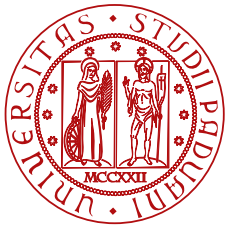


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Computer Engineering

Open Data for Italian Municipalities: Ontology, Data and WebApps

Master Candidate

Luca Martinelli

Student ID 2005837

Supervisor

Prof. Gianmaria Silvello

University of Padova

Academic Year
2021/2022

To my parents
and friends

Abstract

Sommario

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

AgID Agency for Digital Italy

API Application Programming Interface

ASCII American Standard Code for Information Interchange

CSV Comma Separated Values

DBMS DataBase Management System

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IRI International Resource Identifier

JSON JavaScript Object Notation

LOD Linked Open Data

OntoIM Ontology for Italian Municipalities

OWL Web Ontology Language

RDF Resource Description Framework

RDFa RDF in Attributes

RDFS RDF Schema

SPARQL SPARQL Protocol and RDF Query Language

SQL Structured Query Language

TSV Tab Separated Values

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

XML eXtensible Markup Language

1 Introduction

1.1 The OntoIM Ontology

Ontology for Italian Municipalities (OntoIM)

1.2 Scope and organization of the thesis

2 Background

2.1 The Web of Data

The World Wide Web was originally designed to be a space where documents are connected by links without semantic value, and most of these documents are designed for humans to read, not for machines to process. For this reason, Tim Berners-Lee in 2001 introduced the idea of the Semantic Web. In particular, the Semantic Web is an enhancement of the current Web that aims to create a web of data, in which information has a well-defined meaning and can be easily read and processed by programs [BHL01].

Due to this machine-comprehensible capacity, the Semantic Web has enormous potential to automate daily tasks in our lives and is helping to advance scientific and health care fields [Fei+07], such as drug discovery and clinical research, but also in the automotive industry, in the enhancement of cultural heritage, etc.[1] In this context, ontologies play a fundamental role in supporting interoperability and common understanding between different web applications and services, solving the problem of semantic heterogeneity [Tay10].

Although there are different definitions of "ontology" [Tay10], in computer science, an ontology is defined as an "explicit and formal specification of a shared conceptualization" [Gru95], where conceptualization means a simplified view of

[1] https://www.w3.org/2001/sw/sweo/public/UseCases/

the world we wish to represent. An ontology is made up of four main types of components, which are (1) *classes* (or *concepts*), which describe concepts in the domain; (2) *instances* of classes, which represent specific objects or elements of a class; (3) *properties* (or *slots*), which are used to express relationships between a first concept in the domain and a second concept in the range; (4) *axioms* (or *role restrictions*), which are used to impose constraints on the values of instances and classes [Tay10; NM+01]. In addition to the interoperability problem, ontologies are also used to satisfy the following needs:

- To share common understanding of the structure of information among people or software agents;
- To enable reuse of domain knowledge;
- To make domain assumptions explicit;
- To separate domain knowledge from the operational knowledge
- To analyze domain knowledge [NM+01].

Along with ontologies, controlled vocabularies, taxonomies, and thesauri are other resources used in different domains, in particular the medical one [IB14]. A controlled vocabulary is a closed list of named subjects, called *terms*, which is usually used for classification. A taxonomy is a subject-based classification that organizes terms in a controlled vocabulary into a hierarchy. Finally, a thesaurus extends taxonomies, allowing making other statements about the subjects and providing a much richer vocabulary [IB14].

In 2006, Tim Berners-Lee used for the first time the term Linked Data to describe the structured and interlinked data that populate the Semantic Web [Ber06]. He also introduced a set of rules, also known as the Linked Data Principles, to provide some best practices for publishing and connecting data on the Web [BHB11]. These principles, published by W3C, are the following:

1. Use URIs as names for things;
2. Use HTTP URIs so that people can look up those names;
3. When someone looks up URIs, provide useful information, using standards such as RDF and SPARQL;
4. Include links to other URIs so that they can discover more things.

The two main fundamental technologies for Linked Data are Uniform Resource Identifiers (URIs) and HyperText Transfer Protocol (HTTP). In particular, URIs are used to identify any entity that exists in the world, while HTTP provides a simple and universal mechanism for retrieving the resources to which they refer. These two technologies are integrated in RDF, which provides a graph-based data model to structure and link data that describe entities in the world [BHB11]. Using HTTPs, URIs, and RDF, Linked Data builds on the architecture of the Web, called the Web of Data. This means that the Web of Data shares many properties with the traditional Web, which are:

- Web of Data can contain any type of data;
- Anyone can publish data on the Web of Data;
- Publishers are not restricted in the choice of vocabularies used to represent the data;
- Entities are connected by RDF links [BHB11].

However, in addition to those of the traditional Web, the Web of Data also has the following characteristics:

- Data are separated from formatting and presentational aspects;
- Data is self-describing;
- Data access is simplified by the use of the HTTP and RDF standards;
- Web of Data is open, and new data sources can be discovered at run-time by following RDF links [BHB11].

Semantic Web and Linked Data are empowered by technologies developed by the World Wide Web Consortium such as RDF, OWL, serialization formats (Section 2.3), and SPARQL (Section 2.4).[2]

2.2 The five stars of Open Data

Linked Data does not have to be open and can be used internally, such as for personal data. When Linked Data is released under an open license that does not

[2] https://www.w3.org/2001/sw/wiki/Main_Page

impede its reuse for free, such as Creative Commons CC-BY[3] or the Italian Open Data License[4], we can use the term Linked Open Data (LOD) [Ber06]. In 2010 Tim Berners-Lee developed a star rating system to define and classify Linked Open Data, "in order to encourage people, especially government data owners, along the road to good linked data" [Ber06]. The star rating system assigns a star if the information is publicly available under an open license, even if the information is a photo or an image scan of a table. The more stars the information gets, the easier it will be for people (and machines) to use it [Ber06].

★ Available on the Web (any format) but with an open license to be Open Data

★★ Available as machine-readable structured data (e.g., Excel instead of an image scan of a table)

★★★ Available in a non-proprietary format (e.g., CSV instead of Excel)

★★★★ Use URIs to identify things, so that people can point at your stuff

★★★★★ Data are linked to other people's data to provide context

However, as the information receives a greater number of stars, both the benefits for consumers and the costs for the publisher increase. In particular, a five-stars data let consumers discover new data of interest, access to the data schema, reuse parts of the data, and link it to other places. They also do not have to pay for tools in order to read the data (e.g., Excel), and they can download and export the data into other formats and process them. On the other hand, to make these data available, publishers must invest time and resources in slicing and organizing the data, assigning URIs to the data items, thinking about how to represent them, linking the data with other data on the Web and making them discoverable [BK11].

2.3 RDF, OWL, AND SERIALIZATION FORMATS

The Resource Description Framework (RDF) is a W3C graph-based standard model to represent information about resources on the Web (including documents,

[3] https://creativecommons.org/
[4] https://www.dati.gov.it/content/italian-open-data-license-v20

people, physical objects, and abstract concepts). Using RDF, machines can process information on the Web using common parsers and processing tools, and information can be exchanged between different applications without losing meaning [Con+14b]. In particular, in recent years RDF has become the *de-facto* standard for publishing Linked Data on the Web. The core structure of the RDF syntax is a set of statements, called *triples*, because they consist of three elements: a *subject*, a *predicate*, and an *object*, following the structure `<subject> <predicate> <object>`, which can be visually represented in Figure 2.1 [Con+14a].

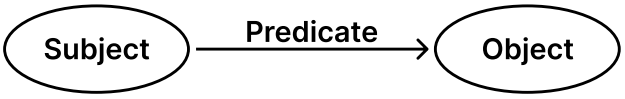


Figure 2.1: The structure of a triple, with two nodes and a predicate connecting them.

The subject and the object represent the two resources being related. The relationship that goes from the subject to the object is called *property*, and its nature is represented by the predicate. A set of statements generate a direct graph, called RDF Graph, where subjects and objects are the nodes of the graph, and the predicates form the arcs. For example, the set of triples below produces the graph shown in Figure 2.2 [Con+14b].

```
<Bob> <is a> <person>.
<Bob> <is a friend of> <Alice>.
<Bob> <is born on> <the 4th of July 1990>.
<Bob> <is interested in> <the Mona Lisa>.
<the Mona Lisa> <was created by> <Leonardo da Vinci>.
<the video 'La Joconde à Washington'> <is about> <the Mona Lisa>
```

In an RDF Graph, resources may be represented using an International Resource Identifier (IRI), a *literal value* or a *blank node*. An IRI is a generalization of URI, where non-ASCII characters are allowed in the IRI character string. IRIs identify resources, and can appear in all three positions of a triple. In the example above, the IRI for Leonardo Da Vinci in DBpedia[5] is `http://dbpedia.org/resource/Leonardo_da_Vinci`.

[5] `https://www.dbpedia.org/`

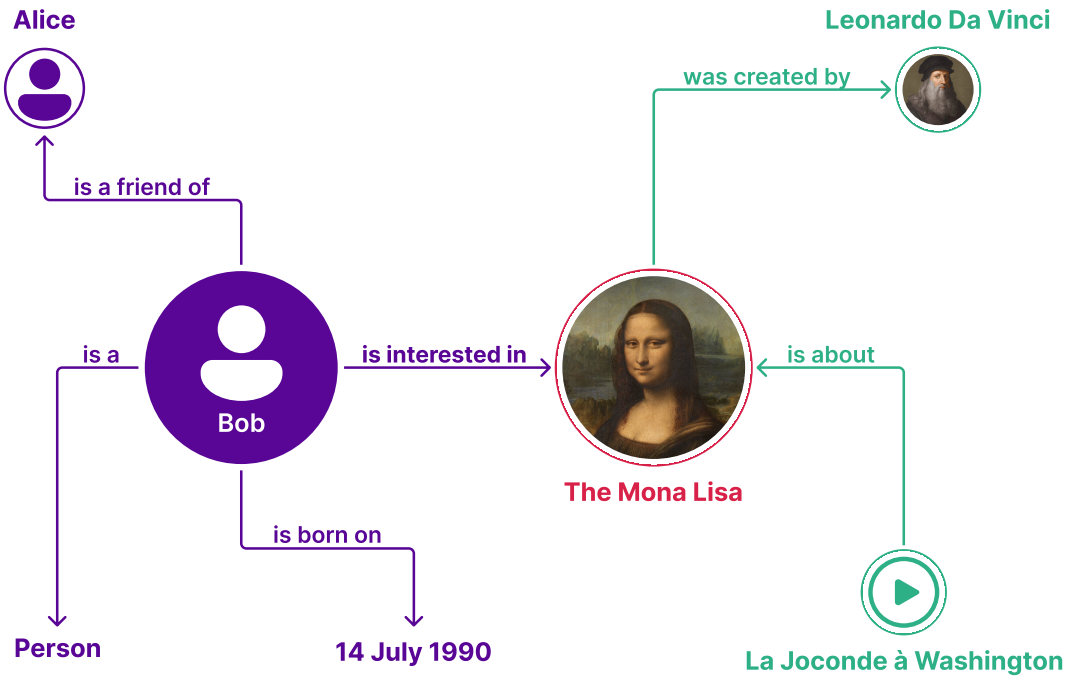


Figure 2.2: The example of the RDF Graph presented by W3C.

Literals are basic values such as strings, dates, and numbers. In the RDF Graph literals can only be used as objects, and consists of two or three elements, which are: (1) the value itself; (2) an IRI that identifies the *datatype* (string, number, date, etc); (3) if and only if the datatype is a `rdf:langString`,[6] a *language tag* (such as `en`, `it`, `fr`, etc) [Con+14a].

Finally, blank nodes can appear in the subject and object position of a triple and are used to represent resources without using a IRI [Con+14b].

In Section 2.1 ontologies and vocabularies are presented as a core element for creating the Semantic Web. The RDF data model does not provide semantic information about the resources. For this reason, RDF provides the RDF Schema (RDFS) language, that allows to define semantic characteristics of data. RDF Schema uses the notion of *class* to classify resources, while uses the *type* property to define a relation between an instance and its class. RDF Schema also allows defining type restrictions on subject and objects of particular triples through *domain* and *range* restrictions. Finally, with RDF Schema it is also possible to define hierarchies of classes and properties, using *subClassOf* and *subPropertyOf*

[6] `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`

predicates [Con+14b]. All of these modeling constructs provided by RDF Schema are summarized in Table 2.1.

Construct	Syntactic form	Description
Class	C rdf:type rdfs:Class	C is an RDF class
Property	P rdf:type rdf:Property	P is an RDF property
type	I rdf:type C	I is an instance of C
subClassOf	C1 rdfs:subClassOf C2	C1 is a subclass of C2
subPropertyOf	P1 rdfs:subPropertyOf P2	P1 is a sub-property of P2
domain	P rdfs:domain C	domain of P is C
range	P rdfs:range C	range of P is C

Table 2.1: The main modeling constructs provided by RDF Schema.

However, in 2004 the World Wide Web Consortium presented Web Ontology Language (OWL), a more complete language for publishing and sharing ontologies on the Web [Bec+04], and replaced in 2009 and then in 2012 by OWL 2. OWL 2 is a Semantic Web language to represent rich and complex knowledge about things, groups of things, and relations between things. In addition, since OWL is a computational logic-based language, the knowledge expressed in OWL can be reasoned with by computer programs either to verify the consistency of that knowledge or to make implicit knowledge explicit. A OWL document, called *ontology*, can be published in the World Wide Web and may refer to or be referred from other OWL ontologies [Hit+09]. In OWL 2 knowledge is represented by statements, called *axioms*. Axioms normally refer to objects of the world and describe them by putting them into categories or saying something about their relation. In OWL 2 objects, categories and relations are called *entities*, and in particular objects are denoted as *individuals*, categories as *classes* and relations as *properties*. Moreover, properties are further subdivided into (1) *object properties* that relate objects to objects; (2) *datatype properties* that assign data values to objects; (3) *annotation properties* that encode information about the ontology itself. Finally, names of entities can be combined into *expressions* using *constructors* to form complex descriptions from basic ones [Hit+09].

In order to publish RDF data on the Web, the RDF Graphs need to be serialized. Today there are several serialization formats, but the most famous one are: N-Triples, Turtle, RDF/XML, RDFa, and JSON-LD. These formats are briefly described below, reporting as example of small excerpt of DBpedia[7] is reported.

N-Triples[8] It's one of the simplest formats, formed by sequences of RDF triples. Each statement is formed by the subject, predicate, object, and a ".", that are separated by white space.

```
<http://dbpedia.org/page/Jotaro_Kujo>
  <http://dbpedia.org/ontology/relative>
    <http://dbpedia.org/page/Joseph_Joestar> .
```

Turtle[9] It's a common data format for serializing RDF Graphs that introduces some features to N-Triples language. In particular, it introduces the use of `@base` IRI and relative IRIs, `@prefix` and prefixed names, predicate lists separated by ";", object lists separated by ",", and the representation of `rdfs:type` with the token `a`.

```
@prefix dbr: <http://dbpedia.org/page/> .
@prefix dbo: <http://dbpedia.org/ontology/> .

dbr:Jotaro_Kujo dbo:relative dbr:Joseph_Joestar .
```

RDF/XML[10] Expresses RDF Graphs as an XML document. The nodes and predicates are represented in XML terms: element names, attribute names, element contents and attribute values.

```
<rdf:RDF xmlns:dbr="http://dbpedia.org/page/"
  xmlns:dbo="http://dbpedia.org/ontology/"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xml:base="http://www.ldf.fi/service/rdf-serializer/">
  <rdf:Description
    rdf:about="http://dbpedia.org/page/Jotaro_Kujo">
```

[7] https://www.dbpedia.org/
[8] https://www.w3.org/TR/n-triples/
[9] https://www.w3.org/TR/turtle/
[10] https://www.w3.org/TR/rdf-syntax-grammar/

```
    <dbo:relative
      rdf:resource="http://dbpedia.org/page/Joseph_Joestar"/>
  </rdf:Description>
</rdf:RDF>
```

RDF in Attributes (RDFa)[11] Provides a set of markup attributes to HTML pages to augment the visual information on the Web with machine-readable hints.

```
<body
  prefix="dbr: http://dbpedia.org/page/
    dbo: http://dbpedia.org/ontology/">
  <div about="dbr:Jotaro_Kujo">
    <div
      rel="dbo:relative"
      resource="dbr:Joseph_Joestar">
    </div>
  </div>
</body>
```

JSON-LD[12] Serializes RDF Graphs into JavaScript Object Notation (JSON). The syntax is designed to easily integrate into deployed systems that already use JSON. It's intended to be a way to use Lined Data in Web-based programming environments, to build interoperable Web services, and to store Linked Data in JSON-based storage engines.

```
[
  {
    "@id": "http://dbpedia.org/page/Joseph_Joestar"
  },
  {
    "@id": "http://dbpedia.org/page/Jotaro_Kujo",
    "http://dbpedia.org/ontology/relative": [
      {
        "@id": "http://dbpedia.org/page/Joseph_Joestar"
```

[11] https://www.w3.org/TR/rdfa-primer/
[12] https://www.w3.org/TR/json-ld/

```
      }
    ]
  }
]
```

2.4 SPARQL

SPARQL is a query language developed by W3C retrieve and manipulate RDF Graph content on the Web or in an RDF store. A SPARQL query contains a set of triple patterns called *basic graph pattern*. These patterns are like RDF triples except that subjects, predicates and objects may be replaced by variables. The basic graph pattern matches a subgraph of the RDF data and returns a new RDF Graph in which the variables are replaced with the matched data. Queries are usually processed by an HTTP service, called *SPARQL endpoint*. [Con+13]. The example below shows a SPARQL query on DBpedia SPARQL endpoint,[13] while Table 2.2 shows its result.

```
PREFIX dbr: <http://dbpedia.org/page/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/name/>

SELECT ?relative ?name WHERE {
    dbr:Jotaro_Kujo dbo:relative ?relative .
    ?relative dbp:name ?name .
}
```

relative	name
dbr:Dio_Brando	"Dio Brando"@en
dbr:Joseph_Joestar	"Joseph Joestar"@en
dbr:Jonathan_Joestar	"Jonathan Joestar"@en

Table 2.2: A query result example from DBpedia.

