Babel: A Framework for managing the Heterogeneity of IoT Applications

BY

FEDERICO MANNA
B.S, Politecnico di Milano, Milan, Italy, 2014

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2017

Chicago, Illinois

Defense Committee:

Ugo Buy, Chair and Advisor
Mark Grechanik
Luciano Baresi, Politecnico di Milano

To my lovely family...

Without you none of this would have been possible.

ACKNOWLEDGMENTS

My first thanks are to professor Luciano Baresi, who followed me during the development of the thesis, who taught me a lot and who inspired me with his words.

I would also like to thank professor Ugo Buy, from Chicago, who helped me even though we are far and who was a model to follow during his lessons last year.

Other important thanks go to the all guys of the JOL S-Cube, who were always available for explanations and supervised me during the development with precious advice.

Finally I have to thank my family: my mom Mirella, my dad Dino and my brother Alessandro for all the support they gave me in these years, including the period I was studying in the U.S.. My soulmate Lucia for the love and comprehension she gave and continues to give me. All my friends: "the Sizzano's guys", whom I have known since my childhood, especially Paolo and Lorenzo who shared with me tons of life experiences. My classmates at "PoliMi", particularly Matteo and Marco, with whom I shared troubles, exams and joys at university. "The Chicago team" with which I have lived one of the craziest periods of my life.

FM

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

IoT	Internet of Things
WoT	Web of Things
HTTP	HyperText Transfer Protocol
TCP	Transmission Control Protocol
IP	Internet Protocol
REST	REpresentational State Transfer
WS	Web Service
HTML	HyperText Markup Language
XML	Xtensible Markup Language
JSON	JavaScript Object Notation
JSON-LD	JavaScript Object Notation for Linked Data
RFID	Radio- Frequency IDentification
W3C	World Wide Web Consortium
M2M	Machine to Machine
M2H	Machine to Human
H2M	Human to Machine
MVC	Model View Controller

SUMMARY

The world will have more than 50 billions devices connected to the internet in 2020. Most of them will belong to the "Smart Objects" category. They are part of the Internet of Things world and it becomes fundamental to understand what this new paradigm involves. The IoT will enter our lives comprehensively and silently but bringing about an important change for every one. More and more smart objects will be used by everyone in public spaces, houses or industries. It will be a revolution: the Internet will be not only in devices designed for communication but also in "everyday objects".

Today the IoT world is fragmented, with a myriad of custom and proprietary solutions. Fragmented does not mean distributed: the IoT, belonging to "everyday objects" must be distributed but nowadays it is fragmented in communication solutions and in protocols stack. The "average user" is confused, he/she has to deal with multiple and different technologies, UIs and conventions. Every vendor and producer has a univocal way to develop systems and provide users with services. It is fundamental, for a technology which will become global, to have a sort of standard to conform to.

This thesis has the aim to bridge the gap between the IoT and the standardized web technologies. In order to do that it is necessary to create a reliable, neutral and universal middleware which can uniform the description of the smart entities. It is necessary to find a common language to describe the smart things or, more generally, the smart spaces. The description should be exchanged among different systems without creating confusion and ambiguity. The vehicle

for the exchange of information must be a file which can be easily exchanged on the network and parsed rapidly when received. My work analyzes all these requirements and develops a prototype to show how it is possible to control the IoT world - which, by definition, is distributed - through a centralized end point.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Nowadays the Internet cannot be seen only as a network of computers connected to each other: technological innovation has slowly integrated "everyday" objects into the internet, creating the so called *"smart things"*. Networks today are also composed of objects which monitor and control the world around us. Home appliances, such as TVs, thermometers, alarms, etc. communicate and share data thus creating the concept of *smart house*; RFID-tagged objects to track products and sells in stores or to monitor the production in industries have generated the concept of *smart retail and industries*; intelligent public infrastructures, green spaces and traffic control are some of the keywords and enabling technologies for the *smart city*, one of the biggest challenges of the 21st Century.

To make such a world possible, the objects of everyday use should be given the chance to communicate. But we cannot apply the paradigm of the classical internet to objects that mainly have other purposes and are not connected to the internet. Smart objects are heterogeneous, each one has a different goal and often constraints on power consumption, battery life and communication power or range. A new type of connection complying with these constraints is needed. The Internet of Things and several researches come with a myriad of different solutions, such as ZigBee, 6LowPAN, IEEE 802.15.4, etc.; new low power protocols for the physical and

transport level to permit communication among devices come out everyday.

Following the classical stack (ISO/OSI) of communication protocols, the Internet of Things standardizes layer 4, the *Transport* one, leaving the upper levels without a common interface to communicate: the consequence is we have lots of different protocols, some have an embedded *Application layer* internally developed that can be open or proprietary, some others stop at Transportation, leaving the rest to third-party solutions and development. The outcome is a fragmented set of protocols, often hardly integrable in more complex systems.

This thesis seeks to create an intermediate layer at application level that tries to uniform the different protocols, providing a unified description of the objects. The goal is trying to create only one "source" of data, the middleware, from which it is possible to retrieve the unified description and send commands for the Web of Things, which identifies what is created on top of the Internet of Things. As we have the classical internet and the classical web, the Web of Things is built on top and "speaks" with the final user. Creating a framework that uniforms the output of the lower levels, it is possible to easily create third-party applications for the final user. This will enable writing neutral applications (web applications, mobile ones or desktop standalone ones) for every house, regardless of the vendor of the smart object or the protocol the device is running.

The proposed solution is a *middleware* which provides a *JSON* file, obtainable through a *HTTP GET*, composed of a hierarchical structure, including all the objects available in a building or in a room, with all their states, descriptions, properties and available commands.

For the *case study* in this thesis I used some specific objects, in order to create appropriate

"plugs" from which to read and create the uniformed expected outcome; while the more general idea is that each vendor could embed in the future this standard in each smart device. Also, in the same way, the discovery part in the implementation is done only for some chosen protocols, while a more general structure would include every kind of device. Maybe an additional layer that can contain the server part for the objects discovery for all protocols (or most of them) could allow my framework to have the whole set of objects installed.

1.2 Outline

The thesis is organized as follows:

In the second chapter the focus is on the state of the art: a full overview of the Internet of Things and the Web of Things is provided, their definition and the differences from the classical internet. The final section explains some existing technologies which can be useful later on.

In the third chapter I focused my attention on the problem.After a contextualization, it is necessary to give a precise and detailed definition of what is the problem, what is missing and what I am going to try to develop. Also, I set technical constraints based on the considerations done before.

In the fourth chapter the solution is analyzed in detail. It can be considered the core and the central point where the innovation is described. Firstly, the idea how to solve the problem is reported, also with images, to make clear to the reader what really has been done.Then the detailed solution, the technical aspects are faced and explained. The chapter also has a full use case commented, to show how the idea really works.

In the fifth chapter attention is focused on the implementation of a real case study. With my

idea I tried to describe the smart space of my laboratory. The focus is on the implementation, all the development is explained, how classes and packages are disposed and how they can make the system works. Finally a "working demo" is shown, thanks to many screen shots the reader can have a full idea of what has been done and how my work has been transformed from a theoretical idea to a working system.

In the sixth chapter it is possible to find final considerations about what has been done. Firstly, an analysis of what I have achieved, what can be considered promising and with a future. Then some ideas of how to extend the work in the future are reported. Having faced the problem for months I think it is necessary to give guidelines if someone wants to approach it and give an extension.

CHAPTER 2

STATE OF THE ART

2.1 Classical Internet

The classical digital world is mainly composed of computers, responsible for connecting the entire humanity, the nodes of a complex and extensive graph describing the electronic world. Computers are standard machines, each one composed of the same components, but, on the other hand, they are versatile, polymorphic. They can be used to do everything humanity needs to do and that is the reason why they were created and they are very popular. The information itself, anyway, is almost always created and manipulated by men.

The term "computers" means all the devices whose main purpose is to support mankind (and to do so it is often necessary to be connected to the internet) and this category includes the computers, smartphones and tablets. They are thought to be *smart* from scratch, they are tools that can be helpful and that can offer features that no other object offers.

The great amount of actions they are capable of is enabled by the internet or, at least, by communication among devices (GPS, physical internet, bluetooth, etc. can be considered direct or indirect forms of communication).

The online part of this world which includes the device described above is known as the *"classical internet"* [1].

2.2 Internet of Things

2.2.1 Definition

Starting from the end of 90s and then in 2000s, a new concept was coined: *Internet of Things (IoT)* [2]. The Internet of Things, in years, has been defined in several ways: it is a global network composed not only of computers and backbones, but also of "everyday" objects, smart objects. The scientific progress in embedded devices and miniaturization of digital components is leading us to have more and more smart objects: they are heterogeneous devices, physical entities with different functionalities [3]. They can be simple sensor nodes created on purpose for the IoT, nodes that sample and send data to other devices but they can also be complex objects such as fridges, which can now share data and receive commands in addition to their main function [4].

As said, IoT can be viewed as a global network even if it has limitations. While the "classical internet", described in 2.1 is dominated by only one common layer, the web, which is the same of every type of "classical" devices connected, accessed by everyone; the IoT creates a global network, because every one can be online, but with different points of access. There is no common layer on top that enables a full cooperation among devices. No one can have a global view of the whole situation due to the solutions adopted: they are not opened and so standardized, every vendor has developed or used a different technology. It is important to point out from the beginning that this fragmentation is due to the strict limits on power consumption which forced to search alternative solutions to the standard structure of the well

known internet. The IoT is not useless, it is a good starting point to open up to a new world, a smart world.

2.2.2 Building Blocks: Smart Objects

In this section smart objects are classified more formally, even if the classification is not set in stone it is possible to identify three main categories: *smart appliances*, *sensor nodes* and *RFID objects*. Smart appliances, as already said, are complex devices with a main purpose that is not IoT and communication but, thanks to digital innovation, possessing a way to communicate over a network. Sensor nodes, as the word says, are created to "sense" the environment and sample data to send, often, to a central unit that monitors lots of aspects in a given smart space. Sensor nodes are identified by four capabilities: *sensing* the world, *processing* information, *storing* information, *communicating* with other smart devices. These devices in particular are responsible for the growth of big data regarding the environment, production processes, etc [5]. RFID means Radio-Frequency IDentification, and it is a technology to exchange information and automatically identify different types of tags that can represent a physical entity in the world. The RFID objects are the simplest, they are composed of a tag, a reader and an information system to manage data for the transfer to and from the readers. The proper smart object is the RFID tag, which possesses a microchip to keep data in memory and an antenna to transmit data.

Having defined this classification, it is easy to notice how the fundamental characteristic to define smart an object is the possibility to communicate to other devices. The classical *ISO/OSI stack* provides 7 layers, which can be divided into two parts: from 1 to 3 and from

4 to 7. The first group is needed to guarantee the *communication* among devices, while the second one is needed to *elaborate information*, constructing applications with no need to worry about the underlying network. While the former group is dealt with later, it is fundamental to have a common layer 4 and 3 to assure the right communication among devices.

Over the years lots of solutions were proposed to provide connection. Different network protocols such as ZigBee, Bluetooth Low Energy, IEEE 802.15.4 or low-power WiFi and 6LoWPAN were released during these years creating a sort of confused cloud of solutions. Available possibilities, pros and cons are analyzed in depth in the next pages. The main focus is the fact that these protocols are not equivalent: some include the full ISO/OSI stack reaching users with the application layer (7), some others stop at lower levels with obvious compatibility problems. Developing third-party applications uniforming them remains a challenging and time-consuming task which requires deep knowledge of each smart things platform and protocols.

The support for the required protocols can be obtained in several ways, and this is going to divide the devices with another cut. It is possible to have the devices that embed the connectivity, they can be considered the most advanced ones, there is no need of additional hardware to elect them as "first class citizens" of the WoT. In this case the devices must have an HTTP connectivity, over Wi-Fi, very often, and sometimes on ethernet. The Wi-Fi is needed for some types of devices whose distance from the plug is considerable, also the Wi-Fi does not imply the presence of cables and it is much better if you think of a room in a house or a retail center. These devices can be considered with *native web support*.

Another option, if embedding the native support is not feasible, is the possibility to add

a *Smart Gateway*, a piece of hardware: the "real" object is not powerful enough to embed an HTTP web server, so a sort of "station" called gateway is installed in the smart space. It is plugged to the power grid, it is more powerful than a simple object and its goal is to act as intermediary for communications: when data is received from the object, with a lower level protocol, the gateway bundles it into an HTTP packet and sends it over the internet to whoever requests. The opposite thing is done when a command, for instance, is received by the web application: the gateway extracts the payload and sends it to the object using "its language".

2.2.3 IoT vs WoT

IoT can be viewed as a sort of integration with the classical internet, not a replacement, because computers and standard devices are still present but now they have to cooperate with all these new objects that provide connectivity. But this statement is not very precise: as it was said, IoT uniforms the top of the *communication* section of the ISO/OSI stack (layer 3 and 4), leaving the upper layers as a fragmented incompatible myriad of solutions. The limits of the IoT become obvious as soon as a developer wants to create an application which contains smart objects coming from different vendors. The *Web of Things* tries to be a unique solution for the highest level of the stack.

2.3 Web of Things

2.3.1 Definition

The *Web of Things (WoT)* is the application layer for the Internet of Things, it describes a world where the "everyday" objects are connected not only to the internet but they are fully integrated in the web. It means that smart objects would be able to access and provide web

services without being re-invented but integrating the existing ones internally.

The IoT handles the physical objects with a connectivity until the transport layer allows them to communicate. But we can state that this kind of communication is coarse: it is only for one vendor, maybe a proprietary solution, and there is no possibility to integrate in larger applications the objects that belong to different brands. The WoT has to face this fragmented situation, with the goal to try to unify it. The Web of Things considers smart things as native elements of the web and considers the WoT as a refinement of the IoT, by integrating smart objects into the Web (the application layer) and not only into the network (the transport layer and lower layers). It is possible to compare the relation between the classical internet and web with the IoT and the WoT. As the web is the highest layer in the standard ISO/OSI stack, the one that "speaks" to the final user and is reachable by everyone, for example, through a browser so the WoT tries to be the highest layer for the new conception of the web, the one that includes the smart objects and their ability to communicate. It can be considered the highest layer of a modified protocol stack which includes all the low power standards presented and created by the Internet of Things.

Figure 1 explains this concept in a graphic way, putting the web and the WoT on the top of other layers. The particularity is in the tools that this layer uses: HTML (HyperText Markup Language) or JSON (JavaScript Object Notation), for example, are created to be readable for men as well as for computers. HTML, as proof of what I am saying, composes the whole web as we know it: it is a markup language that permits to structure a web page or a portion, then the browser downloads and interprets it creating a graphical rendering to make life easier to

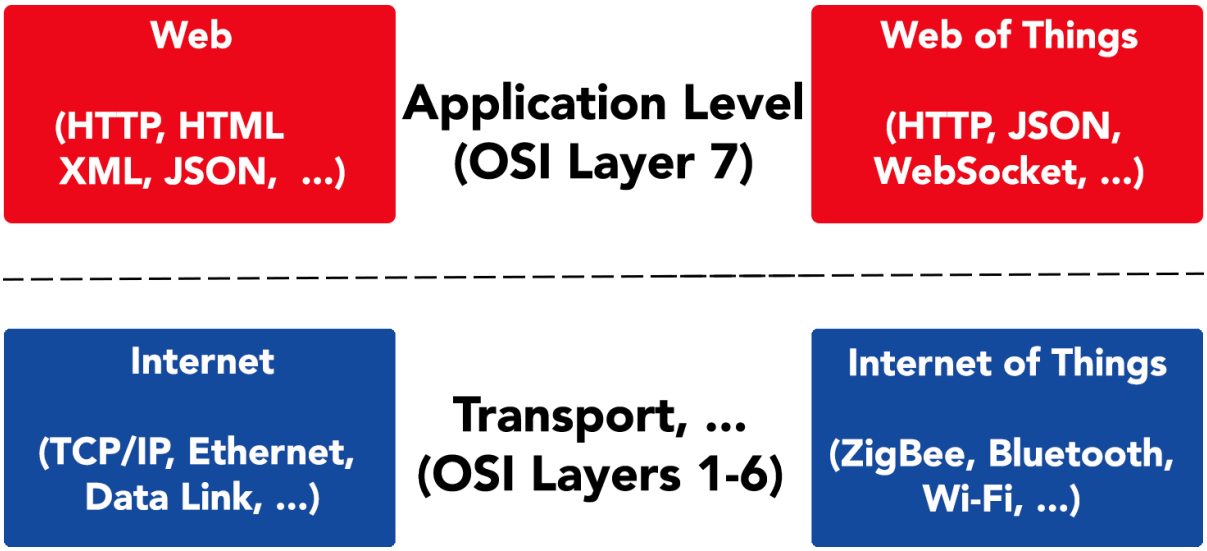

Figure 1: Web is the common factor for communication

the final user. The WoT has the same aim, and to achieve it, the WoT necessarily has to use tools compliant to the goal: make life easier.

The aim in the Web of Things is to represent the "everyday" objects online, as resources, and exchange data with them using REpresentational State Transfer (REST) paradigm. So it is possible to use well known tools for building the different kinds of applications.

2.3.2 Requirements

In this section I will list the requirements a device needs to be a part of the Web of Things. It is not sufficient, for instance, to have a device connected to its hub or its smart center "someway", this can be enough to have a proprietary and closed solution. To be part of the WoT the device must satisfy two requirements:

- Implementation of the TCP/IP protocols. It could be done, hopefully, over a IEEE 802.3 standard (Ethernet) or IEEE 802.11 standard (Wi-Fi) network [6].
- The first requirement is the base for the second: supporting the HTTP protocol. The devices must behave like a web server: they have to provide data, periodically, to who is asking. The web application, developed on top, has to be a server for the user, but, on the other side, it strongly depends on objects, that are the source and destination of data, so it is a sort of client of multiple servers that must be able to process requests and output a valid answer.

This division was formulated in [7] but I completely agree and from my point of view here there is no need of reformulation.

2.4 Technical Background

This section has the aim to give a useful technical background for the implemented and proposed solution.

2.4.1 Linked Data and Semantic Web

The internet is seen as a network composed of nodes that are "bare metal" objects: computers and servers principally, or smart objects. The web, in the same way, is a network, but it is composed of nodes that are websites or web applications. These nodes are linked, with URIs, as widely explained. While the humans can understand what a link stands for, according to where it is put in a page or which keyword is put near it, the machines cannot have this idea, for them a link is simply a reference to another "portion of the web", another web page to download. To make the machines understand what a link is and what it represents in that

particular context, it is necessary to construct linked data. Linked data are simply a way to make the machines conscious of what they are facing: in this way also computers can have a global vision of the web as an almost connected graph, understanding the relations existing among different pages or applications. If, for example, we define on the web the concept of human being and the relation "marriedTo", the computer can know that the the object of the relation is another human being, so it can expect some type of data and it can raise an error if, for instance, the received type is a cat. This understanding can lead, for example, a computer to be more and more precise and capable of answering complex queries like "Who is Thomas's father?": knowing the relation "father", the machine only follows the link provided to get and show the answer. A semantic cognition is what modern search engines are trying to reach [8].

Meta-data is what permits the recognition, like a index for a library: there you can find all the useful information to index the real information, the books. Here it is the same, meta data, specified and filled in the HTML page are what enables the computer to know the relations existing among what it has already downloaded and what it can reach from there [9].

If the web, a complicated graph, can be understood by a machine, we are talking about semantic web. A web with a consciousness of itself. [10]

In Figure 2 the node here called "_:a" is a node of type "person", a URI (`http://schema.org/Person`) defines what a "person" is. The "person" here has two different properties: "name" and "birthPlace". The "name" property is a string ("Paul Schuster"), a "terminal" (because it is not expanded, identified by a rectangular shape) property of the category "person". The property "birthPlace", on the other hand, is not a "terminal one", identified by a

circle shape. It is a "place", another web identity. For this reason it has its own definition with another URI (http://schema.org/Place) and it can have others properties (here not reported) to link a "place" to the remaining rest of the web.

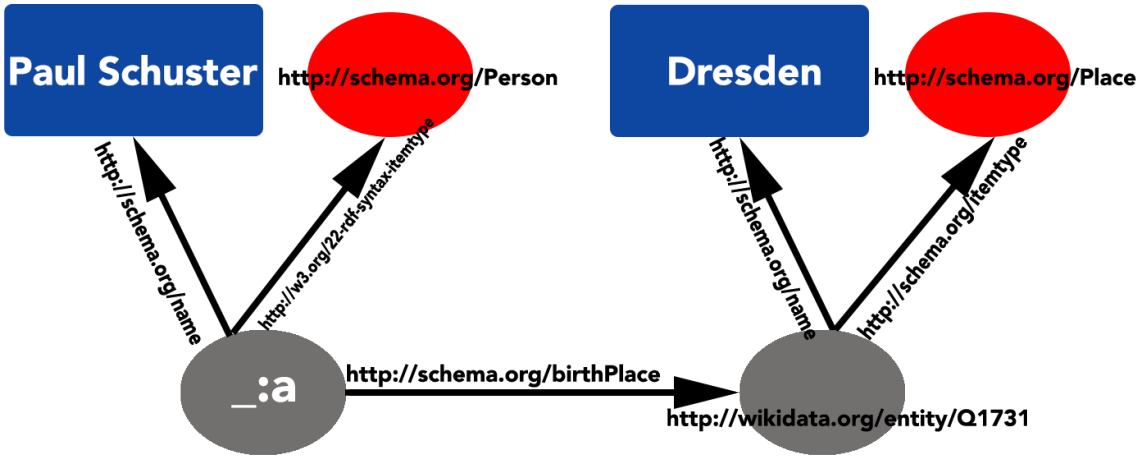

Figure 2: An example of semantic web

2.4.2 JSON-LD

JSON-LD stands for *JavaScript Object Notation for Linked Data* and it is intended to be an extension, an evolution of the JSON: it brings the web to a semantic level. Having given a first explanation of what linked data and semantic web are, we can now explain JSON-LD and its features useful for my purpose. JSON-LD is a type of document, coming from the JSON,

that keeps its structure and its syntax but it adds some keywords in order to be part of the semantic web, and to better describe a smart space [11]. In particular the JSON-LD document uses the concept of *@context* as starting point. The @context can be explained as a context of an ordinary conversation: speaking with someone I have some external elements that help the interlocutors to understand each other, the environment, the place, the weather, etc. The concept is the same when trying to contextualize a conversation between two machines. Here the context maps the *IRIs (Internationalized Resource Identifier)* (more general URIs, using Unicode instead of ASCII) to *terms* defined in the document. Terms must be valid strings, also they are case sensitive and they cannot be JSON-LD keywords, already reserved. Then, in the rest of the document, we can refer to a particular definition given by the IRI, such as "place", with the term, without mapping it every time. A @context can be defined internally, mapping explicitly every term to an IRI or can be defined online somewhere, in this case it is considered external.

