



D3.6 TOOL ARCHITECTURE AND COMPONENTS/PLUGINS PROGRAMMING STATUS REPORT 2

PROJECT

Acronym: **OpenDataMonitor (ODM)**
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>
<http://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:	D3.6
Title:	Tool architecture and components/plugins programming status report 2
Lead beneficiary:	ATHENA
Work package:	WP3: Concept Design and Software Development
Dissemination level:	Public (PU)
Nature:	Report (RE)
Due date:	August 31, 2015
Submission date:	August 28, 2015
Authors:	Vassilis Kaffes, ATHENA Dimitris Skoutas, ATHENA
Contributors:	Thodoris Raiois, ATHENA Ejona Sauli, SYNYO
Reviewers:	Amanda Smith, ODI

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	7
1.1	Scope and purpose	7
1.2	Architecture overview of the ODM platform	8
1.2.1	Main components	8
1.2.2	Outline of the processing workflow	10
2	Catalogue Registry	12
2.1	Registration form for Socrata harvester	12
2.2	RDF support for the HTML harvester	13
2.3	Handler for HTML pages where JavaScript snippets are used for paging	15
2.4	Other enhancements and modifications	17
3	Metadata Harvester	20
3.1	Socrata harvester	20
3.2	HTML harvester enhancements	21
3.2.1	Ability to process RDF content	21
3.2.2	Ability to process JavaScript code	22
3.2.3	Assessment of enhancements	24
4	Harmonisation Engine	28
4.1	De-duplication module	28
4.2	Harmonization process	34
4.3	Levels of applied mappings	36
5	Analysis Engine	38
6	Administration Panel	39
6.1	Harvesting panel	39
6.2	General overview panel	40
6.3	Harmonization panel	42
7	Conclusions and Next Steps	46

8	APPENDIX.....	48
8.1	Examples.....	48
8.2	Code Snippets.....	52

LIST OF FIGURES

Figure 1: Overview of architecture and processing workflow.	8
Figure 2: Additional option for registering a catalogue using the Socrata platform.	12
Figure 3: Web form for configuring the harvester for the registered catalogue.	14
Figure 4: Option to provide link to metadata available in RDF format.	15
Figure 5: Configuring page navigation in the case of JavaScript code.	16
Figure 6: Example of configuring page navigation when JavaScript code is used.	17
Figure 7: Allowing multiple URLs as starting pages for metadata collection within a catalogue.	18
Figure 8: Example of added validation checks in the catalogue registration form.	19
Figure 9: Link to RDF description of a dataset's metadata.	22
Figure 10: Distribution of catalogues handled by each harvester type.	24
Figure 11: Portion of catalogues handled by the HTML harvester using Javascript code.	25
Figure 12: Number of datasets collected from the monitored catalogues.	25
Figure 13: Availability of each metadata attribute in the collected datasets.	26
Figure 14: View of the database for metadata collection and processing.	27
Figure 15: Flow chart for the indexing phase of the de-duplication process.	29
Figure 16: Flow chart for the searching phase of the de-duplication process.	33
Figure 17: Example of duplicate metadata objects.	34
Figure 18: Example of harmonization job.	36
Figure 19: View of the harvesting panel.	39
Figure 20: View of the admin panel.	40
Figure 21: View of the <i>Jobs</i> tab.	40
Figure 22: View of the general overview panel.	41
Figure 23: Dashboard view of the harmonization panel.	42
Figure 24: Example view of the harmonization panel.	43
Figure 25: Example of viewing mapping rules.	44
Figure 26: Example of editing rules.	45

LIST OF TABLES

Table 1: Decision table to label metadata as <i>Candidate</i> or <i>Unique</i>	31
Table 2: A Socrata JSON document instance.....	48
Table 3: RDF example.	49
Table 4: Example of the partialOrder.csv file	50
Table 5: Select harvester type for registration.....	52
Table 6: Select configuration rules to apply in harvesting process.....	52
Table 7: Select available options to parse successive pages with HTML harvester	53
Table 8: Parsing multiple URLs provided as landing pages in a catalogue.....	54
Table 9: Form validation checks for empty and incorrect values	55
Table 10: Fields, language and country, defined as drop-down button in HTML harvester's registration form	56
Table 11: Declare lists for languages and countries as helper functions	57
Table 12: Initialize lists with default language and country values.....	57
Table 13: Quantity metrics	58
Table 14: Quality metrics	58

1 INTRODUCTION

1.1 Scope and purpose

Recognising the increasing availability of open data and the interest for their exploitation, the OpenDataMonitor (ODM) project has designed and developed a platform (<http://opendatamonitor.eu>) that will enable interested stakeholders to gain an overview of this evolving open data “landscape”. To that end, the research and development activities undertaken throughout the ODM project, focus on the main directions outlined below:

- facilitating and automating the collection of metadata from open data catalogues via an extensible and customizable harvesting framework;
- cleaning and integrating the raw collected metadata, overcoming the high heterogeneity of schemas, values and formats found in the various open data sources, via an integration and harmonisation workflow;
- allowing users to browse and explore the results in an intuitive and user-friendly way, obtaining a comprehensive overview of the collected information, via the computation and visualisation of various metrics and comparative reports.

This document focuses on the overall architecture of the ODM platform, and specifically it reports the status of the implementation for each of the main components involved. It constitutes a follow-up of the respective [Deliverable D3.3](#) (which covered the work up to M12 of the project), presenting the extended and new functionalities added to the ODM platform components during the second year of the project.

Thus, for completeness, in the rest of this section we briefly outline the overall architecture and the main components of the ODM platform. Then, throughout Sections 2-6, we present and explain the progress of the design and development work undertaken in each of the main components, namely the catalogue registry, the metadata harvester, the metadata harmonisation engine, the analysis engine, and the administration panel, respectively. Finally, Section 7 summarises and concludes the report.

1.2 Architecture overview of the ODM platform

1.2.1 Main components

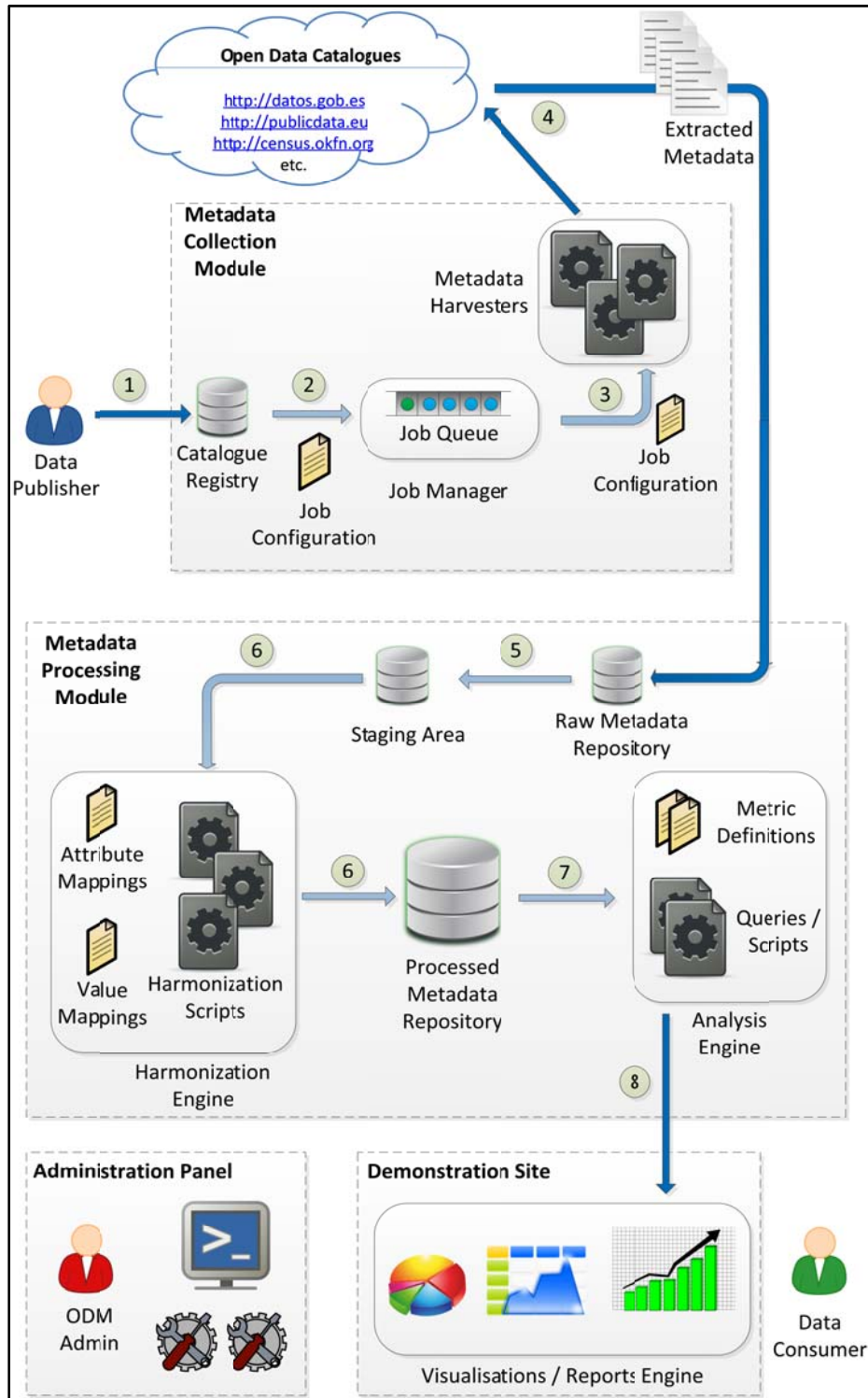


Figure 1: Overview of architecture and processing workflow.

Figure 1 shows an overview of the ODM platform architecture, presenting the main components and illustrating the metadata processing workflow, as was initially designed in the beginning of the project and presented in [Deliverable D3.3](#). The progress of the work throughout the second year of the project has adhered to this architecture, without any major changes or deviations.

Below, we briefly outline the main modules and components comprising the ODM platform:

- **Metadata collection module.** This module is responsible for collecting metadata from a list of registered open data catalogues. It consists of the following main components:
 - **Catalogue registry.** It allows the registration of catalogues for harvesting and monitoring. Registration is done via a Web-based User Interface (UI), where a form is completed with basic information about the registered catalogue, as well as some additional information that is needed in order to setup and configure the respective harvesting process for this catalogue (See section 2 for detailed information). This information provided during the registration step forms the catalogue profile and is stored in the catalogue registry.
 - **Job manager.** The Job Manager schedules the execution of harvesting jobs, periodically or on demand, and is responsible for monitoring their process and reporting the status of execution. A harvesting job is a task that collects metadata from a registered open data catalogue. It provides the required configuration that drives the harvesting process (e.g., which harvester to use and a set of metadata extraction rules to be applied). Harvesting jobs are maintained in a queue and are scheduled for processing.
 - **Metadata harvesters.** These are scripts executed by harvesting jobs in order to perform the actual extraction of metadata from the respective catalogue. Different harvesters are implemented and used to address the different open data platforms and APIs that exist. The configuration included in the harvesting job specifies which harvester should be used and how.
- **Metadata processing module.** This module performs the cleaning, integration and analysis of the metadata that are extracted from the various catalogues that are being monitored. It consists of the following main components:
 - **Harmonisation engine.** It 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 defined for monitoring (Refer to D3.7 for detailed information). 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. For a more detailed description of the demonstration site, and a report on the status of its implementation, see [Deliverable D3.4](#) and forthcoming deliverable D3.7.

1.2.2 Outline of the processing workflow

Next, we describe the main steps of the processing workflow. These steps are also illustrated in Figure 1 (see numbered arrows). As noted above, this workflow was first designed and presented in D3.3, and has also been followed throughout the subsequent development efforts in the second period of the project without any major changes or deviations.

Step 1: Catalogue registration. The first step of the process is to register a new open data catalogue for monitoring. This is done via a Web-based UI, which presents a form requesting several attributes that have to be filled in order to indicate the profile of the catalogue and to guide the metadata extraction 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* de-queues 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* 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 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* (www.opendatamonitor.eu), which produces various charts, visualisations and reports for the end user. The API provides both the metadata records themselves (e.g. all the metadata of the datasets in a given catalogue), as well as aggregate results for various metrics (e.g. the number of datasets uploaded in the previous month).

2 CATALOGUE REGISTRY

The *Catalogue Registry* is the component through which new open data catalogues are registered for monitoring. During the second period of the project, the functionality of this component has been enhanced by both updating and extending tools implemented during the first year of the project and by implementing new tools and functionalities to overcome difficulties that arose in the workflow process and to cover new needs and requirements that came up. More specifically, the main changes can be outlined as follows:

- added a registration form for the newly included Socrata¹ harvester
- added RDF support for the HTML harvester
- added a handler for HTML pages to use pagination in JavaScript snippets
- implemented various minor modifications to cover derived requirements

Next, we present the work done in the Catalogue Registry in more detail.

2.1 Registration form for Socrata harvester

The set of harvesters supported by the ODM platform was extended to include also a harvester for Socrata catalogues. This decision was made because the Socrata platform is widely used in hosting catalogues with open data. Once the Socrata harvester was included, the catalogue registration process was updated to include this new option, as shown in Figure 2. Details about the implementation of the Socrata harvester are provided in Section 3.1.

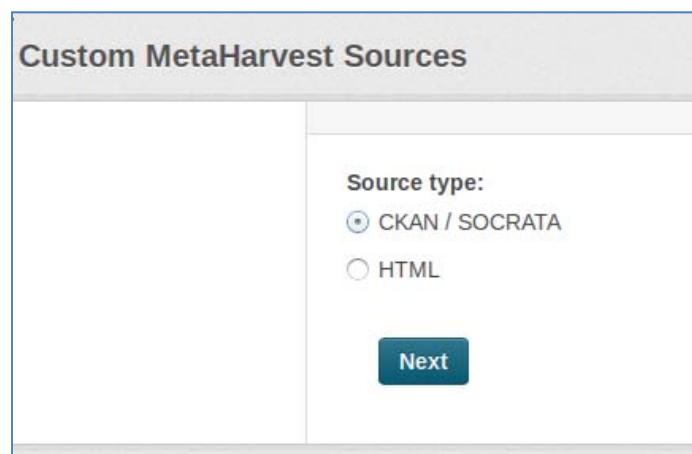


Figure 2: Additional option for registering a catalogue using the Socrata platform.

In Table 5, we list the code snippet that registers the existing options for available harvesters in our platform's registration form. In order to include a new harvester, we simply need to modify the code to provide a new radio button. This radio button should contain a descriptive name for the harvester

¹ <http://www.socrata.com/>

and the path where it is actually installed in the server. A minimum requirement is that any new harvester should implement the template provided by the `ckanext-harvest`. The code snippet listed in Table 5 is part of the `ckanext-htmlharvest` CKAN plugin, which is publicly available under <https://github.com/opendatamonitor/ckanext-htmlharvest>, specifically in `templates/snippets/add_htmlharvest_button.html`.

When the CKAN/Socrata harvester option is selected, the user is transferred to the corresponding form where the needed information is filled in to subsequently configure the harvester to collect the metadata. As shown in Figure 3, the form comprises the same fields as the one for the harvester used for CKAN platforms. This is in order to achieve a consistent template for the process. Thus, the details for the fields included in the form and how to fill them are as described in D3.3 (in particular, Section 3.1). This also explains why both the CKAN and the Socrata harvester options are listed together in the registration form.

Nevertheless, since the two harvesters rely on a completely different mechanism underneath to collect metadata from the corresponding catalogues, we need to indicate this in the harvest job that will be created after filling in appropriately the Web form. For this purpose, the *Source type* option is provided in the form. Hence, in this second configuration form, apart from providing and configuring the rules for collecting metadata, we choose also the harvester instance to be used, between CKAN or Socrata (the HTML button in this case is disabled). Afterwards, the Job Manager creates the appropriate harvester instance according to the provided configuration options.

2.2 RDF support for the HTML harvester

Some catalogues provided the metadata of the listed datasets in RDF format. This is very useful, since it can facilitate and make more reliable the metadata collection process, compared to extracting the metadata from the HTML source code of the page describing the dataset.

For this purpose, the HTML harvester was extended to support collecting metadata in RDF format. As described above, the registration form comprises two successive parts: the first is used to fill in general information related to the catalogue, whereas the second is used to configure rules based on the input from the user. However, instead of defining each metadata attribute we need to harvest separately, the form is enhanced with a new capability, namely the option to provide the RDF link to each dataset's metadata, as shown in Figure 4. This simplifies the process of defining rules to extract metadata from HTML pages, whenever it is supported by the catalogue. Specifically, we need to provide the label that is used in the HTML page to describe the link that contains the metadata of a certain dataset in RDF format.

Create Harvest Source

Catalogue URL:

This should include the http:// part of the URL (eg. <http://data.gov.uk/>)

Creation Date:

Last Update Date:

Language:

This should include the Catalogue's Language

Country:




This should include the Catalogue's Country

Title:

URL:

Description:

You can use [Markdown formatting](#) here

Source type: ☐ CKAN  ☒ Socrata  ☐ HTML 

Update Frequency:

Configuration:

```
{
  "api_version": 1,
  "default_tags": [],
  "default_groups": [""],
  "default_extras": {"new_extra": "Test", "harvest_url": "{harvest_source_url}/dataset/{dataset_id}"}
```

Figure 3: Web form for configuring the harvester for the registered catalogue.

During the fetch stage of the harvesting process, we choose which type of rules is used for the metadata extraction. Here, we either use the link to RDF-based metadata, if available, or specific rules provided by the user for each of the supported metadata attributes. In Section 3.2.1, we explain how this part of the harvester works, and ways to further extend it in the future.