SPARQL queries supports features like union of patterns, nesting queries,

[13] https://dbpedia.org/sparql

optional patterns or filtering values. Once the RDF subgraph is computed, it's also possible to modify it by ordering, limiting and grouping the values.

Another important feature of SPARQL is the possibility to perform federated queries, which explicitly delegates certain subqueries to different SPARQL endpoints, allowing to navigate through the Web of Data.

Finally, to return a more machine-readable form, SPARQL supports four common exchange formats, which are: eXtensible Markup Language (XML), JSON, Comma Separated Values (CSV), and Tab Separated Values (TSV) [Con+13].

2.5 Protégé

Protégé[14] is the most popular free and open source[15] ontology development environment.[16] The first version was developed by Mark Musen in 1987, and has been so far by a team at Stanford University [Gen+03]. The latest version, 5.5.0, has been released in March 2019, and it is written in Java, making it a cross-platform tool. In recent years, in addition to the desktop version, a web version, called WebProtégé[17] is also being developed, focused on collaborative viewing and editing.

Protégé supports creation and editing of one or more ontologies, providing a customizable graphic user interface. Among the several features available in Protégé, the most relevant are the possibility to create, rename and delete entities, add notations, merge ontologies, and more. It also includes a visualization tool for interactive navigation of ontology relationships and different reasoners.

The two most important sections for creating and editing an ontology are the "Active ontology" and "Entities" tabs. The first, that is opened by default, is designed to view and edit the information of the ontology, such as its IRI, its annotations and the imported ontologies. On the right there is also a panel that reports some metrics about the ontology, such as the total number of axioms, classes, properties, and more. Figure 2.3 shows an example of the "Active ontology" tab.

The "Entities" tab is the most important section for creating an ontology. Indeed, in this tab it is possible to manage the classes, the properties (object proper-

[14] https://protege.stanford.edu/
[15] https://github.com/protegeproject/protege
[16] https://protege.stanford.edu/shortcourse/
[17] https://webprotege.stanford.edu/

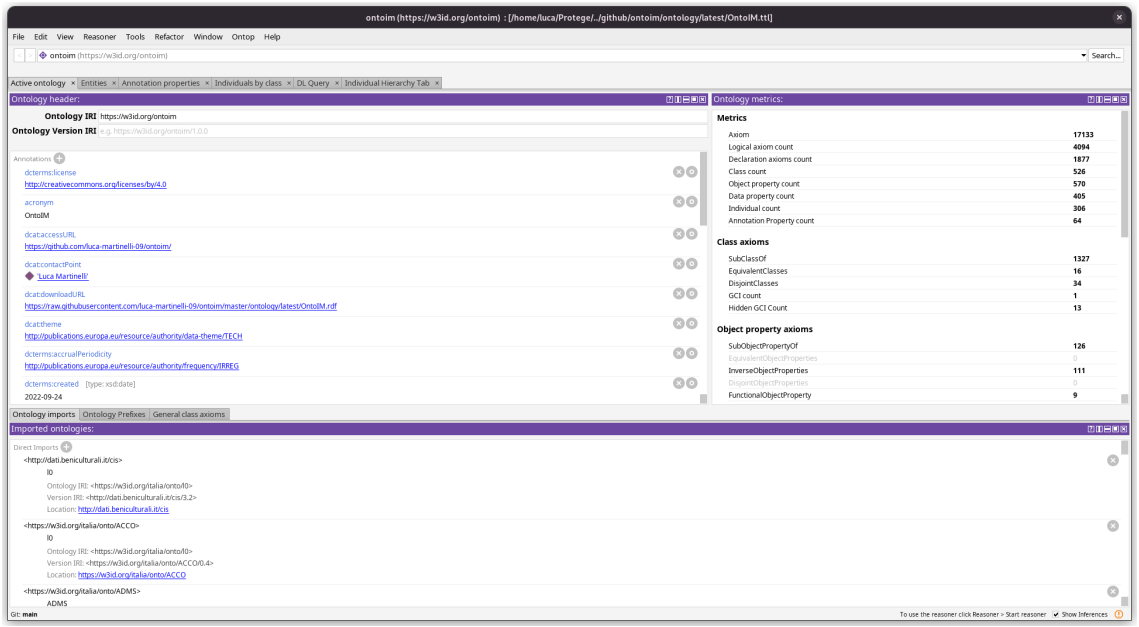


Figure 2.3: A snapshot of the Protégé "Active ontology" tab.

ties, data properties, and annotation properties), datatype and individuals. The left part provides a navigation tool to select, add and deletes entities, while the right part is focused on viewing and editing the selected entity by adding properties and axioms. Figure 2.4 shows an example of the "Entities" tab.

Of course, these were only the most relevant tools of Protégé, whose full documentation is available at `http://protegeproject.github.io/protege/`.

2.6 Virtuoso

Virtuoso Universal Server,[18] often called just Virtuoso, or OpenLink Virtuoso, at core is a high-performance object-relational SQL database. It was born in 1998 when OpenLink Software wanted to merge in a single solution its Universal Data Access Middleware and Kubl DBMS.[19]

Besides the database, Virtuoso has a built-in web server with support to Virtuoso's Web Language (VSP), and the most popular scripting languages such as PHP or ASP.NET. This same web server provides SOAP and REST access to Virtuoso stored procedures, supporting a broad set of WS* protocols. Virtuoso has also a

[18] `https://virtuoso.openlinksw.com/`
[19] `https://vos.openlinksw.com/owiki/wiki/VOS/VOSHistory`

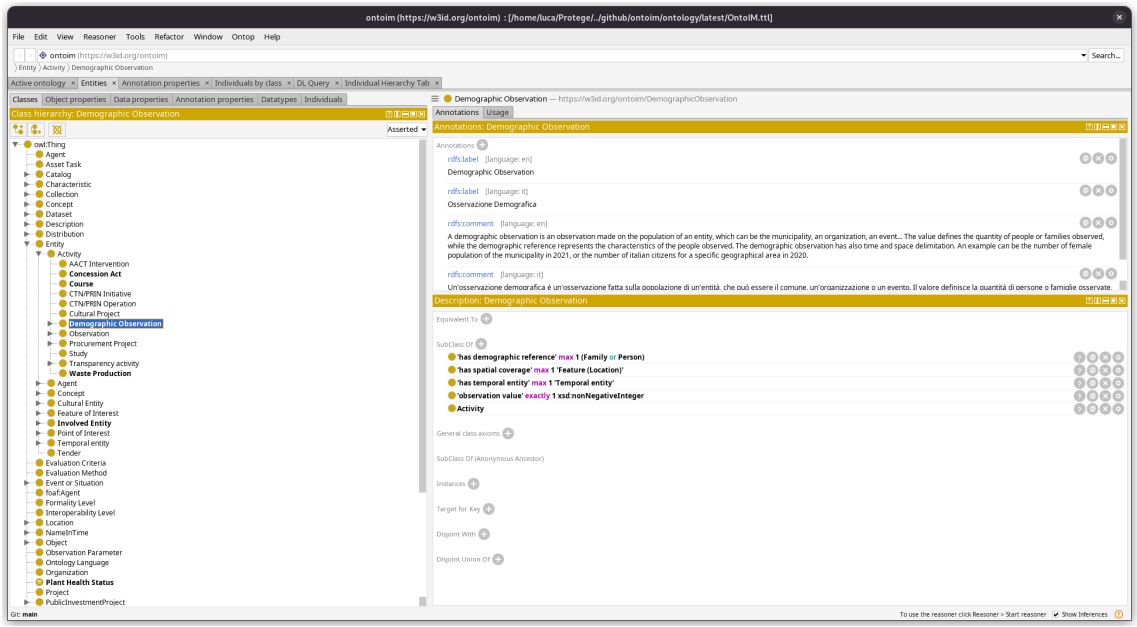


Figure 2.4: A snapshot of the Protégé "Entities" tab.

built-in WebDAV repository to host static and dynamic web content and provide versioning, making it a convenient and secure place for keeping files on the net.[20]

Since 2005, Virtuoso supports SPARQL for querying RDF data stored in its Quad Store database. In particular, it supports the HTTP-based SPARQL Protocol, SPARQL federated queries, different exchange formats such as HTML, CSV, TSV, JSON, RDF/XML, Turtle, N-Triples, and more. For this reasons Virtuoso has become the most popular and efficient tool for serving a SPARQL endpoint, which is usually located at `http://{host}/sparql`. Figure 2.5 shows an example of how the endpoint looks like.

All the aspects of a Virtuoso instance can be managed through the Virtuoso Conductor, that is located at `http://{host}/conductor`. For example, from "Linked Data" tab it is possible to add and remove RDF Graphs, import schemas, declare persistent namespaces, generate statistics such as the number of classes, triples, subjects, etc.

There are many methods to insert an RDF resource into the Virtuoso Quad Store. Some of them are:

Virtuoso Conductor Using Virtuoso Conductor web interface, under "Linked Data" and then "Quad Store Upload" tab it is possible to upload a RDF

[20] `https://vos.openlinksw.com/owiki/wiki/VOS/VOSIntro`

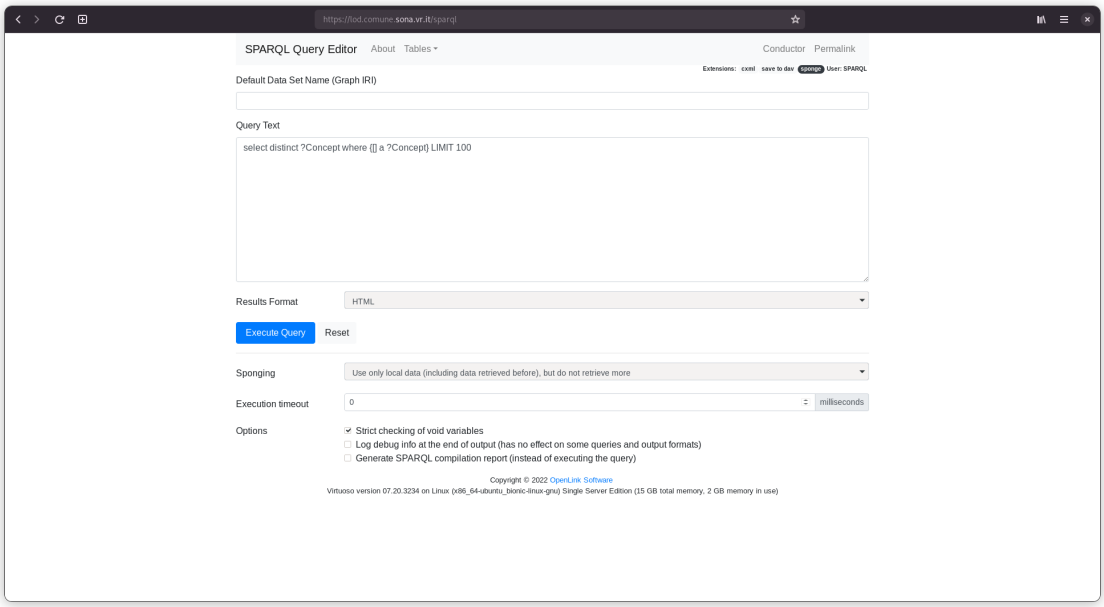


Figure 2.5: A snapshot of the Virtuoso SPARQL endpoint.

resource directly into the Virtuoso Quad Store. It is also possible to assign a graph IRI where to upload the resource. A snapshot of this feature is shown in Figure 2.6.

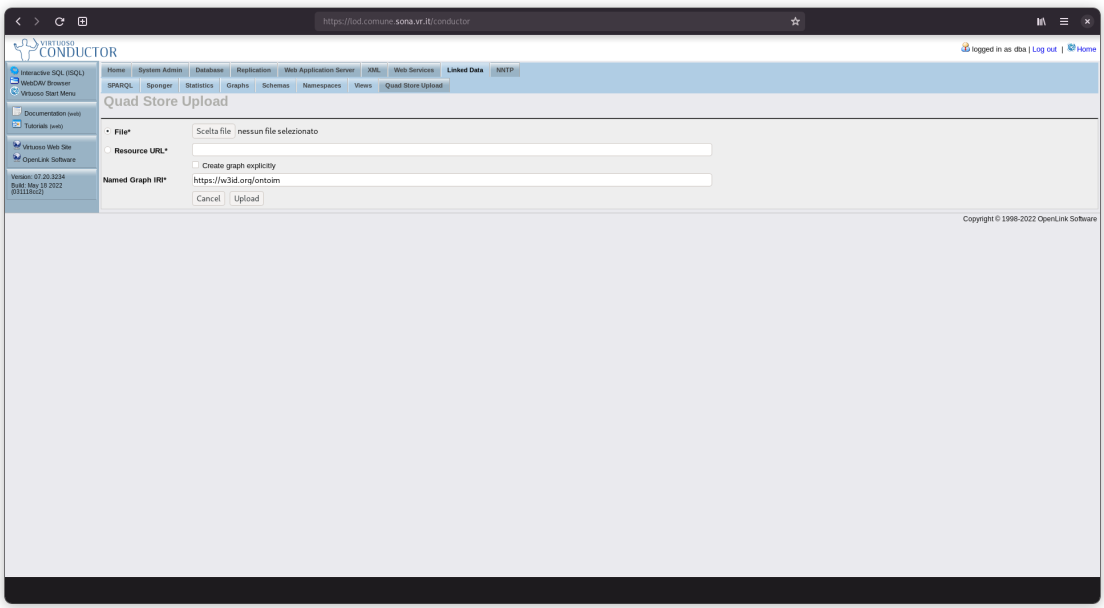


Figure 2.6: A snapshot of the "Quad Store Upload" tab.

RDF Sink Folder WebDAV supports a special folder called `rdf_sink`. This folder

can be used to upload RDF files from any WebDAV client, which are automatically uploaded to the Virtuoso Quad Store.

HTTP PUT RDF files can be uploaded to a `rdf_sink` folder through the HTTP PUT method. Using cURL, an example is:

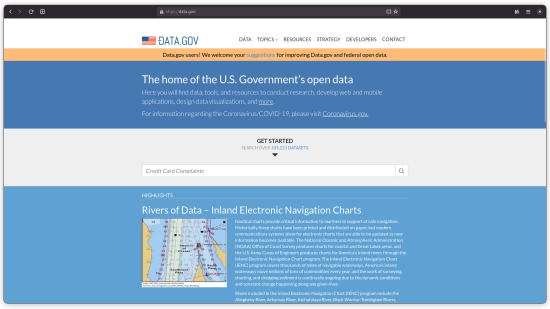
```
curl -T foaf.rdf
  http://localhost:8890/DAV/home/dba/rdf_sink/foaf.rdf
  -u dba:dba
```

HTTP POST Virtuoso supports HTTP POST method to execute SPARQL/Update language using `Content-Type: application/sparql-query` in the HTTP request headers. Using cURL, an example is:

```
curl -i -d "INSERT {
  <http://w3id.org/people/lucamartinelli>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://xmlns.com/foaf/0.1/User>
}" -u "dba:dba"
-H "Content-Type: application/sparql-query"
http://localhost:8890/DAV/home/xx/yy
```

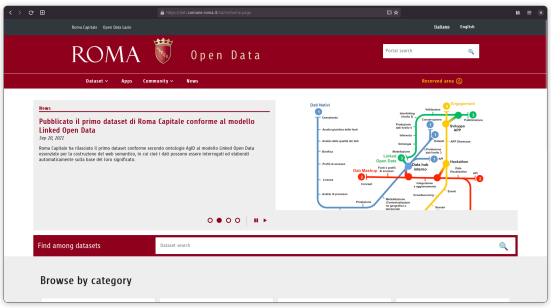
SPARQL endpoint If the user has the permission to insert graphs directly from the SPARQL endpoint, using the SPARQL/Update language, as in the example above.

These were just some features provided by Virtuoso Universal Server in order to use it as a SPARQL endpoint and RDF data store system. The full documentation on how to use Virtuoso is available at `https://vos.openlinksw.com/owiki/wiki/VOS`.



(a) U.S. Government's Open Data portal.[a]

[a] https://data.gov/



(b) Roma Capitale's Open Data portal.[a]

[a] https://dati.comune.roma.it/

Figure 2.7: Examples of CKAN Open Data governments portals.

2.7 CKAN

The Comprehensive Knowledge Archive Network, or CKAN[21], is an open source[22] data management system. In particular, CKAN is the world's leading tool for making Open Data websites, helping to manage and publish collections of data. It is mainly used by national and local governments, research institutions and other organizations who collect data. Two examples are the U.S. Government's Open Data portal, shown in Figure 2.7a or the Roma Capitale's Open Data portal, shown in Figure 2.7b.