The new format does not bring only the @context keyword: some other keywords are fundamental to make the web a semantic graph for a computer. The tag *@id* is introduced to correctly identify nodes inside the web, the @id component contains the place, in form of IRI, where the representation of the resource, and in particular of the node, is. The @id tag can be considered the "arrow" the parser has to follow in order to retrieve the node of the graph. @id is not pointing to a definition of what the resource is but to its online representation.

Figure 3 shows how in the @context two terms have been defined, "name" and "homepage", in particular the latter uses the @id tag to point to the resource and @type that is explained

```
{
  "@context": {
    "name": "http://schema.org/name",
    "homepage": {
      "@id": "http://schema.org/url",
      "@type": "@id"
    }
  }
}
```

Figure 3: JSON-LD: @context, @id example

in the next paragraph.

Another important tag, as already anticipated above, is *@type*, it has a double meaning: it can define the type of the node, it usually points to an IRI containing the formal description of a type of node. The other use of the @type tag is to define the type of a value: in this case it is used with the tag *@value* outside the @context, or inside it defining a term without the need of @value. What is bound to a value type can be a type defined externally, always a IRI helps doing that, or a native JSON type: in this case there is no need of an external link, the JSON parser already knows how to handle it.

Finally, I would like to explain the concept of *active context*: the @context in a JSON-LD document is not unique, more than one can be defined, mapping new terms or reusing some already defined. The active context is how the JSON processor handles the presence of multiple context: a list of terms is kept by the processor, updating the IRIs mapped in case of redefinition. So redefining a term overwrites the old "value", keeping only the last value active.

Between the two definitions of the same term, obviously, the first value is the one that is active. Each @context in a multiple context definition is called *local context*. Setting a local context to "null" resets the entire active context.

```
{
  "@context": {
    "name": "http://example.com/person#name,
    "details": "http://example.com/person#details"
  },
  "name": "Markus Lanthaler",

  "details": {
    "@context":
    {
      "name": "http://example.com/organization#name"
    },
    "name": "Graz University of Technology"
  }
}
```

Figure 4: JSON-LD: multiple @context example

In the example above Figure 4, the term "name" is overridden in the most deeply nested "details" structure. If a term is redefined within a context, all previous rules associated with the previous definition are removed [12].

CHAPTER 3

PROBLEM ANALYSIS

In this chapter the specific problems of this work will be detailed and analyzed, explaining what are the limits and the constraints the challenge has. The chapter starts with a contextualization of the problem, followed by the proper definition of what I faced, while in the last part there is a list of constraints my architecture will have fulfilled in order to have a universal and functional solution.

3.1 Contextualization

In the previous chapter, especially in sections 2.2 and 2.3, I have defined IoT and WoT, pointing out the main differences between the two definitions. In particular I focused on the problem of the interconnection between the two different paradigms. The IoT can be seen as a network, in which not only standard computers are connected as nodes, but also smart objects: they are "first class citizens" of the internet, they have a connectivity module, they can "talk" each other. But, due to some constraints, in particular power consumption constraints, it is not possible to consider the IoT as a big unified network, because devices have different communication protocols and not all of them are compatible with each other to offer communication. The result is a non standardized confused cloud of protocols, systems and technical stacks. Solutions are often proprietary and closed, even if some open solutions exist.

The WoT is the election of the smart objects to "first citizens" of the web, the web has

a fundamental role as unifying layer, it is standardized, it easily reachable by everyone and there is no need of particular requirements on devices. The idea is bringing the devices to the web. The fundamental requirement is the implementation of TCP/IP and HTTP protocols. Not every object can provide this connection, but most of them could, also with the help of a gateway which can be also called *reverse proxy*. In this chapter I am considering only devices that have this feature.

The context of the problem is the current situation of these two worlds: their interfacing and coexistence is the core of the question, they are very different and it is very challenging to have a "full-stack architecture" able to cover all the aspects of a functioning system. On the one side we have a confused cloud of objects, different protocols and behaviors and they are similar to a crowd of people all talking a different language; on the other side there is the web, with well defined protocols, paradigms and frameworks which needs information at least similar each other, in order to be shown and understood by a final user.

3.2 Considered Devices

First of all it is necessary to put a boundary on types of objects considered; I am going to take into account only devices that can be somehow connected to the internet, the ones which provide an implementation of HTTP protocol in particular, while in the substrates the stack can be left "opened": each vendor can implement its solution only if it provides a classical connection on the top. This statement can be seen as a "little relaxation" of the constraints previously announced, 2.3.2. This means that all the devices which cannot provide a HTTP connection are excluded by my work. This decision was taken for two main reasons:

- As previously anticipated in 2.2.2, the technical innovation is leading us to have more and more devices embedding the classical internet interface thanks to the energy requirements, which are everyday decreasing the necessary quantity of power to assure a full working system. In the future, almost all the smart objects will have a classical connection on their "upper side".
- The principal reason for which I have excluded devices without an HTTP end point is the fact that if the connection is not native it must somehow "be injected" to have objects first-class citizens of the web. This statement contains much more than what it seems: how can it be possible to include connection in a proprietary hardware which does not support it natively? Sometimes it can be simply impossible, other times it can be possible but with an incredible effort. It is necessary to understand the internal functioning of the smart device, how it retrieves information from the environment, where the device stores it and in which form. After having done this, the next step should be to understand how to access the information, read it and then send it to a HTTP endpoint, reachable by my system. It probably requires external hardware to send and receive data, a web server constructed on my own to communicate with the object and all this must be done respecting the energy constraints of the IoT objects: if, doing this customization, the battery drains in 5 minutes all the work is completely useless. Finally, it is necessary to point out how this process can be repeated for all kinds of objects I would like to include in my work. It is pretty clear that it is impossible to realize a middleware at this low level, or if it possible it can take months or years.

Having done this clarification, it is important to remark how almost all "everyday smart objects" have a connectivity: only some small and with a particular purpose objects are not able to communicate over the internet. So I am not excluding so many devices from the solution, it can still be considered "universal".

3.3 Definition

Having set a first boundary in the confused world of the IoT, now it is time to define what the problem consist of, giving a complete overview of collateral problems or challenges that my work can create or not completely solve.

As said the Web of Things is the "unification" layer built on top of Internet of Things, it does not replace IoT, it is a completion. There is a sort of vertical hierarchy. If both must coexist a sort of communication between them is fundamental to assure the right functioning of the entire stack. We have clear in mind what can be considered WoT for the final user: a sort of control center, reachable by a web browser, where anyone can control the smart objects he has rights on. Also what is IoT for final users is clear: it is a heterogenous world, a world made of objects they can control through smartphone, for instance. So we have an idea of the two parts we have to connect: on one side the well known web paradigm, with its programming languages, its limited number of web browsers or operative systems; on the other a myriad of objects which can act as web servers, managed through HTTP calls. *How can we let these two parts talk?* This is the question that my thesis is trying to answer. My work is a concrete solution, it is about creating or refining some communication methods.

So I am trying to let different worlds talk using well known tools: I have just said I am putting IoT and WoT in communication, but with the warranty that the smart objects can behave as web servers. So I am defining a method to connect different, "vertical" worlds, put one on the top of the other using a "horizontal" system: devices have HTTP endpoints that elect them to citizens of the web. But this does not guarantee they have a right behavior to be integrated in a heterogeneous and user friendly application. The result is a vertical connection that covers all the protocol stack, from the physical layer to the application one, while the last part of the connection is a "double step connection", a hardware part that guarantees the online presence of the device, and a software part that assure the right behavior and the standardization of the communication Figure 5, describes what has just been explained: the green arrow indicates what is seen externally, the protocol stack, it links directly the application layer with the underlying ones. The red arrows indicate what I am proposing in this thesis: red one in the layer 4 is our requirement, the fact that the smart objects can act as web servers that are components of layer 7. The second red arrow, the one included in the application layer is the representation of what my work is trying to achieve, the software component of the full integration, as above described. As said the red arrow is all in the application layer, it remains "horizontal" in respect to the stack, it can be seen as an HTTP call done from the client, put in the "cloud" (note that what here is considered client can be the backend, the server, of a third-party application). The double side of the arrows stands, obviously, for the possibility to enable communication both sides, packing and unpacking information.

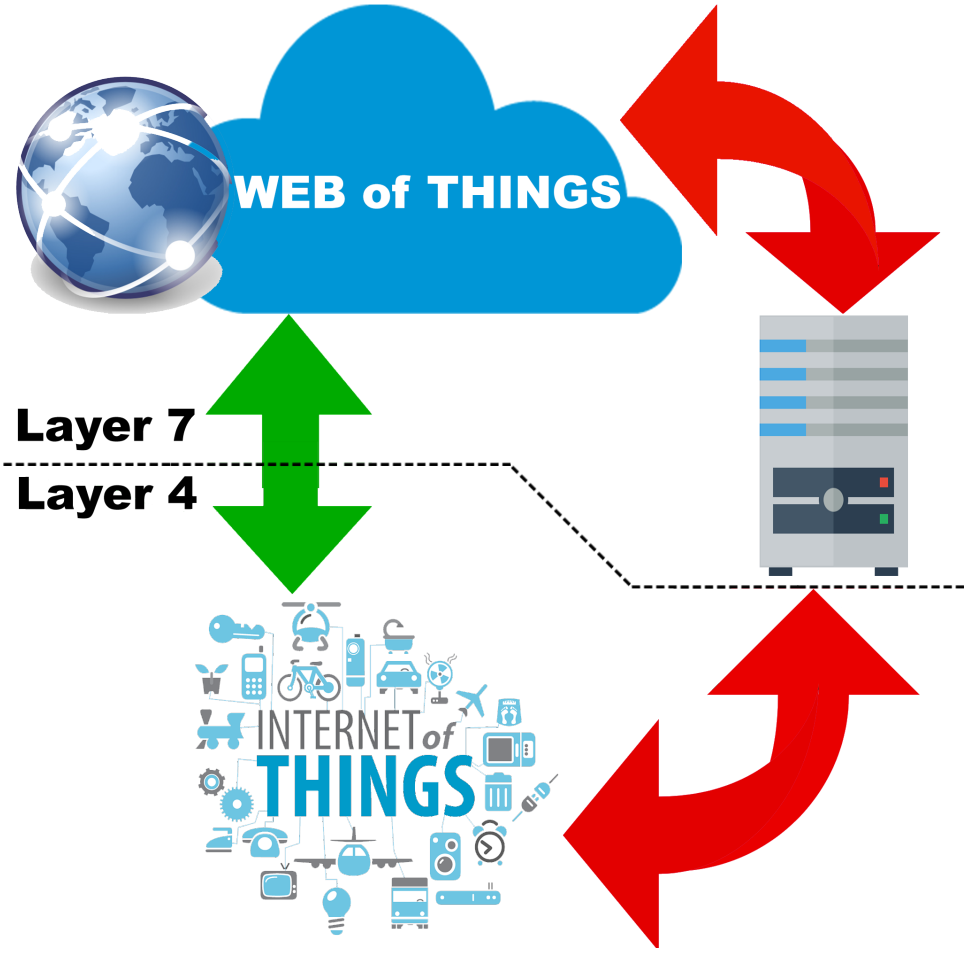

Figure 5: Stack result vs double step implementation

So the problem I am going to face is the *lack of a standard communication method* between the smart objects and the web interfaces already present. Lack of standards which can be somehow "predicted": we have two profoundly different worlds, one is 20 years old, it has already passed through standardization processes, conflicts and it is still under development even if it can be considered stable with the last release of its major components (HTML5, CSS3,

Javascript); the other started only few years ago, solutions are multiple and sometimes they must be different, according to which goal they have to achieve: a type of communication among lamps and a hub in a home is different from the type of communication needed by industrial machines, for instance. The point is here is to find a way to permit communication between these two different entities, it is necessary to put "something" in the middle, "something" able to convert the classical language used in the web to a myriad of different protocols adopted by the IoT objects and viceversa.

What I am going to define in this thesis is a new type of communication, efficient and practical. So it is important to have a "full-duplex" communication between the two sides, anyone must be able to read and write content. Also, in order to do that, a sort of common vocabulary should be defined and learnt by the parts. So the development of the solution is composed of two distinct parts: the choice of the "container" and the proper definition of the syntax. The word "container" means how the system works, the infrastructure of the solution: which type of connection, which type of communication channel, which type of file surfs the web, etc. The "container" is the first step, it establishes how much and which additional hardware, which type of network and connection is needed, it also specifies technologies, programming languages and skills used to assure a good result. As said above, types of communications are a sort of constraints for the defined syntax, but also the choice of the "container" is not completely free, it has to match some fundamental constraints to assure the correct HTTP transmission of a message or a file. In particular the choice of the infrastructure must be compliant with the network and web requirements.

Having defined the "container", the second step is to decide what "flows" inside the channel previously created: if the container is the infrastructure, the choice of the type of file and its syntax is the content. The type of file has to be somehow "serializable" in the sense that it must be able to be sent on the web, it must be obviously readable and writable by both parts. Then, regarding the syntax it is important to have a message with a well defined content: it is what the two parts must write and read, so it has to be clear for machines and must be compliant with all the requests of *M2M (Machine-to-Machine) communication*. On the other hand, a developer must be capable of reading and understanding what is going on in order to debug to solve errors or to write a customizable user application, so the communication must require also to match the *M2H (Machine-to-Human) communication* and *H2M (Human-to-Machine) communication* requirements. These types of communications are some constraints of my work and are explained in the next section 3.4.

The figure is a comparison between my problem and a pipeline: I have two different sides to connect, blue and green stand for different "plugs" to apply to different architectures, I have to construct a channel which directly connects the two parts. So this is what I called "container": the whole infrastructure. Then inside the pipe, information must be exchanged, represented with the two red arrows. The information crosses the pipe but it must be understood by the IoT side and the WoT side. Having well defined the problem, also graphically, next sections will properly list all the constraints of the given problem and propose a solution that fulfills them all.

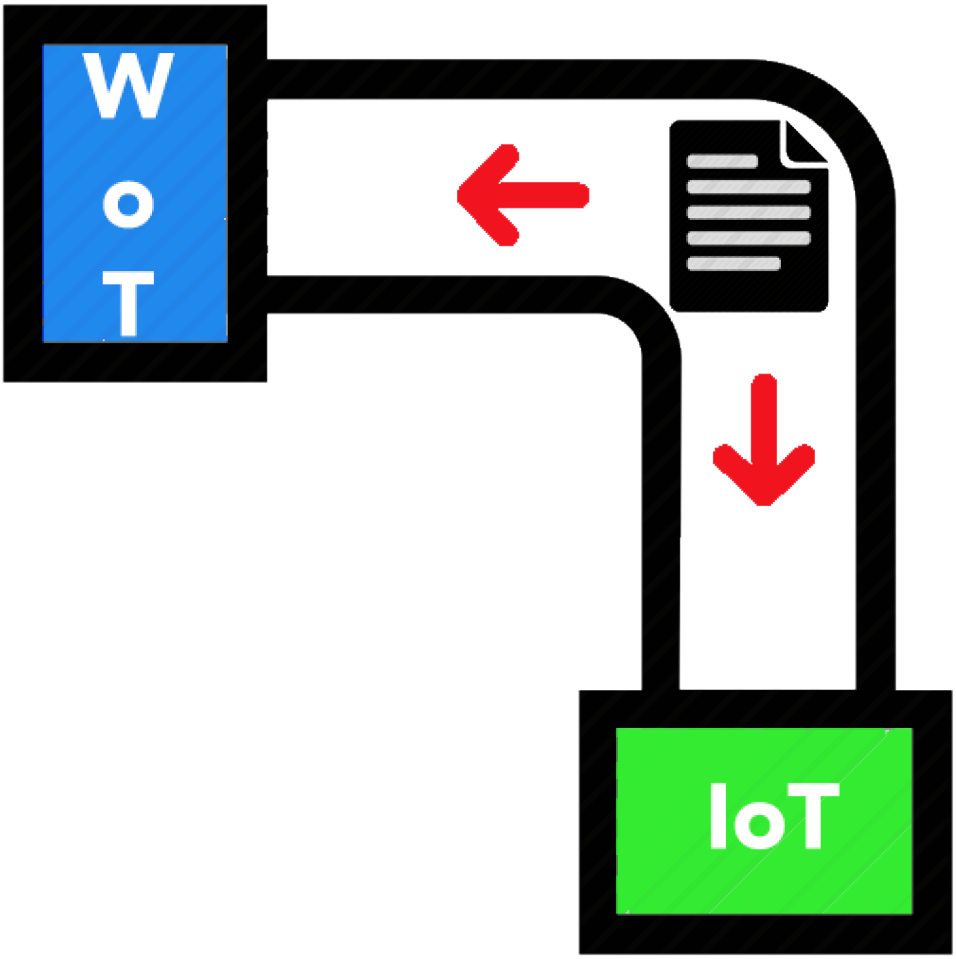

Figure 6: Paragon between my problem and a pipeline

3.4 Constraints

In this section I would like to list a set of constraints for the defined problem, that become requirements that the solution must meet. The section could be divided into two parts, the first for the requirements of the container, the second for the ones of the syntax, but the two sections are closely related so here I preferred to keep the two parts united, indicating wherever

possible which of the part that particular constraint acts on.

Here is the list:

- **M2M communication:** M2M communication is defined as a communication in which the two interlocutors are not humans. It is a communication completely handled by machines and computers [13]. It can be considered one of the fundamental enabling technologies of the Internet of Things, it permits object to communicate without humans being involved. This type of communication has to be stricter than M2H: the subject reading the content is a computer, it has no a semantic idea of what is written, it can, at most, understand the syntax. So a clear, defined syntax with a well fixed structure must be set in order to make everything understandable to a computer. Obviously the machine has also to able to write the code or the file that will be sent. Anyway, the problem of what is understandable for a machine is a common "problem" to all fields in computer science and engineering.
- **M2H/H2M communication:** I grouped these two constraints because they are similar, they include some similar behaviors, in fact somehow the solution I am writing has to reach at least a human being, the owner of the smart space. In order to check what is going on and control the behavior of smart devices the user must understand what he sees on a web application. This constraint can be considered "indirect" for my work: it must be somehow adapted to a UI, the one of a control center application developed on top of the file I will provide. So it means that my work does not reach the user directly but it means that it should be possible to easily embed my output into something the user

can understand without too much effort. That implies I have to use variables and values in a univocal way, in order to permit the web app a clean and meaningful translation. These considerations are valid for M2H communication. Regarding H2M communication the user must be able to easily impart commands and the inverse procedure of translation needs to be as simple as possible. So the user has some constraints: he cannot use free natural language, the problem is still the same, computers are not able to semantically understand natural language.

- **Easy readable:** What I am going to define is a sort of description of a smart space that must be readable by machines and by humans, as explained above. This does not imply the easy readable property: a thing can be read by human or machine but with difficulty. The desired property here is the fact that both humans and machines will be able to read the content of the smart space without much effort. This streamlines all the procedures. A machine that has to execute a command or retrieve information has to do it very quickly, if something must be notified to the user, the content to (write and) read must not be an obstacle, in terms of speed or resources spent to assure the right communication. For humans, we can distinguish two categories, developers and users. For users the situation can be mitigated by the presence of a web application between my work and what he sees on the web pages. I will provide a description of objects which will be embedded into third-party applications. In this way the user does not see directly what I have done: developers are the people who use data provided by my system to construct a web application. Two are the main reasons they need understandable data, developers

need data to be easy readable in order to easily integrate them in applications and also to debug the system. The error can be in the written code, but the source remains my work, having it clear in mind, it can help to quickly debug.

- **Reconfigurable:** we are talking about smart spaces, composed of smart objects, they are usually relatively small and so they can be moved easily (some of them, though may be bigger, such as industrial machinery). Another issue can be the fact that, being powered by batteries, objects can turn off due to low level battery status. The two situations I have just explained appear to my system as a change of the smart space: it is necessary to reconfigure it, in order to output the current situation without having "dead devices" or moved ones into the described space. Moving a device, we can have the situation where a sensor, for example, is moved from a room to another, so the total number of devices is not changed but the configuration is. My system must be able to detect the "loss" of one of them in some place and the "gain" of the same one in another.

- **Fast:** being fast is one of the essential constraints of my work. I have just stated that the reconfigurable property is desired, to handle the change of the spaces my thesis is going to describe. But how fast my system reconfigures all the list of objects becomes fundamental. No latency is permitted. Think about an alarm or a thermometer which goes out due to the level of the battery: the user must be notified and aware as soon as possible. The user cannot be kept without any protection for his home or without heating in winter. There should be a mechanism, a sort of listener, that understands when something changes and reports quickly to the third-party application the user uses.

