Designing Semantic Web of Things Applications with M3

M3 API Documentation and user interface for developers

Creator	Amelie Gyrard (Eurecom)
Send Feedback	Do not hesitate to ask for help or give us feedback, advices to improve our tools or documentations, fix bugs and make them more user-friendly and convenient: amelie.gyrard@insight-centre.org
Goal	• This documentation guides developers to design Semantic Web of Things applications thanks to the Machine-to-Machine Measurement (M3) framework through APIs that we develop or user interfaces. • Get Semantic Web of Things templates with web services or the user interface • Use the M3 converter to semantically annotate IoT data • Interpret IoT data and get suggestions or high level abstractions • A tutorial is also included to design the application with the M3 and Jena framework. • Use LOV4IoT web services • Use M3 nomenclature and M3 ontology web services
Requirement	Use the Jena Framework.
Created	April 22, 2015
Last updated	June 2016 February 2016
Status	Work in progress
URL	http://www.sensormeasurement.appspot.com/documentation/M3APIDocumentation.pdf

This documentation guides developers to design Semantic Web of Things applications thanks to the Machine-to-Machine Measurement (M3) framework.
It will assist them in:

- **Generating IoT templates**. The developers do not need to design any ontologies, datasets and rules, they are provided in the M3 templates.
- **Semantically annotating IoT data.** The developers not need to semantically annotate his IoT data. It will be automatically done by the M3 converter.
- **Interpreting IoT data**. The developers are assisted by the M3 framework to interpret IoT data. They will get high-level abstractions or even M3 suggestions according to the M3 template chosen.

In this documentation, the developers can either use web service or user interface.

Table of Contents

I. Tutorial: Building the naturopathy application with the user interface SWoT generator and the Jena framework

1. Generating the naturopathy template with the SWoT generator

- Go on this web page:

http://www.sensormeasurement.appspot.com/?p=m3api

- Choose the sensor 'Thermometer' in the drop-down list.
- Choose the domain 'Healthcare' in the drop-down list.

- Choose the template 'Body Temperature, Symptoms and Home Remedies' in the drop-down list. In this case, we suggest only one template.
- Click on the button 'Generate ZIP file.'

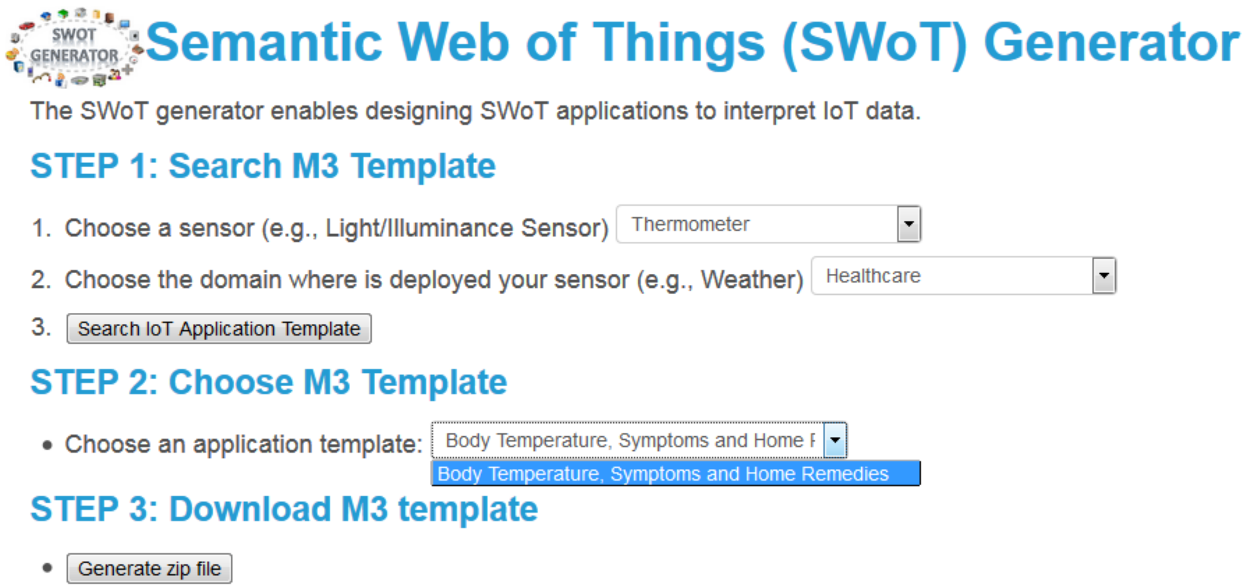


Figure 1. Download the naturopathy template using the SWoT generator

2. Explore the naturopathy template

Open the naturopathy template that you just downloaded. This template is composed of the following files:

- **ruleM3Converter.txt**: a set of rules used to convert sensor data according to our M3 language implemented in the M3 ontology. For instance, we use the term temperature and not term. An essential basis for the reasoning.
- **naturopathy.owl**: the naturopathy ontology
- **naturopathy-dataset.rdf**: the naturopathy dataset
- **m3SparqlGeneric.sparql**: the SPARQL query to get smarter data or even suggestions. For instance, get home remedies when you have the fever.
- **m3.owl**: the M3 ontology essential to describe sensor data in an interoperable manner to ease the reasoning and the interlinking of domains.
- **LinkedOpenRulesHealth.txt**: This file is a dataset of interoperable rules to interpret health measurements. For instance: IF BodyTemperature > 38°C THEN **HighFever**.
- **health.owl**: the health ontology. For instance, **Symptom** is a concept defined in this ontology.
- **health-dataset.rdf**: the health dataset. For instance, **HighFever** is an instance of the **Symptom** concept in this dataset.

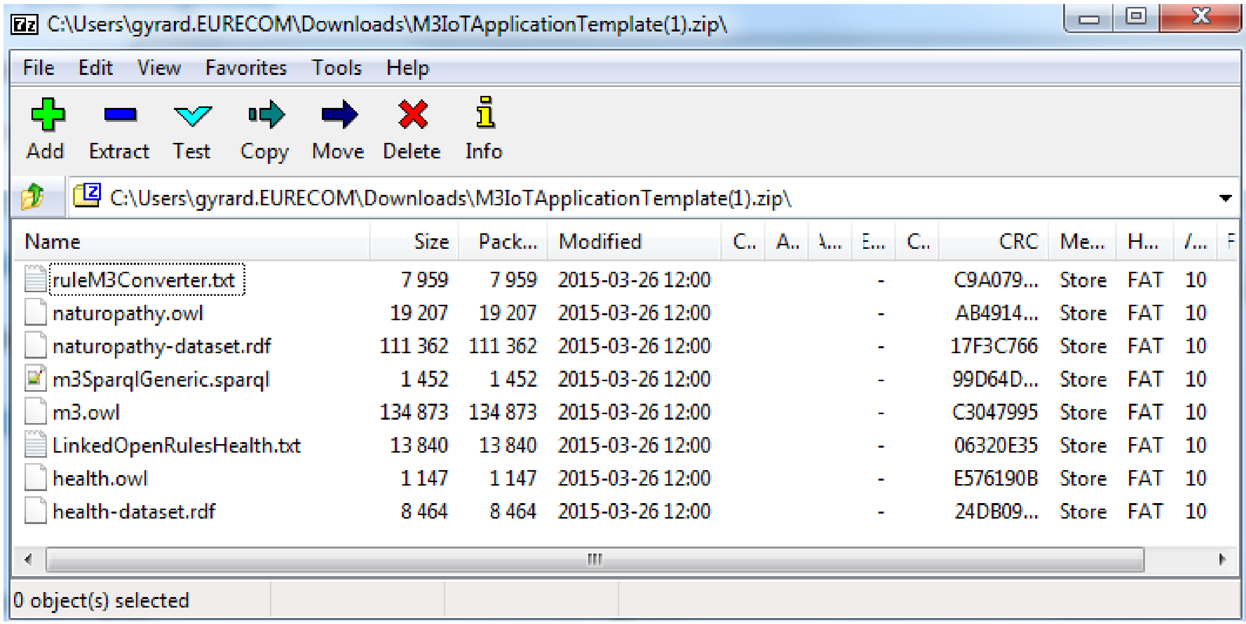


Figure 2. The naturopathy template

3. Get the sensor dataset already converted with M3

- Download the sensor dataset: http://www.sensormeasurement.appspot.com/dataset/sensor_data/senml_m3_health_data.rdf

To begin with, try with the sensor dataset that we have already converted according to the M3 ontology. In the extract below, you have the measurement 'temperature 38°C', a new type has been added 'BodyTemperature' which will be used in the reasoning process to infer high-level abstractions.

