4   "FeatureOfInterest@iot.navigationLink": "Observations(1)/FeatureOfInterest",
5   "Datastream@iot.navigationLink":"Observations(1)/Datastream",
6   "phenomenonTime":"2014-12-31T11:59:59.00+08:00",
7   "resultTime": "2014-12-31T11:59:59.00+08:00",
8   "result": 70.4
9 }
```

Listing 2.7: Example of a Observation Entity

FeatureOfInterest An *Observation* results in a value being assigned to a phenomenon. The phenomenon is a property of a feature — the *FeatureOfInterest* of the *Observation* [15].

```
1 {
2   "@iot.id": 1,
3   "@iot.selfLink": "http://example.org/v1.0/FeaturesOfInterest(1)",
4   "Observations@iot.navigationLink": "FeaturesOfInterest(1)/Observations",
5   "name": "Weather Station YYC.",
6   "description": "This is a weather station located at the Calgary Airport.",
7   "encodingType": "application/vnd.geo+json",
8   "feature": {
9     "type": "Feature", "geometry": {
10      "type": "Point", "coordinates": [-114.06,51.05]
```

```
11      }
12    }
13  }
```

Listing 2.8: Example of a FeatureOfInterest Entity

2.2.2 WebThings API

The Web Thing API[1] is expected to be submitted for approval to the World Wide Web Consortium[2]. The goal of Web Thing API is to also develop a data model and API that can be used in the context of IoT to describe physical devices in a JSON format. Similarly to SensorThings, this API can also be used with the communication platforms like MQTT to create a more standardized way of establishing communication structures and protocols for IoT systems.

2.2.2.1 WebThings API Data Model

The WebThings data model as depicted in Figure 2.2 has four main entities.

Things —- A web Thing can be a gateway to other devices that don't have an internet connection. This resource contains all the web Things that are proxied by this web Thing. This is mainly used by clouds or gateways because they can proxy other devices.

Model —- A web Thing always has a set of metadata that defines various aspects about it, such as its name, description, or configurations.

Properties —- A property is a variable of a web Thing. Properties represent the internal state of a web Thing. Clients can subscribe to properties to receive a notification message when specific conditions are met; for example, the value of one or more properties changed.

Actions —- An action is a function offered by a web Thing. Clients can invoke a function on a web Thing by sending an action to the web Thing. The actions may refer to opening or closing a garage door, or enabling or disabling a smoke alarm. The direction of an action usually starts from the client to the Web Thing. Actions are the public interface of a Web Thing whereas the properties represent their private fields. In the sense that being private means that only privileged parties are the ones who have access to it. But limiting access to actions – that is, the public interface – also allowing to implement various control mechanisms for external requests such as access control, data validation and updating several properties atomically.

With the analysis of the data model, it becomes clear that the main goal with the use of this specification is to be able to interface with IoT devices, thus providing a real-time mechanism to allow the creation of multiple requests and to enable the creation of events for notifications.

[1] https://iot.mozilla.org/wot/
[2] https://www.w3.org/

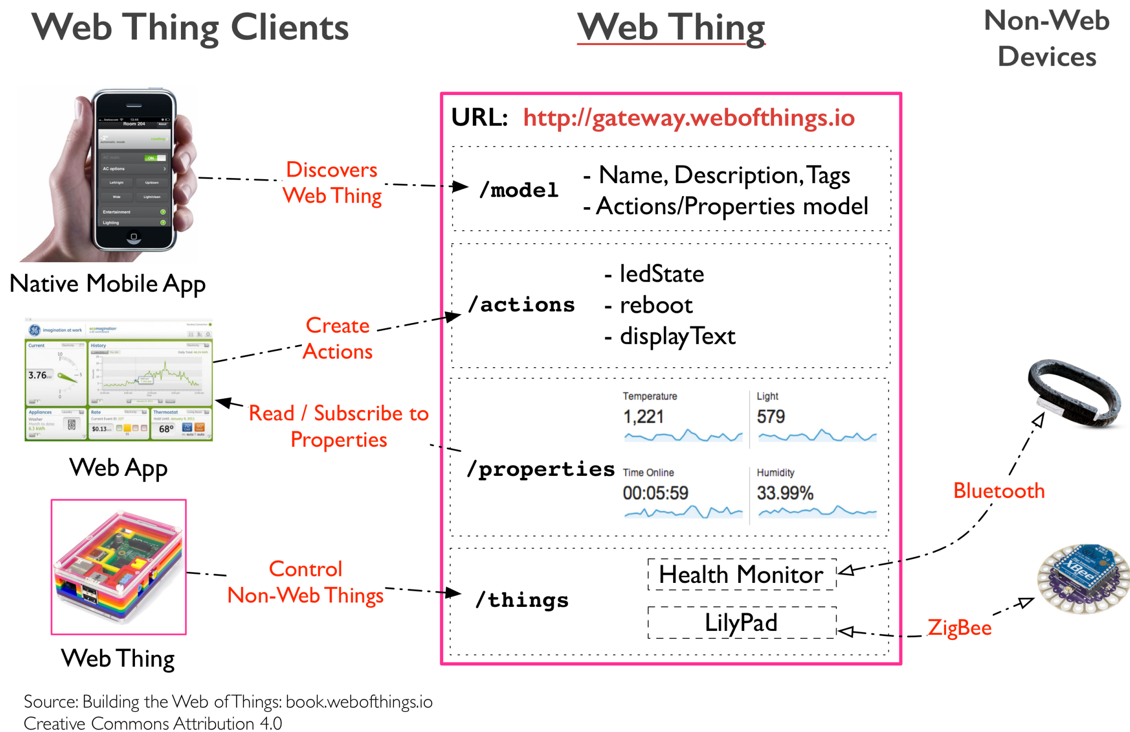


Figure 2.2: Web Things API Data Model

2.2.3 Comparison between SensorThings API and Web Thing API

While being undeniably that these specifications are creating a standard to address the problem of interoperability between IoT systems, it is not true to say that these specifications are meant to address the same purpose.

In the previous sections, it was mentioned that the SensorThings API is derived from the *Observations & Measurements* (O&M) model. Therefore, its logical structure is different from the Web Things API, as it provides a more complete representation of how the values are organized as it uses several entities such as *Datastreams*, *Locations* and the *Observations*. In comparison, the Web Thing API it is not as detailed. For example, the following snippet is the response of a *Thing* from the Web Thing API.

```
1  [
2    {
3      "name":"WoT Pi",
4      "type": "thing",
5      "description": "A WoT-connected Raspberry Pi",
6      "properties": {
7        "temperature": {
8          "type": "number",
9          "unit": "celsius",
10         "description": "An ambient temperature sensor",
```

