



D3.3 TOOL ARCHITECTURE AND COMPONENTS/PLUGINS PROGRAMMING STATUS REPORT 1

PROJECT

Acronym: **OpenDataMonitor**
Title: Monitoring, Analysis and Visualisation of Open Data Catalogues, Hubs and Repositories
Coordinator: SYNYO GmbH

Reference: **611988**
Type: Collaborative project
Programme: FP7-ICT

Start: November 2013
Duration: 24 months

Website: <http://project.opendatamonitor.eu>
E-Mail: office@opendatamonitor.eu

Consortium: **SYNYO GmbH**, Research & Development Department, Austria, (SYNYO)
Open Data Institute, Research Department, UK, (ODI)
Athena Research and Innovation Center, IMIS, Greece, (ATHENA)
University of Southampton, Web and Internet Science Group, UK, (SOTON)
Potsdam eGovernment Competence Center, Research Department, Germany, (IFG.CC)
City of Munich, Department of Labor and Economic Development, Germany, (MUNICH)
Entidad Publica Empresarial Red.es, Shared Service Department, Spain, (RED.ES)

DELIVERABLE

Number:	D3.3
Title:	Tool architecture and components/plugins programming status report 1
Lead beneficiary:	ATHENA
Work package:	WP3: Concept Design and Software Development
Dissemination level:	Public (PU)
Nature:	Report (RE)
Due date:	October 31, 2014
Submission date:	October 30, 2014
Authors:	Vassilis Kaffes, ATHENA Dimitris Skoutas, ATHENA Thodoris Raios, ATHENA
Contributors:	Ejona Sauli, SYNYO Michael Heil, SYNYO Elena Simperl, SOTON Yunjia Li, SOTON
Reviewers:	Tom Heath, ODI Amanda Smith, ODI Bernhard Jäger, SYNYO Peter Leitner, SYNYO

Acknowledgement: The OpenDataMonitor project is co-funded by the European Commission under the Seventh Framework Programme (FP7 2007-2013) under grant agreement number 611988.

Disclaimer: The content of this publication is the sole responsibility of the authors, and in no way represents the view of the European Commission or its services.

TABLE OF CONTENTS

1	Introduction	6
1.1	Scope and purpose	6
1.2	Context within the ODM project.....	6
2	Tool Architecture.....	8
2.1	Architecture Overview	8
2.2	Processing Workflow.....	10
3	Implementation of Components	12
3.1	Catalogue Registry.....	12
3.2	Job Manager.....	16
3.3	Metadata Harvester	19
3.4	Metadata Repository.....	21
3.5	Harmonisation Engine	22
3.6	Analysis Engine	26
3.7	Administration Panel.....	29
3.8	Demonstration Site	30
4	Conclusions and Next Steps	31
5	References.....	33
6	APPENDIX.....	34
6.1	Example of harvesting job configuration	34
6.2	Example of raw collected metadata.....	35
6.3	Dictionary for harmonising licences.....	40

LIST OF FIGURES

Figure 1: Overview of architecture and processing workflow	8
Figure 2: New catalogue registration form	13
Figure 3: Example illustrating an HTML page displaying the information about a dataset	15
Figure 4: Filling in information for metadata extraction from HTML pages	16
Figure 5: List of catalogues registered for harvesting	18
Figure 6: Display status of harvesting job	18

LIST OF TABLES

Table 1: Context of current deliverable	6
Table 2: Internal metadata schema	23
Table 3: Snippet of the response of the API method <code>_commands</code>	27
Table 4: Example of harvesting job configuration for the catalogues <code>publicdata.eu</code> and <code>datos.gob.es</code> 34	
Table 5: JSON document containing the metadata of a dataset returned by the CKAN Harvester	35
Table 6: JSON document containing the metadata of a dataset returned by the HTML Harvester	38
Table 7: Dictionary for licences cleaning and harmonisation	40

1 INTRODUCTION

1.1 Scope and purpose

The main driving idea behind the open data movement, which has been increasingly gaining momentum over the past years, is that certain data should be freely available and usable for everyone. The importance of this is widely recognised especially for open government data (OGD), the availability of which may significantly contribute to the transparency and efficiency of a democratic system. More generally, the availability and use of open data can have a positive impact on society as a whole, stimulating economic and business growth.

The goal of the OpenDataMonitor (ODM) project is to make it possible for interested stakeholders to gain an overview of this evolving open data “landscape”. More specifically, ODM aims to achieve this goal by designing and developing:

- an extensible and customizable harvesting framework, for facilitating and automating as much as possible the collection of metadata from diverse open data catalogues
- an integration and harmonisation workflow, for overcoming the high heterogeneity of schemas, values and formats found in the various open data sources
- scalable analytical and visualisation methods, for allowing end users to explore the results in an intuitive and user-friendly manner and obtain a comprehensive overview of the collected information.

The purpose of this document is to present the overall architecture of the ODM platform, and to report the status of the implementation for each of the main components involved. In particular, Section 2 describes the design of the architecture for the ODM platform and explains the processing workflow for collecting, integrating, analysing and visualizing metadata from various open data catalogues. Then, Section 3 delves into each main component in more detail and presents the current status of its implementation. Section 4 summarizes the progress of the work and presents the next steps.

1.2 Context within the ODM project

The following table summarizes how this document is related to other deliverables in the project.

Table 1: Context of current deliverable

Deliverable	Description
D2.1: Open data topologies, catalogues and metadata harmonisation.	D2.1 provides, among others, an overview of the open data landscape from a technical perspective, in terms of existing open data portals, software and APIs, as well as existing

<i>(June 2014)</i>	metadata standards and mappings among them. These findings have been taken into consideration for identifying the challenges and needs regarding metadata collection and harmonisation, and thus making decisions for the design and functionalities of the respective components.
D2.2: Monitoring methods, architectures and standards analysis report. <i>(July 2014)</i>	D2.2 has provided an overview and comparison of existing open data comparison platforms, catalogues and portals. This has been used to identify novel features and functionalities to be implemented in the ODM platform. Furthermore, it has performed a state-of-the-art analysis of existing methods and software tools for the various steps of the process, i.e. metadata collection, storage and integration, which has provided the foundation for implementing the ODM tools as described in the current document.
D2.3: Best practice visualisation, dashboard and key figures report. <i>(July 2014)</i>	D2.3 defines the metrics to be computed over the collected and processed metadata, and discusses different options and methods for visualizing the results to the end users. Hence, the work on implementing the analysis engine and the demonstration site is based on that report.
D2.5: Open data resources, platforms and APIs collection 1 <i>(September 2014)</i>	Related to, and in conjunction with, D2.1.
D3.1: Development of a scalable open data monitoring concept and framework design. <i>(May 2014)</i>	This report was the first to provide an architecture overview of the ODM platform and to identify the main components needed, hence it constitutes the starting point and the basis for the implementation work described here.
D3.2: Tool specifications, use cases, mockups and functionalities status report 1 <i>(July 2014)</i>	D3.2 is a complementary document to the current one, focusing rather at the system level and the product perspective, describing overall requirements and functionalities, user roles and interfaces, while the current document details the implementation status of individual components.
D3.4: Visualisations, dashboards and multilingual interface status report 1 <i>(October 2014)</i>	The work reported in D3.4 is conducted in parallel with that of D3.3, with the former focusing on the implementation of visualisations and dashboard on the demonstration site.
D3.6: Tool architecture and components/plugins programming status report 2 <i>(August 2015)</i>	Developing the components describing in the current document is an ongoing process, with steps often done iteratively to improve and/or extend implemented functionalities according to testing and needs that arise. D3.6 constitutes the second part of this report, which will conclude the progress of the work at the end of the project.
D4.1: Deployment of the tool demonstrator finished. <i>(November 2014)</i>	The components which are currently under implementation as described in this document will be included in the first demonstration of the tool that will be presented in D4.1