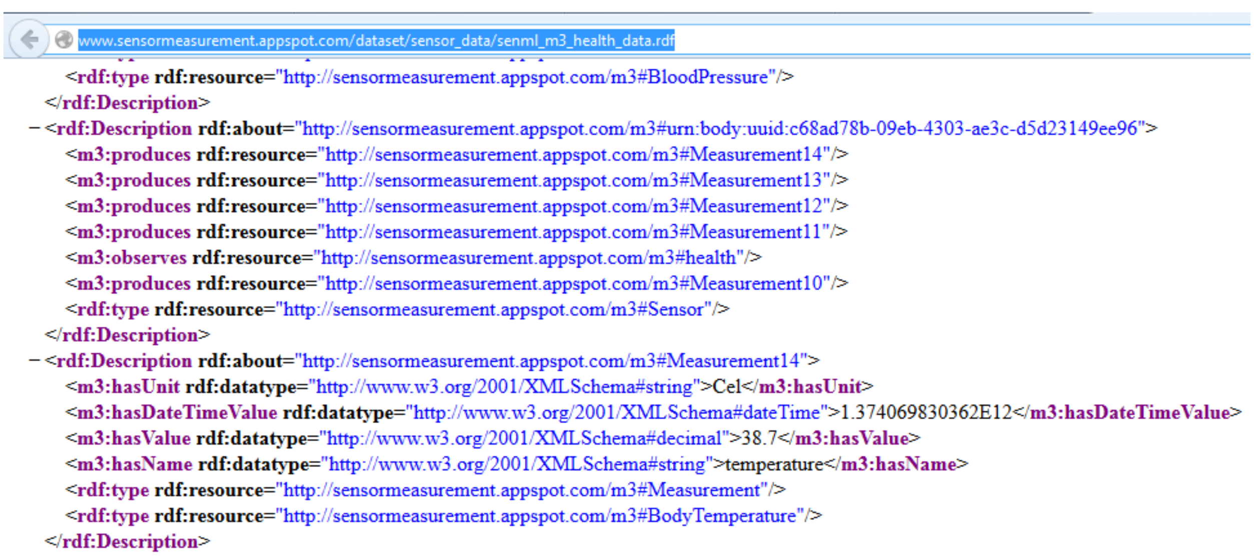


Figure 3. Extract of the sensor dataset

4. Be familiar with the Jena framework

Jena tutorial if you are not familiar with this framework: https://jena.apache.org/

5. Load the sensor dataset in your Java application with the Jena framework

Java code example:

```
public static final String HEALTH_M3_SENSOR_DATA_WAR =
"./dataset/sensor_data/senml_m3_health_data.rdf";

Model model = ModelFactory.createDefaultModel();
ReadFile.enrichJenaModelOntologyDataset(model, HEALTH_M3_SENSOR_DATA_WAR);

//check that the model is not empty, that the sensor data is loaded
Model.write(System.out);

```

Figure 4.Load the Sensor dataset with Jena

ReadFile Java Class:

Java code example:

```
    /**
     * Read ontologies or RDF dataset,
     * included directly from the file (a path) and add it to the jena model
     * @param model
     * @param file
     */
   public static void enrichJenaModelOntologyDataset(Model model, String file){
      try {
          InputStream in = new FileInputStream(file);
          model.read( in, file );//file:"+
          in.close();
      } catch (IOException e) {
          // TODO Auto-generated catch block
          e.printStackTrace();
      }
   }
```

Figure 5.Load a file (ontology or RDF dataset) in the Jena model

6. Load the ontologies and datasets in your Java application with the Jena framework

// load **m3.owl**

ReadFile.enrichJenaModelOntologyDataset(model, ROOT_OWL_WAR + "m3");

// load **naturopathy.owl**
ReadFile.enrichJenaModelOntologyDataset(model, NATUROPATHY_ONTOLOGY_PATH);

// load **naturopathy-dataset.rdf**
ReadFile.enrichJenaModelOntologyDataset(model, NATUROPATHY_DATASET_PATH);

// load **health.owl**
ReadFile.enrichJenaModelOntologyDataset(model, HEALTH_ONTOLOGY_PATH);

// load **health-dataset.rdf**
ReadFile.enrichJenaModelOntologyDataset(model, HEALTH_DATASET_PATH);

1. Load the rules and execute the Jena reasoner

// load **LinkedOpenRulesHealth.txt**

```
//reasoner for jena rules
// the reasoner will infer new triples
// and high level abstraction from sensor data

// read rules
Reasoner reasoner = new GenericRuleReasoner(Rule.rulesFromURL(PATH + LinkedOpenRulesHealth.txt));

//for android use Rule.parseRule
reasoner.setDerivationLogging(true);

//apply the reasoner
InfModel inf = ModelFactory.createInfModel(reasoner, model);
return inf;
}
```

Figure 6.Load rules and execute the Jena reasoner

2. Modify the SPARQL query

Java code example to modify the SPARQL query with variables:

```
//variable in the sparql query
ArrayList<VariableSparql> var = new ArrayList<VariableSparql>();
var.add(new VariableSparql("inferTypeUri", Var.NS_M3 + "BodyTemperature", false));
// we look for BodyTemperature Measurements
```

Figure 7.Modify variables in the SPARQL query

In this example, we are looking for BodyTemperature measurements in the dataset.

VariableSparql Java Class:

```java
/**
* To change the values of some variables in sparql queries
*/
public class VariableSparql {

private String variableName;
private String value;
private boolean isLiterral;
public String getVariableName() { return variableName; }
public VariableSparql(String variableName, String value, boolean isLiterral) {
   super();
   this.variableName = variableName;
   this.value = value;
   this.isLiterral = isLiterral;
}
public void setVariableName(String variableName) { this.variableName = variableName; }
public String getValue() { return value; }
public void setValue(String value) { this.value = value; }
public boolean isLiterral() {   return isLiterral; }
public void setLiterral(boolean isLiterral) {  this.isLiterral = isLiterral; }
}
```

Figure 8. The VariableSparql Java Class example

3. Execute the SPARQL query with Jena

// **load m3SparqlGeneric.sparql**

```java
ExecuteSparqlGeneric reqSenml = new ExecuteSparqlGeneric(inf, sparqlQuery);
String resultSparqlsenml = reqSenml.getSelectResultAsXML(var);
// you should get high level abstractions in XML.