In CKAN data is published in units called *datasets*. Each dataset is owned by an *organization* and contains information about the data like the title, publisher, data or the license; and one or more resources which are the data itself. For example, a dataset can contain different files, like the data for different years, or the same data in different formats. Any user can view, download, and search for public datasets, but there is also the possibility to restrict the access of some datasets only for registered and authorized users.

Despite the core version of CKAN has only few basic features, one of the strengths of this tool is the possibility to add different plugins which extend its functionalities and customize the user interface. The most popular plugins, developed and maintained by CKAN itself, are: (1) different tools to visualize data directly on the web page, such as tables, plots or maps; (2) DataStore extension

[21] https://ckan.org/
[22] https://github.com/ckan/ckan

that provides an *ad hoc* database for storage of structured data from resources and integrates them into CKAN API to return data in JSON format; (3) DCAT extension that includes RDF serialization of datasets and harvesters to import RDF resources into CKAN. An example of this feature can be seen in the Italian Open Data portal[23], that include the datasets from all the local governments.

2.8 OntoPiA

The only ontology OntoIM imports is OntoPiA. OntoPiA[24] is a network of ontologies and controlled vocabularies developed in 2017 by the Agency for Digital Italy (AgID)[25] and the Italian Digital Transformation Team[26] with the collaboration of research entities (CNR) and other Italian public administrations (ISTAT, Agenzia delle Entrate, Ministero della Cultura, etc). OntoPiA aims to facilitate the process of data exchange between public administrations, standardize government data, and create the knowledge graph of Italian Public Administration [Dig17b; Dig17a]. Actually the network is composed by 28 ontologies and 39 controlled vocabularies. The OntoPiA ontological stack, shown in Figure 2.8, consists of the following levels:

Foundation Level It's composed by the top-level ontology *L0*, which allows all the ontologies to be linked, enabling the network of ontologies. This ontology defines a few general concepts, such as *Entity*, *Location*, *Activity*, etc, which are used by the ontologies of the upper levels;

Core Level It comprehends the core ontologies, which describes concepts used by different datasets. In particular, the core level describes people, organizations, and locations;

Supporting Level The third level is composed by supporting ontologies, which describe concepts used in the other ontologies. These concepts are: time, roles, measurement units, access conditions, tickets, social media and languages;

[23] https://www.dati.gov.it/
[24] https://github.com/italia/daf-ontologie-vocabolari-controllati/
[25] https://www.agid.gov.it/
[26] https://teamdigitale.governo.it/

Domain Level The final level comprehends all the ontologies that describe specific domains such as accommodation facilities, events, public contracts, etc.

In addiction, there are two metadata ontologies: (1) *DCAT-AP_IT*, an extension of DCAT,[27] and DCAT-AP[28] ontologies, that aims at facilitating the interoperability between Italian data catalogs; (2) *ADMS-AP_IT*, based on ADMS[29], it is used to add metadata to all ontologies in the OntoPiA network. Table 2.3 describes all the ontologies that are part of OntoPiA, with their URIs and prefixes.

Finally, in order to facilitate the interoperability of the data, and let ontology-based application to work properly [EMS08], `CPV-AP_IT`, `CLV-AP_IT`, `L0-AP_IT`, `POI-AP_IT`, `ACCO-AP_IT`, `Lang-AP_IT`, and `COV-AP_IT` ontologies are aligned with some common ontologies, such as FOAF[30], Org[31] or GeoSPARQL[32].

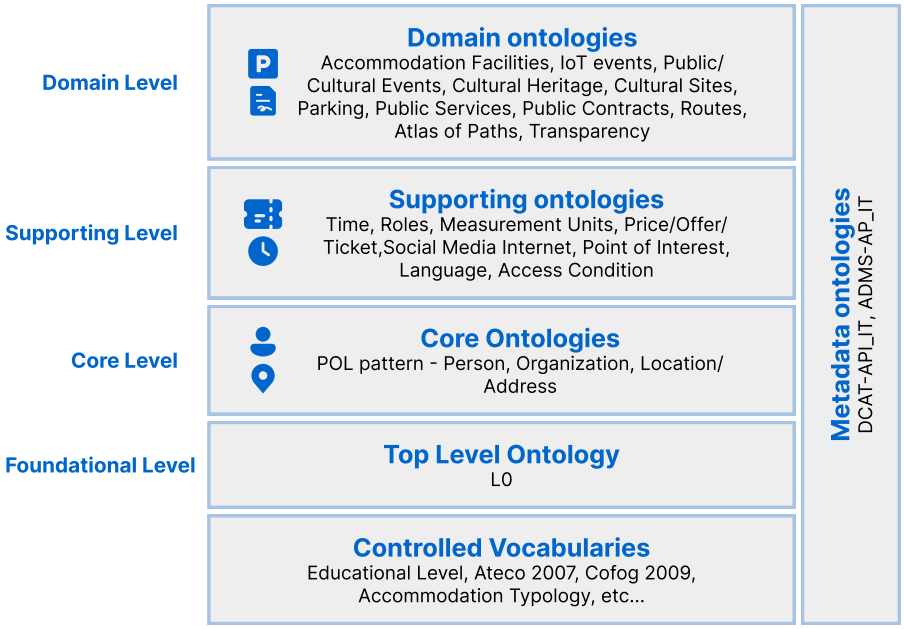


Figure 2.8: The OntoPiA ontological stack.

[27] `https://www.w3.org/TR/vocab-dcat-2/`
[28] `http://data.europa.eu/r5r/`
[29] `https://www.w3.org/TR/vocab-adms/`
[30] `http://xmlns.com/foaf/0.1`
[31] `http://www.w3.org/ns/org#`
[32] `http://www.opengis.net/ont/geosparql`

Prefix	URI	Name
Foundation Level		
l0	https://w3id.org/italia/onto/l0	Level-0
Core Level		
clvapit	https://w3id.org/italia/onto/CLV	Address (Location)
covapit	https://w3id.org/italia/onto/COV	Organization (Public or Private)
cpvapit	https://w3id.org/italia/onto/CPV	Person
Supporting Level		
acapit	https://w3id.org/italia/onto/AccessCondition	Access Conditions
langapit	https://w3id.org/italia/onto/Language	Language
muapit	https://w3id.org/italia/onto/MU	Value and Measurement Unit
poiapit	https://w3id.org/italia/onto/POI	Points of Interest
potapit	https://w3id.org/italia/onto/POT	Price/Offer/Ticket
roapit	https://w3id.org/italia/onto/RO	Role
smapit	https://w3id.org/italia/onto/SM	Social Media/Contact and Internet
tiapit	https://w3id.org/italia/onto/TI	Time
Domain Level		
accoapit	https://w3id.org/italia/onto/ACCO	Accommodation Facilities
aopapit	https://w3id.org/italia/onto/AtlasOfPaths	Atlas of Paths
chapit	https://w3id.org/italia/onto/CulturalHeritage	Cultural Heritage

Prefix	URI	Name
cis	http://dati.beniculturali.it/cis	Cultural Institute/Site and Cultural Event
cpevapit	https://w3id.org/italia/onto/CPEV	Public Events
cpsvapit	https://w3id.org/italia/onto/CPSV	Public Services
herapit	https://w3id.org/italia/onto/HER	Higher Education and Research
indicator	https://w3id.org/italia/onto/Indicator	Indicator
iotapit	https://w3id.org/italia/onto/IoT	IoT event
parkapit	https://w3id.org/italia/onto/PARK	Parking
pcapit	https://w3id.org/italia/onto/PublicContract	Public Contracts
prjapit	https://w3id.org/italia/onto/Project	Project
rtapit	https://w3id.org/italia/onto/Route	Routes
trapit	https://w3id.org/italia/onto/Transparency	Transparency Obligations
Metadata		
admsapit	https://w3id.org/italia/onto/ADMS	Asset Description Metadata Schema
dcatapit	https://w3id.org/italia/onto/DCAT	Data Catalog Vocabulary

Table 2.3: Ontologies part of the OntoPiA network.

3

Related works

As said in Chapter 1, the purpose of this thesis is to design an ontology for Italian municipalities, facilitate the publication on the Web of Linked Open Data, and develop a web application that makes this data easier for people to comprehend and visualize. It is therefore interesting to understand how major Italian, European, and global cities publish their data on the Web, and what data they publish.

The next sections will show the information collected by Italian, European and global cities about their Open Data portal, and in particular: (1) the number of available datasets; (2) the most common data file types; (3) a score from 1 to 5 based on the five stars classification presented in Section 2.2. Since the score is assigned to the entire data catalog and not to a single resource, only the types of files most present in the portal were considered.

3.1 Italian cities

For what concerns Italian cities, has been analyzed the most economically and culturally relevant cities in northern, central, and southern Italy: Bologna, Firenze, Genova, Milano, Napoli, Roma, Torino, and Venezia. The results, collected during April 2022, are shown in Table 3.1. All the cities but Bologna scored three stars, since data are mostly published in non-proprietary format, in particular CSV, JSON, and Shapefile. Firenze and Bologna use API that serves the resources in different formats. Firenze's data can be accessed in JSON format or downloaded

as a ZIP archive containing the CSV file and a metadata file. Bologna reached four stars since it lets export resources in different formats, including RDF/XML, JSON-LD, N-Triples, and Turtle. However, these resources are not accessible through SPARQL, they're not modeled using an ontology, and they're not linked to other data. For these reasons this catalog obtained four stars, and not five.

City	# Datasets	File type	Score	Software
Firenze	1902	Uses API	3	Drupal + CKAN
Bologna	425	Uses API	4	OpenDataSoft
Milano	1618	CSV (1540)	3	CKAN
Torino	1954	CSV (1460)	3	CKAN
Roma	319	CSV (230)	3	CKAN
Venezia	248	CSV (179)	3	Drupal
Genova	138	CSV (111)	3	DKAN
Napoli	62	CSV (35)	3	Custom

Table 3.1: Analysis of Italian cities' Open Data Portals. The data reported in this table was collected during April 2022.

All the Italian cities analyzed, except Napoli, follows the *Linee guida nazionali per la valorizzazione del patrimonio informativo pubblico*.[1] Indeed, they provide their entire catalog as Linked Open Data using the `DCAT_AP-IT` ontology for resource metadata, like the access and download URL, the name and the file type of the resource, the owner of the dataset, the frequency of updating the data, the theme, and more. This approach aims to maintain the ease of publishing data (e.g. using the CKAN portal), but at the same time allows resources to be more accessible, provide additional information about the nature of the data, and enables the ability to access resources from regional, national,[2] and European[3] portals.

Moving on, an example of nearly five-star data comes from Roma, which provided the list and the information of accommodation facilities[4] using the OntoPiA ontology described in Section 2.8. However, this data is provided as RDF files, and not using a SPARQL endpoint.

[1] `https://docs.italia.it/italia/daf/lg-patrimonio-pubblico/`
[2] `https://dati.gov.it`
[3] `https://data.europa.eu`
[4] `https://dati.comune.roma.it/catalog/dataset/suar2021`

Finally, some notable attempts to publish data as Linked Open Data come from Milano and Bologna, which have respectively two portals (Roma[5], and Bologna[6]) dedicated to Linked Open Data. Milano developed a custom ontology called *OntoMI*[7] that partially extends OntoPiA, which is described in Section 2.8. In particular, the ontology describes six subject areas that represent a part of the services offered by the City of Milano: libraries, administrative acts, kindergartens, consumer price detection, sports facilities, and Area C entry detection. However, the SPARQL endpoint is no longer available, and the data can no longer be accessed. For what concerns Bologna, it also developed a custom ontology, called *Onto Municipality*[8], that describes districts, areas, streets, squares and other circulation areas, civic numbering, places and people of interest, schools, and demographic statistics. For the latter, Bologna uses an ontology developed by ISTAT as part of the 2011 census[9] that is no longer maintained and accessible. Despite the SPARQL endpoint, and the data are still accessible, the project has not been maintained since 2016, making it currently useless.

3.2 European and global cities

As for Italian cities, it is interesting to analyze the approach to Open Data (and Linked Open Data) of European and global cities. In particular, has been analyzed the political and economic capitals of major European states, the United States, Canada and Australia. The results, collected during April 2022, are shown in Table 3.2. All the cities except for Amsterdam and London scored three stars, since data are mostly published in non-proprietary format, in particular CSV, JSON, and GeoJSON. On the contrary, Amsterdam and London, despite publishing data under an open license, most resources are available only in the proprietary Excel format. Notice that Berlin, Brussels, Paris, The Hague, New York, Los Angeles, Washington DC, Melbourne, and Sydney use APIs and let export the data in different formats. Brussels, and Paris, which use the same software as Bologna, and New York, Los Angeles, and Melbourne, also allow resources to be exported in RDF format, but without a SPARQL endpoint, without following an ontology,

[5] https://dati.comune.milano.it/sparql/home.html
[6] http://linkeddata.comune.bologna.it
[7] https://dati.comune.milano.it/sparql/onthdoc.html
[8] http://linkeddata.comune.bologna.it/ontologies/2014/04/onto-municipality/
[9] https://www.istat.it/it/archivio/160039

and without links to other data, so the same considerations about Bologna made in Section 3.1 apply.

City	# Datasets	File type	Score	Software
European cities				
Berlin	2470	Uses API	3	Drupal + CKAN
London	1047	XLSX (644)	2	DataPress
Zurich	683	CSV (463)	3	Custom
Vienna	560	CSV (477)	3	CKAN
Brussels	550	Uses API	4	OpenDataSoft
Barcelona	525	CSV (471)	3	CKAN
Lisbon	359	GeoJSON (206)	3	CKAN
Prague	354	CSV (194)	3	CKAN
Amsterdam	327	XLSX (n.d.)	2	Custom
Paris	321	Uses API	4	OpenDataSoft
The Hague	308	Uses API	3	Dataplatform
Madrid	195	JSON (138)	3	CKAN
Munich	176	CSV (175)	3	CKAN
Global cities				
New York	3541	Uses API	4	Socrata
Los Angeles	1635	Uses API	4	Socrata
Washington DC	1333	Uses API	3	ArcGIS Hub
Toronto	425	CSV (175)	3	WordPress + CKAN
Montreal	320	CSV (227)	3	CKAN
Melbourne	221	Uses API	4	Socrata
Sydney	176	Uses API	3	ArcGIS Hub

Table 3.2: Analysis of European and Global cities' Open Data Portals. The data reported in this table was collected during April 2022.

As for Italian cities, all cities belonging to European Union provides their catalog as Linked Open Data using the `DCAT_AP` ontology for metadata, in order to make the resources accessible through the European portal.

A similar approach applies to U.S., Canadian and Australian cities, whose catalog of data is collected using the local government API and is made available also in the central government data portal.

To conclude the analysis on Italian, European and global cities' approach to Open Data, we can definitely see that no cities publish Linked Open Data, but they prefer publish resources in using non-proprietary (and in some cases proprietary) format, reaching a score of three or fewer stars. This is probably due to the fact that convert and publishing data as Linked Open Data has a greater cost in terms of time and economic resources [BK11]. The examples of Milano and Bologna, which have stopped investing in Linked Open Data, are proof of this. However, these costs can be covered by states, ministries, government institutions, or regions, which instead publish a portion of their data as Linked Open Data. Some examples are the Europeana project,[10], Ministero della Cultura,[11] ISPRA,[12] Regione Veneto,[13] or Regione Sicilia.[14]

Of interest is the approach of Italian and EU cities in publishing the catalog in RDF format using the DCAT metadata profile, which allows them to provide some semantic information to the datasets, such as the owner of the data, the frequency of update, or the topic to which the data refer.

Finally, it is also interesting to analyze the choices of different cities regarding the software chosen to publish Open Data. The most popular tool is CKAN (14 out of 28 cities), especially for Italian and European cities (12 out of 21 cities) use CKAN for their data portal. The reasons for this choice certainly lie in the potential offered by CKAN, which, as explained in Section 2.7, is highly customizable and expandable with plugins, is an open source program and easily installed, and offers the possibility of sharing the catalog in RDF with the DCAT metadata profile, facilitating the interoperability with the central governments.

[10] https://www.europeana.eu
[11] https://dati.cultura.gov.it/
[12] http://dati.isprambiente.it/
[13] https://www.culturaveneto.it/it/
[14] https://dati.regione.sicilia.it/i-linked-open-data-nel-catalogo-regionale/

4

Requirements analysis

As introduced in Chapter 1, this thesis aims to facilitate the publication and dissemination of Open Data, and in particular Linked Open Data, by Italian municipalities by designing and developing an ontology to describe the data, and to develop a web application to make it usable for local government, citizens and businesses to consult the data. In particular, the ontology and the web applications are designed taking in consideration the needs and the data of the Comune di Sona[1], as part of the *Innovation Lab*[2] project, a project financed by Regione Veneto that aims to spread digital and Open Data culture.

One of the best practices in designing an ontology is to reuse, where possible, existing ontologies [NM+01]. Following this principle, the OntoIM ontology imports the ontologies of the OntoPiA network. As described in Section 2.8, OntoPiA is maintained by AgID and the Italian Digital Transformation Team, and aims to describe different domains of the Italian public administrations, and in particular: people, public and private organizations, addresses and locations, point of interests, accommodation facilities, paths, cultural heritage, cultural events, public services, parking, public contracts, transparency obligations, projects, routes, IoT events, indicators, and higher education and research. Where possible, these ontologies are also aligned with existing ontologies on the web.

The OntoIM ontology was therefore designed and developed as an extension

[1] `https://comune.sona.vr.it/`
[2] `https://innovationlab.regione.veneto.it/`

of the existing OntoPiA ontology, and with the aim that it would become an integral part of the network.