2 TOOL ARCHITECTURE

2.1 Architecture Overview

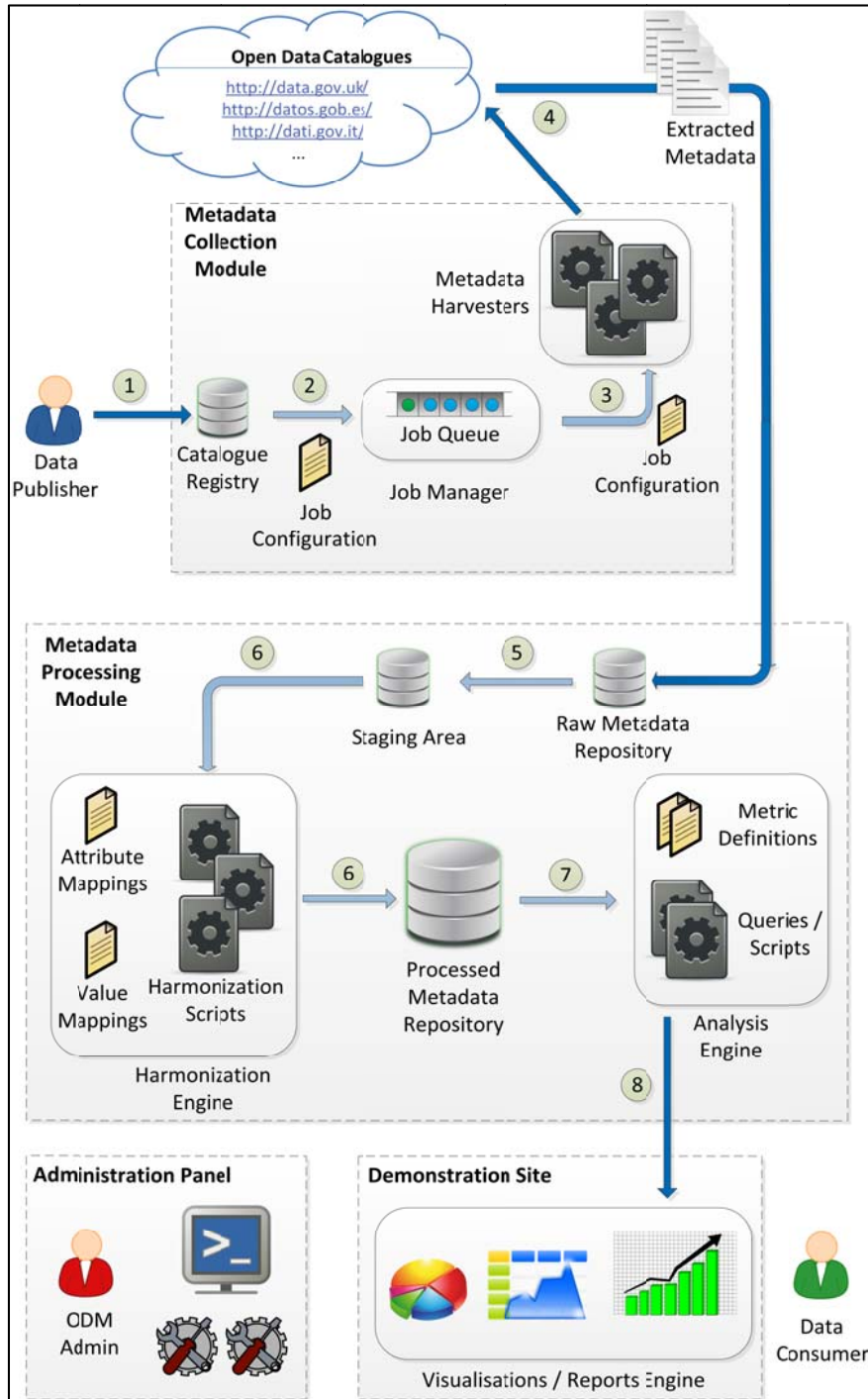


Figure 1: Overview of architecture and processing workflow

Figure 1 provides an overview of the ODM platform architecture, illustrating the main components and indicating the steps of the metadata processing workflow¹. In particular, the ODM platform comprises the following main modules and components:

- **Metadata Collection Module.** This part of the system is responsible for the task of metadata collection from various open data catalogues. It further consists of the following components:
 - **Catalogue Registry.** This component allows data publishers (or the ODM administrator) to register catalogues for harvesting and monitoring. Registration is done via a Web-based User Interface (UI), where a form is filled in with some basic information about the registered catalogue (e.g., title, URL), as well as some additional information that is needed in order to setup and configure an appropriate harvesting job for this catalogue. This provided information forms the catalogue profile and is stored in the catalogue registry.
 - **Job Manager.** A harvesting job represents a task for harvesting metadata from a registered open data catalogue. In particular, it provides the required configuration that drives the harvesting process (e.g., which harvester to use, metadata extraction rules to be applied). Harvesting jobs are maintained in a queue and are scheduled for processing. This is handled by the Job Manager, which schedules the execution of jobs, periodically or on demand, and is responsible for monitoring their process and reporting the status of execution.
 - **Metadata Harvesters.** These are scripts executed by harvesting jobs in order to perform the actual extraction of metadata from the respective catalogue. Different harvesters may be implemented and used to address the different open data platforms and APIs that exist. In this case, the configuration included in the harvesting job specifies which harvester should be used and how. The main challenge here is to deal with the trade-off between keeping the harvesters as generic as possible and adapting to the specificities of each harvested catalogue.
- **Metadata Processing Module.** This part of the system is responsible for cleaning, integrating and analysing the metadata that are retrieved from the various catalogues that have been registered. It consists of the following components:
 - **Harmonisation Engine.** This component processes the raw, original metadata that were retrieved by the harvesters and performs cleaning and integration tasks required to obtain a homogenized dataset in terms of both attribute names and attribute values.
 - **Analysis Engine.** Once the collected metadata have been mapped to a consistent internal schema and representation, the analysis engine performs the required operations (e.g. aggregations) in order to compute the metrics that have been

¹ This is an updated and elaborated version of the initial architecture design presented in D3.1.

defined for monitoring. It also makes these results available to the demonstration site for visualisation and presentation to the end users.

- **Demonstration Site.** This module comprises several components for generating intuitive visualisations and reports that are presented to the end users, allowing them to obtain a comprehensive overview of trends in the evolving open data landscape, based on the monitored open data catalogues.
- **Administration Panel.** This module comprises a set of dashboards that allows the ODM system administrator to monitor, control and configure various aspects of the system's operation (e.g., configure options for metadata collection, monitor the status of harvesting jobs, define rules for metadata harmonisation, specify templates for visualisations).

This report focuses in more detail on the first two modules, i.e. the metadata collection and metadata processing. A more detailed description of the demonstration site, and a report on the status of its implementation, is provided in D3.4. Furthermore, the administration panel will be described in the second version of this report (i.e., D3.6), since this component is not part of the main processing workflow but its purpose is instead to support the ODM system administrator, and hence has not been the focus of the implementation so far.

2.2 Processing Workflow

In the following, we describe the main steps of the processing workflow. These steps are also illustrated in Figure 1.

Step 1: Catalogue registration. The first step of the process is to register a new open data catalogue for monitoring. This can be done by the ODM system administrator or other users that have the “data publisher” role. It is done via a Web-based UI, which presents a form requesting several attributes that have to be filled in to indicate the profile of the catalogue and guide the metadata collection process.

Step 2: Creation of harvesting job. Once a new catalogue is registered for monitoring and its profile is filled in, a corresponding harvesting job is created, configured and submitted to the *Job Manager*. The *Job Manager* inserts the job in the queue and schedules it for execution.

Step 3: Triggering of harvesting job. Periodically and/or on demand (as specified during a catalogue's registration), the *Job Manager* dequeues a harvesting job and initiates its execution. This is done by invoking the appropriate *Metadata Harvester* and using the configuration properties specified in the description of the job.

Step 4: Metadata extraction. The invoked *Metadata Harvester* connects to the corresponding open data catalogue and applies the configured extraction rules to retrieve the relevant metadata. The extracted metadata are stored in the *Raw Metadata Repository*. During this step, some preliminary

actions for cleaning and integrating the metadata also take place. For example, by applying the specified extraction rules, some of the collected metadata are already mapped to the internal representation.

Step 5: Staging of collected metadata. The raw collected metadata are heterogeneous and hence need to undertake a series of cleaning and harmonisation operations before they become available for further analysis and use. Nevertheless, for provenance reasons, it is desirable to also keep the original metadata. For example, this can be useful if needed to trace back the initial form of a processed item or if some steps of the cleaning and harmonisation need to be re-executed (e.g. because new/improved cleaning or harmonisation rules have been configured). Thus, before further processing takes place, the collected metadata are moved to the *Staging Area*.

Step 6: Metadata cleaning and harmonisation. Once moved to the staging area, a series of cleaning and harmonisation operations is executed in order to transform the initial metadata to a consistent, internal representation. This applies to both attribute names and values, and involves tasks such as mapping attribute names from other schemas to the internal one, validating and normalising different date formats, normalising names of file formats, licence titles, etc. The final results are stored in the *Processed Metadata Repository*.

Step 7: Metadata analysis. After the cleaning and integration steps have been performed, the metadata become available to the *Analysis Engine*. This applies the necessary aggregations or other computations to calculate the key metrics that have been defined for monitoring.

Step 8: Accessing the results. Finally, the results are made available through an API to other components, in particular to the *Demonstration Site*, which produces various charts, visualisations and reports for the end user. It is possible to request either the metadata records themselves (e.g. return all the metadata of the datasets in a given catalogue) or compute aggregate results for various metrics (e.g. return the number of datasets uploaded in the previous month).

3 IMPLEMENTATION OF COMPONENTS

In this section, we describe each component in more detail and we report on the current status of its implementation.

3.1 Catalogue Registry

The *Catalogue Registry* allows the ODM administrator or a user who has the “data publisher” role to register new open data catalogues for monitoring. For this purpose, we have implemented a Web-based UI that allows the user to provide the necessary information for registering the catalogue and configuring the corresponding harvesting process for it. For the latter, the information to be provided depends on the type of harvester that will be used for collecting metadata from this catalogue. As will be explained in more detail in Section 3.3, we have currently implemented two different harvesters, one targeting catalogues deployed on a CKAN platform² and one relying on HTML scraping that serves as a general-purpose harvester to handle the remaining cases. This distinction is reflected in the catalogue registration process, as will be shown below.

Moreover, instead of implementing this component as a stand-alone Web application, we have integrated it in a customized CKAN instance. This has the benefit that we can reuse infrastructure and functionalities already provided by CKAN³. For example, during the implementation and testing stage, the collected metadata are also stored in the internal database of this CKAN instance, thus having a basic Web UI to search and explore the collected content.

