

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Data Modeling for the Internet of Things

Bruno Rafael Leite Ribeiro



FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Costa Aguiar

July 5, 2020

Data Modeling for the Internet of Things

Bruno Rafael Leite Ribeiro

Mestrado Integrado em Engenharia Informática e Computação

Abstract

Internet of Things (IoT) has become more popular in recent years and it is expected that by 2022 there will be 75 billion IoT connected devices generating data. These devices include sensors, mobile phones and other smart objects from areas ranging from home automation to healthcare and smart manufacturing. One of the challenges in developing IoT applications is the representation and storage of this data. Every IoT platform is creating its own model for modelling device information, which makes interoperability between devices and data exchange in this environment more difficult.

Accordingly, the main objective in this work is to define a meta data model which is compliant with the current standards in the IoT area. Based on the exploration of these standards, a meta data model was chosen, where it provides semantics on top of the raw data stored, in order to obtain more information for the user.

The second objective is to create a mechanism to recognize sensor data without user input. The system supports all the observation values from the disparate data sources with appropriate meta-data, relating to the magnitude of a physical quantity denoted by a unit of measure. Consequently, this work needs to have a semantically rich way of registering these properties.

Therefore, it is necessary to store the data correctly so it can be queried in an optimized manner. Also, not only the data generated from the IoT devices, but also the metadata associated should be stored. Thus, performance and a flexible database are important factors to consider.

Finally, an evaluation of the proposed meta-model and system architecture was conducted. Different sensors were used to test the accuracy of the system, namely the possibility to register different information and label it correctly.

Keywords: IoT, meta-model, interoperability, semantics, data model, ontology, database, sensors

Resumo

A Internet das Coisas (IoT) tornou-se predominante nos últimos anos e estima-se que em 2022 existam 75 bilhões de dispositivos IoT conectados gerando dados. Esses dispositivos incluem sensores, dispositivos e outros objetos inteligentes, de áreas que vão da automação residencial a cuidados de saúde e fabricação inteligente. Um dos desafios no desenvolvimento de aplicações de IoT é a representação e o armazenamento desses dados. Isto porque todas as plataformas de IoT estão a criar o seu próprio modelo para modelar informações de dispositivos, o que dificulta a interoperabilidade entre esses e a troca de dados.

Consequentemente, o principal objetivo é definir um meta-modelo que seja compatível com os padrões atuais na área de IoT. Com base na exploração desses padrões, modelos compartilhados podem ser criados para permitir aos sistemas a troca de informação, fornecendo também semântica sobre os dados brutos e a correlação dos mesmos.

O segundo objetivo é criar um mecanismo para reconhecer os dados do sensor sem o utilizador ter de fornecer informação ao sistema. Isso significa que o sistema suporta todos os valores de observação das fontes de dados diferentes com metadados adequados e precisos, relacionados à magnitude de uma quantidade física indicada por uma unidade de medida. Desenvolveu-se e aplicou-se uma metodologia consistente que produz uma maneira semanticamente rica de registrar essas propriedades.

Portanto, é necessário otimizar o armazenamento e o acesso aos dados; não apenas os dados gerados a partir dos dispositivos IoT, mas também a meta data dos mesmos. Assim, a escalabilidade dos dados e uma base de dados flexível são fatores importantes a serem considerados. Finalmente, uma avaliação do meta-modelo proposto e da arquitetura do sistema foi feita. Diferentes sensores foram usados para testar o desempenho do sistema, nomeadamente a possibilidade de alterar a informação e a escalabilidade dos dados.

Keywords: IoT, meta-model, interoperability, semantics, data model, ontology, database, sensors

Acknowledgements

First, I would like to thank my supervisor, Ana Aguiar for all the help and guidance during all the phases of the work. I would also like to express my gratitude to all the people in the Domatica company for their orientation and more specifically to Pedro Santos and Ricardo Tavares.

To all my colleagues I met in the past five years, they helped me grow with their support, motivation and good spirit.

To my family, I want to express my gratitude for their genuinely support, love and encouragement.

To all the aforementioned, my sincerest thanks.

Bruno Rafael Leite Ribeiro

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation and Goal	2
1.3	Dissertation's Structure	2
2	Literature Review	5
2.1	Types of interoperability	5
2.2	Communication Protocol	6
2.3	Database Architecture	7
2.3.1	SQL	8
2.3.2	NoSQL	8
2.3.3	TSDB	9
2.4	Data Models	9
2.4.1	Eclipse Vorto Project	9
2.4.2	Fiware Initiative	10
2.4.3	SensorThings API	12
2.5	Ontologies	12
2.5.1	SSN	13
2.5.2	OneM2m Base	14
2.5.3	SAREF	15
2.6	QUDT ontology	15
2.7	SensorThings API Implementations	15
2.7.1	FROST-Server	16
2.7.2	52North	16
2.7.3	SensorUp	16
2.7.4	GOST	17
2.7.5	Mozilla	17
2.7.6	CGI Kinota Big Data	17
3	Problem Statement and Solution Proposal	19
3.1	Problem	19
3.1.1	Choice and implementation of a meta data model	19
3.1.2	Autonomous registration	20
3.2	Solution Proposal	20
3.2.1	System Architecture	21
3.2.2	InfoReceiver	22
3.2.3	System Logic	23
3.2.4	Discover Property	23

3.3 Conclusions	24
4 Choice and implementation of a meta data model	25
4.1 Meta Model Choice	25
4.2 Sensor Things API implementations	26
4.3 Structure	26
4.3.1 Endpoints	27
4.3.2 Register and give contextual information	28
4.3.3 Upload readings	29
4.3.4 Filtering information	29
4.4 Conclusions	29
5 Autonomous Registration	31
5.1 QUDT	31
5.1.1 Unit	31
5.1.2 Quantity Kind	32
5.2 Architecture	32
5.3 String matching algorithms	34
5.3.1 Hamming distance	35
5.3.2 Levenshtein distance	36
5.3.3 Jaro-Winkler	38
5.4 Conclusions	40
6 Evaluation	41
6.1 Experiment	41
6.2 Dataset Description	42
6.3 Perfomance Metrics	43
6.3.1 Confusion Matrix	43
6.3.2 ROC Curve	44
6.4 Results	45
6.4.1 Jaro-Winkler Algorithm	46
6.4.2 Hamming Algorithm	47
6.4.3 Levenshtein Algorithm	47
6.5 Discussion	49
7 Conclusion and Future Work	51
References	53

List of Figures

2.1	MQTT protocol with Domatica	7
2.2	Different Types of Databases	7
2.3	Eclipse Vorto Web Interface	10
2.4	Fiware Platform	11
2.5	SensorThings Data Model	12
2.6	SSN Ontology	13
2.7	oneM2M Base Ontology	14
2.8	QUDT: basic structure	16
3.1	System architecture	21
3.2	Sequence diagram	22
3.3	Finding entities not registered	24
4.1	Frost relational model	27
4.2	Server structure	27
4.3	Register datastream	28
4.4	Upload observation	29
5.1	QUDT: Unit	32
5.2	QUDT: QuantityKind	33
5.3	Discover Property Flowchart	34
6.1	Dataset head	42
6.2	Unit Graph Bar	43
6.3	Confusion matrix	44
6.4	ROC Curve	45
6.5	ROC Curve - Jaro-Winkler	47
6.6	ROC Curve - Hamming	49
6.7	ROC Curve - Levenshtein	49

List of Tables

4.1	Comparison of data models	26
6.1	Results experiment	45
6.2	Breakdown string matching units	46
6.3	Results Jaro-Winkler	47
6.4	Confusion matrix Jaro-Winkler	48
6.5	Results Hamming	48
6.6	Confusion matrix Hamming	48
6.7	Results Levenshtein	48
6.8	Confusion matrix Levenshtein	50

Abbreviations

API	Application Programming Interface
FN	False Negative
FP	False Positive
FPR	False Positive Rate
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
M2M	Machine to Machine
MQTT	Message Queue Telemetry Transport
NoSQL	Not only SQL
OGC	Open Geospatial Consortium
OSS	Operations Support Systems
OWL	Ontology Web Language
QUDT	Quantities, Units, Dimensions and Types
REST	Representational State Transfer
RDF	Resource Description Framework
ROC	Receiver Operating Characteristic
SAREF	Smart Appliance REference
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSN	Semantic Sensor Network
TN	True Negative
TP	True Positive
TPR	True Positive Rate
TSDB	Time Series Database
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
Wi-Fi	Wireless Internet

Chapter 1

Introduction

1.1 Context

The term Internet of Things (IoT) was first mentioned by Kevin Ashton, around 1999 [5]. Since then, Internet of Things had major developments and it is estimated that in 2022 there will be around 75 billion devices generating information. This data is from the most varied areas, ranging from health, transport, home automation or environment [28].

Nowadays, the IoT market is fragmented, because there is a high level of heterogeneity of device protocols, network connectivity, application protocols, standards, data formats and so on. Each IoT platform is creating its own data models to store that information [16]. The concept of IoT is already established and implementations are multiple and there are a widely accepted standard models to allow large-scale interoperability. Standardization projects are working on defining standards and protocols and their adoption requires consensus from users and developers through testing. Currently, there are several proposed open standard solutions already available, each differing slightly in functionality, specialization and structure, but with interoperability as the main objective.

This work was proposed and hosted by Domatica¹, which is a portuguese company specialized in IoT technology, focused on bridging the gap between the physical world and the logical world of software. The company was created in 2002 and today there are 46 countries currently with their technology. Currently, there are more than 1 million devices connected and more than 1 billion data stream per day, using the technology.

Thus, a major challenge is the question of how to manage and store this data in a way that meets the requirements of the applications in real time. There should be an emphasis in the performance of the database, because this performance is directly related to the applications using this data.

¹<https://www.domatica.io>

1.2 Motivation and Goal

The motivation for this work is to advance a little further on the field of data modeling in IoT and to look at potential open standard solutions in the IoT area.

This master thesis aims to allow the creation of a meta data-model compatible with these standards, in order to allow objects to communicate, that is, interoperability between devices. This will provide the existence of semantics, in order to obtain more information regarding these devices and the data they provide. Having all of the data stored from the devices in itself doesn't satisfy all the necessities. 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 and future developers can access this data.

The second objective is to create a mechanism to recognize sensor data without user input. This means, the system supports and tries to find the physical quantity denoted by a unit of measure, from the data coming from the sensors. Removing the user from this process will increase the system performance, allowing him to focus on other necessities and even gaining more information about the data properties compared to a manual process.

At the end, the performance of the system was evaluated, by conducting some experiments with sensors from Domatica and registering them into the system. The goal is to evaluate the performance of the meta data model and to test the accuracy of the mechanism implemented in registering these sensors.

1.3 Dissertation's Structure

The dissertation is divided in the following chapters:

1. Introduction [1](#)

The introduction chapter gives an overview of the overall context of the dissertation, motivation and goal of the developed work.

2. Literature Review [2](#)

This chapter refers to the state of the art of the work and it can be divided in two sections regarding data modeling for internet of things. The chapter contains two ways that are different but complementary approaches to solve the issue in question.

3. Problem Statement and Solution Proposal [3](#)

In this chapter, the methodology of this work is presented. Furthermore, the solution to the issues in question are shown, with the whole architecture of the project.

4. Choice and implementation of a meta data model [4](#)

This refers to the data model standard chosen (SensorThings API) and its implementation. Here, it is explained how it works and how it connects to the rest of the project.

5. Autonomous Registration [5](#)

The autonomous registration chapter contains the unit ontology used and its features. This chapter contains a detailed architecture of the solution to the registration process and all the algorithms used.

6. Evaluation [6](#)

This chapter contains the experimental results obtained and how these experiments were performed. These experiments will evaluate both of the two objectives of this work. Then, a discussion of the results is presented.

7. Conclusion and Future work [7](#)

The conclusion chapter will sum up the work done, and propose future additions to this work, that could be made in the future, in order to improve the project.

Chapter 2

Literature Review

This chapter is divided in two parts. First, it is important to look at how the main IoT organizations present their data models and the main interoperability problems related. Afterwards, the topic of ontologies will be focused and how they can be used to achieve semantic interoperability. In all of them, examples will be provided and their specifications will be analysed.

2.1 Types of interoperability

The IEEE defines interoperability as “the ability of two or more systems or components to exchange information and to use the information that has been exchanged” [13].

However, this issue of interoperability can have different levels. Standard organizations and open source communities have been working to address interoperability issues in different parts or levels [22]. According to [23], interoperability in IoT can be divided in different categories: device interoperability, network interoperability, syntactical interoperability, semantic interoperability, and platform interoperability.

- **Device interoperability**

Device interoperability refers to enable the integration heterogeneous communication technologies and standards supported by different IoT devices. These devices can be separated in two categories: “low-end” and “high-end”. The high-end IoT devices have more computational capabilities such as Raspberry Pi and smartphones. On the other hand, the low-end IoT devices have less processing power. The standard communication protocols used for smart devices are WiFi, 2G/3G/4G, LoRa ANT+, ZigBee. However, not all smart devices can implement all these communication technologies.

- **Network interoperability**

Network interoperability relates to each system being able to exchange messages with other systems through various types of networks. The network interoperability level should handle issues such as addressing, routing, resource optimization, security, QoS, and mobility support.

- **Syntactical interoperability**

This interoperability refers to operate between different data formats in IoT systems. This happens, when the sender's encoding rules are incompatible with the receiver's decoding rules. To solve this problem, web technologies such as HTTP, MQTT, REST and SOAP architecture of the World Wide Web were proposed. The content of the messages need to be serialized to be sent over the channel and the format to do so (such as XML or JSON).

- **Semantic interoperability**

The data models and schemas used by different sources are usually dissimilar and not always compatible. This is the main level of focus of the work. Semantic interoperability is achieved when interacting systems assign the same meaning to part of the data exchanged. To solve this, it is possible to use ontologies.