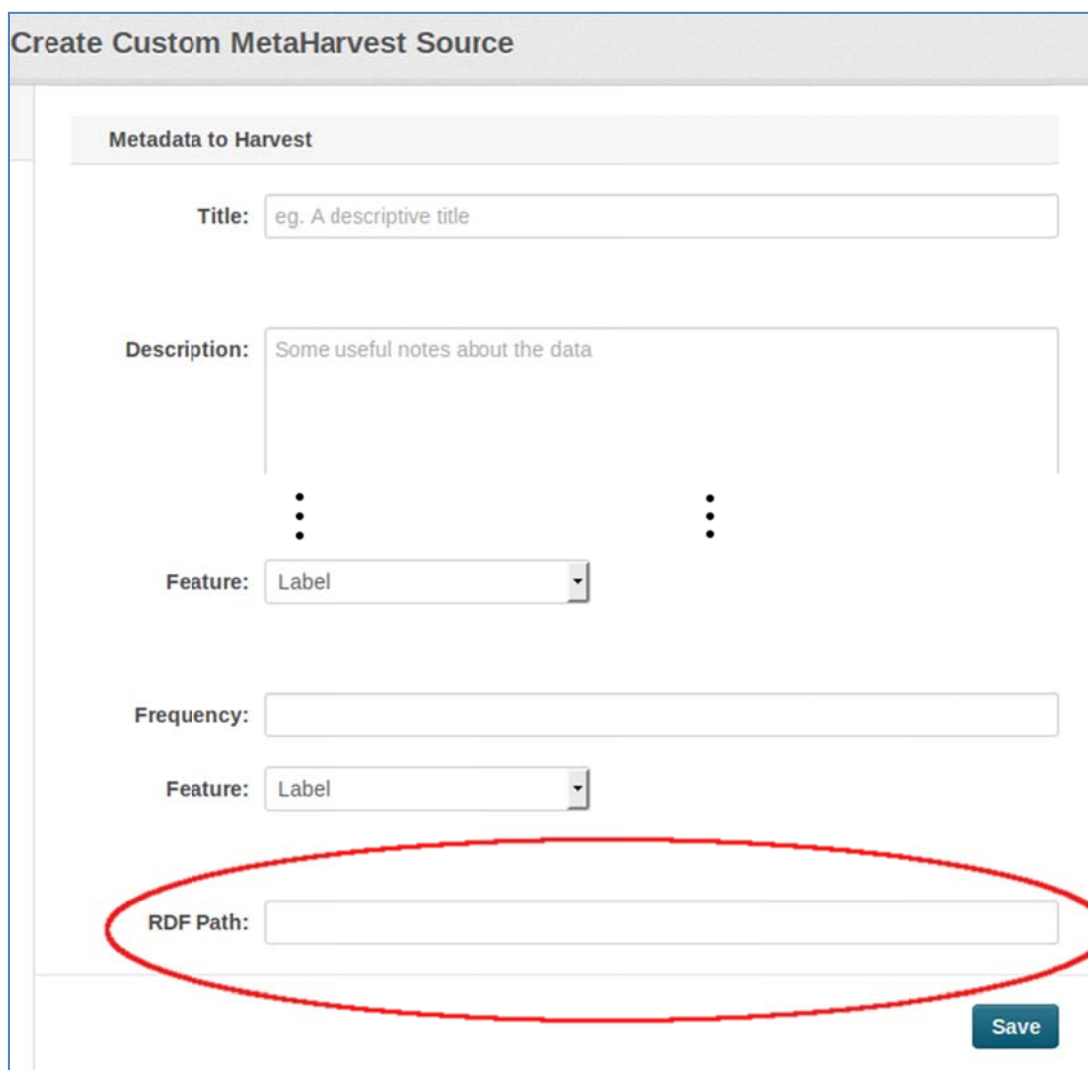
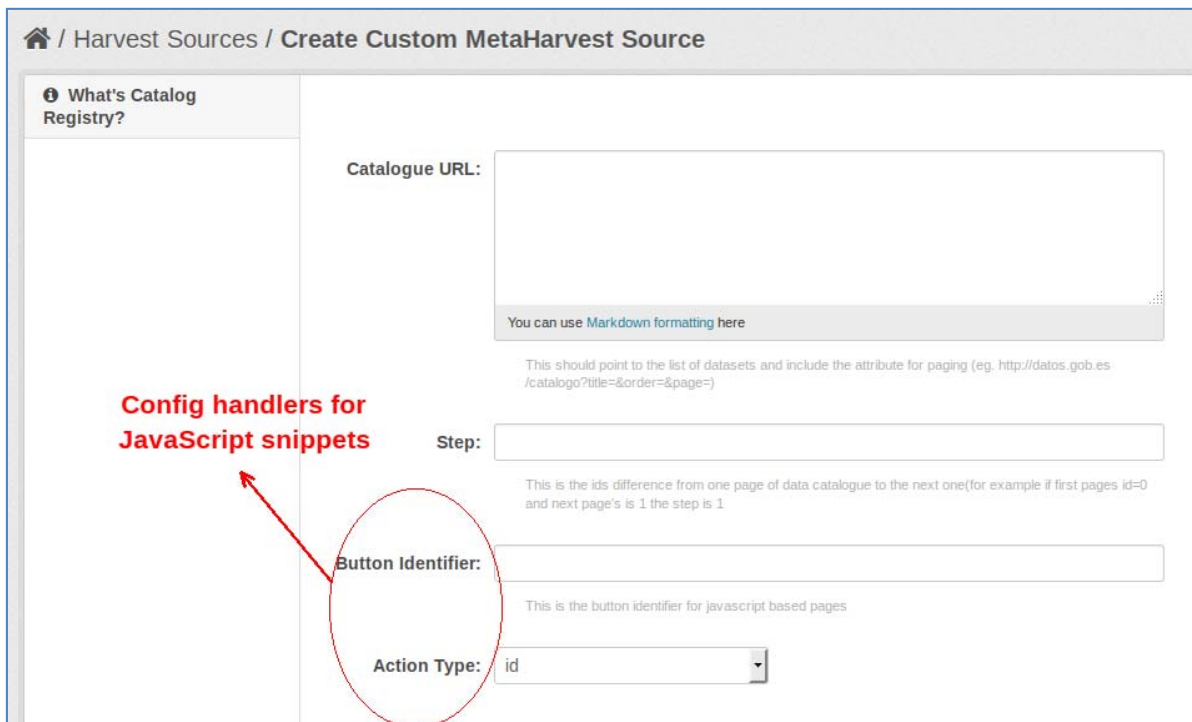


Figure 4: Option to provide link to metadata available in RDF format.

2.3 Handler for HTML pages where JavaScript snippets are used for paging

Another issue that was prohibitive in using the HTML harvester for metadata extraction in some catalogues was the use of JavaScript for certain operations in the respective HTML pages, e.g. for navigating or for presenting a dataset's metadata. Next, we explain how this problem was addressed.

Certain catalogues do not provide a static way to navigate through paging and to collect all dataset URLs pointing to their metadata. One such example is the Multicouncil Open Data catalogue: <http://opendata.cloudbcn.cat/MULTI/es/catalog/>. If we try to identify a pattern to automatically navigate through all pages, we get something like this: `javascript:__doPostBack('ctl00$ContentPlaceHolder1$DataPager1$ctl01$ctl01','')`. This denotes that JavaScript code is used, which prevented the HTML harvester from automatically navigating to subsequent pages beyond the first one.



Home / Harvest Sources / Create Custom MetaHarvest Source

What's Catalog Registry?

Catalogue URL:

You can use [Markdown formatting](#) here

This should point to the list of datasets and include the attribute for paging (eg. `http://datos.gob.es/catalogo?title=&order=&page=`)

Step:

This is the ids difference from one page of data catalogue to the next one(for example if first pages id=0 and next page's is 1 the step is 1)

Button Identifier:

This is the button identifier for javascript based pages

Action Type:

Config handlers for JavaScript snippets

Figure 5: Configuring page navigation in the case of JavaScript code.

As a result, the registration form for the HTML harvester was changed accordingly to handle pages containing JavaScript snippets. Figure 5 displays the two fields that need to be filled in. Now, instead of providing the *Step* value that is the part of the URL that two successive static HTML pages differ on, the user needs to provide the following information:

- *Button identifier*: this is the tag, numbers or set of characters that is placed on links used for paginating (in Figure 6, it is one of the numbers 2,3,4,5 or *Siguiente*)
- *Action Type*: in this dropdown menu, we choose the identifier used in the HTML code to describe the above tag. The accepted values are: *id*, *class* and *link*. However, we must carefully choose an identifier that uniquely describes the above selected tag. Otherwise the reverse process of recovering the marked tag from the source code will be ambiguous. To find it, we need to mouse hover onto the tag we chose in the previous step and go to the source code of the page. For instance, in Firefox, we can perform this by using the built-in source code editor. It is called *Inspect Element (Q)* and can be found in the context-menu of the right-click mouse operation. In our example, as we see in Figure 6, the identifiers that could be used to refer to the *Siguiente* element is either *class* or *href* (*link* in our case). Nevertheless, we choose the *href* type to describe our element since the *class* type is used, for both *Siguiente* and `"javascript: __doPostBack('ctl00$ContentPlaceHolder1$DataPager1$ctl02$ctl01','")"` element, with the exact same content, i.e. *nextprevious*.

The configuration provided in the registration form is used in the gather stage of the HTML harvester. The selection to use or not the JavaScript way of finding all the pages that contain the actual URLs of

hosted datasets is based on whether or not certain fields are filled in or not, as we can see in Table 7. This is discussed in more detail in Section 3.2.2.

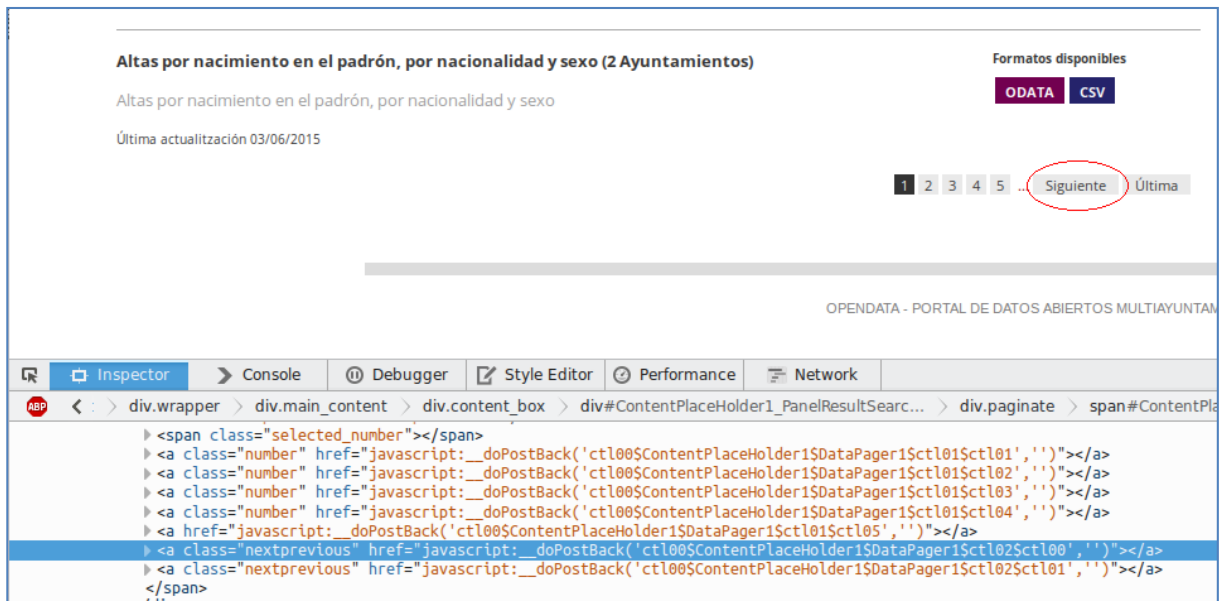


Figure 6: Example of configuring page navigation when JavaScript code is used.

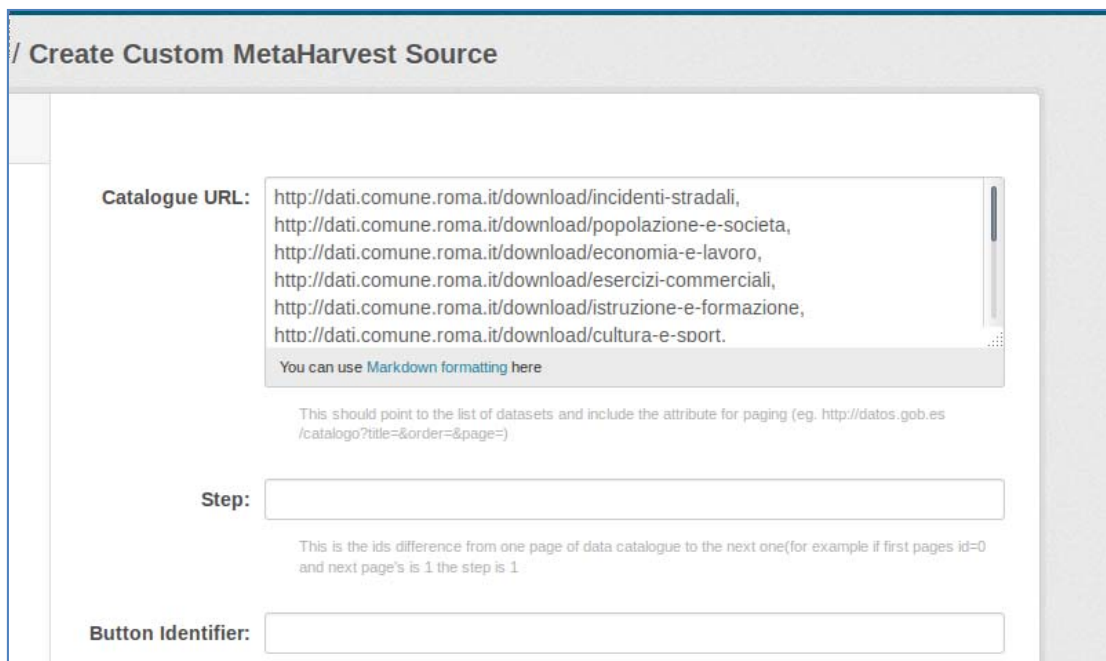
2.4 Other enhancements and modifications

Many catalogues, which do not provide any API compatible to the currently available harvesters in the ODM platform, are harvested via the HTML harvester. One case that came up in practice was catalogues that do not expose their datasets under a root landing page hierarchically, but instead present the available datasets in groups based on custom criteria (e.g. category) where more than one levels of searching are required to access the actual metadata for harvesting.

For instance, the Roma Capitale open data catalogue (<http://dati.comune.roma.it>) provides the datasets under the <http://dati.comune.roma.it/download> URL, grouped in 11 categories. Thus, to collect the metadata we are interested in, we need to follow each one of the above clickable groups.

To achieve this, we modified the *Catalogue URL* field in the HTML harvester form (see Figure 7) to allow for providing more than one URLs, separated with commas. This way, the harvester will now parse all provided URLs to successively collect the metadata. This extension enabled us to support an additional number of catalogues, without any significant impact on the already existing code. In Table 8, we see that no change was needed in order to parse each of the provided URLs in the *Catalogue URL* field. This is because the *Catalogue URL* input from the form is stored in the variable *cat_urls* that is an array. To parse its content a *While* loop is used, thus allowing to handle any number of inputs. However there was a change in the way that the content of the text attribute in

the form is parsed. Again in Table 8, we check if the splitting delimiter exists (in our case, a comma, since by default no URL contains that) and proceed respectively.



Create Custom MetaHarvest Source

Catalogue URL:

[You can use Markdown formatting here](#)

This should point to the list of datasets and include the attribute for paging (eg. `http://datos.gob.es/catalogo?title=&order=&page=`)

Step:

This is the ids difference from one page of data catalogue to the next one (for example if first pages id=0 and next page's is 1 the step is 1)

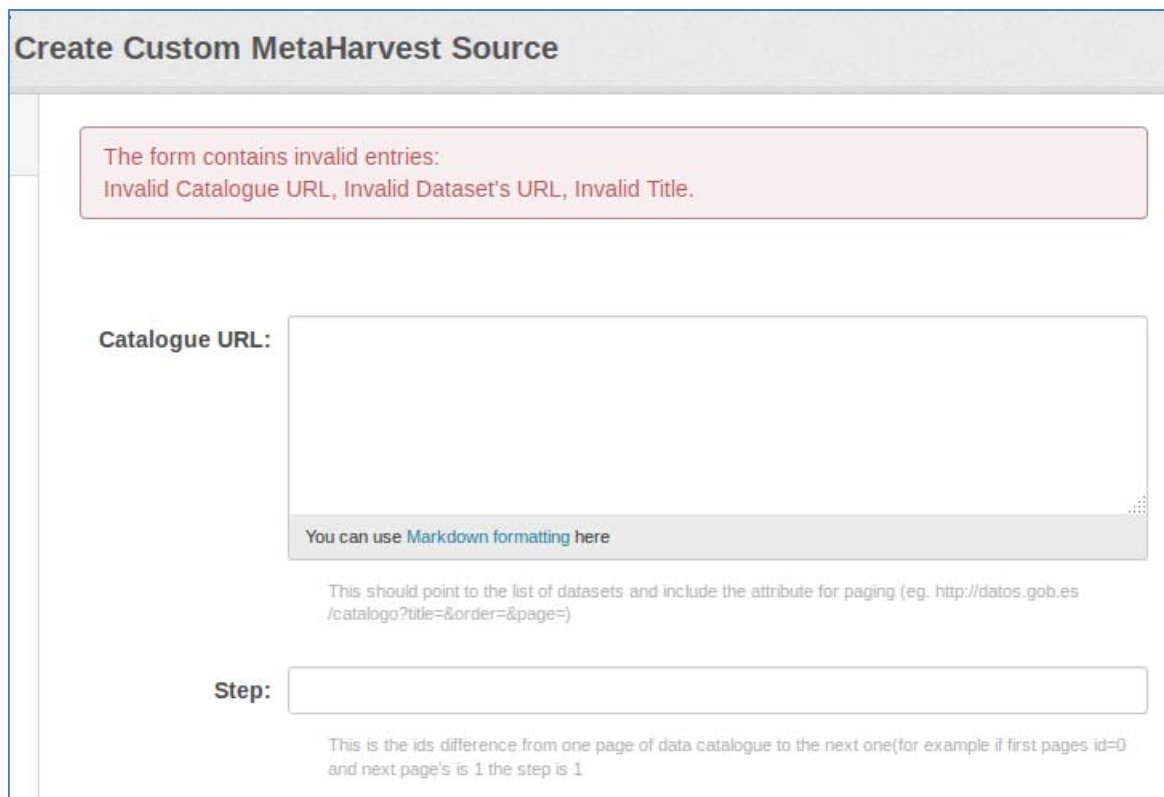
Button Identifier:

Figure 7: Allowing multiple URLs as starting pages for metadata collection within a catalogue.

Another change was to apply validation rules that check if the information provided by the user in the registration form for the HTML harvester adheres to some basic constraints. As already described in Deliverable D3.3, the HTML harvester is configured through completing the catalogue registration in two steps. During the first step, we provide information related to the catalogue itself. In the second step we construct rules based on user input regarding how to access the metadata. Some of these fields, namely the *Catalogue URL*, the *Dataset's URL* and the *Title*, are mandatory before proceeding to the second page. We added code to check for this constraint; in case any of the above information is missing or invalid, a red label with an error message is displayed to the user, as shown in Figure 8. This ensures that all information for each successfully registered catalogue will have undergone a basic validation check before completing the registration process.

In the CKAN platform, we can easily integrate validation rules in a Web form. The *render* function is used to create HTML instances from existing templates. This function takes two arguments as input. The first one is the path to the template and the second one is a variable called *extra_vars*. The *extra_vars* variable is a dictionary with two keys: 'data' and 'errors'. The 'data' key is used to pass information between different templates in the platform, while the 'errors' key is used to include validation errors related to field values of a specific form. So, as shown in Table 9, we can assign to the *errors* key of the *extra_vars* variable information related to entries with missing values. Thus, when we call the *render* function, the web form will contain a red caption with errors and missing

values. Therefore, in case we want to extend the validation rules for a form, we need to update the *extra_vars* variable with the specific rules.