The first part of the design phase involved not only analyzing the data provided by the Comune di Sona, but also analyzing which data the major Italian cities share on their Open Data portals. This choice is due to the fact that we want to create an ontology that can also be reused by other administrations, and takes into account possible future extensions. The work described in Chapter 3, therefore, served not only to analyze how data are made public by various cities, but also what data is available. In addition, some data have been collected from Italian government portals or public agencies, such as Camera di Commercio, ISTAT or Agenzia delle Entrate, to have uniformly structured data across cities.

The data collected comprehends: private organizations, associations, municipal offices, events, cultural heritage, point of interests, accommodation facilities, street directory, traffic and road accidents, municipal heritage, concession acts, waste production, schools and courses organized by private organizations, and demographic statistics (which also includes statistics on tourism, association members, students, and event attendance). In addition to these requests, to make an ontology that is adaptable to other municipalities as well, the census of plants, green areas and street signs, and hospitals were added. Of course, since this is Government Open Data, the privacy of organizations and citizens must also be guaranteed.

Once the data were collected, it was necessary to understand how well OntoPiA ontologies could describe the areas involved. After that, we proceeded to design the ontology by adding the missing classes and properties, and going on to modify the existing ones where necessary.

5

Description of the OntoIM Ontology

The next two sections will describe more in details the OntoIM ontology, designed to describe the semantic areas presented in Chapter 4. In particular, Section 5.1 presents the choices made in developing the ontology, while Section 5.2 will describe the main semantic areas and will present the principal classes in each of them.

5.1 Overall design principles

The design of the ontology started analyzing the data collected from Comune di Sona, and public agencies. Table 5.1 shows what data were collected and where they were collected from.

As said in Chapter 4, the best practice of using existing ontologies where possible was followed. The next step then was to figure out which areas were already described by OntoPiA ontologies and which, instead, needed to be created or imported. The new classes created, moreover, following the design principles of OntoPiA, are subclasses of others existing in OntoPiA ontologies and, in particular, the top-level ontology `L0`. Indeed, as said in Section 2.8, this ontology allows all the ontologies to be linked, enabling the network of ontologies. The next sections of this thesis will focus on the classes and properties that are strictly part of the OntoIM ontology, while excluding those that are part of the OntoPiA ontology.

The first version of OntoIM ontology is composed of 526 classes, 569 object properties, 405 data properties. The URI of the ontology, the controlled vocabu-

Data	Source
Demographic statistics (citizens by location and year, citizenship of foreigners, statistics on names and surnames)	Comune di Sona
Associations	Comune di Sona
Civil status events (births, deaths, emigrations, immigrations, marriages, civil unions, divorces)	Comune di Sona
Concession acts	Comune di Sona
Cultural events	Comune di Sona
List of majors	Comune di Sona
Municipal heritage	Comune di Sona
Museums and cultural heritage	Comune di Sona
Point of Interests	Comune di Sona
Popular University (courses and subscribers)	Comune di Sona
Traffic observations	Local police
Accommodation facilities	Regione Veneto and Comune di Sona
Tourism (arrivals and presences by nationality/region)	Regione Veneto
Private organizations	Camera di Commercio
Addresses and civic numbers	Agenzia delle Entrate
Municipal offices	IPA (AgID)
Waste production	ISPRA
Road accidents	ISTAT
Schools	Ministero dell'Istruzione

Table 5.1: The data collected as reference for designing the OntoIM ontology, and their source.

laries and the resources are secure and permanent by using the W3 Permanent Identifier Community Group, which let create permanent Uniform Resource Locators (URLs) that redirects to defined locations on the Web. Moreover, thanks to this service it was possible to implement a content negotiation mechanisms, to return serialized resources and ontologies in different formats (such as RDF/XML or Turtle), or its visualization/documentation, depending on the request.

The persistent URI, for the OntoIM ontology is `https://w3id.org/ontoim`,

while its prefix is `ontoim`.

Finally, all the files, and the documentation of the ontologies and the controlled vocabularies are open source and available on a GitHub repository[1], which also allows for a permanent location to place the serialization and documentation of the ontology and the other resources.

5.1.1 Semantic areas

The OntoIM ontology that extends OntoPiA is divided into nine semantic areas, which describe seven specific domains about the municipality and the local territory. The semantic areas are as follows, and will be explored individually in Section 5.2:

Demographic Observations and Events This semantic area is used to describe not only the number of citizens by year, by geographic area, and by different properties, but also the number of employees that work in an organization, the members part of an association, the number of tourists, the number of subscribers to an event, and so on. It also includes the number of civic status events, like births or deaths, and singular events, like a subscription to an event, to an accommodation facility or a single civil status event, like a marriage;

Facilities and Cadastral Data The entities of this area describe general facilities and their cadastral data;

Organizations and Associations This semantic area extends the `CLV_AP-IT` ontology of OntoPiA. It describes the private and public organizations, adding information such as the enterprises' life cycle events, the typology of the organizations, and the heritage. This semantic area comprehends also associations, which are treated as private organizations;

Transparency This semantic area includes concession acts and payments from organizations (generally public administrations) to other beneficiaries;

Roads and Traffic The entities of this area describe traffic observations, road signals, and road accidents;

[1] `https://github.com/luca-martinelli-09/ontoim`

Schools This semantic area describes public and private schools, comprehensive institutes, and courses organized by public or private organizations;

Green Zones and Plants This area describes green zones, with their information, and the plants with their status;

Hospitals Entities in this area are used to describe hospitals, and hospital departments;

Waste Production It is the area that describes the observations on waste production by year, by waste category and by geographic area.

5.1.2 Controlled vocabularies

For some classes it was necessary to introduce categories or types to classify them. Some examples are the type of civil status event, the category of an association, the type of traffic signal etc. Instead of introducing numerous named individuals into the ontology, some controlled vocabularies and, in particular, taxonomies were chosen. This makes it easier to modify and keep up-to-date the possible categories and types of the various classes, and it was also possible to define hierarchies and subcategories (e.g., a wedding can be religious wedding or civil wedding).

In three cases, named individuals within the ontology were used instead: for traffic direction, for tourist type, and for plant status. Indeed, in these cases, it was not necessary to define a hierarchy, and the types will remain unchanged over time.

To distinguish controlled vocabularies from the entities of the ontology, the URI prefix `https://w3id.org/ontoim/controlled-vocabulary/` has been used.

5.2 Area-by-Area

In this section, each of the semantic areas introduced in Section 5.1.1 will be explored in depth and, in particular, the main elements that comprise it will be described. Bold font will be used to indicate classes, monospaced font for URIs, and italic font for properties. The prefixes in Table 2.3 will be used for URIs.

5.2.1 Demographic Observations and Events

This semantic area is represented in Figure 5.1, and comprehends the classes and the properties that describe demographic observations and events for the domains of demographic statistics, civil status, tourism, events, schools, and organizations.

There are two main classes in this area. The **Demographic Observation** class (`:DemographicObservation`), and the **Demographic Event** (`:DemographicEvent`) class. This makes it possible to represent data either in aggregate form (e.g., the number of marriages in a year) or, where possible, as individual events (e.g., a wedding occurred on a particular day).

The class **Demographic Observation**, which is a subclass of the **Activity** class (`l0:Activity`) has the data property *observationValue* (`xsd:nonNegativeInteger`), which represents the quantity of the observation (e.g. the number of citizens). Through the properties *hasTemporalEntity* and *hasSpatialCoverage*, **DemographicObservation** is connected to **TemporalEntity** (`tiapit:TemporalEntity`) and **Feature** (`clvapit:Feature`) respectively. In this way it is possible to define the time and space to which the observation refers. To get more accurate statistics, it may be useful to distinguish observations by gender, citizenship or other characteristics. Instead of adding different properties for each of these characteristics, you can use the *hasDemographicReference* property, which connects **DemographicObservation** to a **Person** (`cpvapit:Person`) class or **Family** (`cpvapit:Family`) class for observations about families. These classes then reference the observed characteristics.

Different type of observations are modeled using subclasses of the class **DemographicObservation**. The main subclasses are:

Tourists (`:Tourists`) To describe observations about the number of tourists. We can distinguish two type of observations about tourism: arrivals and presences. The *hasTouristType* property connect **Tourists** to the enumerated class **TouristType** (`TouristiType`), which can be `:Arrival` or `Presence`;

CivilStatus (`:CivilStatus`) To describe the number of civil status events (e.g. births, deaths, marriages, etc). The typology of the event is defined through the class **CivilStatusCategory** (`:CivilStatusCategory`), connected by the property *hasCivilStatusCategory*. The available categories are defined in a

controlled vocabulary following the entries in the ISTAT D.7.A model[2] of civil status statistics.

Subscribers (`:Subscribers`) This subclass is used to describe observations about subscribers to events, school, and courses.

The remaining subclasses, which doesn't have additional properties, are: **Employees** (`:Employees`), **Members** (`:Members`), **Citizens** (`:Citizens`), and **Bookings** (`:Bookings`). They describe respectively: (1) the number of employees that works in a organization; (2) the number of members of an association; (3) demographic observation on the population; (4) the number of bookings in a accommodation facility.

The **DemographicEvent** class is similar to **DemographicObservation**, but, as said before, is used to represent singular events. It shares with the latter the *hasDemographicReference* and *hasSpatialCoverage* properties, but it has the property *date*, which defines the date on which the event occurred. Different subclasses are used to model specific types of events. In particular, such subclasses are: (1) **CivilStatusEvent** (`:CivilStatusEvent`), which is the equivalent of **CivilStats** for the observations; (2) **Subscriber** (`:Subscriber`), to model a single registration to an event, school or course; (3) **Booking** (`:Booking`), to describe a single reservation to an accommodation facility.

5.2.2 Facilities and Cadastral Data

Figure 5.2 represent the semantic area of facilities and cadastral data, which are used to describe schools, hospitals, green zones etc.

The main class is **Facility** (`:Facility`), which is a subclass of the `POI-AP_IT`'s **PointOfInterest** class (`poiapit:PointOfInterest`), and from which it inherits all the properties. The properties *hasOnlineContactPoint*, *hasPhysicalContactPoint*, and *hasAccessConditions* connect the **Facility** class respectively to: (1) **OnlineContactPoint** (`smapit:OnlineContactPoint`), which defines online contact points such as emails, social network usernames, websites, and telephones; (2) **PhysicalContactPoint** (`smapit:PhysicalContactPoint`), which describes the address or the Point of Interest where the facility is located; (3) **AccessCondition**

[2] `https://purl.archive.org/istat-d7a-sona-2021`

(`acapit:AccessCondition`), which defines the facility opening hours. The **OSDFeature** class (`accoapit:OSDFeature`) is connected through the property *hasOfferedService*, and it is used to describe services and features that the facility offers (e.g. air conditioning, food service, parking, etc).

A facility can be owned by an organization, but can be granted for use to another organization or to a person through a deed of grant. This situation is described by the property *ownedBy*, which connect the facility to the **Organization** class that represents its owner; the property *concessedWithAct* and the class **ConcessionAct** (`:ConcessionAct`), which is described in Section 5.2.4.

Finally, a facility is identified in the land registry by one or more cadastral data. The class **CadastralData** (`:CadastralData`) stores this information, and it is connected to the **Facility** class through the *hasCadastralData* property. A facility has also a cadastral category, defined by the class **CadastralCategory** (`:CadastralCategory`). The elements of this class must be defined according to a controlled vocabulary that store the available cadastral categories.[3]

5.2.3 Organizations and Associations

This semantic area is represented in Figure 5.3, and covers the domains of public and private organizations, and associations. The area was designed following the structure of the data provided by the Camera di Commercio.[4]

Three data properties concerning events in the life cycle of an organization have been added to the `COV-AP_IT`'s **Organization** class (`covapit:Organization`). These properties, of type `xsd:date`, are *endActivityDate*, *bankruptcyDate*, and *liquidationDate*.

The property *hasLocalUnitAddress* connects the **Organization** class to an **Address** class (`clvapit:Address`), and it is used to store the addresses of one or more local units of the organization. It should also be specified that an organization may have its primary address in another city and a local unit in the municipality concerned. In that case, the Camera di Commercio provides only the address of the latter, specifying that it is a local unit.

An organization may also have real estate assets. Since there can be different

[3] `https://purl.archive.org/age-categorie-catastali`
[4] `https://www.mn.camcom.gov.it/files/RegistroImprese/Legenda-elenchi.pdf`

types of heritage, such as unavailable heritage or state property, the structures owned by the organization are grouped, through the property *hasFacility*, in the **Heritage** class (`:Heritage`), to which its type defined by the **HeritageType** class (`:HeritageType`) is linked, through the property *hasHeritageType*, and whose elements are defined in a controlled vocabulary.

For what concern demographic observations, as specified in Section 5.2.1, the **Employees** class describes the number of employees that works in the organization.

This semantic area also extends the `COV-AP_IT`'s **PrivateOrganization** class (`covapit:PrivateOrganization`), which is a subclass of an **Organization**. The property *hasOrganizationSection* connect the company to the **OrganizationSection** class (`:OrganizationSection`). The elements of this class are defined in a controlled vocabulary created from the sections provided by the Camera di Commercio. Another controlled vocabulary defines the elements that must be used for the **CompanyDemographicCategory** class (:CompanyDemographicCategory), which defines whether the enterprise is a youth, female or foreign enterprise. Finally, artisan organizations are described by the class **ArtisanOrganization** (`:ArtisanOrganization`), which has the two data properties *artisanRegisterCode* (`rdfs:Literal`), which define the identifier of the organization in the register of artisans, and *artisanRegistrationDate* (`xsd:date`), which define the date when the organization was registered.

Associations are treated as private organizations. Indeed, the **Association** class (`:Association`) is a subclass of the **PrivateOrganization** class. The data property *associationRegisterCode* (`rdfs:Literal`) defines the identifier of the association in the register of associations. The other two data properties, *associationRegistrationDate*, and *associationRemovalFromRegisterDate*, define the life cycle events for an association. The class **AssociationCategory** (`:AssociationCategory`), connected through the property *hasAssociationCategory*, define the category of the association (such as ONLUS, cultural association, etc). The elements of this class must be defined according to a controlled vocabulary.

Finally, as specified in Section 5.2.1, the **Members** class is used to describe demographic observations on the members of the association, while the **Subscriber** class is used to describe singular membership events.

5.2.4 Transparency

This semantic area contains classes that extend the `Transparency-AP_IT` ontology for transparency obligation. These classes are graphically represented in Figure 5.4.

The main class is **ConcessionAct** (`:ConcessionAct`), which defines a deed of concession from an organization to another organization or person, and it is used either to grant the management of a facility, or to send a payment. The data properties of the **ConcessionAct** class are: (1) *actTitle* (`xsd:string`), which defines the title of the act; (2) *actNumber* (`xsd:string`), which defines the code number of the act; (3) *paymentAmount* (`xsd:float`), which defines the amount of the payment; (4) *actDate* (`xsd:date`), which defines the date of the act.

The *hasBeneficiary* property connects the **ConcessionAct** to the **Agent** class (`l0:Agent`), which can be both an organization or a person. The organization that sign the concession act is connected by the *hasOrganization* property.

The signed document of the concession act can be stored as a **TransparencyResource** element (`transapit:TransparencyResource`).

The referent of the organization that sign the concession act is defined by a **TimeIndexedRole** class (roapit:TimeIndexedRole) through the property *hasActReferent*.

Finally, the typology of the concession act is defined by the **ConcessionActType** class (`:ConcessionActType`), whose elements are defined in a controlled vocabulary.

5.2.5 Roads and Traffic

This semantic area, which is represented in 5.5, extends the OntoPiA ontology regarding the traffics flow observations, and contains classes that describe the road signals and the road accidents.

For what concerns the traffic flow observations, the `IOT-AP_IT`'s **TrafficFlow** class (`ioapit:TrafficFlow`) is connected through the property *hasTrafficFlowDirection* to the **TrafficFlowDirection** class (`:TrafficFlowDirection`). This class represents the direction of the flow respect to the sensor, and is thus an enumerated class that can have individual `:In` or individual `:Out` as its elements. The **RoadSegment** (`:RoadSegment`) class, which define the feature of interest for the observation, is connected through the property *hasStreetToponym* to the name of the road

observed, represented by the **StreetToponym** class (`clvapit:StreetToponym`). Finally, the observations can be enriched with the category of vehicle observed. This can be done through the property *hasVehicleCategory* that connects **TrafficFlow** to the **VehicleCategory** class (`:VehicleCategory`). The elements of this class are in a controlled vocabulary, whose elements are defined following the road code classification.[5]

It is also useful for a city to keep track of the status and location of street signs to aid their maintenance. The class **RoadSignal** (`RoadSignal`) describes a street sign, and is a subclass of the **PointOfInterest** class (`poiapit:PointOfInterest`). In addition to those inherited from the **PointOfInterest** class (such as the location), the **RoadSignal** class has three more data properties: (1) *installationDate* (`xsd:date`), which defines the date when the signal was installed; (2) *removalDate* (`xsd:date`), which defines the date when the signal was removed; (3) *signalValue*, which defines the value on the signal (e.g. the speed for a speed limit signal). The typology of the road signal is defined through the property *hasSignalType*, which connects it to the class **RoadSignalType** (`:RoadSignalType`). The elements of this class are defined in a controlled vocabulary generated from the list of road signs in Italy.[6]

The last subarea is dedicated to road accidents, and was developed following the ISTAT model used to collect road accidents, both to design the ontology and to define the controlled vocabularies.[7]

The main class is **RoadAccident** (`:RoadAccident`), which is a subclass of `L0`'s **EventOrSituation** (`l0:EventOrSituation`). The date and time when the incident occurred are defined by the data property *date* (`xsd:dateTime`). The **Geometry** class (`clvapit:Geometry`) describe the geographical location of the accident.