- **Platform interoperability**

Currently IoT providers like Amazon AWS IoT, Apple HomeKit, Google Brillo and others are providing different operating systems (Oss), programming languages, and data structures. The consequence is that developers need to obtain skills and knowledge to deal with each different platform specific APIs and information model, because they vary from each other.

Even though, there are a great number of types of interoperability in IoT, the focus of this work will be mainly regarding semantic interoperability. To accomplish this objective, it is possible to use data models with context-awareness or use an ontology.

2.2 Communication Protocol

Communication is one of the main elements of IoT because there is the need to exchange information between devices [24]. As stated before, the standard communication protocols used for smart devices are WiFi, 2G/3G/4G, LoRa, ANT+, ZigBee. However, after the information is received from these devices into the system, Domatica uses another protocol, namely MQTT.

The MQTT protocol implements many levels of Quality of Service and has a high level of efficiency. This protocol is based on the concept of topics; to which clients can publish updates or subscribe for receiving updates from other clients, in order to exchange information.

The interaction of the application with Domatica is represented in Figure 2.1. First, all the sensors send their data to the Domatica, where the information is processed. Then, the application

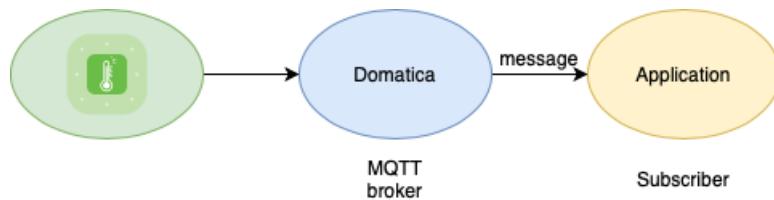


Figure 2.1: MQTT protocol with Domatica

takes the role of the subscriber receiving messages, containing the data processed, from a MQTT broker from Domatica.

With increasing demand, MQTT libraries now support iOS and Android mobile platforms, major IoT development platforms (eg Arduino) and different programming languages. The MQTT community claims the concept of a publication/subscription protocol is the main component for building the future of IoT [12].

This architecture provides a data-oriented protocol, which is more suitable for IoT and can also reduce the burden on a restricted device when exchanging messages. Sometimes, as the publisher and the subscriber do not know each other, there is the need to implement an authentication method, to secure the communication and validate the sender and recipient nodes [8].

2.3 Database Architecture

As referred before, IoT devices generate a large quantity of data, that needs to be saved. It is then necessary to properly treat the storage and access of this data in an optimized way. There are different ways to manage and store data in IoT, including a relational approach (SQL), non-relational storage (NoSQL), which can be divided in four categories, and Time Series Database (TSDB). As represented on Figure 2.2, it is possible to observe the different types of databases with an example of each type.

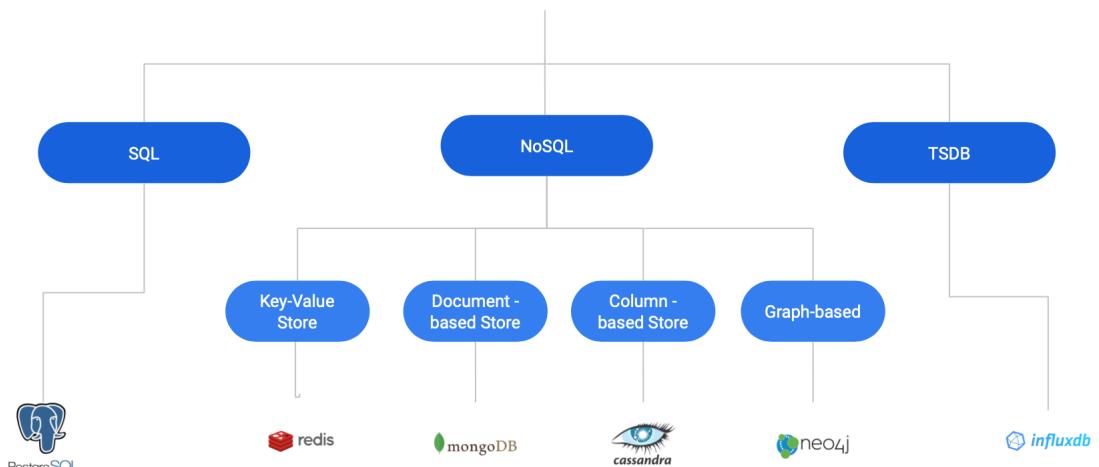


Figure 2.2: Different Types of Databases

2.3.1 SQL

SQL is the standard language for dealing with relational databases. Some of the key advantages of relational databases are that relational models structure the data in a table structure, a format that most users are familiar with. This simplicity also helps users to retrieve certain data more easily by querying through any table, combining related tables to filter out unwanted data and allowing them to retrieve the data easier than navigating a pathway through a tree or hierarchy.

Another advantage of relational databases is that they have been the mainstream databases for so long that most database managers are so used to them and take them for granted.

2.3.2 NoSQL

NoSQL ("not just SQL") is an approach to database design that includes a wide variety of data models, including key-value formats, documents, columns and graphs; which are presented below. This is an alternative to traditional relational databases in which data is placed in tables. These database surged to tackle the issue of dealing with a lot of data types and having the need to build powerful web and cloud applications for a distributed and quickly growing user base.

Many NoSQL stores compromise consistency in favor of availability, partition tolerance, and speed. NoSQL databases are faster in completing simple tasks, but more time-consuming for complex operations. These databases are more flexible and easily scalable. There are still some disadvantages to this option, like the use of low-level query languages (instead of SQL, for instance), lack of ability to perform ad-hoc joins across tables and lack of standardized interfaces.

- **Key-Value Store**

They represent the simplest type, where they basically store data following a unique key and value model, where hash maps can be defined for these pairs of values.

- **Document-Based Store**

These types of databases, unlike the key-value type, have the characteristic of storing any types of objects, which are called "documents". They use JSON to organize pairs containing a unique document identifier and the document itself.

- **Column-Based Store**

Represent a model that works with lines and columns, where they are divided into multiple nodes to provide scalability using hash functions.

- **Graph-based**

They represent a model that follows the graph theory, which works with nodes and edges, and databases of this type deal with stored objects that are strongly interrelated, such as when you want to discover the relationship between people in a social network

2.3.3 TSDB

Time-series databases are designed to store data that changes with time and are collected over time [2].

Initial uses of this technology were in industrial applications, which could efficiently store measured values, but now it is used in support of a much wider range of applications.

These databases have an architectural design that makes them very different from other databases. This includes storing and compressing data with timestamps and the ability to handle large time series-dependent checks for many records and time series-aware queries.

2.4 Data Models

A data model can be described as an abstraction of real-world entities in an organized model of data elements. A data model explains how these entities connect and relate between each other. It also documents the way data is stored and retrieved.

Due to the diversity of data models, it becomes difficult to know each one to select and implement. To solve this, [19] proposed eleven criteria to better evaluate and compare alternative data models. From these criteria, it was chosen in this work: simplicity, elegance and applicability of safe implementation techniques and provision of schemas.

Data models vary from SDO (Standards Developing Organizations) to SDO. Some of the most important SDO's are HyperCat and Mozilla IoT.

- **HyperCat** - is an open, lightweight JSON-based hypermedia catalogue format for exposing collections of URIs. Each HyperCat catalogue lists and annotates any number of URIs (which typically identify data sources), each having a set of relation-value pairs (metadata) associated with it. In this way, HyperCat allows a server to provide a set of resources to a client, each with a set of semantic annotations.
- **Mozilla IoT** - The Mozilla IoT team created a Web of Things implementation, Mozilla WebThings. This implementation is an open platform for monitoring and controlling devices over the web and is composed by: WebThings Gateway and WebThings Framework. WebThings Gateway is a software distribution for smart home gateways focused on privacy, security and interoperability. WebThings Framework is a collection of reusable software components to help developers build their own web things.

Therefore, there is a huge chance that one IoT device may have multiple data models, to cater to each SDO. Next, it will be presented examples of some of the most used models [14] .

2.4.1 Eclipse Vorto Project

Eclipse Vorto¹ is an open source project, which has the goal manage abstract device descriptions (information models). The models are mainly managed and shared in through a repository, called

¹<https://www.eclipse.org/vorto/>

the Vorto Repository. Then, the users have the ability to upload models, search the repository for what they need or to reuse these information models. An example of what this repository looks like is presented in Figure 2.3. These models are written in Vorto Language, but they can be exported in JSON.

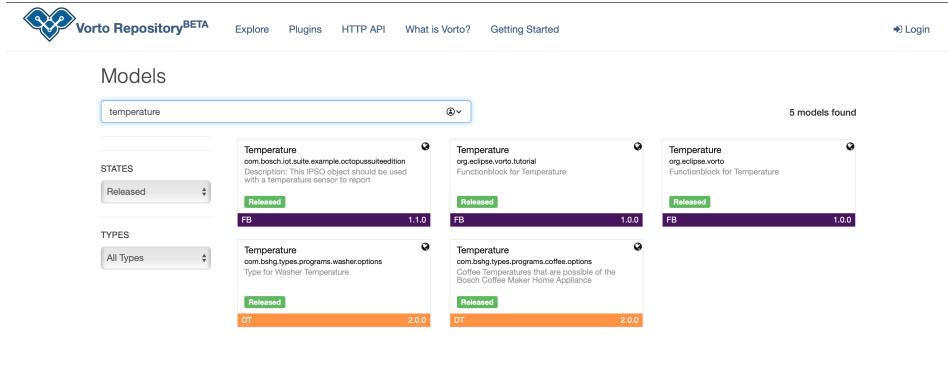


Figure 2.3: Eclipse Vorto Web Interface

Moreover, device manufacturers upload their information models for users to download, using the Vorto Code generator. This is one of the main advantages of Eclipse Vorto, providing interoperability and integrating the devices far more easily.

2.4.2 Fiware Initiative

The Fiware Initiative² is an open source platform created from a program called Future Internet Private Public Partnership. The platform focus mainly in smart cities, however it is being also used in industrial production cases and utilities projects. Here, the data models are divided in twelve categories from the topic mentioned. Each data model is programmatically defined using a JSON Schema. These data models are saved using a MongoDB database. It is open source and thus one can improve the platform if necessary [26].

The platform uses an IoT agent called NGSI to fix the problems of having different environments, where devices, containing different protocols, approach to a common format. The NGSI is a standard set of rules to communicate with a RESTful API.

Another important component of the platform is the FIWARE Context Broker. This component is responsible of the management of Context Information at large scale. This is used by the applications or by potential consumers. It gathers and manages context information, processing the information and informing external actors, which are enable them to actuate and therefore change and improve the current context.

²<https://www.fiware.org>

FIWARE DATA MODELS

Welcome to FIWARE Data Models. These data models have been harmonized to enable data portability for different applications including, Smart Cities, Smart Agrifood, Smart Environment, Smart Sensing, Smart Energy, Smart Water and others domains. Available at <https://github.com/smart-data-models>. They are intended to be used wherever you want but with compliance to FIWARE NGSI version 2 and NGSI-LD. If you want to contribute and create additional data models, please have a look at our [data model development guidelines](#) and fill the form. Please note that as part of a unified approach to smart data, the FIWARE Data Models, along with contributions from GSMA and TMForum

Data models are stored in repositories. The lower level repository is a Topic (i.e. Alert, Streetlighting, etc). Every topic repository is aggregated into Domain repositories. Domain repositories compile several topics (i.e. inside Smart Environment there are 3 topics, Environment, Waste management and Weather). At the same time a topic could appear in several domains (i.e. Weather appear in Smart cities and in Smart Environment and in Smart Cities).

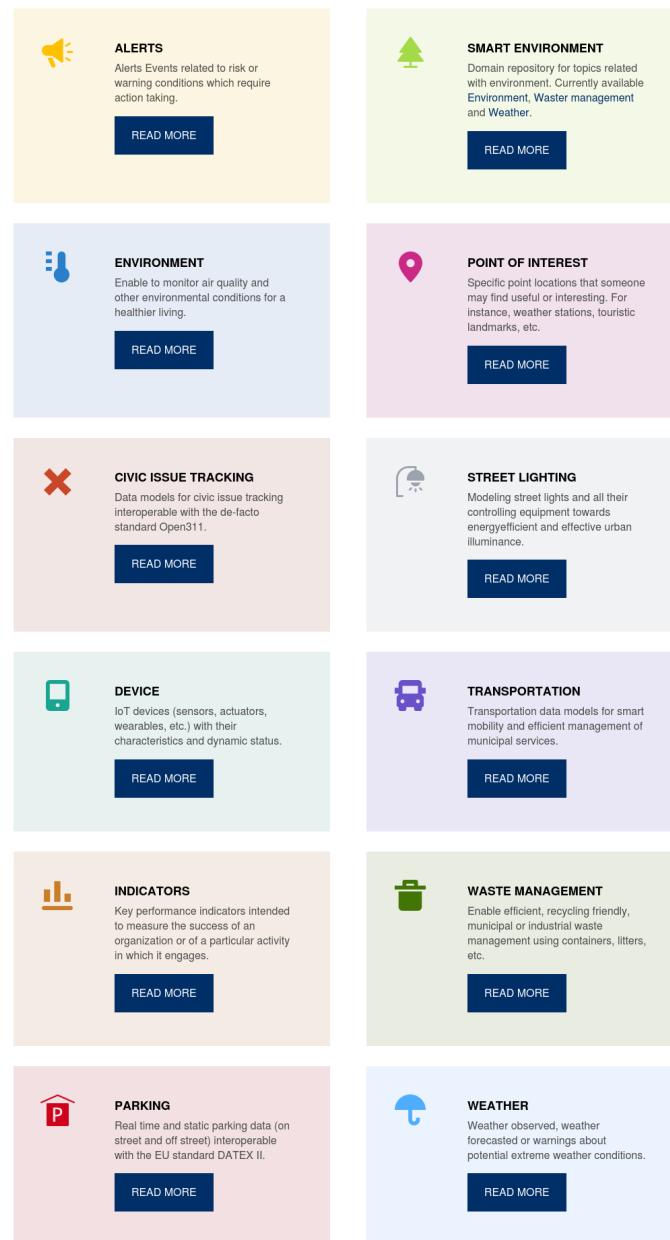


Figure 2.4: Fiware Platform

2.4.3 SensorThings API

The OGC SensorThings API provides an open, geospatial-enabled and unified way to interconnect the Internet of Things devices, data, and applications over the Web. The OGC SensorThings API is an open standard, platform-independent, and perpetual royalty-free. [17]

The OGC SensorThings is different from the two IoT platforms mentioned above (Eclipse Vorto Project and Fiware) because it is an abstract data model, instead of having a group of models specific for the type of sensor. The data model consists of two parts: the Sensing and Tasking profile. The first part (Sensing Profile) manages the IoT data and metadata from heterogeneous IoT sensor systems. The second part (The Tasking Profile) gives a standard manner for parameterizing of taskable IoT devices, for example actuators or sensors.