```

Figure 9. Execute the SPARQL query example

ExecuteSparqlGeneric Java class

```java
 public String getSelectResultAsXML(ArrayList<VariableSparql> var){
      QueryExecution qe = replaceVariableInRequest(this.model, this.query, var);
      //get result from sparql request
      ResultSet results = qe.execSelect() ;
      String res = "No results";
      res = ResultSetFormatter.asXMLString(results);

      qe.close();
      return res;
   }

```

Figure 10. Get the result of the SPARQL query, more precisely the high level abstractions

ExecuteSparqlGeneric Java class

```
public Model model;
public Query query;

public ExecuteSparqlGeneric(Model model, String sparqlRequest) {
    super();
    this.model = model;

    //load the sparql query
    this.query = QueryFactory.create(ReadFile.readContentFile(sparqlRequest));
}

public static QueryExecution replaceVariableInRequest(Model model, Query query, ArrayList<VariableSparql> var){
    QueryExecution qe = null;
    RDFNode node = null;
    QuerySolutionMap initialBinding = new QuerySolutionMap();
    //replace sparql request by variables
    if(var!=null){
        for (VariableSparql variableSparql : var) {
            if (variableSparql.isLiteral()){
                node = model.createLiteral(variableSparql.getValue());
            }
            else{
                node = model.getResource(variableSparql.getValue());
                //System.out.println("node: " + node);
            }
            initialBinding.add(variableSparql.getVariableName(), node);
        }
        qe = QueryExecutionFactory.create(query, model, initialBinding);
    }
    else{
        qe = QueryExecutionFactory.create(query, model);
    }
    return qe;
 }
```

Figure 11. ExecuteSparqlGeneric Java class example

4. Check that the naturopathy application works

You should have the results in xml, if it not empty it works!
Congratulations!

You can then design your own applications, and display the result in a user interface.

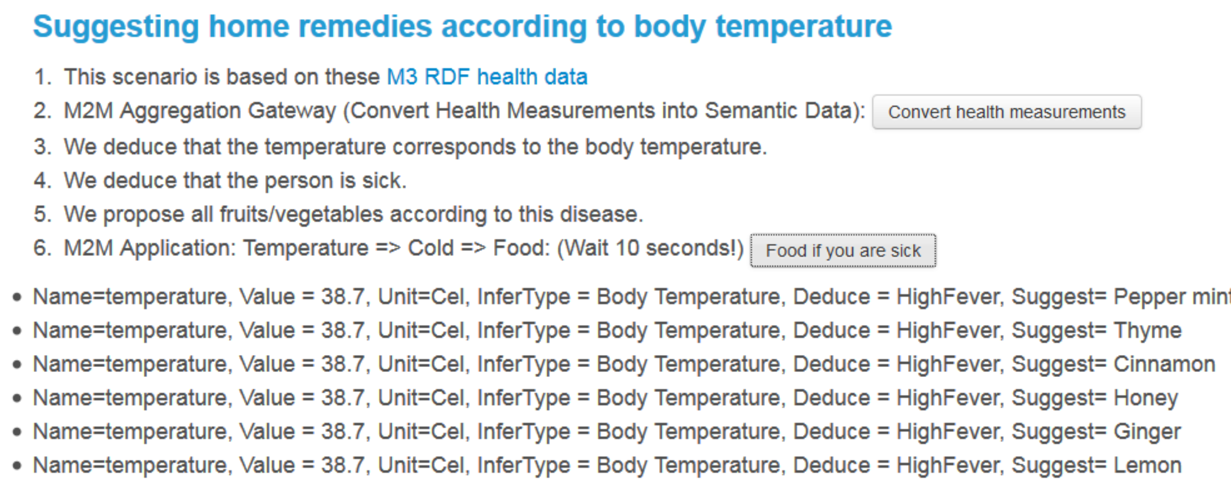


Figure 12. Suggestions provided by the SPARQL query from the template

II. Generating IoT templates with M3 user interface or web services

1. M3 User interface

You can use the user interface: http://www.sensormeasurement.appspot.com/?p=m3api

See user guide: www.sensormeasurement.appspot.com/documentation/UserGuide.pdf

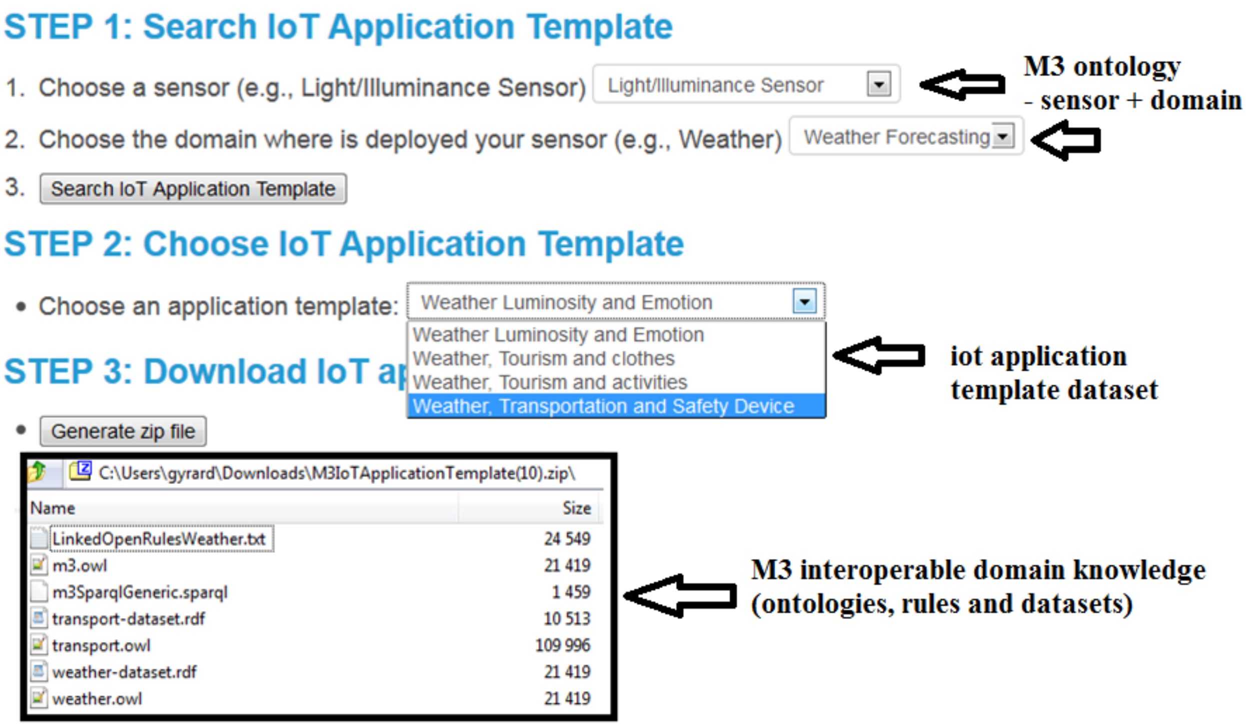


Figure 13. Generating M3 templates using M3 user interface

Be careful, the SPARQL query generated does not have SPARQL variables replaced.
Due to technical issues with Google Web Toolkit (cannot write in a file), please use the M3 web service to generate the SPARQL query with variables replaced.
If you are familiar with SPARQL, you can replace variables yourself.

2. M3 Web Service: looking for IoT application template

Web service URL:
http://www.sensormeasurement.appspot.com/m3/searchTemplate/?sensorName=LightSensor&domain=Weather&format=json

Description: You are looking for IoT application templates with the following parameters:

- sensorName=LightSensor
  The parameter **sensorName** is the name of the sensor.

