ALMANAC

RELIABLE SMART SECURE
INTERNET OF THINGS FOR SMART CITIES

(FP7 609081)

D5.1.2 Design of the abstraction framework and models 2

Date 2015-08-31 – Version 1.0

Published by the ALMANAC Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme
Objective ICT-2013.1.4: A reliable, smart and secure Internet of Things for Smart Cities

Document control page

Document file: D5.1.2 Design of the abstraction framework and models 2.docx
Document version: 1.0
Document owner: FIT

Work package: WP5 – Abstraction of Smart City Resources
Task: All
Deliverable type: R

Document status: ☒ approved by the document owner for internal review
☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Marco Jahn (FIT)	2015-07-08	Initial draft
0.2	Dario Bonino (ISMB)	2015-07-21	Smart City Resources Adaptation Layer, Smart City Ontologies
0.3	Alexandre Alapetite (ALEX)	2015-08-27	Virtualization Layer
0.4	Jaroslav Pullmann (FIT)	2015-08-22	Metadata Framework
0.5	Marco Jahn (FIT)	2015-08-28	Finalization, Exec Summary, Introduction, Conclusion
1.0	Marco Jahn (FIT)	2015-09-01	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Matts Ahlsen (CNET)	2015-08-31	Approved with comments
Roberto Gavazzi (TIL)	2015-08-31	Approved with comments

Index:

1. Executive summary

The main goal of Work Package 5 is to ensure that all Smart City resources (e.g., sensors or actuators operated by utilities, domain entities such as waste bins, personal devices of socially engaged citizens or services providing sources of public information) are properly virtualized within the ALMANAC model, a single inter-operable abstraction shared and managed in distributed fashion across the whole ALMANAC platform.

This deliverable provides an overview of the results of all tasks of the WP, which are

- to design and develop the Smart City Resources Abstraction Layer (T5.1)
- to design and develop the Framework for Virtualization of Smart City Resources, i.e. the Virtualization Layer (T5.2)
- to develop Ontologies and Semantic Representation and tools to maintain those (T5.3)

The purpose of the **Smart City Resource Abstraction Layer** (SCRAL) is to integrate and expose relevant functionalities of heterogeneous physical devices. It allows integration of Smart City resources enabling the ALMANAC platform to be seamlessly linked and kept synchronized with physical IoT resources. The SCRAL underwent several changes and updates related to the overall architecture revision described in D3.1.2. Moreover, during the second year of the project, support to additional device technologies, data publishing formats and modalities, and semantic annotation of data streams has been included.

The **Virtualization Layer**'s role is to ensure that any aspect related to Smart City resources and their relationship is modelled consistently and can be shared through open, web-oriented protocols. Thus, it provides virtualization on different aspects of the ALMANAC platform, e.g. proxying to internal ALMANAC components, routing requests to the local or to remote ALMANAC instances, wiring ALMANAC internal components to hide the complexity from end-users, and transforming data formats to ease interaction with end-users and third-party services. Updates have been made to the architectural role of the Virtualization Layer making it the entry point for application developers and services to an ALMANAC Platform Instance. Thus, development focussed on proxying of requests/responses and events to/from the different ALMANAC internal components and the discovery of services. Further improvements deal with the implementation of federation features based on LinkSmart GlobalConnect.

The main purpose of the **Semantic Representation Framework** is to maintain and provide an easy programmatic access to rich, graph-based, domain models. Its application scope is to store and make accessible reference models and notations and to offer means to perform query and inference on such data. The design of the ALMANAC Smart City ontologies aims to build on existing models such as IoT branch of the DogOnt ontology, providing device-modelling concepts. A lightweight ontology for modelling smart city resources related to waste management has been developed. Compared to the first version of this deliverable, the Metadata Framework has been implemented. It maintains the Smart City ontologies, and offers means for querying and navigating. The framework acts as a transparent proxy to any triple-store compatible with the SPARQL 1.1 protocol. It considerably augments the typical repository functions and interface coverage, by providing native Java/OSGi integration and remote HTTP APIs.

2. Introduction

2.1 Purpose, context and scope of this deliverable

This deliverable reports the status of the activities performed for the WP5 tasks and the status of the developed components and their interactions.

The main objectives of this work package are:

- to develop adaptation techniques easing integration of Smart City resources and thus enabling the ALMANAC model to be seamlessly linked and kept synchronized with physical IoT resources;
- to develop a virtualization framework ensuring that any aspect related to Smart City resources and their relationship is modelled and shared consistently through open, web-oriented protocols;
- to develop a semantic representation framework hosting an ontology inter-linked with the ALMANAC model and tools to maintain, extend and share assertions stored in such ontology;
- to deliver prototype implementations of abstraction software components in the ALMANAC Platform, namely the Adaptation Layer, the Virtualization Layer, and the Semantic Representation Framework.

This public deliverable includes and refines specifications developed and evolved by the ALMANAC consortium throughout year 2.

Figure 1 shows the placement of the WP5 tasks and components in the ALMANAC conceptual architecture (in red). This deliverable is structured adhering to these concepts:

Chapter 3 describes the Smart City Resource Adaption Layer defining abstraction of heterogeneous technologies and devices. Chapter 4 reports on the Virtualization Layer, mainly the concepts and implementation of the Virtualization Layer Core. Finally, chapter 5 describes the Smart City Ontologies and the design and implementation of the Metadata Framework allowing management of the ontologies.

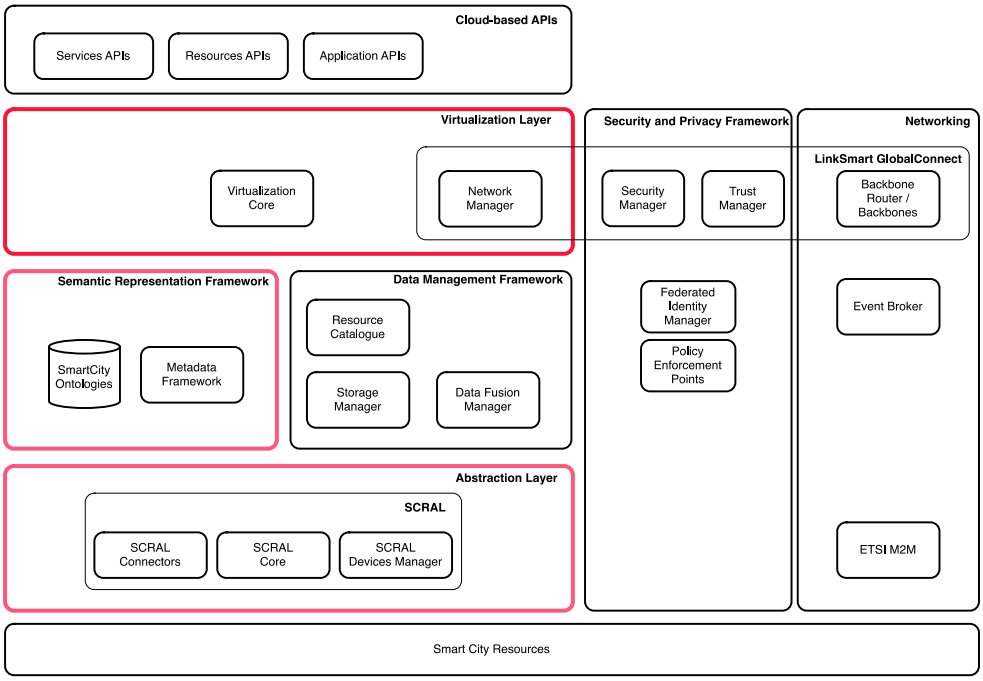


Figure 1: WP5 Layers and Components

2.2 Background

The ALMANAC Smart City Platform (SCP) collects, aggregates, and analyses real-time or near real-time data from appliances, sensors and actuators, smart meters, etc. deployed to implement Smart City processes via an independent, pervasive data communication network.

The main element of the platform is the middleware. This is based on a SOA-based architecture that supports semantic interoperability of heterogeneous resources, devices and services and data management. The various ALMANAC components developed in WP5 tasks are in charge to communicate directly with these physical devices using specific protocols and expose these as smart city resources to the upper layers of the ALMANAC platform architecture.

This deliverable represents an update of D5.1.1 containing the results of WP5 after year 2. The work in this WP is also demonstrated in a series of prototypes, developed in two iterations:

- ID5.2 Adaptation Layer Prototype (M9, M30)
- ID5.3 Virtualization Layer Prototype (M10, M30)
- ID5.4 Ontologies and Semantic Representation Layer Prototype (M15, M30).

3. Smart City Resources Adaptation Layer

The Smart City Resources Adaptation Layer underwent several changes and updates related to the overall architecture revision described in D3.1.2. Moreover, during the second year of the project, support to additional device technologies, data publishing formats and modalities, and semantic annotation of data streams has been included.

To describe the novel solutions adopted, as well as the changes introduced in the SCRAL software stack, a brief summary of the new architectural composition of this layer is reported in the following, together with details on the inner software modules.

3.1 Updated architecture

The Smart City Resource Adaptation Layer (SCRAL) provides a REST-based uniform and transparent access to physical devices, capillary networks, systems and services for monitoring and actuation in a Smart City context. Peculiar device functionalities are uniformed, abstracted and mapped to a well-known set of functions and primitives complying with (device) models handled in the semantic framework of the ALMANAC platform. Moreover, due to its nature of interface between the ALMANAC platform and the real world, the SCRAL offers primitives for applying access-control, data-validation and role-based policies on field-level data sources. SCRAL also interfaces the ETSI M2M Platform linked to the capillary networks deployed in WP4 so it can be seen as the door to data coming from capillary networks and also the interface to the world of ETSI standard for M2M. While typical SCRAL instances are distributed near to physical devices, meaning that more than one SCRAL instance is usually adopted in a single ALMANAC Platform Instance (PI), at least one cloud[1] instance is typically available in a PI to support connection of smart devices, i.e., of devices able to natively exchange data conforming to the ALMANAC data model. In such a case, the main SCRAL duty is to enforce access rights, perform data validation and support needed provisioning primitives.

The SCRAL internal architecture (see Figure 2) encompasses three layers, respectively named API, Core and Field-Access. The topmost layer exploits the SCRAL Connector component, which exposes REST resources to the upper layers of the ALMANAC platform and the MQTT data source, which feeds the ALMANAC PI broker with real-time data purposely uniformed, abstracted and validated by the SCRAL. The latter has been modified in the last months to represent data and observations in compliance with the OGC SensorThings API specification[2]. The core layer hosts core SCRAL components including the SCRAL event-delivery module, the Policy Enforcement Point, the Data Validation module and the Metadata Generation component. Finally, the Field-Access layer integrates components (drivers) to wrap and isolate device-specific and technology-specific implementations used to access real physical devices and systems.

The SCRAL APIs[3] are mainly organized in 4 subsets, respectively named control, streaming, metadata and enforcement endpoints, and can either be based on a standard REST transfer (for Request/Response interactions) or on an MQTT streaming protocol (for real-time data flows). The different communication channels share data formats to better support interoperability and coherence between exchanged data. Following subsections summarize the four different subsets.

Control Endpoint

A RESTful interface exposing all the sensing and actuating features made available by interfaced devices (either directly or through network-level gateways). It is reachable through REST by all components of an ALMANAC PI. While this control endpoint is typically called by the Virtualization Layer, for offering external actuation and querying APIs, it might be leveraged by components belonging to the Data Management Framework for carrying specific tasks, e.g., for populating the Resource Catalogue upon request

The control endpoint does not explicitly support semantics; however, devices representations and functions offered by these APIs are completely aligned to semantic models, through semantic

[1] i.e., deployed on the Internet, and not physically near to any real device/gateway.
[2] http://ogc-iot.github.io/ogc-iot-api/
[3] exposed to components part of an ALMANAC platform instance.

annotations, and with representations defined at the Data Management Framework level, thus enabling the platform to seamlessly integrate real-world data with the corresponding semantics-rich descriptions.

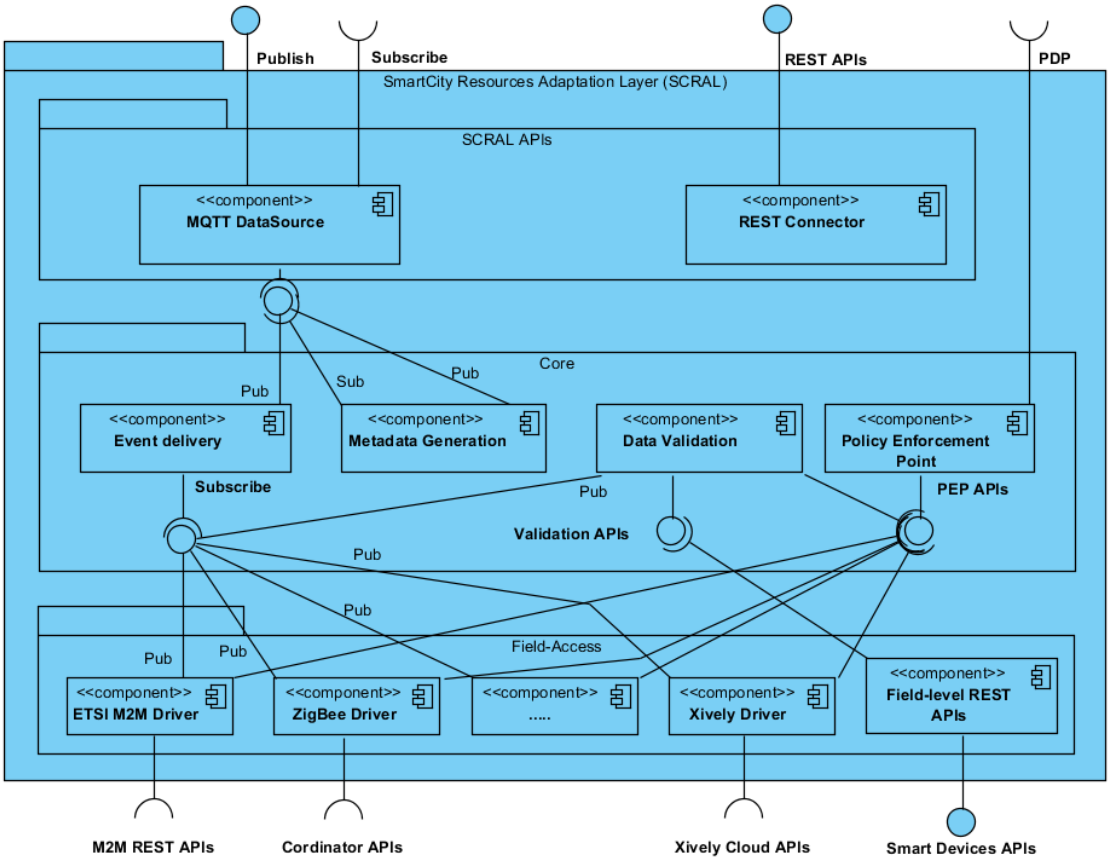


Figure 2: Smart City Resources Adaptation Layer (SCRAL) component diagram.