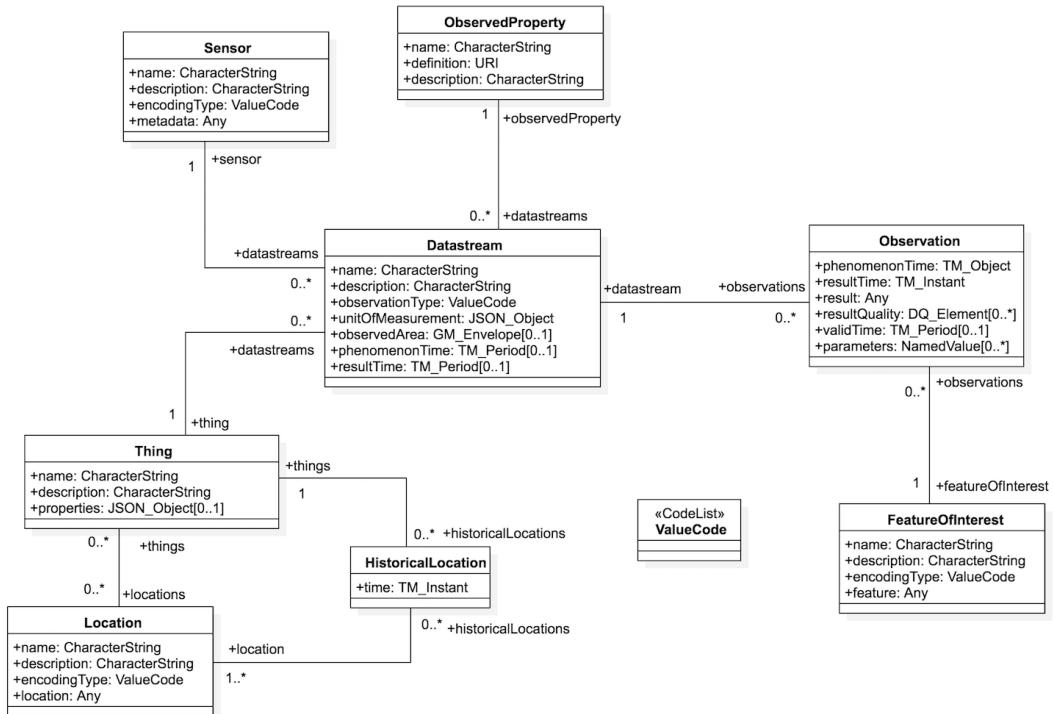


Figure 2.5: SensorThings Data Model

Looking at Figure 2.5, it is possible to observe that the model is based on Observations and Measurements.

2.5 Ontologies

Ontologies are defined as a “well-founded mechanism for the representation and exchange of structured information” [27]. They are a set of objects and relationships, which represent an area of concern. This way, they try to hide the heterogeneity of IoT. Ontologies extend the concept of taxonomy, in order to capture relationships capable of supporting richer operations and higher levels of reasoning, in a certain domain [20].

The goal of an ontology is to provide a semantic framework which can solve the problem of heterogeneity and interoperability associated with domain applications. This requires available ontologies to be comprehensive, readable, extensible, scalable, and reusable [6].

In the context of IoT, many ontologies have been proposed. Some of those ontologies proposed, are introduced as follows.

2.5.1 SSN

The W3C (World Wide Web Consortium), in 2012, proposed a standard ontology, SSN (Semantic Sensor Network). This ontology describes the sensor resources and the data collected through these sensors as observations [9].

However, there are some flaws to this ontology, namely it still fails to address several aspects of the IoT data such as real-time data collection issues, dealing with the different standards for measurement units and exposing sensors to services. Therefore, other ontologies derived from this, have been proposed to overcome the flaws of SSN.

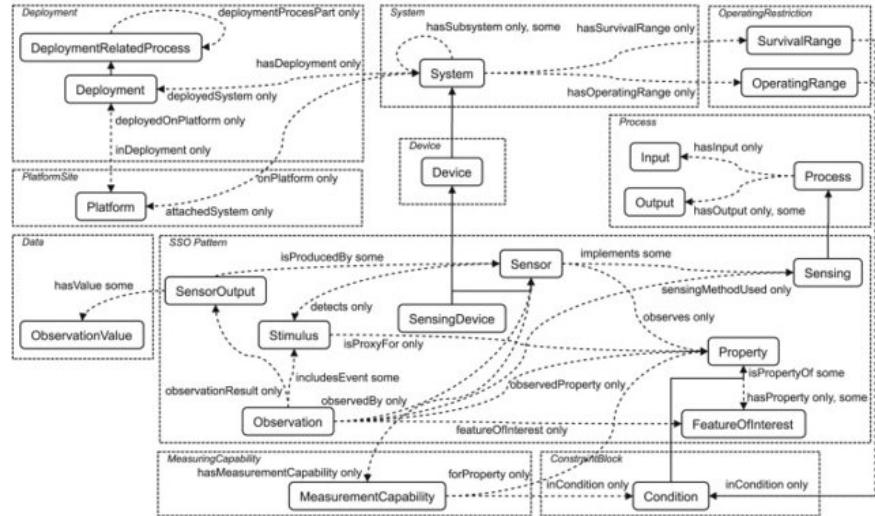


Figure 2.6: SSN Ontology

As observed in Figure 2.6 the SSN ontology can be seen from four main entities:

- Sensor: focusing on what senses, how it senses, and what is sensed;
- Observation: focusing on the observation data and related metadata;
- System: focusing on groups of sensors and their deployments;
- Feature and Property: focusing on particular properties or observations have were made about a property.

This ontology takes a generously comprehensive perspective on what a sensor is: *observes*; and permits such sensors to be portrayed at any degree of detail, for instance, permitting sensors

to not only be seen just as objects that assume a role of sensing, as well as allowing sensors to be described in terms of their components and method of operation.

2.5.2 OneM2m Base

OneM2M base is an ontology that aims to provide a high level ontology for the IoT market. Moreover, it has a minimal set of common knowledge that enable semantic interoperability. It is designed using minimal number of classes, properties, and restrictions [10]. An example of the oneM2M ontology is presented on Figure 2.7.

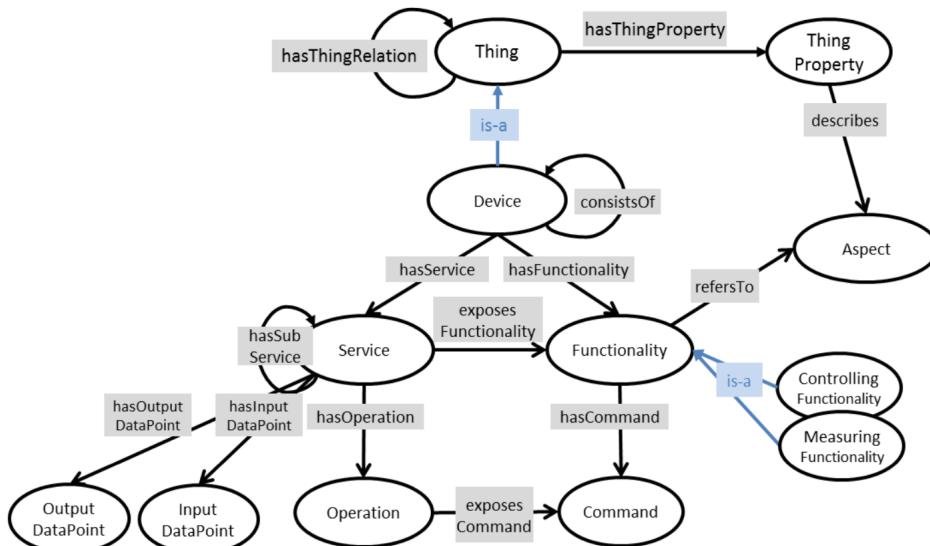


Figure 2.7: oneM2M Base Ontology

- **Thing**: a super class that refers to a device, platform, or service.
- **Device** refers to a device, which can have a particular task using its function.
- **Service** refers to the functions of a device in a communication network
- **Function** gives a meaning to a function of a device, for example, ‘turn on lights or turn off lights’
- **Operation** refers to a procedure occurring in a communication network, for example, ‘pass data to other device.’
- **Command** refers to the action that makes a function accomplish a particular task.’
- **Aspect** refers to metadata properties.

Several independent Open Source foundations and projects have been actively using oneM2M standards: ATIS Open Source Internet of Things (OS-IoT)³, OM2M⁴, etc.

³<https://os-iot.org>

⁴<https://www.eclipse.org/om2m/>

2.5.3 SAREF

SAREF (Smart Appliance REference) ontology exists in the domain of smart appliances and aims to reuse and align concepts and relationships in existing appliance-based ontologies. The concept of functions - one or more commands supported by devices (sensors) defined in SAREF - supports modularity and extensibility of the ontology, and also helps in maintenance of the appliances [7].

The starting point for SAREF is the concept of a Device (for example, a led). Devices are objects designed to perform one or more functions in different scenarios: homes, common public buildings or offices. Then, a Device provides a Service, that represents a function to a network.

2.6 QUDT ontology

Ontologies provide the ability to reference the formal semantics of a model (i.e. a concept). This in turn supports a level of interoperability and information exchange between systems and representations of domain information [25]. In this case, a unit ontology provides a formal way of specifying units explicitly, thereby avoiding tacit conventions that are prone to misinterpretation.

QUDT is an open source established to give semantic details to units of measure, quantity kind, measurements and information types. QUDT is an ontology, which advocates for the usage of norms to measure information communicated in RDF and JSON. A set of semantic web guidelines is distributed by the World Wide Web Consortium (W3C) to encourage trade between frameworks. The Resource Description Framework (RDF) gives the essential establishment. The RDF Schema (RDFS) and the Web Ontology Language (OWL) give a lot of classes and properties that permit different degrees of information portrayal, including Description Logic, to be expressed utilizing RDF.

A set-up of semantic web standards is distributed by the World Wide Web Consortium (W3C) to encourage exchange between frameworks. The Resource Description Framework (RDF) gives the essential establishment. RDF Schema (RDFS) and the Web Ontology Language (OWL) give a lot of classes and properties that permit different degrees of information portrayal, including Description Logic, to be communicated utilizing RDF.

QUDT is comprised of several linked ontologies. QUDT ontologies are sorted out as assortments of various kinds of diagrams, as recorded in the QUDT inventory. Jargon diagrams hold various spaces of amounts and units, which import the fitting QUDT schemas [3]. The center plan example of the QUDT philosophy appears in Figure 2.8.

2.7 SensorThings API Implementations

Regarding to SensorThings API, there are several implementations available as open source software. Among the most popular are the following listed.

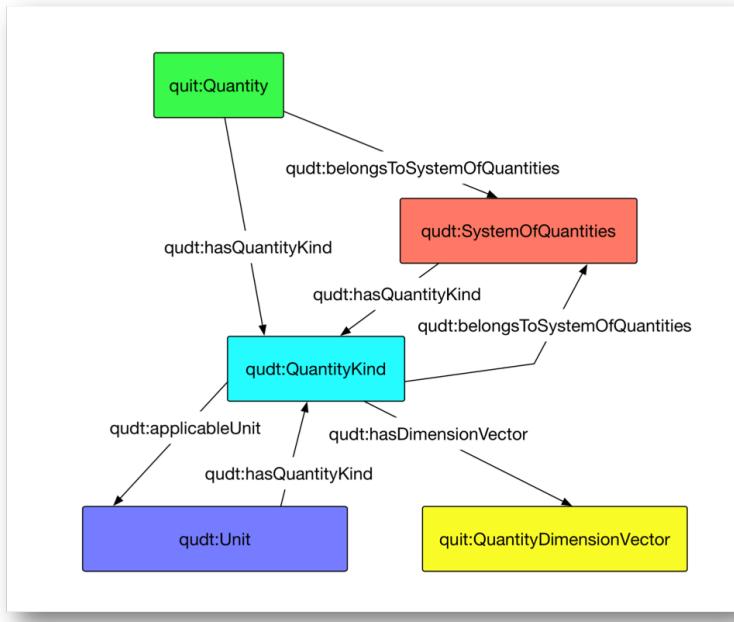


Figure 2.8: QUDT: basic structure

2.7.1 FROST-Server

FROST-Server (FRaunhofer Opensource SensorThings Server)⁵ developed by Fraunhofer IOSB, a German research institute, is an open source implementation of the SensorThings API standard [15]. This implementation addresses the need for an easy to use, standards-based data management and storage. It is written in the Java programming language. The application stores all data in an extended PostGreSQL database extend with PostGIS.

2.7.2 52North

52North⁶ provides an open source implementation containing a variety of interfaces. Therefore, it has the OGC Sensor Observation Service (SOS) interface, and several other complementary interfaces (for example, a non-standard REST API, specially optimized for the development of lightweight client applications).