```
11          "href": "/things/pi/properties/temperature"
12        },
13        "humidity": {
14          "type": "number",
15          "unit": "percent",
16          "href": "/things/pi/properties/humidity"
17        },
18        "led": {
19          "type": "boolean",
20          "description": "A red LED",
21          "href": "/things/pi/properties/led"
22        }
23      },
24      "actions": {
25        "reboot": {
26          "description": "Reboot the device"
27        }
28      },
29      "events": {
30        "reboot": {
31          "description": "Going down for reboot"
32        }
33      }
34    }
35  ]
```

Listing 2.9: Example of a Web Thing API GET response to a Thing Entity

It is possible to state that inside the whole structure of a *Thing* it is possible to obtain information about the **properties**, and these properties are equivalent to the *Datastreams* in the sense that in SensorThings API they contain information about the *Observations* to a given physical property such as the air temperature.

After the comparison between these specifications, it is clear that for this work it is not feasible to use Web Thing API because a paramount feature is a data model that contains data of the **Measurements** in a given geographical location recorded by the sensors.

For the context of a *Smart Home* application-based solution, Web Thing API would be a better choice because its data model is modeled in a way that there is not need to store all the data gathered by the sensors, on the other hand as it already features a tasking capability it might prove to be more useful rather than SensorThings API for the specific use case.

2.2.4 oneM2M

The oneM2M is the global standards initiative for Machine to Machine Communications (M2M) and the Internet of Things. It was established in July 2012 by Europe, the United States, China, Japan and South Korea telecommunication standard organizations and attracted over 200 members. Its architecture, depicted in Figure 2.3 comprises three entities: Application Entity (AE), Common

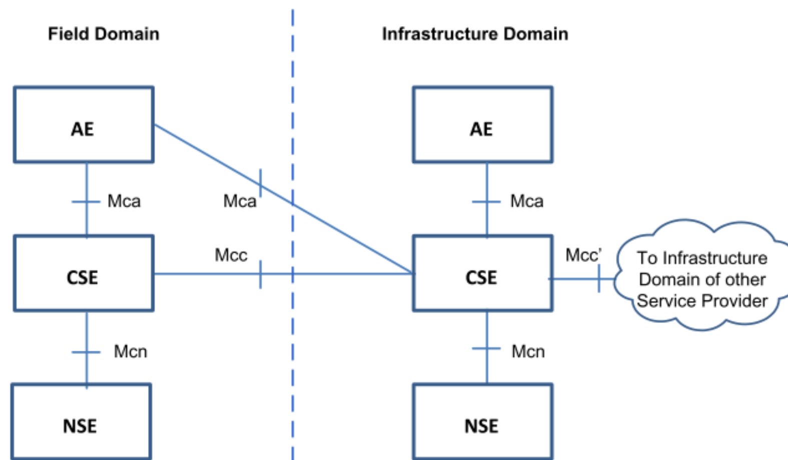


Figure 2.3: oneM2M Functional Architecture [37]

Services Entity (CSE) and Network Services Entity (NSE), and three interface points: Mca, Mcc and Mcn, spanning across a field domain and an infrastructure domain. The CSE supports many common services to AEs and to other CSEs such as discovery, security, device management, etc. The oneM2M adopts a resource-based information model. All entities in the oneM2M System, such as AEs, CSEs and data, are represented as resources, and its resources form a hierarchical tree called resource tree and can be manipulated by RESTful APIs [37] [27].

For this project, oneM2M might not be the best option because comparing with the SensorThings API it does not provide a data model that is well suited to support storage of data generated through sensor activity. Thus, being more guided towards machine-to-machine communication.

2.2.5 AIOTI High-Level Architecture

The *AIOTI High-Level Architecture* (HLA) [23] Functional Model describes functions and interfaces within the domain. It also follows a layered approach composed by three distinct layers each of them being a cohesive set of services:

- **Application layer** — responsible to establish process-to-process communications.
- **The IoT layer** — responsible for data storage and sharing, also exposes them to the application layer via APIs while using the underlying Network layer's services
- **The Network layer** — accounts for providing the connectivity and data forwarding between entities thus allowing communication

However, the AIOTI HLA does not stipulate details on implementation or deployment therefore not being a viable approach.

2.2.6 RIOT-OS

RIOT-OS[3] is an open platform for devices with limited resources [2], it is maintained by the iNET research group that co-founded and develops it. This lightweight operating system (OS) is intended to support most embedded, low-power devices and other micro-controller architectures. This project aims to implement all relevant open standards supporting an IoT environment that takes into account security, connection and durability [18].

2.3 Semantic Interoperability in the Internet of Things

The Sensor Observation Service (SOS) standard is applicable to use cases in which sensor data needs to be managed in an interoperable way. This standard defines a Web service interface which allows querying observations, sensor metadata, as well as representations of observed features. Furthermore, this standard defines means to register new sensors and to remove existing ones. Also, it defines operations to insert new sensor observations. This standard defines this functionality in a binding independent way; two bindings are specified in this document: a Key-Value Pair (KVP) binding and a SOAP binding[4].

The functionality of the Sensor Observation Service is to provide standardized access to measured sensor observations as well as sensor descriptions. Much like to SensorThings API it also abides by the Observations & Measurements (O&M) standards. The main reason this specification is not considered for this project is simply due to the fact that it was not developed with the resource-constrained environment of the IoT and its data model is not best suited for resource constrained devices. However, there are some benefits inherent to SOS that are non existent in the SensorThings API specification.

Analyzing the SOS specification it is possible to denote that there are three main operations:

- **GetCapabilities** — provides access to metadata and detailed information about the operations available by an SOS server.
- **DescribeSensor** — enables querying of metadata about the sensors and sensor systems available by an SOS server.
- **GetObservation** — provides access to observations by allowing spatial, temporal and thematic filtering.

Taking into account that SensorThings API follows a Resource-Oriented Architecture (ROA) because it exposes the API through Resource Path made accessible by URI it would not make sense to implement the *GetCapabilities* as depicted in the SOS specification. In order to retrieve observations using the SensorThings API, the user simply has to send a GET request to the desired endpoint.

But to this work it is not merely enough to have access to the observations of the sensors. Assuming that the objective is to have a multi-platform IoT environment it is necessary for all the different platforms to communicate with each other and most importantly for each different

[3] https://riot-os.org

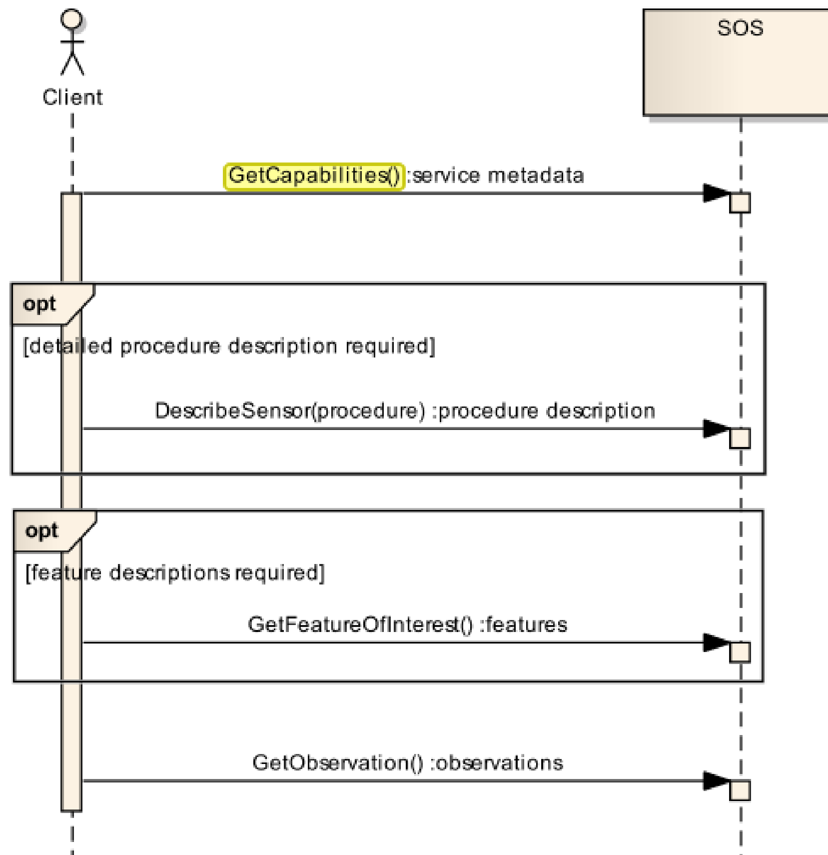


Figure 2.4: Chain of requests for observation retrieval on a SOS platform [4]

organization operating a platform to share more information regarding the metadata authors, the point of contact, description of the service, the exposed operations that are available, the temporal extent and geographical information as well as many other informations that are by default included and expected to exist in a SOS implementation.

The typical workflow to a SOS server, seen in Figure 2.4, for observation retrieval for the case of SOS would be firstly the user to request a listing of available data by sending the *GetCapabilities* request to the server. Followed by an optional *DescribeSensor* or *GetFeatureOfInterest* to find further details about particular procedures or features. Finally, the user would issue a *GetObservation* request to retrieve the observations.

The expected result of this operations would be the same as the user simply issuing a GET request to the available *Datastreams* to the SensorThings API endpoint and then simply following with more GET requests to a specific *Datastream* for its respective *Observations*.

To conclude, the usage of a Resource Oriented Architecture is more favorable for an IoT environment since it simplifies the whole sequence (as depicted in Figure 2.4) making it more lightweight and easier to post-process. According to Guinard et al [12] after conducting an experiment using Web Things API described in Section 2.2.2, another Resource-Oriented Architecture framework,

the results suggested that the verbosity of the HTTP protocol does not prevent highly efficient applications to be implemented, even when low-power wireless nodes communicate using HTTP in place of highly optimized and compressed messages. However, when devices are connected to a power source and do not rely on batteries and the latency is not too high, the advantages of HTTP outweigh the loss in performance and latency.

2.4 SensorThings API Server Implementations

The first part of the SensorThings API specification was released in 2015, and despite being a recent specification it does not exclude the fact that there are several implementations of such specification.

2.4.1 FROST

FROST-Server (FRaunhofer Opensource SensorThings Server) is an open-source[4] implementation of the standard, developed by the German research institute Fraunhofer IOSB [16], to cover their need for a standards-based, easy to use sensor management and sensor-data storage platform, for use in various research programs. For data persistence it uses a PostgreSQL database management system (DBMS).

2.4.2 SensorUp SensorThings

SensorUp, based in Calgary, Canada developed the first compliant SensorThings implementation and this implementation is considered as the SensorThings reference implementation. It is a Java-based implementation and uses a PostgreSQL database. In addition to server development, SensorUp also provides multiple clients to make SensorThings easier to use for client developers.

2.4.3 GOST

GOST is an open source implementation of the SensorThings API in the Go programming language initiated by Geodan[5]. It contains an easily deployable server software and a JavaScript client. At the time of writing it is still in development, although a first version can already be downloaded and deployed. The software may be installed on any device supporting Go and, by default, stores sensor data in a PostgreSQL database.

The Figure 2.5 represents a datastream in the GOST dashboard.

2.4.4 Mozilla

Mozilla has a Node implmentation of SensorThings. The implementation is open source and has passed almost all the OGC test suite tests. This implementation uses PostgreSQL for the persistence of data. However, the development is not active since February 2017.

[4]https://github.com/FraunhoferIOSB/FROST-Server
[5]https://www.geodan.nl/

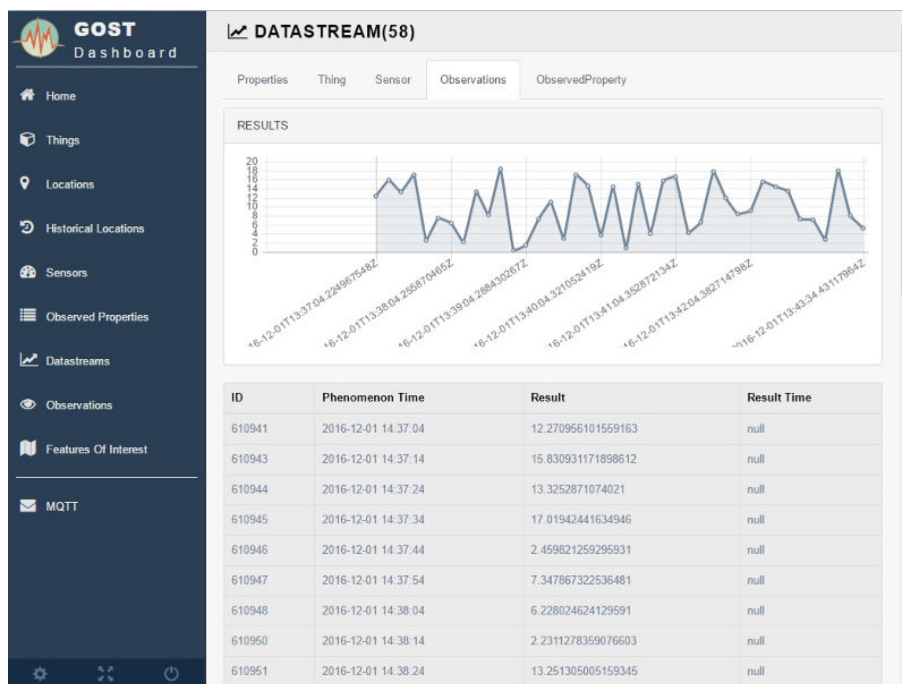


Figure 2.5: GOST Dashboard [11]

2.4.5 CGI Kinota Big Data

CGI developed a modular implementation of SensorThings named Kinota Big Data. Kinota is designed to support different persistence platforms to relational database management systems and to NoSQL databases. The current implementation supports Apache Cassandra. However, Kinota only implements a subset of the SensorThings requirements. It is also written in the Java programming language.

2.5 Applications using SensorThings API

SensorUp[6] premise is to aggregate the information from all different kinds of sensors in a single platform, by using the SensorThings API. According to SensorUp they have the most complete IoT Platform and compliant implementation of the SensorThing API [16]. Moreover they have also developed an API and software development kits (SDK) to support developers to build IoT-based applications.

2.5.1 SensorThings Admin Dashboard

SensorUp has developed a dashboard that provides an easy access to visualize the entities that compose the SensorThings API. However this application fails short when it needs to assess the

[6] https://www.sensorup.com/

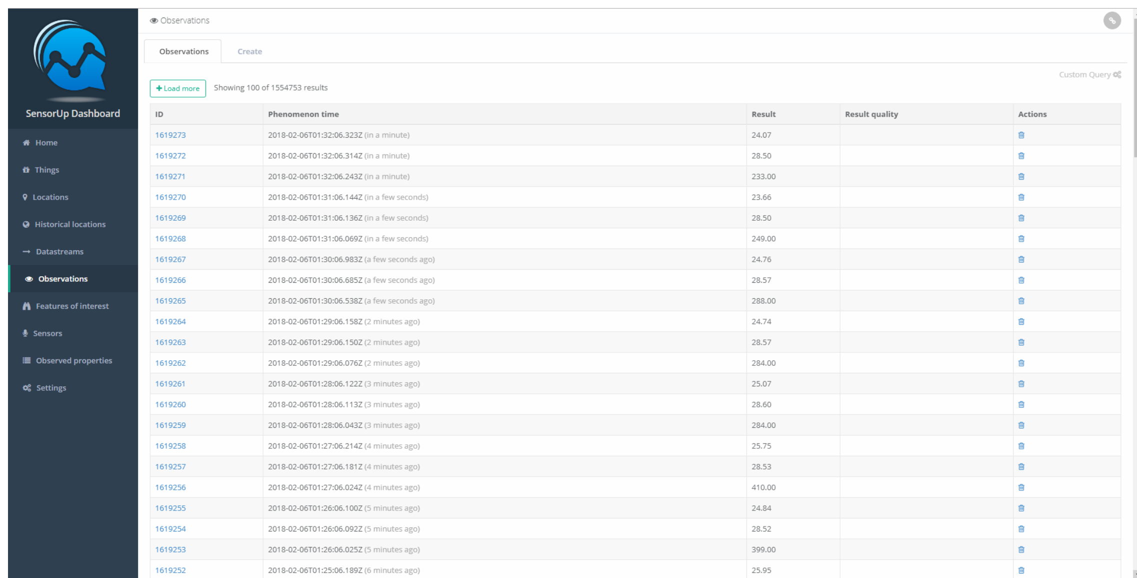


Figure 2.6: SensorThings Admin Dashboard [32]

problems of metadata management and organize its composing *Things* similarly to a catalog service. Moreover, this application only displays the *observations* in tables, shown in Figure 2.6.

2.5.2 SensorThings Map

Besides the administration dashboard, SensorUp has also developed an interactive map to represent the locations of the *Things* by taking advantage of the geographic information supported by the SensorThings API. By using this map it is easy to use and query data of a single *Thing*, furthermore the observation data is presented in a timeseries chart and it is also possible to visualize simultaneously in the same chart other datastreams that belong to the focused *Thing*. For the solution that is later proposed in this document, this map fits as an appropriate inspiration as a possible approach for the data visualization component as similar to Figure 2.7.

2.5.3 Air quality monitoring after wildfires

A distributed, shared, network of PM 2.5 sensors [31] were deployed in St. Albert[7]. The sensors are small and lightweight, and connected to a regular WiFi network, facilitating its deployment. The aim of this intervention was to further gather information about air quality and make it publicly available over a website[8]. This project proved that the usage of the SensorThings is a great fit for, not just the process of collecting air quality data, but also to manage the sensor data and environmental data. Additionally the data that is presented to the end-user does not mean it had been validated nor post-processed, as this was a crowdsourced data specifically aimed for citizenship science initiatives.

[7] https://www.sensorup.com/blog/2017/07/12/alberta-residents-monitor-bc-wildfire-effects-with-s
[8] https://smartstalbert.sensorup.com/

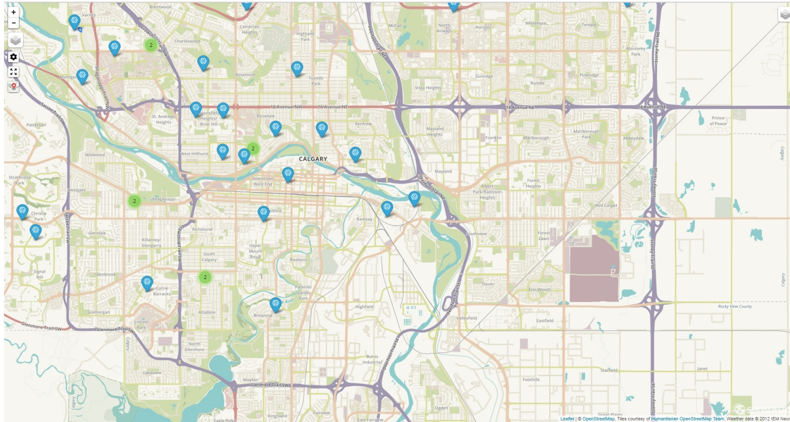


Figure 2.7: SensorThings Map

An important aspect in the web application that is relevant to the project described in this document is the use of heatmaps to take advantage of SensorThings API geographic functionalities to organize the information and present it as a temporal evolution through a web application as depicted in Figure 2.8.

2.6 Conclusions

By analyzing the SensorThings API, it is possible to conclude that it is in fact the most suited specification for this project. It features a RESTful API that will make the communication between the several components of the solution in an easier way. Furthermore, after reviewing the data model it is possible to detect the functionalities supported and what may be missing and what needs to be implemented to complement it.

Also, by studying the WebThings API it was possible to compare it with the SensorThings API and evaluate them to determine which one was better to use for the solution. Naturally, as the SensorThings API is more oriented for geospatial data, it is the best fit. Nonetheless, it was also important to analyze the WebThings API for some of its important features such as its WebSocket mechanism.

During the process of analysis of existing applications that use the SensorThings API, it was possible to understand how it can be used as a way to provide access to information. Moreover, some of the web applications were good representations on how the final solution would be in terms of data representation, as it can be presented in many degrees and ways such as maps, or charts to provide higher-level outputs.

Figure 2.8: Animated map based on a timeseries evolution of Saint Albert air quality monitoring

Chapter 3

Problem Statement and Solution Proposal

The goal of this chapter is to describe the issues that have been detected and how they are planned to be solved. The solution itself will be thoroughly described as well as its components.

3.1 Current Issues

With using such recent specification (SensorThings API) is that there is no actual Catalog Service that implements a protocol to operate with this newly specification. The catalog service is made up of records that describe geospatial data (e.g. Keyhole Markup Language (KML)), geospatial services (e.g. Web Map Service (WMS)), and related resources. This in turn means that if the actual goal is to use a Catalog Web Service it will inevitably lead to the implementation of a similar one based out of GeoNetwork[1].