If you want to indicate another **sensorName** , see:
http://www.sensormeasurement.appspot.com/documentation/NomenclatureSensorData.pdf
domain=Weather
The parameter **domain** is where is deployed your sensor.
If you want to indicate another domain, see:
http://www.sensormeasurement.appspot.com/documentation/NomenclatureSensorData.pdf
format= json
The parameter **format** can be json or xml

Results:

www.sensormeasurement.appspot.com/m3/searchTemplate/?sensorName=LightSensor&domain=Weather&format=json

ioTappli parameter


```
{
  head: {
    vars: [
      "m2mappli",
      "m2mdevice",
      "m2mapplilabel",
      "m2mapplicomment"
    ]
  },
  results: {
    bindings: [
      {
        m2mappli: {
          type: "uri",
          value: http://sensormeasurement.appspot.com/m3#WeatherTransportationSafetyDeviceLight
        },
        m2mdevice: {
          type: "uri",
          value: http://sensormeasurement.appspot.com/m3#LightSensor
        },
        m2mapplilabel: {
          type: "literal",
          "xml:lang": "en",
          value: "Luminosity, Transportation and Safety Device"
        },
        m2mapplicomment: {
          type: "literal",
          "xml:lang": "en",
          value: "IoT application to suggest safety devices according to the luminosity (e.g., sunny -> sun visor)"
        }
      }
```


Figure 14. Looking for the M3 templates

3. M3 Web Service: generating IoT application template

Web service URL:
http://sensormeasurement.appspot.com/m3/generateTemplate/?iotAppli=WeatherTransportationSafetyDeviceLight

Description: To generate the domain knowledge needed to build the IoT application template:

- ioTappli=WeatherTransportationSafetyDeviceLight

The parameter **ioTappli** is the end of the m2mappli URI that you can find in the result provided by the previous web service

(http://www.sensormeasurement.appspot.com/m3/searchTemplate/?sensorName=LightSensor&domain=Weather&format=json)

Results:

```
http://sensormeasurement.appspot.com/ont/m3/transport#@http://sensormeasurement.appspot.com/RULES/LinkedOpenRulesWeather.txt@http://sensormeasurement.appspot.com/SPARQL/m3SparqlGeneric.sparql@http://sensormeasurement.appspot.com/dataset/transport-dataset/@http://sensormeasurement.appspot.com/dataset/weather-dataset/@http://sensormeasurement.appspot.com/ont/m3/weather#@http://sensormeasurement.appspot.com/m3#@
```

All URI files generated as separated by @.

URI finishing with # are ontologies

URI finishing with / are datasets

URI finishing with .txt are rules

URI finishing with .sparql are SPARQL queries to query data (to ignore because of google app engine wa cannot automatically generate/write a new file)

To get the SPARQL query ask the web service: http://sensormeasurement.appspot.com/m3/getSparqlQuery/?iotAppli=WeatherTransportationSafetyDeviceLight (see next section)

4. M3 Web Service: generating the SPARQL query with variables replaced

http://sensormeasurement.appspot.com/m3/getSparqlQuery/?iotAppli=WeatherTransportationSafetyDeviceLight

Generate the generic sparql query with variables replaced

Results:

sensormeasurement.appspot.com/m3/getSparqlQuery/?iotAppli=WeatherTransportationSafetyDeviceLight

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX m3: <http://sensormeasurement.appspot.com/m3#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT DISTINCT ?name ?value ?unit ?inferType ?deduce ?suggest ?suggest_comment WHERE{
        ?measurement m3:hasName ?name.
        ?measurement m3:hasValue ?value.
        ?measurement m3:hasDateTimeValue ?time.
        ?measurement m3:hasUnit ?unit.

        ?measurement rdf:type <http://sensormeasurement.appspot.com/m3#WeatherLuminosity>.
        OPTIONAL { <http://sensormeasurement.appspot.com/m3#WeatherLuminosity> rdfs:label ?inferType. FILTER(LANGMATCHES(LANG(?inferType), "en"))}

        OPTIONAL {
                ?measurement rdf:type ?deduceUri .
                ?deduceUri rdfs:label ?deduce.
                FILTER(LANGMATCHES(LANG(?deduce), "en"))
                FILTER(str(?deduceUri) != str(m3:Measurement) )
                FILTER(str(?deduceUri) != str(<http://sensormeasurement.appspot.com/m3#WeatherLuminosity>) )

                OPTIONAL{
                        {?deduceUri  m3:hasRecommendation ?resUri.} # used by home temp and noise scenario
                        UNION{?resUri  m3:hasRecommendation ?deduceUri . ?resUri rdf:type <http://sensormeasurement.appspot.com/ont/m3/transport#SafetyDevice>. }
scenario
                        ?resUri rdfs:label ?suggest.
                        FILTER(LANGMATCHES(LANG(?suggest), "en"))
                        OPTIONAL{
                                ?resUri dc:description ?suggest_comment.
                                FILTER(LANGMATCHES(LANG(?suggest_comment), "en"))
                        }


                }
        }
}
```

Figure 15. Generating the M3 SPARQL query

5. Code example

```
String URL_M3_API = "http://www.sensormeasurement.appspot.com/m3/";

// STEP 1: Searching the M3 template fitting your needs
String m3_sensor = "LightSensor";
// paramameter sensorName according to the M3 nomenclature
String m3_domain = "Weather";
// paramameter domain according to the M3 nomenclature
String format = "xml"; // or json
String search_M3_template = queryWebService(URL_M3_API + "searchTemplate/?" +
                            "sensorName=" + m3_sensor +
                            "&domain=" + m3_domain +
                            "&format="+ format);

// STEP 2: Choosing the M3 template
String m3_iotAppli = parse(search_M3_template);
// e.g.: = "WeatherTransportationSafetyDeviceLight";

// STEP 3: Generating the M3 template
String m3_template = queryWebService(URL_M3_API + "generateTemplate/?" +
                                    "iotAppli="+ m3_iotAppli);
// paramameter m3_iotAppli found in STEP 2

// STEP 4: Getting M3 ontologies, datasets and rules
String[] url_file = parse(m3_template);
for each url_file
    String[] url_M3_ontology = download(url_file);
    String[] url_M3_dataset = download(url_file);
    String[] url_M3_rule = download(url_file);

// STEP 5: Getting the SPARQL Query (with variables replaced)
String m3_sparql = queryWebService(URL_M3_API + "getSparqlQuery/? +
                                  "iotAppli="+ m3_iotAppli);
```

Figure 16. Generating M3 templates using M3 web services

III. Semantically annotating IoT data with the M3 converter

1. M3 converter user interface

The developer can use the M3 converter user interface:

http://www.sensormeasurement.appspot.com/?p=senml_converter

See user guide: www.sensormeasurement.appspot.com/documentation/UserGuide.pdf

Use Chrome to get the data in a text format, with Firefox you only have the JavaScript alert popup.

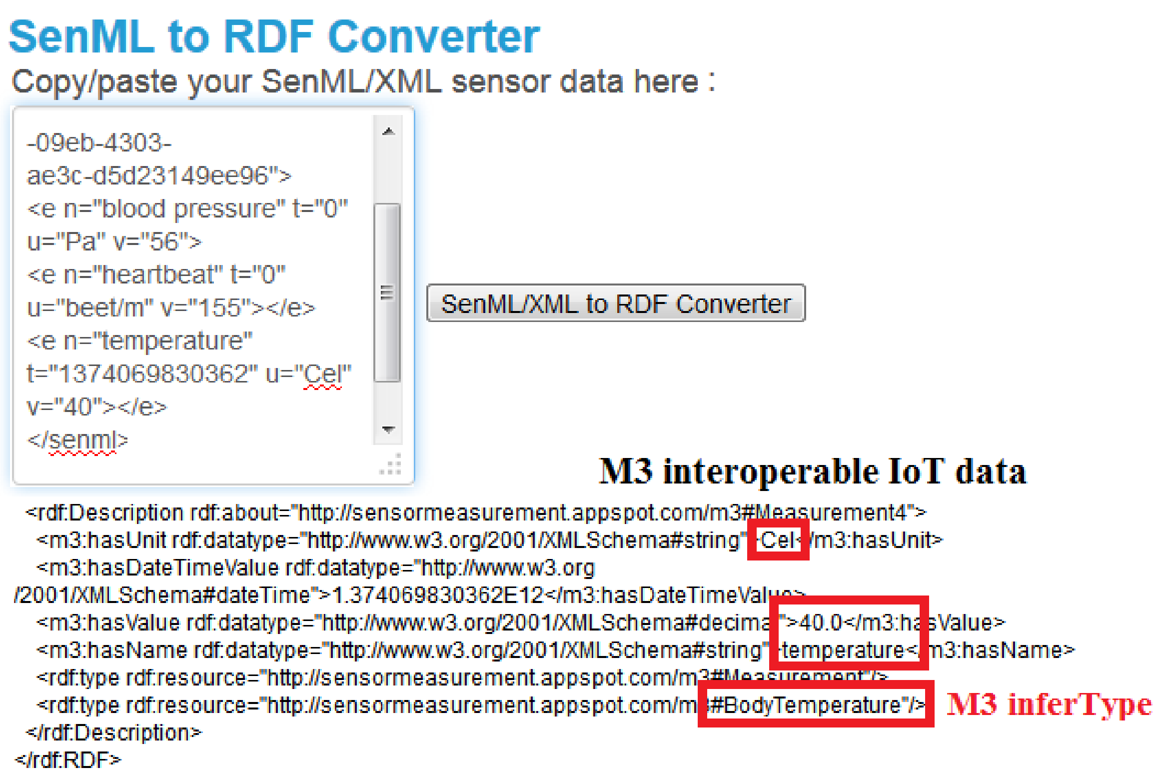


Figure 17. Semantically annotating IoT data with the M3 converter user interface

2. Code example to semantically annotate IoT data with M3