2.7.3 SensorUp

SensorUp⁷, a Calgary-based startup, is considered a reference because it was the first to develop a compliant SensorThings implementation. This implementation is Java-based and the data is stored in a PostgreSQL database. In addition to server, SensorUp also provides documentation and support to make SensorThings even easier to use for client developers, with a sandbox container.

⁵<https://github.com/FraunhoferIOSB/FROST-Server>

⁶<https://52north.org>

⁷<https://sensorup.com>

2.7.4 GOST

GOST⁸ developed by Geodan, an Amsterdam based geo-ICT company, is an open source implementation based on the programming language GO. However, GOST is still considered as alpha software, meaning it is not yet considered adequate for customer use.

2.7.5 Mozilla

Mozilla has another open source implementation of SensorThings API standard and has almost passed all the OGC test suite tests. This implementation uses PostgreSQL for data storage. Unfortunately, Mozilla development has not been active since February 2017.

2.7.6 CGI Kinota Big Data

CGI⁹ developed a modular implementation of SensorThings called Kinota Big Data. Kinota is designed to support different persistence platforms ranging from RDBMS to NoSQL databases. The current implementation supports Apache Cassandra. Kinota implementation does not support all the features and has only a subset of the SensorThings API requirements.

⁸<https://www.gostserver.xyz>

⁹<https://github.com/kinota/kinota-bigdata>

Chapter 3

Problem Statement and Solution Proposal

The purpose of this chapter is to describe the problems that have been detected and how they are planned to be resolved. The solution itself will be described in detail, as well as the definition of the proposed architecture.

3.1 Problem

In this work, it is intended to build an application in order to store information from the sensors. Moreover, not only the observations values are stored, but also the metadata associated to these sensors needs to be stored. The system needs to provide semantics on top of the raw data, in order to obtain more information. This means, future applications should be able to access the data in a meaningful way, in order to make correlations even from different sensors. For example, in a smart house the air conditioner, is only activated when a temperature sensor is below or higher from a certain value.

The problem of this work can then be divided in two issues, namely *Choice and implementation of a meta data model* and *Autonomous registration*.

3.1.1 Choice and implementation of a meta data model

In the first step "Choice and implementation of a meta data model", a group of standard data models, described in Section 2.4, need to be analysed and the one best fit for the project will be chosen. These models will be compared against each other, with the criteria discussed previously.

After a data model is picked, the focus of the problem changes to choosing the best implementation for that choice. The choice has to have in count the information coming from the sensors. This information comes in the form of data events, which need to be processed and stored in the

database. A data event is an event coming from a broker, which has the value of the readings of a sensor and all the information which identifies this sensor and metadata.

3.1.2 Autonomous registration

The second problem "Autonomous registration" represents a mechanism to recognize sensor data without user input.

The capacity to support the annotation of observation values from various data sources with sufficient and exact metadata is critical to accomplish interoperability; that is, depict unequivocally the metadata the magnitude of a physical quantity indicated by a unit of measurement in a machine-meaningful organization. Thus, it is imperative that when new data arrives, the observation values are associated to an unique property, instead of creating a new one every time. Moreover, this allows to comparing data with different units of measurement, but referent to the same properties.

The challenge is to develop a consistent methodology that could produce a semantically rich way of registering these properties. When a new *thing*, *sensor* or *property* arrives in the form of an event and they are not in the database, this system should be able to autonomously register each of these entities. However, the system can not just create new entities, each time a new event arrives because it will not only erase the concept of interoperability but also create problems of performance. In this situation, there will be multiple entities for the same properties, and future applications won't be able to perform data correlation with this information.

In the event received, there is no property declared but only the observation value, the unit of measurement of this observation and the name of the property of that specific sensor which varies for the same properties. For example, event A may have:

```
"name": "Sensor 1 Temp",
"unit": "Kelvin (K)",
"symbol": "K",
```

and event B have:

```
"name": "Sensor Temperature",
"unit": "Celsius (°C)",
"symbol": "°C",
```

However, both of them are referring to the same property, which is Temperature, but the system needs to find out that.

3.2 Solution Proposal

In order to solve the problems stated, it was created an application in NestJS¹ to connect with SensorThings API. Nest was developed by Kamil Myśliwiec and was designed as a framework to

¹<https://nestjs.com>

rapidly build server-side applications. Myśliwiec was heavily inspired by Angular and it shows in the structure of a Nest application.

This framework already facilitates further steps by defining the app architecture and providing a standardized set of guidelines by defining an opinionated architecture. This architecture is composed by a root module available, which will be used to configure database providers, to define the controller, to add middleware, to add the pipe and guards, and to provide services.

Moreover, NestJS is a really well designed and even better-crafted framework. Nest abstracts implementation details so that the same components can run across HTTP-based platforms, Web-Sockets, and Microservices. This allows to the easy implementation of the MQTT protocol and the other parts of this work.

3.2.1 System Architecture

The planned approach for this application can be summarized in the architecture in Figure 3.1. The structure built is composed by several distinct components that interact with each other. There are two components independent from the application, the **Domatica Broker**, which is responsible for sending the events via MQTT and the **SensorThings API**, which contains as a database for the data.

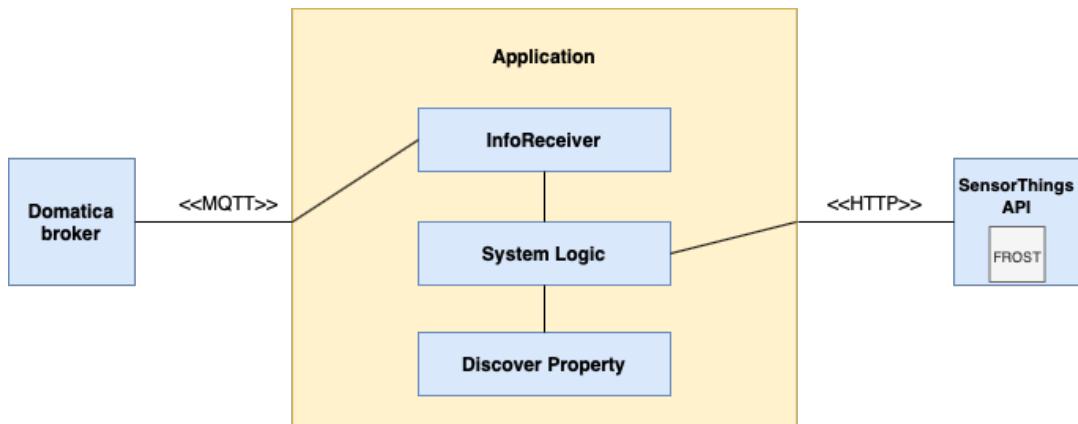


Figure 3.1: System architecture

However, it is still important to observe how the information coming from the broker is managed in the application. Therefore, Figure 3.2, provides a sequence diagram an event. The three components which shall be built in this application, as shown in Figure 3.2 are:

- **InfoReceiver**
- **System Logic**
- **Discover Property**

Accordingly, the breakdown of the event in the 3 components is presented in detail, following the sequence diagram.

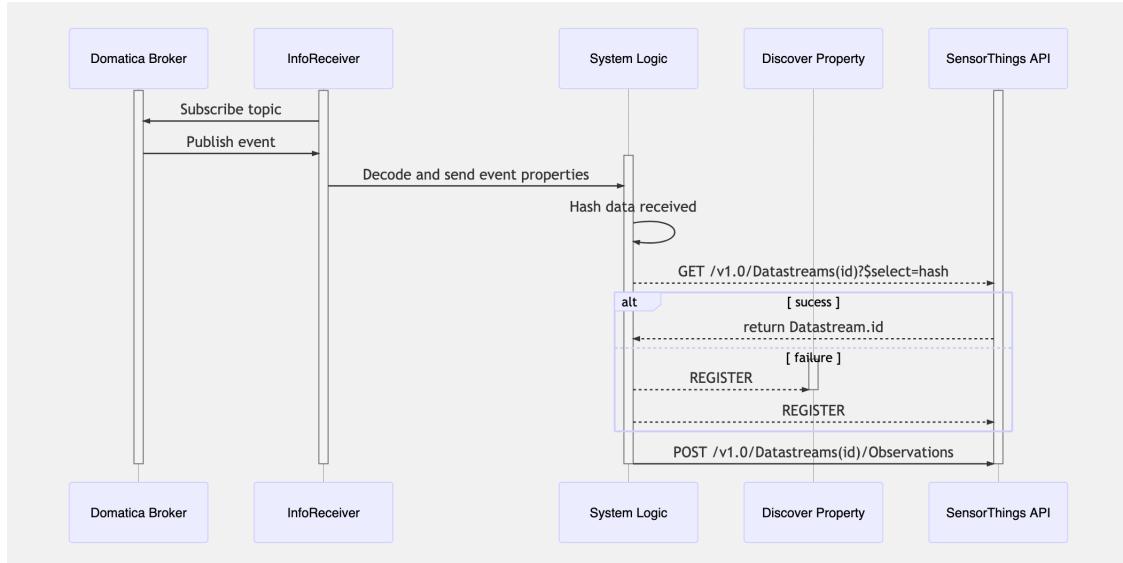


Figure 3.2: Sequence diagram

3.2.2 InfoReceiver

This component is responsible for listening the events coming from the *Domatica Broker*. Therefore, the *InfoReceiver* component will subscribe to the topic in which the events are coming. When a new event arrives, it will contain several properties. These properties will be mapped and the information is passed to the next component *System Logic*.

These events will be sent by the Domatica broker, via MQQT connection. The application shall create a MQQT client, which will connect to the broker and receive the events. When an event arrives, it is necessary to do a process of data cleaning and catalog the properties received against the entities presented in the database.

An example of an event received is shown below:

```

"gateway": "23000A7E",
"device": "DEV0000004B",
"name": "INVERSOR 1 - CURRENT L2",
"interface": "iSTDInstMeter",
"unit": "Ampere (A)",
"symbol": "A",
"decimals": "4",
"unitMember": "instUnitID",
"\n": "\n",
"value": 5

```

And the breakdown of the important properties are the following:

- *gateway* refers to the gateway
- *device* refers to the device

- *name* refers the name of the property sent, this name can be different for same properties
- *interface* refers the interface
- *unit* refers the unit of measurement of the observation sent
- *symbol* refers the symbol of the unit of measurement
- *decimals* refers to the number of decimals cases the value of the observation contains
- *value* refers to value of the observation

Thus, each of these entities shall be matched against the database. This can be a simple process in the case of the entities *thing* and *sensor*, but is more complicated in the entity *property*.

3.2.3 System Logic

This component handles the management of the events. However, the system needs to know if the event is already previously registered in the database or is a new event. Therefore, the name of the thing, name of the sensor, name of the property and the unit of measurement will be combined to form and unique identifier to an event, which is named *hashKey*.

Furthermore, a request will be made to the SensorThings API server, to see if there is any datastream with an identifier equal to this event. If this is true, it means all the entities related to the event were previously registered and it is now possible to store the value of the reading sent. The reading will be associated with this datastream and another request with the value can be sent to the server.

Otherwise, some or all of the entities (Thing, Sensor, ObservedProperty) need to be registered. To achieve this, three individual requests are sent to the server, in order to find the entities that are not registered yet, as seen in Figure 3.3.

If the entity is registered, the request will return the identifier of the entity; otherwise a register method will be called. The registration of the entities of a thing and sensor will be similar. Both of them can be registered with POST HTTP requests to the SensorThings API.

The registration of an ObservedProperty is more complicated than the other two entities, because it is possible a property with the same characteristics is already stored in the database, but not associated with the one in the event. Therefore, this step occurs in the Discover Property component.

3.2.4 Discover Property

All the datastreams currently stored in the database will be analysed. In a first level of analysis, the parameter of the symbol from the event will be searched in a unit ontology. In the second level, it was used a string matching algorithm to query the ontology, by comparing the properties in the event. A deep dive in this subject is made on Chapter 5.

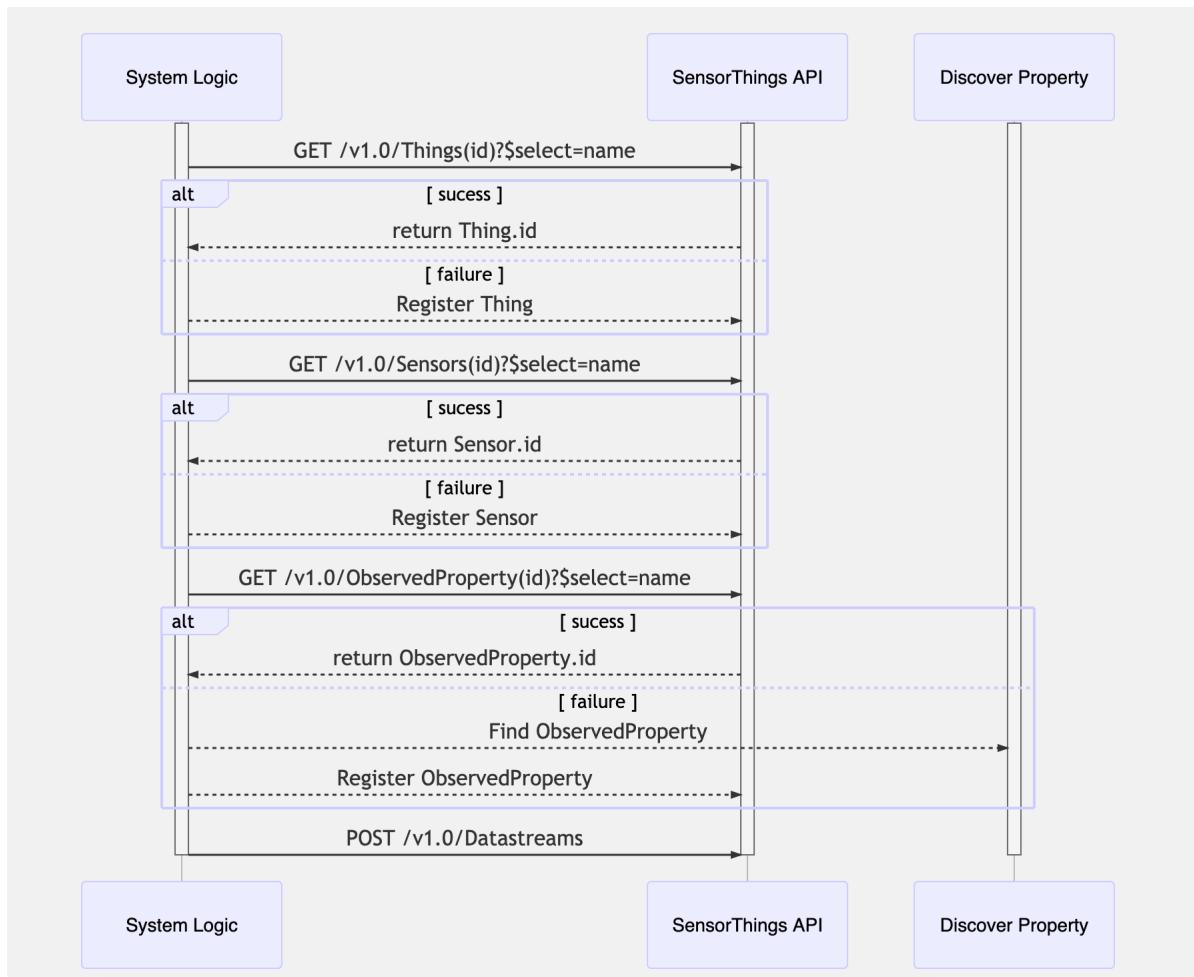


Figure 3.3: Finding entities not registered

3.3 Conclusions

This chapter focused on outlining the problem in question and the approach that was taken in order to resolve it. Furthermore, it divided this problem into different issues, which are going to be analysed more deeply in the following chapters.