After a throughout analysis of the GeoNetwork application, one of the most important features is the capability to automatically gather information of the services that are exposed by the devices. The harvester is responsible to automatically communicate with the endpoint running the SensorThings API server implementation and retrieve the metadata inherent to the services available from that device. In a typical OGC CSW interface such as the Sensor Observation Service, this would simply mean that the harvester would invoke the service *GetCapabilities* and it would be sufficient to collect all the information needed.

Hence, this is the desired behavior that the proposed solution should have. To support this, the desired behaviour of the functionality to be developed should have the mechanism of gathering service metadata from the IoT devices and output it into a machine-readable XML document and at the same time to take advantage of the resource-oriented approach that the REST architure enables.

[1] https://geonetwork-opensource.org/

Previously the *GetCapabilties* request was based, but not only, on SOAP (Simple Object Access Protocol). SOAP is a protocol specification for exchanging structured information in the implementation of web services. Moreover, SOAP uses XML as the typical format of the content that is exchanged between the different processes, in this strict case between a hypothetical GeoNetwork harvester and a Sensor Observation Service implementation exchanging data.

SOAP is based on a service-oriented architecture (SOA). This architecture is a style of software design where services are provided to the other components by application components, through a communication protocol over a network, in this case, the HTTP protocol. However, the SensorThings API does not follow the same principles as the ones followed by the specification of the Sensor Observation Service. The most recent specification has a resource-oriented architecture (ROA). The resource-oriented architecture is a style of software architecture and programming paradigm for designing and developing software in the form of resources with "RESTful" interfaces. These resources are software components which can be reused for different purposes.

One important guideline for this architecture is the avoidance of RPC-style APIs, favoring in turn the Resources and Protocols. This means that for this specific architecture the use of Remote Procedure Calls such as the *GetCapabilities* does not fully meet its architectural principles. One possible solution is to understand how the data model is structured by the SensorThings API specification and achieve the same results of a *GetCapabilities* request but in an architecture that favors the request of the data stored that refers to the very same metadata of the service [26] [30].

Moreover, from the analysis of the current available applications of the SensorThings API described in Chapter 2 there isn't yet implementation that addresses an application that uses more than one endpoint address (i.e. another SensorThings API instance). In order to enable a multi-platform IoT environment it is expected to have a multitude of devices, each one being denominated as a platform. The aspect of interoperability is crucial, because it allows for these devices to communicate seamlessly. This is solved by assuming that all of the devices that compose this IoT environment are running a SensorThings API instance. Furthermore, in order to produce higher-level outputs of data there isn't anything that will aggregate the information contained on the different endpoints, and that is another issue that will be addressed in this work.

3.2 User Stories

As part of the development of the whole system composed by the CoralTools Web Application and the CoralTools Backend Application there are some functionalities to be implemented in order to provide the user with an interactive application as well as to support and prove the enablement of a multi-platform IoT environment. The user stories are described in Table 3.1.

User Story	Name	Description
US001	Login	As a registered user I want to input my credentials to get an authentication token so that I can perform more requests to the backend application.
US002	Register	As a non registered user I want to register myself so that I will be able to login.
US003	Register Node	As a user I want to add an endpoint address so that I can interact with it later.
US004	Register Thing	As a user I want to add a *Thing* entity so that I can retrieve it and view its metadata in the Catalog.
US004	View Catalog	As a user I want to view the metadata records so that I can get more information about the service and operation the service implements.
US005	View Heatmap	As a user I want to view a heatmap representation of a series of datastreams in a timeseries animation so that I can understand the evolution of the Observation data presented in colors representing a third dimensional variable.
US006	View Clustered View	As a user I want to view a map with the observations presented in clusters so that I can understand the density of *Observation* entities in a certain point of the map.
US007	View Historical Locations	As a user I want to view a map with the positions of the *Thing* entity over time so that I can know which positions it traveled through.
US008	View Datastream By Timeline	As a user I want to view the datastream in the form of a timeseries chart so that I can get a graphical representation of a datastream for the time span that I specified.
US009	View Datastream Availability	As user I want to view the periods of time the datastream has *Observations* so that I can know when there were periods of unavailability of *Observation* results.
US010	Associate an authorization token to a thing	As a user I want to be able to add an authentication token to an endpoint address so that when I issue a request to an endpoint address it will be accepted by the issued entity.
US011	Manage Users	As a user I want to be able to manage users that are registered in the platform so that I can change their access and permissions in the web application.
US012	CRUD operations over SensorThings API entities	As a user I want to be able to create, delete, update and view the entities of a given endpoint address.

Table 3.1: User Stories for the CoralTools Application

3.3 Solution Proposal

From the analysis of the problem described previously in Section 3.1, it becomes clear that there is the need to build a tooling mechanism to proceed to the storage of relevant information of the devices, namely the endpoint address, that will be present in the system. Thus, the planned approach, that intends to be a proof of concept to the solution of a larger problem, is to firstly scale down the IoT structure in order to progressively test the system.

The structure that is being planned is composed by several distinct components that interact seamlessly. At the sensing device level, there are devices that have sensing capabilities to gather measurements of temperature, humidity, atmospheric pressure or air quality. These sensors ought to be connected to a device running an instance of SensorThings API. For this solution, it was used several Raspberry Pi 3 with the SenseHat module. The SenseHat is an add-on board with sensing capabilites. Additionally, by using the Raspberry Pi 3, it is possible to take advantage of its processing power and networking capabilities to abstract this particular device into an IoT platform.

The Raspberry Pi ought to have a connection to the Internet to be accessed by other external entities. Moreover, the Raspberry Pi has to be serving a SensorThings API endpoint. This is done by installing a SensorThings API server implementation. Though, in order to correctly set up the server, there is the need to also have a relational database responsible to support the data model and the data being stored, such as *Datastreams* and *Observations*. For data acquisition by the sensors, there is a Python script running in these devices that is responsible to retrieve the measurement results and send an HTTP request to the server endpoint running the SensorThings API, that is usually running in the same device. One of the advantages of this approach is that despite the device may be disconnected from the Internet, the results and measurements are still able to be registered and recorded into the database without compromising the process of collecting observations.

In order to create a system where seamlessly all the devices are accessible, there is the need to build a common structure that contains information about all the other IoT devices. This application is going to fulfill the role of a bridge for the devices that expose the SensorThings API and the end users.

Through the analysis of the SensorThings API data model it is possible to infer that there are some fields that can be used properly with the intent to store metadata information. It is the case of the field Properties in the *Thing*, that is of type *JSON_Object*, giving the entity that is responsible for managing the device the freedom to store any type of object that refers to the Thing. This will grant enough flexibility to contain metadata information about the service. Regardless, the problem is not solved as there is also another important aspect of the harvester, that is the interoperability between different applications. These applications such as GeoNode or GeoNetwork use common OGC standards such as the Cataloguing Service for the Web (CSW). Therefore the proposed solution consists on storing relevant metadata in the SensorThings devices, and additionally develop a mechanism that is responsible to retrieve it and generate a response that is compliant with such specifications in order to be recognized by these platforms (GeoNode, GeoNetwork).

After the step of having the IoT devices correctly labeled under the catalog service, there is

also the need for the platform to serve as a management tool for the device properties. Taking advantage of the RESTful API it is possible to remotely update the IoT devices simply using the HTTP protocol whenever the device has Internet connection. This takes into consideration the inevitable need of having to update the information of the device and can be achieved without having to directly interact with the device.

Finally, the solution should also take into consideration the visualization of the observation data. There is the precondition that this data is scattered between all the sensing devices and may not be entirely present in a single database. Therefore, to present the data to the end users it was developed an web application that provides different visualization options for the data, in graphical form such as maps and charts but also in tabular representation. For different endpoints to be queried at the same time, the web applications knows a set of endpoints on which it can query and seamlessly retrieve the observation data. These set of operations is supported by a backend application responsible for the storage of the endpoint data and to work as a bridge between the devices running the SensorThings API and the web application that presents the data and also allows for the management of the devices, such as the SensorThings API entities configuration.

The proposed solution architecture is depicted in Figure 3.1.

3.3.1 CoralTools Backend Application

The CoralTools Backend Application is one of the three components that composes the solution. This component is independent from all the other components as its main function is to be the link between the SensorThings API endpoints and the web application. It is a RESTful API responsible to handle requests performed by the web application. The list below enumerates the different modules of the CoralTools Backend Application.

- **Manage Users**
- **Websocket Communication**
- **Generate GetCapabilities document**
- **Proxy requests with authentication token**
- **Aggregate results from different endpoints**

3.3.1.1 Manage Users

The management of users by the backend application is used to give access to the web application. The web application allows for the user to register and to be recognized within the platform and most importantly to get a token that will allow him to authenticate in the backend application and perform requests. Moreover there are two distinct types of *Users*.

3.3.1.2 Websocket Communication

When the user has some interaction in the web application, sometimes there might be the need to communicate with other users that are using the application in other machine to be notified of

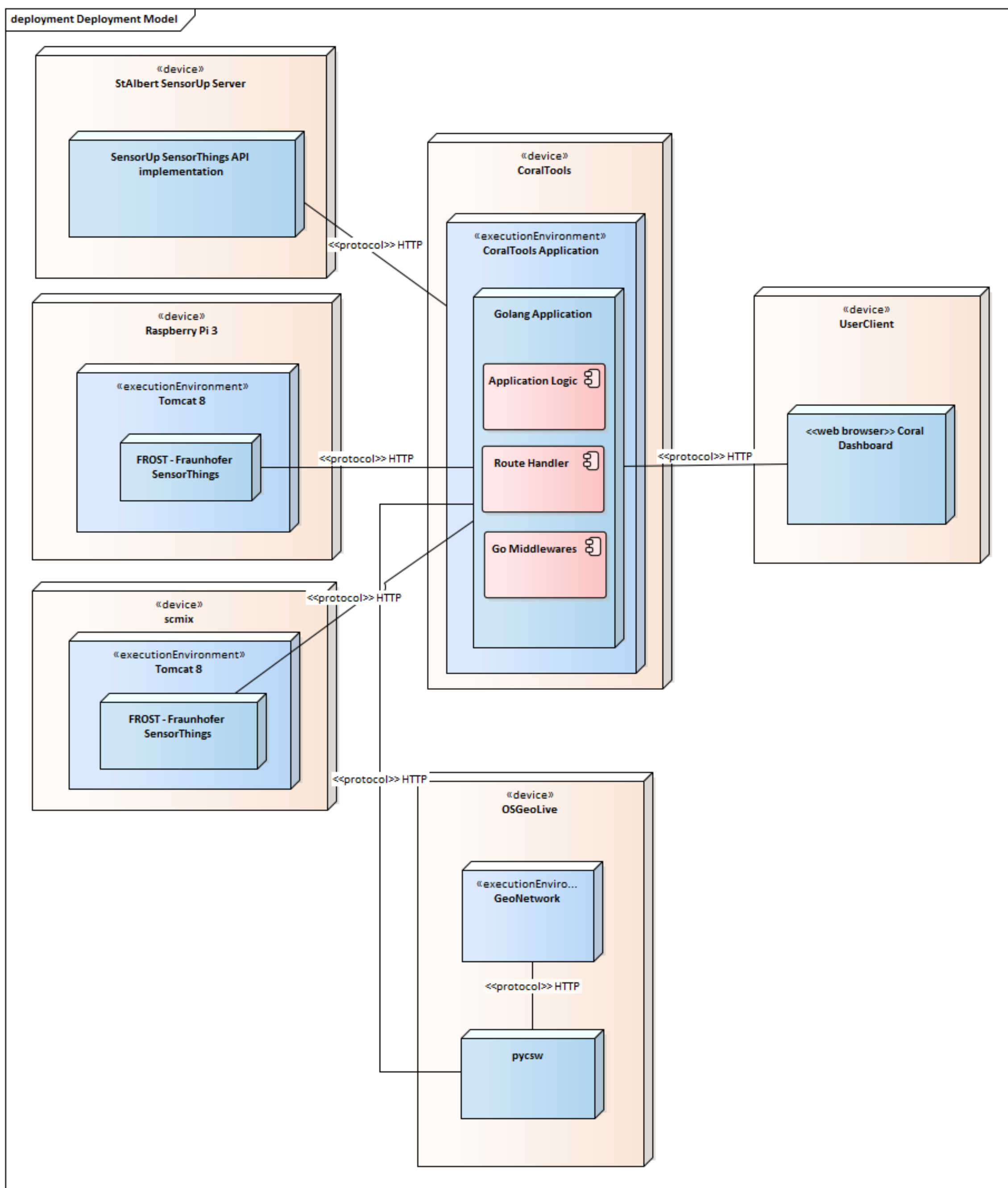


Figure 3.1: Deployment diagram for a Raspberry Pi, the server application and the web application

some message. This message is dynamic and is generated through the websocket mechanism. One example of when a message is generated is after the user adds a new public node, this will issue a notification that will reach all the users that have a open connection at a given time. Furthermore throughout the development of the application the websocket will see more use as the increase of interaction of the user with the web application evolves.

3.3.1.3 Generate GetCapabilities document

This feature is particularly important to interact with the cataloguing service pycsw. Upon receiving a request of harvesting, the backend application is responsible for generating the *GetCapabilities* document, much like other implementations, such as the 52North for the Sensor Observation Service.

3.3.1.4 Proxy requests with authentication token

Some of the SensorThings API endpoints might have an authentication token mechanism. Therefore, to allow for a seemingless interaction, the user when queries the SensorThings endpoint does not need to concern about the availability of the token at the time he issues a request. The proxy request is also relevant for coupling the values received from a SensorThings API entity when not all the entities are shown because of the pagination, therefore as long as there will be an *@iot.NextLink* not null it will go through all the entities.

3.3.1.5 Aggregation of results from different endpoints

Mostly used for the aggregation of the different results from different endpoints, this functionality refers to the web application when it issues a request to visualize the *Datastreams* from different endpoint addresses. Therefore, there is not the need of having the web application to aggregate and parse the results for each one of the visualization tools. By moving this logic to the backend application it will be possible to pre-process the data with any operations that may need to exist, such as calculating the monthly averages of a group of Observations. This also takes into consideration the abstraction of the authentication token for some of the endpoint addresses.

3.3.1.6 Managing the CoralHarvester database

To allow access to the platform and recognize the users it is important to have a database with their respective login details in it. Also, the database stores many other crucial information of the whole system, such as the storage of the endpoint addresses, and their respective authentication token, if it exists. Also, the database is also responsible to hold some of the metadata that is saved after a *Thing* is saved into the CoralTools dashboard catalog.

3.3.2 SensorThings API endpoints

The SensorThings API endpoints are the loosely coupled addresses of the servers that are exposing a SensorThings API implementation. This module of the system is directly accessed by the backend application as described in Section 3.3.1. Each one of the endpoints may constitute a single IoT platform.

```xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Harvest xmlns="http://www.opengis.net/cat/csw/2.0.2" xmlns:xsi="http://www.w3.org
     /2001/XMLSchema-instance" xsi:schemaLocation="http://www.opengis.net/cat/csw
     /2.0.2 http://schemas.opengis.net/csw/2.0.2/CSW-publication.xsd" service="CSW"
     version="2.0.2">
3   <Source>https://seabiotix.inesctec.pt/52n-sos-webapp/service</Source>
4   <ResourceType>http://schemas.opengis.net/sos/2.0/sosGetCapabilities.xsd</
       ResourceType>
5   <ResourceFormat>application/xml</ResourceFormat>
6 </Harvest>
```

Listing 3.1: XML file to harvest an endpoint

3.3.3 Web Application

The Web Application module is the component that is used directly by the end users. It serves as both an interface to manage the SensorThings API endpoints, as to visualize datastreams in diferent ways. The Web Application interacts with the backend application through the RESTful API it exposes and takes advantage of its websocket, mechanism so that the Web Application has a way to receive notifications dynamically.

3.3.4 OSGeoLive

The OSGeoLive is running on a separate virtual machine. The reason for using this distribution was due to it already containing several applications pre-installed and ready to use, such as pyCSW and GeoNetwork that will be covered in the following Sections 3.3.4.1 and 3.3.4.2. This component was crucial in order to test the proposed solution to describe a service-based model such as the *GetCapabilities* request since it already contains existing implementations for the OGC ecosystem.

3.3.4.1 pycsw

pycsw is one of the applications that comes with the OSGeoLive distribution. It is used to harvest a given SensorThings API endpoint address. In this particular case the pycsw is responsible to send a request to the backend application specifying the address of the endpoint it wants to harvest as well as other options passed as parameters on the *GET* request. To do so, pyCSW already contains a few methods that can be called that will allow for the harvesting of an endpoint. To harvest a Sensor Observation Service 2.0 we would have to give the following XML that contains input on the endpoint address, the service type that the endpoint is running. As an example, the Listing 3.1 refers to the harvesting of a SOS2.0 instance on the address defined in the Source tag.

3.3.4.2 GeoNetwork

GeoNetwork is another of the applications that is already installed in the OSGeoLive distribution. After the harvesting is done by the pycsw the GeoNetwork will harvest the pycsw. As pycsw is a

headless metadata harvesting catalogue, it does not provide any interaction with the user. Therefore it is important to take advantage of some of the features that GeoNetwork provides, such as the scheduling of the harvesting tasks and more importantly the search functionalities it has granted by ElasticSearch[2] that comes with it.

3.4 Conclusions

The first step on the development ought to be the development of the Web Application in parallel with the Backend Application as the latter has an important role on managing the users of the Web Application as well as the endpoint addresses, bridging the issuing of the requests from the Web Application to the devices running a SensorThings API server, communicating with the pycsw interface and by producing a *GetCapabilities* document.

These modules can be split into three distinct categories. Chapter 4, refers to the implementation of the functionalities that enable the addition of endpoint addresses to the platform, among other expected actions there is also the management of the devices that expose a SensorThings API server. The Chapter 5 describes the conceptual implementation that aims at bridging the gap between the SensorThings API and other OGC specifications, such as Catalogue Services for the Web implementations namely GeoNetwork by leveraging on inserting JSON-encoded metadata in the description fields belonging to the entities in the STA data model. The last Chapter 6, describes the implementation process of the tools which main function is to produce higher-level outputs for data visualization by aggregating *Observation* data collected from various endpoints.

[2]https://www.elastic.co/

Chapter 4

Device Management

This Chapter will describe thoroughly the implementation details of the Device Management Module. Firstly, it will be described the development of the functionality that refers to the process of adding these endpoints to the web application. The last section refers to the development of the interfacing functionality with the remote SensorThings API endpoints, in order to manage its entities.

4.1 Introduction

The importance of managing the entities without having to know the underlying structure for each request is an advantage for the sake of simplicity and overall usability for all users that interact with any of the platforms. With this purpose in mind one of the core modules of development encompassed the development of an interface built in the web application that is responsible to manage the platforms. This module is composed by the management of the entities such as the *Thing*, *Datastreams* as well as the management of the endpoints within the domain of the whole application.

4.2 Endpoint collection

Through this application, most of the interaction made by the user is dependent on this functionality, as it allows for the inspection of an endpoint that is running a SensorThings API server implementation and to add it to the backend application. The user needs to know the endpoint address of the server running the SensorThing API in order to add it, as there is no mechanism to search for SensorThing API. After entering the endpoint address there will be issued a query through the backend application in which it will retrieve all the data referring to all the entities of the SensorThing API with the exception of the discrete results of the *Observations*, as this aspect is later addressed in the **Dashboard** chapter.

The user also has the option to either specify the visibility of the endpoint address, making it globally visible to all the other users of the application, to make it only visible for him or to a group of users that are inserted in an organization. After submitting the endpoint address it will be named a **node**.

4.2.1 Collecting *Thing* metadata

After adding an endpoint address it is possible to access the data contained on a *Thing* entity in order to store its metadata on the backend application. This is particularly useful because there are no guarantees that the sensor platform will always be available if the user is not the person in charge of such device. With this functionality, it is possible to know the identification of the entity that controls a given endpoint address such as the point of contact of the person in charge. In the context of a decentralized architecture it becomes easier to retrieve relevant information that otherwise would not even be available.

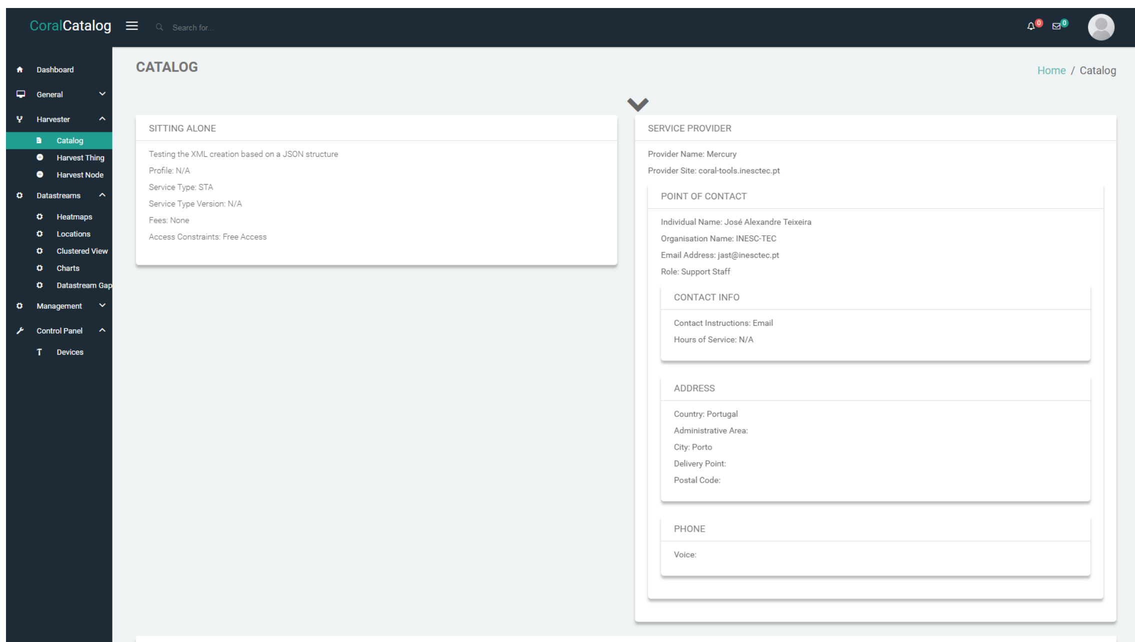
In order to add a new entry referring to a known *Thing* the user must choose the endpoint and the *Thing*. Then the backend application upon receiving the request of the web application will interact with the given node to retrieve the *properties* field. It is in the *properties* field that the metadata is stored. With the response, the web application will fill in the form automatically and allow the user to simply add that to the catalog.

4.2.2 Catalog

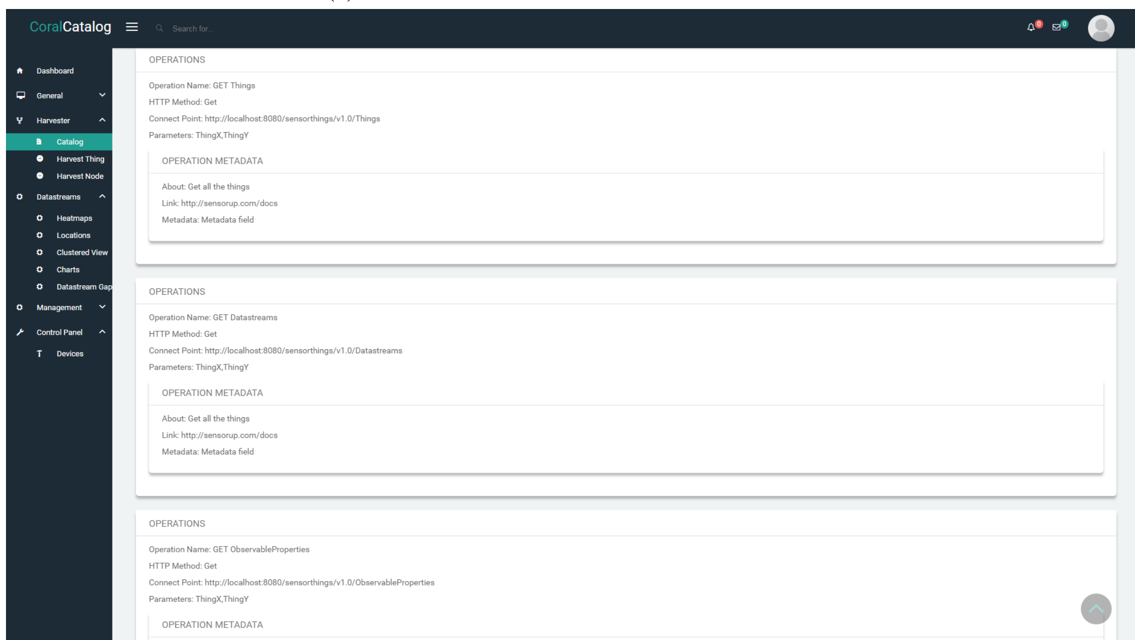
The catalog displays metadata information about the *Things* that were harvested. The information that is presented to the user in the web application is stored in the database. Moreover, this information is mostly tied to the service identification, service provider and operations metadata. Since that in a multi-platform IoT environment we assume that the user may not know every endpoint. Therefore, this information is presented through the web application as depicted in Figures 4.1a and 4.1b.

4.3 Control Panel

Control panel is the name given to the feature responsible for taking advantage of the RESTful API of the SensorThings API to allow for CRUD operations through the web application. However, the control panel is not supposed to interact with any of the *Observation* entities. For the visualization of *Observation* data there is a specifically built module described in a following chapter. The user, after adding an endpoint to his personal area on the application , be it a public or a private endpoint, will then be able to interact with all the six distinct entities that compose the SensorThings API data model (see Section 2.2.1.1. Despite each one of these entities have different fields, the behavior of the application is similar to all the different endpoints as the exposed REST API follows the SensorThings API standard.



(a) Service Provider and Service Identification



(b) Operations Metadata

Figure 4.1: Catalog view on the web application

4.3.1 Management of SensorThing API entities

To create, update and delete the user, is presented firstly with a list of each one of the available entities and must choose one. Afterwards the form is automatically filled with preset values and the user can alter the values on the fields. Finally the user can create a new entity or update the selected one. It is also possible to delete an entity.

4.3.1.1 *Datastream* entity

According to the SensorThings API, the *Datastream* is composed by three other entities, the *Observable Property*, the *Sensor* and the *Thing*. Therefore, when a new *Datastream* is created beforehand there should exist at least one of the aforementioned entities. The *Datastream* entity will also be described by the *Unit Of Measurement* and used to describe the units on the results of the *Observations* for that *Datastream*. The *Unit Of Measurement* is a *JSON_object* composed by the following fields *Name*, *Description* and *Symbol*. Moreover, there are other important fields on the *Datastream* entity that are internally managed by the FROST application, namely the *ObservedArea* which can be described as being a BoundingBox that defines the area on where all the *Observations* are situated from a geographic perspective. Besides the *ObservedArea* there are also the *PhenomenonTime* and *ResultTime* fields that are equally updated with any new *Observation*. They describe the duration of the whole *Datastream* taking the dates of the first and last registered *Observation* of that *Datastream*.

4.3.1.2 Geographical Entites

The entities *Location* and *Feature of Interest* both describe geographic locations. *Location* is associated with the *Thing* and the *Feature of Interest* to the *Observation*. When a new *Observation* is created and there is not a reference to a *Feature of Interest*, a new one will be created based on the *Location*. However, the most important aspect to consider is the GeoJSON object that should be present. By default the web application already assumes that the Geometry Object is a Point. The user simply needs to enter the geographic coordinates that spatially refer to a given Point.

4.3.1.3 *Observable Property* and *Sensor* entities

These two entities describe the *Datastream* entity. However they might be different in terms of semantic value but in terms of application and usability by the user are similar. Both these entities are described by a *Name* a *Description* and a *Definition*. The *Sensor* entity has a field of *Metadata* instead of *Definition* and the user can use this field to provide information related to the sample rate of the sensor.

4.3.1.4 Management of *Thing* entity

Different from all the other entities where the user can simply change the fields, the *Things* entity follows a different logic most importantly on the *Properties* field. This field as stated before is a *JSON_Object*, therefore it is possible to allow for anything the user wants. However to enable for a *Thing* to be harvested by third-party applications, such as the *pycsw* it needs to contain some pre-determined fields. The backend application has a method that will extract the information contained in the aforementioned *properties* field and build a *GetCapabilities* document as it is later described in the following chapter.

4.3.2 Management of Authorization tokens

Some of the endpoints that can be added will be protected by an authentication mechanism to protect the integrity of the data. For that reason, the backend application needs to know which of the endpoints need an authorization token in order to allow for CRUD operations and furthermore retrieve the *Observation* data. Therefore, it was implemented an interface for the user to manage the authorization tokens and associate them with the desired endpoint. This token will be used by the backend application at the time there is any request done and the target endpoint has a token associated to it.

4.4 Conclusions

The functionalities described in this chapter focuses primarily on making the web application more complete by enabling an interaction from the user to the devices, by presenting forms to operate with the SensorThings API. Thus, without having to create the requests by hand it facilitates the process of managing the devices' entities. Additionally, it is also important to store the endpoint addresses in the database so they easily become accessible throughout the application when the user needs, for instance, to retrieve the results of a *Datastream*.

Chapter 5

Metadata Cataloguing Implementation

This chapter refers to the implementation of an operation that is intended to aggregate the varied fields of metadata that are inherent to the device. The goal of this concept is to serve as a proof of concept for the enablement of service metadata description on devices running the SensorThings API specification without having to redesign the proposed specification. Through this chapter the proposed data model will be thoroughly explained and the architecture of the solution.

5.1 Introduction

All services in the OGC common service framework, including the Sensor Observation Service (SOS) [4] have several operations that must be provided by each implementation. One of the core operations is the *GetCapabilities* request that allows to query a service for a description of the service interface and the available sensor data. This is always the first request in a service chain and encompases important information that facilitates machine to machine service discovery, binding and interoperation.

The lack of inclusion of this request introduces a non-linearity in the implementation of service chains. Furthermore, including a GetCapabilities would decrease the level of implicit knowledge that other services in the ecosystem need to have in order to interpret and bind to service end-points. Providing standard metadata descriptions (ISO 19115/ISO 19119) also facilitates the work, for example, to catalogue service harvesters.

While the need for a GetCapabilities is debatable if an API is common, this is often not the case in the scope of a service ecosystem, where different services coexist across a distributed computing environment and need to interoperate in order to deal with different resources towards the same goal. Therefore: 1) the à priori knowledge for a system integrator about each service details can be considerable, rendering service automation nearly impossible and reuse very low. For instance, which web service would be invoked first and what parser should be used for each type of service?; 2) Different implementations may also offer different “optional” features; 3) Additonal

information about complementary services in the service ecosystem could also be included, for example, authentication service URI and available methods.

Even across distributed systems, users need common entry points to find resources, such as services and/or data. This is usually the role played by catalogue services, such as GeoNetwork [20]. Catalogues are mainly used to manage spatially referenced resources, providing powerful metadata harvesting, editing and search functionalities.

For this work, we consider that each user maintains a collection of resources of his own interest. As such, he will have his personalized entry point which contains information about his own devices, but also other IoT devices that are either publicly available, or that he has been granted access to.

In the geo-spatial application domain, GeoNetwork is a reference implementation for the OGC CSW 2.0 ISO profile and currently supports other sources such as the OAI-PMH, OpenSearch and Z39.50.

One of the most interesting features of GeoNetwork is the ability to automatically harvest information of the resources that are exposed by services end-points. The harvester is responsible to automatically communicate with the endpoint running the SensorThings API server implementation and retrieve the metadata inherent to the services available from that device. In a typical OGC CSW interaction, this would simply mean that the harvester would invoke the operation *GetCapabilities* from that service and the response would contain all the required metadata for the resources it contains.

5.2 Overview

The proposed solution takes advantage of the flexibility the data model of SensorThings API provides to store the required metadata. Additionally there are some fields that will need to be generated dynamically during the request. In order to be the most compliant with the OGC specifications, the expected response of the operation *GetCapabilities* should contain the following fields described in Table 5.1.

As stated previously in Section 5.1 it is possible to take advantage of the SensorThings API data model to support the storage of relevant metadata concerning the Service Identifaction field, the Service Provider field and the Operations Metadata field.

5.3 Structuring the metadata model

Firstly, to store the metadata for a single device it is necessary to understand where and how it should be structured and organized within the SensorThings API data model.

After a thorough analysis we came to the conclusion that the best approach would be to add most of the information under the *properties* field in the **Things** table, as seen in Figure 5.1. Because this field is of type *JSON_Object* it means that we can store a string having a specific meaning in the form of a key-value pair.

Table 5.1: GetCapabilities fields [25]

Section name	Meaning
Service Identification	Metadata about this specific server. The contents and organization of this section should be the same for all OWSs.
Service Provider	Metadata about the organization operating this server. The contents and organization of this section should be the same for all OWSs.
Operations Metadata	Metadata about the operations specified by this service and implemented by this server, including the URLs for operation requests. The basic contents and organization of this section shall be the same for all OWSs, but individual services may add elements and/or change the optionality of optional elements.
Contents	Metadata about the data served by this server. The contents and organization of this section are specific to each OWS type, as defined by that Implementation Specification.

Thing
+name: CharacterString
+description: CharacterString
+properties: JSON_Object[0..1]

Figure 5.1: *Thing* entity table

Despite the proposed structure, the *properties* field can contain more customized information depending on the user needs. The application that produces the metadata document is flexible enough to only look for the set of reserved keywords and ignoring the rest of the *JSON* key-value pairs.

To help index the different *datasets* and *services* in a regular harvester, there is also a field of keywords that should contain information that best describe the servers. These keywords ideally should belong to a controlled vocabulary.

The Table 5.2 describes the main fields that compose the *JSON_Object*.

5.4 Generating the Contents field

Due to the nature of the devices, they need to be able to constantly change the information about the *datastreams* and not be encumbered by updating the *properties* field continuously after a small change. Therefore the *Contents* field is generated at the moment of the request. To meet this requirement the application that generates the GetCapabilities document will have to retrieve all the information associated with the *Datastreams* under the targeted *Thing*. According to the OWS

Table 5.2: JSON structure

Key name	Meaning
Version	The Version refers to the Version of the specification chosen, it is defaulted to 2.0.0
Service Type Version	Version of the service type implemented by the server.
Profile	Identifier of OGC Web Service (OWS) Application Profile.
Title	Title of the server, usually refers to the name of the device.
Abstract	Brief description of the server, usually is a summary of the device.
Keywords	Group of one or more words belonging to a controlled set of words that are used to describe the device.
Fees	Fees and terms for using the server.
Access Constraints	Information of any possible restriction from using data from the server.
Provider Name	Unique identifier that refers to the organization that is providing the service.
Provider Site	Reference to the most relevant web site of the entity that provides the service.
Service Contact	Information for contacting the service provider. Usually contains an address, phone, electronic mail address, the name of the individual and other relevant informations.
Contains Operations	Information about the operations provided by the service and implemented by the server, containing the HTTP verb and the required URL for the operation request. It also contains information about additional fields.

specification the *Contents* field should always have a minimum of necessary parts, defined in Table 5.3.

To achieve this, the application has the target SensorThings API endpoint to retrieve the *Datastreams* entity and additionally, their associated *Observed Property* and *Sensor* entities, to provide additional information that will further integrate the *Contents* content. The expected output is an array with each element of the array referring to a distinct *Datastream*. The *result time* and *phenomenon time* values are internally managed by the SensorThings API and they are updated according to the oldest and newest *Observation* that are present in the database. Moreover, the metadata model for the *Contents* field according to the specification can be extended as needed. Thus, as most of the Observable Properties are physical quantities an extra field was added named *Unit of Measurement*. This field is important to provide a semantic meaning to the *result* value of an *Observation*, as for Measurements it is necessary to provide the unit of the property that is being subject to measurement. Finally, the *Contents* field also takes into account the spatial location of the whole dataset. All of the observations are confined to a physical space, therefore the application should also be able to provide the values of the coordinates that define a bounding box where all

Table 5.3: Minimum parts of a single *Datastream* in the Contents field [25]

Name	Meaning
Title	Title of the dataset.
Abstract	Brief description of the dataset.
Keywords	Group of one or more words belonging to a controlled set of words that are used to describe the dataset.
Identifier	Unique identifier of the dataset
WGS84 Bounding Box	Minimum bounding rectangle surrounding the dataset, using WGS 84 CRS with decimal degrees and longitude before latitude
Bounding Box	Minimum bounding rectangle surrounding dataset, in available CRS

the observations where taken.

5.5 Architecture

To allow for cataloging of the several devices it is necessary to understand the underlying architecture of the system under development. The main principle is to have a decentralization of the data generated from sensor activity; meaning that the data collected from the various sensors is not pushed to a single database. However, all of the sensors that compose the whole network must be cataloged in a single database.

Since the SensorThings API specification does not specify a *GetCapabilities* request it is necessary to implement a web service following a Resource-Oriented Architecture that bridges the interaction with the target device to retrieve its stored metadata and to generate the *GetCapabilities* request. To meet this end we developed an application in Golang (Go) whose main function is to generate the metadata document as stated before. The architecture is depicted in Figure 5.2.

To accomplish this, the application will take advantage of the exposed REST API to query for the resources contained in the device that is running the SensorThings API. After obtaining information regarding the *Thing* entity, it will look for the *properties* field and extract the data contained there. As stated previously, the application will search for the known keywords that have a specific meaning and aggregate them to compose the structure of the document. Once all the requested fields are filled in the application will respond back to the GET request with the metadata document.

The architecture is depicted in Figure 5.2.

5.6 Conclusions

The need for a *GetCapabilities* request is debatable if the API is common, but the à priori knowledge for a system integrator about each service details can be considerable, for instance, which web

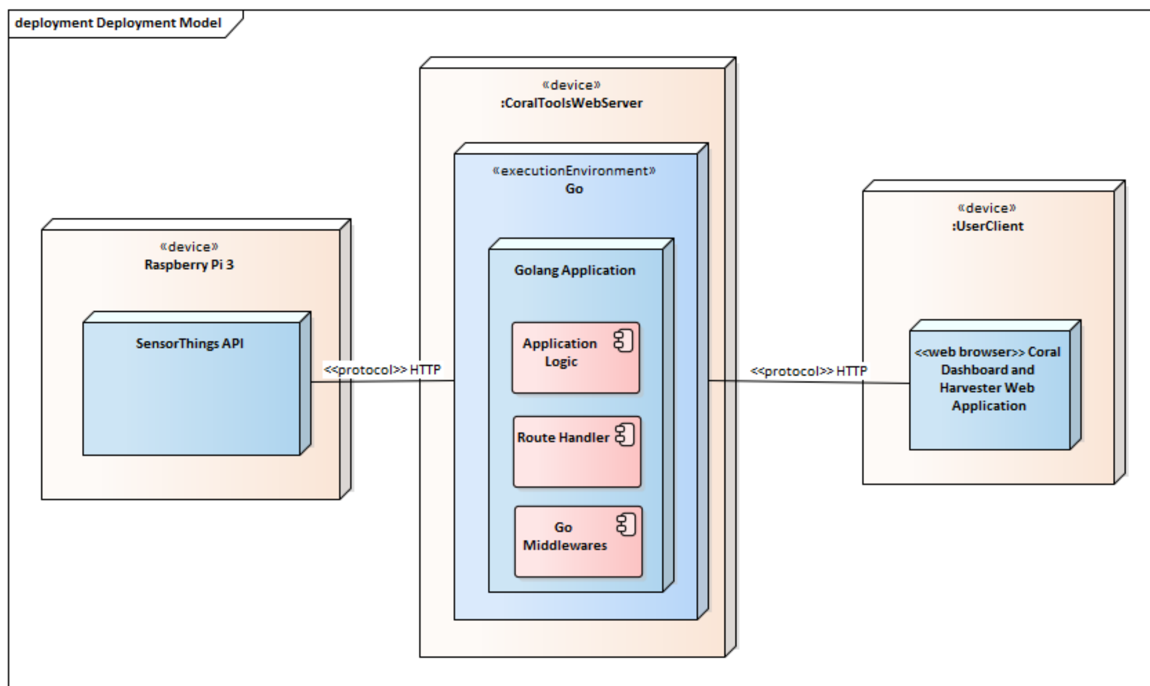


Figure 5.2: Application Deployment Diagram

service shall be invoked first. Different implementations may offer different "optional" features. Additonal information about the service ecosystem can be included, for example, authentication. In order to create a system where seamlessly all the devices seem to be connected, there is the need to build a centralized structure that supports and contains information about all the other IoT devices. Ideally this application is supposed to work as a bridge between the devices that expose the SensorThings API and the end users. For a more correct approach on the solution for this problem, the application needs to have some functionalities that are usually applied in Catalog Services, such as GeoNetwork [20].The catalog is mainly used to manage spatially referenced resources, providing powerful metadata editing and search functions. Thus meeting the requirement of having such centralized structure responsible for organizing all their inherent devices.

By combining the flexibility of the data model of the SensorThings API and an efficient resource-oriented architecture the final result demonstrates that it is possible to reach the same outcome as other metadata retrieval methods on for instance, a Sensor Observation Service (SOS). And since the application that outputs the *GetCapabilities* document is independent from the SensorThings API server-side implementation it has a higher level of flexibility only relying on the conformance of the device that runs the SensorThings API to follow the specified data model.

In addition, by integrating this proposed solution with the OGC SensorThings API standard it is possible to achieve a more uniform and standardized way to manage the devices' metadata. This will have a positive impact in several IoT-based applications that rely on having multiple clusters of data where it is possible for the users that interact with these platforms to retrieve a more detailed report of what a specific device can provide, the organization that operates it and the operations that it provides as well as the datasets that are exposed and able to be queried.

Finally, to validate the interoperability of our solution we tested it using other third-party applications such as pycsw and GeoNetwork [35]. By successfully harvesting the metadata contained in the SensorThing API as we proposed and being able to display it in the GeoNetwork application we could verify that our solution was interoperable and therefore a possible solution for metadata representation of the SensorThings API.

Chapter 6

Dashboard Implementation

This chapter describes the development of the web application that concerns about the visualization of the observation data. With the multitude of ways to present the observation data for visualization, the most important aspect of the development focuses mainly in presenting the data in different ways and from more than one source.

6.1 Datastream selection

The interface for the users to select the datastreams is shown in Figure 6.1. Common to all the visualization tools it is important for the user to be able to retrieve the most updated information related with the datastreams therefore during the whole process of selecting the desired *datastream* firstly it is needed to perform a request to the endpoint address to retrieve the available *Thing* entities. Then it is possible to expand and retrieve the associated *Datastream* to each *Thing* available. Furthermore, the selected *Datastreams* are displayed in the form of a table separated by each endpoint as shown in Figure 6.2.

For each user, there are some rules of the endpoints that are able to be selected. The user can only select the endpoints that are directly associated to him and this is managed by the backend application.

6.2 Heatmaps

The heatmaps visualization that was implemented refers specifically to a 2D visualization with colors that represent a third variable. This third variable that is being displayed in the map is a function of other 2 variables, the latitude and the longitude, that refer to the *Feature of Interest* of its corresponding *Observation*. This functionality relies on the user to select several *Datastreams* and the application is responsible to represent the observation data in a map. Furthermore, as the observation data represents a timeseries it was also taken into consideration to create an animation that spans from the start date until the last recorded date of a single *Observation*. This was accomplished by redrawing the points representing the data at every cycle. To perform a request

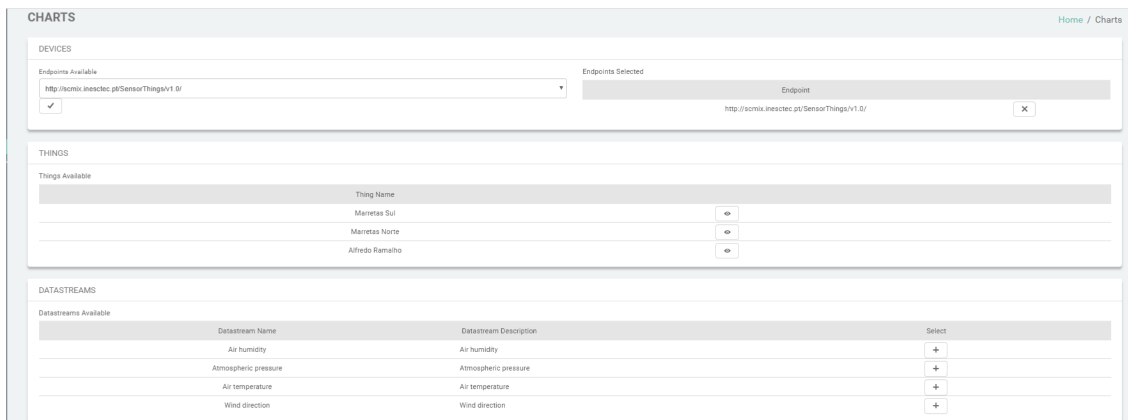


Figure 6.1: Interface for selecting datastreams

the user follows the same steps to select the datastreams as described previously in Subsection 6.1 and then issue the request. The following request depicted in Listing 6.1 will produce the following output shown in Figure 6.3.

The backend application is responsible to query the different endpoints and retrieve all the *Observations* that the selected *Datastreams* contain. After retrieving all the *Observations*, the backend application aggregates all of them and responds back to the web application with all the needed information to properly display it. The web application, after receiving the response, will determine the first and the last date an *Observation* was recorded in order to limit the time frame. An example of the response is shown in Listing 6.2.

In order to determine the value of intensity, it is necessary to normalize the values. Due to the nature of the measurements it is hard to predict a reasonable scale for the values to be represented without a pre-determined ontology that can recognize a given *Observable Property* to set a scale for the result values of the *Observations*. Therefore, the scale is based on a relative analysis of all the results retrieved. It is important to know the maximum and minimum values and then set the value of each intensity according to the following expression:

$$Intensity = \frac{result - min}{max - min} \tag{6.1}$$

