Design of a community sensor network

Joan Salvatella Ibáñez

May 8, 2014

MASTER THESIS
Màster en Enginyeria Informàtica
Facultat d’Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Director: Jorge García Vidal
Departament d’arquitectura de computadors
Abstract

Sensors that are part of the Internet of Things produce a lot of useful data that nowadays is only available for the owners of said sensors. The main goal of this project is to create a platform capable of storing all the data produced by those sensors and provide a RESTful API to share it in a standardized form using RDF and making use of the benefits of the Linked Data paradigm. The system will also include a website from which the sensors information and data will be able to be managed.

Resumen

Los sensores que forman parte de la Internet of Things producen una gran cantidad de información útil que actualmente solo está disponible para los propietarios de dichos sensores. El objetivo principal de este proyecto es crear una plataforma capaz de almacenar esos datos y proveer una API Restful para compartirlos de forma estandarizada usando RDF y aprovechando los beneficios de la paradigma Linked Data. El sistema también incluirá una página web con la que gestionar la información de los sensores y los datos recogidos.
Acknowledgment

I would like to thank my supervisor Professor Jorge Garcia and Professor Jose Maria Barceló for all their support, patient guidance and enthusiastic encouragement. I am particularly grateful to Òscar Trullols for his patience and useful critiques of this research work. I would like to offer my special thanks to Ismael and Farnoosh for their valuable advise and assistance.

Finally, I wish to thank my family and friends for their support throughout my study.

Joan Salvatella Ibáñez
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>CommonSense development Gantt chart</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Simplified SSN Ontology graph</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Entities conceptual map</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>CommonSense ontology based model</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>CommonSense system architecture</td>
<td>23</td>
</tr>
<tr>
<td>5.2</td>
<td>CommonSense development Gantt chart</td>
<td>24</td>
</tr>
<tr>
<td>5.3</td>
<td>API work-flow</td>
<td>28</td>
</tr>
<tr>
<td>5.4</td>
<td>Sensor data output example in graphical view</td>
<td>31</td>
</tr>
<tr>
<td>5.5</td>
<td>Website view structure map</td>
<td>38</td>
</tr>
<tr>
<td>5.6</td>
<td>User control panel</td>
<td>39</td>
</tr>
<tr>
<td>5.7</td>
<td>Administrator control panel</td>
<td>39</td>
</tr>
<tr>
<td>5.8</td>
<td>API output recollected in a graph</td>
<td>40</td>
</tr>
<tr>
<td>5.9</td>
<td>View all posts view</td>
<td>40</td>
</tr>
<tr>
<td>5.10</td>
<td>Add stream view</td>
<td>41</td>
</tr>
<tr>
<td>5.11</td>
<td>Edit project view</td>
<td>41</td>
</tr>
</tbody>
</table>
# Contents

Abstract ................................................................. i
Resumen ................................................................. i
Acknowledgment ...................................................... ii

## 1 Introduction

1.1 CommonSense .................................................. 2
1.2 Competition ...................................................... 3
   1.2.1 Xively .................................................. 3
   1.2.2 Carriots ............................................... 4
1.3 Document structure ............................................ 4

## 2 Goals and planification

2.1 Initial goals and planification ................................. 5
2.2 Final goals and planification .................................. 6

## 3 Technological context

3.1 Semantic Web .................................................. 8
3.2 Linked Data .................................................... 8
3.3 RDF .......................................................... 9
3.4 Ontologies ..................................................... 9
   3.4.1 SSN ................................................... 9
   3.4.2 GEO .................................................. 10
   3.4.3 FOAF ............................................... 10
   3.4.4 CC .................................................. 10
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.5 DC</td>
<td>10</td>
</tr>
<tr>
<td>3.4.6 DUL</td>
<td>10</td>
</tr>
<tr>
<td>3.5 Django</td>
<td>11</td>
</tr>
<tr>
<td>3.6 Bootstrap</td>
<td>11</td>
</tr>
<tr>
<td>3.7 API</td>
<td>11</td>
</tr>
<tr>
<td>3.8 REST</td>
<td>12</td>
</tr>
<tr>
<td>3.9 NodeJS</td>
<td>12</td>
</tr>
<tr>
<td>3.10 Express</td>
<td>12</td>
</tr>
<tr>
<td>4 Design</td>
<td>14</td>
</tr>
<tr>
<td>4.1 Initial model</td>
<td>14</td>
</tr>
<tr>
<td>4.2 Final model</td>
<td>15</td>
</tr>
<tr>
<td>4.2.1 Ontology based model</td>
<td>17</td>
</tr>
<tr>
<td>4.2.2 Final and former model mapping</td>
<td>21</td>
</tr>
<tr>
<td>5 Implementation</td>
<td>22</td>
</tr>
<tr>
<td>5.1 Architecture</td>
<td>22</td>
</tr>
<tr>
<td>5.2 Database</td>
<td>23</td>
</tr>
<tr>
<td>5.2.1 Tables</td>
<td>23</td>
</tr>
<tr>
<td>5.2.2 Views</td>
<td>25</td>
</tr>
<tr>
<td>5.3 API</td>
<td>27</td>
</tr>
<tr>
<td>5.3.1 POST</td>
<td>28</td>
</tr>
<tr>
<td>5.3.2 GET</td>
<td>30</td>
</tr>
<tr>
<td>5.4 Website</td>
<td>34</td>
</tr>
<tr>
<td>5.5 Environments</td>
<td>36</td>
</tr>
<tr>
<td>5.5.1 Development environment</td>
<td>36</td>
</tr>
<tr>
<td>5.5.2 Production environment</td>
<td>37</td>
</tr>
<tr>
<td>6 Conclusions and future work</td>
<td>42</td>
</tr>
<tr>
<td>A Acronyms</td>
<td>44</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The rise in the development of technological devices and the consolidation of Internet as the main way to communicate and interact with the rest of the world has led to the birth of the Internet of Things. Sensors that are part of the IoT produce a lot of useful data that nowadays is only available for the owners of said sensors.

The main goal of this project is to create a platform capable of storing all the data produced by those sensors and provide a RESTful API to share it in a standardized form using RDF and making use of the benefits of the Linked Data paradigm. By doing so, studies will be able to be completed with sufficient information without the need to spend millions on the deployments of thousands of sensors worldwide. The system will also include a website from which the sensors information and data will be able to be managed.

1.1 CommonSense

CommonSense is an open link data community sensing platform with the aim of integrating users that generate information and users that are able to use it. This platform plays a key role in Smart Cities where the availability and quality of the data is one of the most important factors to work on.

What makes appealing the use of this platform is that the sensors can be easily connected without any technical knowledge which facilitates deployment tasks and it also facilitates a RESTful API in order to supply all the data in order to ease the development of applications.
The user can use the data for his own purpose, or can decide to share his data with other users, organizations, or groups of interest (e.g. You could choose to share the data with ecologist organizations or with neighborhoods but not with private companies). To make easier the reuse of data, the system uses standardized protocols and data formats (e.g. it can standardize the predefined words that must be used to obtain a given data such as \( <\text{temperature, } ^\circ\text{C}> \), \( <\text{ozone, mg/m}^3> \)).