- **Lightweight:** Another constraint to our system is the fact that whatever system I choose to be the solution it must be lightweight. This is needed because it goes on the web, the third-party application server requires information from the objects that it collects, packs it and sends to the user. The user can see its configuration through a web browser that, as a client, downloads the required page. If we want information to be notified quickly, information should necessarily be lightweight. So my solution has to take into account the fact we are talking of the Web of Things, and the word "web" implies some constraints by definition: the one we are interested in here is the latency and the transmission time, due to the physical limits of the cable or the materials that constitute the network infrastructure.

- **Modular:** the language or type of file chosen must be modular. The faced problem is modular itself: it is necessary to group lots of IoT devices into one singular description. It must be a description of the smart space: it is composed, for instance, of rooms. And also in each room there can be a hierarchy of devices, in addition the objects with a gateway need to be described as coupled, and only one gateway can have multiple devices to control. So the solution also has to be modular, it must be possible to write different configurations. It is necessary to define some building blocks, that can be placed almost everywhere in the hierarchy. The length or the number of levels that can be nested should be variable, each smart space has different peculiarities: from a city to a living room. My solution must be universal: a thermometer, for instance, can be put in both of the listed

places without specifying where it is and without changing internally its properties due to the position.

- **Complete:** the chosen solution must be complete, in the sense that it must be able to formally describe every configuration. It should be possible to describe any kind of object, it has to own each kind of smart object present on the market, with their properties, status and commands. It should also be possible to insert each object in the hierarchy almost everywhere, even if, obviously, some rules are needed to assure the right syntax and the right structure. A "language" that is not complete cannot be accepted, it must be able to describe any given smart place, my solution tries to be a unifying method for developers that can count on it as a consistent description of the space, the "source" of what they have to output to the user.

- **Extensible:** the solution must be extensible, because every day new objects become smart. With the technical innovation it will be possible to increasingly miniaturize the chips needed to offer a connection, to become smart. My language needs to be extensible, when a new sort of product comes on the market my language has to define a standard for it, according to the object's properties, states and features. It must be always possible to do that, there should not be limits to the number of building blocks my solution is going to define, because we do not know today where the "smartness" of objects is going to go, we do not know the level of miniaturization and power saving that will be reached in the future. Being extensible means being a valid alternative also in the future, writing a solution that only fits "today" objects without the possibility to expand limits my work to

be just a "photograph" of the current state. With the growth of the number and types of smart devices, the extensible property becomes fundamental to obtain another property: completeness.

The listed requirements, as already told in some of them, are, sometimes, general, in the sense that they have to be respected for the final product: a global and complete structure that starts from the smart object and arrives to the user's web browser. This is because the problem I am facing is very big and complex, and it is transversal to the existing technologies, so the whole system must work properly. Keeping in mind what I have just stated, some of these constraints become fundamental requirements that my system must meet, some others, for instance the M2H communication, are only indirect or partial requirements: my work has to be clear for the developer, who knows what variables are, how a file can be accessed, parsed and processed but it can be less clear to an average user who wants, out of curiosity, see what is under the hood.

In the next chapter I am presenting my idea, the so called solution to the given problem, explaining what I have done and my considerations about the situation here faced. I am defining both the container and the syntax, or better the structure, of my work; these two parts are the fundamental ideas that are the bases to construct my thesis.

CHAPTER 4

PROPOSED SOLUTION

In this chapter the development of the solution will be reported step by step, with a full use case. The chapter starts with a list of solutions already developed, already on the market and the differences between them and my work. The remaining part is composed of two mains sections: the first explains the choice of the so called *container*, the architecture, the type of file, etc. It lays the foundations and describes the limitations for the second part: the definition of the *syntax*, or better the *structure* of what is written inside the container. The two parts are closely related, therefore their relation was taken into account when I made my choice.

The real implementation of the valid solution is left for the next chapter.

4.1 Existing Solutions

4.1.1 Commercial Solutions

A lot of companies have already tried to create a solution as "universal" as possible, but as their main aim is profit they are not really interested in creating a "real" universal solution. Their main goal is including more and more smart objects and making them compatible with their systems. The Web of Things for them is not an arrival point to reach, but rather an obstacle for increasing their profits. Companies are not innovating in the sense of creating a common framework that every vendor can use. They are trying only to enlarge their sphere of influence, not to be a real change in the world. This means companies aim to keep the IoT

as final product to offer to users, doing it with some partnerships with the companies which produce smart devices. It is only a matter of time before some exclusive contracts will be signed: if some producer becomes a supplier for only one system, there is no gain for the community, a user has to choose which objects to buy depending on the system he has maybe already chosen and not depending on the specifications or the features of the device.

But they have a different purpose, they want the devices to mount an operative system, a protocol, etc. This still remains in the situation explained in Chapter 2, a war fought by different companies or alliances, leading the world to a myriad of different and incompatible solutions: it is more an attempt to enlarge each own IoT than the creation of an open solution for a common widely resource like the web. Now it is time to start with my reflections about the given problem comparing the available paths to reach a solution.

4.1.2 W3C proposal: pros and cons

The W3C, during the years, has already faced the problem of lack of standards in the highest layer of the structure. So it has published an unofficial draft from which the idea of the thesis has been taken, discussed and changed in some parts, according to my personal needs and the thoughts discussed and analyzed [14].

In particular I took from the W3C idea two main things: firstly the idea of a REST service, a "consumable" solution. In particular I found the idea of a "semantic solution", in order to use the full potential of linked data, an innovative and future oriented solution. Secondly I shared the "thing description" the W3C proposes: in particular it divides the capabilities of a smart object into three different interaction patterns: *properties*, *actions* and *events*.

- **Property:** it provides readable and/or writeable data that can be static (e.g., supported mode, rated output voltage, etc.) or dynamic (e.g., current fill level of water, minimum recorded temperature, etc.).
- **Action:** it targets changes or processes on a thing. These changes take some time to complete (i.e., actions cannot be applied instantaneously like property writes). Examples include a LED fade in, moving a robot, brewing a cup of coffee, etc.
- **Event:** it enables a mechanism to be notified by a thing on a certain condition.

It is fundamental to point out that I am not implementing what W3C is proposing without making reflections, changes and thoughts to the consortium's draft. W3C is nowadays an essential component to standardize protocols and ways of communication on the web, it is composed of web experts with long standing experience; it can be considered "somehow" natural that an idea for a thesis comes from someone or somewhat having more influence on the existing technologies than a single student.

Having specified that my work will anyway introduce something new respect to the draft now I am going to list briefly the innovations compared to W3C ideas. Innovations, more generally, are explained and presented in the following sections and chapters, without making a schematic comparison continuously, because they are only a part of a bigger and heterogenous work.

Firstly, my idea is to provide the developer an indirectly the final user with a description of the whole smart space: a smart space is a very heterogeneous term, it can be outdoor or indoor, composed of different zones or rooms, etc. So my idea is to face all these categories to

create interchangeable blocks of my JSON-LD document. In each of these zones, maybe, the same type of device is installed, so the problem to uniquely identify a smart object is obviously one of the fundamental requirements and challenges to face. Also the discovery of devices can be something to focus on: the object in the right place is another challenge to be overcome to have a full working system. Secondly, W3C has not specified "how to use" what they are proposing: I decided to put the entire system on local host for some cases and online for others, recommending the developers that will work on the control centers to do the same. Even if there may be some exceptions. The advantages of putting the "smart world" on the web, creating the WoT, have already been explained and presented. The advantage of putting the system on local host is the fact that access is restricted without using any password, it can be useful for example for "public" smart spaces: being in a hotel room I must have access to all the devices in my room but not to other devices or to the centralized system in the hotel. So a customer can enter the room and control everything without authentication. On the other side the owner of the hotel needs to have a complete vision of what is happening in the structure: in case of an emergency it is necessary to control everything remotely, making sure that who is using the system is allowed to do it. This simple case ranges among all the shades the problem can have. Maybe other minor features and changes to the problem are not listed here but the whole work is explained in the next sections.

4.2 General idea

To better explain what I consider solution for my work it is important to understand the playground to my work. As said in the previous chapter, 3, I am trying to fulfill the "horizontal"

part of the connection in the stack, staying in layer 7, the application layer. Using already developed and operating tools, and respecting all the above listed constraints I am going to make the communication between WoT and IoT "HTTP ready" devices possible.

In order to understand what is needed and how it is possible to solve the problem it is fundamental to understand the type of stack and the network structure we have to face. Only having clear in mind the problem and the structure it is possible to find the best possible solution. The network structure here is composed of three main actors:

- the user's client, which can be an application developed in several ways: in a web browser, a desktop stand alone application or a native mobile application on the most popular mobile operative systems.
- the web server, which is the real application. It contains the logic and all the controllers needed to handle the structure of the smart space it is going to describe. It is on a machine belonging to the same network of objects or there must be the possibility to reach it remotely. It is called server but in reality it has a double behavior: it is the web server for services which want to use my framework, my system answers to calls of third-party applications with the needed information. On the other side my system depends completely on IoT objects, they are the prime sources of data I have to manipulate: in this case my system is working as client, asking for data to different proprietary solutions I wanted to include in the framework.
- the objects, the ones which have an end point online, reachable with the HTTP protocol over the internet. They behave as servers for my middleware, which periodically asks data

or sends commands. They can be considered "servers of my server" for their behavior, even if they do not have those features. They can also be, under another light the "model" in the MVC software engineering pattern; they are not just a classical database of inactive information stored but they are alive, they can trigger events or can be subject to the external environment.

Having defined the three main actors of the problem I am facing, the next step is understanding which role they have and how they interact. In order to give an easy overview I am reporting Figure 7 which focuses on the structure and interaction of the three components identified. The figure shows clearly how the whole system works: the solution, as said, is divided into two parts. One has a "client" behavior and queries the IoT world, obtaining as answer what I can consider "raw data", they are data formatted with a vendor's solutions or not uniformed standards (they are all different, but I used the same icon for raw data of different objects). The second part, the "server" is responsible for outputting uniformed data to all the clients, belonging to the web and to the WoT in particular, connected with my framework (here the document icon represents same data to all the clients). Inside the solution data are manipulated, they are seen as variables or collections, so I represented them as "binary data". This representation does not go into much detail, there is no representation of components responsible for translating data, the ones responsible for connecting to the IoT world or the ones which notify the clients in the WoT world. That kind of representation is left for the next chapter where a case study with its implementation is deeply explained. The general idea to solve the problem is the core of the thesis, while the technique chosen to connect to the IoT

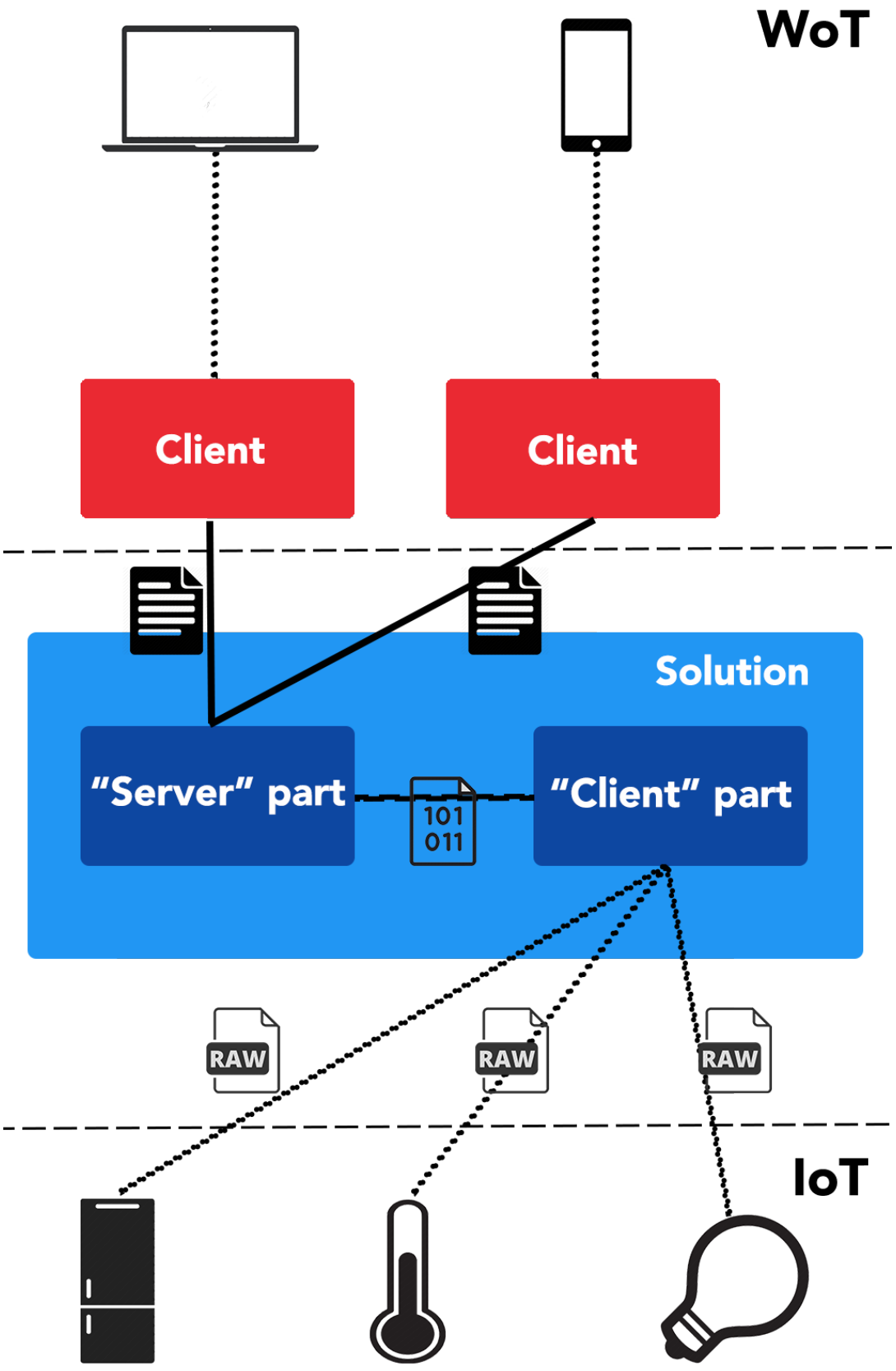

Figure 7: The architecture of the proposed solution.

objects is a matter of "low level" decisions. It does not mean it is not important, but in order to make technical choices it is necessary to first have a clear idea of the problem to be solved.

It is pretty clear that my system acts as middleware: it is something which is introduced between parts to assure the right communication and exchange of data. The solution is, as already pointed out, divided into distinct parts: the choice of the infrastructure, renamed "container" and then the definition of what flows inside the infrastructure. The choice of the so called container is very important: it puts boundaries in the structure/syntax definition, so it is essential to elect as solution the "right candidate". I am searching for technologies that permit clients and servers to communicate, operating on the same network: I am supposing to construct a framework to describe a smart space, it is not fundamental to have the framework online. The translation process between a unified language and the myriad of protocols used by IoT objects can take place locally. When the process is finished and the description is ready it is possible and also advisable to have an end point online, from which different third-party applications can take data to control the space or offer the control to the users.

Referring to the previous figure, it is possible to identify the two components just described. The "container" is what permits data to flow from a world to the other and vice versa, so it is composed by the black arrows: the infrastructure connects the objects to the framework, it elaborates and makes uniform data inside the system with ad-hoc functions and methods and finally it sends data to the client. All this process, comprehensive of the creation of the channels and the maintenance of them active can be considered the "container". What is exchanged in the channels, instead, is some files containing all the needed information. The type of the file

and its syntax are the most important choice to take in my work. It is the second part and the core of my work. In particular, in the figure the so called "syntax" is represented by the three icons of files which are put near the arrows of the infrastructure: data retrieved from objects are "raw data" because they are not unified, as already explained; inside my system data can be binary, while the output of my system represents the goal I am trying to achieve: a general language able to describe neutrally smart objects and, more completely, smart spaces.

Putting together the two parts we have a complete pipeline which starts from the IoT world and ends in the WoT world, so we have fulfilled the requirements proposed in the previous chapter and in particular in Figure 6 where the example of the pipeline was already proposed. Theoretically, a working communication between two completely different worlds has been created.

I would like now to list the goals that my work has to match, in order to be a valid proposal for uniforming the source, the "downside" of the application layer. These goals are not to be intended as set in stone, they are the general motivation that leads to construct a prototype of the proposed software architecture.

According to my thoughts during the development, it is possible to identify the following goals:

- The middleware must work without any proprietary application: it has to interface itself to the upper layer without installing any application of any vendor of the owned object. It must be completely neutral to the market, simply classifying objects for their features, capacities and goals, not for the brand. It is the fundamental requirement to create

heterogenous applications and to separate the various closed solutions of today and an open solution for everyone in the future.

- The middleware has to simplify the life of a developer, he should not have to worry too much about the layers, he should be able to fast prototype. The developer should see my framework as help for his work, a source from which he can obtain useful information in a simple and immediate way. The idea is to provide a RESTful service, from which it is possible to obtain a JSON file with HTTP requests. The file must contain the description of the whole smart space requested.
- The middleware should offer the user the possibility to access directly to smart objects: reading states, configurations and seeing what is not working should be easy. This does not exclude the presence of one or more third-party applications to configure or communicate with the smart space, but it gives the opportunity to the user to see directly what is happening, bypassing the standard web application used to manage the devices. This constraint is fulfilled by my web application that offers a small interface to the user. In particular, it is discussed in the next chapter.
- The information output by the system must be clear, both for developers and users: a smart space must be defined, containing rooms or other architectonic components that can be considered subspaces, a hierarchy must be defined to group the objects in the right way, each one in the correct place; properties, states and commands must be well defined for each object. In case of error messages must be shown, they have to be clear and self explanatory, in order to permit to solve or debug the problem rapidly.

The next sections contain all the steps necessary to have a full working system. Firstly the choice of the "container" and the consequences of the adoption of that particular solution are explained. A complete overview of the chosen syntax is given then a step by step analysis is performed, from the general ("smart space" entity) to the specific and the detailed ("smart object" entity).

4.3 Babel Framework

4.3.1 Choice of the *container*

First of all, the definition of the container is the starting point of the solution, the one which puts some boundaries on the definition of the syntax. With the term container I am identifying the architectural choice of the system which permits the exchange of data, properly treated. The container must be a set of elements which once combined permit to obtain a working system and a system able to respond to the requests for the given problem. In this case the container must be an architecture which permits to send and receive information, in form of file, containing the description of a whole smart space or actions to perform on objects. The actors, the "talking parts", are two different systems, probably developed in completely different ways. Their main purpose is not interacting with each other. In addition, the system must run online because the "talking parts" can be physically in different areas, separated by firewalls, routers, etc.

The most frequently used technology to solve this kind of problem is what is called Web Service. In particular, among Web Services the REST paradigm is the one which better fits the constraints of the problem. We have already discussed in an appropriate way the technicalities

regarding the REST paradigm. Having defined the requirements and the constraints of the problem, the choice that has been made is the REST paradigm. It is most widely used system to allow to interface without worrying about the distribution or vastness of the network. The technology itself has been already explained in the background section, here I would only like to motivate the choice of the solution, briefly highlighting the pros but also a few cons.

- REST is generally easy to use and is very flexible:
  - No expensive tools required to interact with the Web service
  - File format agnostic
  - Small learning curve
  - Efficient (REST can use small message formats)
  - Fast (no extensive processing required)
  - Close to other Web technologies in design philosophy
- But REST has also some cons:
  - REST requires use of HTTP
  - It assumes direct point-to-point communication
  - Not standardized

Taking a look at the points listed above, I would remark how the pros of the REST paradigm match some of the requirements already listed in 3.4. Efficient and fast were two of the main characteristics needed by our system. Then the fact no expensive tools are needed is another

important point developing a sort of framework for third-party applications. Another advantage of REST is being "language independent". In addition it can be chosen only as "container", leaving the choice of the type of file free, even if we stated that it is better to limit the formats of file sent on the web. It is important to underline REST is not independent of HTTP, however this can be considered only a minor problem because we are talking about the Web of Things, we are supposing to put everything on the web, and the HTTP protocol is one of web's fundamental parts. The only point that can tilt in other directions is the fact that REST is not standardized but it is a set of guidelines, not a well defined structure [15]. This has a double side effect: on the one side the developers can use "fantasy" to write down code as they prefer, exchanging custom data with custom functions, but on the other side there is no a univocal way of work and this can create confusion and ambiguity: two systems which do not "know" each other have no chance of automatically talking and maybe the response of one is not what is expected or the access point of the APIs is different from the one reasonably chosen. In this case it is necessary to "teach" the developers who will write third-party applications how they can make their softwares talk with the middleware. In particular, this sentence is about the other way of communication: when a user decides to start an action on an object, it is necessary for my middleware to be notified. The message the various third-party control centers have to sent to my systems must be well defined, a unique identifier for each object and for each action must exist. This identifier is the key to know what to do and on which object of the considered smart space.

As the reader can guess, the choice for my system is the REST paradigm. Despite the fact

it is not standardized, REST provides a set of easier "rules" to build and develop a framework easily regarding the part of the connection and reliability of the resource. The other choices can be discussed later and they can be considered more "matter of style" than requirements to satisfy.

A REST architecture can provide a complete channel for "talking". In reality, REST does create a durable channel as Web Sockets do because REST is stateless, but having the URLs which represent the web "image" of the resource, in this case of smart objects, it is possible to interrogate them periodically or to have a subscription when something changes and also it possible to send them commands, if it is correctly configured. Also it is neutral towards underlying levels: if a HTTP end point is online, for REST this is enough, each client can retrieve and send information to that web address. It is not necessary to satisfy other requirements in the lower levels to construct a RESTful service.