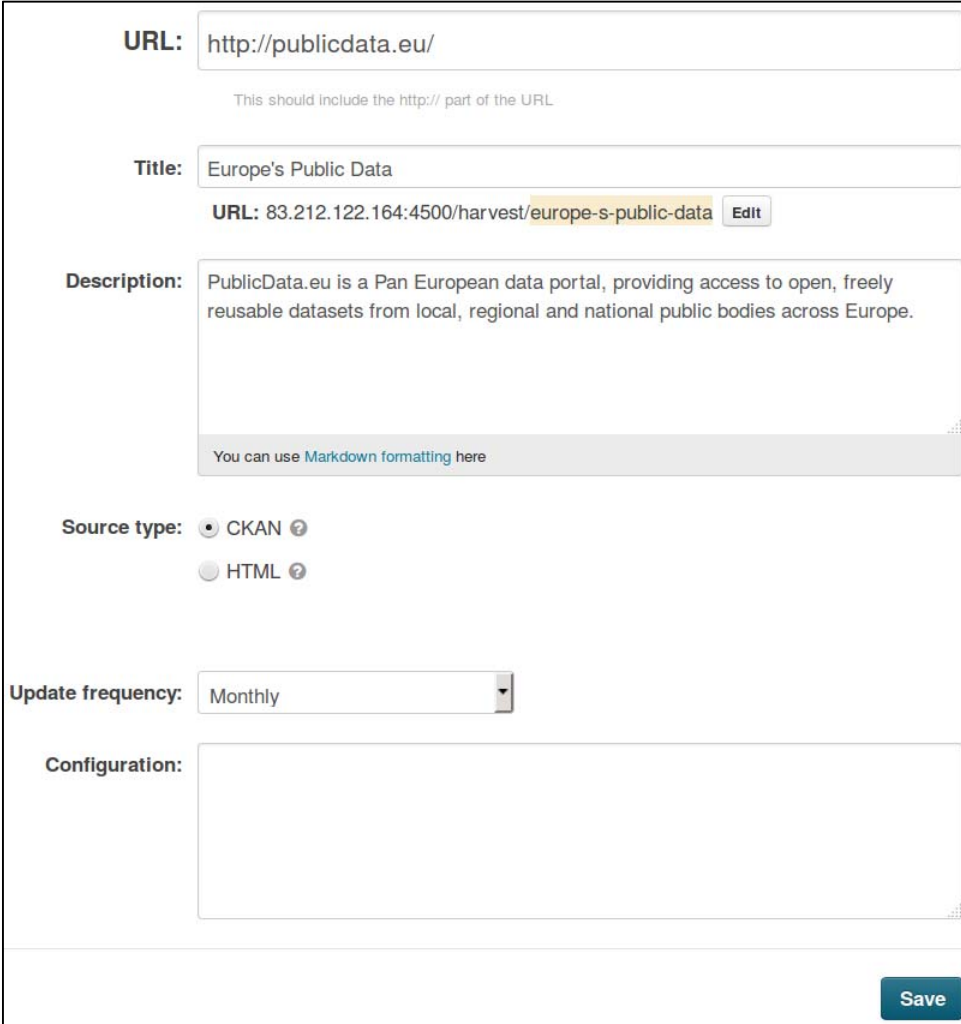
Figure 2 displays a screenshot of the Web form used to register a new catalogue. In the example shown, the URL of the catalogue to be monitored is <http://publicdata.eu/>. Moreover, a title and a description for this catalogue are given. Then the type of harvester to be used is specified. In this case, the catalogue is deployed on CKAN, hence this choice is selected, which means that the CKAN harvester will be used for metadata collection. The update frequency indicates whether a harvesting job for this catalogue should be configured to be run periodically, and how often. The available options are “monthly”, “weekly”, “daily” and “manually”. The latter means that the job is not scheduled to run periodically but it is triggered manually instead by the ODM administrator. Currently, since the *Job Manager* is not yet fully implemented, this is the only option supported in practice. Finally, the Configuration attribute applies to the ‘CKAN’ type. As will be explained in Section 3.3, the CKAN harvester relies on the `ckanext-harvest` extension⁴, a remote harvesting extension for CKAN, which supports a number of configuration options to control the harvester’s

² <http://ckan.org/>

³ The rationale for following a CKAN-oriented approach in this project, as has been explained in previous deliverables (e.g. D3.1), is based on the fact that it is open source, has a rich set of features, and it seems to be the most widely used platform among the existing ones, with an even growing trend.

⁴ <https://github.com/ckan/ckanext-harvest>

behaviour. These are defined as a JSON (JavaScript Object Notation) object, which can be entered in this field. If the field is left empty, the default configuration is used.



The screenshot shows a web form for registering a new catalogue. It contains the following elements:

- URL:** A text input field containing "http://publicdata.eu/". Below it is a small grey box with the text "This should include the http:// part of the URL".
- Title:** A text input field containing "Europe's Public Data". Below it is a smaller text input field containing "URL: 83.212.122.164:4500/harvest/europe-s-public-data" and an "Edit" button.
- Description:** A large text area containing the text "PublicData.eu is a Pan European data portal, providing access to open, freely reusable datasets from local, regional and national public bodies across Europe." Below the text area is a grey box with the text "You can use [Markdown formatting](#) here".
- Source type:** Two radio buttons: "CKAN" (selected) and "HTML".
- Update frequency:** A dropdown menu currently set to "Monthly".
- Configuration:** A large empty text area for entering JSON configuration.
- Save:** A blue button at the bottom right of the form.

Figure 2: New catalogue registration form

When registering a non-CKAN catalogue, some additional information needs to be provided. This is needed because in this case the harvesting is done via the *HTML Harvester*, which relies on HTML scraping. Hence, this information is required in order to configure appropriate extraction rules to be applied when parsing the HTML code of a Web page displaying the dataset. This is done again by filling in a Web form, according to the process explained below.

Typically, in an HTML page, the information about a dataset is presented in a structured manner using lists and/or tables. Figure 3 illustrates an example of such an HTML page displaying a randomly selected dataset taken from the catalogue datos.gob.es⁵. The information comprises the title of the

⁵ <http://datos.gob.es/>

dataset, displayed at the top of the page, a description, the publisher, category and tags, contained files, dates, etc. In most cases, labels are used to indicate what the information refers to, e.g. “Descripción”, “Publicador”, “Categorías”, “Fecha de creación”, etc. Note that since all these HTML pages in a catalogue are produced automatically using the same template, these labels are the same for all the set of pages within the same catalogue. Thus, during the registration process, for each attribute to be extracted, the corresponding label used in this catalogue is indicated. Still, there are also attributes for which no label is used. For example, the dataset’s title in this case is visually distinguished due to its position and size, instead of using a label such as “Title” as a prefix. This is addressed as follows. When filling in the registration form for this catalogue, the user indicates that there is no label for the attribute “title”. Instead, the user selects a random HTML page presenting a dataset, and provides in the form the URL of this page and the value of the title attribute found there (in this example: “Afecciones Importantes de Tráfico”). As will be described in Section 3.3, the *HTML Harvester* uses then this information to create an extraction rule for extracting the titles of the datasets from the rest of the HTML pages found in this catalogue.

Figure 4 displays a part of the form that is filled in when registering a non-CKAN catalogue. For each attribute, the user indicates whether a label is shown in the HTML page or not. If so, the name of the label is indicated, otherwise the value of the item in a randomly selected HTML page is inserted.

To further reduce the burden for the user who registers a new catalogue, we have implemented the following mechanism. Assume the case of the aforementioned example. Once this catalogue is registered, the system “remembers” that the label “Publicador” has been used for the attribute “Publisher”. Then, during the registration of a new (non-CKAN) catalogue, as soon as the user provides the URL of a sample HTML page of that catalogue, the system first recalls all the labels that have been used so far for this attribute, and if any of these labels is found in the HTML page, the corresponding field in the form is filled in automatically. Consequently, the user only needs to check and modify those fields for which the system’s “guess” was not correct, and also to fill in any remaining ones for which the system was not able to find any previously known label.

Once the process is finished, the information is stored and a harvesting job is created. This is stored in the form of a JSON document in a MongoDB database. The result for the presented examples, publicdata.eu and datos.gob.es, is listed in Table 4. Notice that in the latter case, where the harvesting is performed by the *HTML Harvester*, two JSON documents are generated. The first one is analogous to that for the catalogue publicdata.eu, with the difference `"type": "html"` instead of `"type": "ckan"`, while the second one contains the instructions that will guide the metadata extraction for each attribute.

Afecciones Importantes de Tráfico

Descripción:
Relación de calles o viales de Zaragoza afectadas por alguna incidencia. Se describe el tramo afectado, posibles desvíos, las fechas de inicio y finalización estimada, el motivo y otras observaciones.

-  **Publicador:** Ayuntamiento de Zaragoza (OTROS)
-  **Condiciones de reutilización:** > <http://www.zaragoza.es/ciudad/servicios/avisolegal.htm#condiciones>

Categorías: Transporte

Etiquetado como: movilidad urbana vía pública Tráfico incidencias

Distribuciones

Nombre	Acceso	Formato / Tamaño	Accesos + info
Afecciones Importantes de Tráfico		 JSON	0
Afecciones Importantes de Tráfico		 RSS	0

Información Adicional

 Fecha de creación:	Viernes, 12 Septiembre, 2014
 Fecha de última actualización:	Viernes, 12 Septiembre, 2014
 Frecuencia de actualización:	Diaría
 Idioma del conjunto de datos:	Español

Figure 3: Example illustrating an HTML page displaying the information about a dataset

Publisher:	<input type="text" value="Publicador:"/>
Feature:	<input type="text" value="Label"/>
Geographic Coverage:	<input type="text" value="Cobertura geográfica:"/>
Feature:	<input type="text" value="Label"/>
Temporal Coverage:	<input type="text" value="Cobertura temporal:"/>
Feature:	<input type="text" value="Label"/>
Release Date:	<input type="text" value="Fecha de creación:"/>
Feature:	<input type="text" value="Label"/>
Contact Point:	<input type="text"/>
Feature:	<input type="text" value="Value"/>
Keywords/Tags:	<input type="text" value="Etiquetado como:"/>
Feature:	<input type="text" value="Label"/>

Figure 4: Filling in information for metadata extraction from HTML pages

3.2 Job Manager

The *Job Manager* is the component responsible for scheduling, initiating and monitoring harvesting jobs. At the current stage of the implementation, the execution of a harvesting job is triggered manually; however, in future versions of the system this will be included in a background process that will schedule jobs to run periodically, according to the periodicity specified when registering a catalogue (e.g., monthly).