The property *detectedBy* connects **RoadAccident** with the class **RelevationUnit** (`:RelevationUnit`), defining who recorded the incident (e.g., police, carabinieri, etc). The elements of the latter class are defined in a controlled vocabulary. The class **AccidentType** (`:AccidentType`) defines the nature of the accident (e.g. head-on collision, pedestrian investment, etc), and its elements are also defined

[5] `https://www.polizialocaleterredifrontiera.com/codice-della-strada/veicoli/`
[6] `https://it.wikipedia.org/wiki/Segnaletica_stradale_in_Italia`
[7] `https://purl.archive.org/istat-incidenti-stradali-2018`

in a controlled vocabulary. Another class whose elements are defined in a controlled vocabulary is **WeatherCondition** (`:WeatherCondition`), which defines the weather conditions at the time of the accident.

The **Road** (`:Road`) class is used to store information about the road and its status at the time of the accident. The address where the location happened is defined by the **Address** class (`clvapit:Address`) through the property *hasAddress*. Then, there are several classes that defines the context and the status of the road during the accident. The elements of these classes must all be defined according to the relative controlled vocabularies. Such classes are:

RoadCategory (`:RoadCategory`), which defines whether the road is a municipal road, a state road, a highway, etc;

RoadType (`:RoadType`), which defines whether the road is single- or multi-lane or one-way;

PavementType (`:PavementType`), which defines whether the road is paved or uneven;

RoadContext (`:RoadContext`), which defines whether the accident occurred at an intersection, traffic circle, on a straight road, etc;

RoadbedStatus (`:RoadbedStatus`), which defines the status of the roadbed (e.g. dry, wet, frozen, etc);

RoadSignalPresence (`:RoadSignalPresence`), which defines whether the road signals are present or not, or what type of signals are present.

Finally, there are some classes that describe the entities involved (obstacles, persons, vehicles) in the accident. These entities are described by the class **InvolvedEntity** (`:InvolvedEntity`). The property *hasAccidentCirumstance* connects this class to the **AccidentCircumstance** class (`:AccidentCircumstance`), which defines what the entity involved was doing during the accident (e.g., the pedestrian was crossing the street, or the car was proceeding at high speed). The elements of this class are defined in a controlled vocabulary. To define whether the entity is a vehicle, an obstacle, or a person, the **InvolvedEntity** class has three subclasses: **InvolvedVehicle** (`:InvolvedVehicle`), **InvolvedPerson** (`:InvolvedPerson`), and **InvolvedObstacle** (`:InvolvedObstacle`). The **InvolvedPerson** class inherits also the properties of the `CPV-AP_IT`'s **Person** class (`cpvapit:Person`), so it is possible

to define the sex and age of the person involved. The status of the person involved (e.g. wounded, uninjured, dead, etc) in the accident is described using the class **InvolvedPersonStatus** (`:InvolvedPersonStatus`), whose elements are defined in a controlled vocabulary. For what concerns the vehicle, the class **InvolvedVehicle** has the properties *hasFrontPassenger*, *hasBackPassenger*, and *hasConducent*, which are sub-properties of *hasConducent*, and defines the person involved that was inside the vehicle. The **InvolvedVehicle** is also a subclass of the **Vehicle** class (`:Vehicle`). This class, which represents a vehicle, has the data properties *licensePlate* (`xsd:string`), *registrationYear* (`xsd:gYear`), *brand*, *model*, and *color*, which defines the characteristics of the vehicle. The properties *hasEngineDisplacement*, *hasLength*, *hasWidth*, *hasHeight*, and *hasHeight* are used to describe the sizes of the vehicle, using the class **Value** (`muapit:Value`). The category of the vehicle is described by the **VehicleCategory** class. The registration country is defined through the property *hasRegistrationCountry*, which connects the **Vehicle** class to the **Country** class (`clvapit:Country`).

Note that it was still chosen to include properties that in the Open Data environment would not be used for privacy reasons (e.g., the license plate or name of the people involved). This is because in this way the ontology can also be used internally to describe traffic accidents using Linked Data.

5.2.6 Schools

The classes of this area describes the school, comprehensive institutes, and courses organized by schools. It should also be specified that in this ontology schools also include training institutions, music schools, sports associations, popular universities, etc. Figure 5.6 shows the graphical representation of these classes.

For what concern schools, the class **School** (`:School`) is the core part of the area, and it inherits the properties from the **Facility** class. The data property *schoolCode* represents the MIUR[8] identifier for the school. The class **SchoolType** (`:SchoolType`) defines the typology of the school (e.g. primary school, high school, etc) and its elements must be defined according to a controlled vocabulary. The subclasses **PublicSchool** (`:PublicSchool`), and **PrivateSchool** (`:PrivateSchool`) are used to distinguish private and public schools. In Italy,

[8] `https://www.miur.gov.it/`

schools are part of comprehensive schools. This situation is described by the **ComprehensiveInstitute** class (`:ComprehensiveInstitute`), and the property *includesSchool*, which links comprehensive institutes to the schools they include.

As said in Section 5.2.1, the classes **Subscribers** and **Subscriber** are used to describe demographic observations on schoolchildren and enrollment events.

For what concern courses, the core class of this subarea is **Course** (`:Course`). The main information of the course are defined by the data properties *courseCode*, which is the identifier of the course, *name*, *description*, *durationHours*, and by the properties: (1) *atTime*, which connects it to a **TimeInterval** class (`tiapit:TimeInterval`) to define the period in which the course is provided; (2) *hasPriceSpecification*, which connects **Course** to the class **PriceSpecification** (`potapit:PriceSpecification`) to define the cost of the course, if any; (3) *hasAcademicDiscipline*, which connects the **Course** class to the **AcademicDiscipline** class (`:herapit:AcademicDiscipline`) to define the school discipline of the course.

The location (phisical or virtual) of the course is specified by the **Classroom** class (`:Classroom`) through the property *situatedInClassroom*. The subclass **PhysicalClassroom** (`:PhysicalClassroom`) defines a physical classroom with an address defined by the class **Address**. On the other hand, if a course is provided using a video communication service, the classroom can be described as a **VirtualClassroom**, specifying also the service used through the class **VideoCommunicationService** (`:VideoCommunicationService`), which describes the name of the service (e.g. Zoom, Google Meet), and its URL.

5.2.7 Green Zones and Plants

Figure 5.7 represent this semantic area, which describes green zones (e.g. parks, flowerbeds, gardens), and plants.

A green zone is treated as a facility. Indeed, the class **GreenZone** (`:GreenZone`) is a subclass of the class **Facility**, from which it inherits the properties. The size of the green zone is defined by the property *hasSurface*, which connects the class to a **Value** class. A green area may have plants within it, so the *hasPlant* property connects a green area with trees within it.

The class **Plant** (`:Plant`) contains the main immutable information about the plant: (1) *plantCode* is the identifier; (2) *commonName* the common name of the

plant; (3) *species* its species; (4) *birtYear* (`xsd:gYear`) the birth year; (5) *plantingDate* (`xsd:DateTime`) the planting date and time. It also inherits the properties from the class **PointOfInterest**, such as the geographical location of the plant. Finally, it is useful to keep track of the state of the plant over time. The class **PlantStatusInTime** (`:PlantStatusInTime`) do this, storing information about the dimensions of the plant, with the properties *hasDiameter*, and *hasHeight*, and the health status, using the enumerated class **PlantHealthStatus** (`:PlantHealthStatus`).

5.2.8 Hospitals

Figure 5.8 represents the hospitals.

The main class is **Hospital** (`:Hospital`), which is a subclass of the **Facility** class, and form which it inherits the properties. The data property *totalNumberOfBeds* defines the beds available in the hospital. The two subclasses **PublicHospital** (`:PublicHospital`), and **PrivateHospital** (`:PrivateHospital`) are used to specify the typology of the hospital.

Finally, as for school and comprehensive institutes, the class **HospitalDepartment** (`:HospitalDepartment`) is treated as a hospital and describes the various departments into which a hospital is divided.

5.2.9 Waste Production

This semantic area is represented in Figure 5.9, and defines observations on waste production.

This semantic area is structured as demographic observation. The main class **WasteProduction** (`:WasteProduction`) is connected to a **TemporalEntity** class, and a **Feature** class, which define the space and time to which the observation refers. The value of the observation is defined by the class **Value**, for which you can also define the unit of measurement (e.g. kg or t). To indicate an observation made on only one type of waste, the **WasteCategory** (`:WasteCategory`) class can be used, whose elements are defined in a controlled vocabulary populated with the categories defined by ISPRA.[9]

[9] `https://www.isprambiente.gov.it/it`

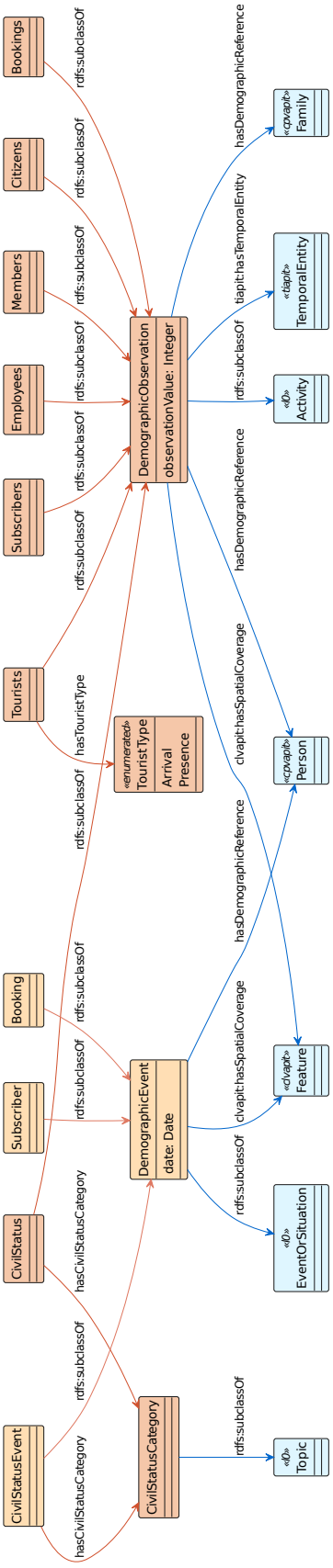


Figure 5.1: Demographic Observations and Events semantic area.

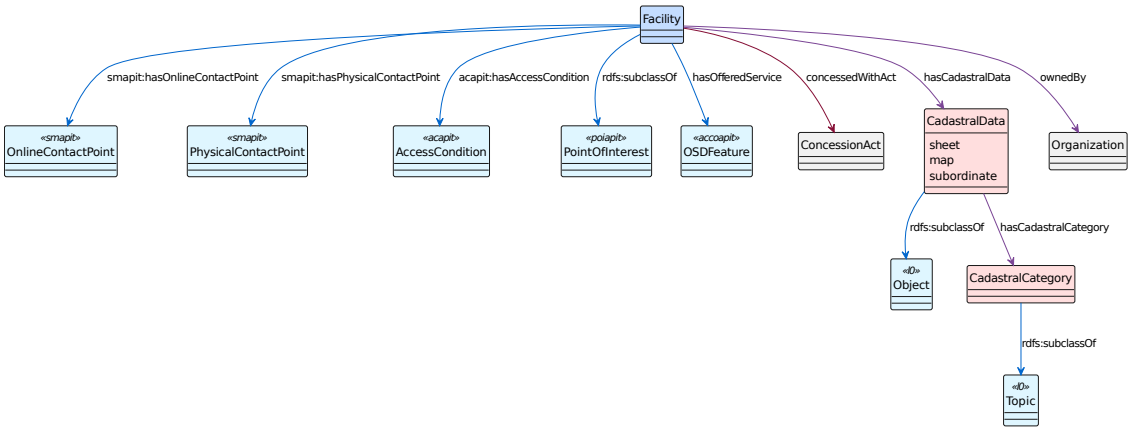


Figure 5.2: Facilities and Cadastral Data semantic area.

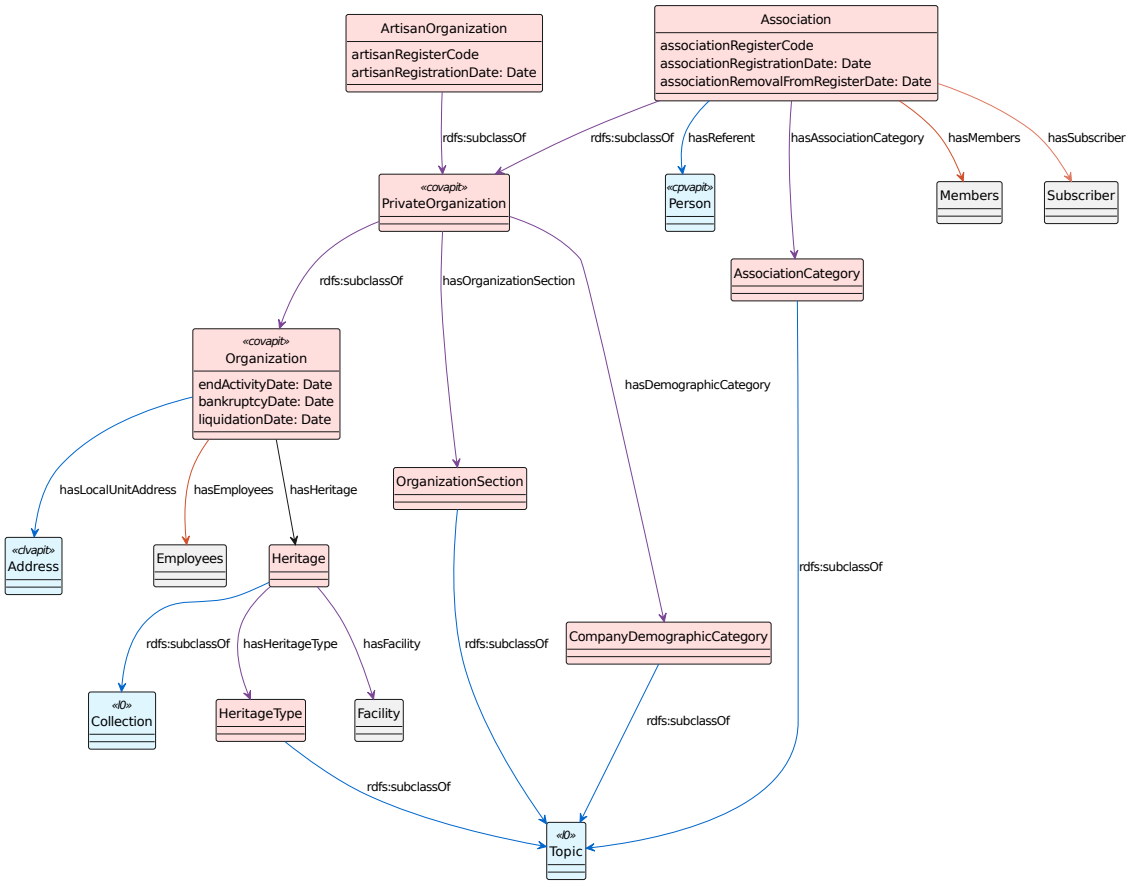


Figure 5.3: Organizations and Associations semantic area.

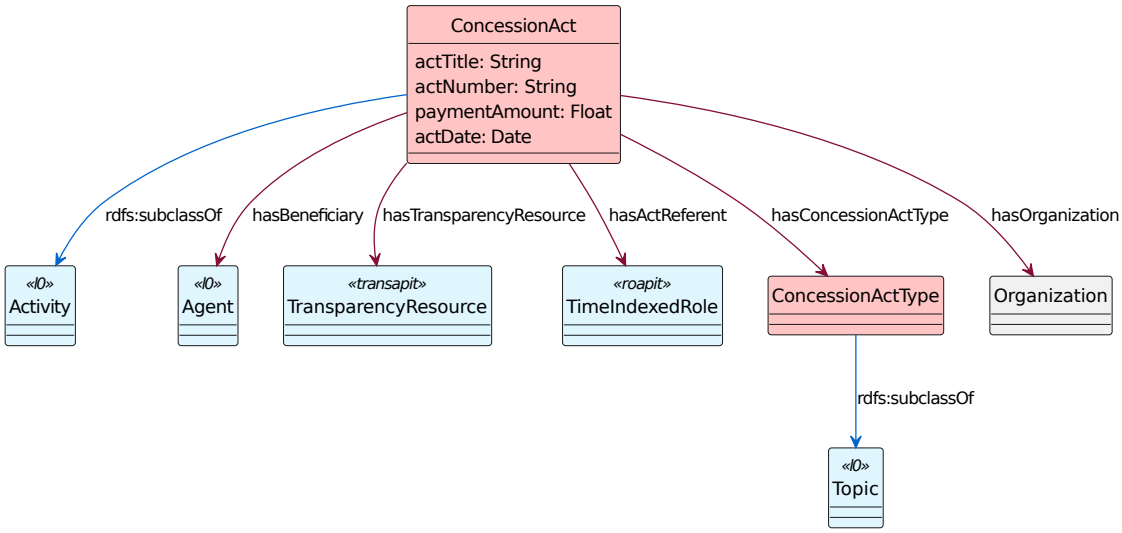


Figure 5.4: Transparency semantic area.