```
1 {
2   "connectEndpoints": [
3     {
4       "address": "http://localhost:8080/sensorthings/v1.0/",
5       "datastreamID": [12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28]
6     },
7   ]
8 }
```

Listing 6.1: Heatmap request

```json
{
    "data": [
        {
            "endpointaddress": "http://localhost:8080/sensorthings/v1.0/",
            "datastreamid": 12,
            "observations": [
                {
                    "int": 20984,
                    "resultTime": "2018-06-20T07:10:00.000Z",
                    "phenomenonTime": "2018-06-20T07:10:00.000Z",
                    "result": 9.142027989982468,
                    "latitude": 41.1671572,
                    "longitude": -8.6087762,
                    "name": "INESCTEC",
                    "description": "INESCTEC"
                },
                {
                    "int": 21778,
                    "resultTime": "2018-06-20T07:20:00.000Z",
                    "phenomenonTime": "2018-06-20T07:20:00.000Z",
                    "result": 17.24926971381685,
                    "latitude": 41.1671572,
                    "longitude": -8.6087762,
                    "name": "INESCTEC",
                    "description": "INESCTEC"
                },
                ...
                ]
        }
    ]
}
```


Listing 6.2: Heatmap Response

DATASTREAMS

Datastreams Selected

http://scmix.inesctec.pt/SensorThings/v1.0/

Datastream ID	Start Date	Start Hour	End Date	End Hour
31	01/01/0001	00:00	01/06/2018	15:21:05

https://stalbert-aq-sta.sensorup.com/v1.0/

Datastream ID	Start Date	Start Hour	End Date	End Hour
129	01/01/0001	00:00	01/06/2018	16:06:03
130	01/01/0001	00:00	01/06/2018	16:06:04

Figure 6.2: Interface for the selected datastreams

The value of the above Equation 6.1 is a value always set between 0 and 1. This value is important to determine the predominant color that will represent the given point on the map, whereas an higher numerical value will represent a color closer to red.

6.3 Clustered View

Related with *Datastream* visualization, all of the *Observations* are geographically represented by its *Feature of Interest*, as the *Feature of Interest* is spatially located, the user can also be presented with the *Observation* data in a map. Due to the nature of some *Datastreams* that can hold thousands of records it is important to optimize the representation in the map and aggregate them in clusters rather than inserting a marker in the map. The process of issuing a request encompasses the selection of the wanted *Datastreams*. The backend application then receives the request and will query for the *Observations* over all the chosen endpoints. As the SensorThings API has no direct way to aggregate the Observations of a given *Datastream* by its corresponding *Feature of Interest*, the handler for that request in the backend application will group all *Observations* in a key-value pair whereas the key is the unique ID for the *Feature of Interest* (the *@iot.id*) and the value is an array with all the *Observations* with the same *Feature of Interest*. When the backend application is done with the request the web application is then responsible with the response to represent in the map the result. The output given is as seen in Figure 6.4. To further inspect the values of the given *Feature Of Interest* each one of the points represented in the map need to be clicked. This will expand a table with the values as seen in Figure 6.5.

This functionality supports the query for different endpoints and represent them seamlessly. The Figure 6.6 is the result of the request that contains two different endpoints, represented by the Listing 6.3.

```json
{
  "connectEndpoints": [
    {
      "address": "https://toronto-bike-snapshot.sensorup.com/v1.0/",
      "datastreamID": [1569,97,17,9]
    },
    {
      "address": "http://localhost:8080/sensorthings/v1.0/",
      "datastreamID": [12,13,14,15,16,17,7,8]
    }
  ]
}
```

Listing 6.3: Clustered View Request

6.4 Historical Locations

Due to the nature of some sensors and their platforms, they can have their positions to be altered during their course of action. Taking the example of a buoy whose position can be altered almost in every new *Observation* or maybe even the reallocation of a device with sensors, it is important to constantly update the coordinates of its respective *Location*. Due to this, the SensorThings API already supports a way to represent the successive *Locations* over the time the sensors were in activity. Whenever a new *Location* entity is created the old *Location* will be addressed as a *Historical Location*. The *Historical Location* contains crucial information, namely the time span and the geographical coordinates of the *Location*. To support the visualization, one of the tools that were implemented to address this issue was the *Historical Locations* map. This feature allows the user to visualize all the locations the *Thing* had over the course of time.

The web application will request the backend application with the example payload in the Listing 6.4 the expected response for this request will be an array of objects that contain the endpoint address of the target SensorThings API server, the unique ID of the Thing and an Array of objects that contain the time the Location was first inserted, descriptive information of the Location such as its name, and most importantly the geographical location as depicted in Listing 6.6. With this response the web application will determine the time span of the map animation based on the latest and most recent dates and represent these locations on the map. For the cases where a *Thing* contains more than one Historical Location a line will be drawn that describes the route over time. Moreover, the marker that represents the active *Location* for a given time will be drawn as red, whereas all the other *Locations* are represented by green markers, such as it is depicted in Figure 6.7.

Equally to the other tools it is also possible the user to obtain in the same map the historical locations of more than one *Thing* and through various endpoints, to achieve this the user needs to select other endpoints and select the *Things*. The request is described in Listing 6.5 and produces the map according to Figure 6.8.

```
{
  "connectEndpoints": [
    {
      "address": "http://localhost:8080/sensorthings/v1.0/",
      "thingid": [2,3]
    }
  ]
}
```

Listing 6.4: Historical Location Request

```
{
  "connectEndpoints": [
    {
      "address": "https://toronto-bike-snapshot.sensorup.com/v1.0/",
      "thingid": [861,853,845,837,5]
    },
    {
      "address": "https://stalbert-aq-sta.sensorup.com/v1.0/",
      "thingid": [144,138,132]
    },
    {
      "address": "http://localhost:8080/sensorthings/v1.0/",
      "thingid": [2,3]
    }
  ]
}
```

Listing 6.5: Historical Location Request for multiple endpoints

```
1  {
2     "data":{
3        "connectEndpoints":[
4           {
5              "endpointAddress":"http://localhost:8080/sensorthings/v1.0/",
6              "thingid":2,
7              "locations":[
8                 {
9                    "time":"2018-06-04T07:01:05.926Z",
10                   "name":"Aliados",
11                   "description":"Aliados",
12                   "encodingType":"application/vnd.geo+json",
13                   "location":{
14                      "type":"Point",
15                      "coordinates":[
16                         -8.609993,
17                         41.1561414
18                      ]
19                   },
20                   "@iot.id":1
21                },
22                {
23                   "time":"2018-06-04T07:32:33.092Z",
24                   "name":"Lapa",
25                   "description":"Lapa",
26                   "encodingType":"application/vnd.geo+json",
27                   "location":{
28                      "type":"Point",
29                      "coordinates":[
30                         -8.6421185,
31                         41.1579809
32                      ]
33                   },
34                   "@iot.id":2
35                }
36                ...
37             ]
38          },
39       ]
40    },
41    "status":"Success",
42    "code":"0000"
43 }
```

Listing 6.6: Historical Location Request

6.5 Datastreams presented in charts

Another way of viewing the Observation data for each *Datastream* is by using line charts for date-based data. As every *Observation* has a specific date it is possible to create a timeseries. To perform the query on the different endpoints, the user needs to select the desired endpoints and select the *Datastreams*. It is also possible to define the time period of the *Observations* that the user wants to retrieve as depicted in Figure 6.9. Afterwards the request is issued to the backend application, containing the Endpoint Addresses on which the queries should be issued and also the unique identifiers of the *Datastreams*, as well as the start and end dates. An example request for two distinct endpoints, each containing three *Datastreams* is depicted in Listing 6.7. According to the SensorThings API implementation it is important to note that a single request may not contain all the *Observations* that fit into the timespan that was provided in the query, this is due to the pagination inherent to each SensorThings endpoint. This means that the backend application must verify if there is a field *@iot.NextLink* in each response of the endpoints to a request issued by the server such as: https://stalbert-aq-sta.sensorup.com/v1.0/Datastreams(9)/Observations?$filter=phenomenonTime ge 2018-05-30T00:00:00.000Z and phenomenonTime le 2018-06-04T15:22:33.000Z&$skip=700.

Different SensorThings endpoints may contain *Datastreams* that can hold *Observations* by the order of dozens of thousand. Due to this, it was important to allow the user to retrieve only a specific time interval in order to limit its need and shorten the query times. Moreover the backend application also has the functionality of grouping the *Observations* per month and calculate the monthly average of the results and find the maximum and minimum values, this will allow for the creation of a second chart which will display these values.

An example of the charts that are produced are depicted in Figures 6.10a and 6.10b.

```
1  {
2    "connectEndpoints": [
3      {
4        "address": "http://localhost:8080/sensorthings/v1.0/",
5        "datastreamID": [
6          {
7            "id": 5,
8            "endTime": "2015-08-25T12:15:38.000Z",
9            "startTime": "2015-08-23T13:54:33.000Z",
10           "name": "Air Temperature"
11         },
12         ...
13         {
14           "id": 7,
15           "endTime": "2015-08-25T12:20:40.000Z",
16           "startTime": "2015-08-23T13:54:33.000Z",
17           "name": "Wind Direction"
18         }
19       ]
20     },
```