Similarly to the *Catalogue Registry*, the *Job Manager* is integrated in the customised CKAN instance and can be accessed there; moreover, the component is designed to reuse the same queue that the `ckanext-harvest` extension uses when managing jobs that collect metadata from other CKAN

catalogues. That is, all harvesting jobs are represented and stored in the same way, and they are inserted in the same queue for execution; then, when a job is triggered, it is checked whether the CKAN or the HTML harvester should be invoked to carry out the task.

Figure 5 shows the list of catalogues registered for harvesting. On the left side, the facets “frequency” and “type” are provided to make it easier to navigate through the list. Once a catalogue is selected, the status of the harvesting job can be displayed. This includes the date when the job was first created, when the last execution was started and finished, and the current status. It also displays a summary of results. An example is shown in Figure 6, which refers to the catalogue `publicdata.eu`. Notice that during the execution of a harvesting job, the process may be interrupted on purpose or accidentally for various reasons (e.g., due to a network failure). In this case, it is possible to resume the process later, continuing from the same point rather than starting a new job from the beginning. Thus, the execution of a job may span over several days, which is in fact the case in this example.

In the case of the *CKAN Harvester*, the aforementioned functionality is provided by the `ckanext-harvest` extension itself. In the case of the *HTML Harvester*, this has been implemented as an additional feature in order to avoid re-harvesting a catalogue from scratch when an unexpected error occurs. More specifically, this is achieved as follows. When the *HTML Harvester* starts collecting metadata from a catalogue, it creates a file in which it writes the progress of the process. This is measured in terms of “pages” that have been processed. Notice that, since a catalogue typically contains a large number of datasets, it employs paging to allow navigating through them. For example, in the case of the catalogue `publicdata.eu`, one can navigate directly to the n -th page of the listed datasets by using the URL: `http://publicdata.eu/dataset?page=n`. Leveraging this mechanism, when a page is fully harvested, it is logged in the aforementioned text file. Thus, if the job is interrupted and then resumed, it first checks this file (if it exists), and continues from the page after the one noted there.

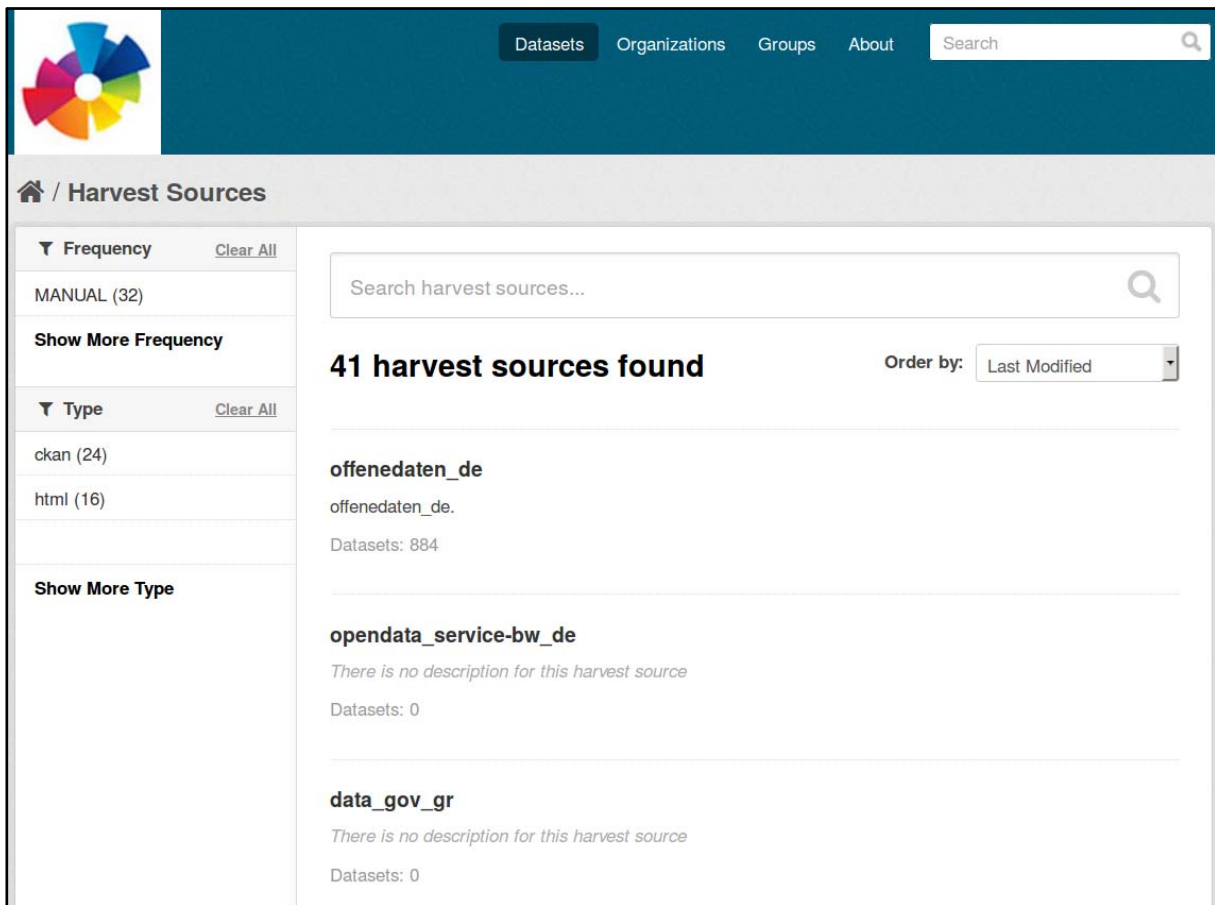


Figure 5: List of catalogues registered for harvesting

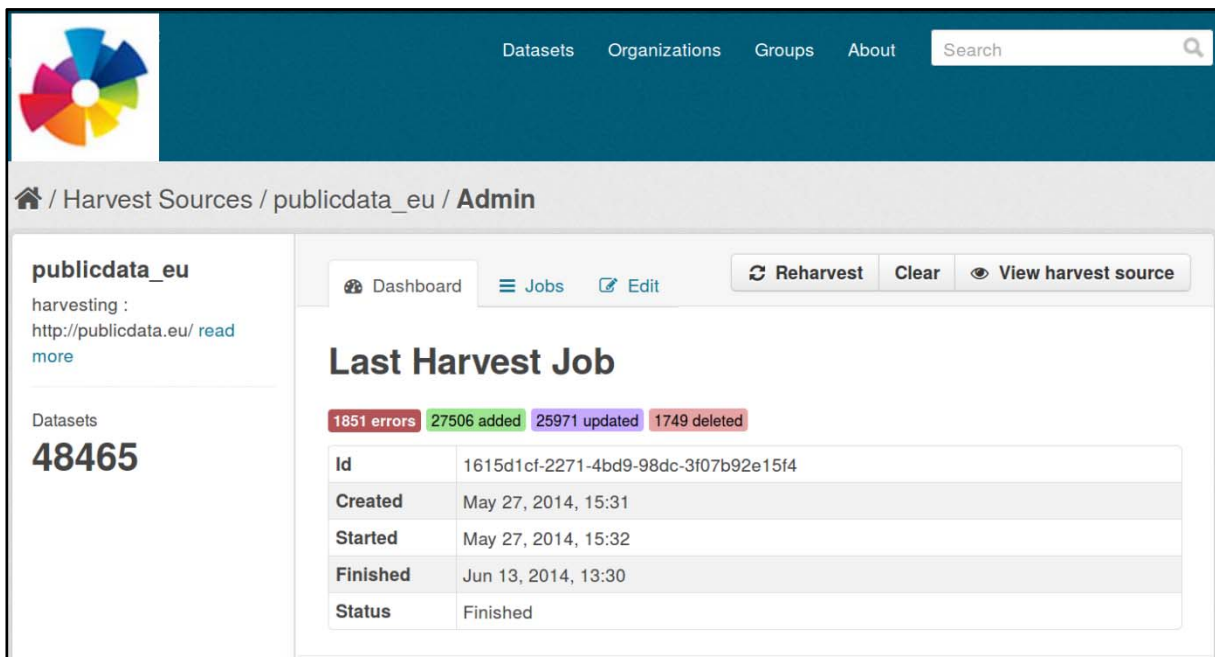


Figure 6: Display status of harvesting job

3.3 Metadata Harvester

This component is responsible for performing the actual task of collecting metadata from the registered catalogues. In particular, to handle the different types of open data platforms and APIs, two different metadata harvesters have been implemented, the *CKAN Harvester* and the *HTML Harvester*, which are described next.

3.3.1 CKAN Harvester

As already mentioned above, this harvester essentially relies on the `ckanext-harvest` extension⁶, which allows a host CKAN instance to collect and import datasets and/or metadata from other CKAN catalogues. This is useful, for example, for a national catalogue that also aggregates datasets from other, local catalogues. In turn, this extension relies on the CKAN API⁷, which allows a client to access -and potentially modify- the contents of a CKAN catalogue. Among others, the API includes a function that returns a list containing the identifiers of all the datasets contained in the catalogue, as well as a function for retrieving the complete set of metadata for a given dataset. In fact, the API itself has a rich set of features, allowing performance of more complex searches or even to modify information -assuming that the client has appropriate authorisation- but the two aforementioned functions are the ones used by the `ckanext-harvest` extension. Moreover, we use this extension to collect only the metadata of the datasets and not the datasets themselves.

The harvesting process comprises three steps:

- *gather*: in this step, the API is called to retrieve the IDs of all the datasets available in the target catalogue;
- *fetch*: in this step, for each dataset ID in the list, a call is issued to retrieve its metadata;
- *import*: this step stores all the retrieved content in the internal database of the host CKAN instance.