Streaming Endpoint

The SCRAL offers to the other components of an ALMANAC PI a constantly updated flow of measures, and data, generated in real-time by connected devices and capillary networks. While each interfaced technology may have its own data-generation patterns (e.g., polling vs event-based), and timing, the SCRAL converts such a data sampling into an event-based data-delivery model, where measures, and more in general observations, are conveyed asynchronously over a "trusted", platform-specific MQTT transport. Data flowing along this transport exploits the OGC SensorThings API format, thus enabling better interoperation and easier information exchange with third party applications and/or platforms.

Metadata Endpoint

Device metadata can either be discovered at the field-access level, or can be declaratively injected into the ALMANAC platform through the available provisioning services. Discovery, creation and management of metadata information about devices connected to capillary networks depends on the SCRAL, which exposes such data to the rest of the platform by means of the SCRAL metadata endpoint. The endpoint exploits both a REST-based http interface and an MQTT-based communication channel. Metadata is typically delivered asynchronously, through MQTT as this better fulfils the dynamics of devices joining / leaving sensing and monitoring networks. However, the same data streamed through MQTT can also be gathered through REST APIs, thus allowing other platform components to request snapshots of metadata information regarding devices and services currently connected to the SCRAL. Formats are those defined in the ALMANAC reference framework and mainly stem from the semantic modelling activities carried during the project activities (see

Section 5.4). In other words, exchanged metadata conforms to schemas and ontologies available in the platform through the Semantic Representation Engine.

Enforcement Endpoint

Since the SCRAL lies at the boundaries between the real world and a given ALMANAC platform instance, it is endowed with the authority and capability to enforce decisions on "acceptability" of incoming data streams. According to the federated identity management model adopted in ALMANAC, and based on the XACML Policy Enforcement Pattern and on the SAML framework, data entering the platform must pass a set of security checks and must undergo specific policies depending on the data type, source, etc.

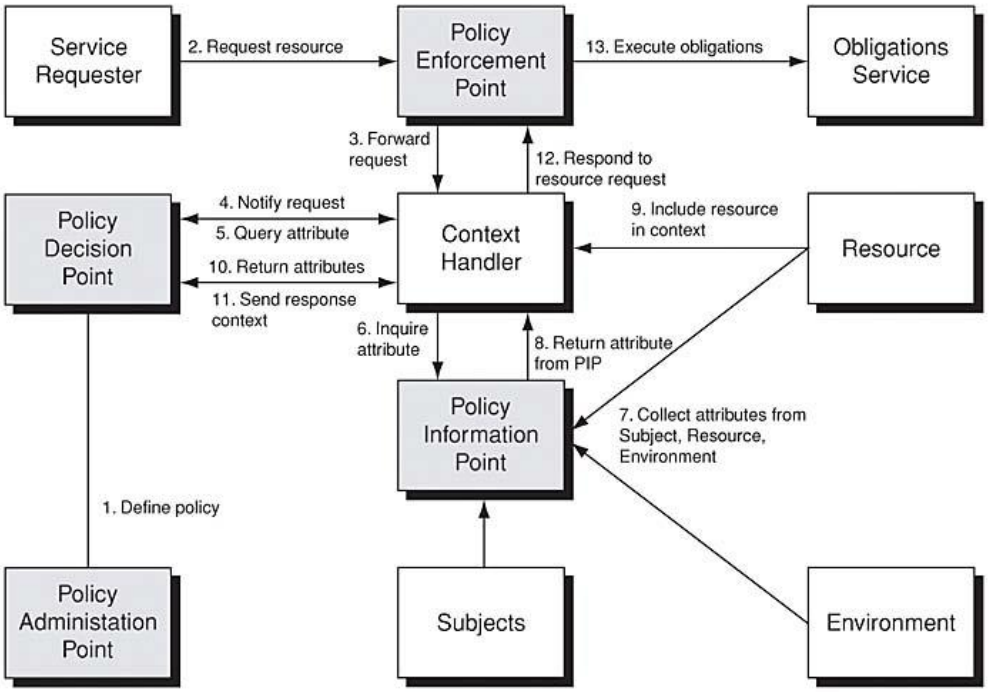


Figure 3: XACML Reference Architecture (Policy Enforcement Pattern)

The SCRAL, as boundary of the platform, implements the Policy Enforcement Point functions allowing or blocking resources from entering into the core ALMANAC platform. Such decisions are taken locally at SCRAL level by exploiting the ALMANAC federated identity manager located at the Virtualization Layer, and which offers Policy Information and Policy decision services. The API used /exposed by the SCRAL to exchange information about currently connected resources and streams, as well as the information needed to authorize or refuse access to the platform services at the field-level, is called Enforcement Endpoint[4].

3.2 Updates

3.2.1 Data Representation

The second year of the project has seen the adoption of the OGC SensorThings API data model as reference for all data exchanges related to sensors. This design choice implied on one hand a substantial re-factoring of the data delivery stack, which needed to be compliant with the new standard, while keeping retro-compatibility, at least for a certain amount of time. On the other hand, it fostered the development of a generic enablement library for handling OGC data, which is reused throughout the platform, and that can be exploited as an "open source" asset of the project[5]. The

[4] This interface is currently under development
[5] This choice is still under evaluation.

library class diagram is reported in Figure 4 whereas the sources, binaries and javadocs are made available through the project repositories and website.

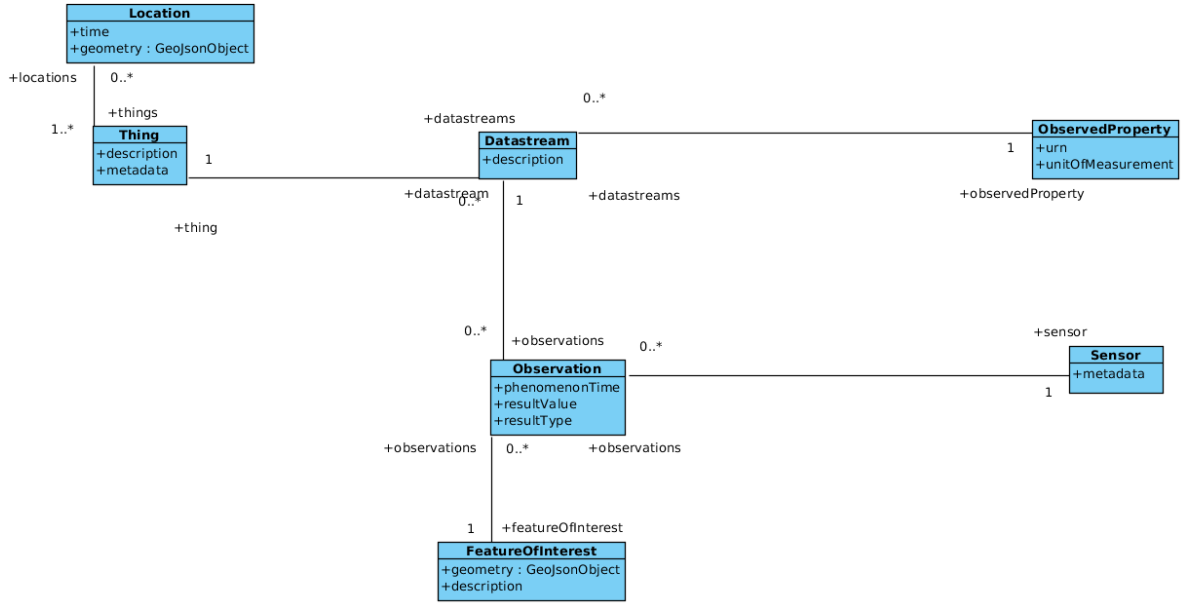


Figure 4. OGC SensorThings API class diagram.

This new set of classes is used as core element to represent ALMANAC data and to generate corresponding data-streams over MQTT.

3.2.2 REST APIs

The adoption of the OGC SensorThings API[6] fuelled the activities on the overall ALMANAC Cloud APIs specifications (D7.3.1) and affected the SCRAL APIs, which undergo several updates and amendments to better comply with such a standard. In particular, the SCRAL REST APIs have been extended to support device and entity provisioning, also in accordance with the updated architectural view defined in D3.1.2. In its last version, the SCRAL supports basic CRUD (Create Update and Delete) operations on resources handled by the almanac platform and represented in terms of OGC Things, Sensors, Datastreams, etc., whereas resource querying and listing is delegated to other platform components, i.e., the Resource Catalogue and the Storage Manager (see Figure 5).

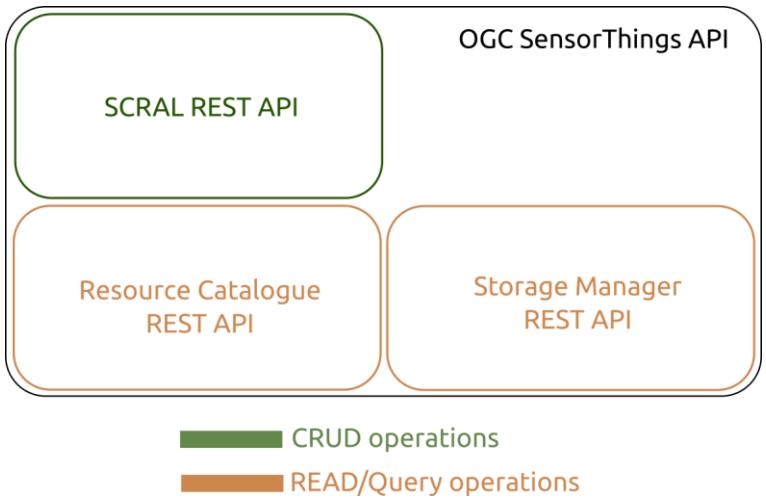


Figure 5. The SCRAL REST APIs and their relations to the OGC SensorThings API and other platform components APIs.

[6] http://ogc-iot.github.io/ogc-iot-api/

To support future scalability and empower approaches based on code-generation, the new SCRAL REST APIs have been defined using the Swagger toolkit[7], and the resulting specification is available in the project repositories. They are fully compliant with the OGC SensorThings API, and support a good subset of the operations defined therein.

The server-side implementation of the SCRAL APIs has been automatically generated from the Swagger specification, and customized on the peculiar needs and requirements of the ALMANAC platform. It features JAX-RS compliant code and exploits the reference JAX-RS server, namely Jersey.

3.2.3 Semantic Annotation of Resources

Resources in the ALMANAC platform are described both as software entities belonging to well-defined data-models and as instances of classes defined in project ontologies, or in linked ones. While the platform modules, which are indeed software, naturally handle software entities, the semantic-level information needs a suitable annotation methodology to grant the proper matching between software and ontologies. The outcomes of such annotation process must be preserved across data transfer between modules, thus supporting semantics-aware computations in the ALMANAC platform.

This challenge is addressed in two phases: when data first enters the platform, i.e., when information is gathered by the SCRAL, suitable metadata is attached to the corresponding software representations, by means of custom-developed Java annotations. Then, when data is exchanged, the corresponding semantic information is preserved by exploiting suitable fields defined in the OGC SensorThings API, i.e., the metadata and description attributes of OGC objects. In such a way, wherever data is delivered, either inside or outside of the platform, the corresponding metadata is available and easy to access.

The SCRAL-level annotation works as follows. At programming time, developers are required to annotate device-representation classes (Java) with a custom annotation as defined in the ALMANAC reference ontology (see Figure 6).

```
@SemanticModel(name="class",value="http://almanac-project.eu/ontologies/smartcity.owl#FillLevelSensor")
```

Figure 6. Custom Java annotation, for metadata association at the SCRAL level.

This permits to establish a one to one mapping between a given Java class, or interface, representing a device, and the corresponding class in the ALMANAC reference ontology (Figure 7).

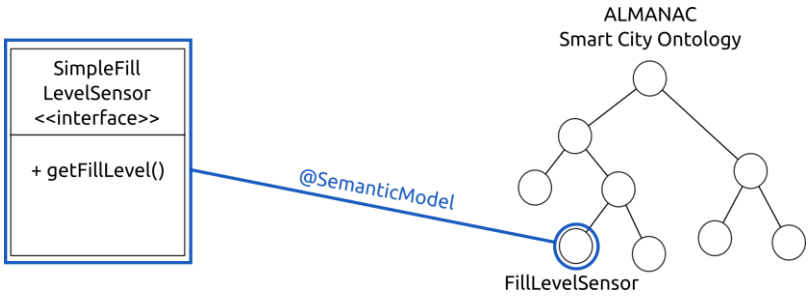


Figure 7. Semantic annotation at the SCRAL level.