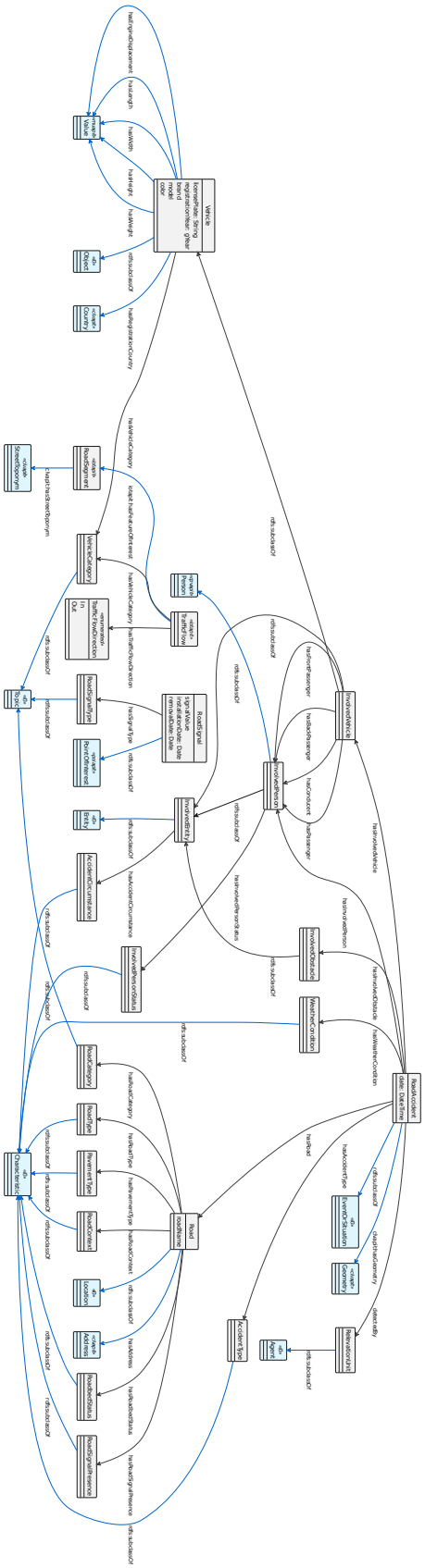


Figure 5.5: Roads and Traffic semantic area.

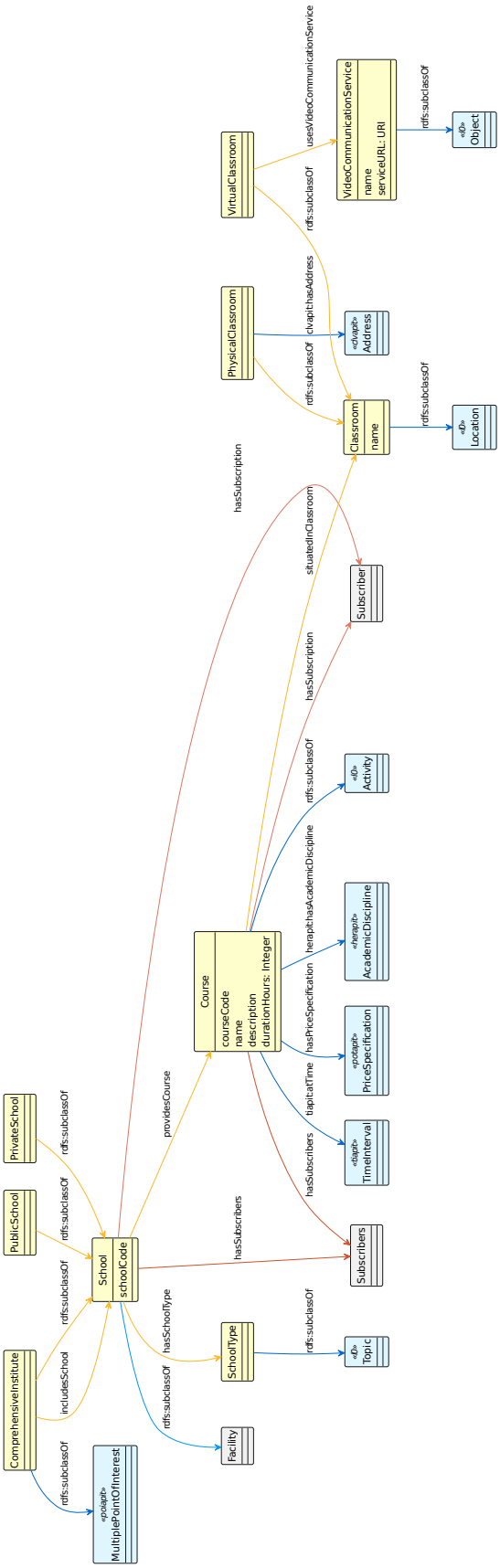


Figure 5.6: Schools semantic area.

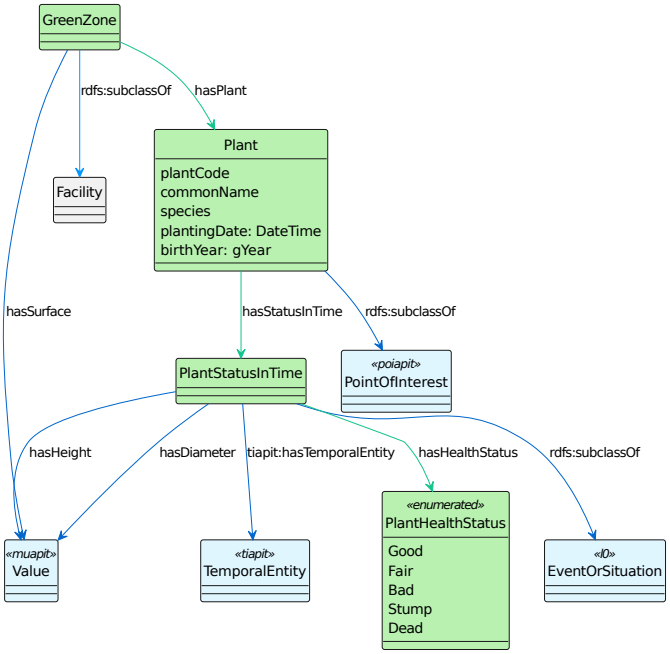


Figure 5.7: Green Zones and Plants semantic area.

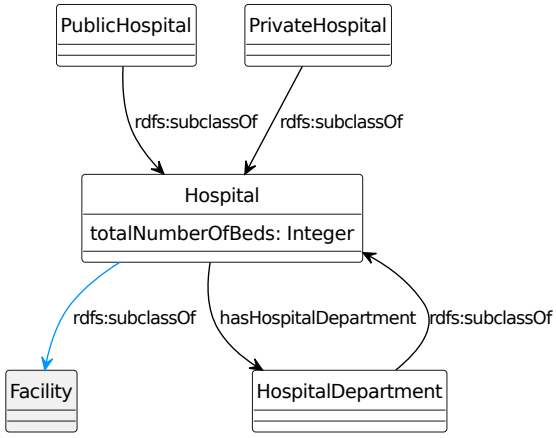


Figure 5.8: Hospitals semantic area.

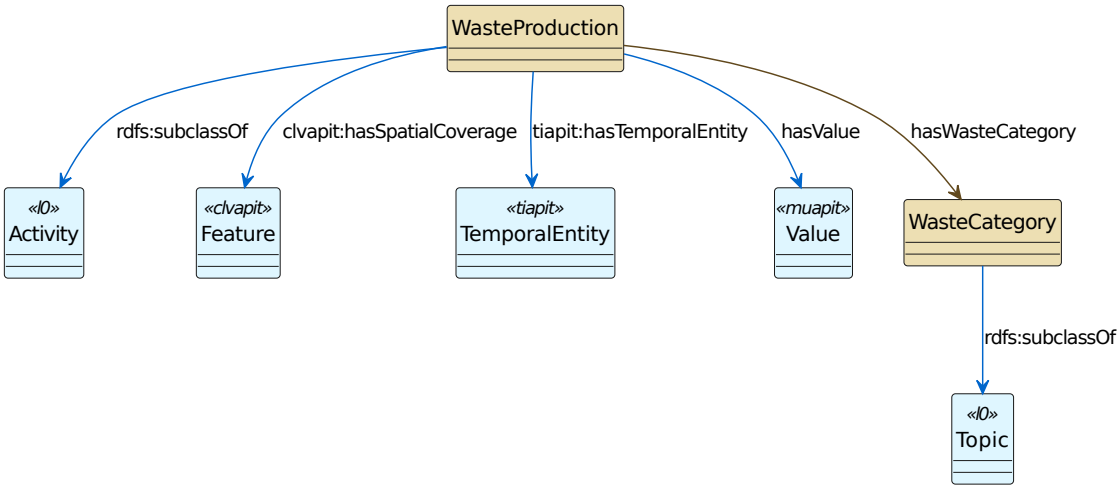


Figure 5.9: Waste Production semantic area.

6

RDF Graph Builder

A fundamental part for providing Linked Open Data on the Web is to build the RDF Graph from the available data. The main obstacle in this project is the fact that different municipalities and public organizations may model the data in different ways, thus making the process of mapping the data into an RDF Graph in accordance with the OntoIM and OntoPiA ontologies difficult. The main idea, as shown in Figure 6.1, is to collect and process data from different sources: the city's CKAN portal, offline files, and Open Data published by state and regional agencies. Thus, two libraries were made in Python to simplify graph creation through object-oriented programming by converting ontology classes into Python classes, and related properties into attributes. This makes data entry from the various sources easier and code reading more understandable. Both the Python libraries and the RDF Graph builder also make use of well-known libraries such as `pandas` and `rdflib` for data manipulation and reading, and graph creation and serialization. Finally, the sources of the various resources and other configurations are all defined in a single `conf.ini` file to allow for better control and easier editing. An example of this file is shown in Code 6.1.

```
[ANNCSU]
streets = anncsu/streets.csv
civics = anncsu/civics.csv
census_sections = anncsu/census_sections.kml
post_code = 37060

[ORGANIZATIONS]
organizations = organizations/organizations.csv
```

```
9
10 [WASTE]
11 waste_production = waste/waste_production.csv
```

Code 6.1: Example of a config.ini file that defines sources for some semantic areas.

Resources will be grouped into datasets defined as `dcat:Dataset`, and, for the specific case of the Comune di Sona, will be identified by URLs formatted as follows:

`https://w3id.org/sona/data/{dataset_id}/.../{resource_id}`

The `dataset_id` is the identifier of the dataset (e.g. *schools*, *demographic-observations*, etc), while the `resource_id` is the identifier of the resource, which is generated following different formats depending on the resource. Some supporting resources (e.g., geographic locations or time intervals) may also be located in additional paths that define the resource type (e.g. *geo* or *ti*).

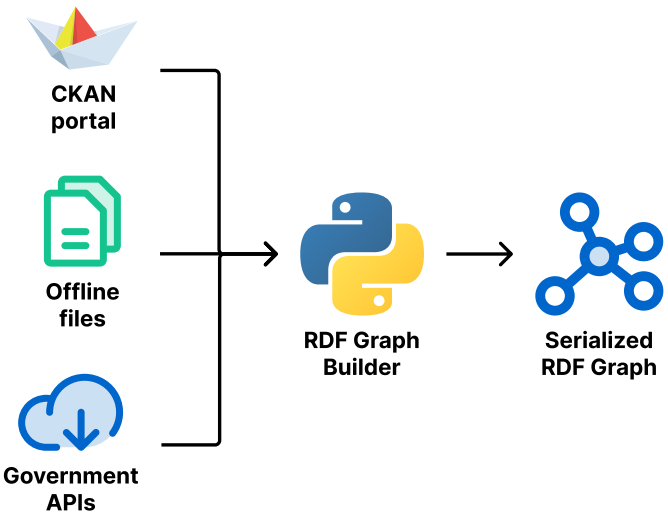


Figure 6.1: RDF Graph Builder architecture.

Section 6.1 will describe the Python libraries developed to facilitate the creation of the RDF Graph, while Section 6.2 will present some examples of data mapping for different data and semantic areas.

6.1 OntoPiA-Py and OntoIM-Py libraries

To simplify and facilitate the process of creating the RDF Graph and make the program more understandable, two libraries were created that allow the cre-

ation of resources and the assignment of properties using the object-oriented programming paradigm. These libraries are called `ontopia-py` and `ontoim-py`, for OntoPiA and OntoIM ontologies, respectively. Following the OntoIM ontology, the `ontoim-py` library imports the `ontopia-py` library. The two libraries are also available under an open source license on GitHub[1][2], and can be installed through the Python PyPI[3] package manager, with the command:

```
pip install ontopia_py ontoim_py
```

The `ontopia-py` package is organized as follows, while the `ontoim-py` package follows a similar structure:

```
/
├── ontopia_py
│   ├── accesscondition
│   │   ├── __init__.py
│   │   ├── AccessCondition.py
│   │   ├── AdmissionType.py
│   │   ├── Booking.py
│   │   ├── OpeningHoursSpecification.py
│   │   └── TemporaryClosure.py
│   ├── clv
│   │   └── ...
│   ├── ...
│   ├── __init__.py
│   ├── Dataset.py
│   ├── ns.py
│   └── Thing.py
└── setup.py
```

This directory `ontopia_py` holds the directories of the OntoPiA's ontologies. Each of these subdirectory holds the files that represent the classes of the ontology. The `ns.py` file contains the namespaces of both ontologies and controlled vocabularies, as shown in Code 6.2.

```
1 from rdflib import Namespace
2
3 L0 = Namespace("https://w3id.org/italia/onto/l0/")
```

[1] https://github.com/luca-martinelli-09/ontoim-py
[2] https://github.com/luca-martinelli-09/ontopia-py
[3] https://pypi.org/

```
TRANSP = Namespace("https://w3id.org/italia/onto/Transparency/")
TI = Namespace("https://w3id.org/italia/onto/TI/")
SM = Namespace("https://w3id.org/italia/onto/SM/")
ROUTE = Namespace("https://w3id.org/italia/onto/Route/")
RO = Namespace("https://w3id.org/italia/onto/RO/")
PUBC = Namespace("https://w3id.org/italia/onto/PublicContract/")
PROJ = Namespace("https://w3id.org/italia/onto/Project/")
POT = Namespace("https://w3id.org/italia/onto/POT/")
POI = Namespace("https://w3id.org/italia/onto/POI/")
PARK = Namespace("https://w3id.org/italia/onto/PARK/")
MU = Namespace("https://w3id.org/italia/onto/MU/")
LANG = Namespace("https://w3id.org/italia/onto/Language/")
IOT = Namespace("https://w3id.org/italia/onto/IoT/")
INDIC = Namespace("https://w3id.org/italia/onto/Indicator/")
HER = Namespace("https://w3id.org/italia/onto/HER/")
CULTHER = Namespace("https://w3id.org/italia/onto/CulturalHeritage/")
CPV = Namespace("https://w3id.org/italia/onto/CPV/")
CPSV = Namespace("https://w3id.org/italia/onto/CPSV/")
CPEV = Namespace("https://w3id.org/italia/onto/CPEV/")
COV = Namespace("https://w3id.org/italia/onto/COV/")
CLV = Namespace("https://w3id.org/italia/onto/CLV/")
PATHS = Namespace("https://w3id.org/italia/onto/AtlasOfPaths/")
ACOND = Namespace("https://w3id.org/italia/onto/AccessCondition/")
ACCO = Namespace("https://w3id.org/italia/onto/ACCO/")
ADMS = Namespace("https://w3id.org/italia/onto/ADMS/")
CIS = Namespace("http://dati.beniculturali.it/cis/")
```

Code 6.2: Part of the ns.py file that contains the namespaces of OntoPiA's ontologies.

The file `__init__.py` contains the library's main information, like the version number, the description, and the author. Two functions are also defined in this file:

createGraph Initializes the RDF Graph and binds the main namespaces (XSD, SKOS, DCAT, etc) and those declared in the `ns.py` file;

saveGraph Serializes the graph in the different formats.

These two functions are the one showed in Code 6.3.

```
def createGraph():
  g = Graph()

```

```
4   g.bind("xsd", XSD)
5   g.bind("foaf", FOAF)
6   g.bind("owl", OWL)
7   g.bind("dc", DC)
8   g.bind("xml", XMLNS)
9   g.bind("dct", DCTERMS)
10  g.bind("rdf", RDF)
11  g.bind("rdfs", RDFS)
12  g.bind("dcat", DCAT)
13  g.bind("prov", PROV)
14  g.bind("skos", SKOS)
15
16  g.bind("l0", L0)
17  g.bind("trapit", TRANSP)
18  g.bind("tiapit", TI)
19  g.bind("smapit", SM)
20  g.bind("rtapit", ROUTE)
21
22  ...
23
24  return g
25
26 def saveGraph(g: Graph, fileName: str):
27  formats = [
28    {"ext": "ttl", "fmt": "turtle"},
29    {"ext": "rdf", "fmt": "xml"}
30  ]
31
32  for format in formats:
33    ext = format["ext"]
34    fmt = format["fmt"]
35
36    with open("{}.{}".format(fileName, ext), "w", encoding="utf-8") as
     fp:
37      fp.write(g.serialize(format=fmt))
```

Code 6.3: Part of the __init__.py file that contains the two functions for creating and saving the graph.