So what composes the container is a REST service, a paradigm which permits to exchange data only knowing an online end point and using the HTTP protocol. This is the skeleton of the solution: a system which creates a consumable resource and offers its image online, available at any moment. For what we have decided until now the solution is a middleware, put between the application layer and the transport one, and it offers a consumable copy of a resource to the third-party applications which can require it and interact with the whole system through the HTTP protocol.

The not-standardized part is what composes the resource made available by the RESTful system. I am going to define it in the next subsection of my thesis, firstly choosing the type

of file and then defining what is missing. Basing my own architecture on the HTTP messages and REST paradigm, the core of the RESTful services, the choice of the file exchanged among systems is still undecided, and the universal syntax I propose is still to be created. Obviously the choices can be considered strictly related, in particular the choice of a certain type of file forces another choice in cascade: the one regarding the syntax. In fact it must respect rules, technicalities and conventions due to the type file chosen. Then, "inside" the syntactic rules, I have developed a uniformed language to describe the smart entities, objects and spaces in particular.

4.3.2 Syntax of the solution

As previously explained, the types of files which can "flow" through a REST infrastructure can be many. I chose to use JSON file format, principally because it matches almost all the formal requirements my application has to have to solve the given problem (section 3.4).

JSON format is both *M2M* and *M2H/H2M* constraints compliant, and also *easy readable* (and writable). Having these properties we can also state it is *fast*: both machines and humans are quick in doing actions on it. It is also *lightweight*, in particular respect to XML, being less verbose and less depending on open-closed tags. Finally, it is *modular* and *extensible* with the fundamental possibility to nest levels.

In this subsection I am going to define step by step the syntax of my solution, explaining with listings and images all the elements.

An important and remarkable reflection is needed here: unfortunately it is not possible for me alone to define every kind of smart space and smart objects existing nowadays. I will

provide the reader with a sample of what I am doing and what is the developed work. The examples are obviously made to cover all the different shadows of the matter, in order to give a complete idea and a full functional system.

Context: here I would like to explain the keyword @context and its meaning in my work. As said, @context must define the terms used in the document, so they change for each particular smart space considered. Here I am reporting an example, that could maybe change on the proof of concept provided in the next chapter.

As said, the @context maps the term used in the document to an IRI that specifies its semantic meaning for the web. So, the first problem to address is the fact that there is no universal description of objects or spaces on the web: no one has semantically defined these concepts, so I am going to firstly point out what are the theoretical features these components must have, and then use a fake IRI: starting from `http://schema.org`. I am going to use relative IRIs which do not really exist. The website *Schema.org* was launched in 2011 by Bing, Google and Yahoo! to have online a set of schemas, shared by everyone, for structured data markup. It contains a sort of description for objects or other entities, descriptions useful to the web processors which are enabled to give " semantic meaning" to entities and web pages. The vocabulary is open and shared among the community, each website can refer to schema.org to give a semantic idea to its processor. Also the development of a new term in vocabulary is open, with an open community process. I am reporting here the example definition schema.org has given to the "Movie" entity, that in my opinion lacks information, but it is the one provided

```
{
  "@context": "http://schema.org/",
  "@type": "Movie",
  "name": "Avatar",
  "director":
    {
       "@type": "Person",
       "name": "James Cameron",
       "birthDate": "1954-08-16"
    },
  "genre": "Science fiction",
  "trailer": "../movies/avatar-theatrical-trailer.html"
}
```

Figure 8: The architecture of the proposed solution.

as example also on the Wikipedia page of schema.org. The real "Movie" entity is much more complete, this is only a short example to give an idea to the reader.

Thinking of what I have to do, here is my task: I have to define a common language spoken by every smart object, but schema.org contains all the information and fields to properly describe an entity. But I cannot define myself entities on schema.org, so I am going to explain what I have done with a sort of "use cases", putting real example values to the terms of the JSON-LD (note that the terms of the JSON-LD are the fields of schema.org, there is a 1:1 mapping between them).

Referring to the schema.org division, it is organized in a hierarchy, all the terms are children of "Thing", the root, and then nested levels are possible. Fields are inherited, so every child of "Thing" has all its fields plus the specific ones added. A more nested level contains all the

fields of its fathers plus the local ones. A leaf of this tree can have more than one path to return to "Thing", this means that multiple definitions for a semantic object can exist.

So in the following example, as anticipated above, I would like to map, inside @context, some local terms to "fake IRIs", which will be indirectly explained in the rest of the document with the values assigned to left terms of the JSON.

In the next sections I am giving an overview of all the allowable values that can be assigned to every significant term of my work.

Smart space: the smart space can be considered the root of my document, it is what I am going to describe. Knowing the structure of the JSON-LD document, the smart space is not actually the root of the document, it is put just after the term @context.

Having clarified the role the keyword "smart space" has, now I am going to define the possible values it can have. I principally divided smart spaces into two categories, each has 2 possible values, for a total number of 4 possible combinations. The first category divides *private* and *public* spaces, the first value includes spaces which are private properties, such as homes. The second value can be applicable to stores, retails or public spaces such as parking lots, parks, undergrounds, etc. The second category I have identified is the differentiation between *outside* and *inside* spaces. In the following table I am going to report the combination of the two categories with examples of smart spaces that belong to one of the four available possibilities.

The importance of this division, in my opinion, comes from the types of elements which are included in smart spaces: a private indoor space can contain rooms while a public outdoor one such as cargo area can have different zones like offices, real cargo part, etc. Due to the

TABLE I: TYPES OF SMART SPACES, DIVIDED BY CATEGORIES.

	Outside	**Inside**
Private	Garden, Cargo area, etc.	House, Industry, Office, etc.
Public	Park, Zoo, Golf Club, etc.	Retail, Hospital, Post office, etc.

heterogeneity of the sections of smart spaces I made this division.

The only category, in the schema.org hierarchy, which can somehow look like the one I am defining is "Place". It contains a list of different places, without dividing them as I previously did. In my idea it is good that places are already present in the hierarchy but they are not ready for the WoT. As said, multiple path for an object can exist, so my idea is to define the term "Smart Space" as direct son of "Thing", replacing or flanking "Place" and taking all its fields (which can be found in "Place" at `http://schema.org/Place`). "Smart Space" differs from "Place" by the addition of the two properties above defined. The four boxes in Table II represent all the possible values. So I am adding two fields that have *boolean* values: *"Private"* and *"Location"*. If the item has the "Private" field set to *false*, anyone can access and control the smart objects there through the web application. If the value is set to *true*, other three fields are required to complete the semantic description: *"Owner"*, *"Access"* and *"Access description"*. I think the first must be a "Person" type, defined by schema, even for companies someone can be identified as owner of a space, or if the town of Chicago wants to restrict some areas putting them private it has to "elect" someone as owner. The second field "Access" is simply a boolean value that indicates if the current user has access or not. This value needs

the concept of authenticated user and relation to the owner to be used. In my system the field "Owner" is defined for future adjustments but not really in use. It will be possible to "stop" the discovery of a smart space if the user has no access and the document will have no other nested elements from now on. The third one, "Access description" is simply a "text" that the owner can fill in order to give instruction to how to obtain the access.

The second field "Location" is "text" based. I personally considered only two values it can assume: "outside" or "inside". It is simply a description which can help a developer or a user to understand from now what he/she is dealing with. Also it can be extended with other values, for example a hybrid solution like "Veranda". All the elements already present under "Place" are put under "Smart Space", maybe adding some others under the various values the fields have. The more nested levels of my architecture, level which contains "Rooms" and then "Smart objects" are explained later. Now I am reporting a figure of an UML diagram which represents a portion of a hypothetical tree that schema.org can assume (red is the added part). Due to problems of available space it is not possible to report all the children of "Thing".

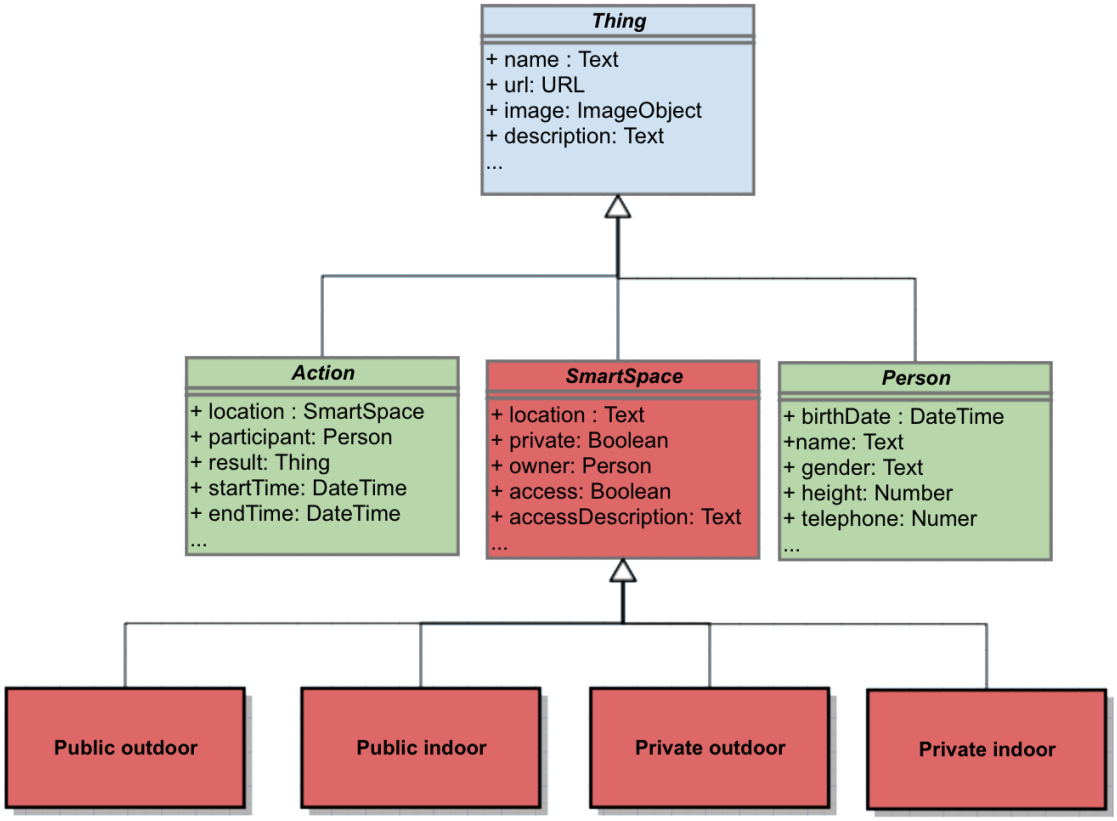

Figure 9: A hypothetical UML for smart spaces (1)

Note how the last level of the tree in the above picture is composed of rectangular shapes with only one box. This is because they are not real categories such as "House" but they are containers: for example, "House" stays under "Private Indoor".

I would like to briefly report the structure of "Smart Space" in a table (Table II), in order to have it well schematized and easily accessible from the index (the first tuple indicates that all the fields taken from "Place" are not reported).

TABLE II: "SMART SPACE" CATEGORY FIELDS

Name	**Type**	**Description**
...	...	...
location	Text	It indicates where the location is, two are the recommended values "Inside" and "Outside".
private	Boolean	If false it is accessible by everyone, if true there may be policies to access the space.
owner	Person	Subject to *true* in "Private", it contains the semantic link to a "Person" who is the owner of the space.
access	Boolean	Subject to *true* in "Private", it indicates if the current user has access to the smart objects.
access Description	Text	Subject to *true* in "Private", it is a description of how to obtain the access.
contains SmartSpace	SmartSpace	It includes smart spaces in others.
containedIn SmartSpace	SmartSpace	Reference to the smart space which contains the current object.

Before defining next components it is important to focus on the relation among the entities I am going define and modify. "Place" declares a field called "containsPlace" of type "Place", here I am only redefining it as "containsSmartSpace" and the type is "Smart Space" or, better, one of its sons. Obviously the same argument is valid for "containedInPlace" which becomes "containedInSmartSpace".

Inside smart spaces: this title identifies what is inside smart spaces. As said all the entities which are children of "Place" are now under "Smart Place" and they are somehow

partitioned, according to the type of smart space they belong to. So we can transform the rectangular shapes of figure Figure 9 to a myriad of places, including houses, retails, industries, etc. Now I am going a step further: all these spaces are composed of smaller zones, the ones which contain the smart objects: these zones and the "verbs" to link all the components need to be defined. Also here, as already anticipated, it is impossible for me to define each type of zone existing in the world. I am going to define some useful examples, to give the reader the opportunity to understand my work, which should be obviously completed before being operative.

First of all, it is necessary to add a field to each of this myriad of places, a field which links the entire place to smaller zones. The field is called *composedOf* and its type is a new category I am going to define called *"Zone"*. In the next figure I am reporting one example, house and the new field.

TABLE III: "HOUSE" FIELDS

Name	**Type**	**Description**
...	...	...
composedOf	Zone [, Zone]*	It indicates which zones compose a house.

The field has been defined with type "Zone", but it is not defined inside schema.org, it is defined by me. It is common category to all the children of "SmartSpace". "Zone" can be

considered a "super type" of the other entities such as "Room", "Aisle", "Garden", etc. I chose to create "Zone" as a direct child of "Thing", so it is "in parallel" with "Smart Space". In zone some fields are defined, in particular the ones which are common to all its sons. Having defined previously the field "ComposedOf", it is necessary to define its opposite called *"InsideOf"*: a field of "Zone" and its type is "SmartSpace". Another useful field can be *"Smart"*, it is boolean and indicates if the zone contains smart objects, it is not mandatory that every place contains them (In particular in the case if "Smart Space" substitutes "Place" in the hierarchy). It can be helpful for discovery systems which will configure the entire space. Another desired property can be *"Dimension"*: it is a number, square meters of the area, it can help for the range of discovering or in case additional hardware to install is needed. It is obvious that an area can have multiple purposes and properties: in this thesis I am defining the fields that are useful for the goal, not every one. For example here these three are enough, but it is possible to add other fields for other purposes.

Figure 10 reports the new hierarchy. Notice that the double blue arrow is the representation of the two fields, one opposite of other, it is not real part of the tree. Finally I would like to formally report the fields I have added to "Zone" in a table. Now, having defined the head of the category, "Zone", it is time to give examples of its sons. They are not particularly complicated or full of properties, they are a division which helps to understand how smart objects are placed: imagine in a house there are many smart lamps, in case of a breaking it is more useful to have an immediate feedback from a particular room, rather than to retrieve information only from

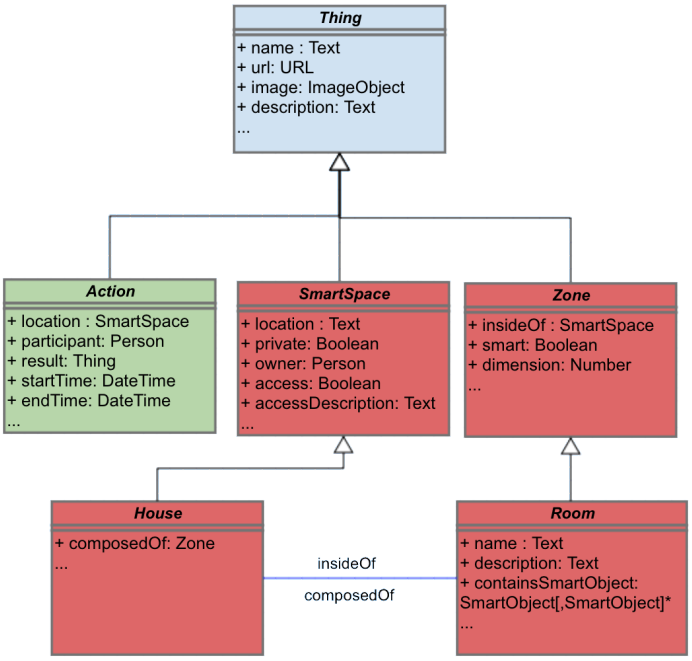

Figure 10: A hypothetical UML for smart spaces (2)

an ID or a production code. The presence of the entity "Room" simplifies the management and maintenance of the application (control center) and the real hardware (smart objects) installed.

It is impossible to give an example of everything but for example I identified the following zones: "Room", "Living Room", "Kitchen", "Aisle", "Basement", "Gazebo", "Garden", "Roof", "Balcony", "Workshop", "Showcase", "Office", etc. The decision to split these elements was taken thinking of the different and unique features they have. For instance, I have considered creating "Room" as generic as possible, in case of ambiguous entities, because they are pretty much the same from the WoT point of view. On the other hand I have well defined "Kitchen", "Living Room", etc. To give a concrete example I chose the most common exam-

TABLE IV: "ZONE" FIELDS

Name	**Type**	**Description**
...	...	...
insideOf	Smart Space	It indicates to which smart space the zone belongs.
smart	Boolean	It indicates if smart objects are present.
dimension	Number	It describes the dimension of the zone.

ple: "Room". I provided "Room" with a "Name" field, a text field which describes briefly the room, maybe specifying the type. Another field, always textual, is "Description", it is a more extended description, with some useful information for a user. The core of the question is the field which links the zone to the list of the smart objects installed: "ContainsSmartObjects" is the name, while the type is one or more "Smart Object" a category defined in the next paragraph. Obviously the presence of this property is subject to the value *true* in the field "Smart", inherited from the "super category" "Zone". As before, it is possible to add fields for other purposes but for the goal of this thesis it is enough. Finally, the formal table is reported.

The division of these attributes can be discussed, it is not set in stone, but I thought that it is better to distinguish them in this way. For instance, the name can be maybe useless for the only garden of a house or the field "ContainsSmartObject" which is absent in case of "Smart" set to *false*.

Smart Objects: until now we have spoken about the environment, the smart space, the place where the objects are. It is time now to face the solution proposed for the core of the system. The core is composed of objects. As previously described they are heterogeneous and

TABLE V: "ROOM" FIELDS

Name	**Type**	**Description**
...	...	...
name	Text	It specifies the name and type of the room.
description	Text	It is a more extensive description of the space.
contains smartObjects	SmartObject [, SmartObject]*	It contains the list of the smart objects.

different for capabilities and features: for this reason, as done before with "Zone", I defined a super category called "Smart Object" which has some features common to each of them. Then I will propose an example of one of them and finally I would like to end the chapter with a real use case, which will be the one implemented in my work and described in chapter 5.

The category "Smart Object" is a direct son of "Thing" and, as it happened with "Zone", it is in parallel with "Smart Space" and "Zone". It is possible to identify some common properties to all objects: first of all it is necessary to define the opposite field of "ContainsSmartObject", the opposite is called "ContainedInZone" and its type is obviously a "Zone". It is mandatory that an object has a "Zone" to belong to. The blue arrows in Figure 11 reports the fields which link the different categories created while the rest of the figure shows the evolution of the tree of hierarchies in schema.org (Notice that the blue arrows are possible also on the three super categories, they are parallel for schema.org but they have somehow a hierarchy of inclusion).

Back to the common fields which is possible to define in "Smart Space" one can be "Name". It is textual, it is a name which can be chosen by the user, such as "Federico's Lamp" for

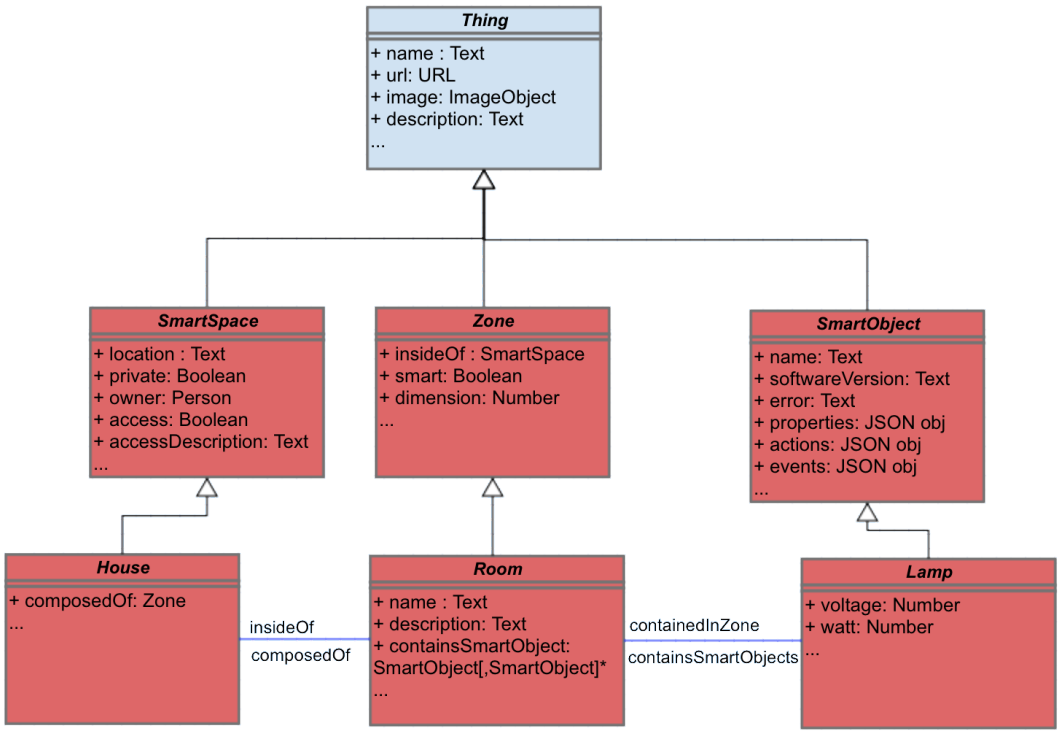

Figure 11: A hypothetical UML for smart spaces (3)

instance. The "ManifacturerName" textual field can also be necessary, such as "ModelNumber" the unique identifier of the hardware, "IPaddress" which is one of the fundamental requirements of my work as previously explained, "SoftwareVersion" which can be useful for updates and maintenance, "Error" which must be a starting point for debugging. These basic features are common properties to every smart object. Also here, a more complete study and above all a daily use of the system will identify if something is missing and will help to create properties where there is shortage of them. I preferred to keep them separated from the field "Properties" which will contain data of a particular type of object. The ones here described can be considered

"boundary properties": for instance the IP address is a technicality while the temperature value for a thermostat is something that differentiates it from a lamp.