In our case, we have modified the *import* step of the process, since we are interested in storing the collected metadata not in the database used by CKAN but in our own database, the *Raw Metadata Repository* (see Figure 1), which is essentially a collection of JSON documents stored in a MongoDB database, as described in Section 3.4. This involves also some content manipulation to replace certain special characters or keywords that are not allowed when importing the data in the MongoDB database. Another change concerns the fact that the `ckanext-harvest` extension is intended for retrieving both the metadata information of the datasets as well as the datasets themselves. In our case, we only store the metadata records, while for the datasets we keep only the

⁶ <https://github.com/ckan/ckanext-harvest>

⁷ <http://docs.ckan.org/en/ckan-2.2/api.html>

URL and some derived information (file format, file size, MD5 checksum) that is needed to compute some of the metrics.

A sample JSON document containing the metadata of a dataset retrieved by the *CKAN Harvester* is shown in Table 5.

3.3.2 HTML Harvester

The `ckanext-harvest` extension covers only the case of CKAN catalogues, since it relies on the CKAN API being provided by the target catalogue to be harvested. Hence, different solutions are needed to cover those catalogues that are not deployed on CKAN (or, in some cases, catalogues that are deployed on CKAN but do not allow access to the API). One possible approach is to implement custom harvesters to address specific cases. This can be done by creating extensions similar to `ckanext-harvest` that implement the *harvesting interface*⁸ specified by CKAN. Indeed, a few such custom harvesters already exist that target specific catalogues⁹, such as `daten.berlin.de`, `data.london.gov.uk`, `opendata.paris.fr`, etc.

However, such an approach is not easily scalable when the aim is to monitor a large number of different catalogues. Instead, we have implemented a rather generic harvester that is based on scraping the contents of HTML pages, and hence does not rely on specific APIs or other knowledge of the underlying platform on which the targeted catalogue is deployed. Instead, the idea is to navigate to the pages of the catalogue presenting the metadata of each dataset, to parse and analyse the HTML tree structure of the page, and then extract from it the elements of interest, as described above.

The first part, i.e. parsing the HTML code of the page and creating the corresponding tree representation, we use the Python package *Beautiful Soup*¹⁰. The second part, i.e. locating the appropriate elements that contain the information of interest, is driven by the information provided during the registration of the catalogue (see Section 3.1). Recall that when a catalogue is registered which is to be harvested using the HTML Harvester, a sample HTML page displaying a randomly selected dataset is given, together with an indication of the elements that are to be extracted (either their labels, if available, or their values). Based on this example, the relevant paths in the HTML tree, for each attribute to be collected, are identified and stored in the job configuration. The *HTML Harvester* leverages this information when processing each HTML page of the harvested catalogue to locate the relevant elements in the HTML tree. Notice that this step is rather involved, since one needs to allow for a certain level of tolerance when searching in the HTML code of a page, e.g.

⁸ <https://github.com/ckan/ckanext-harvest#the-harvesting-interface>

⁹ <https://github.com/okfn/ckanext-pdeu/tree/master/ckanext/pdeu/harvesters>

¹⁰ <http://www.crummy.com/software/BeautifulSoup/>

allowing labels to match approximately if no exact match is found, skipping certain HTML tags (e.g., line breaks) in certain cases, etc.

The *HTML Harvester* has been developed as a CKAN extension, implementing the CKAN harvesting interface mentioned above. Hence, it also implements the three steps specified above, i.e. *gather*, *fetch* and *import*. However, in this case, retrieving the dataset identifiers is not done separately from collecting the metadata of each dataset (as opposed to `ckanext-harvest` in which these two steps are naturally distinguished due to the two different functions supported by the CKAN API); instead, the whole metadata extraction process actually takes place during the *gather* stage, and then they are imported in the *Raw Metadata Repository* in the *import* stage.

A sample JSON document containing the metadata of a dataset retrieved by the *HTML Harvester* is shown in Table 6.

3.4 Metadata Repository

The *Metadata Repository* is a database used to store the collected metadata, both the raw ones that are retrieved by the *Metadata Harvester* as well as the processed ones that are produced by the *Harmonisation Engine* (see Figure 1). For this purpose, we are using a MongoDB database. The rationale for this choice has already been explained in previous deliverables (D2.2 and D3.1). In a nutshell, MongoDB is a document-oriented database, perhaps the most popular one in the category of NoSQL databases. One of the main advantages is that it does not require a specific schema to be defined in advance for storing the data, which is important in our case since the raw collected metadata are quite heterogeneous, using different structures and attribute names. Instead, it allows the metadata to be stored and queried directly as JSON documents, such as those illustrated in Table 5 and Table 6.

More specifically, there are two document collections in the database. The first one is the *Raw Metadata Collection*, where the documents returned by the *Metadata Harvester* after each harvesting cycle are inserted. The second is the *Processed Metadata Collection*, which holds the results of running the *Harmonisation Engine* on the initial documents. More precisely, before executing the *Harmonisation Engine*, the contents of the *Raw Metadata Collection* are scanned, and those that have not been already processed (indicated by a special “flag” used to denote the status of each document) are copied to a temporary collection, referred to as the *Staging Area*, which is then used as input to the *Harmonisation Engine*.

The basic method for querying documents in a document collection is the function `db.collection.find(<criteria>, <projection>)`. For example, the query:

```
db.odm.find( { license: 'Creative Commons Attribution' } )
```

returns all the documents in the collection `odm` that have an attribute `license` with value `Creative Commons Attribution`, while the query:

```
db.odm.find( { catalogue_url: 'http://publicdata.eu' } )
```

returns all the documents that have an attribute `catalogue_url` with value `http://publicdata.eu`. Examples of more complex queries are included in Section 3.6.

3.5 Harmonisation Engine

The metadata extracted from the *Metadata Harvester* from the various catalogues are quite heterogeneous (see Deliverables D2.1 and D2.2 for a survey on the various existing metadata schemas as well as mappings between them). Moreover, even within the same catalogue, it is often not required that a data publisher strictly adheres to a specific metadata schema when publishing a dataset. This is done to lower the barrier and make it easier and quicker to make datasets publicly available. For example, CKAN specifies a few basic metadata attributes as “core”, such as `title`, `url`, `author`, `maintainer`, `license`, `notes`, `tags`, `resources`, while allowing arbitrary, user-defined key-value pairs to be added by the data publisher under a field called “extras” to provide any additional information about the dataset¹¹. The latter may contain information on various other useful attributes, such as geographic coverage, release date, language, etc.

As a result, before being able to analyse the collected metadata and compute any useful metrics, it is necessary to deal with this heterogeneity, involving:

- *different structures*: for example, the same attribute may appear as a top-level element in some schema or nested within another element in some other schema;
- *different attribute names*: very often the same type of information is listed under attributes with different name variations, such as `license`, `license_id`, `license_title`; `release_date`, `date_released`;
- *different value representations*: there exists also high diversity in terms of value representations, such as use of abbreviations, different date formats, use of country names versus country codes, different languages, etc.

Consequently, to be able to more accurately and reliably analyse the data and compute the required measurements, we need to clean and map them to an internal representation. For this purpose, we have designed and use an internal metadata schema, which focuses on the attributes that have been identified as needed for the various metrics to be computed eventually. Moreover, this schema is designed to be close to that of CKAN, so that it is possible to import the processed metadata into a

¹¹ <http://docs.ckan.org/en/ckan-1.8/domain-model.html>

CKAN repository without needing any significant transformations. The schema is shown in the following table.

Table 2: Internal metadata schema

- `id` (type: string) //checksum on the dataset's url
- `title` (type: string) //title of the dataset
- `notes` (type: string) //short description of the dataset
- `author` (type:string) //the entity that is considered as the author of the dataset
- `maintainer` (type:string) //the entity who is responsible for maintaining the dataset
- `organization` (type:string) //the entity that is considered as the publisher of the dataset
- `catalogue_url` (type:string) //the URL of the catalogue where the dataset was found
- `tags` (type: array of strings) //list of tags describing the dataset
- `num_tags` (type:int) //automatically calculated to store the number of tags in order to facilitate computation of relevant metrics
- `license_id` (type: string) //the licence under which the dataset is provided
- `resources` (type:array) //a list of resources (i.e. files) that comprise the dataset
 - `url` (type:string) //the URL of the resource
 - `checksum` (type:string) // automatically computed hash of the URL, used to facilitate finding duplicate URLs
 - `mimetype` (type:string) //the media type of the resource (e.g. text/html)
 - `format` (type:string) //the file format of the resource (e.g. csv, rdf, xls)
 - `size` (type:int) //the file size of the resource (measured in bytes)
- `num_resources` (type:int) // automatically calculated to store the number of resources in order to facilitate computation of relevant metrics
- `language` (type:string) //the language of the metadata of this dataset
- `country` (type:string) //the country the dataset refers to
- `state` (type:string) //the state the dataset refers to
- `city` (type:string) //the city the dataset refers to
- `date_released` (type:date) //the date the dataset was first released (in the monitored catalogue)
- `date_updated` (type:date) //the date the dataset was last updated (in the monitored catalogue)
- `metadata_created` (type:date) //the date the dataset was first harvested
- `metadata_updated` (type:date) //the date the dataset was last harvested
- `update_frequency` (type:string) //the frequency of updates for this dataset (e.g. monthly, yearly)
- `category` (type:string) //the theme of the dataset (e.g. economy, transport)
- `platform` (type:string) //the type of platform of the monitored catalogue (currently allows distinguishing according to the two types of harvesters used, i.e. CKAN/HTML)

Consequently, the metadata extracted by the harvesters need to be mapped to the attributes of the above described schema. For the *CKAN Harvester*, two cases are distinguished. For those attributes belonging to the set of core metadata of the CKAN domain model, the mappings are known in advance; essentially, there is a direct correspondence to our schema (e.g., `title` → `title`; `notes` → `notes`; `author` → `author`, etc.). For the rest, i.e. those attributes that appear under the element `extras`, mapping rules have been specified to deal with different naming variations. In practice, this was done by harvesting metadata from a number of CKAN catalogues and then examining the various attribute names that appear under the node “extras” and defining mappings to the attributes in the internal schema. Once this has been done for a few catalogues, subsequent cases can be addressed by applying an already specified mapping. For the *HTML Harvester*, the mappings have been already indicated during the phase of catalogue registration. Recall that, as described in Section 3.1, for each metadata attribute to be extracted from the HTML pages, an extraction rule is created to identify its location in the page; hence, this also serves as a mapping, so that the JSON document returned by the *HTML Harvester* already complies with the above defined internal schema.