The choice of a meta data model continues to be explained in Chapter 4, namely how it was implemented and the features it provides. In Chapter 5, the autonomous registration topic and the solution provided will be explored. Here, the algorithms used will also be explained, how they interact with the system and some use case examples.

Chapter 4

Choice and implementation of a meta data model

This chapter will describe the implementation details of the SensorThings API and the features it provides to the application built. Also, some examples of these features being used are also shown below.

4.1 Meta Model Choice

For this project, it is intended to have a meta data model, to store the information related to the IoT paradigm. Eclipse Vorto Project provides different models for each device and the Fiware Initiative also separates its models between categories. These alternatives might not be the best option, due to the necessity of having a more abstract model.

One of the first ontologies created is the SSN ontology. This ontology still fails to address several aspects of the IoT data such as real-time data collection issues, dealing with the different standards for measurement units and exposing sensors to services, which is one of the main focus of the project.

Another options are OneM2m and SAREF, which meet the criteria of being a simplicity model, with minimal number of structure types, composition rules, and attributes and elegance, being as simple as possible for a given direct modeling capability. However these data models are not well suited to support storage of data generated through sensor activity. Thus, it fails in the criteria of provision of schemas, which means a model should include structure schemas to permit data definition.

Finally, the SensorThings API is a data model standard, which allows the storage of the data from the sensors. It also passes the simplicity and elegance criteria, being a model easy to understand. Moreover, one of the benefits of this option is the documentation and support presented

Data models	Simplicity	Elegance	Applicability of safe implementation techniques	Provision of schemas
OneM2m	High	High	Medium	Low
SAREF	High	High	Medium	Low
SensorThings API	Medium	Medium	High	High

Table 4.1: Comparison of data models

online, which allow to an easy implementation and integration and meeting the criteria of applicability of safe implementation techniques. modeling uniqueness, simplicity, elegance.

4.2 Sensor Things API implementations

As explained before, the SensorThings API is an open standard, built on Web protocols and the OGC Sensor Web Enablement standards, and applies an easy-to-use REST-like style. There are several implementations of this standard, but the one chosen in this work was the FROST-Server. The criteria in choosing an implementation option of the six options, exposed in Chapter 2, was the available documentation and forums provided. Therefore, the options were reduced to FROST-Server and SensorUp, because the others either lack documentation or the projects were not longer being continued. However, SensorUp wasn't a standalone application and FROST-Server was chosen.

Looking, at the data model of the FROST-Server implementation in Figure 4.1, it is possible to observe some differences to the original one. In this model, most of the entities contain an attribute properties, which is a JSON object. This is a great advantage to the project, because it allows to store the identifier *hashKey* in the Datastream entity, as explained in Chapter 3.

4.3 Structure

Firstly, it is necessary to understand how the data will be sent and organized within the SensorThings API data model. A simple structure of this server can be shown in Figure 4.2.

In this structure, the application created will play the role of the client and will communicate with RESTful HTTP methods with the server, in order to make requests and get the responses in JSON format. The SensorThings API module will then be responsible of the management of the data sent, which will be stored in the entities described.

The five SensorThings key components (Things, Datatreams, ObservedProperties, Sensors, Observations) are interrelated, with the core entity linked with the rest being the Datastreams. There is also two more components (Locations and HistoricalLocations), which can store metada

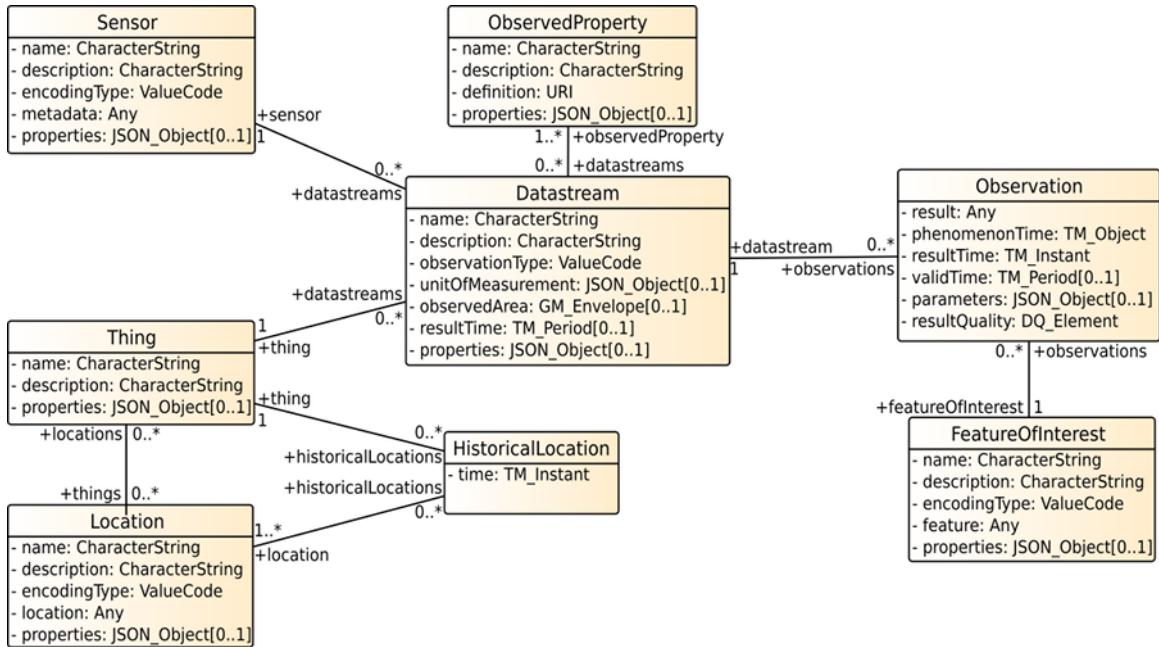


Figure 4.1: Frost relational model

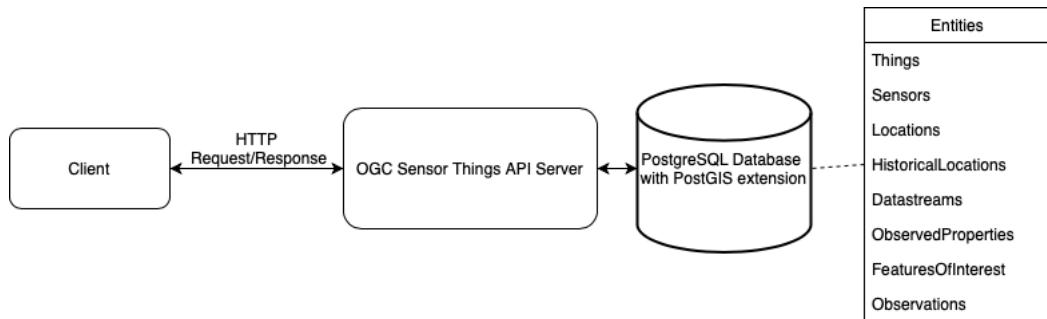


Figure 4.2: Server structure

data for the locations of Things. In these entities of the database, only primary and foreign keys have indexes on them.

4.3.1 Endpoints

Each entity in a SensorThings API has its unique ID and can be referred to by its unique URL to create other entities, update their details and properties and also delete them. This makes data management and system maintenance highly flexible and efficient [18].

Also, the data from the API can be easily viewed using a normal Web Browser. One can simply navigate from one object to the next by clicking the URLs provided within the data. An example of the view of the Datastream entities is provided in Figure 4.3, if the user went to the url:

<http://FROST-Server/v1.1/Datastreams>

Therefore, the user, in order to manage these entities, must be aware of these endpoint addresses. After entering the endpoint address there will be issued a query in which it will retrieve all the data referring to the ID exposed in the query.

4.3.2 Register and give contextual information

One of the most frequent queries is to save the value from an observation of a sensor. However, in SensorThings API there is a decentralized architecture, which can make retrieve relevant information easier than otherwise would not even be available. This type of architecture also creates the need to have the values of observations associated with a Datastream, in order to access this meta data.

In case a Datastream was not previously associated to an observation, it must be registered. As stated before, the Datastream entity is also linked to 3 more entities (Things, Sensors, ObservedProperties) and they also must be registered. In Figure 4.3, there is an example of a registration of a datastream.

Entity	Refers to	Example data	Request URL
Thing		<pre>"name": "Temperature System", "description": " Monitoring area temperature",</pre>	http://localhost:8080/FROST-Server/v1.1/Things
Sensor		<pre>"name": "DHT22", "description": "DHT22 temperature sensor", "metadata": "https://cdn-shop.adafruit.com/datasheets/DHT22.pdf", "encodingType": "application/pdf",</pre>	http://localhost:8080/FROST-Server/v1.1/Sensors
ObservedProperty		<pre>"name": "Area Temperature", "description": "The degree or intensity of heat present in the area", "definition": "http://www.qudt.org/qudt/owl/1.0.0/quantity/instances.html#AreaTemperature",</pre>	http://localhost:8080/FROST-Server/v1.1/ObservedProperties
Datastream	Unique Thing, ObservedProperty and Sensor.	<pre>"name": "Thermometer readings", "description": "Thermometer readings from DHT22", "observationType": "observationType": "http://www.opengis.net/def/observationType/OGC-OM/2.0/OM_Measurement", "unitOfMeasurement": { "name": "degree Celsius", "symbol": "degC", "definition": "http://www.qudt.org/qudt/owl/1.0.0/unit/instances.html#DegreeCelsius", }, "properties": { "hashKey": "123456", },</pre>	http://localhost:8080/FROST-Server/v1.1/Datastreams

Figure 4.3: Register datastream

In order to add a new entry referring to a known Thing an HTTP POST request must be issued to the target URL. The example JSON object creates a Thing with its mandatory properties (name,

description) and an additional field with properties if needed. For the other two entities (Sensors and ObservedProperties), the same must be made.

When all the new entities are created, they are automatically assigned a unique ID. If an entity is removed its ID remains stored in the system and cannot be used again. Furthermore, some entities can be extended with their related entities as optional properties when creating them. Therefore, when finally sending a request to register the new Datastream, the ID's of the entities related will also be in the body of the request.

4.3.3 Upload readings

After a datastream is associated to the observation in the server, it is possible to upload the value of the reading. Accordingly, an HTTP POST request will be sent and the ID of that datastream will also be in the body of the request. An example of this, is represented in Figure 4.4.

Entity	Refers to	Example data	Request URL
Observation	Unique Datastream	"result":"50"	http://localhost:8080/FROST-Ser ver/v1.1/Observations

Figure 4.4: Upload observation

4.3.4 Filtering information

In many use cases, the response should only contain objects that pertain to some specific criteria. The “filter” request parameter makes it possible to filter the data to be returned based on the values of specific attributes of the requested data.

Thus, one could request all values above a certain threshold or measured between two specific points in time. The following request returns all Observations where the result value is greater than 5:

```
/FROST-SERVER/v1.1/Observations?$filter=result gt 5
```

Moreover, there are other filtering options such as (not equal, greater than, lesser than...) which can also be used. Common logical operators like AND, OR, NOT are also available.

4.4 Conclusions

The functionalities described in this chapter focused mainly on the interaction of the application with the server, presenting examples in how to operate with the SensorThings API. Moreover, it was shown an overview of how the entities are stored in the SensorThings API database and how they reference each other. In addition, the important features used in this work, were stated and explained.

Chapter 5

Autonomous Registration

This chapter describes the mechanism, which provides interoperability for the user, namely in the property of the sensors. First, an overview of the QUDT ontology is provided and later the system architecture and the algorithms used are explained.

5.1 QUDT

In this work, it will be used two schemas from the QUDT ontology, namely qudt:Unit and qudt:QuantityKind. QUDT contains 1537 units of measure implemented as instances of qudt:Unit, and 846 implemented as instances of qudt:QuantityKind or its specializations. These schemas are provided by two files, which are in Turtle format. A Turtle document allows writing down an RDF graph in a compact textual form. It consists of a sequence of directives, triple-generating statements or blank lines. Simple triples are a sequence of (subject, predicate, object) terms, separated by whitespace and terminated by '.' after each triple [4].

5.1.1 Unit

A unit of measurement is a defined magnitude of a quantity, defined and adopted by convention or by law, used as a standard for measuring the same kind of quantity.