Then, as anticipated, there the three main areas which distinguish the smart objects from other entities: properties, actions and events.

Properties provides readable and/or writeable data that can be static (e.g., supported mode, rated output voltage, etc.) or dynamic (e.g., current fill level of water, minimum recorded temperature, etc.). Their type is one or more JSON objects and the fields inside it are "basic" values, without the possibility to nest other levels. The main features of a property are "Name", "Description", "ValueType", "Value", "UnitOfMeasure", "Writable".

An important distinction with the W3C proposal is the fact that I am embedding the value of the property in JSON, without pointing to external resources, because I am working with HTTP protocol and so the value has only one representation.

Notice that if a property is writable it will have an associated action and the input the user has to provide must be of the same type of ValueType specified here.

More precisely, as reported in the table, the properties are:

Actions targets changes or processes on a thing that take a certain time to complete. W3C divides actions from writes on properties according to the time it takes to complete the task and according to the type of HTTP function used: PUT for property writes, POST for action. I preferred to keep them united because "the process is the same": the user has to interact, to write down something. How it is handled is a "backend" problem, the important thing is that the user can see under action all that is possible to do on that particular thing. Notice

TABLE VI: ”PROPERTY” OBJECT FIELDS

Name	Type	Description
name	Text	It specifies the name and type of the property.
description	Text	It is a more extensive description of the property.
valueType	Text	It is the keyword describing the type of the property.
value	*	It is the real value, its type is described in ValueType.
unitOfMeasure	Text	It is the definition of the unit of measure for the property.
writable	Boolean	It indicates if the property is writable.

that, anyway, a developer of a control center can put, for instance, a toggle button if a boolean property is writable but it is a matter of style, not a matter of concept.

Actions have the following properties: ”Name”, ”Description”, ”Property”, ”ValueTypeInput, ”inputValue”, ”UnitInput”, ”ValueTypeOutput”, ”OutputValue”, ”UnitOutput”. The last six values are not mandatory: it is not necessary to turn on a light to specify an input value and, on the other hand, this action will only change the property ”OnOff” of the light without producing other data as output. Conversely, the heating of an oven requires a desired temperature and after the action is started it will produce as output the time to reach the goal. These are two simple actions that a user performs everyday: they are the same but at the same time they are very different from a technical point of view.

Table VII reports all the fields of action formally.

TABLE VII: "ACTION" OBJECT FIELDS

Name	Type	Description
name	Text	It specifies the name and type of the action.
description	Text	It is a more extensive description of the action.
valueTypeInput	Text	It is the keyword describing the type of the input data.
inputValue	*	It is the real value, its type is described in ValueType.
unitInput	Text	It is the definition of the unit of measure for the input value.
valueTypeOutput	Text	It is the keyword describing the type of the output data.
outputValue	*	It is the real value, its type is described in ValueType.
unitOutput	Text	It is the definition of the unit of measure for the output value.
property	Text	Name of the property modified.

Events are the last category to be presented. The event interaction pattern enables a mechanism to be notified by a thing on a certain condition. An event is simply a modification of a state, a result of a completed action, which can be immediate such as the shutdown of a light or the oven which reaches a temperature after 20 minutes. An example of an event can be when the user is notified to bake food when the oven reaches the desired cooking temperature. The result is the same as an action, but it is triggered by something which happens in the environment and not by a user. The useful fields here are: "name", "description", "property",

"valueTypeOutput", "outputValue", "unitOutput". Table VIII reports them more formally, as done for the other fields of Smart Object.

TABLE VIII: "EVENT" OBJECT FIELDS

Name	**Type**	**Description**
name	Text	It specifies the name and type of the event.
description	Text	It is a more extensive description of the event.
valueTypeOutput	Text	It is the keyword describing the type of the output data.
outputValue	*	It is the real value, its type is described in ValueType.
unitOutput	Text	It is the definition of the unit of measure for the output value.
property	Text	Name of the property modified.

Having defined these three main components we are done with the common fields to all smart things. They are the three fundamental categories to describe all the possible interactions and "reactions" which is possible to have with the smart objects.

Beyond these categories, there exists some other useful fields which are not common to every object but to many of them. They are an important category which deserves to be explained and addressed. Instead of reporting them in a single object, taken as example, I preferred to list them here, in a generic way, precisely because they are very common even if not "universal". The "BatteryLevel" field is useful for the objects like beacons, not directly plugged in, it is

obviously a number. "Position" can be a description of where the object is, in case of small objects but it can be redundant if we talk about a fridge, for example. Some objects need a hub or a central unit which provide a lower level connection to the real devices and internet connection to my system: in this case smart objects have to be nested inside other objects, obviously in the right way: a hub cannot be the son of another hub. So the field "Devices" must be present and also the opposite, called "Hub". While the former as type can have one or an array of "Smart Object" elements, the latter can have only one "Smart Object".

Table IX contains a brief and formal recap of the fields which belong to the "Smart Object" category. In particular the table is composed of two sections: the first contains the fields common to every object while the second shows the fields just described, usual but not mandatory to all of them.

The three main components (properties, actions, events) are taken from the W3C proposal but their field are different: the security part is not dealt with in my work, it is beyond the purpose of my thesis but this does not mean it is not important or it can be ignored to construct a full working system, also reachable remotely. In addition the "Value" is explicitly reported, in order to be completely working, it does not dereference the value as it is done in the proposal. It is pretty clear that the W3C has given only an "incipit" to my work.

Also the type of the three main components is what is defined in the three tables Table VI, Table VII and Table VIII.

Regarding the names of the fields, a little clarification is needed: in the chapter the names started with a capital letter while in the tables with a lowercase one. The JSON standard uses

TABLE IX: ”SMART OBJECT” FIELDS

Name	**Type**	**Description**
...	...	...
name	Text	It specifies the name of the object.
manufacturerName	Text	It specifies the name of the manufacturer.
modelNumber	Text	Unique identifier of the hardware.
IPaddress	Text	The IP address of the object.
softwareVersion	Text	Version of the software installed.
error	Text	Errors are specified here, with a exhaustive description.
properties	JSON object	It lists the properties of objects.
actions	JSON object	It permits to perform actions on the objects.
events	JSON object	It handles notifications from objects.
...	...	...
batteryLevel	Integer	It indicates the battery level of the object.
position	Text	It indicates the position inside the ”Zone”.
devices	Smart Object [, Smart Object]*	In case of devices not capable of working alone this field links the hub to its sons.
hub	Smart Object	Reference to the hub in case of devices not able to work alone.

lowercase letters to start the name of the field so I will follow this specification, in the speech they are emphasized and they are put between quotation marks and in capital letters while the tables are more technical and formal so the standard was respected.

4.3.3 Use case

In this small subsection I would like to provide a full example of JSON-LD document which represents a theoretical proof of concept. Notice that I did not use all the fields I have defined in the previous pages: not all of them are mandatory to have a working system and maybe they can result redundant for a simple example with only few smart objects.

The use case describes the laboratory in which I have developed my system and where I made my proof of concept work. I would like to specify that anyway the use case was done before the real implementation, so the position of the objects, the types of the zones, the properties, the actions and the events here described can differ from the ones used in the next chapter as working demo. This does not imply that the working system is simpler than the use case or vice versa but that, simply, they can be different. In particular, due to space problems, it was impossible to write a JSON which occupies 10 pages of this document. So I preferred to limit the number of zones to two, one smart and one not, to show the difference. Also inside the smart zone I chose to put only one smart object, A Philips Hue lamp but with a hub, which means that it cannot be controlled directly but it needs additional hardware. Normally the Philips Hue lamps work coupled: one hub for two different bulbs, but it also possible to install only one of them. And that is precisely what has been done in this example.

The file describes a single smart space, as expected, for each JSON. It is basically an office,

constructed as an open space, composed of zones, identified in: Meeting Room and Restroom. The first is considered smart while the second is not. Having the Meeting Room in the interior at least one smart object, it possesses the field "containsSmartObjects", which permits to continue the nesting.

The root of whole JSON file is composed of the "@context" (line 2) which redirects to the hierarchy of all elements of schema.org, where it is supposed that all the previous hierarchy is defined, and of the "smartSpace" (line 3). This first part is right because for my structure each JSON file corresponds to only one real smart space to describe. The element "smartSpace" has many fields such as "name", "address", "geo", etc inherited from the current class "Place" of schema.org. They are not immediately linked to the concept of smart space but they are useful and they are the result of past work of schema.org.

Then the other elements belonging to the entity "smartSpace" are the ones I have defined in this thesis, in particular "composedOf" (line 23) contains an array of zones to nest the next level: the "Zone". Two zones are considered: the meeting room (line 24) and the restroom (line 112). Both of them have the specific properties defined in this chapter. In particular restroom has the field "smart" set to false (line 115) and so it does not have the field "containsSmartObjects". As a consequence it has no sons, while the meeting room has a lamp (line 28), put under the right field: "containsSmartObjects", which has as son an array of smart objects. Here, in particular, the only son is a lamp which needs a hub to be used. So under "containsSmartObjects" (line 29) there is a hub (line 31) not the object directly. The hub has an IP address and this particular category possesses a field, "devices", under which it is possible to find the real object: the lamp

(line 41). The lamp possesses all the three main categories (while the hub does not) previously defined: properties, actions and events.

In particular this lamp possesses three properties (line 49): status, color and intensity. They are readable values, which means an application can show the user their current state. But they are also writable: this means that there exist actions which enable the user to change properties at will.

Then there are precisely the three actions (line 75) to modify all the properties, with the type of the input values specified. In this case there is no output, it is possible to see the change reading again the value in the properties.

Finally, the events section: there is only one event (line 98) to notify the user if the lamp changes the status on its own, for instance if the bulb burns.

```
1 {
2   "@context": "http://schema.org/docs/full.html",
3   "smartSpace": {
4       "@type": "Open space office",
5       "name": "JOL S-Cube",
6       "description": "The S-Cube JOL (Smart Social
           Spaces) aims to define and design innovative
           services that make the space around us 'smarter
           ', by exploiting the opportunities offered by
           ultra-wide mobile broadband, the cloud and the
           latest technologies of Internet of Things (IoT)
           , Web of Things (WoT) and Wearable Devices.",
```

Figure 12: JSON use case example(1)

```
7         "address": "Building 23 Politecnico di Milano, Via
              Camillo Golgi, 42, 20133 Milano",
8         "openingHours": [
9             "Mo-Fr 09:00-17:30",
10            "Sa-Su closed"
11        ],
12        "geo": {
13                "@type": "GeoCoordinates",
14                "latitude": "45.477763",
15                "longitude": "9.234808"
16        },
17        "smokingAllowed": false,
18        "location": "Inside",
19        "private": true,
20        "owner": null,
21        "access": true,
22        "accessDescription": null,
23        "composedOf": [
24         {
25          "@type": "Meeting Room",
26          "name": "JOL Meeting Room",
27          "description": "Room for important business
                meetings",
28          "smart": true,
29          "containsSmartObjects": [
30           {
31            "@type": "Hub",
32            "name": "JOL Philips Hue Hub",
33            "manufacturerName": "Philips",
34            "modelNumber": "00178811aae6",
35            "IPaddress": "192.168.0.100",
36            "softwareVersion": "01033370",
37            "error": null,
38            "position": "Plugged in the nort-east corner
                 of the room",
39            "devices":
40             {
41              "@type": "Lamp",
42              "name": "JOL Philips Hue Lamp 1",
43              "progressiveID": "1",
44              "manufacturerName": "Philips",
45              "modelNumber": "LLC011",
46              "softwareVersion": "01033370",
47              "error": null,
48              "position": "Central table",
49              "properties": [
50               {
```

Figure 13: JSON use case example(2)

```
51                "name": "Status",
52                "description": "The status of the lamp, On
                      (true) or Off (false)",
53                "valueType": "Boolean",
54                "value": true,
55                "unitOfMeasure": null,
56                "writable": true
57               },
58               {
59                "name": "Color",
60                "description": "The current color of the
                     lamp",
61                "valueType": "Text",
62                "value": "#0066ff",
63                "unitOfMeasure": null,
64                "writable": true
65               },
66               {
67                "name": "Intensity",
68                "description": "The current intensity of
                     the lamp, from 0 to 100",
69                "valueType": "Integer",
70                "value": "76",
71                "unitOfMeasure": null,
72                "writable": true
73               }
74              ],
75              "actions": [
76               {
77                "name": "Switch On/Off",
78                "description": "To switch On/Off the lamp
                     ",
79                "valueTypeInput": "Boolean",
80                "inputValue": null,
81                "unitInput": null,
82                "property": "Status"
83               },
84               {
85                "name": "Color changer",
86                "description": "To change the color of the
                      lamp, input data must be a valid
                     hexadecimal color",
87                "valueTypeInput": "Text",
88                "inputValue": null,
89                "unitInput": null,
90                "property": "Color"
91               },
```

Figure 14: JSON use case example(3)

```
92                {
93                 "name": "Intensity changer",
94                 "description": "To change the intensity of
                       the lamp, input data must be a valid
                      integer between 0 and 100",
95                 "valueTypeInput": "Integer",
96                 "inputValue": null,
97                 "unitInput": null,
98                 "property": "Intensity"
99                }
100              ],
101              "events":
102              {
103               "name": "Status changed",
104               "description": "Notification for the change
                      of status of the lamp",
105               "valueTypeOutput": "Text",
106               "value": null,
107               "unitOutput": null,
108               "property": "Status"
109              }
110             }
111           }
112          ]
113         },
114         {
115          "@type": "Restroom",
116          "name": "JOL Restroom",
117          "description": "JOL Restroom",
118          "smart": false
119         }
120        ]
121   }
122 }
```

Figure 15: JSON use case example(4)

CHAPTER 5

CASE STUDY

This chapter is mainly composed of three parts: the first one is a generic information section in which the proof of concept is explained in terms of technologies used, requirements to meet, goals and various technicalities. The second one is the report of the implementation and development of the application, with choices and descriptions of what has been done, which tests were performed and what has been reached with the system. The third part is a working demo of the just described system, the demo is fully working. It contains screenshots of the application while it is running and a complete description to explain each case step by step.

5.1 Design Choices

5.1.1 System Description

This subsection is intended to provide a full overview of what I am developing to prove that the solution presented in the previous chapter is a fully functional and working idea. Obviously we cannot forget that we are talking about Web of Things and all its implications. To be as general as possible I chose to provide a "universal" application, reachable by everyone inside or near the smart space. For these many reasons the solution chosen is a small web application, in my case deployed only on localhost, but it is possible to install it on a public server or on a local (LAN) server. The idea is to prove that what I have stated can work with a real configuration of objects and smart space. In particular the core idea is to test the three

main functionalities of a smart object (property, action and event) and the fact that using my application it is possible to take decisions with a single tap or click, while using the current systems it is necessary to download, install and use different applications, the ones provided by the vendor of the hardware. Doing this, the thesis is somehow "demonstrated": my universal language, defined in a JSON file, is usable and working not to worry the users about the type of smart object they have installed in the space. The "translation process" is an issue of my application: collecting the objects, it has to be able to derive the syntax defined before and, on the other hand, in case of an action performed on a object by a user it should be able to translate from the general to the particular.

This small subsection has the aim to present the work and give a brief description, all the aspects of the web application are treated in section 5.2.

5.1.2 Non-functional Requirements

In section 3.4, the constraints of the problem can be seen as the functional requirements my development has to have, there is another category of requirements: the non-functional ones, they are important properties that my system must have in order to guarantee full functionalities. They are not specific for my problem or more generally for the Web of Things. They are general requirements a system needs to be considered complete, it is pretty clear how a system can use my language but if it takes 15 minutes to be online it is useless. So they are different but not less important than the functional ones. I am briefly listing all the non-functional properties:

- **Portability:** to have my application used by the largest number of users possible, but this is not a real requirement in the sense that the application is developed on the web and it is easily reacheable. We are on the border of functional and non-functional requirements.
- **Stability:** the system must be always available, and able to offer all its services. For example I should avoid possible system failures during the automatic reload of the configuration in case of a change in the smart space. In addition, data must be durable and not lost for any reasons.
- **Availability:** the services must be always accessible in time. In case of malfunctioning, administrator will provide maintenance in order not to affect service availability.
- **Reliability:** since data are shared among a large number of devices, reliability is essential. Users can base their actions on other users actions and on the status of the devices. Moreover, I assume that the memory where data are stored is stable.
- **Efficiency:** within software development framework, efficiency means using as few resources as possible. Thus, the system will provide data structures and algorithms aimed to maximize efficiency. I will also try to use well known patterns reusing as many pieces of code as possible, taking care of avoiding any anti-patterns.
- **Extensibility:** my application must provide a design where future updates are possible. It will be developed in such a way that the addition of new functionalities will not require radical changes to the internal structure and data flow.

- **Maintainability:** also modifications to a code that already exists have to be taken into account. For this reason the code must be easily readable and fully commented.
- **Security:** Using an online service security is always required. The fact that the system will be available only on LANs is the first step in this direction.

5.1.3 Used Technologies

This other subsection is an overview of the tools I have used to develop the web application.

The system has to meet the functional requirements 3.4 and the non-functional ones 5.1.2. Nowadays innovation gives us many languages and frameworks to help developers. My choice is based partly on past experiences in coding and partly on the certainty to develop a full working system.

I decided to develop a Java application, in particular using *Java SE 1.8* and a particular framework, *Springboot*, which embeds a server, *Tomcat*, to deploy Java web applications without using the Enterprise version of Java and its related server configuration which can result very heavy and time consuming. So I developed a backend written completely in Java:

- It handles the resources (the smart objects): the discovering process, the listening process of events and updates coming from the resources and the forwarding process of the actions performed by the user.
- It provides a translator working two ways: from particular to general and vice versa to assure complete communication among components.
- It notifies the front end in case of updates.

The front end part is developed using the most common languages of the web: HTML5, CSS3 and Javascript. The UI is kept clean. I used a theme already developed, obviously edited for my requirements based on the framework Bootstrap.

The reasons of my choice are based on my past experience at university, where Java is the main language taught. In addition Java has all the requirements to implement what I need: creating a big and complete control center maybe Javascript can be a better solution, but for a small proof of concept it is good enough.

Technically, I used an IDE to help in development, in particular IntelliJ IDEA, a modern solution released by Jet Brains. It contains modern tools to check Java compile time errors flagging them, and tools to check run time errors with a complete and verbose stack trace. It also enables to handle the front end: it is possible to edit and read the HTML (and so CSS and Javascript) files inside the IDE without external tools, also the HTML, CSS and Javascript keywords are underlined to provide a better user experience to the developer. Finally, it supports various VCS (Versioning control systems), to push the code inside repositories and to have a complete overview of commits and forks. I chose Git, one of the most famous VCS, and Bitbucket as repository to save my code.

The next section contains the information here briefly anticipated, described step by step to reach the final goal of proving the neutral language proposed in the previous chapter.

5.2 Implementation

This section describes the implementation done and it is composed of several short subsections focused each on a different aspect of the development.

5.2.1 Choice of the Objects

Before starting with the explanation of the implementation it is important to point out why and how the smart objects have been chosen and so which features have been tested and "uniformed" with my language. My development has taken place inside the JOL S-Cube laboratory in collaboration with the Politecnico di Milano, where I did all my work.

All the objects I used are provided by the laboratory. Unfortunately, I could not use all the devices I wanted to use, for example big home appliances such as fridges or ovens, so I have circumscribed the choice to objects which enabled me to test the three main categories of functionalities (property, action, event), two objects of the same type speaking different native languages (which means they use different protocols to communicate at lower levels), objects smart enough to be on the internet alone and others which needs hubs or additional hardware to work. These categories cover a big part, almost all, of the division made in the previous chapters regarding the type of smart things existing.

In particular the used objects are the following:

- A Netatmo weather station to test **properties**. It provides information on the current weather inside or outside a specific place. So we can show lots of different data such as temperature, wind, humidity, pressure, etc. The station also provides cloud APIs to interrogate in order to have the native data, ready to be unified by my system.
- Two types of lamps to test **actions**: Philips Hue lamps and BTicino ones. The Philips lamps can change the color of the light, while the BTicino can change only the intensity. Obviously the possibility to switch them on/off is provided.

- A Wemo presence sensor to test **events**. It is a simple sensor which can understand if there is someone in its field of vision. It can be very useful to test events and notifications, every time a person enters a room I receive a notification in the form of a web pop up.

The four listed objects also cover the other divisions: the Hue lamps work with a hub, while the Wemo sensor does not and so the test case of standalone objects or "weaker" objects is covered. Also the last category, maybe the most important to cover the universality of my approach, is covered: I chose two different types of lamp, each working with a different proprietary system but both uniformed in my JSON. I will provide my web application with a toggle switch to turn off/on the lamps: users do not need to worry about the vendor, the type and the configuration of the single lamp, they just decide to turn off the light when leaving the apartment. My system will identify which lamps are connected, then it will translate the unified command to lower level command for each lamp and run the task. So it is evident how my system works with a double intent: to uniform the smart space for the user (N to 1 approach) and to translate commands from the user to the real objects (1 to N approach). Anyway the next subsection explains in details all the procedure and the system.