The screenshot shows a web form titled "Create Custom MetaHarvest Source". At the top, a red-bordered box contains the message: "The form contains invalid entries: Invalid Catalogue URL, Invalid Dataset's URL, Invalid Title." Below this, the form has two main sections. The first section is labeled "Catalogue URL:" and contains a large text input field. Below the input field, there is a small grey box with the text "You can use [Markdown formatting here](#)". Below that, a smaller text box contains the instruction: "This should point to the list of datasets and include the attribute for paging (eg. [http://datos.gob.es/catalogo?title=&order=&page=](#))". The second section is labeled "Step:" and contains a text input field. Below this field, a smaller text box contains the instruction: "This is the ids difference from one page of data catalogue to the next one(for example if first pages id=0 and next page's is 1 the step is 1)".

Figure 8: Example of added validation checks in the catalogue registration form.

Finally, we added drop-down menus in all harvesters' registration forms related to: *Language* and *Country*, as shown in Figure 3. This was included in order to allow collecting some more detailed information for the catalogue during its registration. The alternative, as was initially planned, was to automatically extract this kind of information during the metadata extraction; however, only a few datasets actually contain such information. Thus, we decided to collect it explicitly during new catalogues' registration, since the additional effort required by the user is negligible, while allowing to avoid errors and to increase the accuracy and credibility. The relevant code snippets in Python are listed in Table 12. We initialized the above lists with a set of values for European countries² and languages³. Afterwards, we declared them as helper functions⁴ (see Table 11), in order to be used within the templates of the ODM platform. Finally, these functions were used in each of the registration forms to enable the drop-down buttons with available values (see Table 10, presenting such a declaration for the HTML harvester).

² <http://www.countrycallingcodes.com/iso-country-codes/europe-codes.php>

³ <http://publications.europa.eu/code/en/en-5000800.htm>

⁴ <http://docs.ckan.org/en/847-new-theming-docs/template-helper-functions.html>

3 METADATA HARVESTER

The ODM platform comprises a set of harvesters which are responsible for performing the task of metadata extraction from the registered catalogues. During the first period of the project, we focused on adapting a CKAN plugin for retrieving metadata from CKAN-based catalogues, as well as on designing and developing a generic harvester that extracts metadata from the HTML source code of the pages describing a catalogues' datasets (see Deliverable D3.3 for details). During the second period, we have included a Socrata harvester in the list of available harvesters, and we have also implemented various additions and enhancements on the HTML harvester. We describe these aspects next.

3.1 Socrata harvester

This harvester is based on the `socrata-harvester` extension⁵, which allows a host CKAN instance to collect and import metadata information from datasets hosted on a catalogue built on the Socrata platform. For this purpose, we used Socrata's built-in SODA API to find the metadata we need to harvest. This extension is based on the harvest template provided by the `ckanext-harvest` extension⁶. Thus, the harvesting process follows the same procedure as the other harvesters, comprising the following main steps:

1. *gather*: the `/api/dcat.json` page, suffix to the base URL of the catalogue, is accessed which gives the list of all dataset IDs in the catalogue
2. *fetch*: through the SODA API, the metadata for each of the above dataset IDs in the list are retrieved
3. *import*: all retrieved metadata are stored in the internal database of the host CKAN instance; In this step, we apply the mappings to our internal schema.

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. This involves also some content manipulation to replace certain special characters or keywords that are not allowed when importing the data in the database.

Another change concerns the fact that the `socrata-harvester` extension does not handle properly duplicate metadata records during harvesting. So, firstly, we query the database and retrieve all stored metadata ids related to a specific catalogue. Then, after we harvested the new metadata from the catalogue, we compare each of the collected ids with the ones retrieved from the

⁵ <https://github.com/socrata/socrata-harvester>

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

database. Records with new ids are directly stored in the raw metadata repository. For each existing id, we replace older ones with new ones and we change the field `metadata_updated` to the date that a harvesting process runs. This way, we avoid having any duplicates, and also all the metadata records are updated and synchronised to the metadata hosted in the original catalogue.

A sample JSON document containing the metadata of a dataset retrieved by the *Socrata Harvester* is shown in Table 2.

3.2 HTML harvester enhancements

Throughout the course of the project, as more catalogues were being registered and added for harvesting, some issues came up requiring improvements and enhancements on the HTML harvester. Some of these have already been partially mentioned in Section 2. These improvements enabled us to increase not only the number of new catalogues (for more visit the online platform) covered but also the accuracy of the harvesting process. In what follows, we present in more detail the changes and improvements made in the way that the HTML harvester operates.

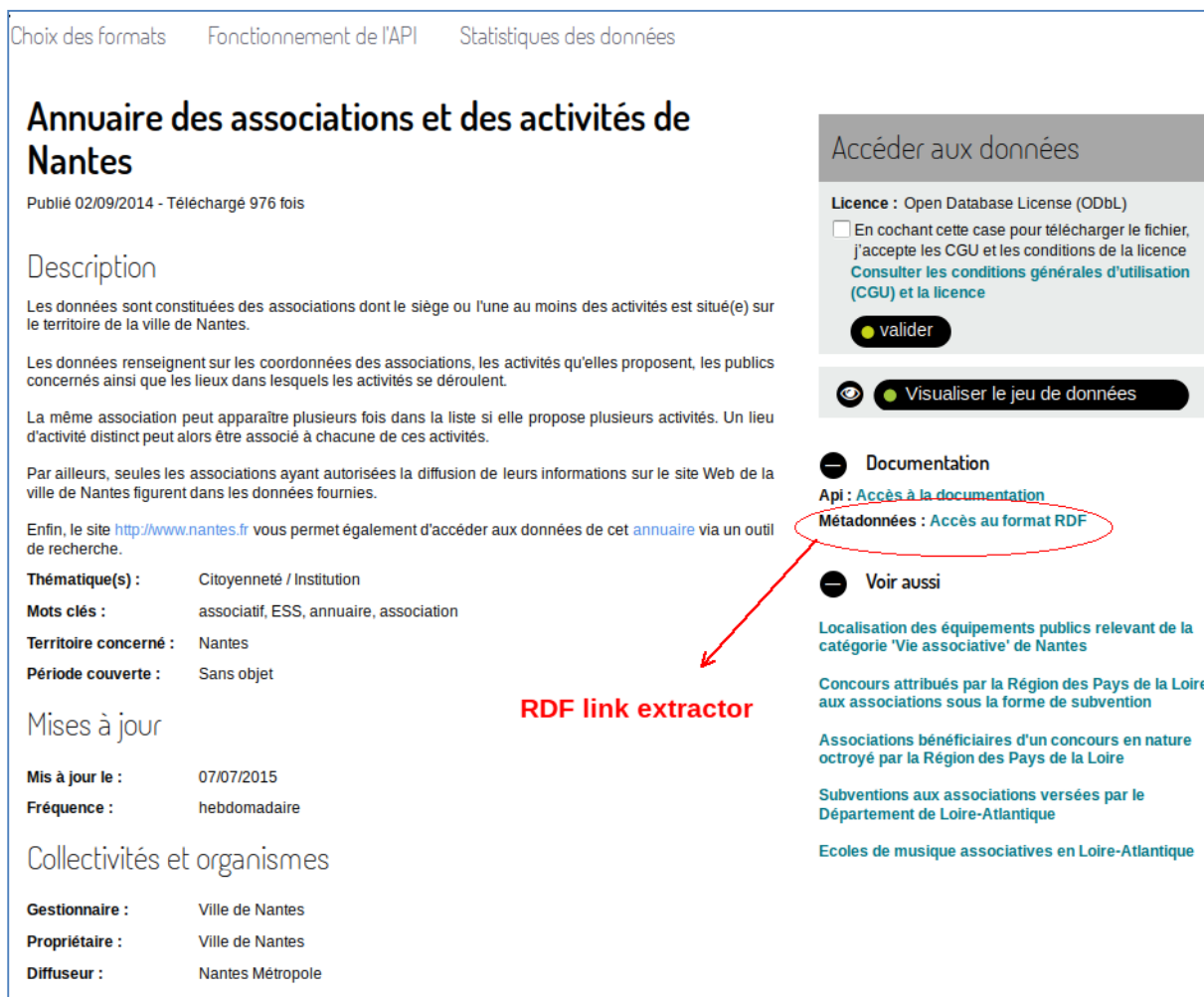
3.2.1 Ability to process RDF content

RDF is a general purpose data model to describe information on the Web. Although many catalogues do not provide an API compatible to what was already supported by the ODM platform, they do publish metadata about their datasets in RDF format. Thus, we decided to enhance the HTML harvester's functionality to increase its accuracy and completeness, by enabling it to collect and process metadata in RDF format, when available, instead of relying on HTML scraping.

In this case, the overall metadata collection and processing still follows the general steps described in D3.3; however, instead of defining rules for each of the attributes to harvest, we use the RDF description. Figure 9 shows one such example from the Loire-Atlantique open data catalogue (<http://data.loire-atlantique.fr /accueil>). For this example, the RDF content can be seen in Table 3. We can see that all information contained in the HTML page is structurally presented in the RDF link. Afterwards, we parse the content with the `xmlltodict`⁷ Python library, which manipulates the XML content as JSON. Finally, we need to map every extracted attribute and its value to our internal schema. Default mappings have been included for the case that the RDF description of the metadata follows the DCAT vocabulary. If that is true, the mappings are applied automatically and the process of harvesting is successfully finalized. Otherwise, there is a need to include custom mapping rules to

⁷ <https://pypi.python.org/pypi/xmlltodict>

the catalogue's schema. In that case, the code that needs to be changed is in the file `RdfToJson.py`, which is available in the project's Github repository⁸.



Choix des formats Fonctionnement de l'API Statistiques des données

Annuaire des associations et des activités de Nantes

Publié 02/09/2014 - Téléchargé 976 fois

Description

Les données sont constituées des associations dont le siège ou l'une au moins des activités est situé(e) sur le territoire de la ville de Nantes.

Les données renseignent sur les coordonnées des associations, les activités qu'elles proposent, les publics concernés ainsi que les lieux dans lesquels les activités se déroulent.

La même association peut apparaître plusieurs fois dans la liste si elle propose plusieurs activités. Un lieu d'activité distinct peut alors être associé à chacune de ces activités.

Par ailleurs, seules les associations ayant autorisées la diffusion de leurs informations sur le site Web de la ville de Nantes figurent dans les données fournies.

Enfin, le site <http://www.nantes.fr> vous permet également d'accéder aux données de cet annuaire via un outil de recherche.

Thématique(s) : Citoyenneté / Institution

Mots clés : associatif, ESS, annuaire, association

Territoire concerné : Nantes

Période couverte : Sans objet

Mises à jour

Mis à jour le : 07/07/2015

Fréquence : hebdomadaire

Collectivités et organismes

Gestionnaire : Ville de Nantes

Propriétaire : Ville de Nantes

Diffuseur : Nantes Métropole

Accéder aux données

Licence : Open Database License (ODbL)

☐ En cochant cette case pour télécharger le fichier, j'accepte les CGU et les conditions de la licence [Consulter les conditions générales d'utilisation \(CGU\) et la licence](#)

valider

Visualiser le jeu de données

Documentation

Api : [Accès à la documentation](#)

Métadonnées : Accès au format RDF

Voir aussi

[Localisation des équipements publics relevant de la catégorie 'Vie associative' de Nantes](#)

[Concours attribués par la Région des Pays de la Loire aux associations sous la forme de subvention](#)

[Associations bénéficiaires d'un concours en nature octroyé par la Région des Pays de la Loire](#)

[Subventions aux associations versées par le Département de Loire-Atlantique](#)

[Ecoles de musique associatives en Loire-Atlantique](#)

RDF link extractor

Figure 9: Link to RDF description of a dataset's metadata.

The rules defined during the registration process are used in the fetch stage of the harvesting. We select one of the pre-defined methods for collecting the meta-attributes by checking whether or not the 'RDF Path' label has a valid value. The check is performed with the `if 'rdf' in rules.keys()` code, shown in Table 6, where 'rdf' is the internal name that is used in the code to describe the specific field in the form.

3.2.2 Ability to process JavaScript code

The gather stage of the HTML harvester is used to collect all available URLs of the datasets. In this stage, we need to go through all the different pages in which these URLs are listed. In the case that a

⁸ <https://github.com/opendatamonitor/ckanext-htmlharvest/blob/master/ckanext/htmlharvest/harvesters/RdfToJson.py>

catalogue uses a navigation system based on JavaScript snippets in order to reach and retrieve each hosted dataset, we use a different mechanism. During catalogue registration, we label the catalogue as such a case and provide inputs in specific fields, as explained in Section 2.3, which are used to handle this type of situations.

To address this issue, we used the `Selenium`⁹ Python library, which allows to execute code enclosed in a JavaScript snippet. In our case, this made it possible to perform automatic paging in the catalogue. However, specific technical issues had to be overcome in order to support this functionality. In particular, this involved the selection of the browser to use that would actually execute the snippet, while being able to run on a server where no GUI environment exists. Our first try was with the PhantomJS¹⁰. This browser is capable of performing all typical tasks related to a browser, including the execution of JavaScript code, without the need to load a GUI. However, after few experiments, we concluded that this software was not mature enough to cover our needs and to perform JavaScript execution successfully. Subsequently, we resorted to the use of the Mozilla's Firefox¹¹ web browser, and specifically the Python library `pyvirtualdisplay`¹², which made it possible to run headless Python Selenium/WebDriver tests in our ODM server. Now that we set up and configured the tools to handle issues related to our problem, we can continue with describing the process performed by the harvester.

First, the harvester takes the value from the `'btn_identifier'` field provided in the registration form. This value, in combination with the value from *Action Type* field, is used to search in the HTML code and retrieve the JavaScript code. Then, it is provided as input to the `Selenium` library and thus executing the code to access the pages with the dataset URL links. This way, the harvester goes through all the pages of the catalogues until there everything is collected. Two types of values can be filled in `'btn_identifier'` that change slightly the process:

- *number* fields, i.e. 1,2,3..., that navigates directly to certain page;
- *text button*, (in the above example, 'Siguiente'), that navigates to the next page that follows the current one.

In the first case, we need to modify the JavaScript code to access all the pages. For instance, in <http://opendata.cloudbcn.cat/MULTI/es/catalog/> catalogue, the code for the first page with datasets is the following:

```
javascript: __doPostBack( 'ctl00$ContentPlaceHolder1$DataPager1$ctl01$  
ctl00', '' ). To access next pages, we need to increment it by one, i.e.  
javascript: __doPostBack( 'ctl00$ContentPlaceHolder1$DataPager1$ctl01$
```

⁹ <https://pypi.python.org/pypi/selenium>

¹⁰ <http://phantomjs.org/>

¹¹ <https://www.mozilla.org/en-US/firefox/products/>

¹² <https://pypi.python.org/pypi/PyVirtualDisplay>

`ctl01', '')`, in order to go to the second page. We do this each time we want to navigate to another page with URLs of datasets. On the other case, things are simpler. Since that button by default navigates to the next page, we only need to retrieve the embedded JavaScript code. Thus, each time we want to navigate to the next page, we only need to execute again and again the same code with the Selenium library. In our example, the reported code from the text button field is: `javascript:__doPostBack('ctl00$ContentPlaceHolder1$DataPager1$ctl02$ctl00', '')`, which is parsed as it is every time from the Selenium. Finally, the harvester returns when all metadata collected. This happens when the executed code returns an empty page, first case, or the identifier button is missing, in the second case.

3.2.3 Assessment of enhancements

Based on the improvements and enhancements described above, we managed to increase the number of catalogues being harvested, reaching a number of 150 distinct catalogues from 24 European countries. Figure 10 shows the distribution of catalogues that are handled by each type of harvester.

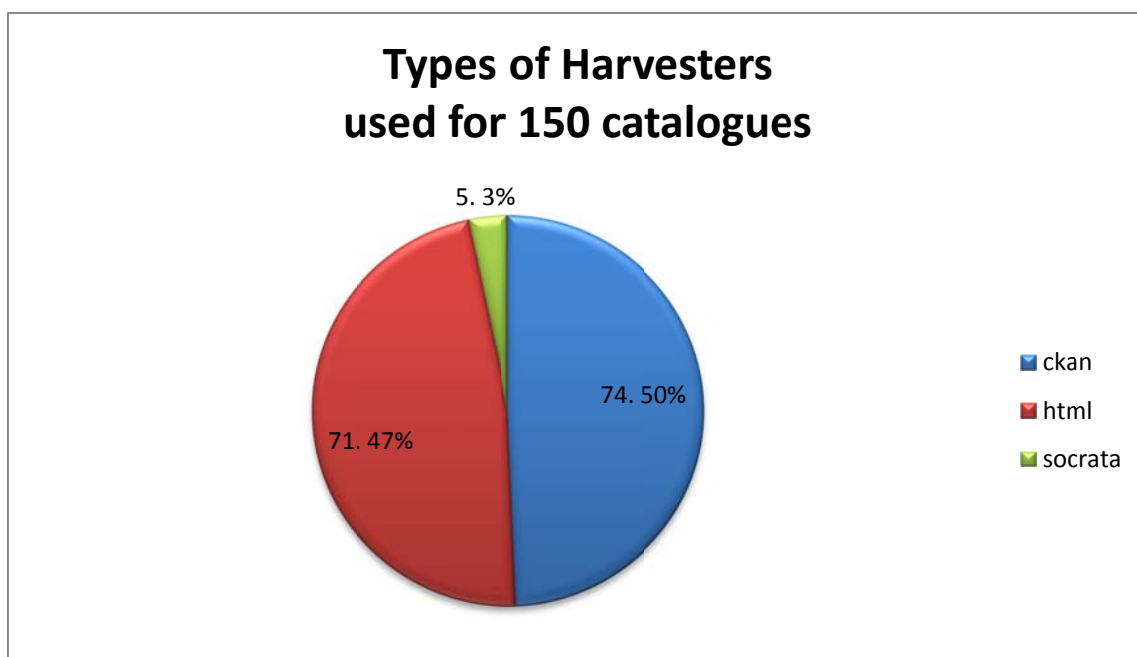


Figure 10: Distribution of catalogues handled by each harvester type.

Moreover, in Figure 11, we see the portion of catalogues where the paging is implemented with JavaScript code. As shown, the enhancement of the HTML harvester to process JavaScript code for paging, allowed to process a significant number of additional catalogues that could not be successfully harvested with the previous implementation.

Based on this improved version of the ODM harvesters, we are currently able to collect metadata from 150 catalogues. Figure 12 shows the distribution of the number of datasets collected from the monitored catalogues.