```
// Converting your IoT data using SenML to RDF converter

// String URL_M3_CONVERTER = "http://www.sensormeasurement.appspot.com/swot/";
String format = "xml"; // or json
String iot_data = getSenMLData();

String m3_data = queryWebService(URL_M3_CONVERTER + "convert_senml_to_rdf/?data=" + iot_data +
                              "&format="+ format);

store(m3_data);
```

Figure 18. Semantically annotating IoT data with the M3 converter web service

3. Enrich the M3 converter and adapt it to your data

When you download a template with the SWoT generator[1] you also get the rules to semantically annotate data, the file is called 'ruleM3Converter.txt'.

We did not have time to implement all the M3 nomenclature. Further, we frequently update the M3 nomenclature[2].

But you can still improve and add more rules to semantically annotate your sensor data.

[1] http://www.sensormeasurement.appspot.com/?p=m3api
[2] http://www.sensormeasurement.appspot.com/documentation/NomenclatureSensorData.pdf

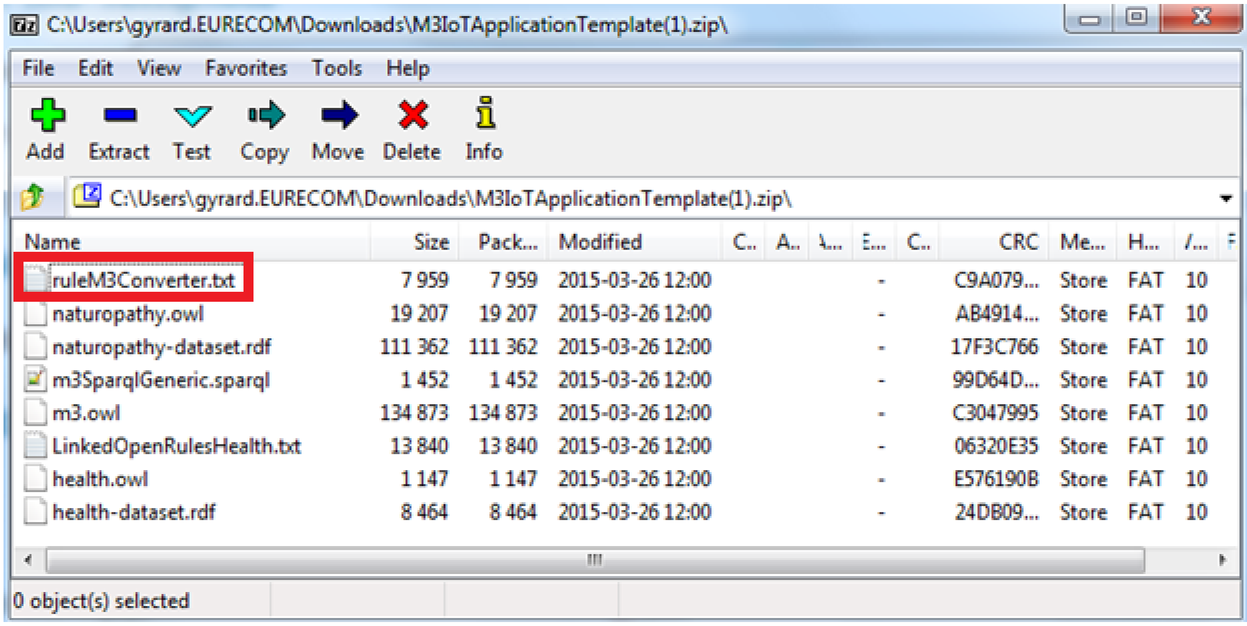


Figure 19. Rules provided in the template to semantically annotate sensor data

The following rule means that we explicitly add the context:
If you get a temperature from health domain (subclassOf m3:FeatureOfInterest), we will explicitly add that it corresponds to a body temperature.

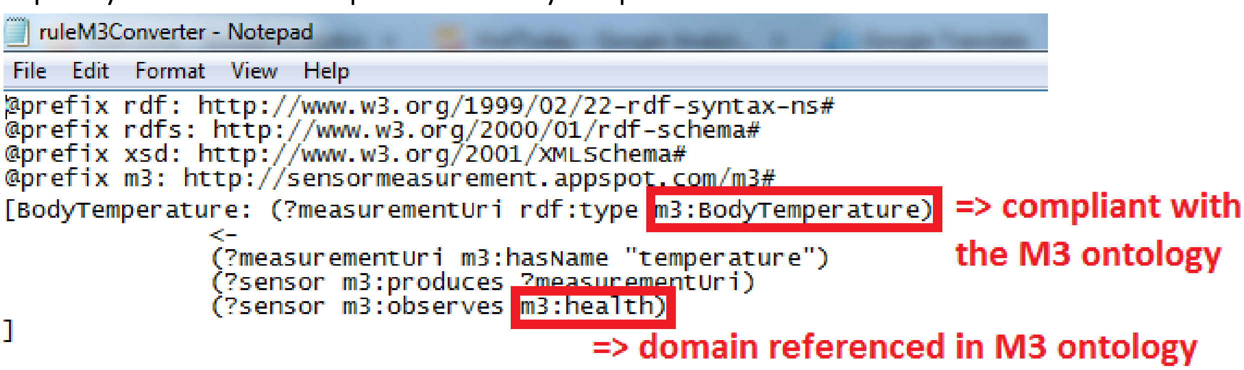
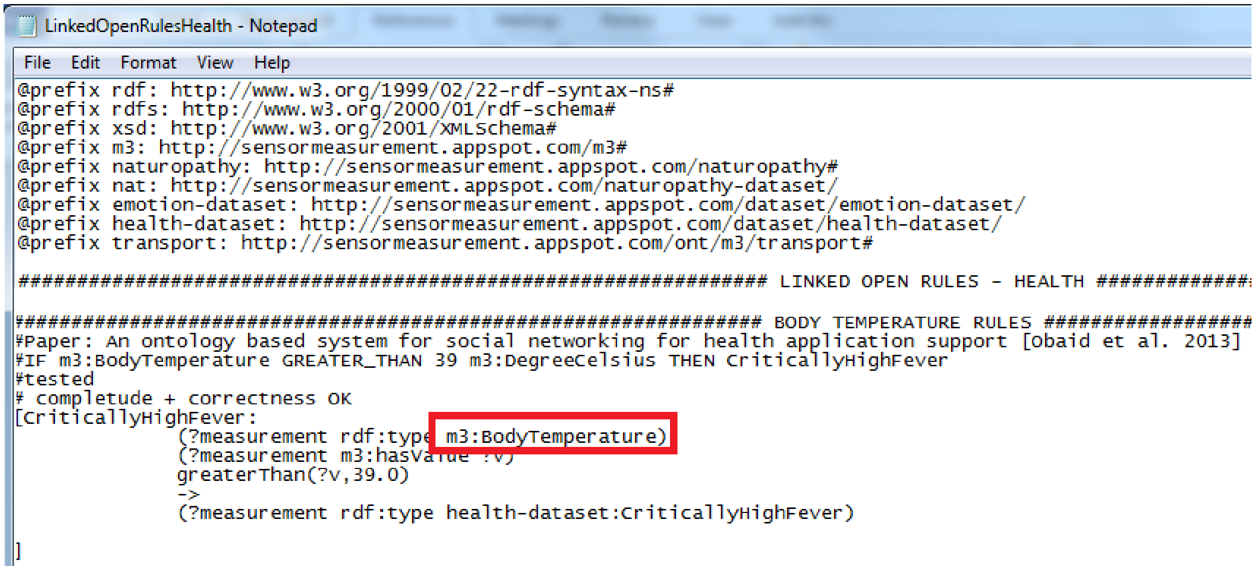


Figure 20. Add new rules to semantically annotate sensor data according to the M3 ontology.

This is important because after, you have the rules adapted to this kind of measurement.