5.2.2 General Structure

This subsection is intended to present in a general but accurate way the work done, while the next subsection will focus on each part of the implementation.

My work can be seen as a middleware between the Internet of Things world and the Web of Things world: I have created a sort of uniformed language, the web application developed is only to prove what I have theorized.

Starting from the bottom, the sources are the smart objects: they are the real components which give me data and possibilities to act on. So I have chosen, as anticipated, some of them and I have integrated them inside my system. I cannot have control on the market and on the format of data output by the various vendors: I developed some ad-hoc *plugs* to connect my system to the objects. The plugs interrogate the various objects using different connections, APIs and provided services; then they are responsible for keeping the connection active and inform the system in case of change of values previously retrieved. The plugs are considered a sort of listeners of objects, the part of my system where my web application acts as client for the objects. This behavior provides my system with raw data to be uniformed and then retrieved by the web. The plugs are considered the "lowest" part of system: they are a kind of interface of the world which stays under my system, the IoT.

Having defined how my system retrieves data, it is now necessary to explain how it can handle them: there is a *unifier* Java class, which transforms the various languages into my language. Also in this case, the various translators are built ad-hoc: if each source provides data organized in different ways, it is necessary to have different functions to unify. The unifier class is a part of the controller, it is part of the core of the system. The unifier is the *N to 1* component, on the other hand the *1 to N* translation is done by the *actuator*, a particular object which handles requests made by the user and translates them into actions the objects can understand. Also the actuator is part of the controller of my application.

Then there is the front-end, the part which is not "strictly necessary to be there": considering my work as a middleware, it should export data to be used and processed by third-party

applications and developers. The data, at the moment, may not be ready to be shown. What I have done is creating a small UI to prove my work. So the interface must be minimal, with all the proofs needed to verify the concept, it can be maybe a little tricky to understand for an average user, but I have tried to apply the KIS (keep it simple) paradigm to be as clear and understandable as possible.

Until here we can somehow apply the *MVC (Model-View-Controller)* pattern to my work: the plugs are an "indirect model", smart objects are the source of data, they are not a classical database, they sample data from the real world but they are the input of my system. The unifier and the actuator are the controller, the core of the application and what really characterizes my work. The front-end is, obviously, the view part and it is done with the "classical" standards used nowadays.

Finally, a mechanism to notify the system and the client that something has changed in the JSON file describing the smart space is necessary: so it is fundamental to keep track of the instances of clients opened on the web by users and it is done trough web sockets. The web server keeps them in a list and notifies them in case of updates of the JSON. The bit-a-bit comparison among versions of the JSON clarifies if there are changes, and in this case a notification through the HTTP protocol is sent to the client, which requests the new version of the JSON from the server through a Javascript script and a HTTP call.

Figure 16 describes the general and global architecture of the implemented system. The figure is not a real and complete UML diagram, due to the limited space of this document. It has been done along the lines of Figure 7, using the same structure and colors, even if they

may have a different shade of meaning. The figure is a general overview of the system with also external elements, while in each of the next subsections, which explain only parts of the system, the UMLs are provided. Anyway, some explanations are necessary to understand completely the figure and the system. First of all, as anticipated, I developed also a small front end, to prove the studies I have made are not only theoretical but also they can be implemented and used in everyday life. This part is generically represented in the figure as a user's terminal, a computer, which represents an instance of a client, the red box, in a client-server architecture like mine. This part is outside my system, it is simply a web page opened in a web browser, so it resides in the Web and it is WoT.

On the other side, we have the objects, the IoT world, what can be considered "source" of my work. My system provides ad-hoc interfaces to connect itself to these outside components.

Between these two worlds is my system: a middleware to proper connect the IoT and the WoT. The application starts from the green arrow in the left middle of the page: having used Springboot, a Java framework which allows to use the Java Standard version to deploy web applications, a *main* function is necessary to boot the system. The *main* is contained in a Java class called *¡name of project¿ Application*. During the booting the necessary objects are created. After the system has booted, it is necessary to launch functions to map HTML files to various sub-URLs of the web application. The mapping methods are inserted in a Java class called *¡name of project¿ Controller*.

The created objects belong principally to three Java packages. They are represented by light blue rectangular shapes, while Java classes are dark blue (except for the two particular classes

of the Springboot system, represented in green). The only rectangular element not explained yet is the light red shape: it is a collection, in particular a Java ArrayList, of web sockets. Each element of the collection represents a web socket: a full-duplex channel in which the two components (clients and server) can talk and listen, exchanging messages with useful information. The black arrows represent the actions among the components and classes of my system: some of them have a key word to better explain the relation, where it is not specified the relation is more complex and it will be explained in the corresponding section. The "core" of the actions can be seen in the output of the "Uniformer" component, which provides the unified JSON to each client connected.

This section has explained the "static" vision of the application, what can be seen at *compile time*: Java packages, classes and relations among classes. A classical "routine" (a normal execution) of the system is missing: at *run time* only the boot is described, all the rest is explained component by component in the next sections.

5.2.3 Plugs

The "plugs" are generic components, here generalized, to connect my system to different IoT objects. The word "generic" means that it is possible to create as many plugs as wanted, each one to connect to a different type of architecture and vendor's services. I have reported in Figure 16 only one generic package called "Plugs", but it is obviously possible to replicate the package and change the parameters to have different versions, "PhilipsHuePlug" for instance.

As said the plug connects and interfaces my system to the real hardware, but this is not its only goal: it has to continuously observe the object, understand when something has changed

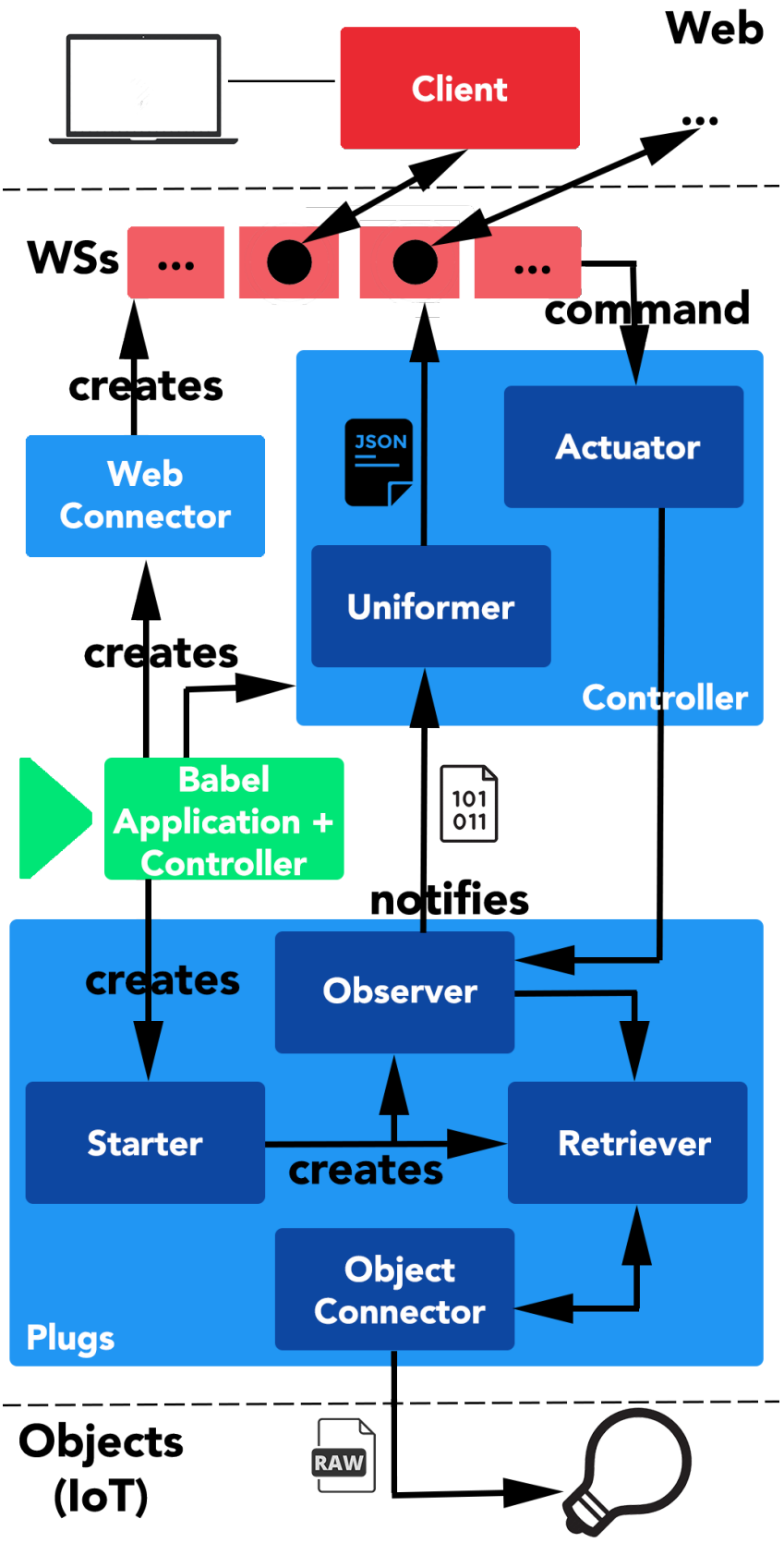

Figure 16: Babel Framework architecture

and notify the controller.

The package "Plugs" is composed of 4 classes, all necessary for the right behavior of the application. The application is developed in Java so it has been possible to use some useful patterns already implemented: for example, in this case it is fundamental to have *parallel processing* because while a sort of "listener" is connected with the smart object other components of my application have to perform other tasks. Also, the listener compares the information retrieved during the time and in case something changes it sends a "notification" to the rest of the application, in order to perform this the *Observer pattern* is used.

In particular analyzing the four Java classes we have:

- **Connector:** the connector provides the methods which act on the network. It possesses the URL, the address of the resource as attribute, it can be defined static and final, it should never change. The methods are principally of two types: to read and to "write" on the object. The read is performed through an HTTP GET, while the write changes on the type and brand of the chosen object: some can use a GET of a modified URL to include the command, some others can request a HTTP POST or other ways. Despite the fact that the plug package can be replicated for the different objects, the connector class can remain the same, offering different signatures of the same methods for different objects and different URLs, all saved in a collection. An hash map with the id of the object can be a reasonable solution.

- **Starter:** the starter class is a simple Java class aims to start the process of the Observer pattern: it creates the object to be observed, the *Retriever* class and then the observer

class, called *Observer*. Finally the *addObserver* instruction is needed to "link" the two objects, to tell the observed one who to notify in case of change.

- **Retriever:** the retriever class is somehow the core of the package, its aim is to interrogate and retrieve information from the objects. In particular it has to be a continuos process, a monitoring phase interspersed by a sleep phase. This class works on a thread different from the main one, to let the application execute other tasks during the sleeping phase.
- **Observer:** the observer class is "who" is notified if something changed in the IoT configuration of the smart space. It aims to activate the uniformer to start to work on new data coming from objects. It is the supervisor in the Observer Java pattern, it has to be continuously active in order to receive changes in any moment. It is in a sort of sleeping status, never dead, and activated when a new configuration comes.

Figure 17 represents the UML class diagram exported directly from my project, only for the "Plugs" package, external relations among packages are not reported. The UML reports the structure above described with properties and methods of each class, also the relations inside the current package are printed. With the term relations I intend all the classical UML components: aggregation, dependencies and associations.

5.2.4 Controller

The package "Controller" contains the core and the real innovation of my work, it is composed of only two classes: "Uniformer" and "Actuator". They are complementary: one translates a multitude of languages into one specific language, the other does the reverse process. The uniformer performs a N to 1 transformation, the actuator a 1 to N transformation. In

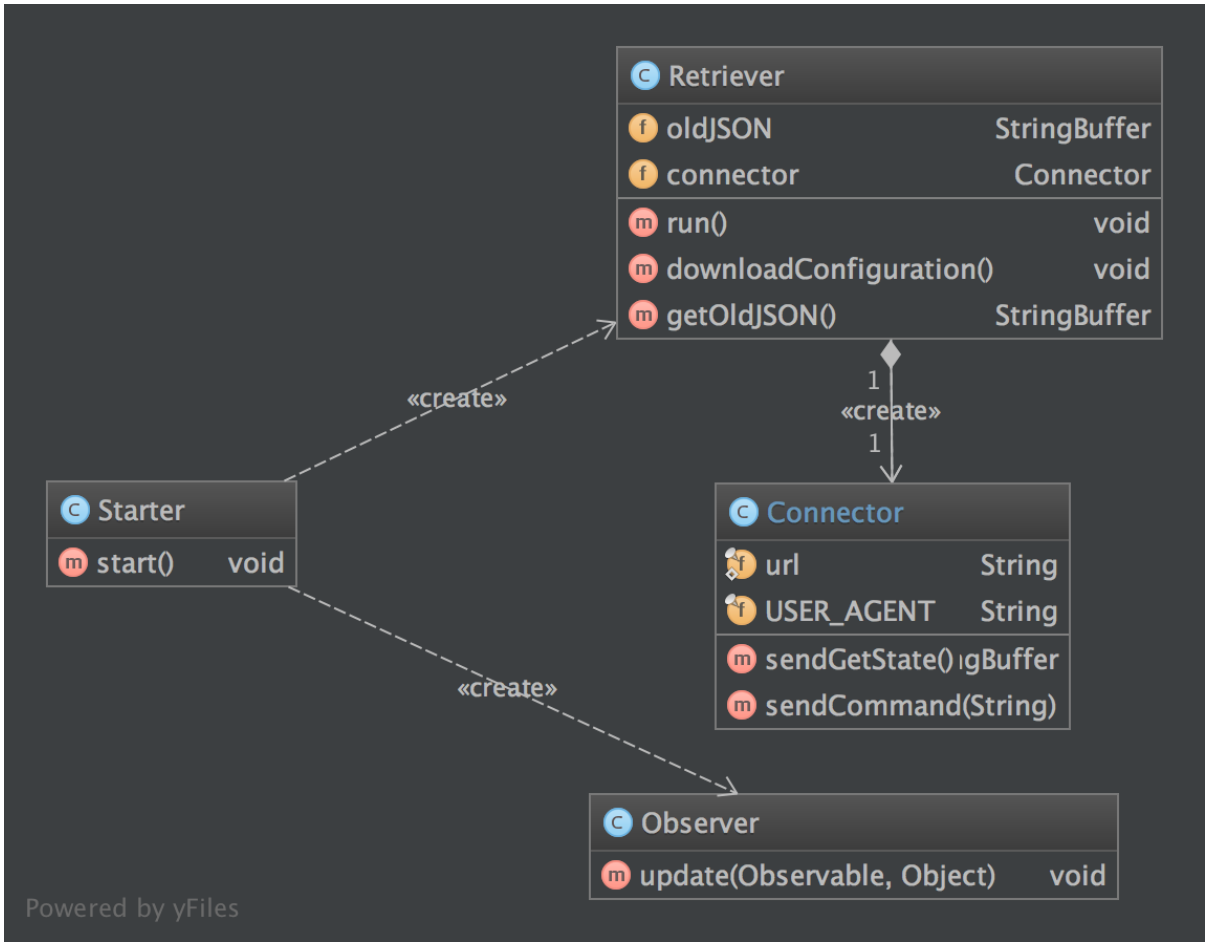

Figure 17: UML Class Diagram of Plugs package

particular the uniformer receives as input binary data containing data coming from the IoT objects. Then with some standard procedures it creates a unified model of language, a model which is easily readable and compliant with all the requirements already pointed out. Obviously in my implementation it was impossible to write down code to translate all the possible inputs coming from heterogeneous objects: I made choices based on available objects, as anticipated

in 5.2.1, so they are a proof of concept regarding the selected set of objects, it cannot work with different objects simply because the information coming from proprietary solution can be saved in different ways: if a keyword is used in a different way or with a different meaning it is very difficult to "understand and convert" it correctly.

The "Uniformer" collects all the data and then performs the translation, creating the JSON and sends them to clients. Being my system a "thing descriptor" or, more generally, "smart space descriptor", a discovery plug is missing and so the informal description of the space must be somehow known a priori by my system: the developer, in this case me, has to write down code to describe the environment. In the future, if my solution is adopted a discovery module is necessary: new objects could be found immediately and maybe a description of the environment saved somewhere online (maybe with the help of schema.org entities for the definition) can be retrieved to have the full description of the smart space. The configuration of the environment happens in "configureJSONEnvironment()" and "configureDemoArea()" methods of the class. Having done this clarification, it is necessary to explain in detail how the class works. It has only two variables: the source of data that is a JSONObject "ohJSON" (in my working demo I managed to use sources of data which are all contained inside JSON) and the String "JSON-ToBuild" which contains the JSON in the making. The "resetObject()" method is needed to start from a clean situation in case of errors in the system, while "uniform()" is the core of the class responsible for calling all the other methods when needed, it has also the goal to be ready to be notified by the Observer class when something changes.

The actuator is the Java class which does the opposite task: when a user selects a command

it is necessary to translate it in a language which can be understood by objects. Also here, as before, it is not possible to write down a universal translator: it is impossible to have an output on every object existing. Considering 1 to N paradigm is pretty clear how N cannot be infinite, it is a finite number, selected a priori. Obviously a large use of my system will increase the number N to cover almost all the cases. The class possesses a method called "act(Sting message)" which is the fundamental part of this class: the method fires when a message is received, thanks to the Web Sockets architecture explained in the next subsection, the parameter is the payload of the message, it should be decoded and then translated. Finally the action is performed on the object, with the help of the "Connector" and HTTP POST method, as already explained.

5.2.5 Web Connector

The "Web Connector" (name of the package: "Webconnection") is the component keeps track of the clients connected to the framework and to maintain the connection working. To have a full-duplex, reliable and stable connection I chose to use the Web Sockets. The Web Sockets protocol is a communications protocol, providing full-duplex channels over a single TCP connection. The two sides of the connection can write and read, always knowing the end point of the channel, without "discover it". This component has a fundamental role: in a full working system it is needed to keep track of the third-party applications connected, while in this case study it keeps track of the instances of my clients opened. The package in my application is composed of only two classes: the Web Sockets does not require a lot of effort to have a working

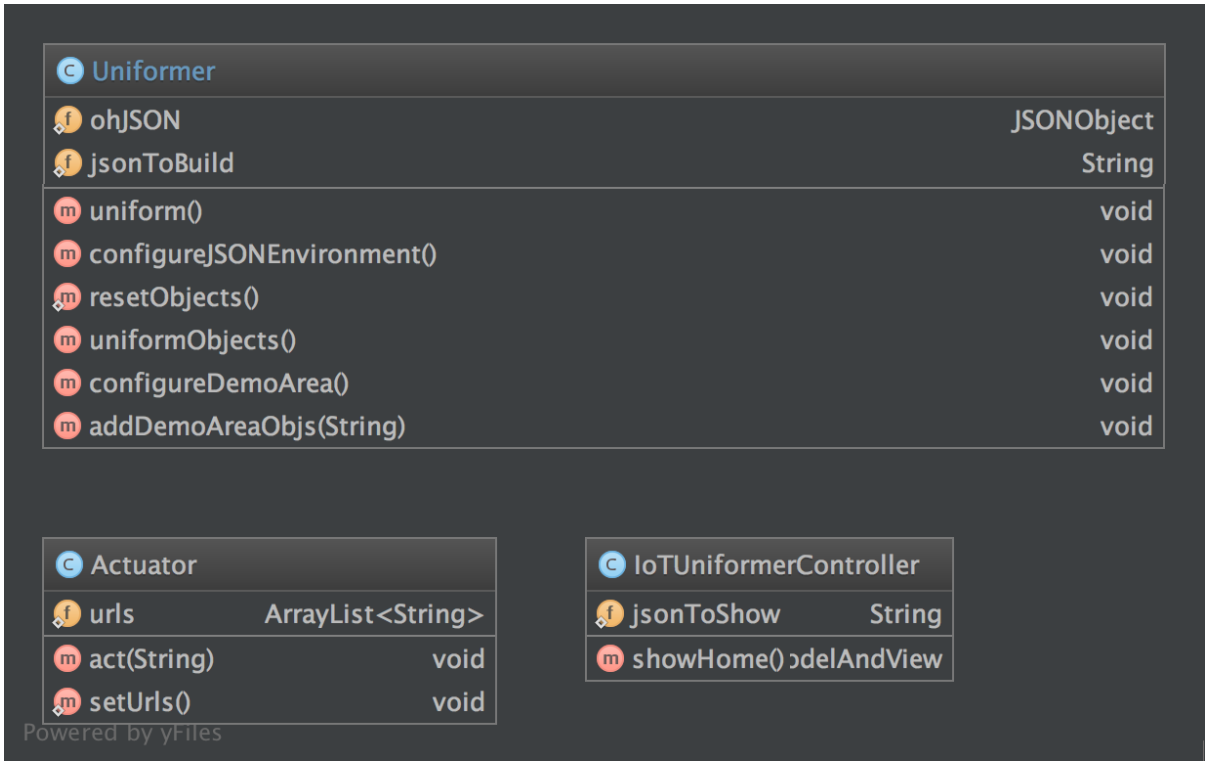

Figure 18: UML Class Diagram of Controller package

architecture.

The two Java classes are:

- **WebSocketConfig:** this class is the starting point of the connection. The first thing to do is to establish the online ending point of the connection: it is necessary to set up an URL on which the messages can be sent and received. For example, for my case study, I deployed my application at *http://localhost:8090* and I created the web socket end point at *http://localhost:8090/websocket*. The method "registerWebSocketHandlers"

creates the end point and saves it in the Web Socket Registry, the component able to keep track of the created end points.