```
21     {
22       "address": "https://stalbert-aq-sta.sensorup.com/v1.0/",
23       "datastreamID": [
24         {
25           "id": 23,
26           "endTime": "2018-06-04T15:20:04.000Z",
27           "startTime": "2018-05-30T00:00:00.000Z",
28           "name": "AirQ:24EB52 :Humidity"
29         },
30         ...
31         {
32           "id": 21,
33           "endTime": "2018-06-04T15:20:05.000Z",
34           "startTime": "2018-05-30T00:00:00.000Z",
35           "name": "AirQ:24EB52 :PM2.5"
36         }
37       ]
38     }
39   ]
40 }
```

Listing 6.7: Request for Charts

6.6 Datastream availability

The last tool developed intended to primarily give the user the understanding on how a specific *Datastream* performed along the time. Assuming that there inevitably may be some downtimes, there is the need to know when they happened and how long they affected the whole *Datastream*. It can also be a solid indicator on how the quality of the *Observation* data can be trustful. To interact with this feature the user needs to select all the *Datastreams* he wants to visualize equally to all the other tooling mechanisms. After selecting the desired *Datastreams*, the user can either set the sample rate on which it will be used later on to determine the interval time between successive *Observations* or issue the request with the sample rate value that is provided in the field **Metadata** under the **Sensor** table that is associated to the *Datastream*. Afterwards a GET HTTP request is issued with the following: `http://localhost:8081/datastream/observation?endpointaddress=https://stalbert-aq-sta.sensorup.com/v1.0/&datastreamid=16&samplerate=360`. The GET parameters are the Endpoint Address, the Datastream ID and the sample rate in seconds. After receiving this request the backend application will search for all the *Observation* records for the selected *Datastream*. However, there is also the issue of a *Datastream* containing a great amount of *Observations* in which it will lead to very long response times for the requests. So, to mitigate the problem there is the option to specify a time span thus allowing for the user to retrieve the results during that period. The criteria the backend application uses to determine whether there occurs a gap in observation is by calculating

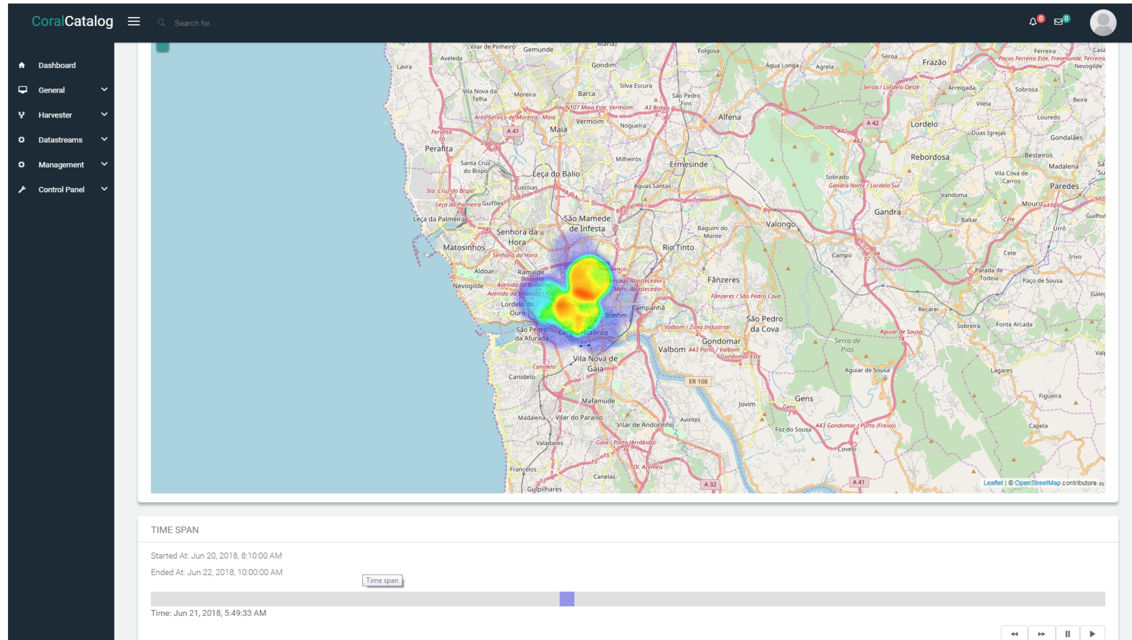
the difference between two *Observations* and the difference should be approximately equal to the sample rate provided. The array of *Observations* that is iterated through has previously been ordered by the **Phenomenon Time** so there is no problem in assuming that the comparison that is being made indeed happens between adjacent *Observations*. If the difference between the two *Observations* is not equal to the provided sample rate then there we assume the existence of a gap. The gap is composed by a start and end date. The start date is the date of the first *Observation* whose difference is not equal to the sample rate and will extend until an *Observation* that correctly does fit into the sample rate. Finally, the web application will display in the form of a Gantt Chart the segments that will indicate the **Gaps** in the *Datastreams*, signaling the period of time on which the sample rate was correct as green and the incorrect as orange as shown if Figure 6.11.

6.7 Conclusions

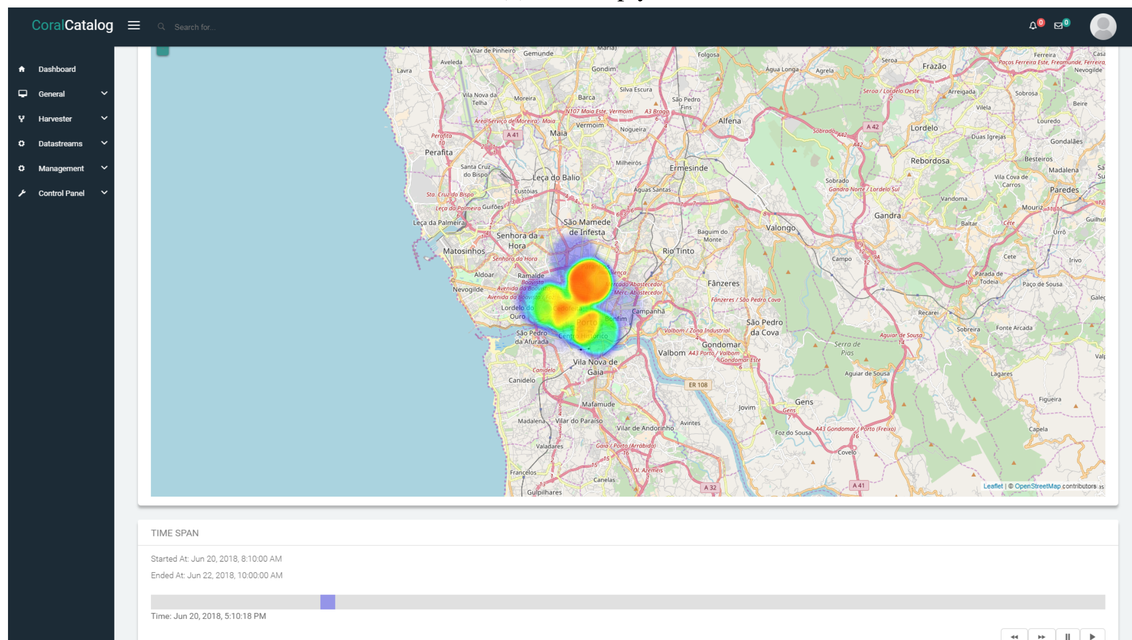
The tools described in this chapter contribute to this work by providing better ways to present the data contained in the devices. By the form of maps, charts and tables there are many ways to display the information based on the user needs. Thus, this set of tools will give higher-level outputs for visualization.

Another key aspect is to test the interoperability of different endpoint addresses. The behavior of each one of the tools is the same regardless of the device that is accessed. This is accomplished since all these devices are running a SensorThings API server.

Moreover, the Go application that aggregates the responses from the various endpoints is headless. This means that other applications may perform the same requests in order to get the data and then, present it in different ways than the ones presented in the web application.



(a) Heatmap y



(b) Heatmap x

Figure 6.3: Heatmap

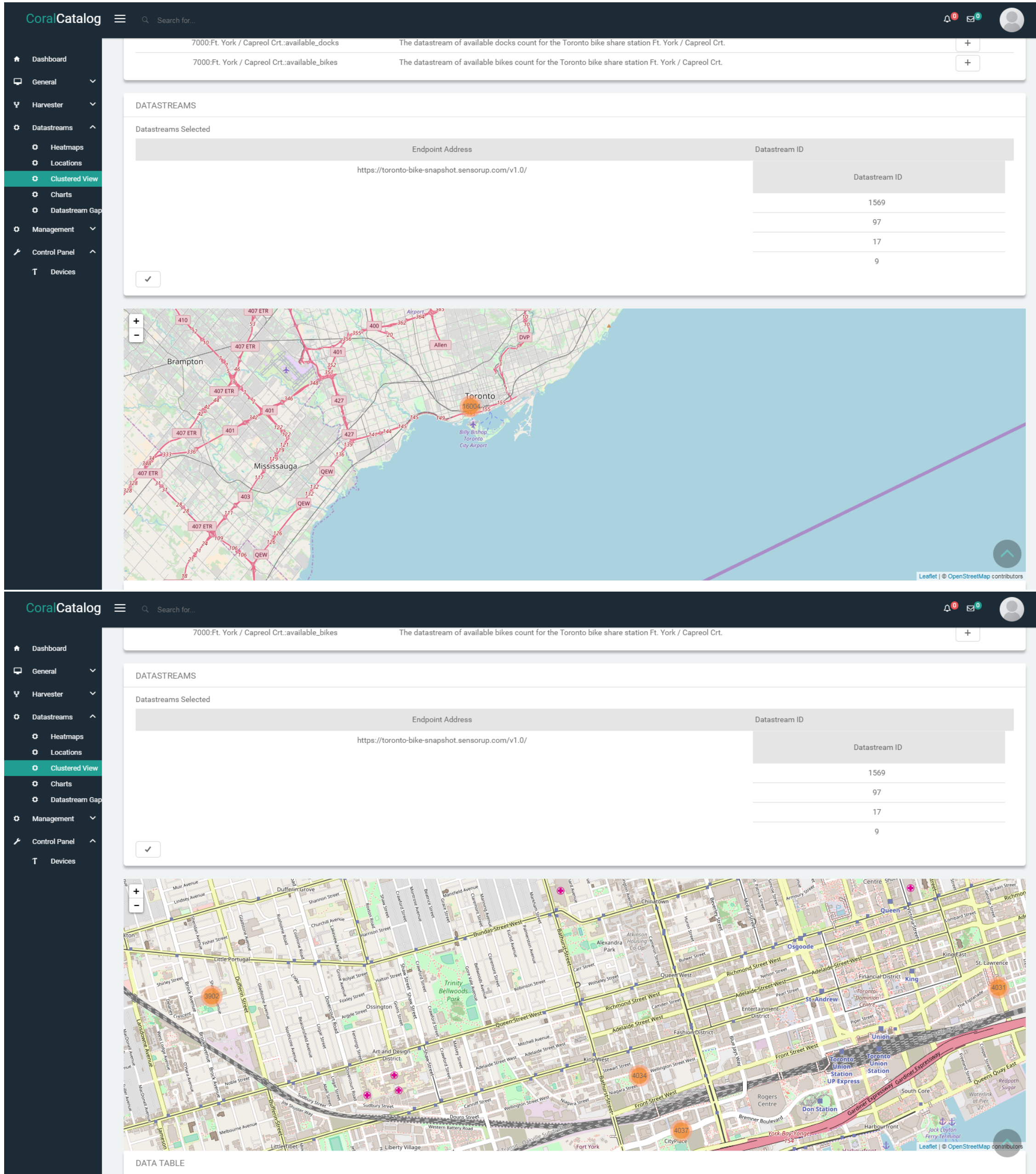


Figure 6.4: Map representation of the clustered View for 4 different datastreams

CoralCatalog

DATA TABLE

Index	Feature of Interest Name	Latitude	Longitude	Phenomenon Time	Result Time	Result
1701	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 7:35:08 PM		3
1702	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 7:40:08 PM		3
1703	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 7:45:09 PM		3
1704	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 7:50:07 PM		4
1705	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 7:55:08 PM		4
1706	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:00:08 PM		5
1707	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:05:08 PM		5
1708	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:10:08 PM		5
1709	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:15:08 PM		5
1710	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:20:08 PM		5
1711	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:25:08 PM		4
1712	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:30:08 PM		4
1713	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:35:08 PM		4
1714	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:40:08 PM		4
1715	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:45:08 PM		4
1716	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:50:07 PM		4
1717	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 8:55:08 PM		3
1718	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 9:00:08 PM		3
1719	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 9:05:08 PM		3
1720	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 9:10:08 PM		3
1721	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 9:15:08 PM		3
1722	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 9:20:08 PM		3
1723	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 9:25:08 PM		3
1724	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 9:30:08 PM		3
1725	7011:Wellington St / Portland St	43.642982	-79.399256	Feb 5, 2017, 9:35:09 PM		3

« Previous 1 ... 34 35 36 ... 162 Next »

Figure 6.5: Table containing the information of the result values for a point represented in the map

CoralCatalog

DATA TABLE

Index	Feature of Interest Name	Latitude	Longitude	Phenomenon Time	Result Time	Result
551	UofC CCIT	53.227481842	-9.261294365	Aug 14, 2015, 1:13:11 PM	Aug 14, 2015, 1:13:11 PM	264.7
552	UofC CCIT	53.227481842	-9.26129818	Aug 14, 2015, 1:18:13 PM	Aug 14, 2015, 1:18:13 PM	278.5
553	UofC CCIT	53.227481842	-9.26129818	Aug 14, 2015, 1:18:13 PM	Aug 14, 2015, 1:18:13 PM	273.7
554	UofC CCIT	53.227481842	-9.261294365	Aug 14, 2015, 1:23:13 PM	Aug 14, 2015, 1:23:13 PM	253.5
555	UofC CCIT	53.227481842	-9.261294365	Aug 14, 2015, 1:23:13 PM	Aug 14, 2015, 1:23:13 PM	248.7
556	UofC CCIT	53.227489471	-9.261291504	Aug 14, 2015, 1:28:13 PM	Aug 14, 2015, 1:28:13 PM	258.5
557	UofC CCIT	53.227489471	-9.261291504	Aug 14, 2015, 1:28:13 PM	Aug 14, 2015, 1:28:13 PM	253.7
558	UofC CCIT	53.227481842	-9.261293411	Aug 14, 2015, 1:33:13 PM	Aug 14, 2015, 1:33:13 PM	292.9
559	UofC CCIT	53.227481842	-9.261293411	Aug 14, 2015, 1:33:13 PM	Aug 14, 2015, 1:33:13 PM	288.1

Figure 6.6: Table containing the information of the result values for a point represented in the map

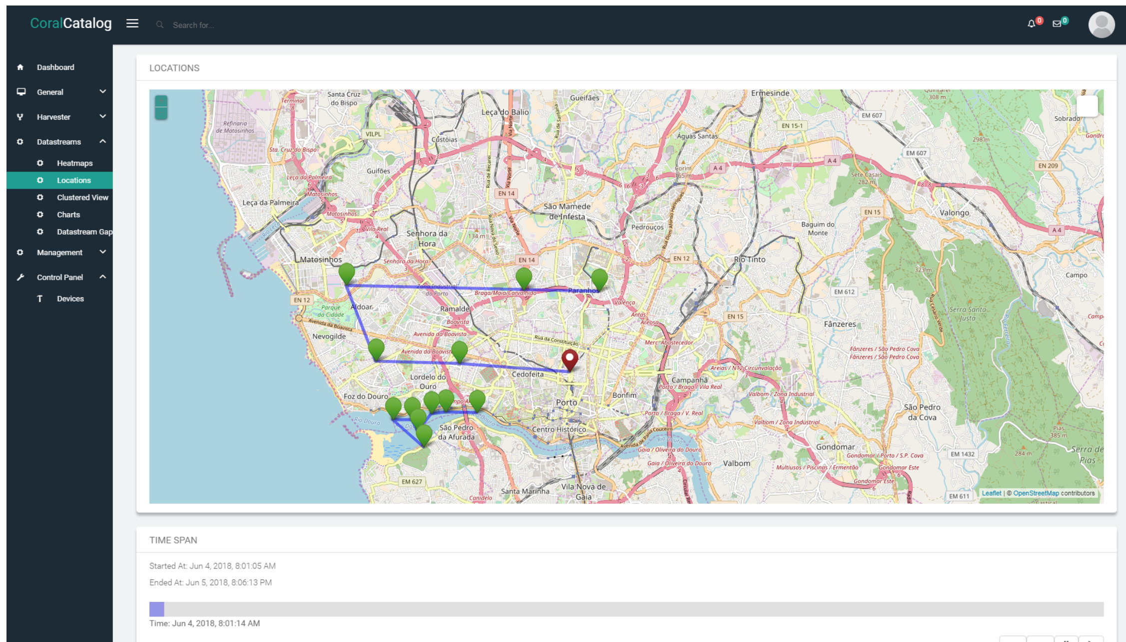


Figure 6.7: Representation of the Historical Locations of two distinct *Things*

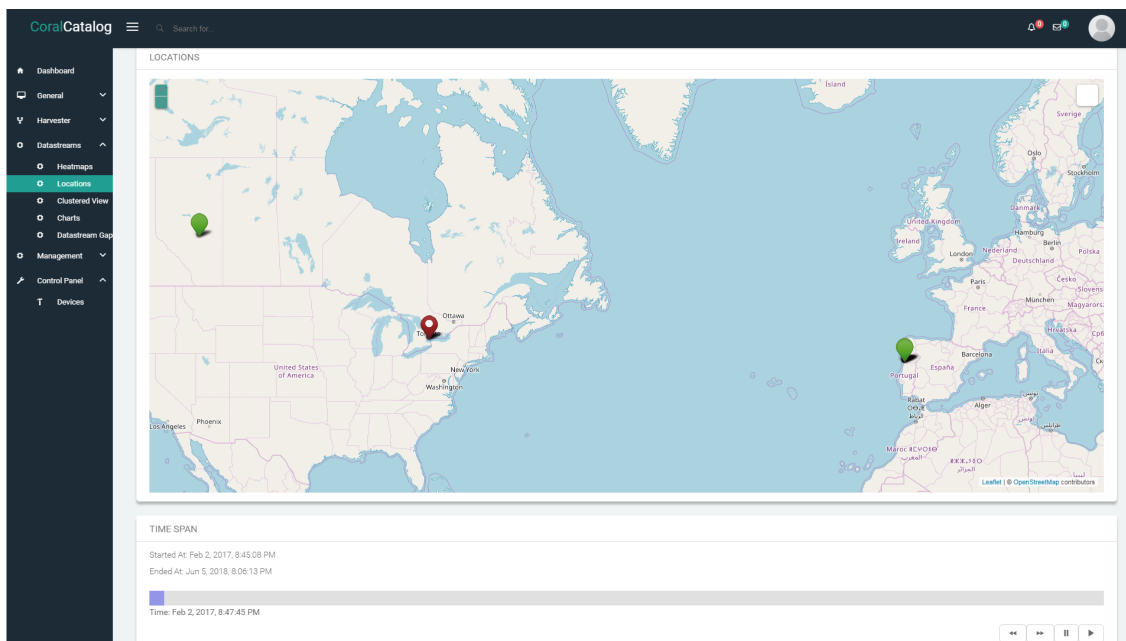


Figure 6.8: Representation of the Historical Locations of three Endpoints and multiple *Things*

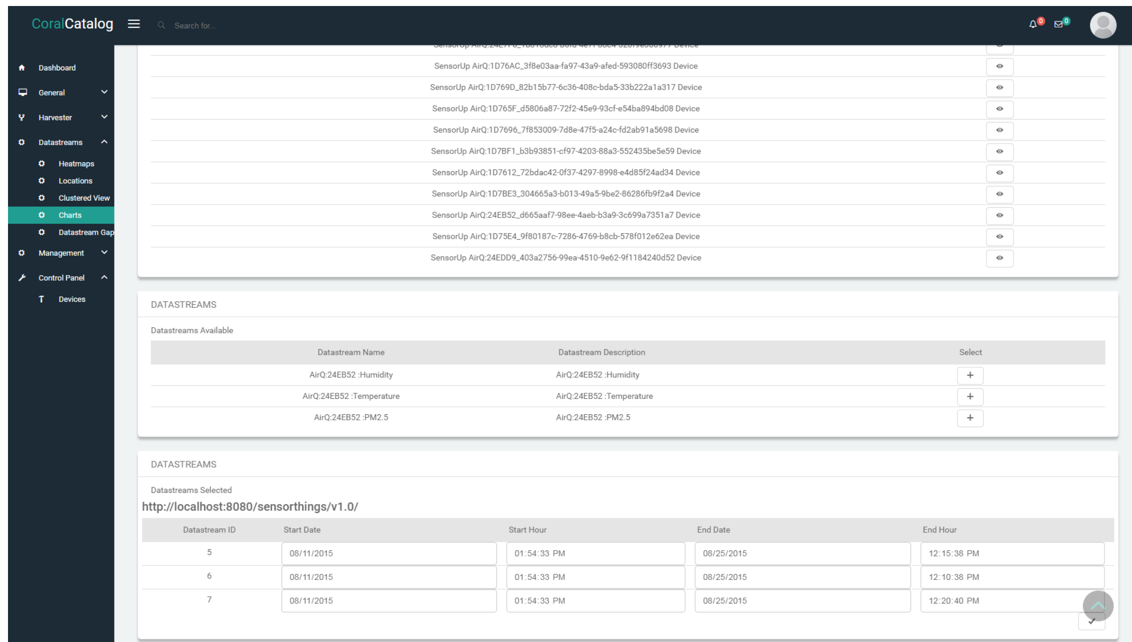
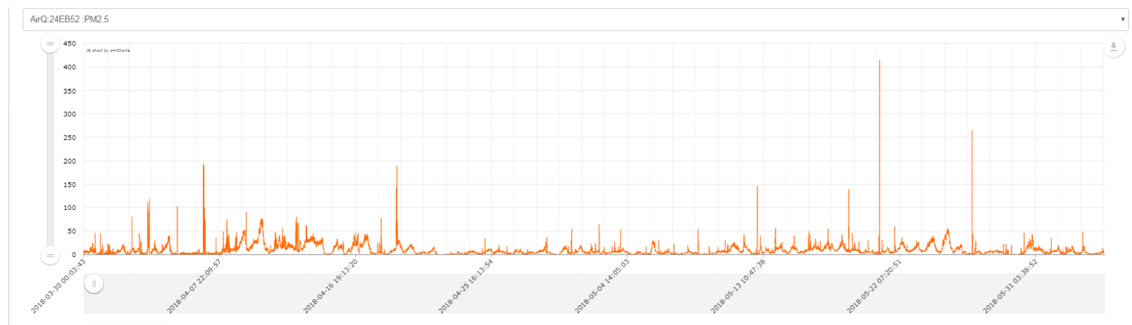
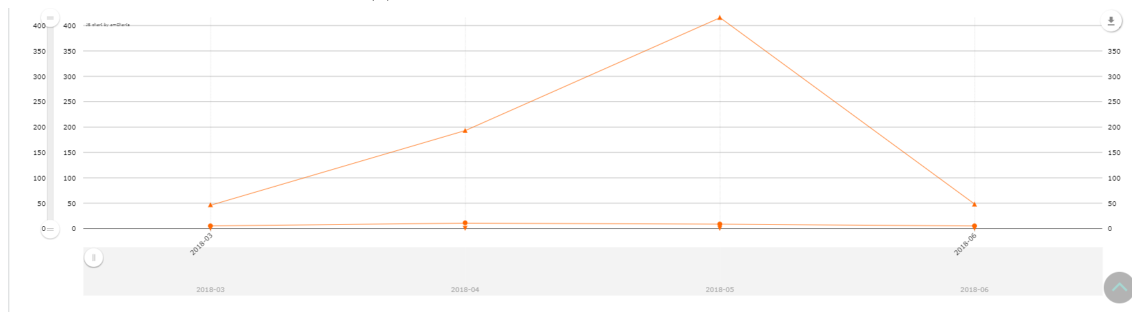


Figure 6.9: Interface for the selection of *Datastreams* and time span



(a) Observation Values of a Datastream



(b) Minimum, Maximum and Average values of the Observation Results grouped by month

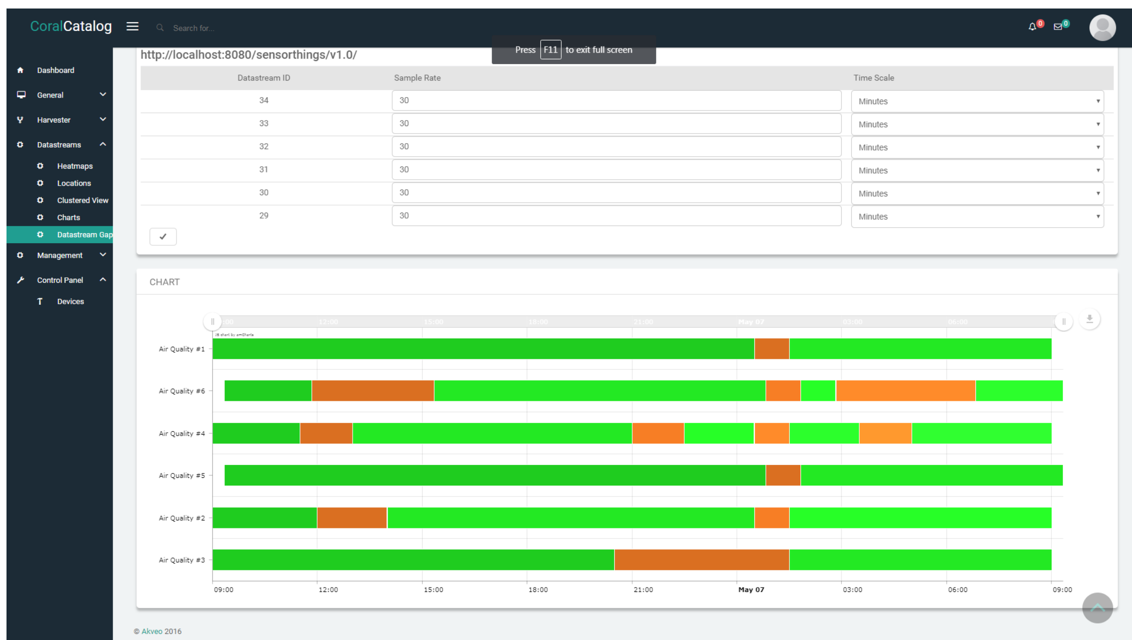


Figure 6.11: Interface for the selection of *Datastreams* and their corresponding gaps

Chapter 7

Evaluation

This chapter, describes the analysis of the solution with the objective to pinpoint the fulfillment of the initially established goals.

7.1 Management of the SensorThings API entities

To test whether the web application can perform the required operations related with the management of entities, some entities were created, deleted and updated. If the web application is working properly then the values in the database ought to be changed in conformance with the actions performed by the user.

7.1.1 Creating a new *Thing* entity

The best example would be to create a new *Thing* entity as it would also mean that the *properties* field would be already filled, meaning that this *Thing* could properly be used to generate a *GetCapabilities* document. After filling form presented in Figures 7.3 that refer to the service identification, service properties and operations metadata. After issuing the request, the web application will generate the JSON for the HTTP POST request, according to the Listing 7.1. To verify if the newly created *Thing* actually occur in the database we can check it in three different ways. The first one is by going to the SensorThings API database and using a tool like PgAdmin3[1] to view the *Things* table. The Figure 7.1 shows a SELECT query to retrieve the newly created Thing entity. The second method is by issuing a GET request to the endpoint address to retrieve the Thing. The last method is by simply using the web application on which the data will be displayed in a table, as depicted in Figure 7.2.