The data may be integrated with data coming from other repositories, forming part of what is called Open Link Data (OLD) in the context of semantic web. This can be achieved due to the fact that data is supplied using RDF, a general method for conceptual description or modeling of information that is implemented in web resources, and data is compliant with standard sensor data ontologies like Semantic Sensor Network (SSN) and many other ontologies described in chapter 3.

The number of applications are numerous and can range topics such as reduction of carbon fossil fuels in domestic and industrial environments, smart electricity distribution networks, reduction of \( CO_2 \) emissions, attentional prevention to the elder, urban resilience or adaptation to the climatic change.

### 1.2 Competition

Internet of Things has been a hot topic in the last years and as proof of that there is the Seventh Framework Programme (FP7) unit called the Internet of Things and Future Internet Enterprise Systems which aimed to support research for the development of the Internet of Things. There are several companies that identified a niche with potential profit so they started working on it. Xively and Carriots are a good example of that as they both developed their own platforms that resemble CommonSense even though they sought different goals.

#### 1.2.1 Xively

Xively (formerly known as Cosm and Pachube) is a division of LogMeIn Inc (LOGM), a global, public company that is a leading provider of essential remote services. Organizations worldwide rely on LogMeIn’s suite of SaaS customer care, remote IT management, access and collaboration
products. Xively is built on LogMeIn's cloud platform Gravity, which handles over 255 million
devices, users and customers across 7 datacenters worldwide.

The main difference between Xively's platform and CommonSense is that it doesn't homogenize the incoming data. Even though they allow the owners of the data to choose if the data is available worldwide or just to them, if it's not homogenized it can't be used along other heterogeneous sources. The process of data processing must be done individually for each source or it needs a prior mapping process to standardize it.

### 1.2.2 Carriots

Carriots is the spin-off of Waterbut's M2M division, a company that has over 10 years experience in engineering IT solutions for large international companies. Prior to founding Carriots, the division expertise in M2M was backed up by engineering contracts with industry leaders like Telefónica Group or Vodafone Group to deliver end-to-end solutions or consultancy services.

One of the aspects that differentiate Carriots' platform from CommonSense is that it's not free. There is a free starting plan which lets you manage up to 10 devices but if you need more than that you will have to pay. Another aspect to take into account is that the users can't share their data publicly and also, just like Xively's platform, they don't standardize the incoming data.

### 1.3 Document structure

In this chapter an introduction to the project has been reviewed focusing on what is intended to do and what there's already done by third-party companies. In chapter 2 the project goals are described with the help of a Gantt chart. In chapter 3 the technological context on which the project is located in is briefly explained. Chapter 4 focuses on describing the model on which all the services are based. Chapter 5 details how all the services are implemented and provides some examples. Finally, chapter 6 describes the final conclusions and the future work guidelines.
Chapter 2

Goals and planification

2.1 Initial goals and planification

CommonSense was initially divided into eight goals which are listed in Table 2.1. Between all the tasks there were resource constraints which in this case is manpower since I was the one that had to do all of them regardless the help my supervisors provided and there were also task dependency constraints. This task dependencies are defined as: T2 couldn't start until T1 had been completed. Once T2 was completed, T3, T6 or T7 could start. The only requirement for T8 was to start only if T6 or T7 had started since it involved the documentation of the code being developed in those tasks.

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Model the data storage requirements for the sensors</td>
<td>3d</td>
</tr>
<tr>
<td>T2</td>
<td>Implement said model in a database</td>
<td>5d</td>
</tr>
<tr>
<td>T3</td>
<td>Design a web interface to manage the data</td>
<td>2d</td>
</tr>
<tr>
<td>T4</td>
<td>Design a web interface to display what the project is about</td>
<td>5d</td>
</tr>
<tr>
<td>T5</td>
<td>Implement the website</td>
<td>60d</td>
</tr>
<tr>
<td>T6</td>
<td>Design and implement an API to supply data to smartphones</td>
<td>10d</td>
</tr>
<tr>
<td>T7</td>
<td>Design and implement an API to gather standardized data from wireless sensors</td>
<td>5d</td>
</tr>
<tr>
<td>T8</td>
<td>Document the API</td>
<td>5d</td>
</tr>
</tbody>
</table>

Table 2.1: Project goals
2.2 Final goals and planificaton

In the middle of the project development it was decided that the system should be improved so that all the communications using the API ought to be in RDF. This fact extended the task list by adding 4 more tasks which are listed in Table 2.2. The dependencies added with these tasks were: T10 needed the completion of T9, T11 needed the completion of T10 and T12 needed the completion of T11.

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>T9</td>
<td>Study the ontologies</td>
<td>5d</td>
</tr>
<tr>
<td>T10</td>
<td>Model the system using the available ontologies</td>
<td>4d</td>
</tr>
<tr>
<td>T11</td>
<td>Map the ontology based model with the previous model</td>
<td>2d</td>
</tr>
<tr>
<td>T12</td>
<td>Rewrite the API to support the RDF communications</td>
<td>8d</td>
</tr>
</tbody>
</table>

Table 2.2: Project goals extension

T1, T4 and T9 could be started the day 0 but T9 wasn't in the scope when the first task to work on had to be decided. Between T1 and T4 it was decided to start with T1 arbitrarily. The final task path to follow was defined as T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11 and finally T12. The Gantt chart is displayed in Figure 2.1

It all should have added up to 114 days, starting on September 1st it was estimated that the finalization date would be February 6th assuming that no work would be done during the weekends. In the end, there were some days that I couldn't work on the project because I had exams or I was required to attend to some subjects laboratory practices. This caused that the project didn't follow the schedule to perfection but I worked the following days some extra hours in order to keep the project in the right track. All tasks were performed within the estimated time, with 1 or 2 days margin, with the exception of T5 which its finalization was relegated to the background as the other tasks were more important. The project finally ended on February 14th.
Figure 2.1: CommonSense development Gantt chart
Chapter 3

Technological context

In this chapter all the technologies used in the project and the relevant concepts for the understanding of this thesis are briefly described.

3.1 Semantic Web

The Semantic Web is a project that intends to add computer-processable meaning (semantics) to the World Wide Web. In February 2004, The W3C released the Resource Description Framework (RDF) and the OWL Web Ontology Language (OWL) as W3C Recommendations. RDF is used to represent information and to exchange knowledge in the web. OWL is used to publish and share sets of terms called ontologies, supporting advanced Web search, software agents and knowledge management.