QUDT recognizes variations of a given unit. For instance, "day", spoke to as a unit of estimation, can allude to a normal sun based day, a sidereal day or a period identical to precisely 86,400 seconds. Every one of these alternatives to "day" shows up as a particular unit in cosmology.

In addition, QUDT recognizes units of various sorts that are usually alluded to with a same name. For instance, "second" can allude to a proportion of time or an edge estimation. Once more, each utilization shows up as a particular term in the cosmology, giving interoperability to the system.

The vocabulary graph of units, currently contains 1537 instances and it is available as a Turtle file. Each instance contains several parameters as described in the following example:



Figure 5.1: QUDT: Unit

In this work, the properties symbol and label will be more relevant. These properties will be compared against the ones received by the application and be used to match the unit of measurement of an event to one in the ontology.

5.1.2 Quantity Kind

A QuantityKind characterizes the physical nature or type of a measured quantity. (E.g. length, mass, time, force, power, energy, etc.). Then, derived QuantityKinds are defined in terms of a small set known as Base Quantity Kinds using physical laws.

In the QUDT vocabulary there are currently 846 instances of QuantityKind entities. An example of an instance can be shown below:

In order to populate the ObservedProperty Entity of the SensorThings API, it will be used the following match:

- name - qudt:label
- description - qudt:plainTextDescription
- definition - qudt:type

5.2 Architecture

The *Discover Property* component is responsible for the task of registering and labeling the ObservedProperty of the data from the sensors.

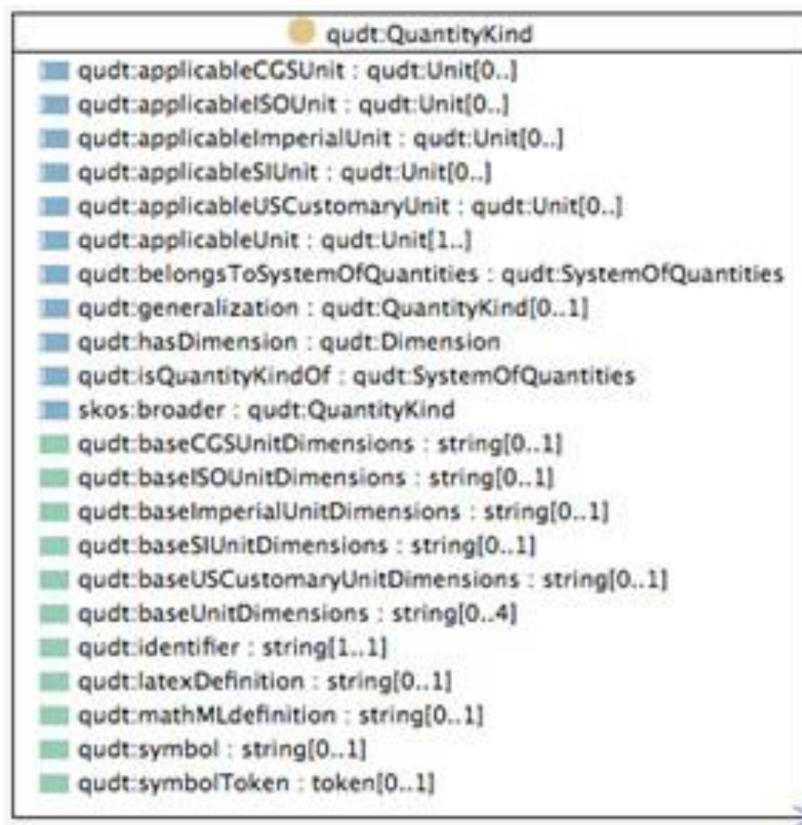


Figure 5.2: QUDT: QuantityKind

As stated before, two documents with the Unit and QuantityKind schemas are imported. This component parses these files and queries them, in order to obtain the ObservedProperty associated to an unit of measurement.

In Figure 5.3, there is a flowchart of how this mechanism works.

Firstly, the system will receive an unit of measurement(UoM) as an input, coming from an event from a sensor. An example of an UoM could be:

```
"name": "INVERSOR 1 - CURRENT L1"
"unit": "Ampere (A),
"symbol": "A",
```

Then, the symbol of the UoM, in this case it is 'A' will be compared against the property `qudt:symbol` from all the instances of the Unit schema document. If there is a match this means both the strings from the UoM and the an instance from the ontology are equal. Then, taking advantage of the QUDT model, this instance will refer to a `QuantityKind` an it is possible to access with the property `qudt:quantitykind`.

In case there is no direct match, the name of the UoM will be compared against the property `qudt:label` from all the instances. This comparison will be made using string algorithms. These algorithms explained in the following section, will give a score between 0-1, referring to the similarity between the two strings. If the value of this score is greater than the value of the threshold



Figure 5.3: Discover Property Flowchart

previously defined, then the quantity associated is added to the results list. When all the instances were compared, the quantitykind with the highest value score from the results list is selected.

Finally, it is possible to populate the ObservedProperty and this component will return this JSON. If there is no ObservedProperty found, the user will be notified and the component will return a null value. Later, the user will have to manually enter in the server the correct values for this event.

5.3 String matching algorithms

The problem of string matching is a recurrent problem in computer science, with applications to spell checking, spam filtering, computational biology, etc [11].

The issue comprises in discovering strings that match a pattern approximately(as opposed to precisely). The closeness of a match is estimated as far as the primitive operations important to change over the string into a definite match [21]. This number is known as the edit distance

between the string and the pattern. The usual primitive operations are: insertion, deletion, substitution.

Moreover, these algorithms can be classified in three different domains, based on the properties of operations.

- **Edit distance based:** Algorithms falling under this category try to compute the number of operations needed to transforms one string to another. More the number of operations, less is the similarity between the two strings.
- **Token-based:** Here, the normal information is a set of tokens, as opposed to complete strings. The thought is to locate the comparative tokens in the two sets.
- **Sequence-based:** In this category, The algorithms attempt to locate the longest grouping which is available in the two strings, the a greater amount of these successions found, the higher is the similitude score.

In this work, the objective is to compare individual groups of strings, which are very small and consist of one or two words. The Token-based and Sequence-based domain are more appropriated for longer texts. Thus, the algorithms used needed to belong in the Edit distance based domain. Moreover, three algorithms were used and compared against each other: Hamming Distance, Levenshtein distance and Jaro-Winkler.

5.3.1 Hamming distance

The Hamming distance between two strings of equivalent length is the quantity of positions at which the comparing symbols are different. As such, this distance is figured by overlaying one string over another and finding the spots where the strings differ. In a progressively broad setting, the Hamming distance is one of a few string measurements for estimating the alter separation between two arrangements. It is named after the American mathematician Richard Hamming.

For example, the Hamming distance between:

- "Text" and "Test" is 1, only change happens in the third character
- "Arow"and "Arrow" is 3, even though the string is only missing one 'r', the 'row' part is offset by 1, making the edit distance 3.

In order to adapt the algorithm to this work, some changes were made. Firstly, both of the strings being compared, were changed to upper case, to improve the accuracy. Second, the original algorithm had some problems, namely: the algorithm measures the distance, while it is wanted the similarity; both of the strings needed to be same length; and the final score is not provided in a scale of 0-1.

```

1 hammingDistanceNormalized(str1, str2) {
2     // String to upper case.

```

```

3      let s1 = str1.toUpperCase(),
4          s2 = str2.toUpperCase();
5      // String lengths.
6      const s1Length = s1.length,
7          s2Length = s2.length;
8      // Absolute distance between 's1' & 's2'.
9      let distance;
10     // Helper variables.
11     let i, imax;
12     let dividend;
13     // Initialize stuff and may have to swap 's1'/'s2'.
14     if ( s1Length < s2Length ) {
15         // Swap 's1' and 's2' values.
16         s1 = [ s2, s1 = s1 ][ 0 ];
17         // Initialize distance as the difference in lengths of 's1' & 's2'.
18         distance = s2Length - s1Length;
19         imax = s1Length;
20         // Initialize dividend by the larger string length.
21         dividend = s2Length;
22     } else {
23         // No need to swap, but do initialize stuff.
24         distance = s1Length - s2Length;
25         imax = s2Length;
26         dividend = s1Length;
27     }
28     // Compute distance.
29     for ( i = 0; i < imax; i += 1 ) {
30         if ( s1[ i ] !== s2[ i ] ) distance += 1;
31     }
32     // Normalize the distance & return.
33     return 1-(( dividend ) ? ( distance / dividend ) : 0);
34 }
```

Listing 5.1: Hamming Algorithm

To fix the issues mentioned, a normalization of the algorithm occurred. In case strings are of different sizes, the result is divided by the larger string length. The result is then subtracted to 1, in order to obtain the similarity and not the distance. Therefore, the results to the examples used before are:

- "Text" and "Test" is 0.75.
- "Arow"and "Arrow" is 0.4.

5.3.2 Levenshtein distance

Levenshtein distance is a measure of the closeness between two strings, the source string (s) and the objective string (t). Levenshtein distance is named after the Russian researcher Vladimir Lev-

enshtein, who formulated the calculation in 1965. The distance is the number of deletions, insertions, or substitutions required to change s into t. For example:

- "Text" and "Test" is 1, because one substitution (change "x" to "s") is sufficient to transform s into t.
- "Arow" and "Arrow" is 0.4, because one insertion (add "r") is sufficient to transform s into t.

```

1 levenshteinDistanceNormalized(str1, str2) {
2     str1 = str1.toUpperCase();
3     str2 = str2.toUpperCase();
4     if(str1.length == 0) return str2.length;
5     if(str2.length == 0) return str1.length;
6
7     const matrix = [];
8
9     // increment along the first column of each row
10    let i;
11    for(i = 0; i <= str2.length; i++) {
12        matrix[i] = [i];
13    }
14
15    // increment each column in the first row
16    let j;
17    for(j = 0; j <= str1.length; j++) {
18        matrix[0][j] = j;
19    }
20
21    // Fill in the rest of the matrix
22    for(i = 1; i <= str2.length; i++) {
23        for(j = 1; j <= str1.length; j++) {
24            if(str2.charAt(i-1) == str1.charAt(j-1)) {
25                matrix[i][j] = matrix[i-1][j-1];
26            } else {
27                matrix[i][j] = Math.min(matrix[i-1][j-1] + 1, // substitution
28                                         Math.min(matrix[i][j-1] + 1, // insertion
29                                                 matrix[i-1][j] + 1)); // deletion
30            }
31        }
32    }
33    return 1-(matrix[str2.length][str1.length]/Math.max(str1.length,str2.length));
34 }
```

Listing 5.2: Levenshtein Algorithm

As the Hamming distance, this algorithm also suffers some changes to adapt to the work. Accordingly, both strings are changed to upper case, improving accuracy. Then, a normalization also occurs to cater different size strings and the result of this distance is subtracted to 1, to obtain the value of similarity. This results in:

- "Text" and "Test" is 0.8.
- "Arow"and "Arrow" is 0.8.

5.3.3 Jaro-Winkler

The Jaro–Winkler distance is a string metric measuring an edit distance between two sequences. It is a variation proposed in 1990 by William E. Winkler of the Jaro separation metric. In other words,, this calculations gives high scores to two strings if, (1) they contain same characters, but within a certain distance from one another, and (2) the order of the matching characters is same.

The Jaro similarity formula between two strings is:

$$sim = \frac{1}{3} \left(\frac{m}{s1} + \frac{m}{s2} + \frac{m-t}{m} \right) \quad (5.1)$$

Where,

- **m** s the number of matching characters (characters that appear in s1 and in s2)
- **s1** is the length of the first string
- **s2** is the length of the second string
- **t** is half the number of transpositions (compare the i-th character of s1 and the i-th character of s2 divided by 2)

The only modification this algorithm suffered was the change of both strings to upper case.

```

1  jarowinkler(str1, str2){
2      let m = 0;
3      str1 = str1.toUpperCase();
4      str2 = str2.toUpperCase();
5      // Exit early if either are empty.
6      if(str1.length === 0 || str2.length === 0 ) {
7          return 0;
8      }
9      // Exit early if they're an exact match.
10     if (str1 === str2 ){
11         return 1;
12     }
13     const range = (Math.floor(Math.max(str1.length, str2.length) / 2)) - 1,
14     s1Matches = new Array(str1.length),
15     s2Matches = new Array(str2.length),
16     transpositions = 0;
```

```

16     s2Matches = new Array(str2.length);
17     for (let i = 0; i < str1.length; i++ ){
18         const low = (i >= range) ? i - range : 0,
19         high = (i + range <= str2.length) ? (i + range) : (str2.length - 1);
20         for (let j = low; j <= high; j++ ) {
21             if ( s1Matches[i] !== true && s2Matches[j] !== true && str1[i] ===
22                 str2[j] ) {
23                 ++m;
24                 s1Matches[i] = s2Matches[j] = true;
25                 break;
26             }
27         }
28         // Exit early if no matches were found.
29         if ( m === 0 ){
30             return 0;
31         }
32         // Count the transpositions.
33         let k = 0;
34         let ntrans = 0;
35         for (let i = 0; i < str1.length; i++ ){
36             let j;
37             if ( s1Matches[i] === true ){
38                 for (j = k; j < str2.length; j++ ){
39                     if ( s2Matches[j] === true ){
40                         k = j + 1;
41                         break;
42                     }
43                 }
44                 if ( str1[i] !== str2[j] ){
45                     ++ntrans;
46                 }
47             }
48         }
49         let weight = (m / str1.length + m / str2.length + (m - (ntrans / 2)) /
50             m) / 3,
51         l = 0;
52         const p = 0.1;
53         if (weight > 0.7){
54             while (str1[l] === str2[l] && l < 4 ){
55                 ++l;
56             }
57             weight = weight + l * p * (1 - weight);
58         }
59         return weight;
}

```

Listing 5.3: Jaro-Winkler Algorithm

If this algorithm is used in the same examples, the results are:

- "Text" and "Test" is 0.95.
- "Arrow" and "Arrow" is 0.88.