```
LinkedOpenRulesHealth - Notepad
File  Edit  Format  View  Help
@prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
@prefix rdfs: http://www.w3.org/2000/01/rdf-schema#
@prefix xsd: http://www.w3.org/2001/XMLSchema#
@prefix m3: http://sensormeasurement.appspot.com/m3#
@prefix naturopathy: http://sensormeasurement.appspot.com/naturopathy#
@prefix nat: http://sensormeasurement.appspot.com/naturopathy-dataset/
@prefix emotion-dataset: http://sensormeasurement.appspot.com/dataset/emotion-dataset/
@prefix health-dataset: http://sensormeasurement.appspot.com/dataset/health-dataset/
@prefix transport: http://sensormeasurement.appspot.com/ont/m3/transport#

################################################################ LINKED OPEN RULES - HEALTH ############

################################################################ BODY TEMPERATURE RULES ##################
#Paper: An ontology based system for social networking for health application support [obaid et al. 2013]
#IF m3:BodyTemperature GREATER_THAN 39 m3:DegreeCelsius THEN CriticallyHighFever
#tested
# completude + correctness OK
[CriticallyHighFever:
              (?measurement rdf:type m3:BodyTemperature)
              (?measurement m3:hasValue ?v)
              greaterThan(?v,39.0)
              ->
              (?measurement rdf:type health-dataset:CriticallyHighFever)

]
```

Figure 21. Explicit M3 measurement type is reused in the reasoning process

IV. Interpreting IoT data and getting M3 suggestions

Several steps need to be achieved to interpret IoT data (see Figure 22):

- Loading M3 ontologies, datasets which have been generated in the M3 template.
- Loading M3 data.= which has been generated by the M3 converter.
- Interpreting IoT data using the Jena reasoned
- Executing the M3 SPARQL query which has been generated in the M3 template
- Parse the result and build the user interface , control actuators or send notification, etc.

1. Loading M3 domain knowledge

Jena tutorial:

http://jena.apache.org/tutorials/rdf_api.html

Code example:

```
// STEP 1: Loading M3 domain knowledge and m3_data
Model model = ModelFactory.createDefaultModel();
InputStream in = new FileInputStream(PATH_FILE + m3_data);
// m3_data has been generated with the M3 converter
model.read( in, fileURL );//read all ontologies generated in the M3 template (.owl)
model.read( in, fileURL );//read all datasets generated in the M3 template (.rdf)
```

```
in.close();
```

2. Executing rules

Jena tutorial:
http://jena.apache.org/documentation/inference/

Code example:

```
// STEP 2: Interpreting M3 data
Reasoner reasoner = new GenericRuleReasoner(Rule.rulesFromURL(PATH_FILE +
LinkedOpenRules*.txt));
// LinkedOpenRules*.txt: rules generated in the M3 template
reasoner.setDerivationLogging(true);
InfModel infModel = ModelFactory.createInfModel(reasoner, model); //apply the reasoner
// infModel has been updated with high-level abstraction
```

3. Executing SPARQL query

Jena tutorial:
http://jena.apache.org/tutorials/rdf_api.html

Code example:

```
// STEP 3: Getting M3 suggestions
//	Executing the SPARQL query:
Query query = QueryFactory(m3_sparql); // m3_sparql has been generated in the M3 template
ResultSet results = QueryExecutionFactory.create(m3_sparql, model)
String m3_suggestions = ResultSetFormatter.asXMLString(results)
```

4. Finishing the application

The main task of the develop is to design a user-friendly interface or control actuators, etc. according to the high-level abstractions deduce by M3 or the M3 suggestions provided by M3.

Code example:

```
 // STEP 4: Parsing and displaying m3_suggestions to build the IoT application
// or control actuators, alerting, etc.
```

5. Code summary

```
// STEP 1: Loading M3 domain knowledge and m3_data
Model model = ModelFactory.createDefaultModel();
InputStream in = new FileInputStream(PATH_FILE + m3_data);
// m3_data has been generated with the M3 converter
model.read( in, fileURL );//read all ontologies generated in the M3 template (.owl)
model.read( in, fileURL );//read all datasets generated in the M3 template (.rdf)
in.close();

// STEP 2: Interpreting M3 data
Reasoner reasoner = new GenericRuleReasoner(Rule.rulesFromURL(PATH_FILE + LinkedOpenRules*.txt));
// LinkedOpenRules*.txt: rules generated in the M3 template
reasoner.setDerivationLogging(true);
InfModel infModel = ModelFactory.createInfModel(reasoner, model); //apply the reasoner
// infModel has been updated with high-level abstraction

// STEP 3: Getting M3 suggestions
//  Executing the SPARQL query:
Query query = QueryFactory(m3_sparql); // m3_sparql has been generated in the M3 template
ResultSet results = QueryExecutionFactory.create(m3_sparql, model)
String m3_suggestions = ResultSetFormatter.asXMLString(results)