3.2 Linked Data

The term Linked Data refers to a set of best practices for publishing and connecting structured data on the Web. Key technologies that support Linked Data are URIs (a generic means to identify entities or concepts in the world), HTTP (a simple yet universal mechanism for retrieving resources, or descriptions of resources), and RDF.
### 3.3 RDF

Resource Description Framework (RDF) is an abstract model, a way to break down knowledge into discrete pieces, with some rules about the semantics, or meaning, of those pieces. While it is most popularly known for its RDF/XML syntax, RDF can be stored in a variety of formats. Examples of RDF serialisations include RDF/XML, Notation-3 (N3), Turtle, N-Triples, RDFa, and RDF/JSON. Although RDFs enables the definition of classes, relationships and attributes, these features are in most of the cases insufficient to model a full ontology because of the existence of certain constraints (like cardinality or class disjointness among others). That’s the reason why the Web Ontology Language (OWL) was defined.

### 3.4 Ontologies

A specification of a representational vocabulary for a shared domain of discourse—definitions of classes, relations, functions, and other objects—is called an ontology. An ontology is useful for sharing common understanding of the structure of information among people or software agents, enabling reuse of domain knowledge, making domain assumptions explicit, separating domain knowledge from the operational knowledge and analyzing domain knowledge. In our case, ontologies are used specially for its ability to share common understanding of the structure of information among people or software agent. The ontologies used are SSN for the representation of sensor metadata, GEO for the geospatial data representation, FOAF for the representation of users, CC for the copyright information, DC for describing generic metadata and DUL as the supporting upper-level ontology. Upper level ontologies are used to facilitate the semantic integration of domain ontologies and guide the development of new ontologies. For this purpose, they contain general categories that are applicable across multiple domains.

#### 3.4.1 SSN

The SSN ontology is based around concepts of systems, processes, and observations. It supports the description of the physical and processing structure of sensors. Sensors are not constrained to physical sensing devices: rather a sensor is anything that can estimate or calculate the value
of a phenomenon, so a device or computational process or combination could play the role of a sensor. The representation of a sensor in the ontology links together what it measures (the domain phenomena), the physical sensor (the device) and its functions and processing (the models). In Figure 3.1 there’s a simplified version of the ontology.

### 3.4.2 GEO

The GeoNames Ontology makes it possible to add geospatial semantic information to the World Wide Web. All over 8.3 million geonames toponyms now have a unique URL with a corresponding RDF web service. In our case it’s used for describing the latitude, altitude and longitude of a sensing device.

### 3.4.3 FOAF

FOAF is a machine-readable ontology describing persons, their activities and their relations to other people and objects. It is used to describe people and social relationship on the Web and it’s mostly focused on people’s existence in the virtual world. In the CommonSense system it’s used for describing users and projects.

### 3.4.4 CC

The Creative Commons Rights Expression Language (CC REL) lets you describe copyright licenses in RDF. In our case it’s used for describing the license under which a project is registered.

### 3.4.5 DC

Dublin Core is a light weight RDFS vocabulary for describing generic metadata. It is ubiquitous in the linked data landscape.

### 3.4.6 DUL

DUL is a lighter OWL axiomatization of DOLCE and DnS which also simplifies the names of many classes and properties, adds extensive inline comments, and thoroughly aligns to the
repository of Content patterns. DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) is the first module of the WonderWeb foundational ontologies library. It has a clear cognitive bias, in that it aims at capturing the ontological categories underlying natural language and human common sense. DnS (Descriptions and Situations) is a constructivist ontology that pushes DOLCE’s descriptive stance even further. DnS does not put restrictions on the type of entities and relations that one may want to postulate, either as a domain specification, or as an upper ontology, and it allows for context-sensitive redescriptions of the types and relations postulated by other given ontologies.

3.5 Django

Django is an open source web application framework, written in Python, which follows the model-view-controller architectural pattern. It was originally developed to manage several news-oriented sites for The World Company of Lawrence, Kansas, and was released publicly under a BSD license in July 2005. Django’s primary goal is to ease the creation of complex database-driven websites. It emphasizes on reusability, pluggability of components and rapid development.

3.6 Bootstrap

Bootstrap is a free collection of tools for creating websites and web applications developed by Mark Otto for Twitter. It contains HTML and CSS-based design templates for typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions.

3.7 API

An application-programming interface (API) is a set of programming instructions and standards for accessing a web-based software application. A software company releases its API to the public so that other software developers can design products that are powered by its service.
3.8 REST

Representational state transfer (REST) is a software architectural style consisting of a coordi-
nated set of architectural constraints applied to components, connectors, and data elements, 
within a distributed hypermedia system. RESTful APIs are defined with these aspects:

- base URI such as http://commonsense.pc.ac.upc.edu/
- an Internet media type for the data which is often JSON
- standard HTTP methods (e.g., GET, PUT, POST, or DELETE)
- hypertext links to reference state
- hypertext links to reference related resources

3.9 NodeJS

Node.js is a platform built on Chrome’s JavaScript runtime for easily building fast, scalable net-
work applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight 
and efficient, perfect for data-intensive real-time applications that run across distributed de-
vices. Web development in a dynamic language as JavaScript on a virtual machine like V8 is 
much faster than Ruby, Python, or Perl. It has the ability to handle thousands of concurrent 
connections with minimal overhead on a single process.

3.10 Express

Express is a minimal and flexible node.js web application framework, providing a robust set of 
features for building single and multi-page, and hybrid web applications. It provides a myriad 
of HTTP utility methods and Connect middleware which allow creating a robust user-friendly 
API be quick and easy.
CHAPTER 3. TECHNOLOGICAL CONTEXT

Figure 3.1: Simplified SSN Ontology graph
Chapter 4

Design

So as to store all the information that the sensors gather it is required to create a model determining which entities from the real world exist and act in our system. Given that after the project started it was decided that all the communications from the system to the outside should be done in RDF the first proposed model, described in section 4.1, wasn’t the most suitable for the task. By the point this decision was made the implementation of this model in a MySQL database, as it will be described in section 5.2, was already finished. It was estimated that it would require less time to add an intermediate layer in all communications between the system and the outside aiming to map the final model to the former already implemented than redoing the whole system.

4.1 Initial model

The first model designed is explained in this simple conceptual map in Figure 4.1. A user can create a project or join an existing one which was created by another user. This relationship is determined by a role which can be: Administrator, only the administrators can control the users that join the project and they also can send and read all the sensor data related to the project; Collaborator, collaborators can send and read all the sensor data related to the project; Observer, observers are only allowed to read the sensor data related to the project. A project has several streams of data which are considered a collection of data related to each other coming from one unique source e.g. a sensor may have 2 streams where in the first one the battery level
measurements are sent and in the latter the humidity level in the air.

Users need to register their nodes so as to be able to send the data to the system and when this is done the ownership of the node is established. This is important in order to determine which user can modify the node properties and which user should be notified when the system detects that the node is not responsive anymore. Node types are used to cluster nodes by their capabilities. Data meta-data is stored as data type and it's connected to the actual data.