Notice that the schema also includes certain attributes whose values are not explicitly included in the original metadata but are instead computed based on other information. In particular:

- `num_tags` and `num_resources`: these attributes store the size of the arrays `tags` and `resources`, respectively; this is done for reasons of convenience and performance, so that these values are readily available to queries computing relevant metrics, such as average number of tags or resources for the datasets of a given catalogue.
- MD5 `checksums` on dataset URL and resource URL: these values are computed with the Python module `hashlib.md5()` and are also stored in order to facilitate further processing, such as quickly identifying (exact) duplicates.
- `mimetype`, `format`, `size` of a resource: this is key information for computing various metrics that provide useful insights for the monitored open data catalogues, such as most common file formats, average size of provided files, etc. Usually, the file format of the resource is provided as a metadata attribute or can be trivially extracted from the resource’s URL suffix; however, the mimetype and size information are often not included in the original metadata record or may not be accurate. Therefore, we compute instead this information by using the provided URLs of the resources included in the dataset. The Python module `urllib2.headers` is used for this purpose as shown by the code below:

```
size = urllib2.urlopen(<resource_url>).headers["Content-Length"]
```

and

```
mimetype = urllib2.urlopen(<resource_url>).headers["Content-Type"]
```


where `<resource_url>` is the URL of the resource for which the size and mimetype information is requested.

The next main aspect to address is to clean and integrate the values of those attributes. This involves dealing with different value formats and representations, naming variations, synonyms etc. We outline below the main cases addressed so far in the implementation.

- `license_id`. Licence information is typically found in the original metadata under the attributes `license`, `license_id` or `license_title`. To clean and integrate the values, we have semi-automatically created a dictionary as shown in Table 7. The list shows the various types of licences that have been found in the metadata collected so far. The dictionary is used to normalize the values by mapping to the same term several variations or translations of the same licence (shown in brackets).
- `resources/format`. Similarly, we have created a dictionary to normalize file formats of the dataset resources. Besides the fact that a large variety of file types are encountered in the monitored catalogues, it is often the case that the value found in this attribute refers to the mime type of the resource or the corresponding application (e.g. excel) instead of the file extension, hence there exist again several variations that need to be mapped to the same value to allow for further comparisons and aggregations. Note that this dictionary, as also the aforementioned one for licenses, will be available in the GitHub repository.
- `language`, `country`. To represent languages and country names, the ISO 639-1¹² and ISO 3166-1¹³ codes are used, respectively.
- `date_released`, `date_updated`. There exists a high diversity in the use of date formats in different catalogues or in the metadata of different datasets. To normalize these values, we use the `dateutil` Python module. Although this was adequate for many of the cases we have encountered, the library is not able to parse successfully dates which contained non numeric characters, e.g. *sábado, 9 Julio, 2005*. To handle these cases, a dictionary mapping was defined that translates non-English date related terms (i.e. names of months and days) to their English equivalents, before trying to parse the date format with the `dateutil` library. Hence, such values are also transformed to a normalized representations, e.g. *sábado, 9 Julio, 2005* → 2005-07-09 00:00:00+00:00 (Spanish), *14 maggio 2012* → 2012-04-14 00:00:00+00:00 (Italian). Finally, a problem that we have encountered is that many date fields contain only the year that a dataset was released or updated. To transform these values to this normalized representation, we followed the convention of filling in the missing values by assigning to it the first day of the first month of the corresponding year. That is, if the release data of a dataset is indicated as 2009, the assigned normalized value is 2009-01-01 00:00:00+00:00.

¹² http://www.iso.org/iso/language_codes.htm

¹³ http://www.iso.org/iso/country_codes.htm

3.6 Analysis Engine

The purpose of the *Analysis Engine* is to provide access to the collected metadata after they have been processed by the *Harmonisation Engine*, as well as to calculate a series of key metrics that are used for monitoring. Hence, there are two main operations involved:

- *Retrieval*. The *Analysis Engine* allows the *Demonstration Site* -or any other external components that may need to access the contents of the *Processed Metadata Repository*- to retrieve the collected metadata after their harmonisation has taken place. This can be done by retrieving first the list of monitored catalogues and then requesting the metadata of the datasets that have been collected from a specific catalogue. This allows the *Demonstration Site* or other components to obtain all or parts of the processed metadata in order to perform any further processing (e.g. querying, analysis) required by end-user applications.
- *Aggregation*. Moreover, the *Analysis Engine* computes the various aggregations specified by the metrics that have been defined for monitoring, and makes those results accessible to the *Demonstration Site* or other components as well. These computations essentially involve aggregate queries (max, min, average) on several dimensions, such as per catalogue, country, date, category or licence type.

To hide the complexity of directly querying the contents of the database, and to make the design more modular, both of the above mentioned sets of operations are exposed via a RESTful API [1], as had been already designed in previous deliverables (see D3.1 and D3.2).

More specifically, the REST interface that was used as a basis for our implementation is the *sleepy.mongoose*¹⁴. *Sleepy.mongoose* is an open source REST interface for MongoDB written in Python. The *pymongo* library¹⁵ is also needed in order to connect and run queries to the MongoDB database. Some additional customisations were also made to accommodate our specific needs, as described below.

The API is running as a server and listens to port 27080. The URL of a request consists of three main parts:

- the base URL, which has the form: `http://[hostname:27080]/api/v1.0/`, where `hostname` is the IP address of the server where this module is hosted;
- the name of the called method, added at the end of the base URL with the underscore as a prefix;
- any arguments that can be provided for filtering the results. These follow the method name, using the character `?` as a separator. Multiple arguments can be provided, which are then separated with the character `&`.

¹⁴ <https://github.com/10gen-labs/sleepy.mongoose/wiki>

¹⁵ <http://api.mongodb.org/python/current>

Hence, the full form of the URL of a request is:

```
http://[hostname:27080]/api/v1.0/_method?arg1=value1&arg2=value2
```

For example:

```
http://83.212.122.164:27080/api/v1/_find?attribute=metadata_created&start_date=2012-7-30
```

returns results by filtering on the attribute `metadata_created`.

Responses are formatted in JSON and consist of the following fields:

- `ok`, which has the value 1 if the query succeeded, or 0 otherwise;
- `result` which is an array of documents from the database;
- `id` which is not always present and is used as cursor id whenever results need to be returned in multiple pages (optional);

The implemented methods can be separated conceptually in three high level categories that provide the means for a third party tool to have full access to the processed metadata.

The first one is the method `_commands` that returns a list of all implemented functions that are provided by the API with a very basic form of documentation. A snippet of the response of this method is shown below:

Table 3: Snippet of the response of the API method `_commands`

```
{
  "ok": 1,
  "result": [
    {
      "_catalogues": {
        "help": "URLs of harvested catalogues and their
total number of datasets"
      }
    },
    {
      "_catdatasetsfreq": {
        "help": "Frequency of catalogued datasets"
      }
    },
    ...
  ]
}
```

The second group includes the functions that return the results for the defined metrics. These functions perform aggregation queries to the MongoDB database using the aggregation pipeline framework¹⁶. Based on the type of request, the data processing is either done on the fly or cached. This is highly dependent on the attributes involved in the aggregated queries and whether it is meaningful or quite feasible with respect to time restrictions to be harvested in the first phase or not. For example, computing a metric that requires calculating the percentage of resources' broken links is done offline and the result is cached and used when requested through the API, since the process of accessing links to verify them is a long running task. Other metrics, such as number of datasets in a given catalogue filtered by license type, are computed online.

The last category provides access to raw datasets in the repository. It contains two functions `_find` and `_more`. The former returns the raw datasets, while the latter allows accessing multiple dataset pages, if they exist. The `_find` method, used without any arguments, returns all the datasets contained in the database. The structure of the response is in accordance to the schema presented in Table 2. The method supports three arguments:

- `batch_size`, that is the number of records that are returned by each call of the API; default is 15
- `catalogue_url`, that restricts the results only to those belonging to a certain catalogue
- `odm_id`, that is the unique id of a dataset; the method returns the metadata record of this specific dataset.

An example of getting the metadata records of the datasets harvested from the national catalogue of the United Kingdom in groups of 50 is shown below:

```
http://83.212.122.164:27080/api/v1.0/_find?batch_size=50&catalogue_url=http://data.gov.uk
```

In order to access subsequent records after the first page, the `_more` method can be used. This method takes as argument the 'id' value that is contained in the JSON result after calling the `_find` method. For instance, the returned JSON in the above example has the value of 0 in the key 'id'. We can use this value to access the next batch of datasets of size 50 as follows:

```
http://83.212.122.164:27080/api/v1.0/_more?id=0
```

The aggregation pipeline framework's general form that was used for calculating the desired metrics is as shown below:

¹⁶ <http://docs.mongodb.org/manual/core/aggregation-pipeline>

```
db.odm.aggregate([
  { $match: { '<attribute_to_match>': { '$nin': [ '', null ] } } },
  { '$unwind': { '<attr_contained_in_array>' },
  { '$group': { '_id': '<$attr_value>',
other_needed_calculated_values... } },
  { '$sort': { 'counter': 1 or -1 } },
  { '$project': { 'attr1': 1 or 0, '<new_attr>': '<$attr2>' } }
])
```

where:

- `$match` is used to exclude all null or empty attribute values related to specific field;
- `$unwind` is used to deconstruct fields that are stored as arrays in the document;
- `$group` aggregates specific attributes in the document to produce specific metrics;
- `$sort` is used when results need to be presented in ascending or descending order;
- `$project` renames, includes (1) or excludes (0) certain attributes from being present in the query result.