5.4 Conclusions

This chapter gave a detailed explanation of how the process of registering and finding a property occurs. The objective of this chapter was to propose and implement algorithms to solve the registration of these properties. This goal was completed and in the diagram shown there is the solution implemented about how the information coming from the sensors is handled and the return property of this component.

Another key aspect is the different string matching algorithms utilized. There were three algorithms implemented, but the accuracy of them still needs to be tested with real data, in order to provide the best algorithm to this work. Moreover, a value of threshold for the algorithms implemented still remains undefined. This parameter will be numerically defined in the next chapter.

Chapter 6

Evaluation

This chapter analysis the solution implemented in order to evaluate the goals initially established, by conducting some experiments and analysing the results.

6.1 Experiment

For the correct validation of the proposed solution, an experiment was carried out focusing on the evaluation of the handling of receiving events from a dataset from Domatica. The experiment relates to the Discover Property component, which means it will test how well the system can identify new properties, coming from new sensors. Here, all three algorithms will be compared against each other, and a correct value of a threshold can be chosen. After all the experiments are completed, a discussion of the results obtained is presented.

The experiment, done to evaluate the system built, consists in registering every entry of the dataset in the SensorThings API server.

To achieve this, an application was built to simulate the Domatica Framework. It was also created a Mosquitto Broker, to simulate the Domatica Broker. Mosquitto is a lightweight open source message broker that Implements MQTT and is very easy to implement. This application client will parse the CSV file with the dataset. For every entry, the application will create a request and send it to the Mosquitto Broker, where the original application will be listening.

The objective is to test if the application can correctly store every event coming from the dataset, without user input. By analysing the results of the experiment it is possible to check if all entities were stored in the SensorThings API database, even the properties the system could not recognize. As the experiment, sends multiple HTTP requests to the SensorThings API, if all properties from the dataset are stored, it validates the web application.

Moreover, it was tested which entries were stored using only the unit symbol and which ones needed to used the string algorithms. In this last case, the accuracy of the three algorithms is

evaluated, to later decide the best one to use. An entry is incorrectly stored, if the system can not find a property associated with his unit of measurement or it associated to an incorrect property.

6.2 Dataset Description

The dataset which will be used to do the experiments, was provided by the Domatica company and contains data about real devices and sensors, they are currently using. This dataset contains 692 entries and 9 attributes/features:

- **gateway** - id of the gateway
- **device** - id of the device
- **name** - name of the property of the sensor
- **interface** - type of interface
- **unit** - name of the unit of measurement
- **symbol** - symbol of the unit of measurement
- **decimals** - number of decimal cases in the observation value
- **unitMember** - type of unit
- **\n** - this feature is not relevant

In Figure 6.1, there is a sample of the first 5 entries of this dataset.

	gateway	device	name	interface	unit	symbol	decimals	unitMember	\n
0	23000A7E	DEV0000004A	INVERSOR 1 - CURRENT L1	iSTDInstMeter	Ampere (A)	A	4	instUnitID	\N
1	23000A7E	DEV0000004B	INVERSOR 1 - CURRENT L2	iSTDInstMeter	Ampere (A)	A	4	instUnitID	\N
2	23000A7E	DEV0000004C	INVERSOR 1 - CURRENT L3	iSTDInstMeter	Ampere (A)	A	4	instUnitID	\N
3	23000A7E	DEV0000004D	INVERSOR 1 - CURRENT TOTAL AVERAGE	iSTDInstMeter	Ampere (A)	A	4	instUnitID	\N
4	23000A7E	DEV0000004E	INVERSOR 1 - VOLTAGE L1-L2	iSTDInstMeter	Volts (V)	V	1	instUnitID	\N

Figure 6.1: Dataset head

An interesting analysis to be made in this dataset refers to the unit attribute, because is this feature that will affect the Discover Property component. With the bar graph in Figure 6.2, it is possible to conclude there are 35 unique units, and the most frequent is the *State Unit* unit with 84 entries.

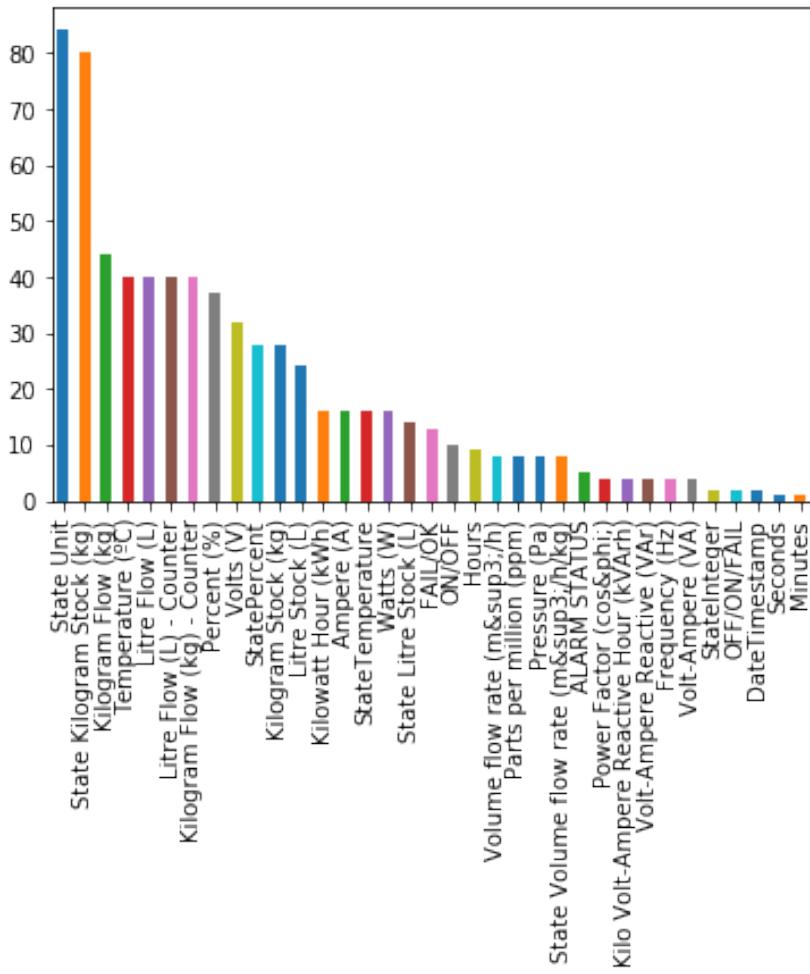


Figure 6.2: Unit Graph Bar

6.3 Performance Metrics

The first test, which is related to the web application, is evaluated by measuring the number of errors, it occurs in registering every entry of the dataset.

In the second part it is measured the performance of the three algorithms. Therefore, this problem was modeled as a binary classification problem. Thus, the algorithms will classify each string comparison and if the score of the comparison is greater than the threshold, the classification is a match. Otherwise, the classification is no match.

6.3.1 Confusion Matrix

To measure the performance for this problem, a confusion matrix will also be used. Each column of the matrix represents the instances in a predicted class while each row represents the instances in an actual class (or vice versa). Figure 6.3 represents the structure of a confusion matrix.

- **True Positive (TP)** = The algorithm predict a unit was a match and it was a correct.

		PREDICTIVE VALUES	
		POSITIVE (1)	NEGATIVE (0)
ACTUAL VALUES	POSITIVE (1)	TP	FN
	NEGATIVE (0)	FP	TN

Figure 6.3: Confusion matrix

- **True Negative (TN)** = The algorithm predict a unit was not a match it was correct
- **False Positive (FP)** = The algorithm predicted a unit was a match and it was not a match.
- **False Negative (FN)** = The algorithm predicted a unit was not a match and it was a match.

Once the confusion matrix is made, some metrics can be applied:

- **Accuracy** = $(1 - \text{Error}) = (\text{TP} + \text{TN})/(\text{PP} + \text{NP}) = \text{Pr}(C)$, the probability of a correct classification.
- **TPR/Recall/Sensitivity** = $\text{TP}/(\text{TP} + \text{FN}) = \text{TP}/\text{PP}$ = the ability of the classifier to detect a match in a group of correct strings.
- **Specificity** = $\text{TN}/(\text{TN} + \text{FP}) = \text{TN} / \text{NP}$ = the ability of the test to correctly rule out a match in group of wrong strings.
- **Precision** = $\text{TP}/(\text{TP} + \text{FP})$ = how precise the model is out of those predicted a match, how many of them are actual a match.
- **F1 score** = $2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$ = is a measure of the accuracy and considers both the precision and the recall of the test to compute the score.

6.3.2 ROC Curve

A Receiver Operating Characteristic (ROC) curve is a performance measurement for classification problem at various thresholds settings. This curve is equal to the plot of TP (sensitivity) against FP (1 – specificity) for each threshold used. ROC curves were developed for use in signal detection in radar returns in the 1950's, and have since been applied to a wide range of problems [1].

For an ideal classifier the ROC curve will go straight up the Y axis and afterward along the X axis. A classifier with no power will sit on the diagonal, meaning this classifier is equivalent to random. An intermediate classifier has the curve similar to the curve *Intermediate* in the Figure 6.4.

A ROC curve can be utilized to choose a threshold for a classifier which boosts the true positives, while limiting the false positives.

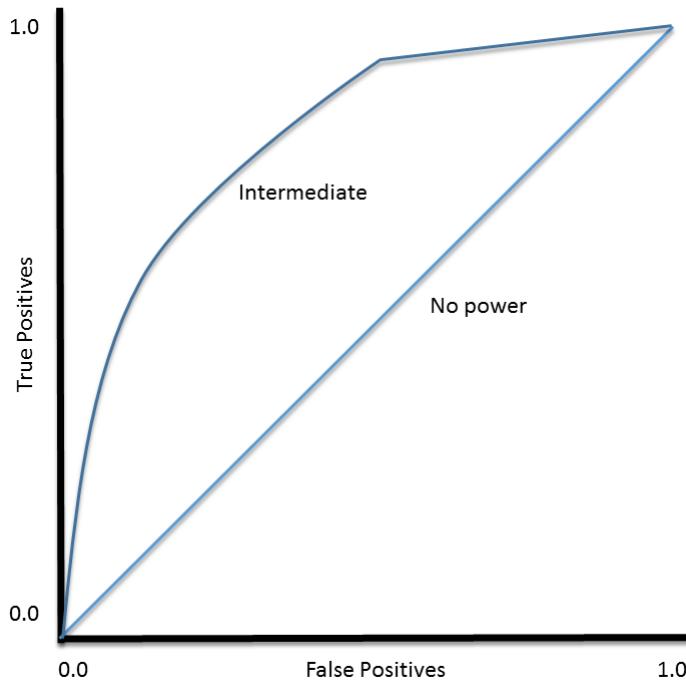


Figure 6.4: ROC Curve

6.4 Results

After running the experiment, it is possible to conclude that all the entities were stored in the SensorThings API database correctly.

Furthermore, it is also possible to conclude that of the 35 unique units, 16 properties were stored only based on the symbol as seen in table 6.1.

Some properties are not present in the unit ontology, so even though the system will compare them using string algorithms, it will not find any match. These properties are: *Power Factor (cosφ)*, *FAIL/OK*, *DateTimeStamp*, *OFF/ON/FAIL*, *ALARM STATUS*, *StateInteger*, *State Volume flow rate (m³/h/kg)*, *ON/OFF*. Although, these properties are not in the ontology, the system will store them with a default *NULL* property and notify the user.

Therefore, before applying the string matching algorithms to the 11 units; it is important to look at them closer to know the units, which the system will compare.

By observing table 6.2 containing the symbols not found in the ontology, there is a comparison between the symbol and unit coming from the sensors against the symbol and label from the ontology. In the last column, it is represented the quantity the system is suppose to find.

Symbols Found	String Matching Algorithms	Not in ontology
16	11	8

Table 6.1: Results experiment

Symbol	Unit	QUDT symbol	QUDT label	QUDT QuantityKind
VAr	Volt-Ampere Reactive	V-A_Reactive	V A Reactive	ReactivePower
VA	Volt-Ampere	V-A	V A	ApparentPower
kWh	Kilowatt Hour	KiloW-HR	KiloWattHour	Energy
kVArh	Kilo Volt-Ampere Reactive Hour	KiloVA-HR	KiloV A HR	ReactivePower
h	Hours	HR	Hour	Time
m	Minutes	MIN	Minute	Time
s	Seconds	SEC	Second	Time
%	Percent	PERCENT	Percent	DimensionlessRatio
ppm	Parts per million	PPM	Parst per million	DimensionlessRatio
kg	Kilogram Flow	KILOGM	Kilogram	Mass
°C	Temperature	DEG_C	Degree Celsius	Temperature

Table 6.2: Breakdown string matching units

6.4.1 Jaro-Winkler Algorithm

For every unique unit, belonging to the list of 11 units presented, the Jaro-Winkler algorithm was used. This means, each unit was compared to the 1537 instances of the unit ontology, plus 11 headers of the file, which are meaningless. These headers are included in the file and represented information about the ontology, but are not instances of units. However, as they could not be removed the algorithm will also compare them.

As discussed, the values of FPR (False positive rate) and TPR (True Positive Rate), were calculated for the different values of threshold, varying between 0 and 1. Also, the value of the F1 scored was computed, in order to determine the best value for the threshold. The results are shown in table 6.3.

Moreover, for each threshold value, a confusion matrix was built. It is expected the values of TP (True Positive) + FN (False Negative) of the confusion matrix, to be equal to 11; representing the total number of units. Also, the number of correct properties is very low, because for each property, there is only one correct match.

It is now possible to plot the ROC curve, for the values of the table as seen in Figure 6.5.

This curve is very close to an ideal ROC Curve. The values of the TPR (True Positive Rate), are always 1, for thresholds smaller than 0.6, which means the algorithm correctly detects all matches. However, it is important to look at the FPR, because even with very small values, it represents a large number of properties being incorrect matched.