In Table 4.1 the attributes of each entity in the model explained before are listed. It's important to emphasize that in the node entity there's a geopositioning redundancy since the city, country and region could be obtained through the latitude and longitude but it was decided to keep both ways of expressing it in order to make it easier for developers.

4.2 Final model

In subsection 4.2.1 the new ontology-based model is introduced and in subsection 4.2.2 the mapping between the former and the latter is explained.
<table>
<thead>
<tr>
<th>Entity</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>first name</td>
</tr>
<tr>
<td></td>
<td>last name</td>
</tr>
<tr>
<td></td>
<td>email</td>
</tr>
<tr>
<td></td>
<td>APIkey</td>
</tr>
<tr>
<td>Project</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>License</td>
</tr>
<tr>
<td>Node</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>Timezone</td>
</tr>
<tr>
<td></td>
<td>Status</td>
</tr>
<tr>
<td></td>
<td>Serial Number</td>
</tr>
<tr>
<td></td>
<td>Enabled</td>
</tr>
<tr>
<td></td>
<td>Latitude</td>
</tr>
<tr>
<td></td>
<td>Longitude</td>
</tr>
<tr>
<td></td>
<td>Altitude</td>
</tr>
<tr>
<td></td>
<td>City</td>
</tr>
<tr>
<td></td>
<td>Country</td>
</tr>
<tr>
<td></td>
<td>Region</td>
</tr>
<tr>
<td>NodeType</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>Stream</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td></td>
<td>Last update</td>
</tr>
<tr>
<td>Data</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td>Date</td>
</tr>
<tr>
<td>DataType</td>
<td>Name</td>
</tr>
<tr>
<td></td>
<td>Units</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
</tbody>
</table>

Table 4.1: Model entities attributes
4.2.1 Ontology based model

In order to represent the data in RDF several ontologies were studied and they are briefly explained in section 3.4. All the information required to represent the system was modeled using those ontologies and the result can be seen in Figure 4.2. Each of the entities and their relationships will now be described:

![CommonSense ontology based model](image)

**ssn:Observation** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Observation](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Observation))

An Observation is a Situation in which a Sensing method has been used to estimate or calculate a value of a Property of a FeatureOfInterest. Links to Sensing and Sensor describe what made the Observation and how; links to Property and Feature detail what was sensed; the result is the output of a Sensor; other metadata details times etc.

*Attributes:*

- `rdfs:label`
- `ssn:observationResultTime`

**ssn:Property** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Property](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Property))

An observable Quality of an Event or Object. That is, not a quality of an abstract entity as is
also allowed by DUL's Quality, but rather an aspect of an entity that is intrinsic to and cannot exist without the entity and is observable by a sensor.

Attributes:

<table>
<thead>
<tr>
<th>rdfs:label</th>
<th>rdfs:comment</th>
</tr>
</thead>
</table>

**ssn:SensorOutput** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#SensorOutput](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#SensorOutput))

A sensor outputs a piece of information (an observed value), the value itself being represented by an ObservationValue.

**DUL:Amount** ([http://purl.org/ifgi/dul#Amount](http://purl.org/ifgi/dul#Amount))

A quantity, independently from how it is measured, computed, etc.

**DUL:UnitsOfMeasure** ([http://purl.org/ifgi/dul#UnitOfMeasure](http://purl.org/ifgi/dul#UnitOfMeasure))

Units of measure are conceptualized here as parameters on regions, which can be valued as datatype values.

**ssn:SensingDevice** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#SensingDevice](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#SensingDevice))

A sensing device is a device that implements sensing.

Attributes:

<table>
<thead>
<tr>
<th>rdfs:label</th>
<th>rdfs:comment</th>
</tr>
</thead>
</table>

**ssn:Deployment** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Deployment](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Deployment))

The ongoing Process of Entities (for the purposes of this ontology, mainly sensors) deployed for a particular purpose. For example, a particular Sensor deployed on a Platform, or a whole network of Sensors deployed for an observation campaign. The deployment may have sub processes, such as installation, maintenance, addition, and decommissioning and removal.

**ssn:Platform** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Platform](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#Platform))

An Entity to which other Entities can be attached - particularly Sensors and other Platforms. For example, a post might act as the Platform, a bouy might act as a Platform, or a fish might act as a Platform for an attached sensor.

**geo:SpatialThing** ([http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing](http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing))
A point, typically described using a coordinate system relative to Earth, such as WGS84. Uniquely identified by latitude, longitude and altitude.

Attributes:
- geo:lat
- geo:long
- geo:alt


The Project class represents the class of things that are 'projects'. These may be formal or informal, collective or individual. It is often useful to indicate the homepage of a Project.

Attributes:
- dc:title
- dc:description
- cc:license

**foaf:Agent** ([http://xmlns.com/foaf/spec/#term_Agent](http://xmlns.com/foaf/spec/#term_Agent))

The Agent class is the class of agents; things that do stuff.

Attributes:
- foaf:name
- foaf:nick
- foaf:mbox

**addr:Address** ([http://wiki.foaf-project.org/w/AddressVocab](http://wiki.foaf-project.org/w/AddressVocab))

The Address class represents a region in space using political boundaries.

Attributes:
- addr:thoroughfareName
- addr:town
- addr:region
- addr:country
**ssn:ObservationResult** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#observationResult](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#observationResult))

Relation linking an Observation (i.e., a description of the context, the Situation, in which the observation was made) and a Result, which contains a value representing the value associated with the observed Property.

**ssn:ObservedProperty** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#observedProperty](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#observedProperty))

Relation linking an Observation to the Property that was observed. The observedProperty should be a Property (hasProperty) of the FeatureOfInterest (linked by featureOfInterest) of this observation.

**ssn:isProducedBy** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#isProducedBy](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#isProducedBy))

Relation between a producer and a produced entity: for example, between a sensor and the produced output.

**ssn:hasValue** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#hasValue](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#hasValue))

**DUL:isClassifiedBy** ([http://purl.org/ifgi/dul#isClassifiedBy](http://purl.org/ifgi/dul#isClassifiedBy))

A relation between a Concept and an Entity

**ssn:hasDeployment** ([http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#hasDeployment](http://www.w3.org/2005/Incubator/ssn/ssnx/ssn#hasDeployment))

Relation between a System and a Deployment, recording that the System/Sensor was deployed in that Deployment.


Relation between a deployment and the platform on which the system was deployed.

**DUL:hasLocation** ([http://purl.org/ifgi/dul#hasLocation](http://purl.org/ifgi/dul#hasLocation))

A generic, relative localization, holding between any entities.


A currentProject relates a Person to a Document indicating some collaborative or individual undertaking. This relationship indicates that the Person has some active role in the project, such as development, coordination, or support.

The maker property relates something to an Agent that made it.

### 4.2.2 Final and former model mapping

Using the information stored in the database, which had been implemented following the first model, it was required to add a mapping layer between models. Since both models represent the same part of the world but defining different relations and entities there should exist a mapping function between them. Table 4.2 shows from which former entities implementation do the new entities gather the information. Those new entities that aren't in the table don't store any literals but they are the vehicle to relate other entities that store them.