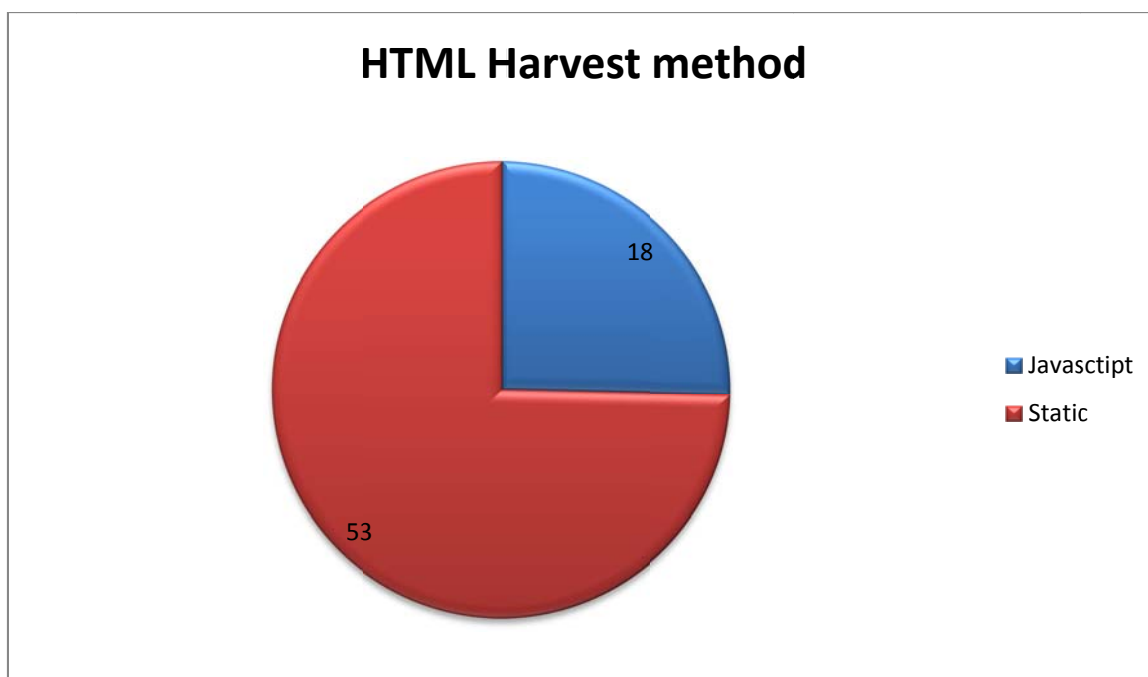


Figure 11: Portion of catalogues handled by the HTML harvester using Javascript code.

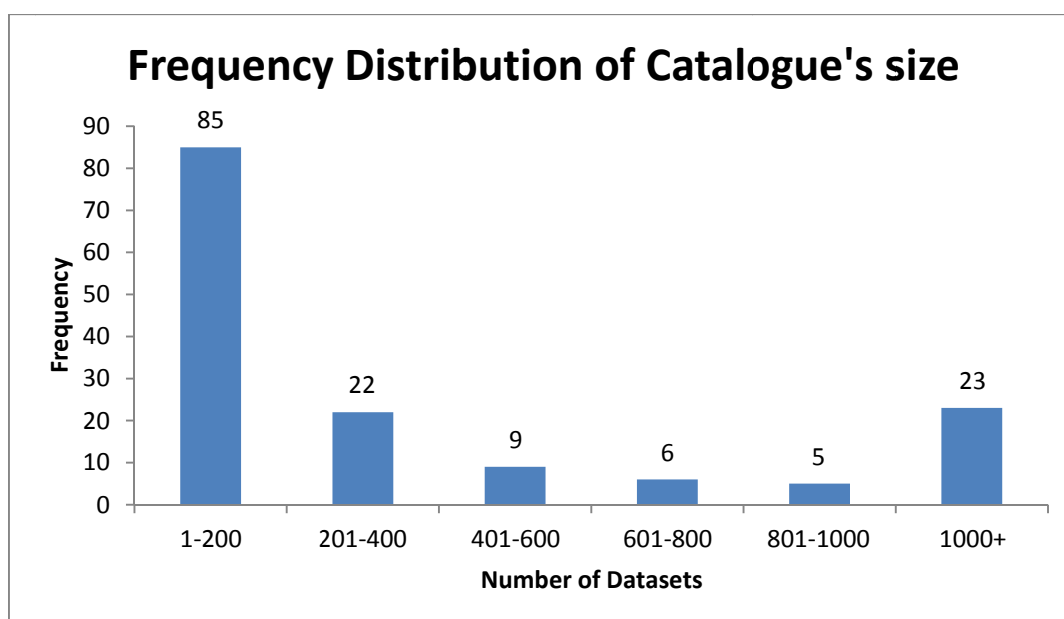


Figure 12: Number of datasets collected from the monitored catalogues.

Figure 13 shows the presence of each metadata attribute in the collected datasets. This provides useful and important insights about the completeness of the metadata provided from the datasets related to a minimum defined number of metadata attributes. As observed, we can identify 5 attributes, namely title, resources, notes, license_id and tags, which are most often present in the collected metadata. On the contrary, other attributes, such as author, category and organization, are more rarely present. Finally, the rest of them, are very rarely provided in the metadata. Therefore, these cannot be effectively used in any calculation or metric implementation providing useful and reliable results.

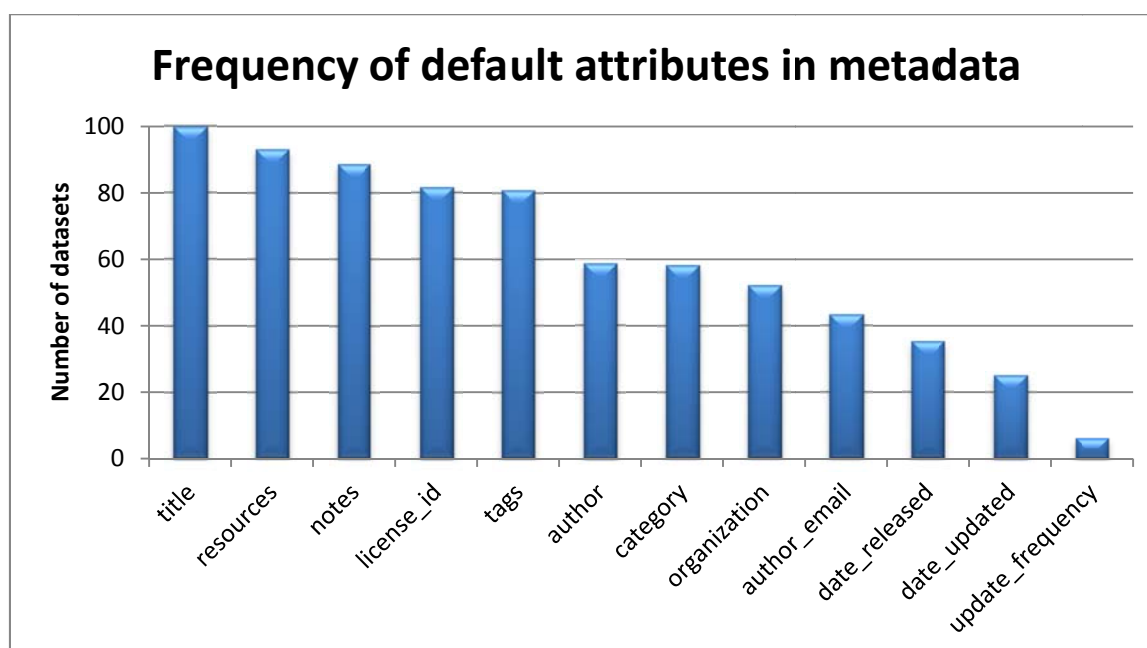
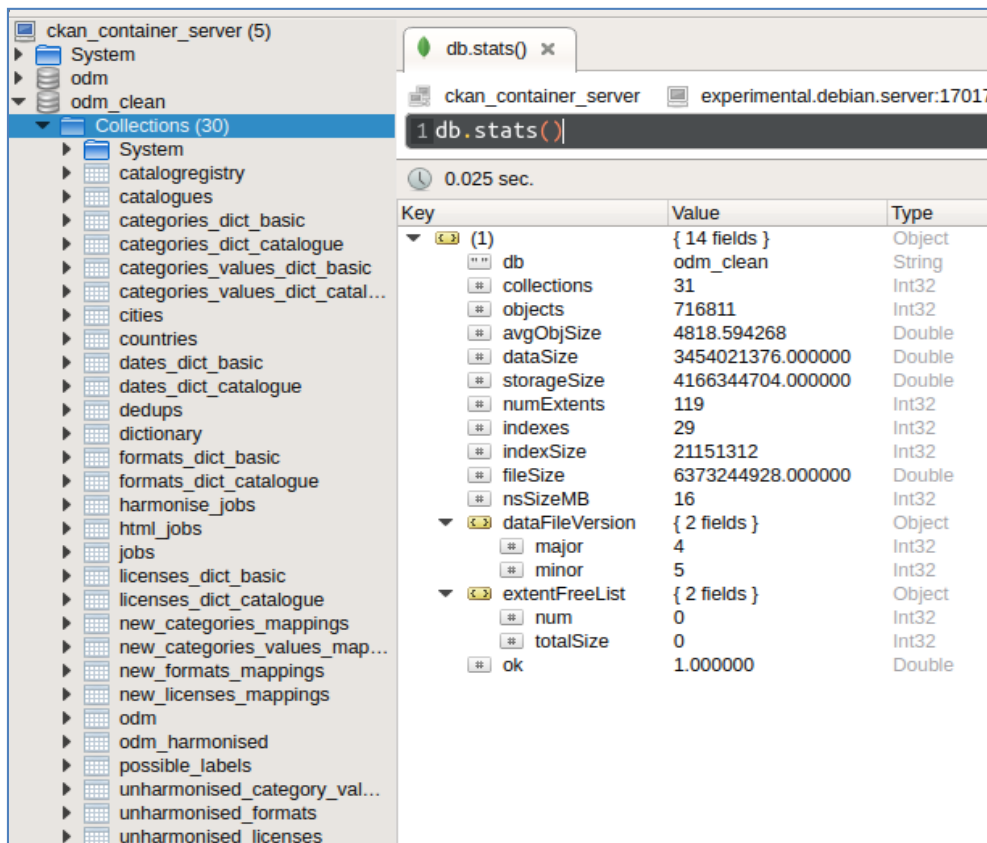


Figure 13: Availability of each metadata attribute in the collected datasets.

Finally, in Figure 14, we see an overall view of the database content. On the left side, we can view an instance of existing collections in the DB and on the right side we get several overall statistics. For instance, the overall storage size of the DB in disk is around 4GB, 92% of which is occupied by the collections *odm* and *odm_harmonised*, which are the ones containing the raw and the harmonised (see next section) metadata, respectively.



db.stats() x

ckan_container_server experimental.debian.server:17017

```
1 db.stats()
```

0.025 sec.

Key	Value	Type
(1)	{ 14 fields }	Object
db	odm_clean	String
collections	31	Int32
objects	716811	Int32
avgObjSize	4818.594268	Double
dataSize	3454021376.000000	Double
storageSize	4166344704.000000	Double
numExtents	119	Int32
indexes	29	Int32
indexSize	21151312	Int32
fileSize	6373244928.000000	Double
nsSizeMB	16	Int32
dataFileVersion	{ 2 fields }	Object
major	4	Int32
minor	5	Int32
extentFreeList	{ 2 fields }	Object
num	0	Int32
totalSize	0	Int32
ok	1.000000	Double

Figure 14: View of the database for metadata collection and processing.

4 HARMONISATION ENGINE

An important part of the system is the Harmonization Engine, which is the module responsible for integrating and reconciling the raw metadata collected from the monitored catalogues into an internal representation. This allows for further processing and analysis, in particular for the calculation of the various metrics employed for monitoring. Some first steps of this process were undertaken during the first period of the project (see Deliverable D3.3). During the second period, the harmonization process has been extended to a fully functional component with multiple levels of mappings and different processing handlers. Additional components were also implemented to remove existing duplicate metadata. Below we describe these enhancements and changes in detail.

4.1 De-duplication module

The various catalogues that have been included in the system for harvesting and monitoring come from different geographical levels around Europe. Thus, the list includes catalogues ranging from regional level to national and pan-European. Although this improves coverage, a problem that arises is that duplicate datasets often exist among the monitored catalogues, since a catalogue at a higher regional level (e.g., national) may often aggregate datasets from lower levels (e.g. city-level). As a result, there is a need to identify and exclude duplicates when computing the various metrics in order to avoid bias in the results.

Before describing how we handled the problem, we note that we consider that datasets hosted within the same catalogue to be duplicate free. There were two reasons for making this assumption: (a) an investigation on a sample of the collected metadata did not provide evidence for the opposite, and (b) limiting the duplicate detection task to only consider different catalogues reduces the time and resources required. Nevertheless, it is straightforward to modify this process to also check for duplicates within the same catalogue.

Before we proceed, we need to define the terms *duplicates* and *candidates*. As *duplicate* we describe a pair of metadata stored in the database and refer to the same dataset which exists and harvested from two different catalogues. Respectively, a *candidate* is a pair of metadata that is highly probable to describe the same dataset which again is hosted in different catalogues. Having clarified these terms, we continue on presenting how the module operates.

The de-duplication module consists of two distinct phases:

- *indexing*: this step indexes all initial metadata, as described in more detail later, in order to reduce the number of comparisons that are needed to identify duplicates
- *searching*: this step performs the actual comparisons to identify (potential) duplicates

During the indexing phase, we perform operations that subsequently make it faster and more accurate to identify potential duplicates in the database. Specifically, this includes the following steps, performed for all metadata records:

1. prepare the content for each metadata record that will be used as indicator for identifying duplicates; specifically, we use for this purpose the concatenation of the fields <title> and <notes>;
2. tokenize the content string and remove punctuations, stop words etc. The Whoosh API used for this¹³;
3. calculate the md5sum on the content string on every metadata; this attribute will be used for exact matching (see below);
4. create 4-gram Shingles from content string and calculate *minhash*¹⁴; this attribute will be used for approximation matching (see below);
5. store the above calculated attributes (this is done in the MongoDB database, under the collection *dedups*), and create an indexer on the *minhash* field.

The process is illustrated in the diagram below.

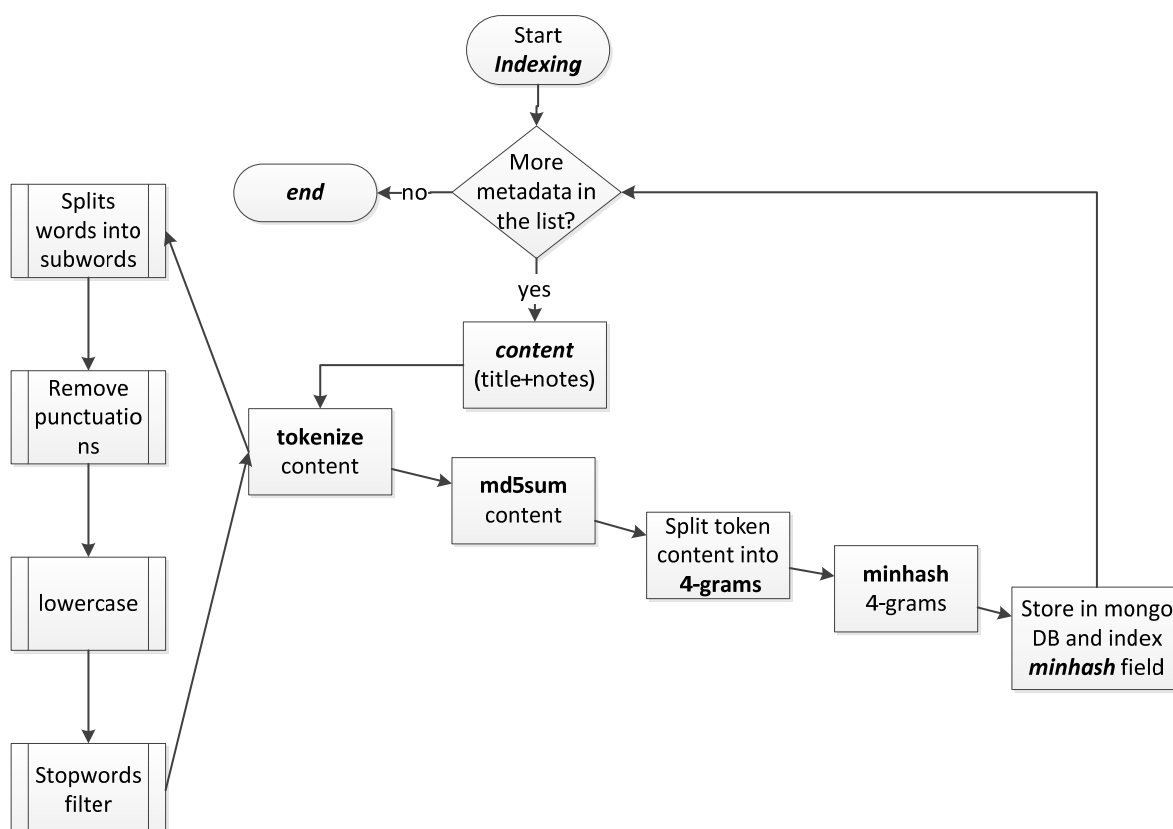


Figure 15: Flow chart for the indexing phase of the de-duplication process.

¹³ <https://pythonhosted.org/Whoosh/api/analysis.html>

¹⁴ <https://github.com/go2starr/lshhdc>

The result of the above process is a fully indexed set of metadata records in our repository that are about to be used by the searching phase.

During the searching phase, we identify and log *candidates* of metadata. The procedure comprises the following steps:

1. select a newly harvested metadata record in the repository;
2. find records with identical *minhash*;
3. apply similarity criteria (see below)
4. *candidates* produced for verification
5. repeat steps 1-4 for every metadata record.

Criteria for similarity

In order to identify candidates among metadata, we check a number of attributes to comply with specific rules. The number and the type of fields that were chosen should fulfil a few requirements. Firstly, the number should be as small as possible to minimize the processing time but also sufficient to produce reliable results. Also they should exist in most of the metadata, ideally in every metadata, and have valid values in order to be useful. Empirically and consulting Figure 13, we ended up with the following attributes: *title*, *notes*, *resources* and *date_updated*. Except for the *date_updated*, the rest are in accordance to our requirements. However, we choose also the *date* because it was necessary, whenever it existed, in automating steps of our process, which will be clear below. For each of the aforementioned attributes, we apply the following checks:

- *Content (<title>+<notes>)*: given the *contents* of a pair of metadata, C and C' , we calculate the edit distance (Levenshtein distance), $dist(C, C')$. The requirement is that the edit distance must be lower or equal to a threshold. An optimization is applied for the special case of $dist(C, C')=0$. We use the *md5sum*, already calculated in the indexing phase, to find equal string. This saves time in string similarity calculations performed with the edit distance algorithm.
- *Resource*: given a pair of metadata, they have a set of resources attached to them, R and R' . A resource $R_1 \in R$ is equal to $R_2 \in R'$ if and only if they have the same URL and size. The following cases are valid:
 - $R \cap R' \neq \emptyset$: the two sets have equal resources and the type of relations between them could be: a) $R = R'$, when the two sets are equal, b) $(R \subset R' \vee R \supset R')$, when R is strict subset to R' or R is strict superset to R' respectively and c) $(R \not\subset R' \vee R \not\supset R')$, when none of them is superset to the other;
 - $R \cap R' = \emptyset \wedge (R \neq \emptyset \vee R' \neq \emptyset)$: the two sets have no common resources and at least one of them is not empty;
 - $R = \emptyset \wedge R' = \emptyset$: both sets are empty.