```
1 {
2   "name": "Create Operation Test",
3   "description": "Get Capabilities test",
4   "properties": {
```

[1] https://www.pgadmin.org/

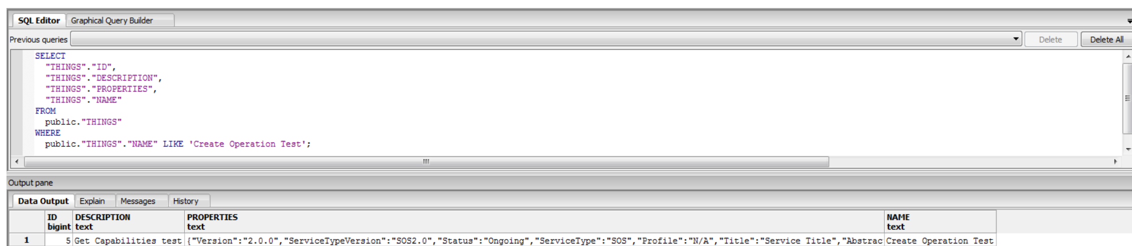


Figure 7.1: Form with the *properties* field containing metadata about the service.

```
5      "Version": "2.0.0",
6      "ServiceTypeVersion": "SOS2.0",
7      "Status": "Ongoing",
8      "ServiceType": "SOS",
9      "Profile": "N/A",
10     "Title": "Service Title",
11     "Abstract": "Testing the XML creation based on a JSON structure",
12     "Fees": "None",
13     "AccessConstraint": "Free Access",
14     "ProviderSite": "coral-tools.inesctec.pt",
15     "ProviderName": "Mercury",
16     "ServiceContact": {
17       "OrganisationName": "INESC-TEC",
18       "IndividualName": "Jose Alexandre Teixeira",
19       "EmailAddress": "jast@inesctec.pt",
20       "Role": "Student",
21       "ContactInfo": {
22         "HoursOfService": "N/A",
23         "ContactInstructions": "Email",
24         "Address": {
25           "DeliveryPoint": "Forge",
26           "City": "Porto",
27           "AdministrativeArea": "Paranhos",
28           "PostalCode": "N/A",
29           "Country": "Portugal",
30           "ElectronicMailAddress": "jast@inesctec.pt"
31         },
32         "Phone": {
33           "Voice": "N/A",
34           "Facsimile": "N/A"
35         }
36       }
37     },
38     "Date": "2018-06-15T12:15:43.000Z",
39     "ContainsOperations": [
40       {
41         "OperationName": "GET Things",
42         "HTTPMethod": "Get",
43         "ConnectPoint": "http://localhost:8080/sensorthings/v1.0/Things",
```

Name	Description
Things 1	Get Capabilities test
Things 2	Something that is used for testing
Things 3	Something that is used for testing
Create Operation Test	Get Capabilities test

Figure 7.2: Table containing all the *Thing* entities of an endpoint address.

```
44         "Metadata": {
45           "Metadata": "Metadata field",
46           "Link": "http://sensorup.com/docs",
47           "About": "Get all the things"
48         },
49         "Parameter": [
50           "ThingX",
51           "ThingY"
52         ]
53       },
54       {
55         "OperationName": "GET Datastreams",
56         "HTTPMethod": "Get",
57         "ConnectPoint": "http://localhost:8080/sensorthings/v1.0/Datastreams",
58         "Metadata": {
59           "Metadata": "Metadata field",
60           "Link": "http://sensorup.com/docs",
61           "About": "Get all the things"
62         },
63         "Parameter": [
64           "ThingX",
65           "ThingY"
66         ]
67       },
68       {
69         "OperationName": "GET ObservableProperties",
70         "HTTPMethod": "Get",
71         "ConnectPoint": "http://localhost:8080/sensorthings/v1.0/
             ObservableProperties",
72         "Metadata": {
73           "Metadata": "Metadata field",
74           "Link": "http://sensorup.com/docs",
75           "About": "Get all the things"
76         },
77         "Parameter": [
78           "ThingX",
79           "ThingY"
80         ]
81       },
82       {
83         "OperationName": "GET Sensors",
84         "HTTPMethod": "Get",
```

```
85          "ConnectPoint": "http://localhost:8080/sensorthings/v1.0/Sensors",
86          "Metadata": {
87            "Metadata": "Metadata field",
88            "Link": "http://sensorup.com/docs",
89            "About": "Get all the things"
90          },
91          "Parameter": [
92            "ThingX",
93            "ThingY"
94          ]
95        }
96      ]
97    }
98  }
```

Listing 7.1: Payload of a Thing POST request

7.2 Using pycsw and GeoNetwork for metadata publishing

Another key aspect to validate the degree of conformance of the GetCapabilites document generated is to use it with other cataloguing tools. After a thorough analysis of several available options the chosen implementation was pycsw due to its functionalities for publishing and discovery of geospatial metadata, including the OGC CSW metadata format.

In order to test it, firstly it was needed to install the OSGeoLive[2] distribution that already contains a pycsw instance installed. Afterwards, to proceed to the cataloguing of the sensors' metadata it was necessary to execute a series of commands to properly setup the database and the endpoint address of the devices that were to be harvested. The database is automatically created by running a configuration file, inherent to pycsw. However, to allow for the harvesting of the OGC SOS 2.0 it was necessary to create an XML that contains the endpoint address of the source to be harvested. In this case, the Go API that was developed provided a route whose main purpose was to supply the GetCapabilities document in an XML encoding for it to be processed by the pycsw. To test its conformance with a generic client, such as GeoNetwork that is also included in the OSGeoLive distribution, the pycsw instance needed to be able to be harvested and its information to be catalogued properly in the GeoNetwork platform.

7.2.1 Comparison of a Sensor Observation Service *GetCapabilities*

This first test is supposed to determine if the produced document is structurally similar with other *GetCapabilities* response. To assess this, there was made a comparison between the *GetCapabilities* document produced by the 52North SOS application and the metadata document produced by the application. In the code listings 7.2 and 7.3 it is shown each one of the results for the Service Identification field.

[2]https://live.osgeo.org/

```
1  "serviceIdentification" : {
2      "title" : {
3        "eng" : "52N SOS"
4      },
5      "abstract" : {
6        "eng" : "52North Sensor Observation Service - Data Access for the Sensor Web"
7      },
8      "accessConstraints" : [
9        "NONE"
10     ],
11     "fees" : "NONE",
12     "serviceType" : "OGC:SOS",
13     "profiles" : [
14       "http://www.opengis.net/extension/SOSDO/1.0/observationDeletion",
15       (...)
16       "http://www.opengis.net/spec/waterml/2.0/conf/xsd-xml-rules"
17     ],
18     "versions" : [
19       "1.0.0"
20     ]
21   },
```

Listing 7.2: Output of a GetCapabilities

```
1  "ServiceIdentification": {
2              "ServiceType": "SensorThings",
3              "ServiceTypeVersion": "0.0.1a",
4              "Profile": "SensorThings API OWS Profile",
5              "Title": "Raspberry C3",
6              "Abstract": "This is a brief narrative description of this server,
                   normally available for display to a human",
7              "Keywords": {
8                  "Keyword": ""
9              },
10             "Fees": "NONE",
11             "AccessConstraints": "N/A"
12         },
```

Listing 7.3: Output of the application

Despite a small difference between both results, nonetheless it is proven that the solution developed has the same structure. For sake of simplicity the fields Service Provider and Operations Metadata are omitted but they were compared and the differences noticed in the Service Provider field are very similar. However, due to the nature of the SensorThings API resource-oriented architecture the Operations Metadata field had to be restructured and some differences can be found between the snippets 7.4 and 7.5 where both represent an operation exposed by the server.

```
"Batch" : {
        "dcp" : [
          {
            "method" : "POST",
            "href" : "http://seabiotix.inescporto.pt:8080/52n-sos-webapp/service/
                json",
            "constraints" : {
              "Content-Type" : {
                "allowedValues" : [
                  "application/json"
                ]
              }
            }
          }
        ]
      },
```

Listing 7.4: Output of the SOS on operations metadata

```

  "Name": "POST Thing",
  "DCP": {
      "HTTP": {
          "HTTPVerb": "POST",
            "URL": "http://194.117.25.107/sensorthings/v1.0/Things",
            "Constraint": "N/A"
     }
  },
    "Parameter": null,
    "Constraint": "N/A",
    "Metadata": {
      "Metadata": "Field for metadata",
        "Link": "http://developers.sensorup.com/docs/",
        "About": "Create a Thing."
  }
```

Listing 7.5: Output of the application on operations metadata

7.2.2 Harvesting a SensorThings API endpoint

To test whether the backend application was able to correctly generate the *GetCapabilities* document pyCSW was used. To do this the pycsw method *post_xml* was responsible to request the *GetCapabilities* to the endpoint address of the backend application. This request will result in the backend application to query the *Thing* entity that corresponds to the given ID and endpoint address provided as parameters of the *GET* request. The configuration file of Listing 7.6 was used

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Harvest xmlns="http://www.opengis.net/cat/csw/2.0.2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.opengis.net/cat/csw/2.0.2 http://schemas.opengis.net/csw/2.0.2/CSW-publication.xsd" service="CSW" version="2.0.2">
  <Source>http://192.168.1.102:8081/getcapabilities?&endpointaddress=http://192.168.1.102:8080/sensorthings/v1.0/Things(2)&thingid=2&xml=false&parameters=all</Source>
  <ResourceType>http://schemas.opengis.net/sos/2.0/sosGetCapabilities.xsd</ResourceType>
  <ResourceFormat>application/xml</ResourceFormat>
</Harvest>
```

Listing 7.6: XML file to harvest the Mercury endpoint

to harvest a *Thing* entity with ID equal to the value 2 on the **Mercury** endpoint, known as having the IP address of 192.168.1.102 as the OSGeoLive instance was running on a local Virtual Machine.

7.2.3 Using GeoNetwork

After the pyCSW proceeded with harvesting the endpoint it now contains the data described by the *GetCapabilties* document. However, to enable the visualization of this information and to present it in GeoNetwork and to assess if the service was well described to prove it integrates with the CSW specification GeoNetwork was used to harvest the pyCSW instance running in the same machine. The following Figure depicts the interface of the GeoNetwork that will proceed with the harvesting.

7.3 Interoperability of the SensorThings API

One of the most important aspects over the development was to take into consideration the interoperability that the SensorThings API was designed to. Therefore all the queries that are issued by the backend application will be the same regardless of the SensorThings API server implementation. To assess this there were used some SensorThing API endpoints that were running different implementations and were also maintained by different entities. In the list below there is the description of the several endpoints that were used and a brief summary on the SensorThings API server implementation and the entities responsible for its maintenance.

- **StAlbert Wildfires** — contains crowdsourced data from sensors that measure the air quality in the region of St. Albert, Canada. It is managed by a SensorUp SensorThings Cloud Server.
- **Snapshot Toronto Bikes** — relates to Toronto's data on cycling and cyclists. It is also managed by a Sensorup SensorThings Cloud Server.
- **Scmix Server** — contains a huge collection of observation data dating to 2010 and is maintained by INESC-TEC. It is running a FROST implementation.

- **O2SOS Raspberry Pi & C3P0 Raspberry Pi** — Raspberry Pi that are running a FROST implementation additionally they have been configured to run a script that will be using the SenseHat module to gather data from the surrounding environment of Temperature, Humidity and Atmospheric Pressure. They were maintained within this work.
- **Mercury** — contains collections of data that was extracted from comma-separated values files, also running a FROST implementation. It was maintained within this work.

Depending on the endpoint address, some differences are expected on the sense the SensorThings API server is going to respond. For example, the **Scmix** server has a very large pagination which means that any entity that is going to be queried, the response will contain a very high number of result values. This means that if a *Datastream* contains over nine hundred thousand observations they will all be retrieved, this will naturally incur in a longer response time of a given request. To address, that particularity the SensorThings API has the option that a given query will only retrieve a given number of values with the **$top** parameter.

Moreover, the backend application works in the way that regardless of the endpoint address the query will always be issued the same. If there is a *@iot.NextLink* key in the JSON response then a new request will be issued to that address until the value returned of that key is null.

7.4 Creating an IoT platform

This section documents all the steps related to the creation of a sensor platform in order to evaluate the work from the widest perspective. The source of the data is from a comma-separated-values file which contains measurements of different physical properties such as air temperature, wind direction, barometric pressure and expressed in different units of measurement. The dataset being analyzed for this test occured during the period of the 11st of August, 2015 to the 19th of September of 2016 at roughly the same Location, in Galway, Ireland.