<table>
<thead>
<tr>
<th>Final</th>
<th>Former</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn:SensingDevice</td>
<td>Node</td>
</tr>
<tr>
<td>geo:SpatialThing</td>
<td>Node</td>
</tr>
<tr>
<td>foaf:Agent</td>
<td>User</td>
</tr>
<tr>
<td>foaf:Project</td>
<td>Project</td>
</tr>
<tr>
<td>DUL:UnitsOfMeasure</td>
<td>DataType</td>
</tr>
<tr>
<td>DUL:Amount</td>
<td>Data</td>
</tr>
<tr>
<td>ssn:SensorOutput</td>
<td>Stream</td>
</tr>
<tr>
<td></td>
<td>Node</td>
</tr>
<tr>
<td>ssn:Observation</td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>DataType</td>
</tr>
<tr>
<td>ssn:Property</td>
<td>DataType</td>
</tr>
</tbody>
</table>

Table 4.2: Final and former models mapping
Chapter 5

Implementation

The goal of this chapter is to explain how the system was implemented and show some examples. All the code developed for CommonSense is stored in the Git repository located at https://www.ac.upc.edu/projects/commonsense/git/commonsense and in Appendix B it's explained how to obtain it. In section 5.1 the system architecture is addressed and in sections 5.2, 5.3 and 5.4 each subsystem is explained in detail. In projects of this scale is important to define several environments in order to allow the developers to work on software enhancements without disturbing the normal operation. All the environments are described in section 5.5.

5.1 Architecture

The CommonSense system architecture is represented in Figure 5.1 and it is divided in 3 main subsystems: a MySQL database, a NodeJS API and a website built on Django and served by an Apache server. The MySQL database is the part of the system that's in charge of storing all the data generated by the sensors and also all the necessary meta-data to manage them. Any app working on an Android-based device can connect to the data offered by the system using RESTful URL queries through the NodeJS API developed using the Express web application framework. The sensors from which the systems gathers the data from also use the API to send the data. It's important to note that all the data sent from and to the API are valid XML/RDF files. And the last part that forms CommonSense is a website with the aim of informing users about what the project is about and also to give a user-friendly interface from which users can manage
their sensors and data.

Figure 5.1: CommonSense system architecture

### 5.2 Database

For the database implementation MySQL was chosen because of its scalability and flexibility against other database servers. Another important reason was its high performance and availability in conjunction with the fact that I already worked on it.

#### 5.2.1 Tables

Figure 5.2 represents the MySQL implementation of the model described in section 4.1. Every box represents a table in the database stating all the columns and its data type. Every table has at least one small yellow key that represents the primary key. The red dot symbolizes that this column is a foreign key to another table. In Table 5.1 there's a brief description of each table.

In the database there are also the tables that are collected in Table 5.2 and Table 5.3 which
Figure 5.2: CommonSense development Gantt chart
Table 5.1: Database model tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Information about the data</td>
</tr>
<tr>
<td>DataTypes</td>
<td>Information about the data types</td>
</tr>
<tr>
<td>NodeTypes</td>
<td>Information about the node types</td>
</tr>
<tr>
<td>Nodes</td>
<td>Information about the nodes</td>
</tr>
<tr>
<td>Projects</td>
<td>Information about the projects</td>
</tr>
<tr>
<td>ProjectsStatus</td>
<td>Name of all the possible projects status</td>
</tr>
<tr>
<td>ProjectsStreams</td>
<td>Relationship between a project and a stream</td>
</tr>
<tr>
<td>Roles</td>
<td>Name of all the possible roles</td>
</tr>
<tr>
<td>Streams</td>
<td>Information about the streams</td>
</tr>
<tr>
<td>UsersProjects</td>
<td>Relationship between a project and a user</td>
</tr>
<tr>
<td>UserProfiles</td>
<td>Extension of the django default user</td>
</tr>
<tr>
<td>Conversions</td>
<td>Information about how to transform from one unit of measure to another</td>
</tr>
</tbody>
</table>

aren't in the model and that's because they were included by Django for the administration dashboard that are initially provided by default and the ones that Zinnia, a blogging app, included.

5.2.2 Views

In order to hide complexity in the API queries 3 views were designed and developed.
Table 5.3: Zinnia blogging tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tagging_tag</td>
<td>Information about the tags</td>
</tr>
<tr>
<td>tagging_taggeditem</td>
<td>Relationship between a tag and an entry</td>
</tr>
<tr>
<td>zinnia_category</td>
<td>Information about a category</td>
</tr>
<tr>
<td>zinnia_entry</td>
<td>Information about a blog entry</td>
</tr>
<tr>
<td>zinnia_entry_authors</td>
<td>Relationship between a blog post and a user</td>
</tr>
<tr>
<td>zinnia_entry_categories</td>
<td>Relationship between a blog post and a category</td>
</tr>
<tr>
<td>zinnia_entry_related</td>
<td>Relationship between a blog post and another blog post</td>
</tr>
<tr>
<td>zinnia_entry_sites</td>
<td>Relationship between a blog post and a dashboard</td>
</tr>
</tbody>
</table>

**AuthorizedView**

This view gets all the required attributes in order to check if the user is authorized to do a POST operation.

```sql
CREATE VIEW AuthorizedView AS
SELECT u.username, pup.Value AS 'APIkey', p.name AS 'projectname',
       up.role_id, s.Name AS 'streamname'
FROM auth_user AS u
LEFT OUTER JOIN UserProfiles AS pup
    ON u.id=pup.user_id
LEFT OUTER JOIN UsersProjects AS up
    ON u.id=up.user_id
LEFT OUTER JOIN Projects AS p
    ON p.id=up.project_id
LEFT OUTER JOIN ProjectsStreams AS ps
    ON ps.project_id=p.id
LEFT OUTER JOIN Streams AS s
    ON s.id=ps.stream_id
WHERE up.status=1;
```

**FeasableProjectsView**

This view aims to merge all the public available projects with the active private ones.
CREATE VIEW FeasableProjectsView AS

SELECT up.status AS 'status', up.role_id, p.name AS 'project_name',
    p.id AS 'project_id', up.user_id
FROM Projects AS p
    LEFT OUTER JOIN UsersProjects AS up
    ON up.project_id=p.id
WHERE p.License=0 or up.user_id IS NOT NULL;

AuthorizedGetView

This views gets all the required attributes in order to check if the user is authorized to do a GET operation.