- *Update date*: given the *date_updated* for a pair of metadata, *DU* and *DU'*, we compare these values. One requirement for the comparison is that both *DRs* field values must exist ($\exists DU \wedge \exists DU'$). In this case, we can have the following:
 - $DU = DU'$: *date_updated* values are equal;
 - $DU \neq DU'$: one of the two dates is newer than the other;

To formulate requirements and cover all alternative conditions for the above rules, we use the decision table presented in Table 1. Based on this, the process results to label each of the metadata records stored in the database as:

- *Unique*: the metadata object is successfully identified as unique (i.e. no candidate duplicates were found) and no further processing is needed;
- *Candidate*: a pair of metadata is marked as candidate and waits for verification.

Table 1: Decision table to label metadata as *Candidate* or *Unique*

Conditions		Resource		Edit distance	Update date		Candidate	Unique
Rules	R1	$R \cap R' = \emptyset \wedge (R \neq \emptyset \vee R' \neq \emptyset)$		-	-	Actions		x
	R2	$R \cap R' \neq \emptyset$	$R = R'$	$dist(C,C')=0$	$DU = DU' \vee (\nexists DU \vee \nexists DU')$		x	
	R3				$DU \neq DU'$		x	
	R4			$dist(C,C') \neq 0$	-		x	
	R5		$R \subset R' \vee R \supset R'$	$dist(C,C')=0$	$DU = DU' \vee (\nexists DU \vee \nexists DU')$		x	
	R6				$DU \neq DU'$		x	
	R7			$dist(C,C')=[1,2]$	-		x	
	R8			$dist(C,C')>2$	-			x
	R9		$R \not\subset R' \vee R \not\supset R'$	$dist(C,C')=0$	-		x	
	R10			$dist(C,C')=[1,2]$	-		x	
	R11			$dist(C,C')>2$	-			x
	R12	$R = \emptyset \wedge R' = \emptyset$		$dist(C,C')=[0,1,2]$	-		x	
	R13			$dist(C,C')>2$	-			x

Finally, having the list of the *candidate* pairs, we need to figure out the pairs which are indeed duplicates and reject those that are not. Moreover, for each of the identified pairs as duplicates, we must label one of the members of the pair as *original*. This flag is used when we need to query the database and take into account only one of the metadata pairs that were found and labelled as duplicates. This distinction is made with one of the following ways:

- *Automatically*: this is the case where the applied rules results in classifying a *candidate* pair as *duplicates* and assigning to one of them the *original* flag. If we look at the decision table, we find that this is the outcome of applying rules R3 and R6;
- *Semi-automatically*: in this case, the *duplicate* pairs are produced automatically. However, this is not also the case for the *original* flag. The rules R2, R5 and R9 result in this occasion. Thus, we need to apply another step that is called *partial ordering*. We construct a hierarchy of importance between catalogues. This means that metadata belonging in a catalogue of greater importance can be considered as *original* to those that come from catalogues lower in the hierarchy. Many things can result in characterizing a catalogue as more important to another one. For instance, it could be how trustworthy or up to date is, if it is considered as official for a country or even the geographical area that is covered from the hosted datasets;
- *Manually*: this is the trivial case, where each of the *candidate* pairs needs to be manually verified as *duplicate*. This is the result when the rest of the rules in Table 1 are applied.

The *partial ordering* is saved in a file called `partialOrder.csv`. An example of such a file is presented in Table 4, where the vertical line '|', separates the catalogue on the left that are superseded by those on the right. The right part of declaration can contain more than one catalogues which are separated by comma, ','.

The whole process is illustrated in the diagram below.

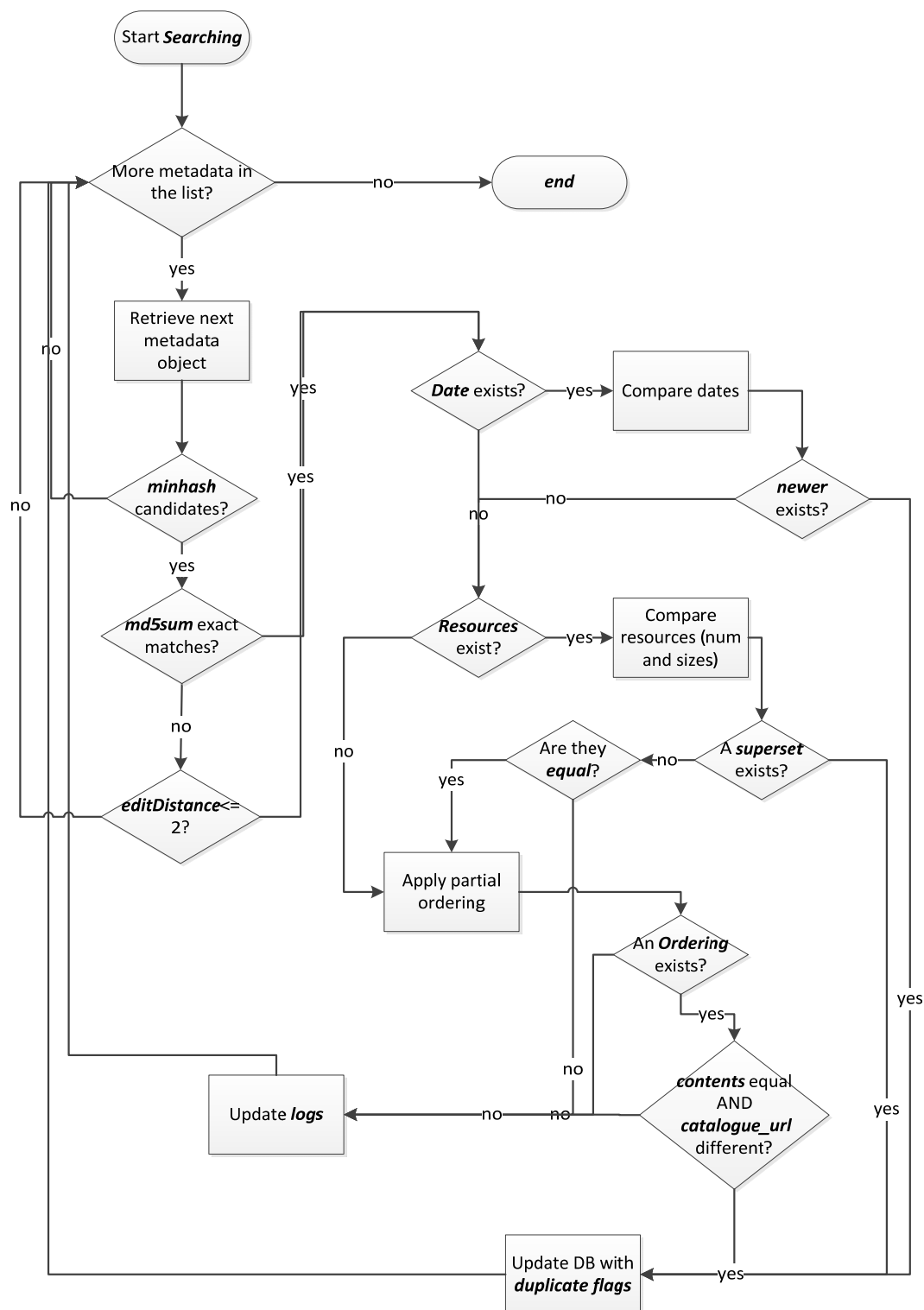
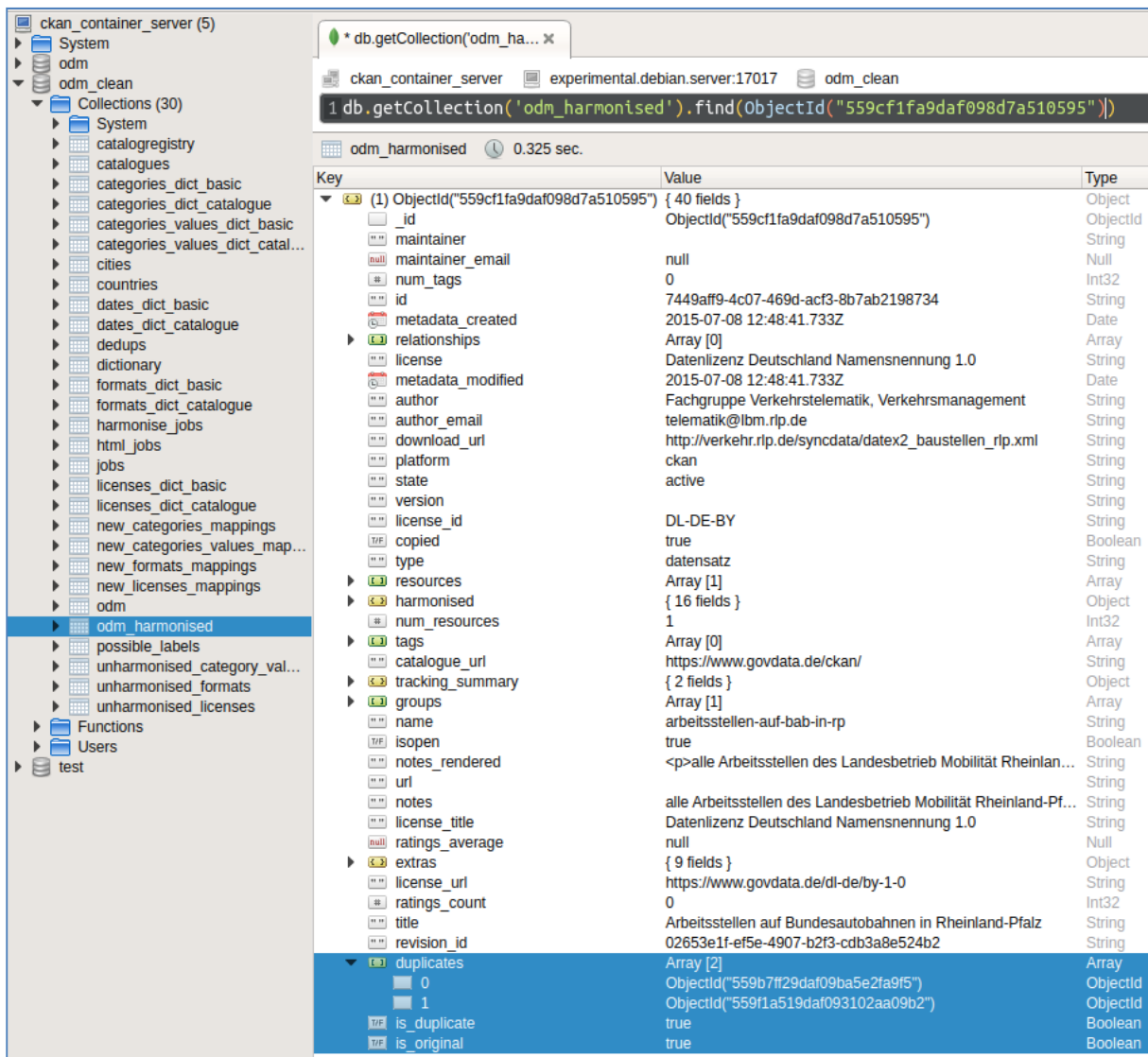


Figure 16: Flow chart for the searching phase of the de-duplication process.

Finally, we modify each metadata object in our harmonised instance of the collected metadata to mark our findings, as in Figure 17. The following meta-attribute fields are used for this:

- *is_duplicate*: this flag is set to true when the metadata is labels as duplicate. Otherwise it is set to false.
- *duplicates*: it is an array of ids of the metadata that are found to be duplicates to current one;
- *is_original*: this is used as a flag whether this metadata is used as *original* or not. It takes the values *true* or *false*.



Key	Value	Type
(1) ObjectId("559cf1fa9daf098d7a510595")	{ 40 fields }	Object
_id	ObjectId("559cf1fa9daf098d7a510595")	ObjectId
maintainer		String
maintainer_email	null	Null
num_tags	0	Int32
id	7449aff9-4c07-469d-acf3-8b7ab2198734	String
metadata_created	2015-07-08 12:48:41.733Z	Date
relationships	Array [0]	Array
license	Datenlizenz Deutschland Namensnennung 1.0	String
metadata_modified	2015-07-08 12:48:41.733Z	Date
author	Fachgruppe Verkehrstelematik, Verkehrsmanagement	String
author_email	telematik@ibm.rlp.de	String
download_url	http://verkehr.rlp.de/syncdata/datex2_baustellen_rlp.xml	String
platform	ckan	String
state	active	String
version		String
license_id	DL-DE-BY	String
copied	true	Boolean
type	datensatz	String
resources	Array [1]	Array
harmonised	{ 16 fields }	Object
num_resources	1	Int32
tags	Array [0]	Array
catalogue_url	https://www.govdata.de/ckan/	String
tracking_summary	{ 2 fields }	Object
groups	Array [1]	Array
name	arbeitsstellen-auf-bab-in-rp	String
isopen	true	Boolean
notes_rendered	<p>alle Arbeitsstellen des Landesbetrieb Mobilität Rheinlan...	String
url		String
notes	alle Arbeitsstellen des Landesbetrieb Mobilität Rheinland-Pf...	String
license_title	Datenlizenz Deutschland Namensnennung 1.0	String
ratings_average	null	Null
extras	{ 9 fields }	Object
license_url	https://www.govdata.de/dl-de/by-1-0	String
ratings_count	0	Int32
title	Arbeitsstellen auf Bundesautobahnen in Rheinland-Pfalz	String
revision_id	02653e1f-ef5e-4907-b2f3-cdb3a8e524b2	String
duplicates	Array [2]	Array
0	ObjectId("559b7ff29daf09ba5e2fa9f5")	ObjectId
1	ObjectId("559f1a519daf093102aa09b2")	ObjectId
is_duplicate	true	Boolean
is_original	true	Boolean

Figure 17: Example of duplicate metadata objects.

4.2 Harmonization process

The harmonization process is a Python service responsible for checking created harmonization jobs that need to be executed. These jobs are created automatically when the harvesting process for a catalogue finishes. The collection *harmonise_jobs* in the database contains all created jobs. Every such created job stores information that is required to run correctly the process. The fields assigned

to every job are grouped in two categories. In the first category, we have fields that describe which attributes of the collected metadata need to be harmonized.

The following metadata attributes, whenever they exist, are harmonized: dates, formats, mime-types, licenses, categories, languages and countries. On some of these, we harmonize both labels and values. That is, there are cases where the name used to describe the above attributes does not comply with our internal schema. For instance, the attribute *date_released* could be encountered as *publish-date*, *deposit_date*, etc. These fields are the *date*, *categories*, *languages* and *countries*. A dictionary of mappings is used to perform the above transformations¹⁵.

The second one contains the fields that provide information on which metadata to apply the harmonization rules and being able to get an overview of the current status of the process and its execution. Especially, the *id* field is used to collect statistics related to the process of harmonization from another collection, jobs, like when last process is executed, the fields that where successfully harmonized etc. The *cat_url*, references the catalogue whose metadata are about to be harmonized. And finally, the *harmonised* and *status* fields are used by the platform to be aware of the progress of the harmonization process, i.e. whether it has ever been executed for the specific catalogue, whether it is currently running, pending etc. The possible values for the *harmonised* field are:

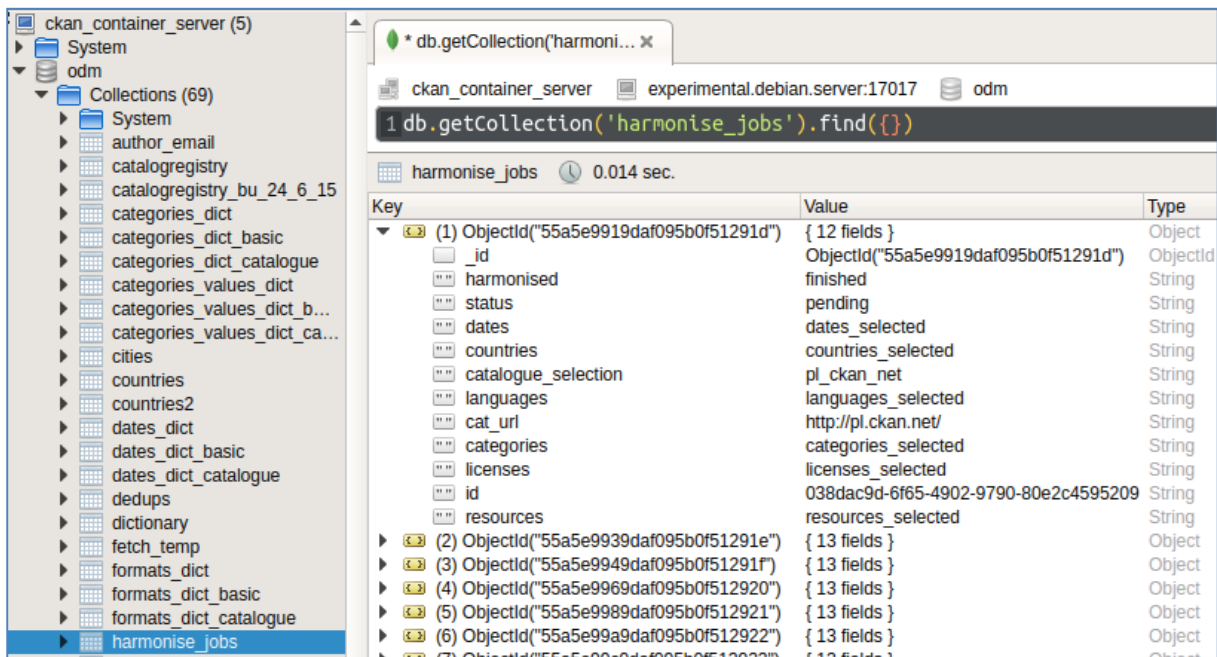
- *pending*, meaning that the process of harmonization has never been executed;
- *started*, meaning that it is currently being executed for the first time;
- *finished*, which means that it has been executed at least once in the past for the specific catalogue.

Similarly, the possible values for the *status* field are:

- *unharmonised*, meaning that the metadata are still in their raw form;
- *pending*, meaning that a harmonization job is in the queue waiting for execution;
- *harmonisation_started*, meaning that the harmonization process is running at the moment;
- *harmonised*, when the harmonization is finished and returned.