// STEP 4: Parsing and displaying m3_suggestions to build the IoT application
// or control actuators, alerting, etc.
```

Figure 22. Code example to interpret IoT data and get M3 suggestions

V. Summary: Developing Semantic Web of Things applications

- ➔ STEP 1: Getting the domain knowledge to interpret sensor data
  - o Use the web service (see M3 Web Service: generating IoT application template)
  - o Generating ontologies (.owl)
  - o Generating datasets (.rdf)
  - o Generating rules (LinkedOpenRules*.txt)
- ➔ STEP 2: Getting the SPARQL query
  - o Use the web service (see M3 Web Service: generating the SPARQL query with variables replaced)
  - o Generating the SPARQL query: m3_sparql (.sparql)
- ➔ STEP 3: Converting your data using SenML to RDF converter
  - o http://www.sensormeasurement.appspot.com/?p=senml_converter
  - o See documentation: www.sensormeasurement.appspot.com/documentation/UserGuide.pdf
  - o Storing the RDF M3 sensor data in a file "m3_data.rdf"
- ➔ STEP 4: Building the cross-domain IoT application:
  - o Tutorial: An Introduction to RDF and the Jena RDF API
  - o Storing RDF sensor data "m3_data.rdf" file in a Jena Model:
    Model model = ModelFactory.createDefaultModel();
    InputStream in = new FileInputStream(PATH_FILE + "m3_data.rdf");
    model.read(in, fileURL);//read all ontologies generated in STEP 1 (.owl)

model.read(in, fileURL);//read all datasets generated in STEP 1 (.rdf)
in.close();

- Tutorial: Reasoners and rule engines: Jena inference support
- Download the files with rules:

Sensor-based Linked Open Rules (S-LOR) or generated in see M3 Web Service: generating IoT application template
Syntax: Jena rules, fileName = "LinkedOpenRules*.txt"

- Reasoning on sensor data:
  Reasoner reasoner = new GenericRuleReasoner(Rule.rulesFromURL(PATH_FILE + LinkedOpenRules*.txt));// read rules
  reasoner.setDerivationLogging(true);
  InfModel infModel = ModelFactory.createInfModel(reasoner, model); //apply the reasoner
  // infModel model updated with sensor data inferred
- Executing the SPARQL query:

Query query = QueryFactory(m3_sparql)
ResultSet results = QueryExecutionFactory.create(m3_sparql, model)
Return ResultSetFormatter.asXMLString(results)

- Parse and display the results to build the IoT application

VI. Query the M3 nomenlature/ontology

1. Web service: querying sensors

Search for all M3 sensors:
http://www.sensormeasurement.appspot.com/m3/subclassOf/?nameClass=Sensor&format=json

Results:

www.sensormeasurement.appspot.com/m3/subclassOf/?nameClass=Sensor&format=json


```
{
  ▾ head: {
      ▾ vars: [
            "subject",
            "object",
            "label",
            "comment",
            "imgUrl"
        ]
    },
  ▾ results: {
      ▾ bindings: [
          ▾ {
              ▾ subject: {
                    type: "uri",
                    value: http://sensormeasurement.appspot.com/m3#WindDirectionSensor
                },
              ▾ object: {
                    type: "uri",
                    value: http://sensormeasurement.appspot.com/m3#Sensor
                },
              ▾ label: {
                    type: "literal",
                    "xml:lang": "en",
                    value: "Wind Direction Sensor"
                },
              ▾ comment: {
                    type: "literal",
                    "xml:lang": "en",
                    value: "WindDirectionSensor, unit in Degree"
                },
              ▾ imgUrl: {
                    type: "uri",
                    value: http://sensormeasurement.appspot.com/images/sensor/windDirection.png
                }
            },
```


2. Web service: querying actuators

Search for all M3 actuators:

http://www.sensormeasurement.appspot.com/m3/subclassOf/?nameClass=Actuator&format=json

3. Web service: querying domains

Search for all M3 domains (=FeatureOfInterest):

http://www.sensormeasurement.appspot.com/m3/subclassOf/?nameClass=FeatureOfInterest&format=json

4. Web service: querying health devices

Search for all M3 health devices:

http://www.sensormeasurement.appspot.com/m3/subclassOf/?nameClass=HealthM2MDevice&format=json

5. Web service: querying transport devices

Search for all M3 transport devices:
http://www.sensormeasurement.appspot.com/m3/subclassOf/?nameClass=TransportM2MDevice&format=json

6. Web service: querying home devices

Search for all M3 home devices:
http://www.sensormeasurement.appspot.com/m3/subclassOf/?nameClass=HomeM2MDevice&format=json

VII. LOV4IoT web services

See updates regarding LOV4IoT documentation:
http://sensormeasurement.appspot.com/documentation/LOV4IoTDocumentation.pdf

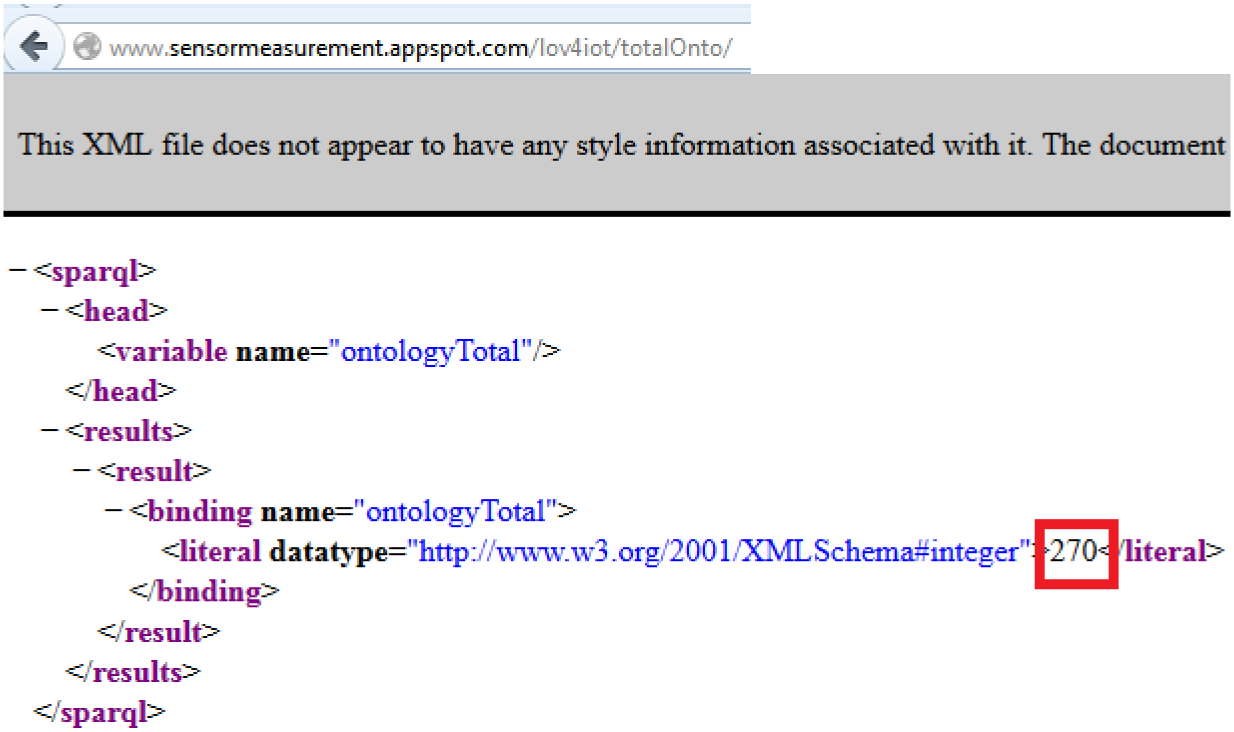
You can download the LOV4IoT RDF dataset[3].

Otherwise, we design some web services:

1. Web service: Get the total number of ontologies

Query:
http://www.sensormeasurement.appspot.com/lov4iot/totalOnto/

[3] http://www.sensormeasurement.appspot.com/dataset/lov4iot-dataset