7.4.1 Setting up the IoT platform

Firstly, to allow for the insertion of *Observations* there needs to exist the underlying information to give a meaning of the measurements recorded in the dataset, also referred as the *results*. In order to do so, a new *Thing* was created to further associate the other entities to it. Moreover, after the creation of the *Thing*, its corresponding *Datastreams* were created. But before proceeding to the creation of the *Datastreams* it was necessary to also create its associated *Observable Property* and *Sensor* entities. To achieve this, the Device Management module functionalities were used to interface with the endpoint address to create these entities.

To understand which *Datastreams* ought to be created, it was needed to analyze the CSV file. For this specific approach it was assumed that a distinct column referring to a physical property would be associated to a single *Datastream*. Furthermore, to feed the SensorThings API server, running on the **Mercury** machine, with the *Observation* results, it was necessary to create a script in *JavaScript* that was responsible for parsing the file and generate the required HTTP Post requests

to start inserting the results in the STA database. The Listing 7.7 refers to one of the requests that was made, specifically to the *Datastream* whose ID is 5.

```
1  request.post('http://localhost:8080/sensorthings/v1.0/Datastreams(5)/Observations',
       {
2              body: {
3                  "phenomenonTime": elem.time,
4                  "resultTime": elem.time,
5                  "result": elem.dr1,
6                  "FeatureOfInterest": {
7                      "name": "Galway, Ireland",
8                      "description": "Location of the Feature Of Interest",
9                      "encodingType": "application/vnd.geo+json",
10                     "feature": {
11                       "type": "Point",
12                       "coordinates": [elem.lon, elem.lat]
13                     }
14                   },
15                 },
16                 json: true
17             });
```

Listing 7.7: HTTP Post Request

7.4.2 Querying the Observation Data

After parsing the CSV file and inserting its correspondent results into the STA database on the **Mercury** machine it would be now possible to use the developed tools refered in Chapter 6. Naturally, not all the Tools would be the best suited to present the data as there is no particular geographical richness since the measurements all refer to roughly the same geographical location. However, it is indeed possible to retrieve the measurements of the *Datastreams* and present them from a chart view. Therefore it was mostly used the tool referred in Section 6.5 to visualize the result values in charts. The final output of the chart for the *Datastream* Wind Speed, in the units of meters per second with filtering the first date as 11th of August, 2015 and the last date 25th of August, 2015 would produce the following chart depicted in Figure 7.4.

7.5 Conclusions

This chapter described several experiments that were aimed to each one of the modules that compose the whole solution. In order to verify whether the functionalities addressed the goal that this project was initially designed to, both web application and the backend application were used to perform several actions from a user perspective. Moreover, this chapter also includes the results of the validation of the proposed metadata model for the SensorThings API. It includes the static analysis

of a document produced from the backend application and a *GetCapabilities* document generated from a third-party implementation. And it also contains the result of the harvesting experiment executed by a pycsw instance of the *GetCapabilities* document that is generated to assess if it is valid.

Figure 7.3: Form with the *properties* field containing metadata about the service

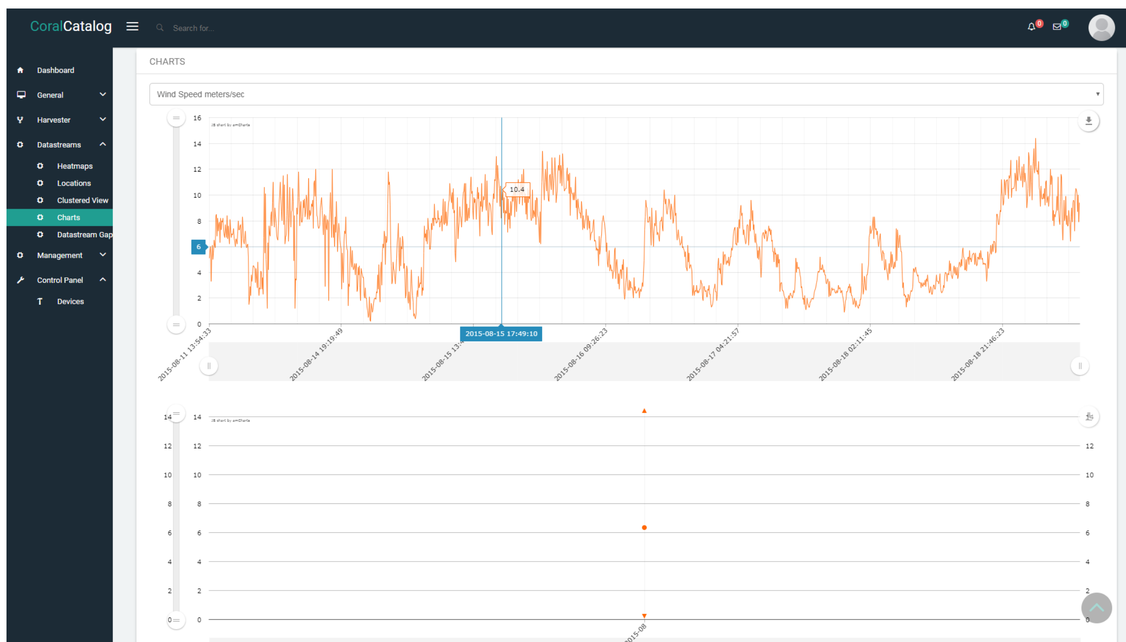


Figure 7.4: Values of the *Datastream* Wind Meters/second of the **Mercury** endpoint

Chapter 8

Conclusions and Future Work

This chapter sums all the conclusions of the overall work that was produced throughout the development of this dissertation. Furthermore it also will contain a brief description of the future work intended to be carried on and a small reflection on the limitations that arose during the development process.

8.1 Summary

The work presented here, focused on the development of several components that comprise to create and enable an heterogeneous IoT environment. This environment is composed by any number of independent platforms that can be managed by third-party organizations, or even individuals that want to expose their data to contribute to a richer environment. Inevitably leading to problems of interoperability between these platforms by using the SensorThings API specification this will contribute to mitigate the problem of communication between devices. By taking advantage of this specification this helped to build an application that will be responsible to establish a communication to these platforms with a common language, through the same requests and independently. Thus, it was possible to build an application that was responsible to manage the entities under the SensorThings API of each platform that is recognized in the application as well as enable the user to have an interface to proceed to that same management and maintenance thus avoiding the use of manual HTTP requests, by providing an intuitive web application interface. Moreover it was also possible to develop a set of tools that are responsible to retrieve the result of measurements carried out by the sensors of the IoT platform in a seamless way. This would lead to the production of higher-value outputs such as Heatmaps or aggregation of *Observation* records by their *Feature of Interest*. Moreover, it also makes possible to aggregate the data that the platforms contain and present it to the user without him having to know the underlying topology of these sensing platforms. And finally, another important aspect that this work focused on was the adaptation of a resource-based model into a compatible service-based model due to the need to create a *GetCapabilities* document that contains metadata information about the service. This would mean

that several other implementations of the OGC ecosystem would be able to communicate to enrich machine-to-machine discovery and operations.

After analyzing the state of the art it was possible to get to know the other initiative that are being carried out by organizations such as SensorUp and the tools they develop with the SensorThings API as well as giving the solution presented in this dissertation a more distinct approach for the interoperability problem. During the phase of creating the solution it was clear that to support the claim of being possible to create a multi-platform IoT environment based on SensorThings API it was needed to create an application based on a client-server architecture that creates an interface with its users through an web application, communicating with a backend application. This backend application operates with the data inherent to the application such as managing endpoints and users but will also take the task of communicating with the external Iot platforms to perform all the CRUD operations that are enabled by its underlying Resource-Oriented Architecture. To complement this solution and to assess the quality of the *GetCapabilities* document it was also used third-party applications such as pyCSW and GeoNetwork to perform harvesting operations. From the development perspective it encompassed in three different segments, one being the management of SensorThings API entities of a given IoT endpoint, the second being the harvesting and cataloguing of these endpoints and the last the development of tools to visualize the data contained in these endpoints. The web application was also used to prove as a validation mechanism as the more platforms were being added thorought the development it was possible to test how the proposed solution was behaving. As concluded, this complete solution proved to be a viable proof of concept in terms of creating a whole ecosystem of independant IoT platforms that share a same common language, the SensorThings API.

8.2 Future Work

This section introduces the work that is going to be developed to increase the functionalities of the current state of the web application. These features are important to fulfill the requirements of the project that this dissertation is based on.

Storage of results in the user's personal space

This feature allows for the storage of previous queries the user has issued. This is important due to the fact that we assume the platforms will not always be available. Therefore, if the user has the need to persist a particular segment of data from a query such as the *Observations* from a particular date there will have to be a mechanism to save it. The proposed idea for this is to implement a RESTful API in nodeJS that communicates with a MongoDB database. So, this application is intended to accept requests from the web application, those requests contain in its payload a pre-processed document that is ready to be stored in the database. This document contains information that the user might find relevant such as the result of a query of one of the tools available such as Heatmaps, as well as the entities of a SensorThings API endpoint such as the *Observations*. This document can afterwards be retrieved and presented in the web application. From the web application perspective

there should be an interface that displays the collection of documents that the user owns and each document can be displayed in the web application.

Further development of tools

The different tools that were developed are a good starting point for the creation of a rich mechanism to produce high-level visualization outputs of the SensorThings API entities. As it already contains the principles of multi-platform aggregation of entities, filtering options and pagination mechanisms it is possible to be further expanded with more options to view this data. Now, the development of these tools will take into consideration a deeper understanding on the *Observable Property* entity for instance. As the *Observable Property* is the entity that describes the physical property that is being measured there may be created new tools that will specifically better-suit and give a more indepth visualization, such as presenting deep sea measurements of temperature, conductivity and depth for instance.

References

[1] Arjun P. Athreya and Patrick Tague. Network self-organization in the Internet of Things. *2013 IEEE International Workshop of Internet-of-Things Networking and Control, IoT-NC 2013*, pages 25–33, 2013.

[2] Emmanuel Baccelli, Cenk Gundogan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wahlisch. RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 4662(c):1–12, 2018.

[3] Adhitya Bhawiyuga, Mahendra Data, and Andri Warda. Architectural Design of Token based Authentication of MQTT Protocol in Constrained IoT Device. *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*.

[4] Arne Bröring, Christoph Stasch, and Johannes Echterhoff. OGC Sensor Observation Service. *OGC Implementation Standard*, page 163, 2012.

[5] Nicola Bui and Michele Zorzi. Health care applications. *Proceedings of the 4th International Symposium on Applied Sciences in Biomedical and Communication Technologies - ISABEL '11*, (August):1–5, 2011.

[6] Angelo P. Castellani, Nicola Bui, Paolo Casari, Michele Rossi, Zach Shelby, and Michele Zorzi. Architecture and protocols for the internet of things: A case study. *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops, PERCOM Workshops 2010*, pages 678–683, 2010.

[7] Francisco Cercas and Nuno Souto. Comparison of Communication Protocols for Low Cost Internet of Things Devices. *Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM), 2017 South Eastern European*, 2017.

[8] Luis Cruz-Piris, Diego Rivera, Ivan Marsa-Maestre, Enrique De La Hoz, and Juan R. Velasco. Access control mechanism for IoT environments based on modelling communication procedures as resources. *Sensors (Switzerland)*, (3), 2018.

[9] Dave Evans. The Internet of Things - How the Next Evolution of the Internet is Changing Everything. *CISCO white paper*, (April):1–11, 2011.

[10] Weichao Gao, James Nguyen, Wei Yu, Chao Lu, and Daniel Ku. Assessing Performance of Constrained Application Protocol (CoAP) in MANET Using Emulation. *Proceedings of the International Conference on Research in Adaptive and Convergent Systems - RACS '16*, pages 103–108, 2016.

[11] GOST. GOST, 2018. Available at `https://www.gostserver.xyz/`, accessed in 2018-06-23.

[12] Dominique Guinard, Vlad Trifa, and Erik Wilde. A resource oriented architecture for the Web of Things. *Proc. of 2010 Internet of Things (IOT'10)*, pages 1–8, 2010.

[13] Ayesha Hafeez, Nourhan H. Kandil, Ban Al-Omar, T. Landolsi, and A. R. Al-Ali. Smart home area networks protocols within the smart grid context. *Journal of Communications*, 9(9):665–671, 2014.

[14] International Telecommunication Union. Overview of the Internet of things. *Series Y: Global information infrastructure, internet protocol aspects and next-generation networks - Frameworks and functional architecture models*, page 22, 2012.

[15] ISO. ISO 19156: 2011-Geographic information: Observations and measurements. *Open Geospatial Consortium. Implementation Standard*, 2011:54, 2011.

[16] Alexander Kotsev, Katherina Schleidt, Steve Liang, and Hylke Van Der Schaaf. Extending INSPIRE to the Internet of Things through SensorThings API. (May):1–21, 2018.

[17] Chao Hsien Lee, Yu Wei Chang, Chi Cheng Chuang, and Ying Hsun Lai. Interoperability enhancement for Internet of Things protocols based on software-defined network. *2016 IEEE 5th Global Conference on Consumer Electronics, GCCE 2016*, pages 4–5, 2016.

[18] Martine Lenders, Peter Kietzmann, Oliver Hahm, Hauke Petersen, Cenk Gündoğan, Emmanuel Baccelli, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. Connecting the World of Embedded Mobiles: The RIOT Approach to Ubiquitous Networking for the Internet of Things. (2), 2018.

[19] Juan Li, Yan Bai, Nazia Zaman, and Victor C. M. Leung. A Decentralized Trustworthy Context and QoS-Aware Service Discovery Framework for the Internet of Things. *IEEE Access*, 5:19154–19166, 2017.

[20] Ming Li, Xinyan Zhu, and Yifang Mei. Incremental harvesting model of distributed geospatial data registry center based on CSW. *Proceedings - 2011 19th International Conference on Geoinformatics, Geoinformatics 2011*, (126), 2011.

[21] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications. *IEEE Internet of Things Journal*, 4(5):1125–1142, 2017.

[22] Sebastian Meiling, Dorothea Purnomo, Julia-Ann Shiraishi, Michael Fischer, and Thomas C Schmidt. MONICA in Hamburg: Towards Large-Scale IoT Deployments in a Smart City. 2018.

[23] Tino Miegel and Richard Holzmeier. High Level Architecture. pages 1–15, 2002.

[24] Umakanta Nanda and Sushant Kumar Pattnaik. Universal Asynchronous Receiver and Transmitter (UART). *ICACCS 2016 - 3rd International Conference on Advanced Computing and Communication Systems: Bringing to the Table, Futuristic Technologies from Arround the Globe*, 2016.

[25] OGC. OGC Web Services Common Specification. *Open Geospatial Consortium Technical Reports*, page 167, 2007.

[26] OGC. OGC SensorThings API Part 1 : Sensing. *Open Geospatial Consortium. Implementation Standard*, pages 1–105, 2016.

[27] oneM2M. Technical Specification TS-0001-V2.10.0: Functional Architechture. 2.10.0:1–427, 2016.

[28] Maria Rita Palattella, Nicola Accettura, Xavier Vilajosana, Thomas Watteyne, Alfredo Grieco, Luigi, Gennaro Boggia, and Mischa Dohler. Standardized Protocol Stack for the Internet of (Important) Things. pages 1–18, 2012.

[29] Hetal B Pandya and Tushar A Champaneria. Internet of things: Survey and case studies. *2015 International Conference on Electrical, Electronics, Signals, Communication and Optimization (EESCO)*, pages 1–6, 2015.

[30] Public Engineering Report, Ingo Simonis, and Engineering Report. Open Geospatial Consortium, Inc. *Engineering*, pages 10–061, 2010.

[31] Design Resources, Design Features, and Featured Applications. PM 2.5 /PM 10 Particle Sensor Analog Front-End for Air Quality Monitoring Design. (May):1–44, 2016.

[32] SensorUp. SensorUp SensorThings Dashboard, 2018. Available at https://www.sensorup.com/situation-management/, accessed in 2018-06-23.

[33] Jasper Tan and Simon G.M. Koo. A survey of technologies in internet of things. *Proceedings - IEEE International Conference on Distributed Computing in Sensor Systems, DCOSS 2014*, pages 269–274, 2014.

[34] Andrew S Tanenbaum. *Computer Networks*, volume 52. Prentice Hall, 1996.

[35] José Alexandre Teixeira, Artur Rocha, and João Correia Lopes. Extending the SensorThings API to enable an OGC OWS compliant service chain. 2018. Prepared for submission.

[36] Kevin I.K. Wang, Waleed H. Abdulla, and Zoran Salcic. Ambient intelligence platform using multi-agent system and mobile ubiquitous hardware. *Pervasive and Mobile Computing*, 5(5):558–573, 2009.

[37] Chia Wei Wu, Fuchun Joseph Lin, Chia Hong Wang, and Norman Chang. OneM2M-based IoT protocol integration. *2017 IEEE Conference on Standards for Communications and Networking, CSCN 2017*, pages 252–257, 2017.

[38] Shanhe Yi, Cheng Li, and Qun Li. A Survey of Fog Computing: Concepts, Applications and Issues. *Proceedings of the 2015 Workshop on Mobile Big Data - Mobidata '15*, pages 37–42, 2015.

[39] a Zanella, N. Bui, a Castellani, L. Vangelista, and M. Zorzi. Internet of Things for Smart Cities. *IEEE Internet of Things Journal*, 1(1):22–32, 2014.