In Figure 18, we see an example of such a harmonization job for the Polish open data catalogue (<http://pl.ckan.net>). We can identify that it has already been harmonised once in the past (*harmonised: 'finished'*), and that there is a waiting job to be executed (*status: 'pending'*), first in the queue when the service will be released from its work. The rest of them are referring to the attributes which are actually going to be harmonised, e.g. dates, categories, licenses etc.

¹⁵ <https://github.com/opendatamonitor/ckanext-harmonisation/tree/master/ckanext/harmonisation/controllers/dictionaries>



The screenshot shows the OpenDataMonitor interface. On the left, a tree view shows the database structure under 'ckan_container_server (5)' > 'System' > 'odm' > 'Collections (69)'. The 'harmonise_jobs' collection is selected. The main panel shows a query: `1 db.getCollection('harmonise_jobs').find({})`. Below the query, a table displays the results of the query, showing 7 objects. The first object is expanded, showing its fields and values.

Key	Value	Type
(1) ObjectId("55a5e9919daf095b0f51291d")	{ 12 fields }	Object
_id	ObjectId("55a5e9919daf095b0f51291d")	ObjectId
harmonised	finished	String
status	pending	String
dates	dates_selected	String
countries	countries_selected	String
catalogue_selection	pl_kan_net	String
languages	languages_selected	String
cat_url	http://pl.ckan.net/	String
categories	categories_selected	String
licenses	licenses_selected	String
id	038dac9d-6f65-4902-9790-80e2c4595209	String
resources	resources_selected	String
(2) ObjectId("55a5e9939daf095b0f51291e")	{ 13 fields }	Object
(3) ObjectId("55a5e9949daf095b0f51291f")	{ 13 fields }	Object
(4) ObjectId("55a5e9969daf095b0f512920")	{ 13 fields }	Object
(5) ObjectId("55a5e9989daf095b0f512921")	{ 13 fields }	Object
(6) ObjectId("55a5e99a9daf095b0f512922")	{ 13 fields }	Object
(7) ObjectId("55a5e99c9daf095b0f512923")	{ 13 fields }	Object

Figure 18: Example of harmonization job.

During the harmonisation phase, the newly collected metadata are first transferred into an intermediate database ('odm_harmonised_temp'), in order to initiate the harmonization process. Every metadata object in the odm collection could be in one of three states: *new*, *copied* or *updated*. We transfer all metadata that are new or updated. Before we start to process the temp collection, we add a flag `copied:true` and delete the updated flag for all metadata transferred in the temp collection. Then, we start to apply the harmonization rules to the fields defined in the harmonization job. Each one that is executed is copied to the `odm_harmonised` collection. However, before storing it, we check if the metadata object actually exists from a previously applied harmonization. This is crucial because if it is updated, certain created fields need to be maintained, e.g. duplication flags. After that, we can safely delete the temp collection.

4.3 Levels of applied mappings

The Harmonization Engine applies a set of specified rules (i.e. mappings) in order to reconcile metadata records collected from different catalogues. At the beginning, during the first period of the project, we started by specifying these rules individually for each newly added catalogue. However, clearly a large part of these mappings are applicable for many catalogues, with fewer ones being exceptions that have to be specified separately. Thus, to make this process more easy and efficient to scale and maintain, we have defined a three-level hierarchy of mappings, as described:

1. *top-level*: these are global rules, applied by default to all catalogues; only the administrator can define and change them;

2. *middle-level*: rules in this level apply to a group of catalogues, i.e. catalogues belonging to the same user;
3. *bottom-level*: these are rules that have local scope, i.e. are associated and applied to a specific catalogue.

Having this hierarchy provides much more flexibility for defining, maintaining and applying mapping rules during the harmonization process. Specifically, the rules that are finally applied to each catalogues' fields are constructed from all three levels according to the respective scope. That is, we start with the top-level rules that are applied to a field we are trying to harmonize and then we check successively to find whether any more specific mappings exist from the middle or the bottom level, that are applicable to this specific catalogue. If so, the mapping with the most specific scope overrides the previous ones. Moreover, new mappings that do not exist in the top-level category are included to the mapping list. This enables us to be flexible, being able to specify generally applicable mappings at the top-level while also defining exceptions that are to be applied to specific catalogues or groups of catalogues.

For instance, let's say that our mapping list contains the following: Creative Commons By 3: CC BY-3.0. This is a generally applied mapping to most of the catalogues. Although it is designed to ensure worldwide validity, jurisdictions differ on certain countries¹⁶, e.g. Germany – CC By-3.0 De, Spain—CC BY-3.0 ES etc. Thus, we wanted to easily be able to retain and apply that information in our internal database without affecting the general rules or needing to specify multiple versions of the same license for each of the harvested catalogues. Therefore, we can group together the German catalogues and we can introduce the following mapping: e.g. defining its scope to be this group.

¹⁶ https://meta.wikimedia.org/wiki/Open_Content_-_A_Practical_Guide_to_Using_Creative_Commons_Licences/The_Creative_Commons_licencing_scheme

5 ANALYSIS ENGINE

The harmonized metadata are accessed through the *Analysis Engine*. This component calculates a series of key metrics defined by ODI in the context of Task 2.3, which were enhanced and enriched during the second year of the project and makes the results available to the demonstration site via a RESTful API (initially introduced in Deliverable D3.3). This API is publicly available¹⁷.

These metrics are used to analyze and compare catalogues' metadata and can be grouped in two general categories: Quantity and Quality. The first one is about reporting statistics by counting or summing selected attributes of the harvested metadata. For instance, we count all datasets and distributions or add up the distribution size contained for every catalogue in the database. All implemented API functions for this category are presented in Table 13. On the other hand, the defined quality metrics attempt to quantify how good and useful the collected metadata could be to end users. They combine data from different attributes and report how complete the specific metadata are and how possible it is that they could provide adequate and meaningful data for someone that is interested to use them. For instance, we try to identify the percentage of the datasets of a catalogue that has at least one resource which is in *machine-readable* format, i.e. CSV, TSV, JSON, XML or RDF, or each catalogue's *accessability* which means that the following attributes exist for every metadata object and have valid values: a description, at least one valid link and an author email. Table 14 contains all the functions for quality metrics.

The above metrics are independent meaning that quality does not imply quantity and the same stands the other way around. That means catalogues with a big number of resources does not imply these resources will be provided in any of the so called machine-readable formats. So the metrics are tools that could be used by the end users to find meaningful datasets on what he wants to accomplish. The metrics implemented are basically calculated in two ways:

- Catalogue level, meaning that counts or aggregations are applied on each catalogue;
- European level, which refer to metrics calculated on top of all datasets harvested among the 24 European countries.

The dashboards used in the demonstration site contain another level, the country one. Metrics for this level are calculated based on results provided for each catalogue and grouping all that belong to a specific country.

These metrics were implemented or updated according to newly defined requirements during the second period of the project.

¹⁷ http://odmapi.magellan.imis.athena-innovation.gr/api/v1.0/_commands

6 ADMINISTRATION PANEL

An administration panel was designed and implemented to provide an easier and more intuitive way to control and monitor the main parts of the harvesting and harmonization process. We describe the main parts of this panel below.

6.1 Harvesting panel

In this part, we basically integrated both the HTML and the Socrata based harvesters to the existing panel of the ckanext-harvester. This panel provides an overview of the harvested metadata for each one of the registered and monitored catalogues.

Specifically, there are two tabs, *Datasets* and *About*, as illustrated in Figure 19. The first one provides a summary of all the collected metadata, allowing to navigate to information about individual datasets by clicking on the title of the respective dataset. The second one contains general information about the catalogue, such as its URL, the language, country etc., which is filled in during registration.

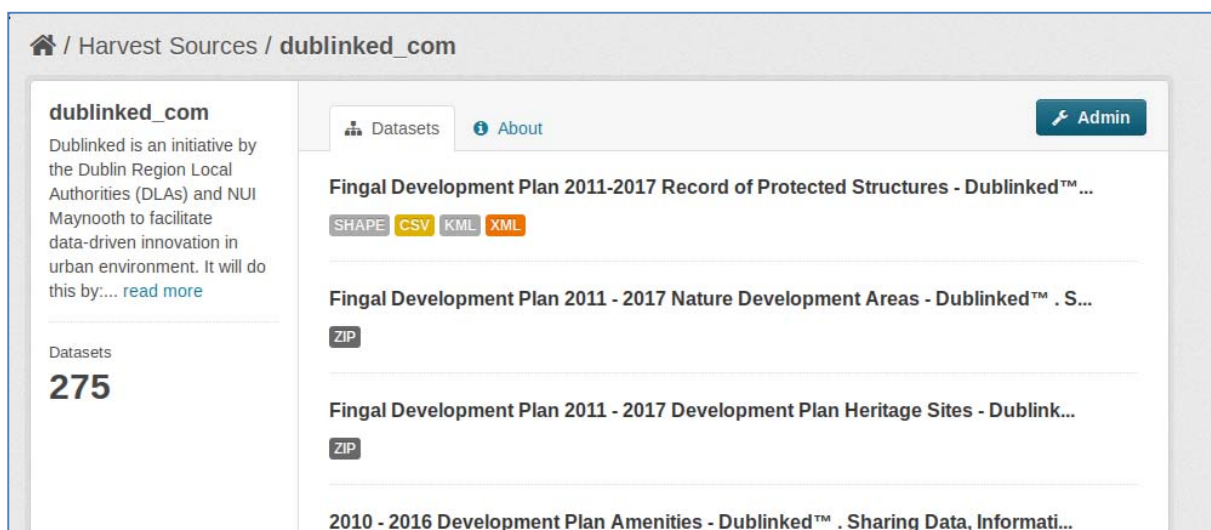


Figure 19: View of the harvesting panel.

Moreover, it provides access to registered users to view information related to the harvesting process through the Admin icon (see Figure 19). This opens up another view that consists of three tabs, *Dashboard*, *Jobs* and *Edit*, as shown in Figure 20. The *Dashboard* tab provides information about the most recent running process, i.e. whether it has finished or is still running, and last date of execution. The *Jobs* tab (see Figure 21) displays a history of all running processes. A detailed history for every such process is provided too, where information about any errors that may have occurred is also included, to facilitate debugging and maintenance.

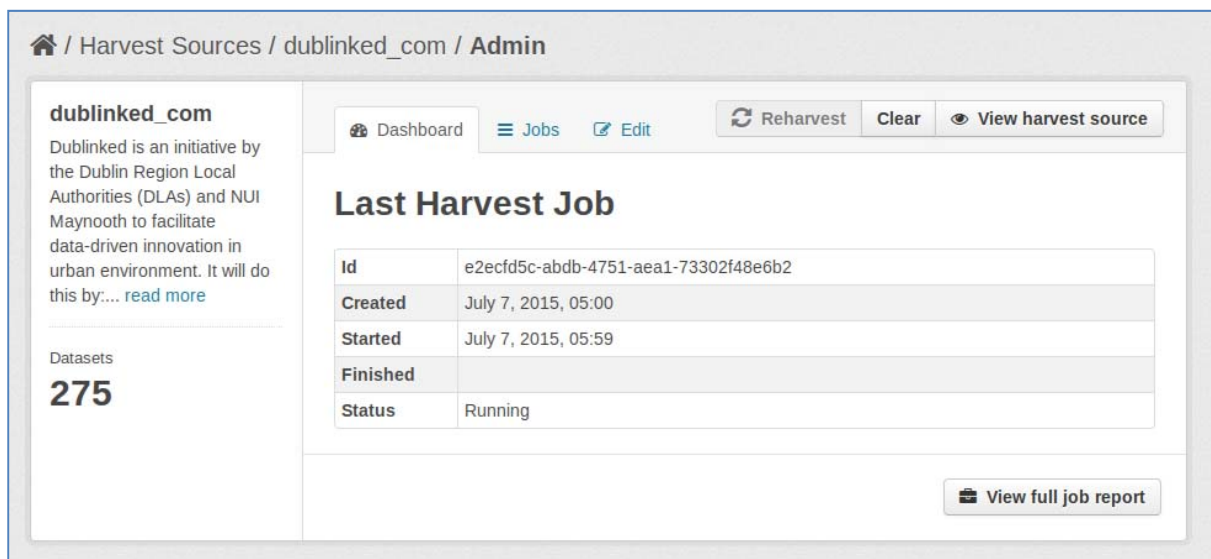


Figure 20: View of the admin panel.

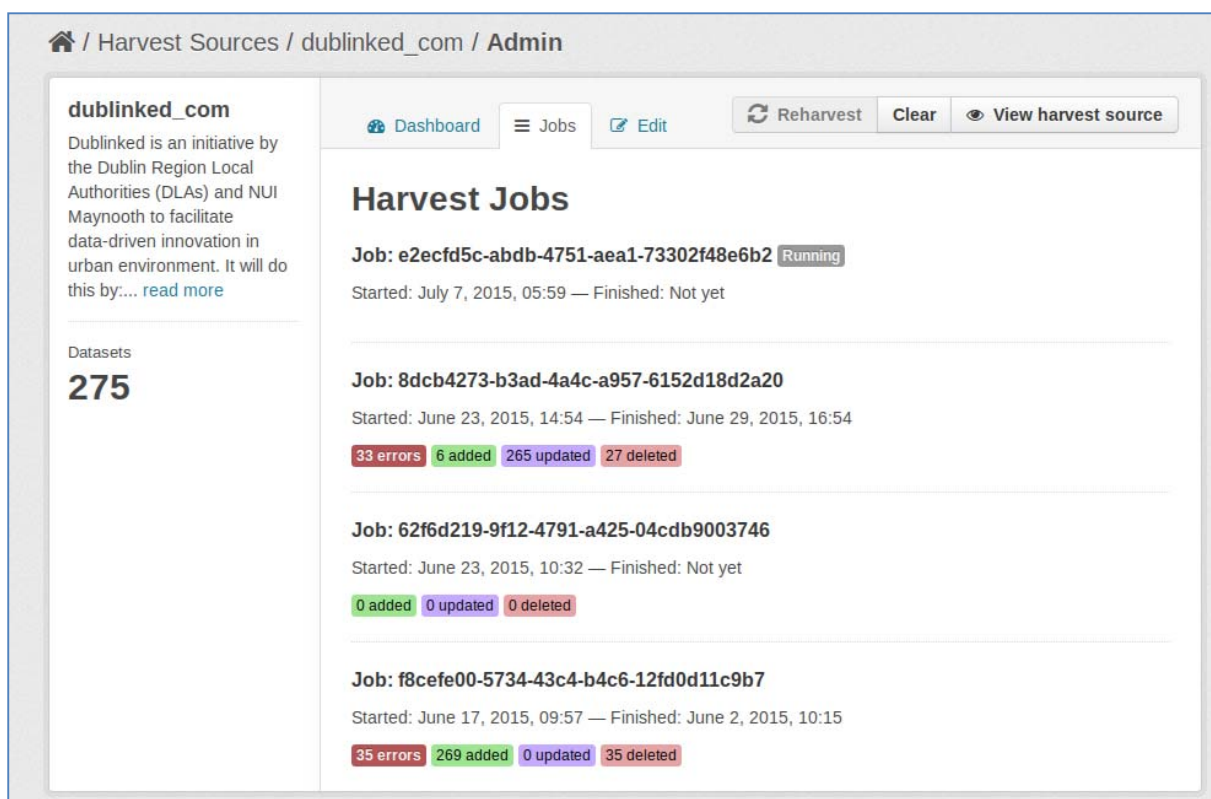


Figure 21: View of the Jobs tab.

6.2 General overview panel

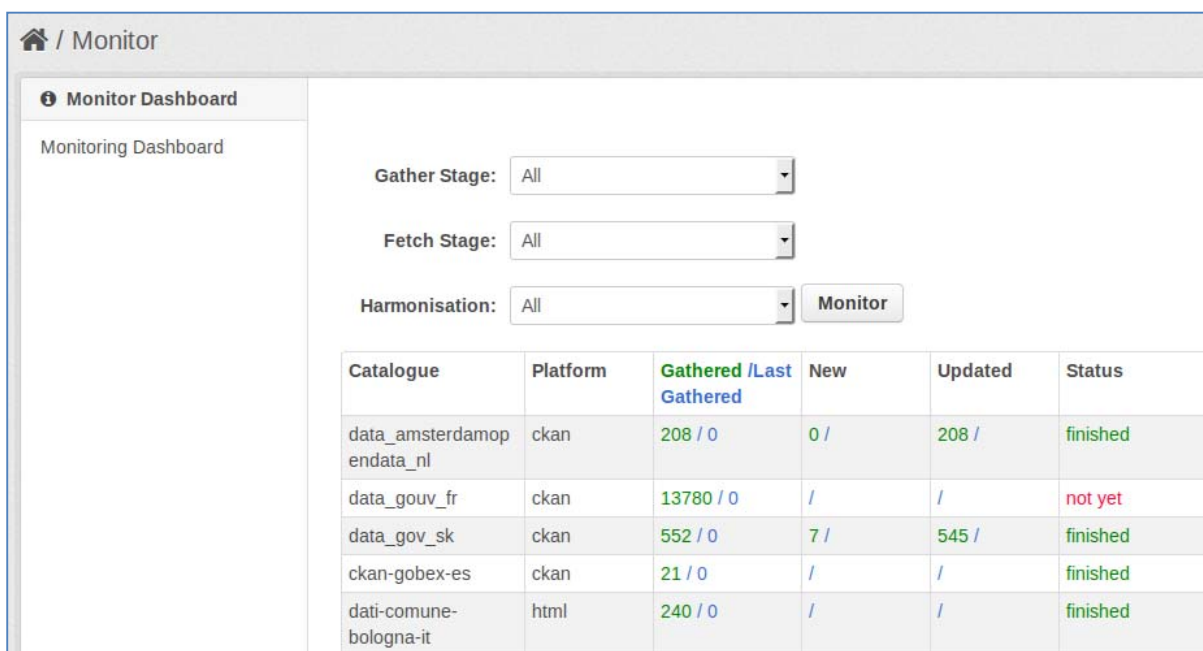
Although the previous panels give a detailed and thorough view of the executed harvesting jobs and their history, they apply to specific catalogues. This, however, is not very useful when we need to

monitor the harvesting process as a whole, e.g. to obtain an overview about which of the scheduled harvesting processes succeeded or not and possible errors that may have occurred in the harvesting process, based on indications arising from the presented results.

For this purpose, we have designed and implemented an additional dashboard, which is shown in Figure 22. This includes information for both the gather stage of the harvesting process, under the Gather/Last Gathered column, and the fetch stage, under New and Updated columns, and finally the current status of the running job. All this information is provided for each catalogue in the platform. In both cases, the presented results are referring to the current executed process and the last one. Empirically, this is adequate to lead us to useful conclusions for possible errors in the process. For instance, if after the completion of a harvesting process, the number of datasets collecting from the catalogue is lower than that found from the same catalogue during the last harvesting attempt, it is an indication that an error may have occurred, preventing the correct and complete harvesting of that specific catalogue.