```
- <sparql>
  - <head>
      <variable name="ontologyTotal"/>
    </head>
  - <results>
    - <result>
      - <binding name="ontologyTotal">
          <literal datatype="http://www.w3.org/2001/XMLSchema#integer">270</literal>
        </binding>
      </result>
    </results>
  </sparql>
```


Figure 23. LOV4IoT Web service to count the total number of ontologies

In the picture, 270 is the total number of ontologies referenced in the LOV4IoT RDF dataset.

2. Web service: Get the number of ontologies by domains

Query:
http://www.sensormeasurement.appspot.com/lov4iot/nbOntoDomain/?domain=BuildingAutomation

For instance domain is: BuildingAutomation, Weather, Emotion, Agriculture, Health, Tourism, Transportation, City, Energy, Environment, TrackingFood, Activity, Fire, TrackingCD, TrackingDVD, SensorNetworks, Security.
The domain is referenced in the M3 nomenclature which is implemented in the M3 ontology (subclassOf FeatureOfInterest).

www.sensormeasurement.appspot.com/lov4iot/nbOntoDomain/?domain=BuildingAutomation

This XML file does not appear to have any style information associated with it. The documen

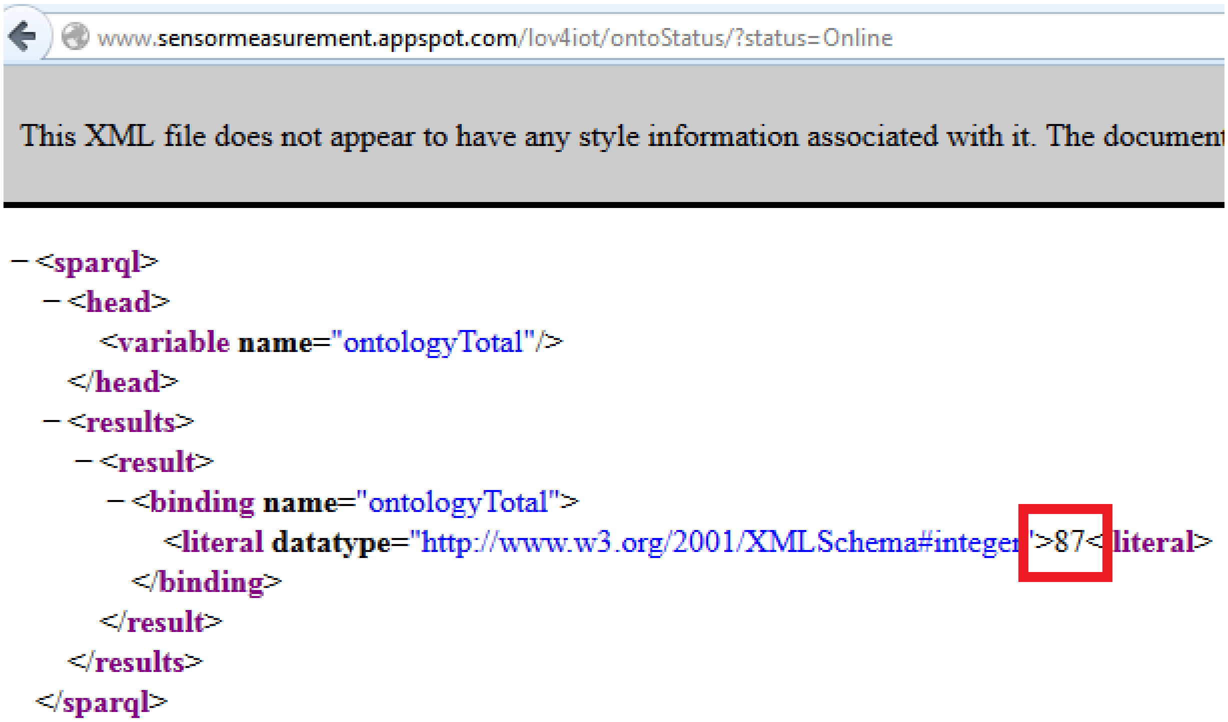

```
- <sparql>
  - <head>
      <variable name="nbOntoDomain"/>
    </head>
  - <results>
    - <result>
      - <binding name="nbOntoDomain">
          <literal datatype="http://www.w3.org/2001/XMLSchema#integer">45</literal>
        </binding>
      </result>
    </results>
  </sparql>
```


Figure 24. LOV4IoT Web service to count the number of ontologies by domain

3. Web service: Get the number of ontology by ontology status

Query:
http://www.sensormeasurement.appspot.com/lov4iot/ontoStatus/?status=Online

For instance, status is: Confidential, OngoingProcessOnline, WaitForAnswer, Online, OnelinLOV, AlreadyLOV.



www.sensormeasurement.appspot.com/lov4iot/ontoStatus/?status=Online

This XML file does not appear to have any style information associated with it. The document


```
- <sparql>
  - <head>
      <variable name="ontologyTotal"/>
    </head>
  - <results>
    - <result>
      - <binding name="ontologyTotal">
          <literal datatype="http://www.w3.org/2001/XMLSchema#integer">87</literal>
        </binding>
      </result>
    </results>
  </sparql>
```


Figure 25. LOV4IoT Web service to count the number of ontologies by ontology status

The web service returns that 87 ontologies referenced in the LOV4IoT RDF dataset are online.

4. Use case

All of these web services have been used in the HTML LOV4IoT web page[4] to automatically count the number of ontologies in the dataset (e.g., by domains, by ontology status, etc.)

[4] http://www.sensormeasurement.appspot.com/?p=ontologies

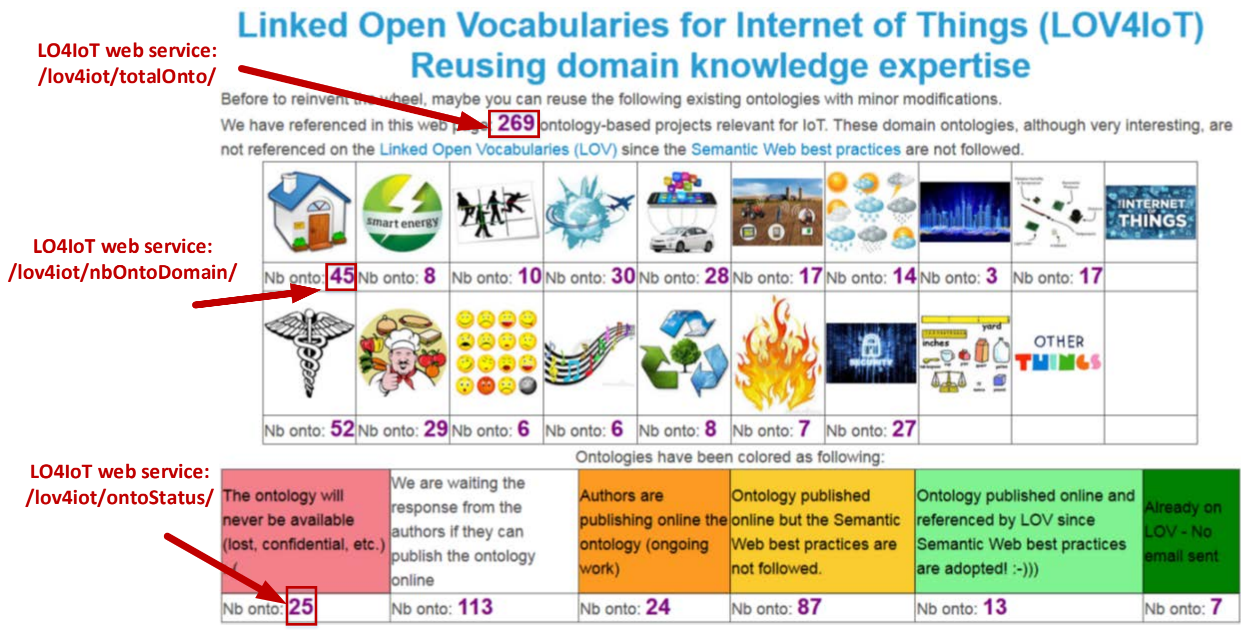


Figure 26. LOV4IoT web services

VIII. Citations

If you use our work, please do not forget to cite us:

- Standardizing generic cross-domain applications in Internet of Things [Gyrard et al. 2014]
- Helping IoT application developers with sensor-based linked open rules [Gyrard et al., ISWC SSN 2014]
- Enrich machine-to-machine data with semantic web technologies for cross-domain applications [Gyrard et al., WF-IOT 2014]