For example, if we need to get the number of unique organisations publishing data, the executed query has the following form:

```
db.odm.aggregate([
  { '$match': { 'organization.title': { '$nin': [ '', None ] } } },
  { '$group': { '_id': '$catalogue_url', 'unique_org_size':
{ '$addToSet': '$organization.title' } } },
  { '$unwind': "$unique_org_size" },
  { '$group': { '_id': "$_id", 'counter': { '$sum': 1 } } },
  { '$sort': { 'counter': -1 } }
])
```

3.7 Administration Panel

This component will comprise dashboards for allowing the ODM system administrator to monitor, configure and control the various operations of the platform, such as checking the status of harvesting jobs, viewing statistics of the harmonisation process, configuring mappings, etc. This will also allow for a manual curation of the metadata collection if and when needed, identifying incorrect results and configuring the process accordingly.

The design and implementation of this component is planned for the second period of the project and will be reported in Deliverable D3.6.

3.8 Demonstration Site

The *Demonstration Site* comprises a set of components that are responsible for generating various visualisation and reports that will allow the end user to obtain a comprehensive overview of the monitored open data catalogues, based on a set of key metrics that have been identified.

An overview of the metrics that have been selected for monitoring, as well as a survey of appropriate visualisations techniques to help convey the analysis results to the end users, have been presented in Deliverable D2.3. The current status of the implementation of this module is reported in Deliverable D3.4.

4 CONCLUSIONS AND NEXT STEPS

In this report, we have first presented the overall architecture and the processing workflow for the ODM system, and then we have described the status of the implementation of the main components of the platform. The progress of the work is summarised below¹⁷:

- *Catalogue registry.* We have implemented a web-based user interface for allowing the ODM administrator or a user with the data publisher role to register new catalogues for monitoring. This involves providing some basic information that is used to configure a harvesting job to collect metadata records for this catalogue. Essentially, this refers to distinguishing between catalogues that are deployed on CKAN, in which case metadata can be harvested by leveraging the CKAN API, and those for which metadata are collected via HTML scraping. For the latter, some information needs to be provided to guide the extraction process from the HTML pages. To reduce the burden for the user, when configuring a new catalogue, existing configurations of previously registered catalogues are checked and reused whenever possible. The goal for the second period is to examine ways to further reduce the manual input required for registering a new catalogue.
- *Job manager.* We have implemented a component that, given as input the configurations of the registered catalogues, creates and manages corresponding harvesting jobs. This is based on a queue for scheduling these tasks and monitoring their status and progress. At the current stage, harvesting jobs are scheduled for execution manually. The plan for the second period involves deploying this component as a background process that can schedule jobs periodically, as well as providing a visual interface and statistics regarding the execution of these jobs.
- *Metadata harvester.* For this component we have addressed two cases: harvesting metadata from CKAN catalogues and extracting metadata via HTML scraping. Applying these methods to collect metadata from several open data catalogues is an ongoing process. Based on these results, we will determine any improvements that can be made to improve the accuracy of this process, e.g. how to better configure/learn extraction rules or whether there is a need to develop other types of harvesters to address specific cases.
- *Metadata repository.* A MongoDB database has been setup to store the collected metadata records. Each record is stored as a JSON document, and two different collections are maintained, one comprising the raw metadata extracted by the harvester and one comprising the metadata after their harmonisation.
- *Harmonisation engine.* For this component, we have implemented a set of scripts for performing data cleaning and transformations. This involves mapping the collected metadata from their original schemas to an internal representation comprising the attributes that are needed to compute the selected metrics for monitoring. It also involves

¹⁷ A code repository has been setup on GitHub, where the code will be made available during the course of the project: <https://github.com/opendatamonitor>

value mapping and translation for those attributes so that further analysis can be performed. This task has focused on attributes such as licence types, file formats, sizes and dates. Resolving spatial attributes and harmonizing dataset categories is the focus of the current work. The goal for the second period is to improve the accuracy and robustness of these processes while reducing the amount of manual curation required (for defining mappings and/or validating results).

- *Analysis engine.* For this component we have implemented: (a) a series of scripts that compute the specified metrics by executing corresponding queries in the database, and (b) a RESTful API for providing access to the results of these computations as well as the contents of the metadata repository in general. Several metrics have already been implemented and are at the stage of testing as more metadata are collected by the harvester. The current code base will be extended in the second period to compute additional metrics, both from the ones already identified but currently pending because the harmonisation of involved attributes is not completed as well as others that may be identified in the course of the project.
- *Administration panel.* The implementation of this component will take place in the second period. Its purpose will be to provide a graphical user interface to facilitate the ODM administrator to monitor and control various aspects of the system and the processing workflow.
- *Demonstration Site.* The progress of the work for this component is reported in Deliverable D3.4.

5 REFERENCES

- [1] Fielding, Roy T., and Richard N. Taylor. "Principled design of the modern Web architecture." ACM Transactions on Internet Technology (TOIT) 2.2 (2002): 115-150.

6 APPENDIX

6.1 Example of harvesting job configuration

Table 4: Example of harvesting job configuration for the catalogues publicdata.eu and datos.gob.es

```
{
  "description" : "",
  "title" : "publicdata_eu",
  "cat_url" : "http://publicdata.eu/",
  "frequency" : "MANUAL",
  "type" : "ckan",
  "id" : "d7158756-58e2-4cb7-8974-c042d12675cf"
}

{
  "description" : "",
  "title" : "datos_gob_es",
  "cat_url" : "http://datos.gob.es/catalogo?title=&order=&page=",
  "frequency" : "MANUAL",
  "type" : "html",
  "id" : "4e6e4c22-a945-43a0-9450-2e5860e3177f"
}

{
  "maintainer" : "@/@label",
  "frequency" : "@/@label",
  "geographic_coverage" : "Cobertura geográfica:@/@label",
  "category" : "Categorías:@/@label",
  "license" : "@/@label",
  "title" : "@/@value",
  "next" : "",
  "version" : "@/@label",
  "tags" : "Etiquetado como:@/@label",
  "step" : "1",
  "updatedate" : "Fecha de última actualización:@/@label",
  "date" : "Fecha de creación:@/@label",
  "enddate" : "@/@label",
  "publisher" : "Publicador:@/@label",
  "resource" : "Distribuciones@/@link",
  "name" : "",
}
```

```

    "language" : "spanish",
    "url" : "http://datos.gob.es/catalogo/estadistica-del-impuesto-
de-matriculacion",
    "afterurl" : "",
    "notes" :
"text_file.write(str(soup2.find_all('html',recursive=False)[0].find_
all('body',recursive=False)[0].find_all('div',recursive=False)[1].fi
nd_all('div',recursive=False)[1].find_all('div',recursive=False)[0].
find_all('div',recursive=False)[5].find_all('div',recursive=False)[0
].find_all('div',recursive=False)[0].find_all('div',recursive=False)
[0].find_all('div',recursive=False)[1].find_all('div',recursive=Fals
e)[0].find_all('div',recursive=False)[0].find_all('div',recursive=Fa
lse)[0].find_all('div',recursive=False)[0].find_all('div',recursive=
False)[0].find_all('div',recursive=False)[0].find_all('div',recursiv
e=False)[1].find_all('div',recursive=False)[1].find_all('div',recurs
ive=False)[0].find_all('div',recursive=False)[1].find_all('div',recu
rsive=False)[0].getText().encode('utf-
8').lstrip().rstrip()))@/@value",
    "temporal_coverage" : "Cobertura temporal:@/@label",
    "cat_url" : "http://datos.gob.es/catalogo?title=&order=&page=",
    "contactpoint" : "@/@label",
    "identifier" : "/catalogo/",
    "type" : "html"
}

```

6.2 Example of raw collected metadata

Table 5: JSON document containing the metadata of a dataset returned by the CKAN Harvester

```

{
  "_id" : ObjectId("5397f1b2ce2e3b6ff6ba92b7"),
  "license_title" : "UK Open Government Licence (OGL)",
  "maintainer" : null,
  "private" : false,
  "maintainer_email" : null,
  "num_tags" : 6,
  "id" : "c580285c-0dd1-4b7f-ab1d-c733a51e9f41",
  "metadata_created" : "2011-06-13T19:25:51.144044",
  "relationships" : [],
  "license" : "UK Open Government Licence (OGL)",
  "metadata_modified" : "2011-06-13T19:25:51.912066",