This dashboard also contains useful filters for grouping the results. The existing filters that can be applied are: *Gather Stage*, *Fetch Stage* and *Harmonisation*, which select the type of process for which the information is to be displayed. After selecting a value for each of the above filters, we need to click the *Monitor* button in order to apply the filtering on the presented results.

This panel is available only to the Administrator in order to regularly check for the status of the running processes and any indications for possible errors. When such an error is deduced, the more detailed panels described above can be used to find more information on that catalogue and thus to help find the reason for the error that might have happened.



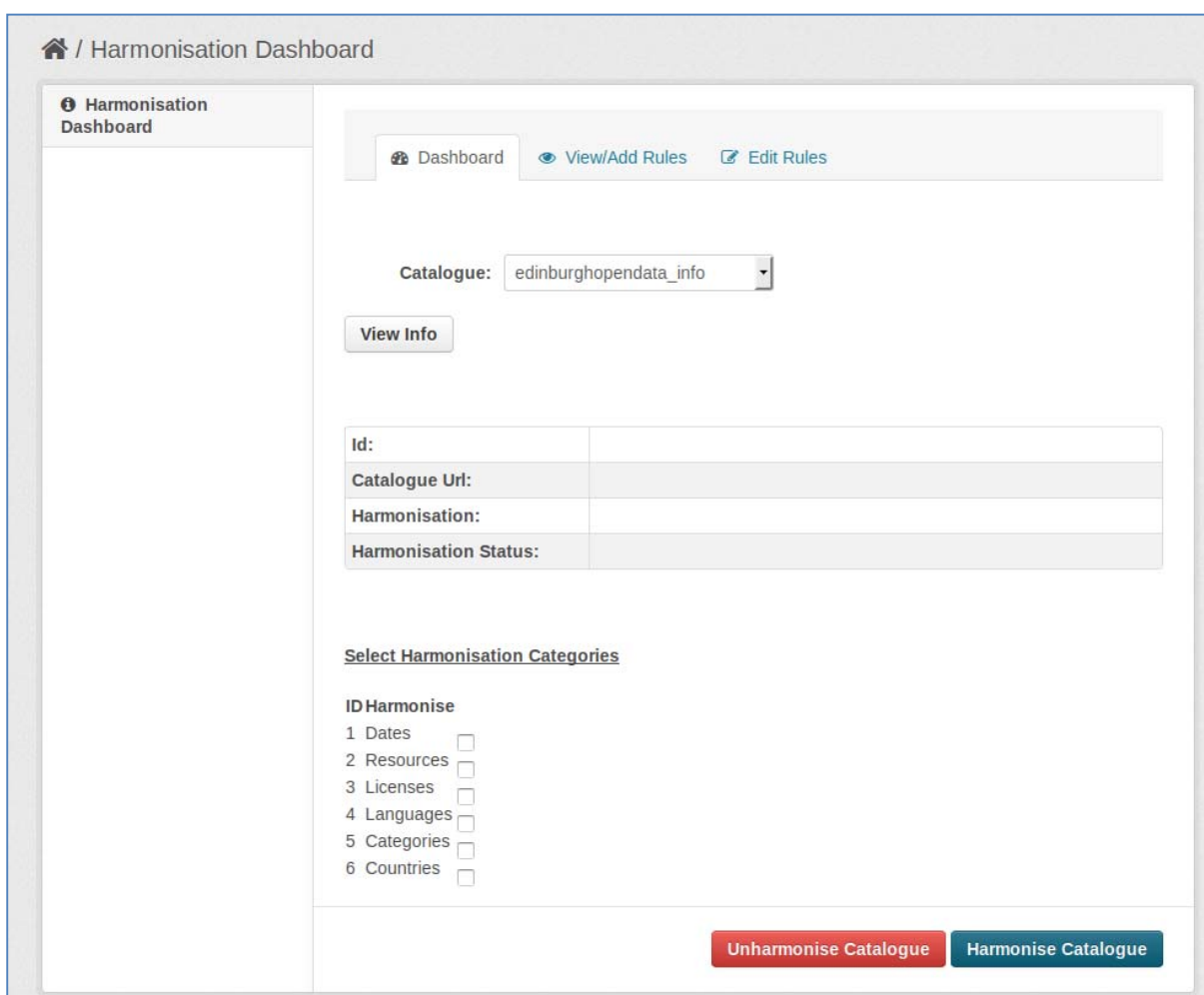
Catalogue	Platform	Gathered / Last Gathered	New	Updated	Status
data_amsterdamopendata_nl	ckan	208 / 0	0 /	208 /	finished
data_gouv_fr	ckan	13780 / 0	/	/	not yet
data_gov_sk	ckan	552 / 0	7 /	545 /	finished
ckan-gobex-es	ckan	21 / 0	/	/	finished
dati-comune-bologna-it	html	240 / 0	/	/	finished

Figure 22: View of the general overview panel.

6.3 Harmonization panel

We implemented a separate panel to monitor the harmonization processes. This panel enables us to manage rule definitions, to manually execute customized harmonization processes and generally to collect statistics about previous or current running processes. It consists of the following tabs:

- *Dashboard*, which gives a detailed overall view of the last executed harmonization process and the ability to manually schedule a harmonization job;
- *View/Add Rules*, that enables us to view/update existing mappings or propose new ones;
- *Edit Rules*, which is used to accept or reject mappings proposed in previous step.



Home / Harmonisation Dashboard

Harmonisation Dashboard

Dashboard View/Add Rules Edit Rules

Catalogue: edinburghopendata_info

View Info

Id:	
Catalogue Url:	
Harmonisation:	
Harmonisation Status:	

Select Harmonisation Categories

ID Harmonise

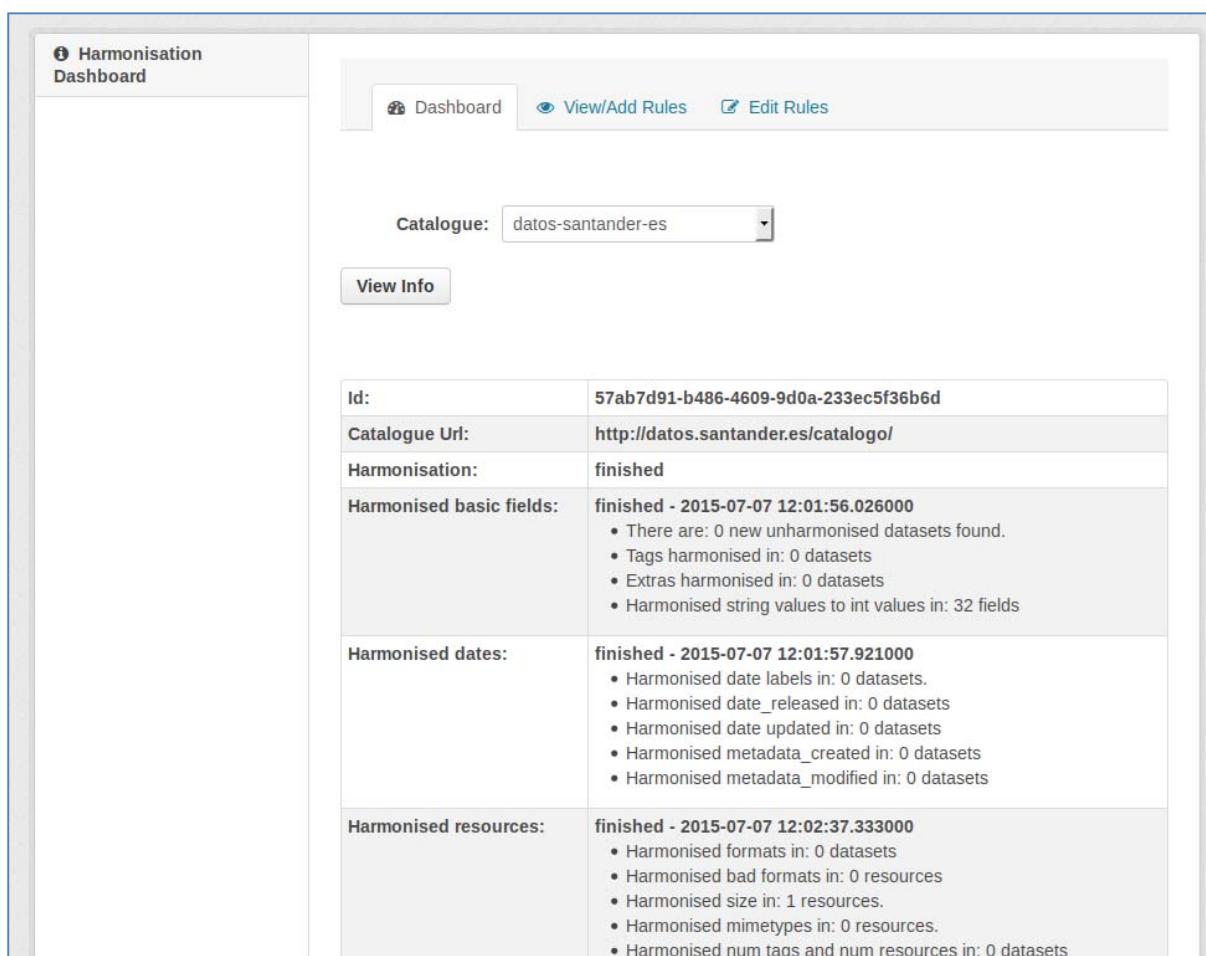
- 1 Dates ☐
- 2 Resources ☐
- 3 Licenses ☐
- 4 Languages ☐
- 5 Categories ☐
- 6 Countries ☐

Unharmonise Catalogue Harmonise Catalogue

Figure 23: Dashboard view of the harmonization panel.

The first panel, *Dashboard*, consists of two parts (see Figure 23). The top part contains a drop-down menu with all harvested catalogues in the ODM platform. We can select one of these to get information related to the process of harmonization, by clicking the *View info* button. The information we get back includes the id of the process that was assigned upon job creation and the

URL of the catalogue. Also we get the status of the process under the Harmonisation field which could be finished, started and not yet executed. Below these, we have detailed information about each of the specific harmonization cases that were executed. We start with a general overview under Harmonised basic fields, which reports the new datasets that have not yet been harmonized and certain actions applied to collected metadata. Moreover, detailed information about harmonized attributes is provided, such as the timestamp of the last such process that was executed on that attribute, the number of distinct values mapped, etc. Such an example for the Santander open data catalogue (<http://datos.santander.es/catalogo>) is presented in Figure 24.



The screenshot shows the 'Harmonisation Dashboard' for the 'datos-santander-es' catalogue. It includes navigation links for 'Dashboard', 'View/Add Rules', and 'Edit Rules'. A 'View Info' button is present below the catalogue selection. The main content area displays a table with harmonization details.

Id:	57ab7d91-b486-4609-9d0a-233ec5f36b6d
Catalogue Url:	http://datos.santander.es/catalogo/
Harmonisation:	finished
Harmonised basic fields:	finished - 2015-07-07 12:01:56.026000 <ul style="list-style-type: none"> • There are: 0 new unharmonised datasets found. • Tags harmonised in: 0 datasets • Extras harmonised in: 0 datasets • Harmonised string values to int values in: 32 fields
Harmonised dates:	finished - 2015-07-07 12:01:57.921000 <ul style="list-style-type: none"> • Harmonised date labels in: 0 datasets. • Harmonised date_released in: 0 datasets • Harmonised date updated in: 0 datasets • Harmonised metadata_created in: 0 datasets • Harmonised metadata_modified in: 0 datasets
Harmonised resources:	finished - 2015-07-07 12:02:37.333000 <ul style="list-style-type: none"> • Harmonised formats in: 0 datasets • Harmonised bad formats in: 0 resources • Harmonised size in: 1 resources. • Harmonised mimetypes in: 0 resources. • Harmonised num tags and num resources in: 0 datasets

Figure 24: Example view of the harmonization panel.

The second part of the presented form is used to manually execute specific harmonization rules. Under the heading *Select Harmonisation Categories* (see Figure 23) there are the available fields that we can choose to harmonize. We can use the checkboxes next to each attribute to choose the ones we wish to harmonize. After that, when we click the Harmonise Catalogue button, a customized harmonization job is created according to the user's input. This job will remain in pending status until it is chosen by the service for execution. This process is applied only to metadata that have not been harmonized in previous stages because, for instance, available mappings did not apply to this

catalogues' values. In cases that we need to re-harmonize the metadata for the whole catalogue, e.g. if an existing mapping has been updated, we can do that using the *Unharmonise Catalogue* button. This resets all metadata to their initial, raw form, i.e. as they have been harvested from the catalogue.

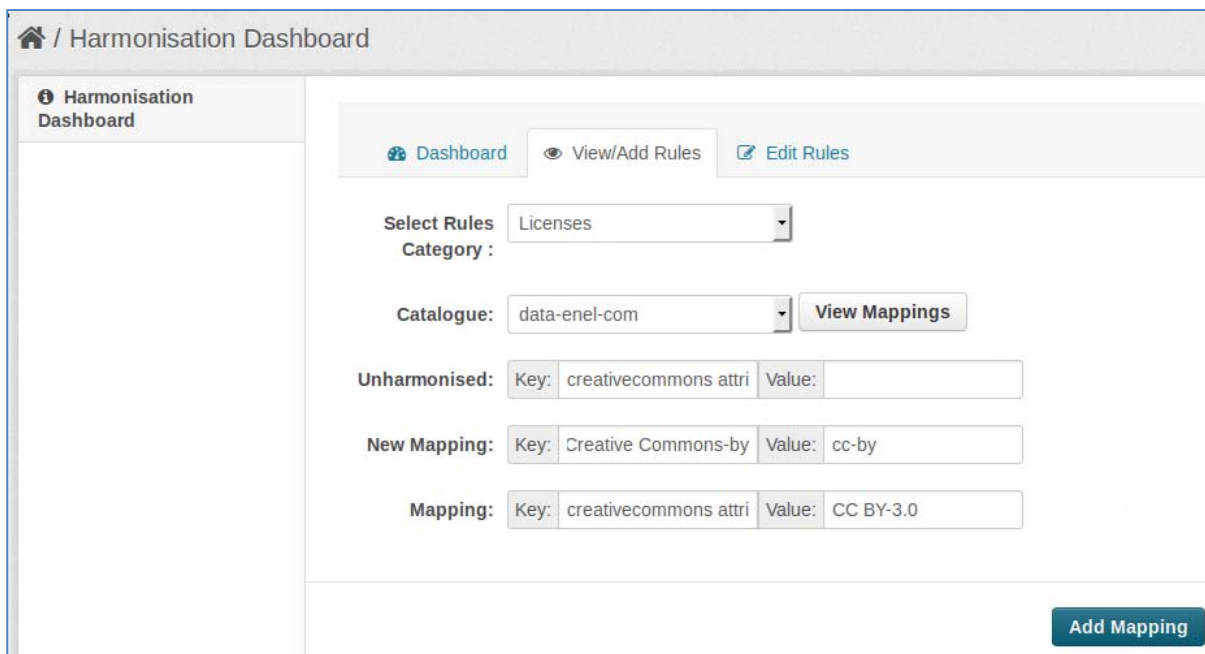


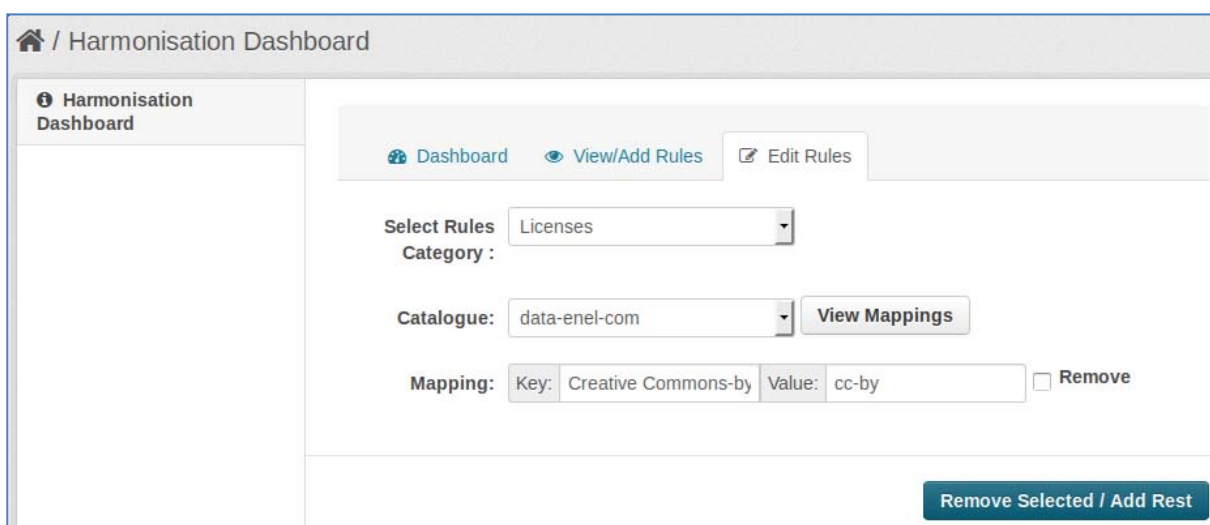
Figure 25: Example of viewing mapping rules.

The second tab of the panel, *View/Add Rules*, is used to handle the mappings. This form has two basic fields. The *Select Rules Category* drop-down list specifies the rule that we are interested to check. The possible values are: *Categories Values*, *Categories Labels*, *Dates*, *File Formats* and *Licenses*. These are the fields that are applied in a semi-automatic way. These rules use a list of pre-defined mappings to decide how to proceed. The *Catalogue* drop-down list selects a catalogue from the list of harvested ones. Additionally, this field has the value *All Catalogues*, which gives results for all the catalogues in one shot. After selecting combined values from the above lists, we can click the *View Mappings* button. We get back one or a combination of the following possible fields:

- *Unharmonized*, that contains the raw metadata attributes for the selected category that none of the pre-defined mappings is applied;
- *New Mapping*, which both key-value pairs are empty and is used to propose new ones;
- *Mapping*, which contains full list, in dictionary format, of all the available mappings applied to the selected catalogue.

Figure 25 presents an example of a catalogue which has one record for each of the previously described cases.

Finally, the last tab is only accessible to the administrator. In this, we can view all the proposed mappings, new and updated, in previous step and accordingly accept or reject them. Therefore, since it affects an important functionality of the harmonization process, we need to be sure that imported values are verified. As seen in Figure 26, there are two fields, as in the previous tab, which make a combined selection of the category rule to use and the selected catalogue. Then, after clicking the *View Mappings* button, we get all proposed mappings for the above selected combination. For the example shown, we see that each mapping pair is followed by a *Remove* checkbox. This information is used when we click the *Remove Selected / Add Rest* button. It simultaneously makes two different operations: it adds all proposed mappings and, in the same time, it deletes all those that the checkbox is checked.