At runtime, such annotations are harvested, by means of the Java Reflection framework, and used to populate the metadata and description fields of the OGC-compliant payload of events generated by the SCRAL (see Figure 8 for an excerpt of the Java code handling metadata harvesting).

[7] http://swagger.io/

```
/**
 * Provides the corresponding ontology class representing the given device,
 * if a suitable annotation is present.
 *
 * @param device
 *          The device for which the ontology class shall be retrieved.
 * @return The ontology class as a {@link String} representing the class
 *       URI.
 */
private String generateThingClass(Device device)
{
   String classURI = "";

   // get directly implemented interfaces
   Class<?>[] implementedInterfaces = device.getClass().getInterfaces();

   // iterate over interfaces
   for (int i = 0; i < implementedInterfaces.length; i++)
   {
     // check if the current interface extends
     if (Device.class.isAssignableFrom(implementedInterfaces[i]))
      {
         // get the annotation if exists
         SemanticModel model = implementedInterfaces[i].getAnnotation(SemanticModel.class);

        // check not null
        if (model != null)
            // the annotated class uri
            classURI = model.value();
      }
   }
   return classURI;
}
```

Figure 8. An Excerpt of the metadata harvesting code in the SCRAL.

3.2.4 Supported Technologies

During the second year of the project, additional technologies have been introduced regarding both the waste scenario, thanks to a collaboration with the SmartBin company[8], and the water scenario, with a new water meter simulator generating pseudo-realistic consumption and leakage data for 20k sensors spread in Turin.

The first integration effort has been concentrated on integrating the services offered by the SmartBin company, which reached an agreement with the ALMANAC consortium to share data about more than 100 real-waste bins deployed all around the world, with higher concentrations in UK. Waste-related information is offered by means of a protected-access web service, which allows gathering available data for a subset of the smart bins currently operated by SmartBin. Such a data is integrated in the platform by the SCRAL, and handled in the same way of other similar resources, be they synthetic (generated through the WasteBin emulator module of the SCRAL) or real.

Additionally, a new water meter simulator has been created on the code-base of the waste bin emulator built in year 1. Such a simulator is designed to exploit parallel generation of synthetic

[8] https://www.smartbin.com/

water meters and of relative data using and adaptive set of generation threads. This permits to scale quite well to high numbers of emulated devices, 20 thousand in the water-scenario application. The generation figure is actually quite far from the maximum achievable rate that in the preliminary scalability tests, with no specific optimization, reached nearly 400k bins each generating new data every 10 minutes[9], son a single machine.

3.2.5 Security Framework

Activities at the SCRAL layer also included preliminary design and implementation of the platform security framework, with a particular focus on the SCRAL Policy Enforcement Point. After a first state-of-the-art survey, we selected the PicketLink[10] library from JBoss as a basis for implementing the platform security modules. In the first, preliminary, prototype simple policy examples have been defined in XACML files and used as static definitions against which testing the SCRAL PEP. Figure 9 shows one sample policy denying access to devices made by the ACME Corporation.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy PolicyId="DeviceDenialExample1" RuleCombiningAlgId="urn:oasis:names:tc:xacml:1.0:rule-
combining-algorithm:permit-overrides">
   <Target/>
   <Rule RuleId="RejectDevice" Effect="Deny">
    <Target>
     <Subjects>
      <AnySubject/>
     </Subjects>
     <Resources>
      <AnyResource/>
     </Resources>
     <Actions>
      <AnyAction/>
     </Actions>
    </Target>
    <Condition FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
     <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-one-and-only">
      <SubjectAttributeDesignator DataType="http://www.w3.org/2001/XMLSchema#string"
AttributeId="urn:oasis:names:tc:xacml:1.0:subject:manufacturer"/>
     </Apply>
     <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">ACME
Corporation</AttributeValue>
    </Condition>
   </Rule>
</Policy>
```

Figure 9. Sample device denial policy.

Users, roles and access policies will be better defined in the upcoming months, and design choices and solutions will be reported both in the internal deliverable ID5.2.2 (for what concerns the Smart City Resource Adaptation layer) and as short summary (appendix) in the periodic activity report.

[9] Test have been executed on an Intel core i5 laptop, with 8 Gbytes of RAM (of which 512Mbyte assigned as maximum heap space).
[10] http://picketlink.org/

4. Virtualization Layer

Due to the central location of the Virtualization Layer within the ALMANAC architecture, it is interacting with many of the other ALMANAC components. This chapter will attempt to sum-up the Virtualization Layer's role, based on the overall system architecture definition, as well as details from the implementation so far, and implementation plans for the remaining period (this component is due in ID5.3.2 in February 2016).

From the DoW, the Virtualization Layer's role is to *"**ensure that any aspect related to Smart City resources and their relationship is modelled consistently and can be shared through open, web-oriented protocols**"*.

This role is further detailed in chapter 4.3 of D3.1.2 "System Architecture Analysis & Design Specification", specifying in particular the different types of virtualizations to support: proxying to internal ALMANAC components, routing requests to the local or to remote ALMANAC instances, wiring some ALMANAC internal components to hide the complexity from end-users, and transforming some data formats to ease interaction with end-users and third-party services.

The Virtualization Layer's role can be summarized as to providing various methods to end-user applications that actively want to interact with the ALMANAC platform in a consistent manner, with open data format standards, and through Web-oriented protocols.

So far, the Virtualization Layer has focused on the following roles:

- API layer for end-user applications not running LinkSmart (e.g. for Web browsers, mobile apps);
- Transparent data format conversion service into different Web-friendly open formats;
- Adding compatibility with popular third-party Web services for them to consume ALMANAC data (e.g. Google Maps);
- Web protocol allowing pushing to listening end-users the events of interest coming from e.g. the Storage Manager and the SCRAL, as well as allowing end-user applications to send requests to the Virtualization Layer at a higher frequency;
- Building a bridge to LinkSmart.

The Virtualization Layer is mostly state-less, as it does not store information in its own database, besides a little amount in memory. Instead, it must ask the appropriate components in the ALMANAC platform for the needed information.

To help further understanding the scope of the Virtualization Layer's role, it may help to list explicitly some closely related activities that fall outside its role.

In particular, it is the Adaptation Layer's role (SCRAL) and not the Virtualization Layer's role to:

- Interact with physical sensors
- Interact with the capillary network
- Query non-ALMANAC services (such as SmartSantander)

Similarly, it is the Data Management's role and not the Virtualization Layer's role to:

- Keep a database about the known sensors
- Keep a database about the last known values
- Store rules of interest for clients
- Generate events from the aggregated data (Data Fusion)

Finally, it is the role of the "Ontologies and Semantic Representation Framework" to:

- Store a semantic graph of selected data of relevance

- Provide a semantic query service that the Virtualization Layer can use.

4.1 VL Architecture overview

The Virtualization Layer is the main public end-point exposed to end-user applications, as visible in Figure 10. Although it is possible for systems administrators to expose the Data Management and SCRAL components to the public Internet (thanks to their REST interface), this will probably not be the main situation, and most requests coming from end-user applications will go through the Virtualization Layer and the respective policy management.

The Virtualization Layer is in charge of adapting and routing the end-user applications' request to the appropriate ALMANAC component, and returning the results using an adequate format. This is for the "pull" approach, in which clients are asking the ALMANAC platform for something (data or actuation), and expect a relatively fast answer.

A "push" approach is also offered to clients, who can subscribe to a certain type of information, and receive it in near real-time when the Virtualization Layer is informed by the SCRAL or the Data Management that new data is available, and after transformation into an adequate format. This will be detailed in chapter 4.2.2.

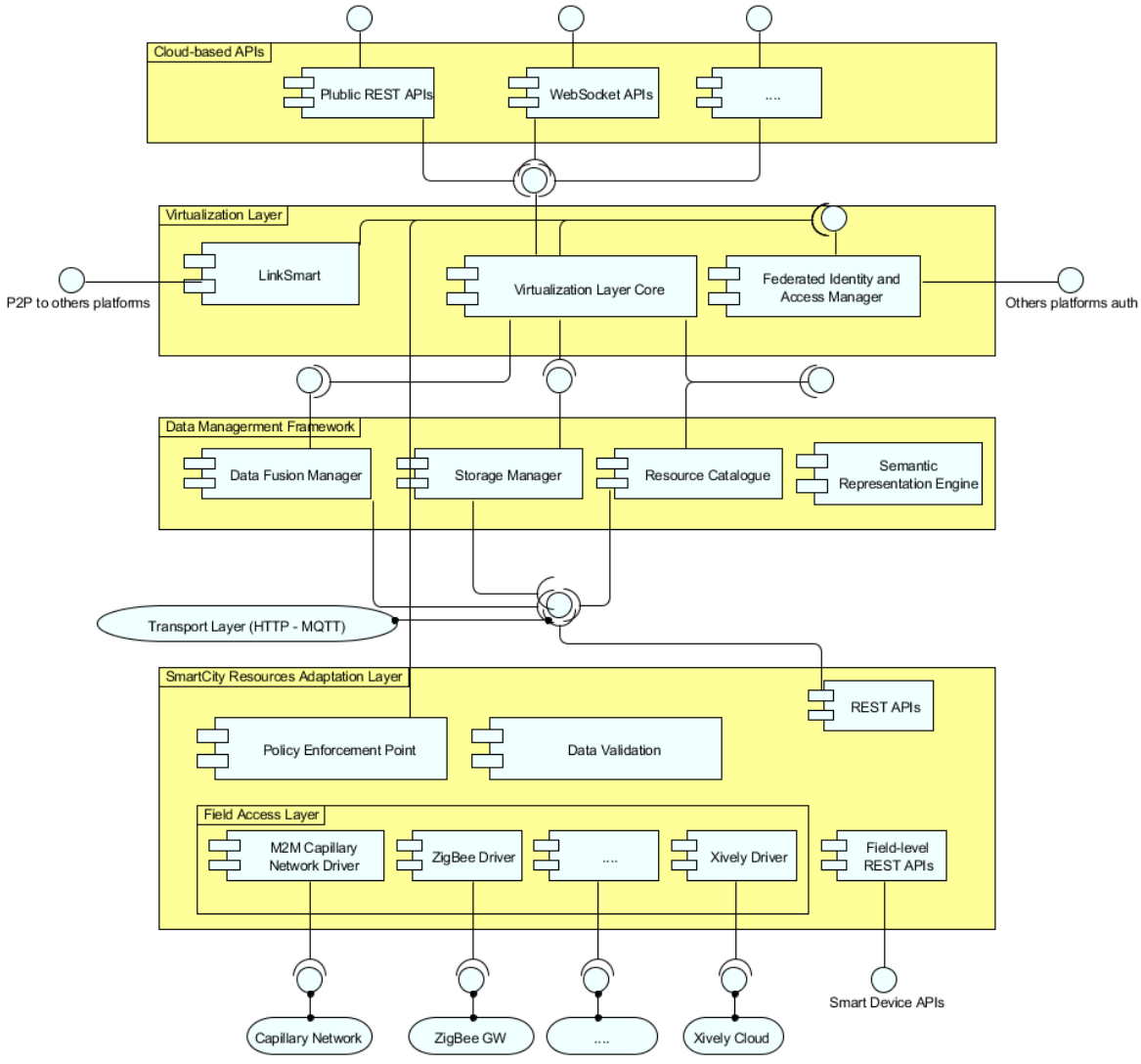


Figure 10 - Component diagram of the ALMANAC platform from D3.1.2 "System Architecture Analysis & Design Specification"

4.1.1 Proxying to ALMANAC internal components

The proxying of requests/responses and events to/from the different ALMANAC internal components make the ALMANAC instance look like a single Web service to the eye of external applications.

One of the central roles of the Virtualization Layer is to help end-user applications with the discovery of services (e.g. sensors) of interest.

To do so, different approaches are taken, depending on the situation:

1. The Virtualization Layer can query the local Data Management instance for resources, using the Data Management's API such as `` `?like=xxx` ``. It will then return the list of matching resources to the client, possibly using another data format when appropriate. This is the most classic approach.
2. The Virtualization Layer can query the local SCRAL using the SCRAL's API such as `'getDeviceByType()'` (currently), and return the list of matching resources. The SCRAL may be able to filter by some aspects not known to the Data Management.
3. The Virtualization Layer can formulate a request to the "Ontologies and Semantic Representation Framework", and return the list of matching resources. This should allow more expressive requests by using a combination of standard ontologies.
4. The Virtualization Layer can formulate a request using the LinkSmart federation interface, which will in turn use LinkSmart's ability to discover resources based on e.g. their description (`'getServiceByDescription()'`) or a set of attributes (`` `getServiceByAttributes()` ``). This is a slower process that does not scale to the same extent than the previous approaches, but which leverages the power of a federated architecture, in particular the ability to discover sensors that are not directly exposed to the SCRAL of the local ALMANAC instance.

Besides the discovery of services, end-users also need to get data from specific sensors as well as to take advantage of some of the distinct abilities of specific ALMANAC components, such as the actuating sensors through the SCRAL.

Therefore, end user applications interact with the Virtualization Layer, which will route the request similarly to the case of service lookup requests. The Virtualization Layer will adapt both the request and the response for providing maximum convenience to end-user applications (a subset of that is described in chapter 4.3).

In order to comply with ALMANAC's promise of scalability and near-realtime delivery, the Virtualization Layer takes advantage of knowledge of events passing inside the ALMANAC instance (initially through HTTP calls, now with MQTT). Indeed, events emitted by the SCRAL and the Data Management can immediately be forwarded by the Virtualization Layer to clients listening for this type for information.

This approach is necessary for the Virtualization Layer, both to be able to scale to a larger amount of simultaneous clients (by reducing the overhead of repeated HTTP request), as well as to reduce the delivery time to something that can be qualified "near-realtime".

4.1.2 Routing of requests/responses inside a federation

The Virtualization Layer will route some requests/responses either to an ALMANAC internal component of the same ALMANAC instance, or to another ALMANAC instance when needed (through the LinkSmart federation interface (i.e. Network Manager)), leveraging the federated nature of the ALMANAC platform.

The routing decision will be taken based on a set of internal rules, taking advantage of a naming mechanism similar to DNS (developed in Section 5 of D3.1.2 "System Architecture Analysis & Design Specification"), as well as LinkSmart new MQTT broadcasting abilities (Figure 11) allowing the Virtualization Layers of distinct ALMANAC instances within a federation to talk together.

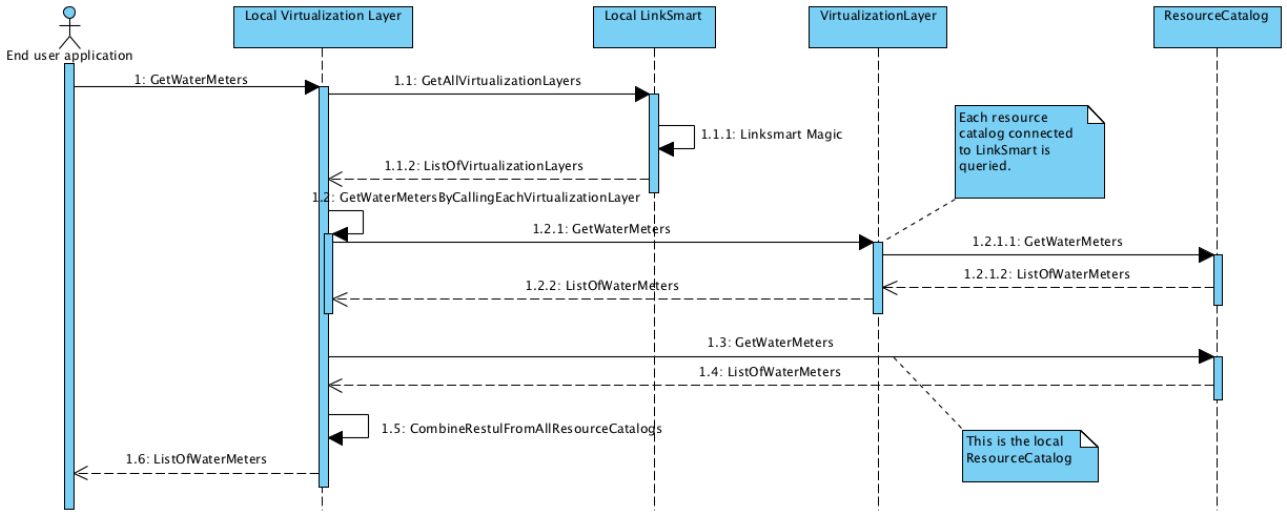


Figure 11: Illustration of an example of call for water meters in a federation of ALMANAC instances.

4.1.3 Wiring ALMANAC components

For some selected types of requests, the Virtualization Layer will "wire" ALMANAC internal components together, in such a way that requests requiring the collaboration of several ALMANAC internal components may be executed by external applications in a single call. It can be seen as a kind of mashup.

For instance, a request may require discovering some IoT resources via the Resource Catalogue or the Semantic Layer, followed by a query for historical data to the Storage Manager, before being aggregated and returned to the external application.

4.2 VL's communication protocols

4.2.1 HTTP REST

The Virtualization Layer exposes a public HTTP REST interface.

The current implementation does the routing of client's requests, as well as on-the-fly format conversion, according to explicit instructions from the client.

There is a test instance of the Virtualization Layer running at http://almanac.alexandra.dk and as visible in the script extract below, a few explicit paths are made available. For instance:

- http://almanac.alexandra.dk/dm/IoTEntities to route a request to the Data Management
- http://almanac.alexandra.dk/dm-geojson/IoTEntities idem, but with on-the-fly conversion to GeoJSON (see chapter 4.3.1).
- Etc. as visible below:

```
switch (s1) {//Routing
    case 'dm/':     //Proxying to Data Management
        req.url = req.url.substring(s1.length);
        almanac.proxyDataManagement(req, res);
        break;
    case 'dm-geojson/':    //Conversion of Data Management JSON to GeoJSON
        req.url = req.url.substring(s1.length);
        almanac.proxyDataManagementToGeojson(req, res);
        break;
    case 'dm-atom/':       //Conversion of Data Management JSON to ATOM (RSS)
    case 'dm-rss/':
```

```
        req.url = req.url.substring(s1.length);
        almanac.proxyDataManagementToAtom(req, res);
        break;
    case 'dm-txt/': //Conversion of Data Management JSON to TXT
    case 'dm-tsv/':
    case 'dm-csv/':
        req.url = req.url.substring(s1.length);
        almanac.proxyDataManagementToText(req, res,
            s1 === 'dm-csv/' ? 'csv' : 'tsv');
        break;
    case 'scral/':  //Proxying to SCRAL
        req.url = req.url.substring(s1.length);
        almanac.proxyScral(req, res);
        break;
    case 'santander/':     //Proxying to SmartSantander
        req.url = req.url.substring(s1.length);
        almanac.proxySmartSantander(req, res);
        break;
    case '': //Serve a welcome page
        almanac.serveHome(req, res);
        break;
    default: //Serve a static file
        basic.serveStaticFile(req, res);
        break;
}
```

Just to illustrate how the current prototype code looks like, here is the proxying to the SCRAL:

```
proxyScral: function (req, res) {
    req.url = config.hosts.scralPublic.path + req.url;
    proxy.web(req, res, {
            headers: {
                'Connection': 'close',  //Ability to change HTTP headers
                host: config.hosts.scralPublic.headers.host,
            },
            forward: null,
            target: {
                host: config.hosts.scralPublic.host,
                port: config.hosts.scralPublic.port,
            },
            xfwd: true,    //Include X-Forwarded-For header
        }, function (err) {
            basic.serve500(req, res, 'Error proxying to SCRAL: ' + err);
        });
},
```

4.2.2 WebSocket

While HTTP REST is appropriate for many cases, it does not support eventing, and does not support a very high frequency of requests (although good performances can be achieved when using the HTTP Keep-Alive and pipelining mechanisms).

Therefore, for users in need of near-realtime delivery as well as for more intensive users (sending data, receiving data, or both), another protocol is needed for talking to the Virtualization Layer.

Hence WebSocket, which is a full-duplex protocol initiated by HTTP. It is an IETF/W3C standard supported by major modern Web browsers.

In the current implementation, WebSocket support is provided by Socket.IO[11], a popular open source library for doing so, as illustrated in this code extract:

```
ioInit: function (server) {
    almanac._io = require('socket.io')(server);
    almanac._io.on('connection', function (socket) {
            var remoteAddress = socket.request.connection.remoteAddress,
                remotePort = socket.request.connection.remotePort;
            /* ... */
```

[11] http://socket.io

```
            almanac._ioSockets[socket.id] = remoteAddress + ':' + remotePort;
            almanac._io.emit('chat', 'Connected ' +
                almanac._ioSockets[socket.id]);
            socket.emit('chat', 'Welcome ' + almanac._ioSockets[socket.id]);
            console.log('Connected ' + almanac._ioSockets[socket.id]);

            socket.on('chat', function (msg) {//Chat for humans, for testing
                    msg = almanac._ioSockets[socket.id] + '> ' + msg;
                    almanac._io.emit('chat', msg);
                    console.log('Chat ' + msg);
                });

            socket.on('disconnect', function () {
                    var clientId = almanac._ioSockets[socket.id];
                    almanac._io.emit('chat', 'Disconnected ' + clientId);
                    /* ... */
                });
        });
},
```

The Socket.IO library provides some valuable features out-of-the-box, such as auto-reconnect in case of loss of connection client-side and/or server-side, as well as a compatibility fall-back for older Web user-agents.

Humans can try to chat and to see in near-realtime the log of the Virtualization Layer activity at http://almanac.alexandra.dk/socket.html as visible in Figure 12:

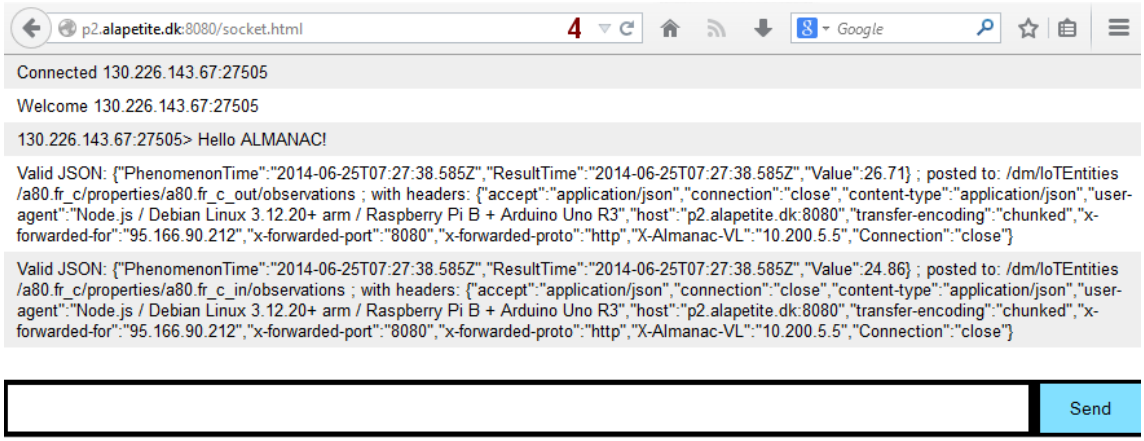


Figure 12 - WebSocket demonstration for humans, also used for debugging purposes.

4.3 VL Data formats

As discussed in the introduction already, one of Virtual Layer's missions is to expose ALMANAC's data in different standard formats for easing interoperability.

So far, the Virtualization Layer's prototype offers on-the-fly transformation to GeoJSON, CSV or TSV.

4.3.1 GeoJSON

GeoJSON[12] is a format for encoding a variety of geographic data structures – as well as custom properties – and based on JSON[13].

Here is a simple sample of how it looks like:

```
{
    "type":"FeatureCollection",
    "features":[
        {
```

[12] http://geojson.org
[13] http://json.org

```
            "type":"Feature",
            "geometry":{
                "type":"Point",
                "coordinates":[
                    9.754045,
                    55.960869
                ]
            },
            "properties":{
                "name":"Thomas' iPhone | Location of phone",
                "description":"A superclever smartphone | Location of the phone"
            }
        },
        {
            "type":"Feature",
            "geometry":{
                "type":"Point",
                "coordinates":[
                    12.4864,
                    55.6793
                ]
            },
            "properties":{
                "name":"Temperature Vanløse",
                "description":"Temperature in Vanløse, Denmark"
            }
        }
    ]
}
```


The main reason for implementing this format is that it is supported by popular third-party services such as Google Maps14.

Figure 13 is a screenshot of a map generated by taking advantage of this GeoJSON format, with some data converted on-the-fly from the Storage Manager, and plotted using the GeoJSONLint[15] third-party service.

[14] Google Maps API with GeoJSON support
https://developers.google.com/maps/documentation/javascript/reference#Data.GeoJsonOptions
[15] http://geojsonlint.com/validate?url=http://almanac.alexandra.dk/dm-geojson/IoTEntities

Figure 13 - Map produced by a third-party service using the GeoJSON format

The script below shows how a non-blocking request is performed by the Virtualization Layer, which is then sleeping and reacting upon the completion event `.on('end', {})` before calling the conversion to GeoJSON and returning the response to the client:

```
proxyDataManagementToGeojson: function (req, res) {
    var get = http.request({
            host: config.hosts.masterStorageManager.host,
            port: config.hosts.masterStorageManager.port,
            path: config.hosts.masterStorageManager.path + req.url,
            method: 'GET',
            headers: {
                'Accept': 'application/json',
                'Host': config.hosts.masterStorageManager.headers.host,
                'Connection': 'close',
            }
        }, function(res2) {
            var body = '';
            res2.setEncoding('utf8');
            res2.on('error', function (err) {
                basic.serve500(req, res, 'Error getting from DataManagement: ' +
                    err);
            });
            res2.on('data', function (chunk) {
                body += chunk;
            });
            res2.on('end', function () {//Example of non-blocking event
                try {
```