The most important file, that initialize a resource, is `Thing.py`, and all the other classes extends the class `Thing`. The class `Thing` initialize the resource, add the labels and, if specified, connect the resource to a `dcat:Dataset`. In this way, the functions that are responsible for entering the main properties, common to all

resources, are handled by the `Thing` class. The subclasses then only have to deal with defining the attributes, which correspond to the properties to be inserted into the graph.

```
class Thing:
  _dataset: URIRef = None
  _titles: List[Literal] = []
  uriRef: URIRef = None

  def __init__(self, id: str, baseUri: Namespace, dataset: Dataset =
   None, titles: List[Literal] = []):
    self._dataset = dataset
    self._titles = titles
    self.uriRef = URIRef(baseUri[id])

  def _addProperties(self, g: Graph):
    pass

  def addToGraph(self, g: Graph, isTopConcept=False, onlyProperties=
   False):
    if not onlyProperties:
      g.add((self.uriRef, RDF.type, self.__type__))
      for title in self._titles:
        g.add((self.uriRef, DC.title, title))

      if self._dataset:
        g.add((self.uriRef, SKOS.inScheme, self._dataset.uriRef))

      if isTopConcept:
        g.add((self._dataset.uriRef, SKOS.hasTopConcept, self.uriRef))

    self._addProperties(g)
```

Code 6.4: The ontopia-py's Thing class.

Finally, the `rdfs:subclassOf` property is handled through object-oriented programming inheritance. In the Code 6.5 example, the `Sequence` class, defined in the `L0` ontology, inherits the properties of the `Collection` class, which inherits those of the `Entity` class, which is a subclass of `Thing`. This way it is not necessary to consult the ontology documentation to trace back all the properties that a given class admits, and it is more difficult to make mistakes in creating the graph. In addition, the class type, which is mapped to the `rdf:type` property, is defined by each class in the `__type__` attribute, which defines the URI of the ontology class.

```python
# Sequence.py
class Sequence(Collection):
  __type__ = L0["Sequence"]

  hasFirstMember: Entity = None
  hasLastMember: Entity = None

  def _addProperties(self, g: Graph):
    super()._addProperties(g)

    if self.hasFirstMember:
      g.add((self.uriRef, L0["hasFirstMember"], self.hasFirstMember.uriRef))

    if self.hasLastMember:
      g.add((self.uriRef, L0["hasLastMember"], self.hasLastMember.uriRef))

# Collection.py
class Collection(Entity):
  __type__ = L0["Collection"]

  hasMember: List[Entity] = None

  def _addProperties(self, g: Graph):
    super()._addProperties(g)

    if self.hasMember:
      for hasMember in self.hasMember:
        g.add((self.uriRef, L0["hasMember"], hasMember.uriRef))
```

Code 6.5: The ontopia-py's Sequence and Collection classes. Thanks to the object-oriented programming inheritance it is possible to map ontology classes and properties into Python classes and attributes.

Regarding `ontoim-py` library, the structure is the same as that of `ontopia-py`. The `ns.py` file imports all the OntoPiA's namespaces, and declares the OntoIM one. The `__init__.py` file imports the `ontopia-py`'s `saveGraph` function, and extends the `createGraph` function adding the new bindings. Classes such as `Organization`, whose declaration is shown in Code 6.6, for example, which extend the OntoPiA ontology class, inherit attributes and functions from the `ontopia-py` library, adding the new ones declared in OntoIM.

```python
from ontopia_py.cov.Organization import Organization

class Organization(Organization):
  __type__ = ONTOIM["Organization"]

  hasEmployees: List[Employees] = None
  hasHeritage: List[Heritage] = None
  hasLocalUnitAddress: List[Address] = None
  endActivityDate: Literal = None
  liquidationDate: Literal = None
  bankruptcyDate: Literal = None

  def _addProperties(self, g: Graph):
    super()._addProperties(g)

    if self.hasEmployees:
        for hasEmployees in self.hasEmployees:
            g.add((self.uriRef, ONTOIM["hasEmployees"], hasEmployees.uriRef))

    if self.hasHeritage:
      for hasHeritage in self.hasHeritage:
        g.add((self.uriRef, ONTOIM["hasHeritage"], hasHeritage.uriRef))

    if self.hasLocalUnitAddress:
      for hasLocalUnitAddress in self.hasLocalUnitAddress:
        g.add((self.uriRef, ONTOIM["hasLocalUnitAddress"], hasLocalUnitAddress.uriRef))

    if self.endActivityDate:
      g.add((self.uriRef, ONTOIM["endActivityDate"], self.endActivityDate))

    if self.liquidationDate:
      g.add((self.uriRef, ONTOIM["liquidationDate"], self.liquidationDate))

    if self.bankruptcyDate:
      g.add((self.uriRef, ONTOIM["bankruptcyDate"], self.bankruptcyDate))
```

Code 6.6: The ontopim-py's Organization class. This class inherits the one defined in ontopia-py, declaring the new attributes.

The process of creating and saving an RDF Graph using these two libraries is the following one:

1. Initialize the graph with the function `createGraph`;
2. Create a `dcat:Dataset` object using the class `Dataset`, specifying its URI and adding properties like the name or the author of the Dataset;
3. Add the dataset to the graph calling the function `addToGraph`;
4. Initialize the object that must be inserted into the graph, specifying the identifier of the resource, the base URI, the dataset into which it is to be inserted, and the name of the resource;
5. Add the properties to the resource declaring the attributes of the object created in the step before;
6. Add the resource to the graph calling the function `addToGraph`;
7. Save the graph using the function `saveGraph`, specifying the filename where to save it.

The example code of Code 6.7 creates an RDF Graph with the *ANNCSU* dataset, into which a resource of type `StreetToponym` is inserted.

```
from rdflib import XSD, Graph, Literal, Namespace

from ontopia_py import Dataset, createGraph
from ontopia_py.clv import StreetToponym

# Set namespace for data
DATA: Namespace = Namespace("https://w3id.org/sona/data/")
ANNCSU: Namespace = Namespace("https://w3id.org/sona/data/ANNCSU/")

# Create the graph and bind the namespace
g = createGraph()
g.bind("anncsu", ANNCSU)

# Create the concept scheme
ANNCSU_DATASET: Dataset = Dataset(DATA["ANNCSU"])
ANNCSU_DATASET.label = [
    Literal("Numeri civici e strade urbane", lang="it"),
    Literal("Civic Addressing and Street Naming", lang="en")
]

```

```
# Add to graph
ANNCSU_DATASET.addToGraph(g)

# Create the street toponym
streetToponym: StreetToponym = StreetToponym(
  id="street-1",
  baseUri=ANNCSU,
  dataset=ANNCSU_DATASET,
  titles=[Literal("Via Roma", datatype=XSD.string)]
)
streetToponym.toponymQualifier = "Via"
streetToponym.officialStreetName = "Roma"

# Add to graph
streetToponym.addToGraph(g)
```

Code 6.7: An example of the creation of an RDF Graph with the ontoim-py and ontopia-py libraries.

6.2 Data mapping for different semantic areas

After presenting and describing the `ontopia-py` and `ontoim-py` libraries, the next sections will show how the data entry of the Comune di Sona took place for the following semantic areas: Addresses, Organizations, and Schools. Indeed, these three semantic area are respectively examples of data mapping from the CKAN Open Data Portal, offline file, and government Open Data portals.

All written programs make use of common functions grouped in a package called `utils`. These functions are: (1) `getConfig`, to retrieve the configuration file; (2) `getOpenData`, to retrieve data from the three different sources presented in Chapter 6 and return it in the form of pandas `DataFrame`; (3) `standardizeName`, which transforms strings into a standard format where only initials are capitalized, and apostrophe characters are converted to the relevant accented characters; (4) `genNameForID`, which generates an identifier from the name, removing special characters and using a hyphen as a separator. This function is used only in some cases where the name is known to be unique, such as for localities. In other cases, it is preferable to use identifiers such as vat number or a sequence number.

The full code of the RDF Graph Builder for the Comune di Sona is available on GitHub at `https://github.com/luca-martinelli-09/sona-lod`.

```
def getConfig(fileName):
  config = configparser.ConfigParser()
  config.read(fileName)

  return config

def getOpenData(resID, baseURL=None, whereSQL="", rawData=False, dtype=
    None, strip=True):
  config = getConfig('../../conf.ini')

  offline = False if baseURL else config.getboolean("API", "use_offline
    ")
  baseURL = baseURL if baseURL else config.get("API", "base_url")

  dataURI = "{}/api/3/action/datastore_search_sql?sql=SELECT * FROM
    \"{}\" {}".format(baseURL, resID, whereSQL) if not offline else "
    ../../off_data/{}".format(resID)

  if resID.startswith("http"):
    offline = False
    dataURI = resID

  if rawData:
    if offline:
      return dataURI

    getDataRequest = urlopen(dataURI)

    return getDataRequest

  df = None

  if offline:
    df = pd.read_csv(dataURI, dtype=dtype)
  else:
    if resID.endswith("csv"):
      df = pd.read_csv(dataURI, dtype=dtype)
    else:
      tries = 0
      res = {"success": False}
      while not res["success"] and tries < 20:
```

```
        res = requests.get(dataURI).json()
        if res["success"]:
          df = pd.DataFrame(res["result"]["records"], dtype=dtype)
          break
        tries += 1

  if strip and not df is None:
    df = df.applymap(lambda x: x.strip() if type(x) == str else x)

  return df

def standardizeName(name):
  name = name.lower().title()

  if name.endswith("a'"):
    name = name.removesuffix("a'") + "à"

  return name.strip()

def genNameForID(name):
  nameID = ""

  name.replace("'", "")
  name = unidecode.unidecode(name.lower())

  for char in name:
    nameID += char if char.isalnum() else "-"

  return nameID
```

Code 6.8: The common functions in the utils package.

6.2.1 Addresses

The addresses (streets and civic numbers) is the only semantic area that relies exclusively on the OntoPiA ontology. In particular, the address ontology is `CLV_AP-IT`, and was created in collaboration with ISTAT and the Agenzia delle Entrate. The ontology was developed to describe the data collected in the National Archive of Urban Street Numbers, or ANNCSU[4], a computerized repository

[4] `https://purl.archive.org/anncsu`

containing the street and house numbers of all Italian municipalities. Access to the archive should be public, but at the moment it is possible to download a municipality's data only through the dedicated portal of the Agenzia delle Entrate and reserved for the municipal contact person for toponymy, who is responsible for managing the updating of the data. From this dedicated portal it is possible to download the list of streets and the list of house numbers. The files are in CSV format, and follows a specific structure designed by the two agencies[5]. The information about the census sections are available on the ISTAT website, as kml file that defines the geographic geometries of each section.

The URL of the ANNCSU Dataset for the Comune di Sona is `https://w3id.org/sona/data/ANNCSU/`, so the resources will be organized in this Dataset and that URL will be their namespace. In detail:

Sona Is the municipality, for which `owl:sameAs` properties have been defined that link it to external resources concerning the city, such as dbpedia, ISPRA, and the Ministry of Cultural Heritage. The URL of this resource is `https://w3id.org/sona/data/ANNCSU/023083`, where `023083` is the ISTAT identifier for the city;

Census Sections The file of the census sections has been processed thanks to the `pykml` Python library, thanks to which it is possible to retrieve the section number and its geographic geometry. The URL of the resources is formatted as `https://w3id.org/sona/data/ANNCSU/cs/{id}`, where `id` is the number of the census section. The URL of the geometry resource that defines the geographical polygon of the census section is instead formatted as `https://w3id.org/sona/data/ANNCSU/gsc/{id}`;

Streets Names The street names are retrieved from the file of the streets provided by the Agenzia delle Entrate. The URL of the resources is formatted as `https://w3id.org/sona/data/ANNCSU/street/{id_street}`, where the `id_street` is the national identifier for the street, which can be retrieved by the field `PROGR_NAZIONALE` in the CSV file;

Civic Numbers As for street names, the civic numbers are retrieved from the file of house numbers provided by the Agenzia delle Entrate. The resources are

[5] `https://purl.archive.org/age-anncsu-specifiche`

located at `https://w3id.org/sona/data/ANNCSU/civic/{id_num}`, where `id_num` in this case is the field `PROGR_CIVICO`, and it is the national identifier of the house number;

Addresses Addresses are combination of street names and civic numbers. These are the resources part of the ANNCSU dataset, and their URLs are formatted as `https://w3id.org/sona/data/ANNCSU/ad-{id_street}-{id_num}`. For each street, in addition to house numbers, an address with `id_num` equals to `snc` is created, indicating the absence of a house number. Finally, for some addresses, thanks to OpenStreetMap[6], it was possible to estimate the location, which is described in geometry resources located at `https://w3id.org/sona/data/ANNCSU/gcn/{id_num}`.

Code 6.9 shows the part of the `config.ini` file relative to the ANNCSU, and the part of the RDF Graph Builder that inserts the streets toponyms into the graph.

```
# config.ini
[ANNCSU]
streets = 7b3401e5-7235-43d4-b8ed-8e11f30ec404
civics = e22af300-c934-4d7e-b84a-3f58e0862e4c
census_sections = 1f929641-0aa5-49b7-b012-6ea29ceca759
post_code = 37060

# rdf_builder_anncsu.py
config = getConfig("conf.ini")

anncsuAddresses = getOpenData(config.get("ANNCSU", "streets"))
anncsuAddresses.set_index("PROGR_NAZIONALE", inplace=True)

for streetID, address in anncsuAddresses.iterrows():
  dugName = standardizeName(address["DUG"])
  streetName = standardizeName(address["DENOM_COMPLETA"])

  fullName = "{} {}".format(
    standardizeName(address["DUG"]),
    standardizeName(address["DENOM_COMPLETA"]))

  streetToponym = StreetToponym(
    id="street/" + str(streetID),
```

[6] https://www.openstreetmap.org

```
    baseUri=ANNCSU,
    dataset=ANNCSU_DATASET,
    titles=[Literal(fullName, datatype=XSD.string)])

streetToponym.toponymQualifier = [Literal(dugName, datatype=XSD.
 string)]
streetToponym.officialStreetName = [Literal(streetName, datatype=XSD.
 string)]

streetToponym.addToGraph(g)
```

Code 6.9: The part of the RDF Graph Builder that inserts the streets toponyms into the graph, and the config.ini file relative to the addresses.

6.2.2 Organizations

As specified in Table 5.1, the information on private organizations comes from the Camera di Commercio. Some of this information, for privacy reasons, such as the tax code, cannot be published. However, thanks to the tax code it is possible to categorize businesses into women's, youth and foreign businesses, as specified in Section 5.2.3. It is therefore necessary, in this case, to retrieve the data from an offline file that contains all the information, and publish the censored version on Open Data portals.

The URL of the dataset of organizations situated in the Comune di Sona is `https://w3id.org/sona/data/organization/`, so the resources will be organized in this Dataset and that URL will be their namespace. Resources are identified by their vat number, which is unique to each enterprise. Only in the case of individuals for whom no vat number is indicated, the tax number is used.

In addition to enterprises, instances describing the online contact points of organizations have as their URL `https://w3id.org/sona/data/organization/ocp/{vat_num}`, which in turn link back to pecs located at URL `https://w3id.org/sona/data/organization/pec/{id_pec}`, where `id_pec` is generated with the `genNameForID` function described in Section 6.2.

As said in Section 5.2.3, the Camera di Commercio provides a list of all the organizations present in the city, both if they are local units and if they are the head office. The typology is specified in the CSV file in the field `UL-SEDE`. All the main offices of the organizations are then entered first. Next, the enterprises indicated as local units are processed and, if already present in the graph, the

`hasLocalUnitAddress` property is added along with their address. Otherwise, the organization is inserted without specifying the `hasPrimaryAddress` property, but only `hasLocalUnitAddress`.

Another problem encountered at this stage concerns the addresses of organizations, but also of other resources in other semantic areas, such as associations, schools, and municipal offices. The addresses, in fact, are present in text format and, often, do not correspond to the official addresses in the national archive (e.g., "Via Francesco Petrarca" is listed as "Via Petrarca") and do not have standard formatting. This is due to the fact that the address is given by the company and is not normalized by the Camera di Commercio. The goal is to retrieve from each address the street identifier and that of the house number (if any). To do this, the program makes use of the function shown in Code 6.10. This function retrieves the street directory from the files presented in Section 6.2.1, and compares the address to be searched against this list, returning a ranking of the most similar results. The `extract` function of the `rapidfuzz` library is used for this part. From here, the desired address must be manually selected, and the two searched identifiers will be returned.

```
def queryStreetCode(q):
  config = getConfig('../../conf.ini')

  streetsDF = getOpenData(config.get("ANNCSU", "streets")).set_index(["
   PROGR_NAZIONALE"])
  civicsDF = getOpenData(config.get("ANNCSU", "civics")).set_index(["
   PROGR_CIVICO"])

  streetsForSearchIDs = [(c["PROGR_NAZIONALE"], progrCivico) for
   progrCivico, c in civicsDF.iterrows()]
  streetsForSearch = ["{} {} {}{} {}".format(
    streetsDF.loc[c["PROGR_NAZIONALE"]]["DUG"],
    streetsDF.loc[c["PROGR_NAZIONALE"]]["DENOM_COMPLETA"],
    c["CIVICO"],
    "" if pd.isna(c["ESPONENTE"]) else c["ESPONENTE"],
    streetsDF.loc[c["PROGR_NAZIONALE"]]["LOCALITA'"],
  ).lower() for _, c in civicsDF.iterrows()]

  streetsForSearchIDs.extend([(progrNazionale, None) for progrNazionale
   , _ in streetsDF.iterrows()])
  streetsForSearch.extend(["{} {} {}".format(s["DUG"], s["
   DENOM_COMPLETA"], s["LOCALITA'"]).lower() for _, s in streetsDF.
```

```
    iterrows()])
18
19  searchResults = process.extract(q.lower(), streetsForSearch, scorer=
      fuzz.WRatio, limit=10)
20
21  print(f"\n\n[RESULTS FOR] {q}")
22  for res, val, i in searchResults:
23    print(f"{i}) {res} ({val})")
24
25  selectedResult = input("Choose one or type custom search: ")
26
27  if selectedResult.isnumeric():
28    return streetsForSearchIDs[int(selectedResult)]
29  elif selectedResult == "":
30    return None, None
31  else:
32    return queryStreetCode(selectedResult)
```

Code 6.10: The function that retrieve the street identifiers from the string of the address.