CREATE VIEW AuthorizedGetView AS

SELECT u.username, pup.Value AS 'APIkey', fpv.project_id AS 'project_id', fpv.role_id
FROM auth_user AS u
    LEFT OUTER JOIN UserProfiles AS pup
    ON u.id=pup.user_id
    LEFT OUTER JOIN FeasableProjectsView AS fpv
    ON u.id=fpv.user_id or fpv.user_id IS NULL
WHERE fpv.status=1 or fpv.user_id IS NULL;

5.3 API

At the beginning of the project it was discussed whether the API should be also developed in Python like the website. After studying both technologies it was decided that NodeJS in conjunction with the Express web application framework would be the best option because it gave faster responses and the API development was easier and therefore the development time would be inferior.
The CommonSense API serves two purposes: To store the data that the sensors send to the system and to answer to requests that are used to query the data stored in the database. The first kind of requests are always sent using the POST HTTP method and the latter are always sent using the GET HTTP method. There has been implemented a security layer on top of both methods in order to protect the system from fraudulent data insertion and to protect the user’s data that don’t want to share.

This security layer is provided by the username and the given API key when the user is registered to the system. Nowadays it’s implemented as a better password system but a signature should be calculated instead as $S = F(K, R)$, where $K$ is the API key and $R$ is the entire request, including all request parameters. The $F$ function should be a secure message authentication code algorithm, such as AES-CMAC or SHA1-HMAC. The API work-flow is shown in Figure 5.3.

Figure 5.3: API work-flow

5.3.1 **POST**

When a sensor has data to insert to the system it forms a HTTP POST request with a valid RDF document containing the data, the username and the API key in the headers and 3 parameters
in the URL: the API version, the project name and the stream name. The first parameter was
introduced for compatibility reasons since that if some nodes where deployed with the current
version of the API but after some time the API was extended and the system changed the old
nodes would stop working. If an API version is included in the URL both versions can work
simultaneously. The project name and the stream name parameters are necessary to identify
where the data should be stored to.

Below is an example of a valid RDF/XML document sent from a sensor to the system with the
URL http://commonsense.pc.ac.upc.edu/v1/ProjectA/StreamA where v1 is the API ver-
sion, Project A is the name of the project and Stream A is the name of the stream.

```
POST /v1/ProjectA/StreamA HTTP/1.1
Host: commonsense.pc.ac.upc.edu
X-APIKey: hBTlrl5x8vrWXJfIqlxK0BqGb2nV3gJNILtAzx0uKY
User: jsalvatella
Content-Type: rdf+xml
Cache-Control: no-cache

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:ssn="http://purl.oclc.org/NET/ssnx/ssn#"
xmlns:DUL="http://www.loa-cnr.it/ontologies/DUL.owl#">
<ssn:Observation>
<ssn:observationResultTime>20131217140046</ssn:observationResultTime>
<ssn:observedProperty rdf:resource="http://commonsense.pc.ac.upc.
edu:3000/properties/battery"/>
<ssn:observationResult>
<ssn:SensorOutput>
<ssn:isProducedBy rdf:resource="http://commonsense.pc.ac.upc.
edu:3000/sensors/2"/>
<ssn:hasValue>
<DUL:Amount>
```
When the system receives the request, it checks that the user authenticated with the API key has the right permissions to perform an insert operation on the given project and the given stream. Finally, the data is extracted from the POST request, inserted to the database and a HTTP 200 OK message is sent back.

5.3.2 GET

A HTTP GET request needs to be created in order to retrieve data from the system. This request should include the username and the API key in the headers as the POST request. Unlike the POST request, there are multiple ways of performing this operation depending on the entity you want to obtain and on the constraints you are applying. The required parameters are different if all the sensors from a project are requested in comparison to those needed to obtain the last measure of the humidity sensor closest to the White House.

The API has been designed to allow the permutation of the parameters in the URL thus making the order not matter. All the examples provided below are referenced to the base URL http://commonsense.pc.ac.upc.edu in order to make them shorter.

Observations

- URL: /observations
• Input parameters: sensors, projects, samplingTimes, observedproperties and geo.

The parameter sensor is used to determine the sensor from which the data is obtained by providing its id. The parameter projects is used to determine from which project the data will be obtained from providing its id. The parameter samplingTimes may be used in 2 different ways: It can be used as a single data with the format YYYYMMDDThhmmss and it can also be used as an interval between two dates separated by two colons. If the parameter samplingTimes is used the observations obtained will correspond to that interval of time. The parameter observedproperties is used to determine what property is the user trying to obtain by providing its id. Finally, the parameter geo is a string formed by the latitude, longitude and radius separated by a semicolon. The latitude and longitude determine a central point and the radius is used to establish how far from that point should the data be gathered.

For example, if the user wanted to obtain all the observations made between the 30th of April of 2014 at 06:08:31 and the 1st of May of 2014 at 06:08:31 and that also come from the project with the id 6 (which should be public or only the owner would be able to retrieve this data) the URL would be /observations/projects/6/samplingTimes/20140430T060831::20140501T060831. The RDF/XML is too big to add to this document so in Figure 5.4 there’s the graph of the output.

![Sensor data output example in graphical view](image)

**Figure 5.4: Sensor data output example in graphical view**

**Sensors**

• URL: /sensors

• Input parameters: country, city, project and geo.

Parameters country and city are used to determine from which country and city, respectively, should the sensors that are being searched located. Parameters geo and project work like their
homonyms in the Observations get function. If no extra parameter is introduced it’s possible to query the sensor by id like this /sensors/2 for example.

In case the user wanted to get a list of the sensors that are located in the city of Barcelona the URL would be /sensors/city/Barcelona. Below there's the abbreviated response to this URL:

```xml
<?xml version="1.0"?>
  <ssn:SensingDevice rdf:about="http://commonsense.pc.ac.upc.edu/sensors/2">
    <rdfs:label>waspmote1</rdfs:label>
    <rdfs:comment>waspmote</rdfs:comment>
  </ssn:SensingDevice>
</RDF>
```

```xml
"<snm:deployedOnPlatform>
  <snm:Platform rdf:about="http://commonsense.pc.ac.upc.edu/platforms/deployments/sensors/2">
    <DUL:hasLocation>
      <geo: SpatialThing>
        <geo:lat>20</geo:lat>
        <geo: long>21</geo:long>
        <geo: alt>100</geo:alt>
      </geo: SpatialThing>
    </DUL:hasLocation>
  </snm:Platform>
</snm:deployedOnPlatform>
```

```xml
</DUL:hasLocation>
</DUL:hasLocation>
```
Projects

- URL: /projects
- Input parameters: none

Agents

- URL: /agents
- Input parameters: none

Properties

- URL: /properties
- Input parameters: none

UnitsOfMeasure

- URL: /unitsofmeasure
- Input parameters: none
Projects, agents, properties and unitsofmeasure can only be queried by id.

## 5.4 Website

The client-server architecture is a strategy widely used in all types of networks, it assigns two roles to each network element: client and server. Obviously, the website corresponds to this architecture. The primary role of the server is the processing of all the clients petitions and the delivery of the web pages using HTTP. The software required for the server is listed in the production environment explained in section 5.5 and how to configure it properly is detailed in appendix C.