The screenshot displays the 'Harmonisation Dashboard' interface. On the left is a sidebar with a 'Harmonisation Dashboard' link. The main content area has three tabs: 'Dashboard', 'View/Add Rules' (which is active), and 'Edit Rules'. Below the tabs, there are two dropdown menus: 'Select Rules Category' (set to 'Licenses') and 'Catalogue' (set to 'data-enel-com'). A 'View Mappings' button is positioned to the right of the 'Catalogue' dropdown. Below these, a 'Mapping' section shows 'Key: Creative Commons-by' and 'Value: cc-by', followed by an unchecked 'Remove' checkbox. At the bottom right of the dashboard is a large button labeled 'Remove Selected / Add Rest'.

Figure 26: Example of editing rules.

7 CONCLUSIONS AND NEXT STEPS

In this report, we have first presented the overall architecture and the processing workflow for the ODM system that was designed in the first year of the project, and then we have described new functionalities or enhancements implemented for the main components of the platform. The progress of the work is summarised below¹⁸:

- *Catalogue registry.* We have added a registration form for the newly included Socrata harvester. We modified the HTML harvester's form to integrate the newly available method of collecting metadata in RDF schema and handling pagination systems wrapped in JavaScript code. Additionally, various other modifications made, mainly to this form, that enable the multiple URLs handling used as landing pages for harvesting or user friendly error messages during registration.
- *Metadata harvester.* For this component, we extended and updated the harvester for Socrata catalogues provided by the community. Also the HTML harvester enhanced with the ability to parse catalogue's metadata in RDF schema, increasing its accuracy, and handle catalogues that wrap pagination in JavaScript code, collecting this way metadata otherwise not feasible to do with current means. General statistics provided for the collected metadata and the catalogues presenting the most frequently used meta-attributes across catalogues, the catalogues' sizes in number of datasets, the applied usages for each of the implemented harvesters and their sub cases (i.e. JavaScript vs static pagination) and the overall size of the repository in our server.
- *Harmonisation engine.* For this component, the harmonization process has been extended to a fully functional component with multiple levels of mappings and different processing handlers for performing data cleaning and transformations. Additional components were also implemented to remove existing duplicate metadata.
- *Analysis engine.* For this component, we have extended and defined new metrics adapted to the existing collected and harmonized attributes. These metrics grouped in two categories: Quality and Quantity. Additionally the RESTful API was extended to provide access to the results of these new computations as well as the contents of the metadata repository in general.
- *Administration panel.* We implemented a graphical user interface to facilitate the ODM administrator to monitor and control various aspects of the system and the processing workflow. The harvesting panel for the HTML harvester is fully integrated to the provided one from the ckanext-harvester. A general panel for the harvesting process was also implemented that gives the administrator the overall overview of the whole process. And finally, a graphical interface was implemented to support and supervise the harmonization process for the existing catalogues in the platform.

¹⁸ 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>

Currently the system is up and running. It periodically harvests and harmonises collected metadata. We monitor this process on an ongoing basis and address any bugs and issues that arise. We have managed to harvest over 150 catalogues from 24 European countries.

8 APPENDIX

8.1 Examples

Table 2: A Socrata JSON document instance.

```
{
  "_id" : ObjectId("5590dde09daf091cb8995c59"),
  "maintainer" : "Carmen Lavado",
  "num_tags" : 6,
  "updated_dataset" : true,
  "isopen" : true,
  ...
  "resources" : [
    {
      "mimetype" : "text/html",
      "name" : "municipals-2003-participació.html",
      "metadata_modified" : "2015-05-22T14:33:28",
      "format" : "html",
      "url" : "https://gavaobert.gavaciutat.cat/resource/ifan-we9g",
      "metadata_created" : "2015-05-22T14:32:14",
      "description" : "Participació per mesa de les eleccions municipals de 2003"
    },
    ...
  ],
  "num_resources" : 2,
  "tags" : [
    "eleccions",
    "elecciones",
    "municipals",
    "municipales",
  ]
}
```



```
"2003",
  "Processos Electorals"
],
"catalogue_url" : "https://gavaobert.gavaciutat.cat",
...
"extras" : {
  "Dataset-Information:Data-d-incorporació-al-catàleg" : "",
  "category" : "Processos Electorals",
  "Dataset-Information:Data-darrera-actualització" : "",
  "Dataset-Information:Any" : "",
  "Dataset-Information:Update-Frequency" : "Anual",
  "Dataset-Information:Department-Owner" : "Ajuntament de Gavà"
},
"title" : "MUNICIPALS 2003 Participació"
}
```

Table 3: RDF example.

```
<rdf:RDF>
  <dc:Dataset rdf:about="http://data.nantes.fr/donnees/detail/opendata/annuaire-des-associations-et-des-activites-de-nantes/">
    <dc:identifiant>24440040400129_VDN_VDN_00132</dc:identifiant>
    <dc:title>Annuaire des associations et des activités de Nantes</dc:title>
    <dc:description>Les données sont constituées des associations dont le siège ou l'une au moins des activités est situé(e) sur le territoire de la ville de Nantes.
    ...
    Enfin, le site
      <a href="http://www.nantes.fr">http://www.nantes.fr</a> vous permet également d'accéder aux données de cet
      <a href="http://www.nantes.fr/infonantes/association">annuaire</a> via un outil de recherche.
    </dc:description>
```

```

<dc:dataset>http://data.nantes.fr/donnees/detail/opendata/annuaire-des-associations-et-des-activites-
de-nantes/</dc:dataset>

<dc:theme>Citoyenneté / Institution</dc:theme>

<themeInspire/>

<dc:keywords>associatif, ESS, annuaire, association</dc:keywords>

<dc:licence>Open Database License (ODbL)</dc:licence>

<dc:issued>1409608800</dc:issued>

<dc:modified>1436220000</dc:modified>

<lastModificationDescription>Mise a jour hebdomadaire</lastModificationDescription>

...

<dc:themeTaxonomy>Thésaurus InterDoc</dc:themeTaxonomy>

<dc:distribution>

  <dc:Distribution>

    ...

<dc:WebService>/api/publication/24440040400129_VDN_VDN_00132/ANNUAIRE_ASSOCIATIONS_NANTES_
STBL/content/?format=excel</dc:WebService>

    <dc:format>XLS</dc:format>

<dc:accessURL>http://data.nantes.fr/api/publication/24440040400129_VDN_VDN_00132/ANNUAIRE ASSO
CIATIONS_NANTES_STBL/content/?format=excel</dc:accessURL>

  </dc:Distribution>

</dc:distribution>

</dc:Dataset>

</rdf:RDF>

```

Table 4: Example of the partialOrder.csv file

```

http://publicdata.eu/|http://data.gov.uk/,http://www.nosdonnees.fr/,https://offenedaten.de/,https
://www.govdata.de/ckan/,http://www.dati.gov.it/catalog/,http://www.daten.rlp.de/,https://www.d
ata.gv.at/katalog/,http://data.gov.sk/,http://opengov.es/,http://cz.ckan.net/en/,http://data.kk.dk/,h
ttp://it.ckan.net/,http://data.gov.ro/,http://ie.ckan.net/,http://portal.openbelgium.be/,http://rs.cka
n.net/,http://suche.transparenz.hamburg.de/,http://www.opendata.provincia.roma.it/,http://opend
ata.comune.bari.it/,http://www.opendatahub.it/

```

<http://www.daten.rlp.de/> | <https://www.govdata.de/ckan/>
<https://offenedaten.de/> | <https://www.govdata.de/ckan/>
<http://www.opendata-hro.de/> | <https://www.govdata.de/ckan/>
<http://dati.venezia.it> | <http://www.dati.gov.it/catalog/>
<http://aperto.comune.torino.it> | <http://www.dati.gov.it/catalog/>
<http://dati.trentino.it/> | <http://www.dati.gov.it/catalog/>
<http://dati.veneto.it/> | <http://www.dati.gov.it/catalog/>
<http://daten.berlin.de> | <https://www.govdata.de/ckan/>
<http://ckan.data.linz.gv.at/> | <https://www.data.gv.at/katalog/>
<http://ckan.data.graz.gv.at/> | <https://www.data.gv.at/katalog/>
<http://datahub.io/> | <https://www.govdata.de/ckan/>
<http://datahub.io/> | <http://portal.openbelgium.be/>
<http://opendata.awt.be/> | <http://portal.openbelgium.be/>
<http://data.opendataforum.info/> | <http://portal.openbelgium.be/>
<http://dati.toscana.it/> | <http://www.dati.gov.it/catalog/>
<http://www.opendatahub.it/> | <http://www.dati.gov.it/catalog/>
<http://opendata.comune.bari.it/> | <http://www.dati.gov.it/catalog/>
<http://www.opendata.provincia.roma.it/> | <http://www.dati.gov.it/catalog/>
<http://data.no.e.gv.at/> | <https://www.data.gv.at/katalog/>
<http://opendata.provincia.lucca.it/SpodCkanApi/> | <http://dati.toscana.it/>
<http://opendata.cmt.es> | <http://opendata.cnmc.es/>
<http://data.nantes.fr> | <http://data.loire-atlantique.fr>
<http://data.wu.ac.at/> | <http://data.opendataportal.at/>
<http://opendata.comune.bari.it/> | <http://www.opendatahub.it/>
<http://dati.toscana.it/> | <http://www.opendatahub.it/>
<http://data.digitaliser.dk> | <https://data.digitaliser.dk>

8.2 Code Snippets

Table 5: Select harvester type for registration¹⁹

```
<label class="radio">
    <input type="radio" value={{ app_globals.site_url ~"/harvest/new"}}
name="button_harvester_type" id="navRadio01" checked>CKAN / SOCRATA</input>
</label>
<label class="radio">
    <input type="radio" value={{ app_globals.site_url ~"/htmlharvest1"}}
name="button_harvester_type" id="navRadio02">HTML
</label>
<label class="radio">
<br>

<input type="button" value="Next" class="btn-primary btn"

    onclick="ob=this.form.button_harvester_type;for(i=0;i<ob.length;i++){
        if(ob[i].checked){window.open(ob[i].value,'_top');};}">
</input>
```

Table 6: Select configuration rules to apply in harvesting process²⁰

```
if rules!="":
    if 'rdf' in rules.keys():
        if rules['rdf']!='' and rules['rdf']!=None:
            content=str(RdfToJson.harvest_rdf(dataset_url,rules))
        else:
            content=str(harvest_url.harvest_url(dataset_url,rules))
    #print(rules)
    ## content must be a string
```

¹⁹ https://github.com/opendatamonitor/ckanext-htmlharvest/blob/master/ckanext/htmlharvest/templates/snippets/add_htmlharvest_button.html

²⁰ <https://github.com/opendatamonitor/ckanext-htmlharvest/blob/master/ckanext/htmlharvest/harvesters/htmlharvester.py>

```

else:
    content=str(harvest_url.harvest_url(dataset_url,rules))
harvest_object.content = content
try:
    harvest_object.save()
except:
    pass

```

Table 7: Select available options to parse successive pages with HTML harvester²¹

```

if 'btn_identifier' in document.keys():
    if document['btn_identifier']!=None and
document['btn_identifier']!='':
        cat_url=document['cat_url']
        dataset_identifier=document['identifier']
        btn_identifier=document['btn_identifier']
        action_type=document['action_type']
        try:
            sleep_time=document['sleep_time']
        except:
            sleep_time=3
        package_ids=javascript_case.ParseJavascriptPages
(cat_url,dataset_identifier,btn_identifier,action_type,sleep_time)
        print(package_ids)
    else:
        package_ids=harvester_final.read_data(id1,backupi)
else:
    package_ids=harvester_final.read_data(id1,backupi)

```

²¹ <https://github.com/opendatamonitor/ckanext-htmlharvest/blob/master/ckanext/htmlharvest/harvesters/htmlharvester.py>

Table 8: Parsing multiple URLs provided as landing pages in a catalogue²²

```
cat_urls=[]

counter=0

text_file_mails = open('/var/local/ckan/default/pyenv/src/ckanext-
htmlharvest/ckanext/htmlharvest/harvesters/emails.txt', "a")
text_file_maintainer_mails=open('/var/local/ckan/default/pyenv/src/ckanext-
htmlharvest/ckanext/htmlharvest/harvesters/maintainer_emails.txt', "a")

print(mainurl)

log.info('Started')

print(url)

if ',' in url:

    url=url.replace('\n','').replace('\r','').rstrip(',')

    cat_urls=url.split(',')

else: cat_urls.append(url)

print(cat_urls)


##custom sensibility cases handling
document=collection.find_one({"cat_url":{"$regex": mainurl}})

if document!=None:

    if 'sensibility' in document.keys():

        sensibility=document['sensibility']

    else: sensibility=0.9

else: sensibility=0.9

print("sensibility set to: "+str(sensibility))

count=0

while count<len(cat_urls):

    break_count=0

    while endpoint in soup1:

        ...
```

²² <https://github.com/opendatamonitor/ckanext-htmlharvest/blob/master/ckanext/htmlharvest/harvesters/HarvestProcedure.py>

Table 9: Form validation checks for empty and incorrect values²³

```
...
errors= ''

language=str(data['language'])
step=str(data['step'])
cat_url=str(data['cat_url'].encode('utf-8'))
url=str(data['url'])
if 'http' not in cat_url and len(url)<8:
    errors= "Invalid Catalogue URL, "
if 'http' not in url or len(url)<8:
    errors =errors+"Invalid Dataset's URL, "
after_url=str(data['afterurl'].encode('utf-8'))
catalogue_date_created=str(data['catalogue_date_created'])
catalogues_description=str(data['catalogues_description'])
catalogue_date_updated=str(data['catalogue_date_updated'])
identifier=str(data['identifier'])
catalogue_country=str(data['catalogue_country'])
catalogue_title=str(data['catalogue_title'])
if catalogue_title=="":
    errors =errors+" Invalid Title"
harvest_frequency=str(data['harvest_frequency'])
btn_identifier=str(data['btn_identifier'])
action_type=str(data['action_type'])
if errors!='':
    vars = {'data': data, 'errors': str(errors.rstrip(', ')+'.')}
    return render('htmlharvest1.html', extra_vars=vars)
```

²³ <https://github.com/opendatamonitor/ckanext-htmlharvest/blob/master/ckanext/htmlharvest/controllers/package.py>

```
try:
    autofind=AutoMetadataFinder.AutoFindingElements(url)
except:
    errors =errors+"Invalid Dataset's URL, "
    vars = {'data': data, 'errors': str(errors.rstrip(', ')+'.')}
    return render('htmlharvest1.html', extra_vars=vars)
...
```

Table 10: Fields, language and country, defined as drop-down button in HTML harvester's registration form²⁴

```
...
{% call form.select('language', id='language', label=_('Language'),
options=h.languages_list(), selected=data.language, error=errors.language)
%}

    <span class="info-block">
        {{ _("This should include the Catalogue's Language") }}
    </span>
{% endcall %}

{% call form.select('catalogue_country', id='catalogue_country',
label=_('Country'), options=h.countries_list(),
selected=data.catalogue_country, error=errors.catalogue_country) %}

    <span class="info-block">
        {{ _("This should include the Catalogue's Country") }}
    </span>
{% endcall %}
...
```

²⁴ https://github.com/opendatamonitor/ckanext-htmlharvest/blob/master/ckanext/htmlharvest/templates/snippets/htmlharvest_form_first.html

Table 11: Declare lists for languages and countries as helper functions²⁵

```
...
def countries_list():
    return [{'text': p.toolkit._(f.title()), 'value': f}
            for f in COUNTRIES]

def languages_list():
    return [{'text': p.toolkit._(f.title()), 'value': f}
            for f in LANGUAGES]
...
```

Table 12: Initialize lists with default language and country values²⁶

```
COUNTRIES = ['United
Kingdom', 'Albania', 'Andora', 'Armenia', 'Austria', 'Azerbaijan', 'Belaru
s', 'Belgium', 'Bosnia and
Herzegovina', 'Bulgaria', 'Croatia', 'Cyprus', 'Czech
Republic', 'Denmark', 'Estonia', 'Finland', 'France', 'Georgia', 'Germany'
, 'Greece', 'Hungary', 'Iceland', 'Ireland', 'Italy', 'Kazakhstan', 'Latvia
', 'Liechtenstein', 'Lithuania', 'Luxembourg', 'Macedonia'
, 'Malta', 'Moldova', 'Monaco', 'Montenegro', 'Netherlands', 'Norway', 'Pol
and', 'Portugal', 'Romania', 'Russia', 'San
Marino', 'Serbia', 'Slovakia', 'Slovenia', 'Spain', 'Sweden', 'Switzerland
', 'Turkey', 'Ukraine', 'Vatican City']

LANGUAGES =
['English', 'Bulgarian', 'Croatian', 'Danish', 'Dutch', 'Estonian', 'Finni
sh', 'French', 'German', 'Greek', 'Hungarian', 'Italian', 'Latvian', 'Lithu
anian', 'Maltese', 'Polish', 'Portuguese', 'Romanian', 'Slovak', 'Spanish'
, 'Swedish']
```

²⁵ <https://github.com/opendatamonitor/ckanext-harvestodm/blob/master/ckanext/harvestodm/helpers.py>

²⁶ https://github.com/opendatamonitor/ckanext-harvestodm/blob/master/ckanext/harvestodm/model/__init__.py

Table 13: Quantity metrics

API name	Formula	Description
_catdatasetstotfreq	Count number of metadata objects per catalogue	Total number of datasets
_catdistribstotfreq	Count number of resources per catalogue	Total number of distributions
_catfreq	Count number of different catalogues	Total number of catalogues
_catdatasizetotal	Sum resources' size per catalogue (KB)	Total distribution size
_eddistribsize	Total Sum of resources' size in Europe (KB)	
_categories	Count categories found in	Categories
_catpublishersfreq	Count unique publishers per catalogue	Unique publishers

Table 14: Quality metrics

API name	Formula	Description
_catopenlicfreq	total count of open licences / total count of distributions with a licence	Number of Open licences per catalogue
_edopenlicfreq		% of Open licences aggregated across Europe

_catnonpropformatfreq	total count of distributions with a non-proprietary format / total count of distributions with a format	Number of Non-proprietary formats per catalogue
_ednonpropformatfreq		% of Non-proprietary formats aggregated across Europe
_catmachinereadformatfreq	sum(1 if a dataset has at least one MR distribution, 0 otherwise)/ total count of datasets	Number of Machine-readable datasets per catalogue
_edmachinereadformatfreq		% of Machine-readable datasets aggregated across Europe
_catcoremetadafreq	Defined set of fields as requirement: - License: 0.25 - Author/Maintainer (maintainer field exists only in CKAN): 0.25 - Organisation: 0.25 - Date released or Date updated: 0.25	Number of Complete core metadata per catalogue
_edcoremetadafreq		% of Complete core metadata aggregated across Europe
_cataccessabilityfreq	Defined set of fields as requirement: - description (notes in database): 1 if contains some text, 0 null/empty - not a broken link: 1 if not broken - author email/maintainer_email (only in CKAN): 1 if present sum(descrtiption && not_broken && author_email)/total count of datasets	Accessability in %
_catsitepagerank	-	Metric using Google and Alexa traffic ranking to define catalogue's discoverability