In conclusion, looking at the best F1-score (0.58), the best threshold is at 0.9 and it generates the confusion matrix in table 6.4.

With this threshold, the number of correct properties the system found is 7, and the number of false positives is 7.

Threshold	FPR	TPR	F1 score
0	1	1	0
0.1	1	1	0
0.2	1	1	0
0.3	0,88	1	0
0.4	0,65	1	0
0.5	0,26	1	0,01
0.6	0,04	0,91	0,03
0.7	0	0,82	0,19
0.8	0	0,73	0,24
0.9	0	0,64	0,58
1	0	0	-

Table 6.3: Results Jaro-Winkler

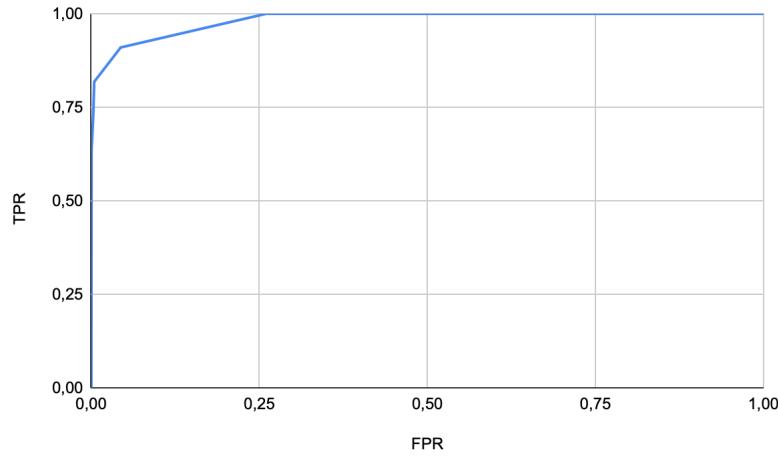


Figure 6.5: ROC Curve - Jaro-Winkler

6.4.2 Hamming Algorithm

In this algorithm, the same approach was taken: the values of FPR (False positive rate), TPR (True Positive Rate) and the F1 score were calculated and it is shown in table 6.5.

In sequence, the ROC graph was plotted and is represented in Figure 6.6.

According to the value of the F1 score (0,56), the confusion matrix in table 6.6 is built. The best value for the threshold is 0,7. With this value, the number of correct properties is 5, but the number of false positives is only 2.

6.4.3 Levenshtein Algorithm

In the Levenshtein algorithm the same methodology was made. The results are in table 6.7 and it produces the ROC curve in Figure 6.7.

	Match	No Match
Property correct	7	4
Property wrong	6	17011

Table 6.4: Confusion matrix Jaro-Winkler

Threshold	FPR	TPR	F1 score
0	1	1	0
0.1	0.18	1	0
0.2	0.04	1	0
0.3	0.01	0.82	0.06
0.4	0	0.64	0.24
0.5	0	0.64	0.52
0.6	0	0.45	0.5
0.7	0	0.45	0.56
0.8	0	0.36	0.5
0.9	0	0.09	0.17
1	0	0	1

Table 6.5: Results Hamming

	Match	No Match
Property correct	5	6
Property wrong	2	17015

Table 6.6: Confusion matrix Hamming

Threshold	FPR	TPR	F1 score
0	1	1	0
0.1	0.8	1	0
0.2	0.3	1	0
0.3	0.07	0.9	0.02
0.4	0.02	0.82	0.06
0.5	0	0.82	0.28
0.6	0	0.54	0.46
0.7	0	0.54	0.6
0.8	0	0.45	0.56
0.9	0	0.09	0.17
1	0	0	-

Table 6.7: Results Levenshtein

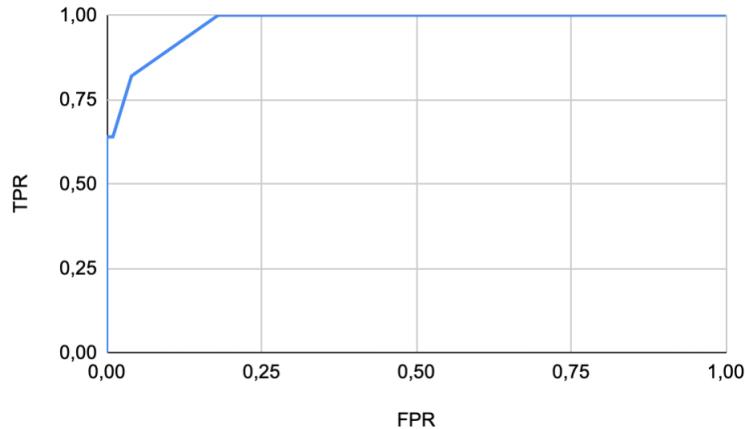


Figure 6.6: ROC Curve - Hamming

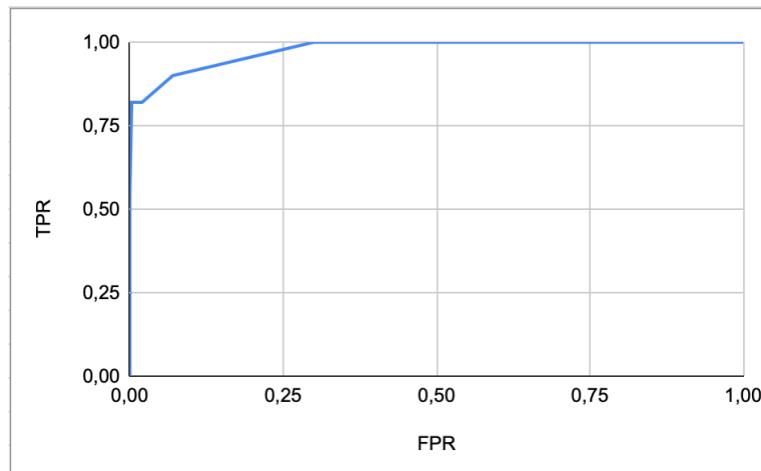


Figure 6.7: ROC Curve - Levenshtein

The highest value for the F1 score (0.6) is when the value of the threshold is equal to 0,7. With this value, the number of correct properties is 6 and the number of false positives is 3, according to table 6.8.

6.5 Discussion

In summary, it is possible to observe the approach taken to solve the problem of autonomous registration was successful.

The SensorThings API component was validated and no errors occurred. From the 35 uniques units, 16 of them are correctly stored in the first step of the method. It is also stated from the beginning, 8 units were registered as NULL properties, because they are not present in the ontology.

In regards to the 11 units being compared using the three different algorithms, the success rate varies between 5-7 properties, using the best values of the thresholds. The algorithm which provides the best results was the Levenshtein algorithm, with a value of F1-score equal to 0.6.

	Match	No Match
Property correct	6	5
Property wrong	3	17014

Table 6.8: Confusion matrix Levenshtein

However, the results may have been differ if the current dataset was bigger and a different algorithm might have been better because they results were very similar. In this situation, there is a lower sample size of units being compared, against the extremely large sample of the ontology.

Chapter 7

Conclusion and Future Work

This chapter summarizes the general work produced during the development of this dissertation. In addition, it also contains a description of the future work, which could be done to improve this thesis and a reflection on the difficulties and problems that emerged.

Conclusion

In this study, several components were used to solve the arising problems in an heterogeneous IoT environment. SensorThings API was selected as the most competent data model to solve the problems presented. FROST implements the standard SensorThings full data model and functionalities as a back-end server instance. The API is scalable, flexible and follows modern web development trends. It contains the essential functionality needed in an IoT environment, while providing interoperability and semantics to the system. The data is stored in compact JSON encoding and can be easily inserted, updated and removed via HTTP requests.

The process of registration was intended to have no user input, so the application uses a unit ontology to resolve the process of autonomous registration. This solution produced a semantically rich way of registering these properties. In this process, three string matching algorithms were considered: Hamming Distance, Levenshtein Distance and Jaro-Winkler. The objective was to compare the name of the units from the event to the ones in the ontology. In summary, these algorithms produced quite good solutions, being the Levenshtein Algorithm considered the best one. However, of the difficulties of this work, was the small sample size of units being compared, due to the dataset. This means, Hamming Distance might have been the best option to this dataset, but with a larger dataset, the results might have been different.

Finally, it was also proved that the application could perform the required operations related with the management of entities, creating, deleting or updating, meeting all the goals of this work.

Future Work

Looking at the work done, there are some features that could be added that could improve and enhance the application. Although, it was not in the scope of the project, it would be useful to add a dashboard of the application, where the user could see in a visual way the entities present in the database and have a better understanding of the management of the events.

Moreover, in regards to the process of autonomous registration, there are two feature, which could had more value. First, is , as referred before, a bigger dataset, in order to test the string matching algorithms. Secondly, it would also be interesting to improve the process of autonomous registration, by utilizing the values of the observations. This process, would imply the system would look in the database, and did a comparison to the readings from the event. This second layer to the mechanism implement, could protect the system from mislabeling the properties and improving the overall accuracy.

References

- [1] Assessing and comparing classifier performance with roc curves. <https://machinelearningmastery.com/assessing-comparing-classifier-performance-roc-curves-2/>. Accessed: 2020-06-10.
- [2] An introduction to time series databases. <https://severalnines.com/database-blog/introduction-time-series-databases/>. Accessed: 2019-12-10.
- [3] Qudt ontologies overview. <http://www.qudt.org/pages/QUDToverviewPage.html>. Accessed: 2020-05-23.
- [4] Turtle - terse rdf triple language. <https://www.w3.org/2010/01/Turtle/>. Accessed: 2020-05-30.
- [5] K. J. Ashton. That ‘internet of things’ thing. page 97–114, 1999.
- [6] Garvita Bajaj, Rachit Agarwal, Pushpendra Singh, Nikolaos Georgantas, and Valérie Issarny. A study of existing ontologies in the iot-domain. 07 2017.
- [7] Garvita Bajaj, Rachit Agarwal, Pushpendra Singh, Nikolaos Georgantas, and Valérie Issarny. A study of existing ontologies in the iot-domain. 07 2017.
- [8] A. Bhawiyuga, M. Data, and A. Warda. Architectural design of token based authentication of mqtt protocol in constrained iot device. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, pages 1–4, 2017.
- [9] Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, David Kelsey, Danh Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, and Kerry Taylor. The ssn ontology of the w3c semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17, 12 2012.
- [10] Sheik Fattah, Nak-Myoung Sung, Il-Yeup Ahn, Minwoo Ryu, and Jaeseok Yun. Building iot services for aging in place using standard-based iot platforms and heterogeneous iot products. *Sensors*, 17, 10 2017.
- [11] Dan Gusfield. Algorithms on strings, trees and sequences: Computer science and computational biology. 1997.
- [12] Chih-Yuan Huang and Cheng-Hung Wu. A web service protocol realizing interoperable internet of things tasking capability. *Sensors*, 16:1395, 08 2016.

- [13] A. Geraci J. Radatz and F. Katki. *IEEE standard glossary of software engineering terminology*. IEEE Std, 1990.
- [14] Antonio J. Jara, Martin Serrano, Andrea Gomez, David Fernandez, Germán Molina, Yann Bocchi, and Ramón Alcarria. *Smart Cities Semantics and Data Models*, pages 77–85. 01 2018.
- [15] Alexander Kotsev, Katherina Schleidt, Steve Liang, Hylke van der Schaaf, Tania Khalafbeigi, Sylvain Grellet, Michael Lutz, Simon Jirka, and Mickael Beaufils. Extending inspire to the internet of things through sensorthings api. 05 2018.
- [16] Chao-Hsien Lee, Yu-Wei Chang, Chi-Cheng Chuang, and Ying Lai. Interoperability enhancement for internet of things protocols based on software-defined network. pages 1–2, 10 2016.
- [17] Steve Liang, Chih-Yuan Huang, and Tania Khalafbeigi. *OGC SensorThings API Part I:Sensing*. 08 2016.
- [18] Daniel Marsh-Hunn. Interoperability enhancement of iot devices using open web standards in a smart farming use case. 2019.
- [19] William C. McGee. On user criteria for data model evaluation. *ACM Trans. Database Syst.*, 1(4):370–387, December 1976.
- [20] Paul Murdock, Louay Bassbouss, Andreas Kraft, Martin Bauer, Oleg Logvinov, Mahdi Alaya, Terry Longstreth, Rajdeep Bhowmik, Patrica Martigne, Patrica Brett, Catalina Mladin, Rabindra Chakraborty, Thierry Monteil, Mohammed Dadas, John Davies, Philippe Nappey, Wael Diab, Dave Raggett, Khalil Drira, and Imran Khan. Semantic interoperability for the web of things. 08 2016.
- [21] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, March 2001.
- [22] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. *Interoperability in Internet of Things Infrastructure: Classification, Challenges, and Future Work: Third International Conference, IoTaaS 2017, Taichung, Taiwan, September 20–22, 2017, Proceedings*, pages 11–18. 01 2018.
- [23] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. Interoperability in internet of things: Taxonomies and open challenges. *Mobile Networks and Applications*, 07 2018.
- [24] Maria Palattella, Nicola Accettura, Xavier Vilajosana, Thomas Watteyne, Luigi Grieco, Genaro Boggia, and Mischa Dohler. Standardized protocol stack for the internet of (important) things. *IEEE Communications Surveys and Tutorials*, 15, 01 2013.
- [25] Bruce Simons, Jonathan Yu, and Simon Cox. Defining a water quality vocabulary using qudt and chebi. 12 2013.
- [26] Charles Steinmetz, Greyce Schroeder, Alexandre Roque, Carlos Pereira, Carolin Wagner, Philipp Saalmann, and Bernd Hellingrath. Ontology-driven iot code generation for fiware. 07 2017.

- [27] JUAN YE, LORCAN COYLE, SIMON DOBSON, and PADDY NIXON. Ontology-based models in pervasive computing systems. *The Knowledge Engineering Review*, 22(4):315–347, 2007.
- [28] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. Internet of things for smart cities. *Internet of Things Journal, IEEE*, 1, 01 2012.