- **ConnectionHandler:** while the WebSocketConfig Java class creates the skeleton of the connection, the ConnectionHandler is the component which manages the various events which can happen during the connection. In particular, some inherited methods are already present to handle all the possible events, it is necessary only to override them with the desired logic to have the application behave according to the needs. In particular the methods I have overwritten are the following: *afterConnectionEstablished*, *afterConnectionClosed* and *handleTextMessage*. The first method fires when the connection is established, in my system this method adds the session (which is an instance of the object Session, ready to be used) just opened to the ArrayList of the opened sessions. Its "opponent" is the second method listed: when the connection is closed the session must be removed from the collection which contains only the active connections. Finally the last method is what happens between the opening and closing of the connection. The methods receive the entire message as parameter, the "handleTextMessage" extracts the payload and then it decodes it to understand which action the user has chosen, finally the "actuator" component is activated and the action is translated to the IoT vendor's language and executed on the smart object.

In the previous classes there is trace of how it is possible to send messages: the method is very simple and it is already provided by the Java WebSocket library. It is sufficient to call the method "sendMessage(String message)" on the object session, which has to be active. In

this case, there is no need of overwriting the method, but only to keep track correctly of the all opened sessions and call the send message function when needed on desired opened sessions.

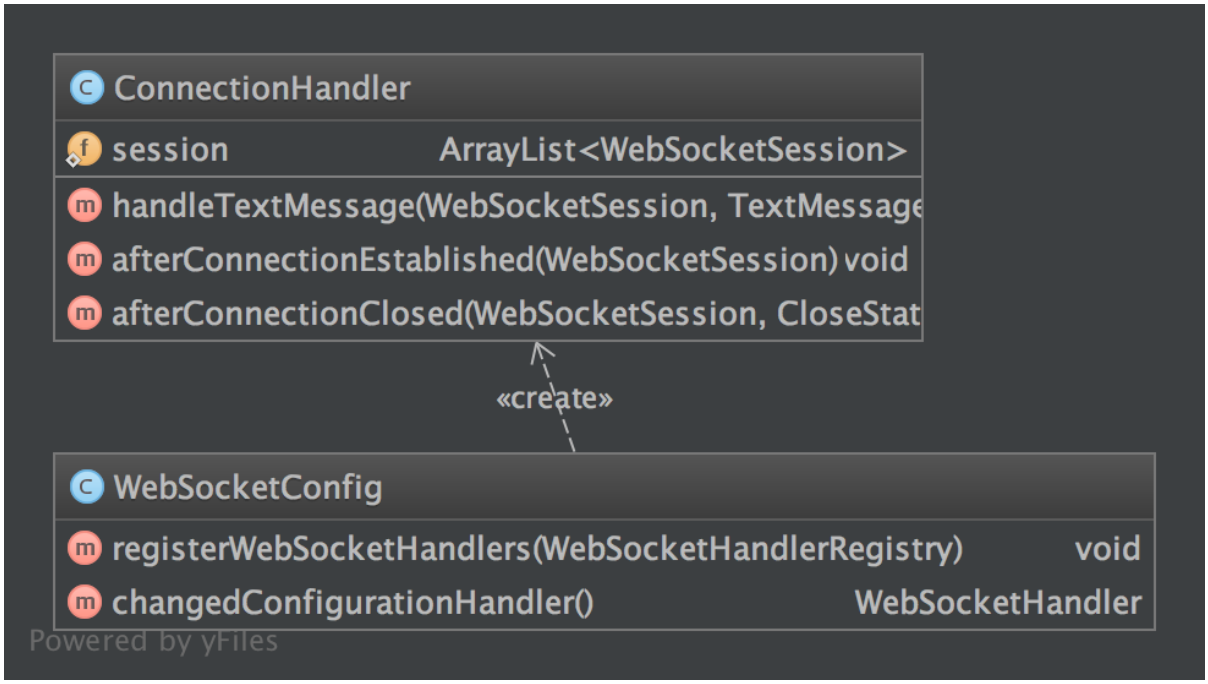

Figure 19: UML Class Diagram of Webconnector package

Before going on in describing the last part, the front end part of the instance of my solution, I would like to point out how the packages and the code until now described (subsections 5.2.3, 5.2.4 and 5.2.5) are universally portable: obviously the code is written in Java but the functionalities and the logic can be easily reused, for instance, in a native Android application (which uses Java for the backend). This statements are reported to remark once again how I

only chose one of the possible implementations: the web application solution. In the future it will be possible to change the technology used to reach the final user without changing the logic until here used. Obviously if the programming language changes some adjustments are needed: for example, something which can substitute the Java Observer pattern could not exist and maybe it could be necessary to go to lower levels to implement a sort of custom listener for the situation.

5.2.6 Front End

As just said, the Front End side is part of a web application, so it must satisfy and be compliant with the web standards. It is very simple, in the sense that it should be only a graphical interface to show the main features of my work. It is composed, as most web pages nowadays, of three main elements: an HTML file, which describes the structure and all the elements of the page; a CSS theme based on the framework Bootstrap, which determines the graphical part of the web application using the responsive pattern to better adapt to each type of screen on which the web application is opened; and Javascript files which contain some logic which must mandatorily be on the client. Trivially all the code to create and make operative the listeners on the buttons to activate an action must be put in Javascript code. The same is valid for decoding the JSON when it is received through the web socket channel.

Obviously this part of the application is not inside a Java package but is differently organized: it is under the "resources" part of the Java project. In particular, inside "resources", there is a distinction between "static" and "templates" folders. The first one contains all the CSS (in particular the Bootstrap theme), the .js files, the images and "Font Awesome" which permits

to have web icons easily. They are somehow the skeleton of the application, independent from the page displayed or feature currently running on the client. Under the second, "templates", there are all the HTML pages used, in this case only one "index.html". They are considered not static, they can be added and linked to each other independently from the Javascript code or CSS theme used. The structure is reported in the following figure, it arrives to the folder levels, the nested level, the files are explained just later. I will not explain standard parts of the web development in detail: what is inside the "css" folder for example, is a standard theme made with Bootstrap, all the internal configurations and settings are already made up, the files are "somehow standard". The same is valid for the folder "font" and "font-awesome", nothing I have done personally is inside here.

It is a different matter for the remaining folders: "images" and "js". The "images" folder contains, as the word says, the images used in my web application. In this case they are only two and they are the icons to represent the objects used to show the "Properties" field of my work. In particular they represent the Netatmo Weather stations, the inside and outside models.

Regarding the "js" folder, it contains the Javascript logic needed to make the application work correctly. In particular the methods to communicate over the web sockets are contained: a listener which receives the new JSON and updates the status of each object according to the last received data. Also it is necessary to put listeners on the page, to register each action the user does to make the right call on the right object; after the action has been captured it is mandatory to have a component which writes on the web socket what has just happened to permit my system to start the translation process to the lower level language, spoken by the

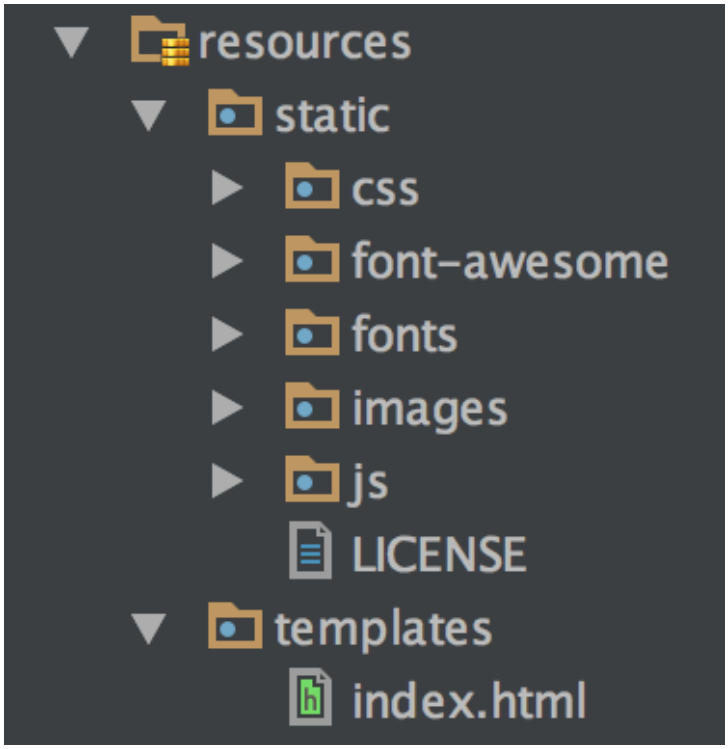

Figure 20: The front end part inside the Java project

IoT object. In particular, the listening part is done by the file called "listener.js", while the action capturing and the message sending are inserted in the "actuator.js" file, which recalls the name of the Java class responsible for continuing the process of acting on the object after a human input. The two files are imported in the body of the HTML file "index", they are active on the page when it is created by a HTTP GET in a normal browser.

The two next figures report, for completeness, the just described files in their position and folder.

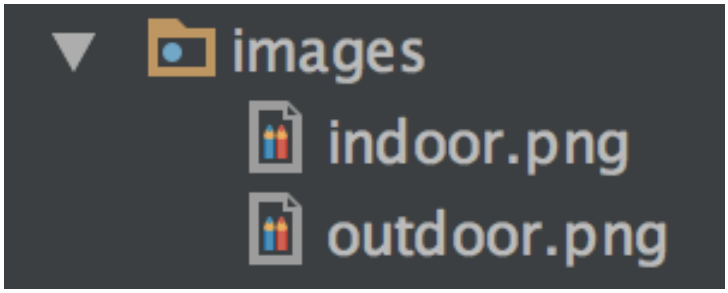

Figure 21: Content of "images" folder

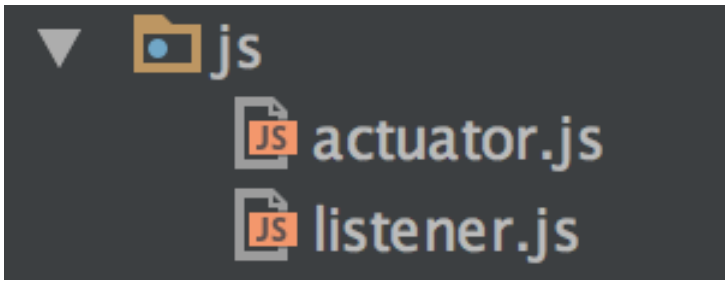

Figure 22: Content of "js" folder

The front end part is not very extensible: it strongly depends on the fact that the instance is a web application. It uses the "classical" tools of the web: languages such as HTML5, CSS3 and some Javascript functions. In a iOS or Android application it must be redesigned. It also represents only a proof of concept of my work: a third-party application should be very different. It is only a ad-hoc application to show the potential of my work, it is configured to represent my laboratory and it is designed only for it. It is not something which can be considered extensible. It uses the most common web developing tools but with a very restricted

purpose.

It is also necessary to specify how it is possible to compose front end pages and graphical UIs only reading my JSON: the file contains all the structure and hierarchy of a whole smart space, so a machine can easily read and parse the content of the file creating and putting graphical elements in the right positions inside a hypothetical control center. It is important to specify how no human intervention is needed to create appropriate pages: it is sufficient to have a mapping between the values that it is possible to read inside the JSON and images, button or whatever is required to give the user a catchy application. Also the controls can be mapped to actions: a switch can be applied in case of a boolean value, a dimmer in case of a numeric value etc. So it is possible to state that the UIs can be created dynamically depending on the JSON, without people to encode the content of the smart space.

5.3 Working Demo

This section is intended to show the reader the web application while it is working. After the structure and the implementation of the application were explained it is necessary to show the finished work. As anticipated, my application is simply a proof of concept of how it is possible to control something (IoT objects) which is heterogeneous and differentiated from a single UI or application, which is homogeneous. Users should not worry about substrates, they can control everything with a single and well known interface.

My demo describes the laboratory in which I have developed the application, so it can be seen as a case study designed for a specific building and smart space. It does not aim to be the a general and universal solution, my work wants to be a middleware on which it is possible to

construct catchy applications for the final users. The application is simply *one-page application*, which works only under the network installed in the laboratory. It is composed of colored boxes, each with a different purpose, but put together they are the full proof of the work.

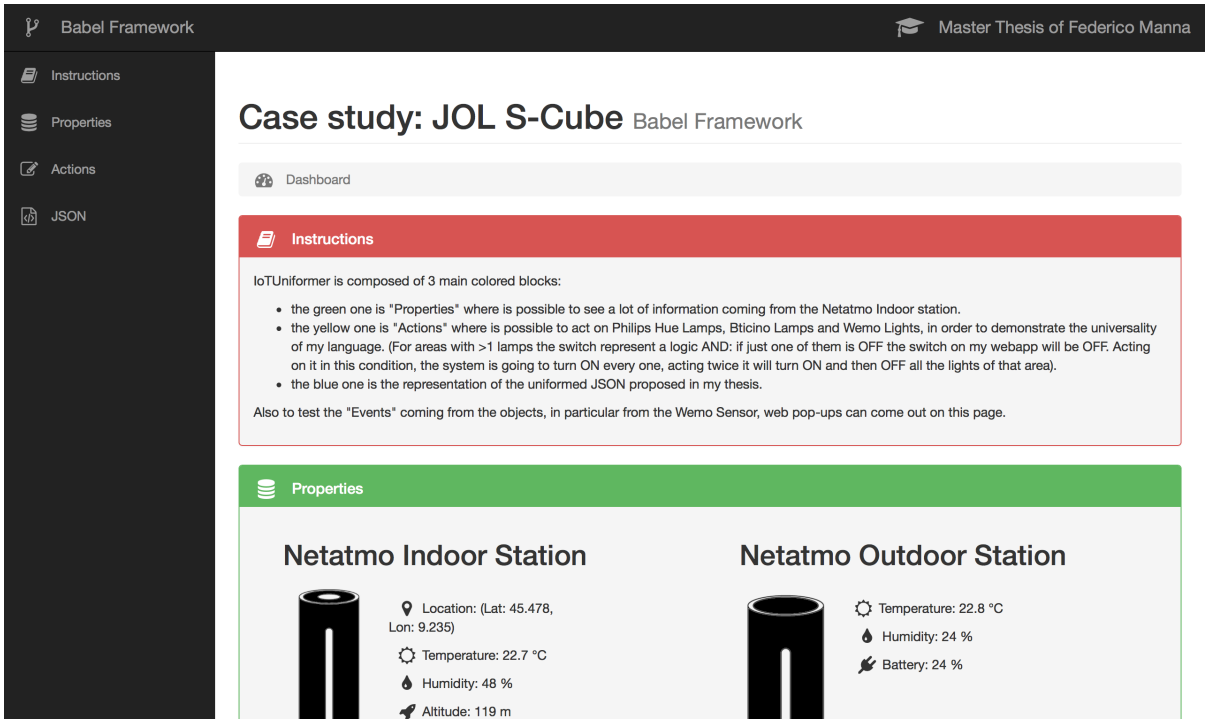

Figure 23: Babel example application

Figure 23 shows a portion of the screen while the application is running, in particular the beginning of the page. The web application is composed of a lateral left menu which contains anchors to each box displayed. The horizontal bar on the top of the screen is simply a sort of

title of my application and the reason why it is my web application has been developed.

The first red box is reported only on the above figure and it is simply an explanation of what the application is doing and how the results are shown. The others are explained later with ad-hoc screen shots. The red box only explains how the application shows mainly four things: the three fundamental fields of each smart object (property, action and event) and the current JSON in which it is possible to find data in a less flashy but more technical way. Events, being something which depends on external factors, do not have a dedicated box, but when an event comes up a "snack bar" is shown at the bottom of the screen.

The figure, as said, shows the application running. But if the underlying infrastructure is completely down for external reasons the UI is different: the red box, "Instructions", remains the same while "Properties" and "Actions" are empty, "Events" cannot show up but the interesting point is what is contained inside the JSON which drives all: there exists a simple assignment of value to the JOSN document in case the IoT is not working. It is reported in the next figure:

Figure 24: JSON section with inactive system

It is not a classical and valid JSON, but it is a particular string which identifies the fact that the system is not working. It can be the underlying IoT world or maybe the machine which contains the application server is running on a different network and it retrieves no objects. These are the most common problems which justify this initialization. When the error is limited to an object - for example if it is impossible to read properties maybe because the hardware sensor is broken - it is shown in the field "error" of the JSON object "smartObject", designed for the purpose.

In this last case the system is rightly configured and it is working, so the JSON section contains the proper description of the smart space and the smart objects. In Figure 25 the report of the starting part of the JSON describes the case study done for my thesis. Due to problems of available space it was impossible to report the entire document, I chose to screen shot the beginning to show the reader what it looks like when the system is active. Anyway it is possible to identify the full structure: the figure reports all the levels, finishing with properties, actions and events. It is possible to note the "double root" composed of "@context" and "smartSpace", then the "smartSpace" contains all the useful fields, among which there is "composedOf" and all the relatives zones. For instance here we have the complete description of the "Meeting Room" and the partial description of the "Open Space" of the laboratory. Focusing on the meeting room, where the structure is complete, the valuable part is the fact the zone is considered smart, in fact the value under "smart" is put to true.

The "containsSmartObjects" has as value a JSON array of objects, in this case composed only of one object: a smart lamp. The smart lamp is defined as "Switch Lamp", which means

JSON


```
{
  "@context": ["http://schema.org/docs/full.html"],
  "smartSpace": {
      "@type": "Open space office",
      "name": "JOL S-Cube",
      "description": "The S-Cube JOL (Smart Social Spaces) aims to define and design innovative services that make the space
      "address": "Building 23 Politecnico di Milano, Via Camillo Golgi, 42, 20133 Milano",
      "openingHours": [
          "Mo-Fr 09:00-17:30",
          "Sa-Su closed"
      ],
      "geo": {
              "@type": "GeoCoordinates",
              "latitude": "45.477763",
              "longitude": "9.234808"
      },
      "smokingAllowed": false,
      "location": "Inside",
      "private": true,
      "owner": null,
      "access": true,
      "accessDescription": null,
      "composedOf": [
          {
            "@type": "Meeting Room",
            "name": "JOL Meeting Room",
            "description": "Room for important business meetings",
            "smart": true,
            "containsSmartObjects": [
                   {
                    "@type": "Switch Lamps",
                    "name": "JOL Meeting Room Lamps",
                    "manufacturerName": "BTicino",
                    "error": "null",
                    "position": "ceiling",
                    "properties": [
                       {
                        "name": "Status",
                        "description": "The status of the lamp, On (true) or Off (false)",
                        "valueType": "Boolean",
                        "value": true,
                        "unitOfMeasure": null,
                        "writable": true
                       }
                      ],
                    "actions": [
                       {
                        "name": "Switch On/Off",
                        "description": "To switch On/Off the lamp",
                        "valueTypeInput": "Boolean",
                        "value": null,
                        "unitOfMeasure": null
                       }
                      ],
                    "events": []
                   }
              ]
          },
          {
            "@type": "Open Space",
            "name": "JOL Demo Area",
            "description": "Area intended to show visitors the prototypes",
            "smart": true,
            "containsSmartObjects": [
                   {
                    "@type": "Weather Sensor",
                    "name": "JOL Indoor Weateher Sensor",
                    "manufacturerName": "Netatmo",
                    "error": "null",
                    "position": "on the central table",
                    "properties": [
                       {
                        "name": "Location",
                        "description": "The location detected from the inside sensor",
                        "valueType": "Float",
                        "value": "(Lat: 45.478, Lon: 9.235)",
                        "unitOfMeasure": "null",
                        "writable": false
                       },
                       {
                        "name": "Temperature",
                        "description": "The temperature detected from the inside sensor",
                        "valueType": "Float",
```


Figure 25: JSON section example

it can only switch it on or off. The position helps to understand where the object is. Then the three main areas are there. This lamp has only one property: the status property which describes the current status, it can be ON or OFF, indicated here under the field "value", where true indicates ON and false indicates OFF. Then under "actions" it is possible to find the action which enables to change the status of the lamp. The name of the action and the position in the JSON document uniquely identifies the destination and the action to perform. The field "events" here is empty, the lamp has no possibility to send events to clients.

Having briefly explained the three fields inside the JSON box of the application I would like now to describe other colored boxes. I am going to start with the "Properties" box. The green box contains the information provided by an object. In particular, I have chosen to show the user an object which has lots of data, not only a status, like the smart lamps described before. The choice was made accordingly to available objects, and it was about a couple of weather stations which give information about the environment. In particular one should be installed in an indoor area and one outdoors. This box simply reads data from the JSON which come from the server and formats them in an attractive way to be shown, using also icons and images. When data are not available, the writing which appears is "not available".