The website has been implemented using Django a Python-based web framework that follows the Model-View-Controller (MVC) architectural pattern. It was decided to implement it using Django because it’s a very well-known framework that provided many of the functionality that had to be developed like the administrator panel which allows to manage all the entities in the database. The MVC pattern on which Django is based on is an abstract model that separates the data from an application, the user interface and business logic (inputs, generating queries or data) into three different components: model, view and controller. In addition to dividing the application into three types of components, the MVC pattern also defines its interactions.

- The model represents the data structures, and usually provides functions that help you retrieve, insert or update information in the database. It also contains functions to access other data from other web services or APIs.

- The View is the information that is presented to the user.

- The controller acts as intermediary between the Model, the View and other resources.

All the models that Django uses are loaded from the database so there's a class for every entity previously detailed in the database section. Below there's the Data entity class as an example to illustrate how one of them is implemented.

```python
class Data ( models . Model ) :
    stream = models . ForeignKey ( 'Stream' , db_column= 'idStream' )
```
value = models.CharField(max_length=450L, db_column='Value')
date = models.DateTimeField(db_column='Date')
dtype = models.ForeignKey('DataType', db_column='idDataType')
def __str__(self):
    return self.value + ' ' + self.dtype.Data2String()
def __unicode__(self):
    return u'%s %s' % (self.value, self.dtype.Data2String())

class Meta:
    db_table = 'Data'

The class must inherit from the base class models.Model, the primary key must be specified using the class models.ForeignKey and every attribute specifies the name of the database column it is mapped to for proper synchronization and it's necessary to implement the str and unicode methods for proper string representation. At last, in the class Meta the name of the database table is defined.

The structure of the views implemented in CommonSense can be seen in Figure 5.5. From the home page you can access the part of the website that explains what the project is about, the project's blog and the private part. The private part includes the dashboard from which an overview can be seen in Figure 5.6 and, in case that the user is also an administrator of the system, it will also include the administration dashboard from which an overview can be seen in Figure 5.7.

Using the user dashboard, the user can add projects, subscribe to a project, view the list of all its projects and edit them. The user can also add and edit sensors, view the list of all the sensors that the user has registered, add a stream or edit an existing one. In the dashboard there's also access to a list of all the data gathered by its sensors and a query interface that uses the API which facilitates the results in RDF/XML, formatted in a table or if it's possible, recollected in a graph as it can be seen in Figure 5.8.

The administrator dashboard allows the administrator manage all the entities that control the website. This entities are the same as the user can control but there's no restriction so the admin can change other users information. There are also some special entities like Group, User,
ProjectStatus or DataType that are only available to the administrator. It may be emphasised that most of the "View all ...", "Add ..." and "Edit ..." seen in Figure 5.5 are the same view but they use different models to populate them. In Figures 5.9, 5.10 and 5.11 there's an example of each of them.

The website design has been developed using Bootstrap as the base and this allowed us to focus on the functionality rather than in the looks of the interface. The website is currently available at http://commonsense.pc.ac.upc.edu.

5.5 Environments

In this section the different environments that will contain the project are listed. The development environment main goal is to provide a separate environment were bugs are allowed in order for the programmer to work on them without having to worry if an external user encounters it thus deteriorating the user experience. The production environment is the environment were only the software enhancements that have been intensely tested are uploaded and available to all users. Appendices B and C contain each an easy tutorial on how to set up the abovementioned environments.

5.5.1 Development environment

The development environment that was configured was on a Intel core i7 2.3GHz with 8GB RAM as basic features. The following list enumerates the basic software required for the deployment of the project:

- Ubuntu 12.0.4 LTS as the operating system
- Python 2.7.3, Django 1.6.2 and Bootstrap 3.0 for the website
- NodeJS 0.10.26 and Virtualenvwrapper 1.7.1.2 for the easy deployment of the API
- MySQL 14.4 as the database
- Git 1.7.9.5 for code sharing
5.5.2 Production environment

The production environment that was configured was on a Intel core 2 quad 2.83GHz with 4GB RAM as basic features. The following list enumerates the basic software required for the deployment of the project:

- Ubuntu 12.0.4 LTS as the operating system
- Python 2.7.3, Django 1.6.2, Bootstrap 3.0, Apache 2.2.22 and libapache-mod-python 3.3.1 for the website
- NodeJS 0.10.26 and Virtualenvwrapper 1.7.1.2 for the easy deployment of the API
- MySQL 14.4 as the database
- Git 1.7.9.5 for code sharing
Figure 5.5: Website view structure map
Figure 5.6: User control panel

Figure 5.7: Administrator control panel
CHAPTER 5. IMPLEMENTATION

Figure 5.8: API output recollected in a graph

Figure 5.9: View all posts view
Figure 5.10: Add stream view

Figure 5.11: Edit project view
Chapter 6

Conclusions and future work

In this master thesis the foundations of CommonSense were designed and developed. CommonSense main core is composed by a MySQL database where all the sensor data and metadata is stored, a Node.js API which allows developers to work on data-processing applications and a Django-powered website which informs the users about the project and enables an easy to use web interface to manage projects, sensors meta-data and data.

The system is currently working with several sensors sending data in RDF/XML using the API, another student is working on an Android-powered smartphone application for the assessment of air pollution using also the API for the data-processing and the website is not currently completed due to some time constraints since I had to leave to Japan for an internship at the National Institute of Informatics but it’s almost finished. In summary, CommonSense 1.0 is currently running and allowing users all around the world to share their data with others but it lacks a complete user-friendly website that explains the project.

As future work several expansion projects were proposed, some of them are described here:

**Finish the website**  It’s necessary to finish the website in order to publicize the project and to allow users without any technical knowledge register sensors and easily obtain their data.

**Provide the system with image gathering from camera sensors**  This would require the modification of the model and the extension of the SSN ontology which currently doesn’t include sensor image meta-data.

**Add an SPARQL endpoint**  SPARQL is an RDF query language, that is, a query language for databases,
able to retrieve and manipulate data stored in Resource Description Framework format. With this feature enhancement there would be two ways of obtaining data from the system.

**Rebuild the database using a graph database** It would be very interesting to use a graph database to assess if there's a performance improvement since the API wouldn't need to translate the response from the database to RDF format.

**Expand the API** Support RDF communications in more formats other than XML like N3, Turtle, N-Triples, RDFa or RDF/JSON.
Appendix A

Acronyms

API  Application Programming Interface
CC   Creative Commons
DC   Dublin Core
DOLCE  Descriptive Ontology for Linguistic and Cognitive Engineering
DnS  Descriptions and Situations
DUL  DOLCE + ultra-lite DnS
FOAF  Friend of a friend
GEO  GeoNames Ontology
HTTP  Hypertext Transfer Protocol
IoT   Internet of Things
IT    Information Technology
JSON  Javascript Object Notation
OLD  Open Link Data
OWL  OWL Web Ontology Language
M2M  Machine to Machine

RDF  Resource Description Framework

REST Representational State Transfer

SaaS Software as a Service

SQL Structured Query Language

SSN Semantic Sensor Network

URI Uniform Resource Identifier

URL Uniform Resource Locator

XML eXtensible Markup Language
Appendix B

Production server setup

In the following guide you will find all the necessary steps to configure and deploy the system. The only prerequisite for this setup guide is a fresh installation of Ubuntu 12.04 or 13.10.