```
                    //Do the conversion to GeoJSON (subroutine not shown)
                    var geoJson = almanac._dmToGeojson(JSON.parse(body));
                    res.writeHead(res2.statusCode, {
                            'Access-Control-Allow-Origin': '*',
                            'Access-Control-Allow-Methods': 'GET',
                            'Content-Type': 'application/json; charset=UTF-8',
                            'Date': res2.headers.date,
                            'Server': basic.serverSignature,
                        });
                    //Send the response back to the client
                    res.end(JSON.stringify(geoJson));
                } catch (ex) {
                    basic.serve500(req, res,
                        'Error GeoJSON conversion from DataManagement: ' + ex);
                }
            });
        });
    get.end();
},
```

It should be noted that GeoJSON stores the geographical coordinates in the following order: [longitude, latitude].

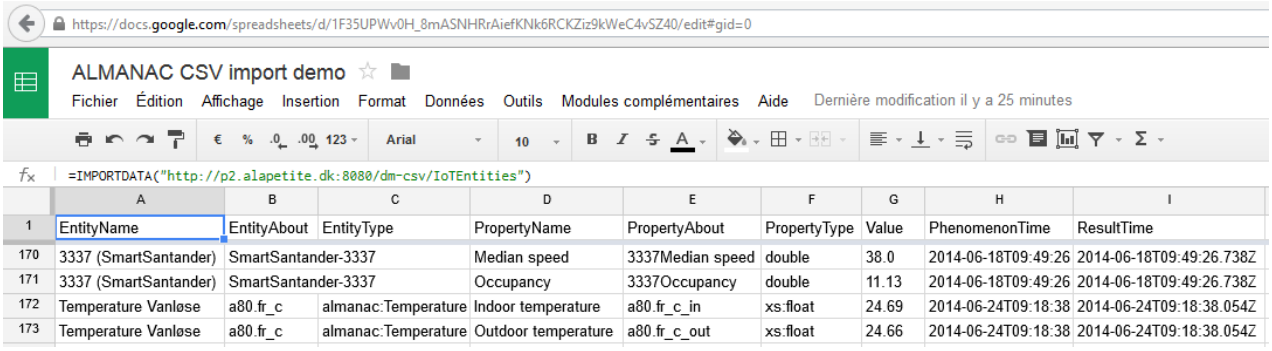
4.3.2 CSV / TSV

When browsing sources of open data (e.g. open government data from UK[16]), the venerable plain-text format CSV is by far the most popular.

Note: CSV is comma-separated; its twin format TSV is tab-separated. Both are supported.

This is already a good reason to support it. Furthermore, it is one of the most convenient formats for importing data into spreadsheets, a tool that many end-users are familiar with.

As illustrated by Figure 14, thanks to the on-the-fly conversion operated by the Virtualization Layer, it is possible to conveniently import ALMANAC data into Google Docs Spreadsheet, using the spreadsheet formula `=IMPORTDATA("•http://almanac.alexandra.dk/dm-csv/IoTEntities")`, which could be further refined using the Storage Manager's API.



	A	B	C	D	E	F	G	H	I
1	EntityName	EntityAbout	EntityType	PropertyName	PropertyAbout	PropertyType	Value	PhenomenonTime	ResultTime
170	3337 (SmartSantander)	SmartSantander-3337		Median speed	3337Median speed	double	38.0	2014-06-18T09:49:26	2014-06-18T09:49:26.738Z
171	3337 (SmartSantander)	SmartSantander-3337		Occupancy	3337Occupancy	double	11.13	2014-06-18T09:49:26	2014-06-18T09:49:26.738Z
172	Temperature Vanløse	a80.fr_c	almanac:Temperature	Indoor temperature	a80.fr_c_in	xs:float	24.69	2014-06-24T09:18:38	2014-06-24T09:18:38.054Z
173	Temperature Vanløse	a80.fr_c	almanac:Temperature	Outdoor temperature	a80.fr_c_out	xs:float	24.66	2014-06-24T09:18:38	2014-06-24T09:18:38.054Z

Figure 14 - Automatic import into Google Docs Spreadsheet

The software process is similar to the conversion to GeoJSON explained in greater details in the previous section.

4.3.3 ATOM (similar to RSS)

Some preliminary work has been done to support the ATOM[17] format (RFC 4287), as it can be consumed by a large amount of systems aggregating or reacting upon news and other events.

For instance, the popular service IFTTT[18] can consume ATOM/RSS, and then trigger a "recipe" such as sending an alert/SMS to a smartphone, switching a lamp on or off, etc.

[16] http://data.gov.uk/data/search
[17] ATOM http://tools.ietf.org/html/rfc4287

5. Semantic Representation Framework

5.1 Metadata Framework

5.2 Overview

The main purpose of this Metadata Framework is to maintain an easy programmatic access to rich, graph-based domain models developed as part of the T5.3 activities. It differs from the application scope of the "Storage manager" component, as it does not serve the mass storage of measurement data, events etc. but it manages the structured description of their sources (devices, services) providing for query and inference services on such data as reported in Table 1.

Table 1. Role of the Storage Manager in comparison with the Semantic Representation Layer

Storage Manager	Semantic Representation Layer
Generalized access to mass storage of data[19]	Graphs, descriptive schema, dynamic
Large amount of flat data (recent and historical values)	Complex metadata, recent values
Explicit value selection (OGC SensorThings)	Graph pattern matching (SPARQL)

Implementation of this component is fully inline with the LinkSmart Middleware. The ongoing development of the framework is part of the T5.3 activities.

5.2.1 Metadata Framework Architecture

The Metadata Framework maintains a graph representation of domain entities, together with their attributes, and of relationships between them, and offers means for querying and navigating such data. The framework acts as a transparent proxy to any triple-store compatible with the SPARQL 1.1 protocol[20]. It considerably augments the typical repository functions and interface coverage, by providing native Java/OSGi integration and remote HTTP APIs. The following sections will introduce the main constituents of the Metadata Framework's architecture depicted in Figure 15.

[18] https://ifttt.com
[19] The StorageManager supports further storage technologies. See D6.1 for reference.
[20] Examples of such graph stores are Apache Fuseki or the Openlink Virtuoso Server.

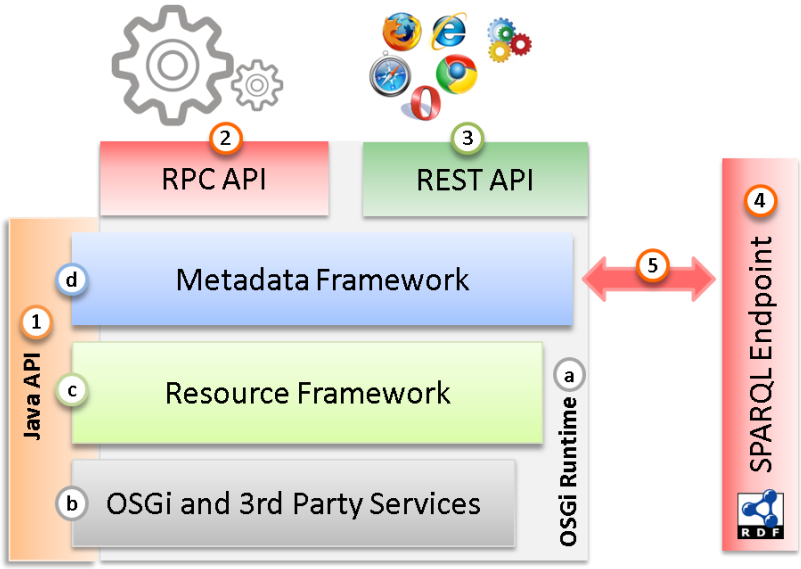


Figure 15. Metadata Framework outline

OSGi Runtime environment (a)

A runtime environment implementing the *Open Services Gateway initiative* (OSGi) specification[21] for dynamic, service-oriented component systems in Java is the execution platform for the Metadata Framework and the core building block. Its *Service Registry* acts as the central integration hub for publication and resolution of services.

OSGi and 3rd party services (b)

Next to the core OSGi services the Metadata Framework leverages the *Service Component Runtime* (Declarative Service Specification[22]) for declarative component resolution and lifecycle management, the *ConfigurationAdmin* service[23] for management of persistent component configuration and other 3rd party services.

Resource Framework (c)

The *Resource Framework* defines a thin generic service layer for RESTful management of resources, including among others their life-cycle management, querying, conversion and validation. Default implementations of this API for different data types and persistence technologies (Java persistence API[24], native JSON[25] and XML document databases[26] etc.) are part of the framework.

Metadata Framework (d)

The *Metadata Framework* implements the resource management API for graph-based "semantic resources". It is compliant to and extends the SPARQL 1.1 web protocols by provision of persistent SPARQL queries/updates and support of RESTful manipulation of resources at more fine grained levels (discrete graph fragments) as explained in the next section 5.3.

Java data models and service interfaces (1)

According to the OSGi specification, data models and service interfaces of the installed bundles have to be explicitly exported and conversely imported via the manifest headers mechanism in order to be accessible to other bundles, allowing the framework to manage dependencies among them. The

[21] http://www.osgi.org/
[22] http://wiki.osgi.org/wiki/Declarative_Services
[23] http://www.osgi.org/javadoc/r4v42/org/osgi/service/cm/ConfigurationAdmin.html
[24] http://download.oracle.com/otndocs/jcp/persistence-2.0-fr-oth-JSpec/
[25] https://www.mongodb.org/
[26] http://exist-db.org/exist/apps/homepage/index.html

packages `eu.linksmart.resource` and `eu.linksmart.metadata` and their sub-packages comprise such a public interface of the abovementioned components (c) and (d).

Remote-procedure call (RPC) interfaces (2)

The RPC API serves the remote *invocation of operations with custom semantics* (e.g. data queries, actuation commands or calculations). These lower-level calls are often performed by higher levels services and not by human clients, as indicated by the icons. This interface handles SPARQL 1.1. Protocol requests and the invocation of custom persistent queries and updates.

Representational state transfer (REST) interfaces (3)

The REST API serves the retrieval and manipulation *of resources* identified by a Universal Resource Identifier (URI) by exchange of resource representations, i.e. their discrete textual serializations. The requests are issued towards a uniform service interface built upon the standard HTTP methods (verbs), mainly `GET`, `PUT`, `POST` and `DELETE` and result in CRUD[27] operations on the internally maintained resource state. Different root path prefixes are used to distinguish the **RPC** and **REST endpoint** semantics:

`/resource/...` Root path of RESTful operations, a uniform interface applied to any resource type. This remote API is used to manage the lifecycle of resources.

`/service/...` Root path for invocation of RPC-services (via `GET` or overloaded `POST`). Some of the provided resources (e.g. persistent queries) allow for execution at this service endpoint.

SPARQL 1.1. Endpoint (4)

An SPARQL endpoint is the address/reference to an HTTP listener capable of handling SPARQL Protocol requests (standardized interface to an RDF graph store).

SPARQL 1.1. Protocol (5)

SPARQL 1.1 Protocol[28] defines a convention for transmission and execution of SPARQL Queries and Updates to a SPARQL endpoint via HTTP. LSM uses this protocol in communication to the underlying RDF graph store.

5.3 Semantic resource handling

In contrast to file-based, document-oriented or RDBMS systems which share an implicit concept of entity boundaries (like a file, XML root element, JSON object or SQL table etc.) there is no simple mean, in graph-based RDF models, to shape boundaries of a particular sub-graph (entity description). The available options are:

1. Usage of **named graphs per entity** to provide context to and consolidate all statements about a single entity. The drawback of this approach is a missing support within the RDF abstract model itself (only available via TriG[29] serialization and manipulation level via SPARQL named graph support[30]), high fragmentation of the data base and difficulty to link and query.
2. Usage of **intermediate** blank nodes[31] to express boundaries, context, provenance etc. Drawback: this solution would require a proprietary data schema and framework implementing such a blank node traversal and would be incompatible with most of the existing vocabularies.

[27] http://en.wikipedia.org/wiki/Create,_read,_update_and_delete
[28] http://www.w3.org/TR/sparql11-protocol/
[29] http://www.w3.org/TR/trig/
[30] http://www.w3.org/TR/sparql11-query/#namedGraphs
[31] http://en.wikipedia.org/wiki/Blank_node

3. Usage of **SPARQL 1.1. Query and Update** languages to purposefully construct and manipulate entity sub-graphs. Drawback: development of a management framework and an initial configuration effort setting up the required queries/updates. Such a programmatic handling of graph fragments may optionally build on top of an additional named graph organization 1).

The latter solution 3) was selected for implementation in LSM, since it employs a standard tool chain and is not limited to a particular data schema.

5.3.1 Processing pipeline

As mentioned previously, SPARQL 1.1 query and update expressions are used to ad-hoc "construct" discrete entities out of the graph continuum. For this purpose the Metadata Framework adopts the generic resource processing pipeline of the Resource Framework as depicted in Figure 16:

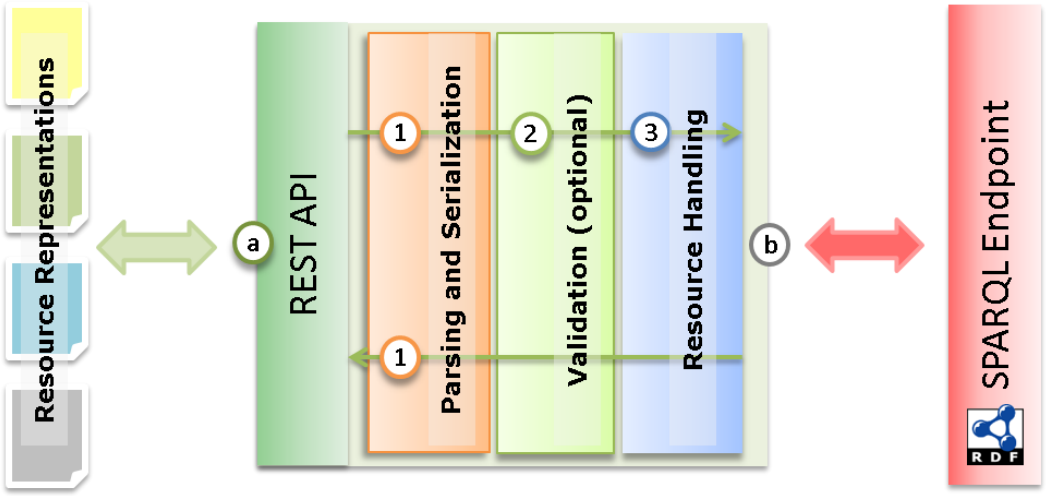


Figure 16. Semantic resource processing pipeline

REST-API Clients (a)

External clients of the REST API interact by invoking the uniform interface and exchanging resource representations. These vary in terms of resource type and media type (serialization).

Parsing and serialization handlers (1)

The textual resource representations are parsed on input or serialized on output to/from an internal graph model (Apache Jena `Model`[32]). Independently of the resource type a `ResourceRequest` object wrapping the resource is created in accordance to the request properties (HTTP headers, query or form parameters) and passed to the pipeline.

Input validation handlers (2)

Resources passed along the `PUT` or `POST` requests are optionally validated by a `ResourceValidator` service. Validation of semantic resources is by default implemented via SPARQL `ASK` query (supplied as part of an RDF-based handler definition). It returns true, for a valid (matching) graph input, false otherwise. This example query is registered to validate resources of the type `PrefixMapping` and to ensure, that neither the newly supplied prefix, nor the associated namespace URI are in use already:

```
PREFIX rdfa: <http://www.w3.org/ns/rdfa#>
ASK
{
 # Bin "prefix" and "uri" of local resource. Neither of both should be empty.
```

[32] http://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/rdf/model/Model.html

```
    {
     ?m a rdfa:PrefixMapping .
     ?m rdfa:prefix ?prefix. FILTER (strlen(str(?prefix)) != 0) .
     ?m rdfa:uri ?uri. FILTER (strlen(str(?uri)) != 0) .
    }
    # Neither "prefix" nor "uri" mapping should already exist.
    {
      SERVICE ?endpoint
       {
     FILTER NOT EXISTS { _:m1 rdfa:prefix ?p.  FILTER (str(?p) = str(?prefix)) } .
     FILTER  NOT EXISTS { _:m2 rdfa:uri ?u.  FILTER (str(?u) = str(?uri)) } .
       }
     }
   }
```

Resource handlers (3)

Based on the parameters of the `ResourceRequest` the most appropriate `RequestHandler` service is chosen to handle the request. For this purpose `RequestHandler` instances are expected to indicate their applicability (type of request they are capable of handling and identifiers of individual resources or resource types they apply to). The Metadata Framework ships with a set of default request handlers that serve operations on simple semantic resources out of the box. Such, for example, if no custom handler could be found the default `ReadHandler` implemented via a SPARQL `CONSTRUCT` query is used. A complete example of the `ReadRequest` handler is given:

```
<rdf:RDF
    xmlns:ls="http://linksmart.eu/ontology#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <ls:ConstructQuery rdf:about="urn:res:ReadRequestHandlerResource">
   <rdf:type rdf:resource="http://linksmart.eu/ontology#ResourceHandler"/>
   <rdfs:label>Generic retrieval handler for flat resources</rdfs:label>
   <ls:service>
      <ls:ReadRequestHandler>
         <ls:targetClass rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
      </ls:ReadRequestHandler>
    </ls:service>
    <ls:source>
      <ls:ParameterizedSparqlString>
        <rdf:value rdf:datatype="http://www.w3.org/2001/XMLSchema#string"><![CDATA[
          CONSTRUCT { ?resource ?predicate ?object }
          WHERE {  ?resource ?predicate ?object }
        ]]></rdf:value>
      </ls:ParameterizedSparqlString>
    </ls:source>
    <ls:input>
      <ls:Parameter ls:name="resource" rdfs:comment="URI of resource to be retrieved" />
    </ls:input>
  </ls:ConstructQuery>
</rdf:RDF>
```

This fallback handler retrieves all immediate properties of a node being read (via `HTTP GET`), when no custom handler was provided.

SPARQL 1.1 endpoint request (b)

A standard SPARQL 1.1 Protocol request is created and issued against the remote RDF graph store.

5.3.2 Remote HTTP APIs

The remote HTTP service interfaces (on top of Figure 15) are the main entry point for Metadata Framework web clients. They implement and extend SPARQL 1.1 Web Protocols allowing for RPC (Remote Procedure Call) and REST-style interactions:

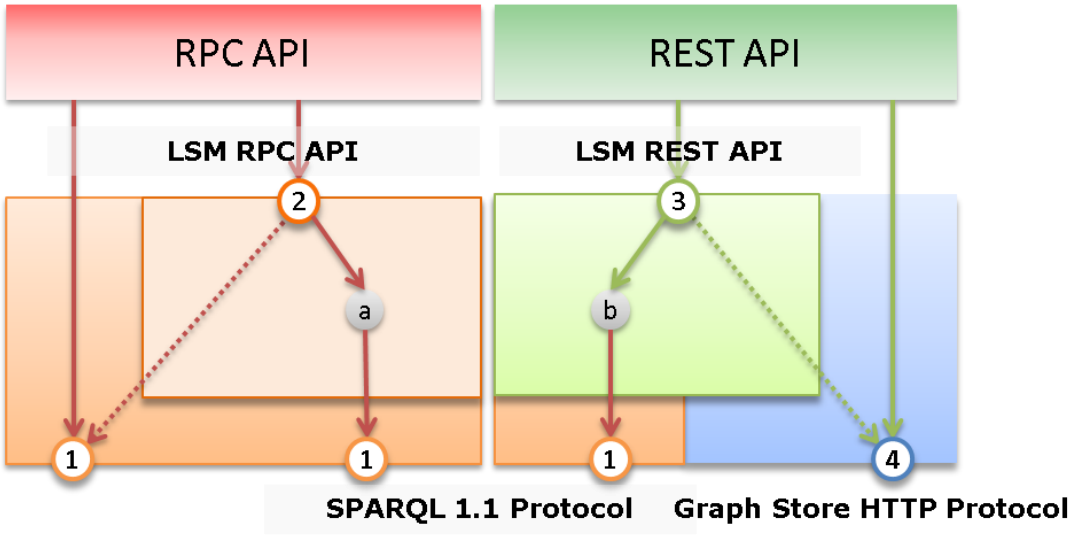


Figure 17. LSM Remote APIs

The RPC API comprises the *invocation of services* to query or update data whereas the REST API serves *the retrieval and manipulation of (textual) resource representations*. Client requests to remote HTTP interfaces offered by the Metadata Framework are transparently proxied and transformed to invocation of standardized interfaces on a SPARQL endpoint **(1)** and **(4)** at the bottom.

SPARQL 1.1 Protocol[33] **(1)**

Standard interface exposed by a SPARQL endpoint, **mandatory** for (2).

Metadata Framework RPC API (2)

The RPC API accepts query or update invocations. Standard requests are immediately proxied to the endpoint (1), custom requests are internally processed (a) and forwarded to endpoint (1). "Processing" may involve query rewriting, resolution and parameterization of stored queries and further request enhancements etc.

Metadata Framework REST API (3)

If available, the REST API will proxy standard Graph protocol requests to (4), otherwise standard requests are converted into invocation of the RPC API (1) according to predefined translation rules. The API extends standard API (4) by exposing wide range of resources (URIs) for sub-graph manipulations (b).

Graph Store HTTP Protocol[34] **(4)**

Graph Store protocol is an optional standard interface for RESTful operations on graph level.

5.3.3 SPARQL Protocol Service URL Design

The SPARQL Protocol standard does not predefine a URL convention for SPARQL Protocol Services (endpoints). It states that the implementation of the data-modifying "update" operation shall be optional for security reasons. We chose to separate the service endpoints for *read-only* (query) and *read-write* (update) operation semantics in order to support different security policies, specifically tailored on the allowed interactions between externals service consumers and resources managed by the Semantic Representation Framework (for the same reason the underlying OSGi services are distinguished). The SPARQL Protocol Services are assumed to co-exist in a context of other RPC services provided by the platform, therefore the standard RPC prefix path "`/service`" is assumed:

[33] http://www.w3.org/TR/sparql11-protocol/
[34] http://www.w3.org/TR/sparql11-http-rdf-update/

Table 2. RPC-endpoints for SPARQL Protocol in Metadata Framework

Endpoint URL	Description
`/service/sparql/query`	RPC-endpoint for execution of ad-hoc and persistent SPARQL 1.1 queries (read-only semantics, no data updates)
`/service/sparql/update`	RPC-endpoint for execution of ad-hoc and persistent SPARQL 1.1. updates (write-only semantics, no data retrieval)

5.3.4 Persistent queries and updates

Like other resources SPARQL query and update statements might be stored using the REST-API and invoked by a request to above mentioned endpoints:

Table 3. Examples of SPARQL RPC-endpoint invocations

Request example	Description

`POST /service/sparql/update/res:id251`
 	
`temperature=10.3`
	Request to update a value by executing a SPARQL Update specified inline by its compact resource URI. The required parameters are supplied as `application/x-www-form-urlencoded` request payload.

`POST /service/sparql/update`
 	
`update-name=res:id251&temperature=10.3`
	Request to update a value by executing a SPARQL Update referenced by the mandatory parameter `update-name`.

5.3.5 SPARQL Graph Protocol URL Design

The standard Graph Store HTTP Protocol operates on level of entire graphs which are comparable to entire relational databases, NoSQL collections or file folders. Content manipulation at graph-level is too coarse grained for applications targeting large number of resources. The main contribution of the Metadata Framework is the definition and implementation of a standards-based infrastructure to allow the retrieval and modification of resources at configurable sub-graph levels. A conceptual summary of the current REST API is given. All requests URLs start with the path `<prefix>: /resource/semantic`. Resources and their classes are identified by means of compact URIs[35].

Table 4. Metadata Framework REST API

Function	Request	Description
Resource listing	**GET** `/<prefix>/<class_CURIE>`	Request to list resource representations of given type (class).
Resource creation	**POST** `/<prefix>/<class_CURIE>` `...representation...`	Request to create a new resource of given type based on supplied representation.
Programmatic resource creation	**POST** `/service/sparql/update` **update-name**`=...&foo=...`	Request to create a new resource by executing an annotated SPARQL Update configured via query parameters.
Resource retrieval	**GET** `/<prefix>/<resource_CURIE>`	Request to retrieve a representation of given resource in a particular representation (format).
Resource replacement	**PUT** `/<prefix>/<resource_CURIE>` `...representation...`	Request to replace an existing resource by supplied representation.
Resource extension	**POST** `/resource/<resource_CURIE>` `...partial representation...`	Request to augment the given resource by given **partial** representation. The supplied graph will be unified with contents of the repository.
Resource	**DELETE** `/resource/<resource_CURIE>`	Request to delete specified resource.

[35] http://www.w3.org/TR/curie/

deletion		

5.3.6 Storage context

The storage context of a semantic resource is defined by the *repository* URL (comparable to a relational database system) and its *graph* (comparable to a database or collection), which is either the implicit, "default" graph or a "named graph" identified by a URI. The notion of a named graph[36] as management concept was introduced by the SPARQL standard[37], in which queries operate on RDF Datasets[38] – the union of a default graph and selected named graphs. In compliance with the Graph Store HTTP Protocol, query parameters `default`, `graph` and `repository` are used to express resource's storage context:

Storage context supplied as (optional) query parameter	
`/resource/...`	Implicit reference to the default graph and a pre-configured, default repository
`/resource/...?default`	Explicit reference to the default graph and repository
`/resource/...?``repository`**`=<repository>`**	Explicit reference to a non-default repository URL
`/resource/...?``repository`**`=<repository>&`**`graph`**`=<graph>`**	Explicit reference to a non-default repository and graph URI. Default configuration is used if both are omitted

5.3.7 Platform integration

Almanac's Semantic Representation Layer allows any platform client to maintain, retrieve and query structured metadata. The *Smart City Resources Adaptation Layer* (SCRAL) may rely on it for description of the exposed devices and corresponding functionality. It is a natural extension to the *Resource Catalogue* enhancing its directory and look-up services by a powerful query language (SPARQL), built-in inheritance and reasoning (thanks entailment regimes support[39] of the underlying RDF databases).

5.4 Smart City Ontologies

The design of the ALMANAC Smart City ontology adheres to a well-defined set of guidelines synthetized during the project activities (and formerly reported in ID5.4.1). These guidelines provide suggestions on vocabularies to be considered and linked, on annotation and mapping techniques and on practical modelling solutions, and patterns, to adopt. They include:

Vocabularies

- Prefer established vocabularies vs proprietary, "home-made" schemes.
- Research and selection other than on development of an own schema, which must always be motivated (e.g., no other schema available/applicable for the given knowledge domain).
- Consider aspects like: maturity, expressiveness, community and software support.

[36] http://en.wikipedia.org/wiki/Named_graph
[37] http://www.w3.org/TR/sparql11-query/#namedGraphs
[38] http://www.w3.org/TR/sparql11-query/#rdfDataset
[39] http://www.w3.org/TR/sparql11-entailment/

- Use possibly one, single vocabulary for any particular purpose, avoiding redundant descriptions.

Annotation, mapping

- Provide mappings to existing established vocabularies whenever possible
- If applicable, prefer built-in annotation properties[40] of the W3C Semantic Web standards: (`rdfs:label, rdfs:comment` etc.)

5.4.1 Relevant vocabularies

As a general guideline, standardized and relevant models should be re-used whenever possible. The set of reference models considered in the ALMANAC Smart City ontology design is iteratively updated during the work package activities and, starts from a preliminary survey of currently adopted modelling frameworks. A subset of relevant resources has been identified among many models and ontologies defined in the Linked Open Data community, and includes:

- The Semantic Sensor Network Ontology (SSN);
- The DogOnt Ontology Modeling for Intelligent Domotic Environments;
- The SCRIBE IBM Smart City ontologies (currently not published yet);
- The models listed in the Smart Cities ontology catalogue, among others: Places Ontology, Cadastre and Land Administration Thesaurus (CaLAThe).
- The models listed in the Linked Open Vocabularies (LOV) vocabulary repository;
- The Schema.org vocabulary for representing well known geographical properties, e.g., latitude and longitude;
- The GoodRelations ontology for representing any kind of goods, products and services, and the relations involving their offering, exchanges, etc.;
- The MUO unit of measure ontology, representing unit of measures according to the UCUM vocabulary (including International System of Measures values and other relevant units);
- The vcard ontology for representing name and addresses of people and organizations;
- The GeoSPARQL vocabulary and functions, to support representation and querying of geo-spatial data.
- The GeoNames ontology, for representing cities and other geographical entities.

5.4.2 Ontology Design and Development

A mature web-based ontology editor was chosen to support distributed, collaborative development and annotation of the ontology files: WebProtege[41]. The editor instance with pre-loaded ontologies, including the current ALMANAC Smart City ontology is available online.[42]

A first, preliminary, modelling effort for representing smart city data with respect to waste management issues has been performed, leading to the definition of a light-weight "waste bin" ontology (see Annex 8.1 for ontology source in Turtle format). At the same time a preliminary experimentation with the IoT branch of the DogOnt ontology[43] has started, aiming at reusing the valuable device modelling concepts therein. This last model is linked with the former at the bin-definition level, by means of `owl:equivalentClass` constructs[44].

[40] http://www.w3.org/TR/owl2-syntax/#Annotation_Properties
[41] https://github.com/protegeproject/webprotege/releases
[42] http://almanac.fit.fraunhofer.de:8080/webprotege
[43] http://iot-ontologies.github.io/dogont/, https://github.com/iot-ontologies/dogont/tree/iotdogont
[44] Note that the use of `owl:equivalentClass` does not imply class equality. Class equality means that the classes have the same intensional meaning (denote the same concept), whereas equivalence means that the two class descriptions involved have the same class extension (i.e., both class extensions contain exactly the same set of individuals).

Waste Bin ontology

The rationale of the ALMANAC waste bin ontology is to focus on waste-related issues with an open, extensible and scalable approach. According to this design goal, no assumption is performed on the remaining smart city data, and modelling is restricted to the aspects specifically related to waste collection and management, e.g., by representing waste bins, type of disposables generated by the citizenship and so on. All represented concepts leverage existing, well-known definitions of properties (and classes / super-classes) with a typical Linked Open Data approach. In such a sense, for example, the city concept is directly linked (`owl:equivalentClass`) to the corresponding schema and places concepts.

The ontology mainly represents waste bins, waste types and the geographical / administrative context in which they are deployed. It is organized along three main hierarchies (*isA* or *partOf*) respectively rooted at the classes: `WasteBin`, `City` and `Waste` (see Figure 18).

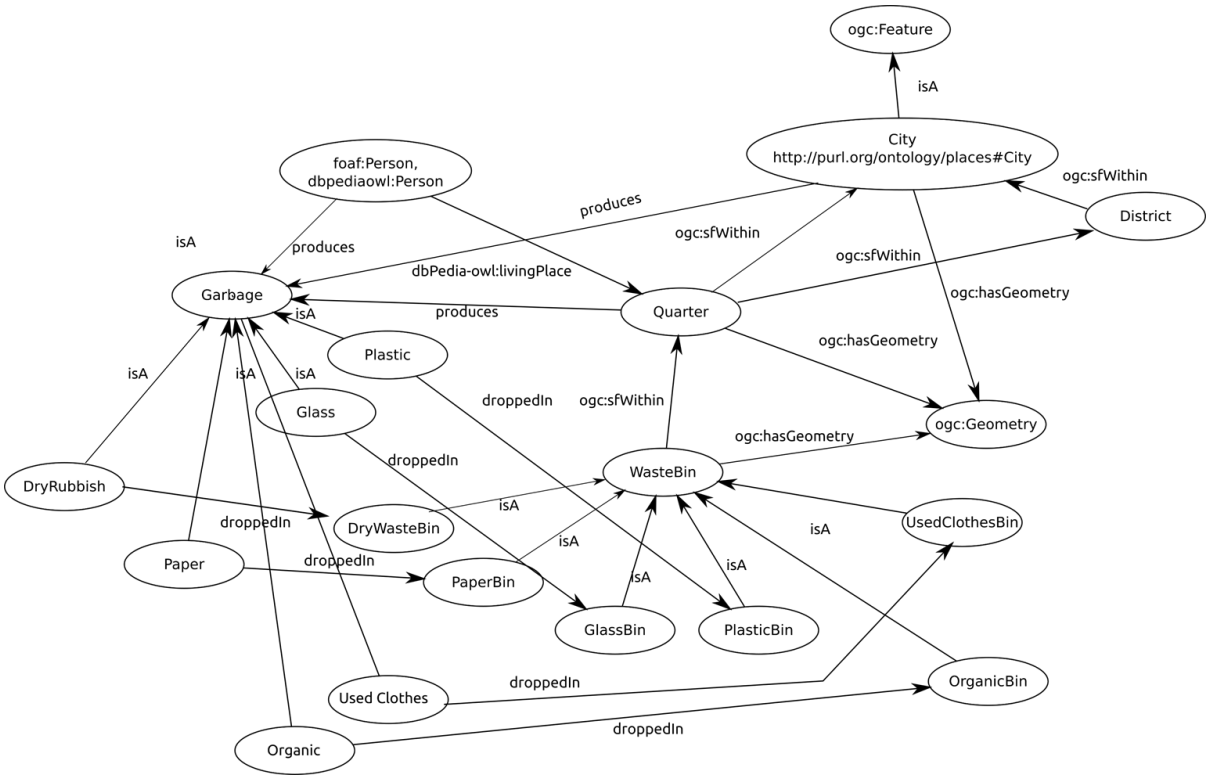


Figure 18. An overview of the Waste Bin ontology.

The former represents different bin types such as bins for gathering organic waste, glass, paper, etc. Six different types of bins are represented including: dry waste, glass and aluminum, organic, paper, plastic and used clothes bins. These types are clearly emerging from a national (Italian, as one of the ALMANAC end users is the Turin's municipality) "standpoint" on waste collection, but it can easily be extended to represent typical waste collection in Europe.

The second hierarchy of objects, is organized along a *partOf* containment tree and includes both physical (`City`) and administrative (`District` and `Quarters`) concepts. Since the main concepts represented in this tree are widely used in several knowledge domains, and application scenarios, the WasteBin ontology definition is mainly built through equivalence classes, and it only adds geo-spatial information for automatically inferring spatial containment between waste bins and administrative / physical regions.

Finally, Waste types are defined to represent the kind of "garbage" collected by a specific bin and to represent, by means of suitable relationships, the waste generation behaviour of quarters and districts.

Under a more "technical" standpoint, the ontology counts 20 concepts, 6 object properties and 3 data-type properties; it features a SHIQ(D) DL expressivity (cf. Annex 8.2) and it is interconnected

with 6 different and widely recognized vocabularies including: the geonames ontology, the places ontology, the GeoSPARQL vocabulary, the Schema.org vocabulary, the VCard and Good Relations and the Unit of Measurment ontologies (MUO[45]). A full instantiation representing the public information available on waste bins deployed in Turin is provided as initial use case, and includes around 29k waste bins, one city, 10 districts and 25 quarters.

The waste bin ontology is exploited by the SCRAL waste bin emulator to generate synthetic information about fill level and temperature of collection sites in Turin.

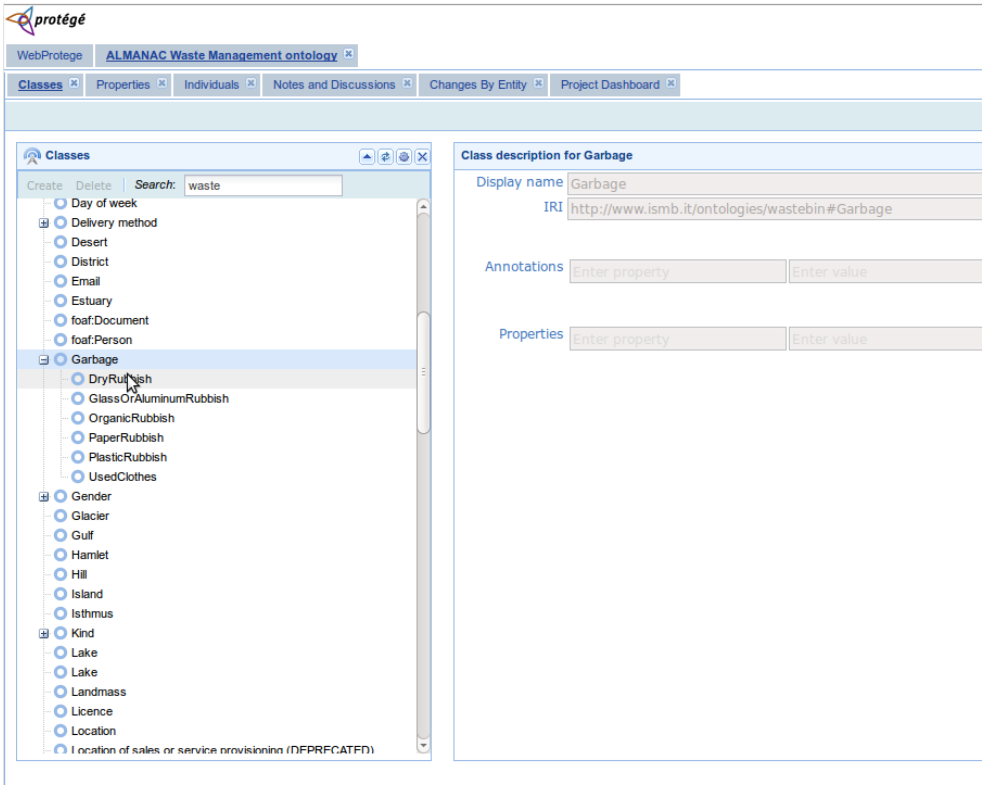


Figure 19. WasteBin ontology loaded in WebProtege

IoT DogOnt

The DogOnt ontology aims at offering a uniform, extensible model for all devices being part of a "local" Internet of Things inside a smart environment (such as smart building or smart city). Its major focus is on device modeling, for all the aspects needed to abstract device "capabilities" from low-level idiosyncrasies and communication issues. This enables both abstract reasoning on devices, e.g., to find similar devices or to identify the most suitable output to which forward urgent notifications, and actual integration of different technologies, and paradigms. DogOnt was firstly introduced in 2008 (Bonino et al 2008) and was originally meant to represent home automation devices for interoperability support. In the past years it underwent several reviews and amendments, and its scope was widened to include devices and technologies typically part of an indoor IoT network. If the original focus was more on modeling operational aspects enabling device control, the latest version, from which the IoT branch stems, has moved to a more informed, modular and linked modeling approach which enables adoption of DogOnt-based representations at different abstraction layers. While device control and interoperability is still one of the pillars of the representation, extensibility, modularity and service-based representation of IoT entities empower the latest ontology, enabling modular integration and reconciliation of different specifications, e.g., the cluster-based ZigBee Home Automation model and the registry-based Modbus data

[45] http://idi.fundacionctic.org/muo/

representation. More attention is also devoted to the Linked Open Data initiative: the ontology is now listed in the Linked Open Vocabulary data set[46] and its connections with well-known ontologies (see Figure 1) are being updated day by day.

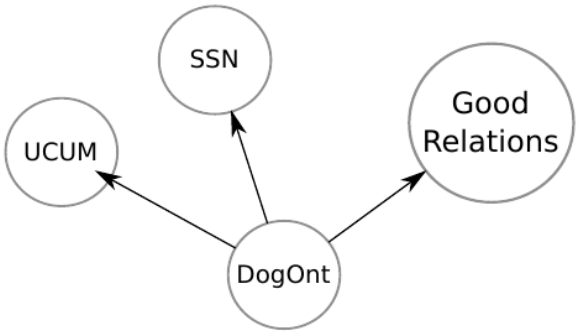


Figure 20. Links between DogOnt and other relevant domain models.

From a very high-level perspective, the IoT branch of the ontology is deployed along 3 main hierarchies of concepts rooted at Entity, Function and State (see Figure 21). These three main modelling axis are supported by complementary subtrees representing the surrounding environment, the technology specific details needed to interface IoT resources, etc.

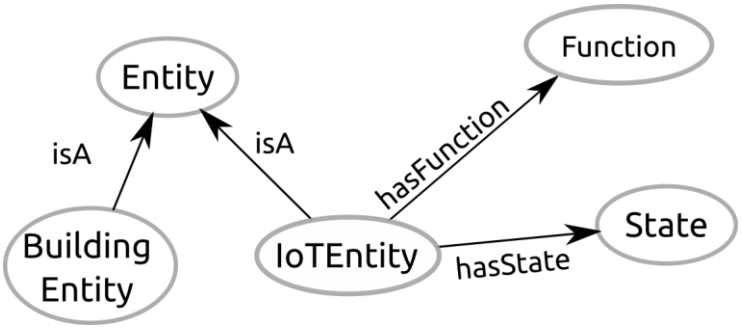


Figure 21. The root concepts of the IoT branch of DogOnt.

On the formal standpoint, The IoT branch of DogOnt exploited as starting point in the ALMANAC Smart City ontology is an OWL2 DL compliant ontology with ALCRIQ(D) expressivity (see 8.2). It counts 961 classes and 6950 axioms. The current version (4.0.0) is released under the Apache v2.0 License and is reachable https://github.com/iot-ontologies/dogont/tree/iotdogont.

IoT Entity modeling

Overall, the contemporary adoption of the WasteBin ontology and of the IoT branch of DogOnt enables an almost complete representation of entities involved in the ALMANAC waste scenario. Moreover, the approach can easily be extended to support representation of several other domains related to smart cities, including the water and citizen-centric scenarios considered in ALMANAC. For the sake of clarity, we report here a complete modelling example of a smart paper bin (Figure 22).

[46] http://lov.okfn.org/dataset/lov/

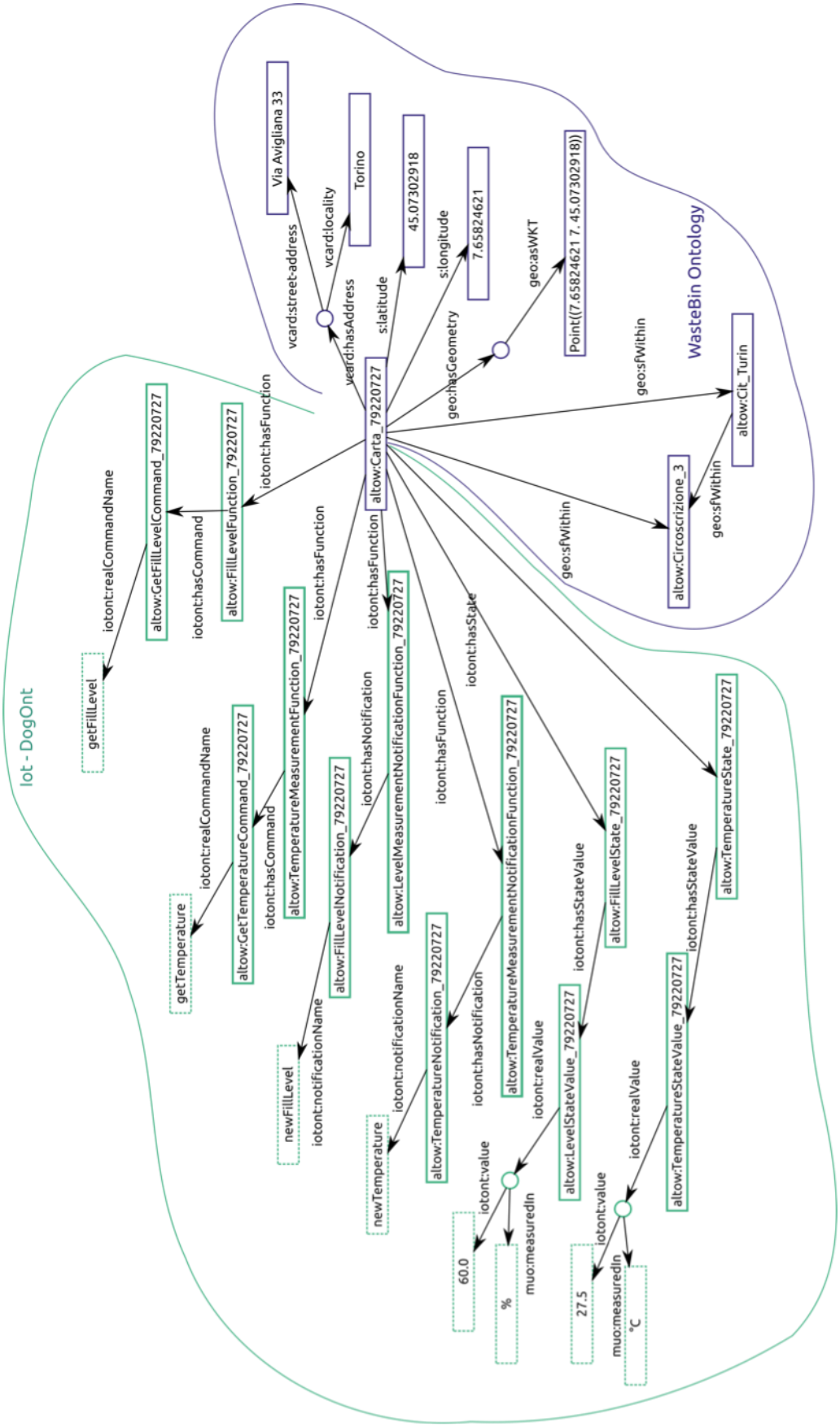


Figure 22. A complete modelling example showing how a Paper Bin is modelled in the ALMANAC Smart City ontology. Few simplifications to the actual model have been applied for the sake of clarity.

6. Conclusion

This deliverable provided an overview of the tasks and components developed in Work Package 5 at M24. Prototypes for each of the three parts described in this document are foreseen in M30. Furthermore, the functionality as integrated ALMANAC Platform is demonstrated by the Water, Waste, and Citizen-centric applications covering abstraction and virtualization of large amounts of heterogeneous devices, data, and services.

7. References

(Bonino et al 2008) DogOnt - Ontology Modeling for Intelligent Domotic Environments. In International Semantic Web Conference (LNCS), Amith Sheth, Steffen Staab, Mike Dean, Massimo Paolucci, Diana Maynard, Timothy Finin, and Krishnaprasad Thirunarayan (Eds.). Springer-Verlag, 790–803.

8. Appendix

8.1 Waste ontology