The function is then used as follows:

```
address = organizationInfo["INDIRIZZO"]
progrNazionale, progrCivico = queryStreetCode(address) if not pd.isna(
    address) else (None, None)
```

6.2.3 Schools

The list of schools present in the municipality is directly taken from the official data portal of the Ministero dell'Istruzione[7]. In this way, as with businesses, the data structure is standard nationwide, and the code can be easily reused without making further changes. From the portal are available both public and private schools, and the relative comprehensive institutes, for which the comprehensive institute code is the same as the school code. Other information regards the address, the email, and the website of the schools.

The URL of the dataset of schools situated in the Comune di Sona is `https://w3id.org/sona/data/school/`, so the resources will be organized in this Dataset and that URL will be their namespace. Resources are identified by the code

[7] https://dati.istruzione.it/opendata/

assigned by the Ministero dell'Istruzione, which is unique to each school. As for organizations, the online contact point are located at `https://w3id.org/sona/data/school/ocp/{id_school}`, while the email, the certified email, and the website are located at `https://w3id.org/sona/data/school/{contact_type}/{id_c}`, where `contact_type` is respectively `email`, `pec`, and `web`, and `id_c` is an identifier generated by the `genNameForID` function.

The same address considerations made for organizations in Section 6.2.2 also apply.

Finally, the typology of the school are mapped using a Python dictionary into the relatively identifiers of the controlled vocabulary described in Section 5.2.6

For what concerns demographic observations on the schoolchildren, the Ministero dell'Istruzione provides, for the public schools, information about the number of the children, both categorized by gender, by course, and by age.

In this case, the URLs of the observations resources are defined as follows: `https://w3id.org/sona/data/school/statistics/{id}`, where `id` is the sequence number of the data inserted. The demographic references are located at `https://w3id.org/sona/data/demographic-references/{id}`. In this case the `id` can be `M` for males, `F` for females, or `age-{age}` for the age. For what concerns the temporal entity that defines the period to which the observation refers, the resources of type `TemporalEntity` are located at `https://w3id.org/sona/data/ti/{sy}-{ey}`, where `sy` stands for the beginning year of the academic year, and `ey` for the end. Since in this case the temporal entity represents a period, the start date is fixed to `{sy}-09-01`, and the end date to `{ey}-07-01`.

Code 6.11 shows the part of the code that build the RDF Graph for demographic observations on the schools' schoolchildren.

```
for (schoolCode, academicYear), statsInfo in sumDataDF.iterrows():
  academicYear = int(str(academicYear)[:4])

  school = School(
      id=schoolCode,
      baseUri=SCHOOL_DATA
  )

  schoolInfo = schoolsDF.loc[schoolCode]

  temporalEntity = TimeInterval(
      id="ti/{}-{}".format(academicYear, academicYear + 1),
```

```
13        baseUri=SCHOOL_DATA,
14        dataset=SCHOOL_DATASET,
15        titles=[Literal("{}/{}".format(academicYear, academicYear + 1),
      datatype=XSD.string)]
16    )
17    temporalEntity.startTime = Literal(str(academicYear) + "-09-01",
      datatype=XSD.date)
18    temporalEntity.endTime = Literal(str(academicYear + 1) + "-07-01",
      datatype=XSD.date)
19    temporalEntity.addToGraph(g)
20
21    for sexCode in ["M", "F"]:
22      subscribers = Subscribers(
23        id="statistics/{}-{}-{}".format(schoolCode, sexCode, academicYear
      ),
24        baseUri=SCHOOL_DATA,
25        dataset=SCHOOL_DATASET,
26        titles=[
27          Literal("{} - {} - A.A. {}/{}".format(
28            "Alunni (maschi)" if sexCode == "M" else "Alunne (femmine)",
29            standardizeName(schoolInfo["DENOMINAZIONESCUOLA"]),
30            academicYear,
31            academicYear + 1
32          ), datatype=XSD.string)
33        ]
34      )
35
36      demReference = AlivePerson(
37        id="demographic-reference/" + sexCode,
38        baseUri=SCHOOL_DATA,
39        dataset=SCHOOL_DATASET,
40        titles=[
41          Literal("Male" if sexCode == "M" else "Female", lang="en"),
42          Literal("Maschio" if sexCode == "M" else "Femmina", lang="it")
43        ]
44      )
45      demReference.hasSex = Sex(id=sexCode, baseUri=PERSON_SEX)
46      demReference.addToGraph(g)
47
48      subscribers.hasDemographicReference = demReference
49      subscribers.hasTemporalEntity = temporalEntity
50
51      subscribers.observationValue = Literal(statsInfo["ALUNNI" + ("
```

```
    MASCHI" if sexCode == "M" else "FEMMINE")], datatype=XSD.
    positiveInteger)
52
53   subscribers.addToGraph(g)
54
55   school.hasSubscribers = [subscribers]
56   school.addToGraph(g, onlyProperties=True)
```

Code 6.11: The part of the code that build the RDF Graph for demographic observations on the school.

7 Web Applications

The third and final key part of this thesis project concerns the web applications that allow on the one hand to download and, for the municipality, upload Open Data, and on the other hand to visualize Linked Open Data clearly through summary tables and graphs. The first web application is CKAN, and it is described in Section 7.1. In this case, the work was to configure the open source portal CKAN, described in Section 7.1, with the necessary plugins and extensions to follow the "Linee Guida Nazionali per la Valorizzazione del Patrimonio Informativo Pubblico".[1] The second web application, described in Section 7.2, was built from scratch and is intended to retrieve data from the SPARQL endpoint and display it graphically in the form of graphs, maps and tables.

7.1 CKAN

As said in Chapter 6, the RDF Graph Builder get the data from three different sources, and one of these sources is the CKAN Open Data Portal of the city (Comune di Sona in this case). The main idea is to use this open source portal in order to facilitate the publication of data on the web and be aligned with other Italian and foreign cities. At the same time, this portal provides the catalog compliant with the `DCAT-AP_IT` metadata profile, and this allows published data to be made available in regional, national and European portals as well. In

[1] `https://docs.italia.it/italia/daf/lg-patrimonio-pubblico/it/stabile`

addition, the presence of data in Linked Open Data format does not preclude the use of the CKAN portal, despite the fact that the latter's data achieve a three-star rating according to Tim Berners-Lee's classification introduced in Section 2.2. In fact, in addition to having a lower management cost [BK11], this portal allows the publication of data that in Linked Open Data format would not be publishable, or for which an ontology describing them has not yet been developed. It also allows original resources to be reused for other tasks, either by the municipality or by companies or citizens.

The entire project is available under an open source license on GitHub at the link `https://github.com/luca-martinelli-09/ckan`. The final result built for the Comune di Sona is shown in Figure 7.1. For ease of installation and deployment, moreover, the portal has been containerized to produce a Docker[2] image.

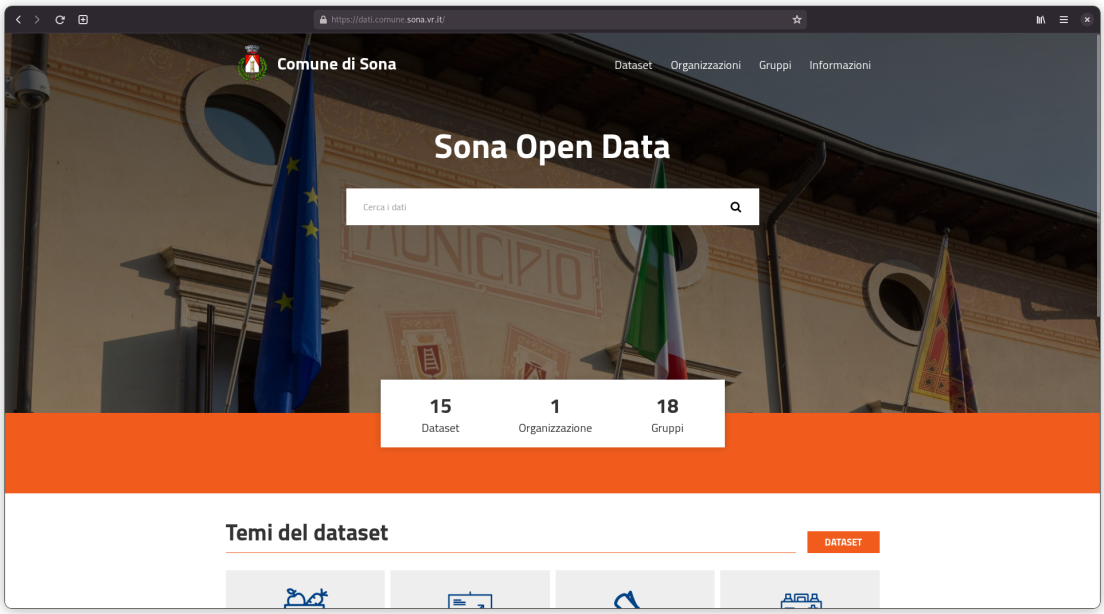


Figure 7.1: The CKAN Open Data portal for the Comune di Sona.

The Open Data portal extends the CKAN 2.9 Docker image provided by the Open Knowledge Foundation,[3] installing the required plugins.

[2] `https://www.docker.com/`
[3] `https://github.com/okfn/docker-ckan`

7.2 Data Reports

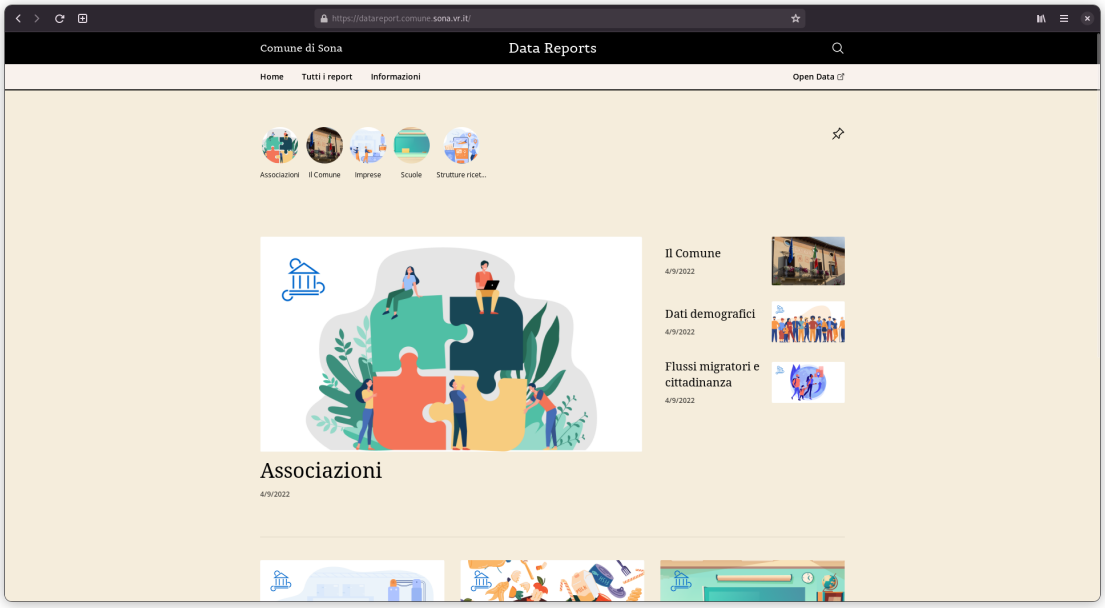


Figure 7.2: The CKAN Open Data portal for the Comune di Sona.

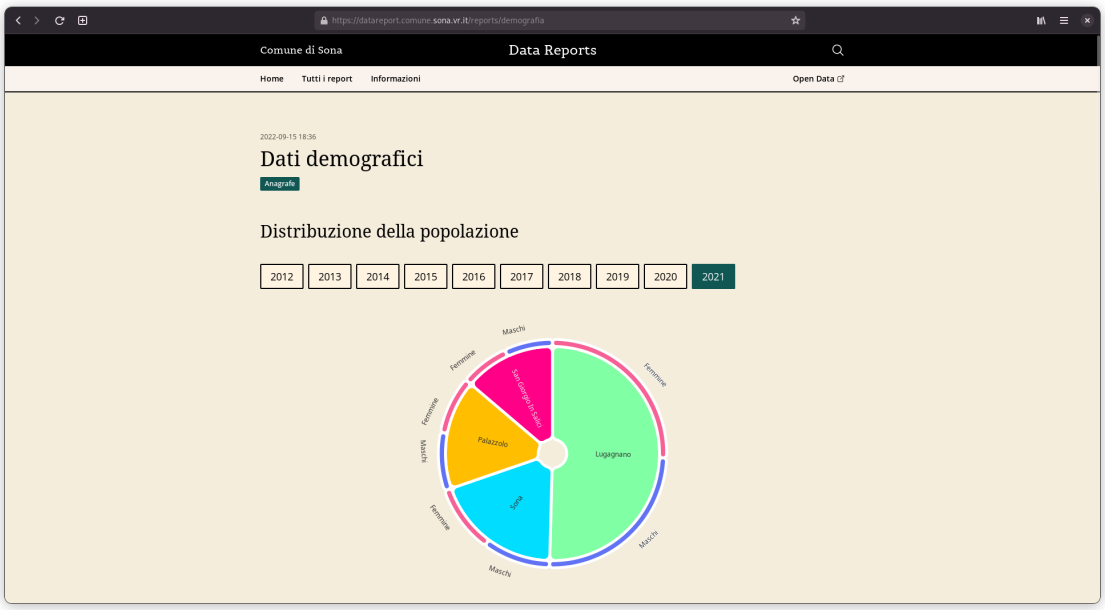


Figure 7.3: The CKAN Open Data portal for the Comune di Sona.

8

Conclusions and Future Works

References

[Gru95] Thomas R Gruber. "Toward principles for the design of ontologies used for knowledge sharing?" In: *International journal of human-computer studies* 43.5-6 (1995), pp. 907–928.

[BHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. "The semantic web". In: *Scientific american* 284.5 (2001), pp. 34–43.

[NM+01] Natalya F Noy, Deborah L McGuinness, et al. *Ontology development 101: A guide to creating your first ontology*. 2001.

[Gen+03] John H Gennari et al. "The evolution of Protégé: an environment for knowledge-based systems development". In: *International Journal of Human-computer studies* 58.1 (2003), pp. 89–123.

[Bec+04] Sean Bechhofer et al. "OWL web ontology language reference". In: *W3C recommendation* 10.2 (2004), pp. 1–53.

[Ber06] Tim Berners-Lee. "Linked Data - Design Issues". In: (July 2006). URL: `http://www.w3.org/DesignIssues/LinkedData.html`.

[Fei+07] Lee Feigenbaum et al. "The semantic web in action". In: *Scientific American* 297.6 (2007), pp. 90–97.

[EMS08] Jérôme Euzenat, Adrian Mocan, and François Scharffe. "Ontology alignments". In: *Ontology Management*. Springer, 2008, pp. 177–206.

[Hit+09] Pascal Hitzler et al. "OWL 2 web ontology language primer". In: *W3C recommendation* 27.1 (2009), p. 123.

[Tay10] Mohammad Mustafa Taye. "Understanding semantic web and ontologies: Theory and applications". In: *arXiv preprint arXiv:1006.4567* (2010).

[BK11] Florian Bauer and Martin Kaltenböck. "Linked Open Data: The essentials". In: *Edition mono/monochrom, Vienna* 710 (2011).

[BHB11] Christian Bizer, Tom Heath, and Tim Berners-Lee. "Linked data: The story so far". In: *Semantic services, interoperability and web applications: emerging concepts*. IGI global, 2011, pp. 205–227.

[Con+13] World Wide Web Consortium et al. "SPARQL 1.1 Overview". In: (Mar. 2013). URL: https://www.w3.org/TR/sparql11-overview/.

[Con+14a] World Wide Web Consortium et al. *RDF 1.1 Concepts and Abstract Syntax*. Feb. 2014. URL: https://www.w3.org/TR/rdf11-concepts/.

[Con+14b] World Wide Web Consortium et al. "RDF 1.1 Primer". In: (June 2014). URL: https://www.w3.org/TR/rdf11-primer/.

[IB14] Mirjana Ivanović and Zoran Budimac. "An overview of ontologies and data resources in medical domains". In: *Expert Systems with Applications* 41.11 (2014), pp. 5158–5166.

[Dig17a] Agenzia per l'Italia Digitale. "Linee guida nazionali per la valorizzazione del patrimonio informativo pubblico". In: (2017). URL: https://docs.italia.it/italia/daf/lg-patrimonio-pubblico/.

[Dig17b] Agenzia per l'Italia Digitale. "Piano triennale per l'informatica nella Pubblica amministrazione 2017-2019". In: (2017). URL: https://docs.italia.it/italia/piano-triennale-ict/pianotriennale-ict-doc/it/2017-2019/.

Acknowledgments

No thanks