B.1 Website and database setup

Create a new user and set the password

$ adduser cs

Add the user to the sudoers file

$ adduser cs sudo

This step may not be always needed but it may be useful if you have some LOCALE problems

$ sudo locale-gen en_GB en_GB.UTF-8
$ sudo dpkg-reconfigure locales

Install the following software using apt-get, beware that mysql server will request a password for the root user.

$ sudo apt-get install git apache2 libapache2-mod-python apache2-
   threaded-dev
$ sudo apt-get install mysql-server python-mysqldb python-pip
$ sudo apt-get install libapache2-mod-wsgi
If the last command doesn't work, then you can follow this website to compile for your current version of Python https://code.google.com/p/modwsgi/wiki/InstallationInstructions.

Do not install python-django from the Ubuntu repositories, it may give you some problems if you are using Ubuntu13.04.

$ sudo pip install Django
$ sudo pip install django-admin-bootstrapped
$ sudo pip install django-filter
$ sudo pip install django-yamlfield
$ sudo pip install django-registration
$ sudo pip install django-blog-zinnia
$ sudo pip install zinnia-theme-bootstrap

Add apache2 module wsgi and restart the server

$ sudo a2enmod wsgi
$ sudo service apache2 restart

Create a folder for the project

$ cd ~
$ mkdir project

Clone the git repository and use as user "csconfine" and as password "compnet2013".

$ git clone https://www.ac.upc.edu/projects/commonsense/git/

Optionally, you can edit the '.netrc' file so you can avoid typing the username and password any time you update the git project.

machine www.ac.upc.edu
login USERNAME_TO_AC_GIT_SERVER
password PASSWORD_TO_AC_GIT_SERVER

Create the wsgi configuration file, make sure you change the values of <SERVER_ADMIN_MAIL> and <URL_TO_SERVER>, and place it in "/etc/apache2/sites-available/commonsense".
ServerAdmin <SERVER_ADMIN_MAIL>
ServerName <URL_TO_SERVER>

DocumentRoot /home/cs/project/commonsense/projects/static

WSGIScriptAlias / /home/cs/project/commonsense/CommonSense/wsgi.py
WSGIPythonPath /home/cs/project/commonsense

Alias /robots.txt /home/cs/project/commonsense/projects/static/robots.txt
Alias /favicon.ico /home/cs/project/commonsense/projects/static/favicon.ico
Alias /static/ /home/cs/project/commonsense/projects/static/

<Directory /home/cs/project/commonsense/projects/static/>
    Order deny, allow
    Allow from all
</Directory>

<Directory /home/cs/project/commonsense/CommonSense>
    <Files wsgi.py>
        Order deny, allow
        Allow from all
    </Files>
</Directory>

Tell Apache to enable the site

$ sudo a2ensite commonsense

Move the Apache server to port 3001, to do that edit the file "/etc/apache2/ports.conf" and change "Listen 80" to "Listen 3001". Restart the server to let the changes take effect.
$ sudo service apache2 restart

Login to the mysql client
$ mysql -u root -p

Create the database for Django commonsense
create database commonsense;

Move to the project directory and sync the database
$ python manage.py syncdb

### B.2 NodeJS setup

Install the following software using apt-get
$ sudo apt-get install python-software-properties python g++ make
$ sudo add-apt-repository ppa:chris-lea/node.js
$ sudo apt-get update
$ sudo apt-get install nodejs

Install the Ubuntu upstart script to autospawn the NodeJS server automatically
$ sudo npm install -g forever --save
$ sudo cp /home/cs/project/commonsense/api/daemon/csapi.conf /etc/init
  → /
$ sudo cp /home/cs/project/commonsense/proxy/csproxy.conf /etc/init/.
$ sudo start csapi
$ sudo start csproxy

In order to be able to run the unit testing tests for the API you will also need to install globally the "mocha" library
$ sudo npm install -g mocha

Install the CommonSense API dependencies
$ npm install
B.3 Backup

A periodic backup of the database using automysqlbackup is also set up.

```
$ sudo apt-get install automysqlbackup
$ sudo automysqlbackup
```

The default configuration will put all the files in "/var/lib/automysqlbackup"
Appendix C

Development server setup

In the following guide you will find all the necessary steps to configure and deploy the system. The only prerequisite for this setup guide is a fresh installation of Ubuntu 12.04 or 13.10.

C.1 Website and database setup

Install the following software using apt-get, beware that mysql server will request a password for the root user.

```bash
sudo apt-get install git mysql-server python-mysqldb libmysqlclient-dev python-pip python-dev
```

Clone the git repository and use as user "csconfine" and as password "compnet2013".

```bash
$ git clone https://www.ac.upc.edu/projects/commonsense/git/
```

Login to the mysql client

```bash
$ mysql -u root -p
```

Create the database for Django commonsense

```bash
create database commonsense;
```

```
sudo apt-get install virtualenvwrapper
```

Open your ~/.bashrc and add the following lines
export WORKON_HOME = $HOME/.virtualenvs

source /usr/local/bin/virtualenvwrapper.sh

Create a python virtual environment (it is created on /virtualenvs folder)

mkvirtualenv csproject --distribute

Run the virtualenv environment (this step is mandatory every time you restart the PC)

workon csproject

Install Django and all the requirements

pip install -r requirements.txt

NOTE: If you update the django project, including new external libraries, you will need to update the requirements.txt file

pip freeze > requirements.txt

Now, everything is installed but you still need to edit some configuration files that are not included in the git repository.

cd $YOURPROJECTDIR/commonsense/Commonsense

cp dev_settings.py.template dev_settings.py

Open the dev_settings.py file and set the variables

    PASSWORD // Your db password, the one you used when installed the DB
    SECRET_KEY // It’s used as a random generation seed, you can write → whatever
    EMAIL_HOST_USER // If you plan to register new users, Django is → configured to use gmail as a mail server
    EMAIL_HOST_PASSWORD // Write the gmail User/password you want to use

Check everything is working

cd $YOURPROJECTDIR

./manage.py runserver

and open a web browser at localhost:8000
C.2 NodeJS setup

Install the following software using apt-get

$ sudo apt-get install python-software-properties python g++ make
$ sudo add-apt-repository ppa:chris-lea/node.js
$ sudo apt-get update
$ sudo apt-get install nodejs

In order to be able to run the unit testing tests for the API you will also need to install globally the "mocha" library

$ sudo npm install -g mocha

Install the CommonSense API dependencies

$ npm install

Check that everything is working

code server.js
Bibliography