```

```
"author" : null,
"author_email" : null,
"download_url" : "http://www.gro-
scotland.gov.uk/statistics/publications-and-data/population-
estimates/mid-2007-population-estimates-scotland/index.html",
"platform" : "ckan",
"state" : "active",
"version" : null,
"creator_user_id" : null,
"type" : "dataset",
"resources" : [
  {
    "resource_group_id" : "c3954e94-44eb-4986-93f4-
6b46f5665e6f",
    "cache_last_updated" : null,
    "package_id" : "c580285c-0dd1-4b7f-ab1d-c733a51e9f41",
    "webstore_last_updated" : null,
    "id" : "7bcb438b-0e15-4e44-94cf-aabd97da1602",
    "size" : null,
    "openness_score" : "0",
    "hash" : "",
    "description" : "hub/id/119-32178",
    "format" : "",
    "last_modified" : null,
    "openness_score_failure_count" : "0",
    "url_type" : null,
    "openness_score_reason" : "URL unobtainable",
    "mimetype" : null,
    "cache_url" : null,
    "name" : null,
    "created" : null,
    "url" : "http://www.gro-
scotland.gov.uk/statistics/publications-and-data/population-
estimates/mid-2007-population-estimates-scotland/index.html",
    "webstore_url" : null,
    "mimetype_inner" : null,
    "position" : 0,
    "resource_type" : null
  }
],
```

```

"num_resources" : 1,
"tags" : [
  "lifeinthecommunity",
  "population",
  "populationandmigration",
  "populationchange",
  "populationestimates",
  "scotland"
],
"catalogue_url" : "http://publicdata.eu",
"groups" : [],
"license_id" : "UK Open Government Licence (OGL)",
"organization" : null,
"name" : "mid-year_population_estimates_for_scotland_-_mid-2007",
"isopen" : true,
"notes_rendered" : "<p>Presents key findings from the Registrar General's Annual Review.\n Source agency: General Register Office for Scotland\n Designation: National Statistics\n Language: English\n Alternative title: Scotland's Population: The Registrar General's Annual Review of Demographic Trends\n</p>",
"url" : null,
"ckan_url" : "http://publicdata.eu/dataset/mid-year_population_estimates_for_scotland_-_mid-2007",
"notes" : "Presents key findings from the Registrar General's Annual Review.\r\nSource agency: General Register Office for Scotland\r\nDesignation: National Statistics\r\nLanguage: English\r\nAlternative title: Scotland's Population: The Registrar General's Annual Review of Demographic Trends",
"owner_org" : null,
"ratings_average" : null,
"extras" : {
  "eu_country" : "UK",
  "temporal_coverage-from" : "",
  "date_updated" : "",
  "published_via" : "General Register Office for Scotland [12077]",
  "temporal_coverage_to" : "",
  "import_source" : "ONS-ons_data_2008-07",
  "geographical_granularity" : "Country",

```

```

    "openness_score" : "0",
    "temporal_granularity" : "",
    "agency" : "General Register Office for Scotland",
    "geographic_granularity" : "",
    "temporal_coverage-to" : "",
    "department" : "Scottish Government",
    "harvest_dataset_url" :
"http://catalogue.data.gov.uk/package/c580285c-0dd1-4b7f-ab1d-
c733a51e9f41",
    "precision" : "",
    "temporal_coverage_from" : "",
    "published_by" : "Scottish Government [11414]",
    "taxonomy_url" : "",
    "harvest_catalogue_url" : "http://catalogue.data.gov.uk",
    "categories" : "Population",
    "geographic_coverage" : "010000: Scotland",
    "external_reference" : "ONSHUB",
    "harvest_catalogue_name" : "Data.gov.uk",
    "national_statistic" : "yes",
    "openness_score_last_checked" : "2011-06-
06T17:24:21.335775",
    "update_frequency" : "annual",
    "date_released" : ISODate("2008-07-24T00:00:00.000Z")
  },
  "license_url" : "http://reference.data.gov.uk/id/open-
government-licence",
  "ratings_count" : 0,
  "title" : "Mid-Year Population Estimates for Scotland - Mid-
2007",
  "revision_id" : "e967ae15-ba55-4d30-89b9-4ba4c3520456"
}

```

Table 6: JSON document containing the metadata of a dataset returned by the HTML Harvester

```

{
  "_id" : ObjectId("53b2dd1cce2e3b6a70b7ce70"),
  "name" : "3525a92ab605e5ac41892b15f8ea301c",
  "title" : "Directorio de la Administración General e
Institucional de la Comunidad datos.gob.es",
  "url" : "http://datos.gob.es//catalogo/directorio-de-

```

```

administracion-general-institucional-de-comunidad-10",
  "notes" : "Conjunto de 6rganos administrativos, centrales y
territoriales que desarrollan funciones ejecutivas de car6cter
administrativo. Incluye Consejer6as, Direcciones Generales,
Delegaciones Territoriales, as6 como Organismos Aut6nomos y Entes
P6blicos.",
  "catalogue_url" : "http://datos.gob.es",
  "platform" : "html",
  "extras" : {
    "Geographic Coverage" : "Castilla y Le6n",
    "Release Date" : ISODate("2012-04-19T00:00:00.000Z"),
    "Language" : "Espa6ol"
  },
  "resources" : [
    {
      "url" :
"http://www.datosabiertos.jcyl.es/web/jcyl/risp/es/directorio/admini
stracion-general/1284217430398.csv",
      "mimetype" : "text/csv;charset=ISO-8859-15",
      "name" : "Distribuciones",
      "format" : "CSV"
    },
    {
      "url" :
"http://www.datosabiertos.jcyl.es/web/jcyl/risp/es/directorio/admini
stracion-general/1284217430398.xml",
      "mimetype" : "application/xml;charset=UTF-8",
      "name" : "Distribuciones",
      "format" : "XML"
    },
    {
      "url" :
"http://www.datosabiertos.jcyl.es/web/jcyl/risp/es/directorio/admini
stracion-general/1284217430398.rdf",
      "mimetype" : "application/rdf+xml;charset=UTF-8",
      "name" : "Distribuciones",
      "format" : "RDF"
    }
  ]
}

```

6.3 Dictionary for harmonising licences

Table 7: Dictionary for licences cleaning and harmonisation

- **CC BY** [*Creative Commons Attribution (CC-BY); Creative Commons BY; creative-commons-attribution-cc-by; cc-by; Creative Commons Attribution; Creative Commons Attribution License 3.0; Creative Commons Attribuzione; Creative Commons Attribution 3.0 (CC-BY-3.0); Internet CC-BY; Creative Commons Reconocimiento 3.0 España*]
- **CC BY-SA** [*Creative Commons Attribution Share-Alike; Creative Commons Attribution-ShareAlike 3.0 Italy; Creative Commons Attribuzione - Condividi allo stesso modo; naamsvermelding---gelijkdelen-cc-by-sa*]
- **CC BY-ND** [*cc-nd*]
- **CC BY-NC** [*cc-nc; Creative Commons Non Commerciale (Qualsiasi tipo); Creative Commons Non-Commercial (Alle); Creative Commons Non-Commercial (Any)*]
- **CC BY-NC-SA** [*CC BY-NC-SA 3.0; Creative Commons Attribution Non-Commercial Share-Alike*]
- **CC BY-NC-ND** [*Creative Commons Attribution Non-Commercial Non-Derivative*]
- **CC0** [*Creative Commons CCZero (CC0); cc-zero; CC0 1.0 Universal; Creative Commons 0 apart from images; Creative Commons CCZero Public Domain; Creative Commons Zero (CC0); Public Domain Dedication; Internet CC0*]
- **GPLv2** [*gpl-2.0*]
- **GPLv3** [*gpl-3.0*]
- **ODC-BY** [*Open Data Commons Attribution License*]
- **ODC-ODbL** [*Open Data Commons Open Database License (ODbL); Open Database License (ODbL)*]
- **ODC-PDDL** [*Open Data Commons Public Domain Dedication and Licence (PDDL)*]
- **GFDL** [*GNU Free Documentation License*]
- **OSL** [*osl-3.0; Open Software License*]
- **OS OpenData License** [*OS Open Data Licence; Ordnance Survey Open Data Licence*]
- **OS Open Government Licence** [*Ordnance Survey Open Government Licence; Open Government Licence apart from images*]
- **OS Public Sector End User Licence - INSPIRE** [*Ordnance Survey Public Sector Inspire End User Licence*]
- **PSMA** [*Public Sector Mapping Agreement*]
- **UK Crown** [*UK Crown Copyright; ukcrown-withrights*]
- **OGL** [*UK Open Government Licence; uk-ogl*]
- **Against DRM**
- **DL-DE-BY** [*dl-de-by-1.0; Datenlizenz Deutschland - Namensnennung - Version 1.0*]
- **DL-DE-BY-NC** [*dl-de-by-nc-1.0; Datenlizenz Deutschland - Namensnennung – nicht kommerziell - Version 1.0*]
- **IODL v1.0** [*Italian Open Data License 1.0; Italian Open Data License v1.0; IODL-1.0; iodl1*]
- **IODL v2.0** [*Italian Open Data License 2.0; Italian Open Data License v2.0; IODL-2.0; iodl2*]
- **Licence Ouverte** [*Licence Ouverte / Open Licence*]
- **Other (Non-Commercial)** [*Altro (Non Commerciale); Ostatní (Licence pro nekomerční*]

využití); other-nc]

- **Other (Attribution)** [*Altro (con Attribuzione); Andere (Namensnennung); Annet (navngivelse); other-at]*]
- **Other (Open)** [*Altro (di tipo Open); Andere (Offen); Annet (åpen); Ostatní (Otevřená licence); other-open]*]
- **Other (Not Open)** [*Annet (ikke åpen); Ostatné (zatvorená licencia); other-closed]*]
- **Other (Public Domain)** [*Andere (gemeinfrei); Annet (public domain); Otra (Public Domain); other-pd; publik-domein; Остало (Явни домен)]*]
- **Non-Commercial Government Licence v1.0** [*Non-Commercial Government Licence v1.0]*]
- **INTEL OSL** [*intel-osl]*]
- **Private** [*Dataset is not available for publication as the content is sensitive]*]
- **License Not Specified** [*Licence není uvedena; Licentie is niet gespecificeerd; Licenza non specificata; Lisens ikke angitt; Lizenz nicht angegeben; Nie je uvedená licencia; Not specified; Other::License Not Specified; notspec; notspecified; See website dataset provider]*]