The yellow box, the next one I am presenting is the actions panel. This box is the only one in the web application which enables the user to interact with the system. To demonstrate how my work controls different vendors objects without worrying about the substrates I implemented the box with on/off switches to control the state of the lights. I have created a switch button for each area containing lamps. In particular they are 5 areas. Four of them (Off. A, B and C

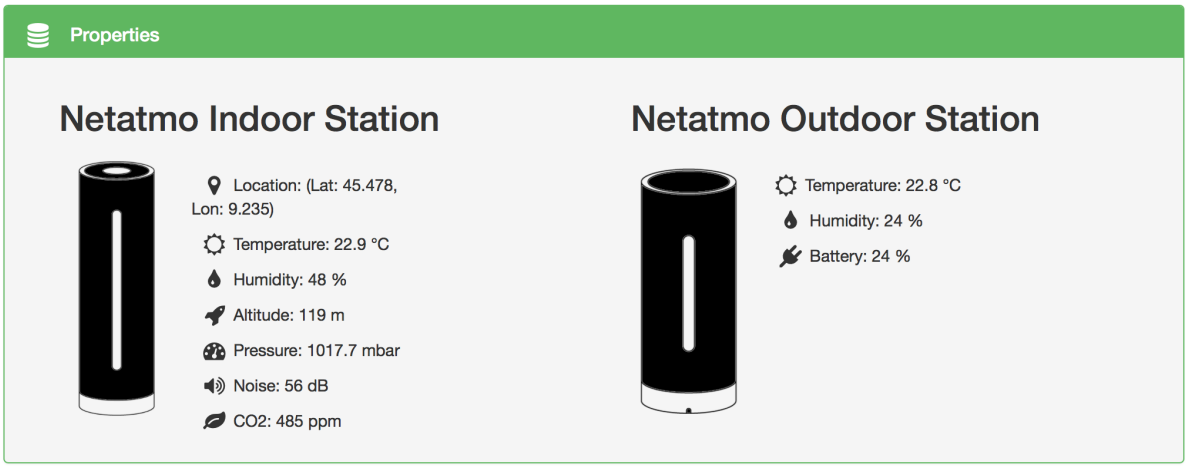

Figure 26: Property section example

and the Meeting Room) contain only one type of lamps: the BTicino lamps. The Demo Area is the core of the entire development in actions: it contains four different kinds of lamps: BTicino, 2 different types of Philips Hue and a Wemo Lamp. Switching that button I am controlling 4 different types of lamps with only one click: this is what I have proposed as the problem to solve in the previous chapters. Note that if the state of the lamp is ON the switch is blue, otherwise it is grey. In this case we have both reading and writing, implicitly this box also represents the properties in addition to the actions. The button "Demo Area" is set to ON only if all the lamps in the area are set to ON (remember that the lights can be switched also manually, like all the normal lamps). The switch button represents a logical AND among all the states of the bulbs of that zone. What has just been stated is valid also for the button "All" which is a global logic AND to control all the lamps in the smart space. In the case one

or more lamps are enabled to work the corresponding button will be disabled and its status is not considered inside the AND.

Figure 27: Action section example

The ”events” are a particular category, they are triggered by the environment, they are something permanent or chosen by users. When a particular situation comes up, it can be converted in an event (it can maybe depend on hypothetical user's preferences in a third-party application). The event can come completely from the environment, such as someone passing in front of a camera or a sensor, or it can come indirectly from a user: if the user activates the action ”heat the oven”, it takes time, it is not an immediate action such as turning on/off a lamp. When the oven is ready, an event shows up, this event was indirectly triggered by a user.

In my proof of concept I chose to include only one object able to send events, a motion sensor: it is a simple sensor used to notify if someone or somewhat is present in a space, in that case it can trigger a process to send a notification or, for instance, to turn off a light. In my case I chose to send a notification to all the clients connected, in particular I preferred to show a non-invasive snack bar notification at the bottom of the screen. The notification active, which lasts for 3 seconds every time someone passes in front of the sensor, is visible in Figure 28.

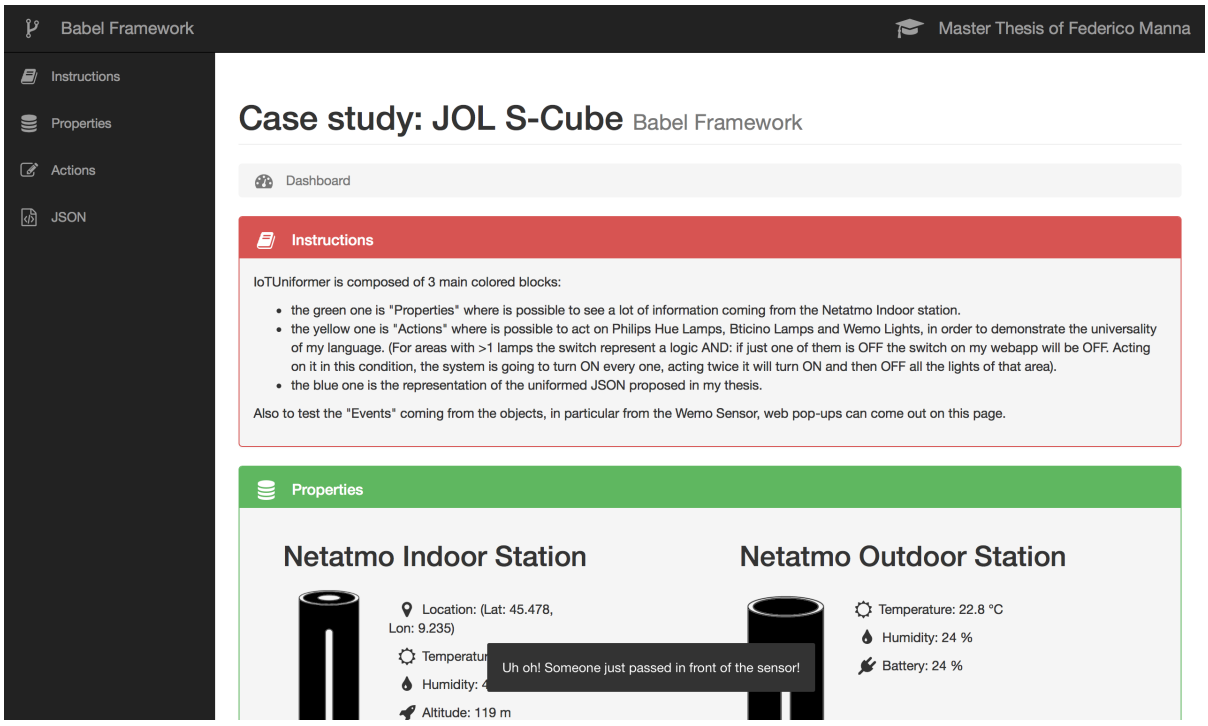

Figure 28: Events snackbar example

It is a grey box which warns the user that someone is in the kitchen of my laboratory. The event is present in the JSON file, only for the time it fires: when an event is present inside a JSON, a request of another JSON is immediately sent to the server, in order to update the status (in case it changes a status) and/or to see if it finished. Usually events are situations which end in a moment: if the oven reaches the temperature the event fires but it does not fire continuously if the oven maintains the desired temperature.

This section exhausts the topic, the next chapter deals with the conclusions which can be drawn from my thesis and any future work which can be done to extend and make the work as universal as possible.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter aims to analyze what has been done in order to give feedback and understand if it is worth enhancing the framework with other modules to complete the work.

The chapter is divided into two parts, the first is the conclusion of the work, the analysis of what I have reached with my idea and my implementation; the second part is focused on future implementation and actions to be carried out to make my system work in the current fragmented IoT and Wot worlds.

6.1 Conclusions

The previous chapter showed how, despite the fact it was only a case study, it is possible to use my system to really uniform and control objects which, otherwise, speak different languages. The proposal was to develop a middleware able to create a uniformed language for the description of the smart entities. It should happen inside a complete stack which embraces all the layers: it must be the missing link between the heterogeneity of the IoT and the standardized world of the Web and so of the WoT. More precisely the system has to cover the following aspects:

- **Internally** the system must be able to understand retrieved data, analyze them and perform two tasks, depending on the source of the information. If it comes from objects it must uniform data for the "clients", which will use it to create UIs and will enable the

user to have the control of the smart space. If the information comes from clients it must be able to convert it into a lower level command and send it to the right "address" in order to start the task.

- On the **"WoT side"** the system must be able to exchange the manipulated data with different third-party systems, without knowing them a priori.
- On the **"IoT side"** the middleware must be able to control the objects like every native application developed by the vendor.

So the whole idea can be summarized as an intermediate layer between layer 4 and 7 of the ISO/OSI stack, able to communicate on both sides and with a role of uniformer, making invisible the heterogeneity of the underlying world to layer 7 applications and to final users. Theoretically speaking the goal was reached: having defined a common language suitable for the purpose of describing spaces and objects, it is not difficult to create a middleware able to wrap the description into a file and able to exchange it on Internet networks. The research part was the study and the definition of "conventions" able to give an idea of smart entities, this was translated in a real set of rules to write down a JSON file. The creation of the middleware and of reliable connections was more part of the implementation, where I did not introduce innovations in the used technologies. That is because reliable and powerful technologies assure the right behavior of the whole architecture.

As proof of what just stated, in the implementation one of the goals was to prevent the user from downloading tons of different applications or from visiting many web sites. Thanks to my new system the user can concentrate the process in a single control center. It is possible to state

that the goal was completely reached, in the sense that with the proper relaxations (objects on HTTP, ad-hoc plugs) it is possible to control different IoT objects from only one unified UI. The previous chapter presents a working demo to show the result. The idea of developing a mechanism to "describe smart things" or, more completely, "describe smart spaces" has been successful. Controlling something distributed from only one point, a centralized panel, is challenging and it is what has been accomplished. The realization of the middleware has created a new possibility: all the information coming from the IoT world can pass through a "procedure". Firstly, the information is retrieved form objects, then analyzed, uniformed and finally sent to the WoT applications. The viceversa is valid for commands, instead of unifying the spoken languages, each case is differentiated. In conclusion, a sort of "production chain" has been created and passing through it can be considered a necessary step to reach the final user in a "standardized way" or to suitably control each entity without native solutions of the vendor.

I think the work which has been done is positive, interesting and promising: the idea of having a single entry point for a world which is distributed is something convenient but at the same time hard to achieve. It does not depend only on the architecture developed in this thesis but also on the substrates, on which I do not have "any power", it is the market and it has its own rules. It is impossible to make the vendors produce IoT smart objects with my requirements but Figure 29 shows how the process could be simplified if the objects came out with a uniformed definition of themselves. My framework in this case becomes only a "composer" of the right pieces of a puzzle: having the description of the smart space and of the

zones it has only to introduce the objects in the right place inside the JSON document to have the full description before presented. There would be no need of semantically understanding data in order to translate them, simplifying the internal process. The only information needed is the position of the object inside the space. The description of the smart thing can be considered a "black box", also for my middleware.

The figure shows what I have just stated: on the left column we have the current work, with my system acting with two main goals, the translation and the composition of pieces to have the entire description. On the right we have what it could like if the IoT objects standardized their output from HTTP APIs. Note that with the icon "raw data" I mean those data which are uniformed with my standard, they may be JSON files anyway. Obviously the blue rectangular shapes represent my system, a middleware between WoT and IoT.

I chose to report this figure under the "conclusion" section because it can be real only with a further development of my work and at the moment it is theoretical only. Only if the market adapts to these requirements it is possible to have the type of architecture represented on the right.

6.2 Future Work

In this section I would like to briefly report and talk about the future developments which can be done on my work, in order to extend and make the framework as universal as possible.

My development was only a proof of concept, the general idea is the creation of the "uniformed thing description" I have made for different kinds of smart objects. My thesis aims to find a univocal and uniform way to describe these new entities which are part of everyday life:

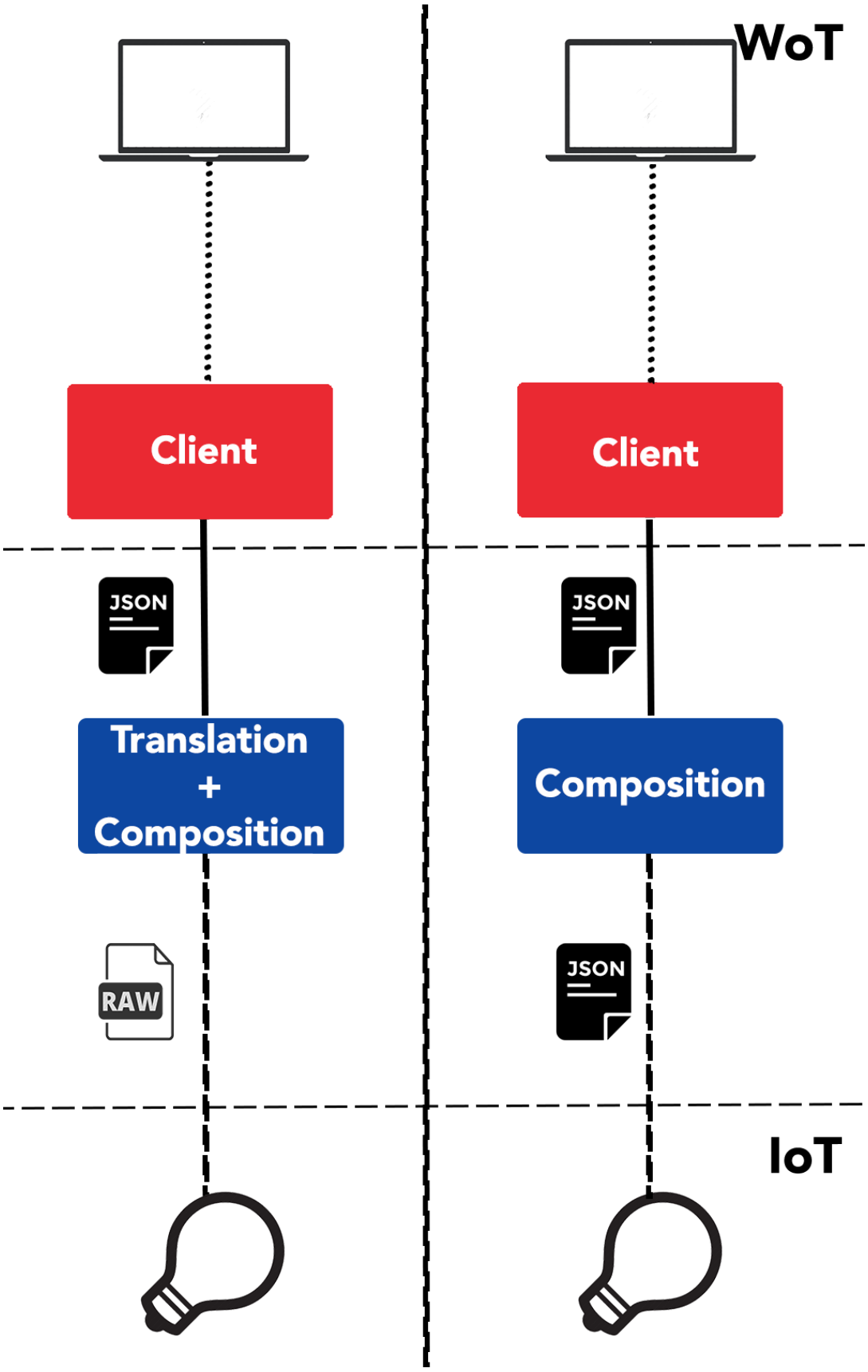

Figure 29: Today vs tomorrow functioning

the smart objects. It was not possible for reason of time to develop a system to be put on the market nowadays. It can be a part of a more complex and complete architecture which can be widely adopted. In order to have a full complete stack system which starts from the discovery of the objects and arrives to the UI for the final users, some "modules" are missing:

- **Plugs:** as already said, my application uses ad-hoc plugs to work correctly. If the market is not ready to be as the right column of the previous Figure 29 describes, it is necessary to develop more and more plugs to make the system as universal as possible. Obviously it is a "brute force" solution: every time a new object with new APIs comes out it is necessary to develop something new and write down code. But as said, the underlying layer cannot be controlled by a researcher or a student.
- **Discovery:** the discovery of the objects is very important. It enables to find new objects which are plugged into the network over time. In the system here developed I chose some objects already plugged and active, the aim was to show the uniformed language created as a possible alternative to the current way of controlling objects. In a system which will be on market it is necessary to have a discovery module for smart objects. Also for the environment it is necessary somehow to have a description of a smart space. This case is a bit more complicated: schema.org should offer the possibility to create the entity "Smart Space" (entity created in this thesis), in this case the system can simply download the configuration and wait for the objects to fill the spaces and create the complete JSON. This is only one of the possible ways to have the full description of the environment, a description which is external to the smart things.

- **Security:** the security issue is very important anywhere but particularly in the IoT where all the architecture is distributed. With a distributed architecture it is harder to control and grant access to objects, in a centralized system it can be hard but we know a priori the "borders" of what we are trying to protect. In particular I chose to run my application locally, on the same network of the objects. This can be seen as a sort of protection in the sense that an attacker should be physically there to enter the system. If we want to control the smart space remotely other problems obviously come up. Using the WoT as completion of the IoT, it is possible to juxtapose our requirements to the ones which guarantee the security of a web application or a mobile application.
- **Control center/UI:** the goal is to give the user a simpler world than today's world with tons of different applications for the IoT. To maintain the *KIS (keep it simple)* principle, it is necessary to construct a clean user friendly UI. It has to be complete in order to give full access and, at the same time, flashy to catch the user's attention. It would be a good thing maybe to develop a limited number of UIs, not to confuse the user. If all the users use the same UI it is easier for everyone to understand and be to able to use all the provided features.

CITED LITERATURE

1. Pastor-Satorras, R. and Vespignani, A.: Evolution and structure of the Internet: A statistical physics approach. Cambridge University Press, 2007.

2. Ashton, K.: That internet of things thing. RFiD Journal, 22(7):97–114, 2009.

3. Tan, L. and Wang, N.: Future internet: The internet of things. In 2010 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE), volume 5, pages V5–376. IEEE, 2010.

4. Beigl, M. and Gellersen, H.: Smart-its: An embedded platform for smart objects. In Smart Objects Conference (sOc), volume 2003, 2003.

5. Zaslavsky, A., Perera, C., and Georgakopoulos, D.: Sensing as a service and big data. arXiv preprint arXiv:1301.0159, 2013.

6. Vasseur, J.-P. and Dunkels, A.: Interconnecting smart objects with ip: The next internet. Morgan Kaufmann, 2010.

7. Guinard, D.: A web of things application architecture-Integrating the real-world into the web. Doctoral dissertation, Citeseer, 2011.

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

9. Duval, E., Hodgins, W., Sutton, S., and Weibel, S. L.: Metadata principles and practicalities. D-lib Magazine, 8(4):16, 2002.

10. Bizer, C., Heath, T., and Berners-Lee, T.: Linked data-the story so far. Semantic Services, Interoperability and Web Applications: Emerging Concepts, pages 205–227, 2009.

11. Lanthaler, M. and Gütl, C.: On using json-ld to create evolvable restful services. In Proceedings of the Third International Workshop on RESTful Design, pages 25–32. ACM, 2012.

12. W3C: A json-based serialization for linked data, 2014.

CITED LITERATURE (continued)

13. Cha, I., Shah, Y., Schmidt, A. U., Leicher, A., and Meyerstein, M. V.: Trust in m2m communication. IEEE Vehicular Technology Magazine, 4(3):69–75, 2009.

14. W3C: Wot current practices - unofficial draft, 2016.

15. Pautasso, C., Zimmermann, O., and Leymann, F.: Restful web services vs. big'web services: making the right architectural decision. In Proceedings of the 17th international conference on World Wide Web, pages 805–814. ACM, 2008.

16. Wired: Hackers remotely kill a jeep on the highwaywith me in it, 2015.

17. American, S.: Why car hacking is nearly impossible, 2016.

18. Intern, M.: Beacons: Exploring location-based technology in museums, 2015.

19. W3C: Soap 1.2 specifications, 2007.

20. W3C: Introducing json, 2007.

21. Bui, N., Castellani, A. P., Casari, P., and Zorzi, M.: The internet of energy: a web-enabled smart grid system. IEEE Network, 26(4):39–45, 2012.

22. Farooq, M., Waseem, M., Khairi, A., and Mazhar, S.: A critical analysis on the security concerns of internet of things (iot). International Journal of Computer Applications, 111(7), 2015.

23. Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E.: Wireless sensor networks: a survey. Computer networks, 38(4):393–422, 2002.

24. Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L.: Http authentication: Basic and digest access authentication. Technical report, 1999.

25. Yu, J., Benatallah, B., Casati, F., and Daniel, F.: Understanding mashup development. IEEE Internet computing, 12(5):44–52, 2008.

26. Wilde, E.: Putting things to rest. School of Information, 2007.

27. Hardt, D.: The oauth 2.0 authorization framework. 2012.

CITED LITERATURE (continued)

28. Romer, K., Ostermaier, B., Mattern, F., Fahrmair, M., and Kellerer, W.: Real-time search for real-world entities: A survey. Proceedings of the IEEE, 98(11):1887–1902, 2010.

29. Morville, P.: Ambient findability: What we find changes who we become. " O'Reilly Media, Inc.", 2005.

30. Alonso, G., Casati, F., Kuno, H., and Machiraju, V.: Web services. In Web Services, pages 123–149. Springer, 2004.

31. Rodriguez, A.: Restful web services: The basics. IBM developerWorks, 2008.

32. Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., and Weerawarana, S.: Unraveling the web services web: an introduction to soap, wsdl, and uddi. IEEE Internet computing, 6(2):86, 2002.

33. Fielding, R.: Representational state transfer. Architectural Styles and the Design of Netowork-based Software Architecture, pages 76–85, 2000.

VITA

NAME	Federico Manna
EDUCATION	
	Master of Science in Computer Science, University of Illinois at Chicago, December 2016, USA
	Master Degree in Computer Engineering , December 2016, Polytechnic of Milan, Italy
	Bachelor's Degree in Computer Engineering, September 2014, Polytechnic of Milan, Italy
LANGUAGE SKILLS	
Italian	Native speaker
English	Full working proficiency
	2013 - TOEIC examination (895/990)
	2014 - TOEFL examination (92/120)
	A.Y. 2015/16 Six months of study abroad in Chicago, Illinois
	A.Y. 2014/15, A.Y. 2015/2016 Lessons and exams attended exclusively in English
TECHNICAL SKILLS	
	Java SE, Java EE, Android SDK, Python, Javascript, HTML, CSS, C, C++, MySQL, NoSQL Databases, SAPbyDesign