```
@prefix s: <http://schema.org#> .
@prefix geo: <http://www.opengis.net/ont/geosparql#> .
@prefix geonames: <http://www.geonames.org/ontology#> .
@prefix vcard: <http://www.w3.org/2006/vcard/ns#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix wbin: <http://www.ismb.it/ontologies/wastebin#> .
@prefix places: <http://purl.org/ontology/places#> .
@prefix gr: <http://purl.org/goodrelations/v1#> .
@prefix muo: <http://purl.oclc.org/NET/muo/muo#> .


<http://www.ismb.it/ontologies/wastebin> rdf:type owl:Ontology ;
        owl:imports <http://schema.org/> ,
        <http://www.opengis.net/ont/geosparql> ,
        <http://www.w3.org/2006/vcard/ns> ,
        <http://purl.org/ontology/places> ,
        <http://purl.org/goodrelations/v1> ,
        <http://purl.oclc.org/NET/muo/muo> .



# object properties
wbin:yearlyWasteProduction rdf:type owl:ObjectProperty ;
        owl:ObjectPropertyDomain [a owl:Class; owl:ObjectUnionOf (wbin:Quarter wbin:City
wbin:District)] ;
        rdfs:domain [a owl:Class; owl:unionOf (wbin:Quarter wbin:City)] ;
        owl:ObjectPropertyRange wbin:YearlyWasteAmount ;
        rdfs:range wbin:MonthlyWasteAmount .

wbin:monthlyWasteProduction rdf:type owl:ObjectProperty ;
        owl:ObjectPropertyDomain [a owl:Class; owl:ObjectUnionOf (wbin:Quarter wbin:City
wbin:District)] ;
        rdfs:domain [a owl:Class; owl:unionOf (wbin:Quarter wbin:City)] ;
        owl:ObjectPropertyRange wbin:MonthlyWasteAmount ;
        rdfs:range wbin:MonthlyWasteAmount .

wbin:producedAmount rdf:type owl:ObjectProperty ;
        owl:ObjectPropertyDomain [a owl:Class; owl:ObjectUnionOf (wbin:YearlyWasteAmount
wbin:MonthlyWasteAmount)] ;
        rdfs:domain [a owl:Class; owl:unionOf (wbin:YearlyWasteAmount wbin:MonthlyWasteAmount)] ;
        owl:ObjectPropertyRange wbin:WasteAmount ;
        rdfs:range wbin:WasteAmount .

wbin:type rdf:type owl:ObjectProperty ;
        owl:ObjectPropertyDomain wbin:WasteAmount;
        rdfs:domain wbin:WasteAmount;
        owl:ObjectPropertyRange wbin:Garbage ;
        rdfs:range  wbin:Garbage .

wbin:collects rdf:type owl:ObjectProperty ;
        owl:ObjectPropertDomain wbin:WasteBin;
        rdfs:domain wbin:WasteBin;
        owl:ObjectPropertyRange wbin:Garbage ;
        rdfs:range wbin:Garbage .

wbin:contributesTo rdf:type owl:Objectproperty ;
        owl:ObjectPropertDomain wbin:MonthlyWasteAmount;
        rdfs:domain wbin:MonthlyWasteAmount;
        owl:ObjectPropertyRange wbin:YearlyWasteAmount ;
        rdfs:range wbin:YearlyWasteAmount .



# data properties
```

```
wbin:year rdf:type owl:DataTypeProperty ;
        owl:DataTypePropertyDomain wbin:WasteAmount;
        owl:DataTypePropertyRange xsd:gYear .

wbin:monthOfYear rdf:type owl:DataTypeProperty ;
        owl:DataTypePropertyDomain wbin:MonthlyWasteAmount;
        owl:DataTypePropertyRange xsd:gYearMonth .

wbin:amount rdf:type owl:DataTypeProperty ;
        owl:DataTypePropertyDomain wbin:WasteAmount;
        owl:DataTypePropertyRange xsd:gYearMonth .


#  classes

# city
wbin:City rdf:type owl:Class ;
        rdfs:subClassOf geo:Feature ;
         owl:equivalentClass s:City , places:City .

# district
wbin:District rdf:type owl:Class ;
        rdfs:subClassOf geo:Feature ;
        geo:sfWithin wbin:City .

# quarter
wbin:Quarter rdf:type owl:Class ;
        rdfs:subClassOf s:Place , geo:Feature ;
        geo:sfWithin wbin:District ;
         geo:sfWithin wbin:City .

#----------------------------------------
#        WASTE BINS
#----------------------------------------

# waste bin
wbin:WasteBin rdf:type owl:Class ;
        rdfs:subClassOf geo:Feature ;
        geo:sfWithin wbin:District ;
        geo:sfWithin wbin:Quarter .

# Dry waste bin storing undifferentiated rubbish
wbin:DryWasteBin rdf:type owl:Class ;
        rdfs:subClassOf wbin:WasteBin ;
        rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects; owl:ObjectAllValuesFrom
wbin:DryRubbish].

# Glass and Aluminum trash bin
wbin:GlassBin rdf:type owl:Class ;
        rdfs:subClassOf wbin:WasteBin ;
        rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects; owl:ObjectAllValuesFrom
wbin:GlassOrAluminumRubbish].

# Organic trash bin
wbin:OrganicBin rdf:type owl:Class ;
        rdfs:subClassOf wbin:WasteBin ;
        rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects; owl:ObjectAllValuesFrom
wbin:GlassOrAluminumRubbish].

# Paper trash bin
wbin:PaperBin rdf:type owl:Class ;
        rdfs:subClassOf wbin:WasteBin ;
        rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects; owl:ObjectAllValuesFrom
wbin:OrganicRubbish].

# Plastic trash bin
wbin:PlasticBin rdf:type owl:Class ;
        rdfs:subClassOf wbin:WasteBin ;
        rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects; owl:ObjectAllValuesFrom
wbin:PlasticRubbish].

# Used Clothes bin
wbin:UsedClothesBin rdf:type owl:Class ;
        rdfs:subClassOf wbin:WasteBin ;
        rdfs:subClassOf [ a owl:Restriction; owl:onProperty wbin:collects; owl:ObjectAllValuesFrom
wbin:UsedClothes].
```

```
#----------------------------------------
#        WASTE / GARBAGE
#----------------------------------------

# general Waste root,
wbin:Garbage rdf:type owl:Class .

# Plastic
wbin:PlasticRubbish rdf:type owl:Class ;
       rdfs:subClassOf wbin:Garbage .

# Glass or Aluminum
wbin:GlassOrAluminumRubbish rdf:type owl:Class ;
       rdfs:subClassOf wbin:Garbage .

# Paper
wbin:PaperRubbish rdf:type owl:Class ;
       rdfs:subClassOf wbin:Garbage .

# Dry rubbish
wbin:DryRubbish rdf:type owl:Class ;
       rdfs:subClassOf wbin:Garbage .

# Organic
wbin:OrganicRubbish rdf:type owl:Class ;
       rdfs:subClassOf wbin:Garbage .

# Used Clothes
wbin:UsedClothes rdf:type owl:Class ;
       rdfs:subClassOf wbin:Garbage .

#----------------------------------------
#       PRODUCTION RATES
#----------------------------------------

# waste amount
wbin:WasteAmount rdf:type owl:Class ;
       rdfs:subClassOf gr:QuantitativeValue ;
       rdfs:subClassOf muo:QualityValue .

#yearly waste amount
wbin:YearlyWasteMount rdf:type owl:Class .

#monthly waste amount
wbin:MonthlyWasteMount rdf:type owl:Class .
```

8.2 Ontology DL expressivity

Ontology expressivity is described by means of a specific notation describing the peculiarity of the Description Logic dialect needed to describe concepts expressed in the analysed model. As an example, the IoT-branch of DogOnt exploited as one of the pillars of the ALMANAC SmartCity ontology is described as having an ALCRIQ(D) epressivity. This means:

AL - Attributive language. This is the base language which allows:

- Atomic negation (negation of concept names that do not appear on the left hand side of axioms)
- Concept intersection
- Universal restrictions
- Limited existential quantification

C - Complex concept negation;

R - Limited complex role inclusion axioms; reflexivity and irreflexivity; role disjointness;

I - Inverse properties;

Q - Qualified cardinality restrictions (available in OWL 2, cardinality restrictions that have fillers other than T);

(D) - Use of datatype properties, data values or data types.