H2020-INSO-2014
INSO-1-2014 ICT-Enabled open government
YDS [645886] "Your Data Stories"

D3.9 Open Data Repository v1.0

Project Reference No	645886 — YDS — H2020-INSO-2014-2015/H2020-INSO-2014
Deliverable	D3.9 Open Data Repository v1.0
Workpackage	WP3: Data Layer
Nature	Demonstrator
Dissemination Level	Public
Date	31/12/2015
Status	Final v1.0
Editor(s)	Aad Versteden, Uroš Milošević, Erika Pauwels (TF)
Contributor(s)	N/A
Reviewer(s)	Michalis Vafopoulos (NCSR-D)
Document description	This deliverable will constitute the YDS open data repository prototype. This repository will offer harvested and possibly interlinked data to other YDS components efficiently and effectively.

Document Revision History

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
V0.1	01/12/2015	Table of Contents	TF
V0.6	22/12/2015	First draft	TF
V0.8	24/12/2015	Peer-review ready	TF
V0.9	28/12/2015	Peer-reviewed	NCSR-D
V1.0	28/12/2015	Final version	TF
V1.0	31/12/2015	Final version for submission to EC	ATC

Executive Summary

The Open Data Repository will host the contents for widely varying views on a large interlinked dataset. We support this by offering a platform with both standardized and custom data-offering services. The current variant of the open data repository supports SPARQL [1] and JSON API [2] based calls. New outputs can be supported by implementing new microservices.

We should expect the YDS project to offer very different views on the same ground truth. The information which is available through the unified data model will be shown to users in new and exciting ways. The goal is to offer insights based on innovative visualizations. This provides challenges in terms of performance and display of the necessary content for the Open Data Repository.

The architecture of the Open Data Repository is based on mu.semte.ch [3, 4]. This project provides a microservices based architecture for offering views on linked data. Central in this architecture is the triplestore, which contains all the information on the platform. Many microservices can offer support for updating and viewing the data in the triplestore. These microservices can be developed by the repository developers, or independently by component developers.

JSON [7] views on the YDS data model are offered by use of the mu-cl-resources [5] project. This microservice acts as the glue between the contents of the JSON responses, and the linked data model. SPARQL access is offered by Virtuoso [6]. A single endpoint for all microservices is offered by the mu-dispatcher [10], which is configured accordingly. The JSON APIs are offered by an API which adheres to the JSON API specification. This will limit internal discussions and help developers find tooling for their development environment. Deployment is made easy by using Docker [8]. The whole system is orchestrated using Docker Compose [9].

The future vision of the Open Data Repository consists of making the repository scalable by allowing microservices to offer views from different stores. A document store with the right structure could help in rendering contents. Furthermore, we could be able to make the triplestores scale by replicating the Virtuoso database. Lastly, the repository will evolve by offering tailored output for particular YDS components.

The Open Data Repository currently has support for rendering standard views on the content in the triplestore. A future view has been presented to evolve the Open Data Repository into a flexible and reusable component.

Table of Contents

List of Figures

List of Terms and Abbreviations

Abbreviation	Definition
API	Application Programming Interface
JSON	JavaScript Object Notation
REST	Representational State Transfer
SPARQL	SPARQL Protocol and RDF Query Language

1 Introduction

1.1 Purpose and Scope

This task will develop the main repository where all data will be stored, after they have been collected, cleaned, validated, aligned, and cross-linked in T3.3. This repository will implement the data model that will be defined in T3.2, and will make the data available to all components of the YDS platform, through suitable semantic services. This Open Data Repository will be the central component for managing the data that will be provided by YDS, not only to the rest of the platform layers, but also to all applications that operate on top of the YDS platform, constituting the repository an essential element of the YDS platform offerings. The repository will offer search services, including SPARQL queries, delivering results in multiple formats to satisfy the needs of the YDS pilot and third-party applications.

In a first release a general form of the platform is delivered which may be extended with features in later releases. Later releases will focus on scaling up the architecture. We expect to cope with changes in the data model and in the semantic services offered by the Open Data Repository. We also expect to cope with a feed of linked data received from T3.3.

1.2 Approach for Work Package and Relation to other Work Packages and Deliverables

The primary focus of the third Work Package is data collection, management, storage, and provision to other components of YDS through effective programming services and interfaces. The relevant technologies can be indexed in four main layers:

1. Harvesters, which acquire data from open data sources and the social Web.
2. Validators, assessing the quality of harvested data, cleaning data, and validating them according to the data model requirements.
3. Ontology population and alignment, where diverse data sources are aligned, cross-linked, and transformed to use a common vocabulary, ultimately becoming part of a unifying data model.
4. Data storage and querying, where data are accumulated in a data store, which provides retrieval and management through suitable semantic interfaces (such as SPARQL queries).

In particular, the third layer provides a common language across all the four layers by enabling the orchestration of diverse specifications from heterogeneous systems and methodologies. Actually, it serves as a detailed and interactive reference of the modeling choices that have been made for all relevant data sources.

1.3 Methodology and Structure of the Deliverable

The topics in the deliverable can be considered to be sectioned in chronological order. The first portions of the project should be constructed before the latter ones can be used. The topics by which the deliverable was split, indicate different uses of the platform. Given some particular interest, it should be clear which topics will be of interest to the reader.

The first section aims to build a shared understanding of the technical problems in the problem domain. We don't make claim to understand all the problems in this domain, but rather seek to find a general direction of issues. With these constraints in mind, we focus on the general architecture of a platform that could solve these needs. We follow up by discussing in what way the platform needs to be constructed in order to support the model of the YDS platform. The APIs, which are the result of this, are then discussed. Lastly, we show a vision on how the repository could be extended in order to cope with the future needs of the YDS project.

2 Problem scope

The YDS platform will cater for widely varying views on showing publicly available content. The platform also needs to be comparatively easy to install, whilst still offering a way to scale up in the future. The Open Data Repository will receive contents from T3.3, but will need to cope with the structure and offer views on the data. In this section we try to derive the problems which the Open Data Repository will need to cope with. We don't make claim to cover the whole problem space, but aim to offer support for at least these challenges.

The users of the data offered by the YDS platform might be linked data experts, but we aim to support other users too. Some of them may have limited understanding of the linked data model. The platform should also cater for web developers who mainly want to gain and share insights. The linked data model allows us to easily integrate the data, but it should not be a requirement for those using the platform. We also don't want to scare linked data experts away. Offering an endpoint which talks in common formats should also be offered. Given the web-based nature of the platform, we mainly foresee JSON to play an important role here.

Although the initial release of the repository will not need to scale to very large amounts of data, the architecture should not limit such use. It should be easy to scale up the platform, or it should be possibly to evolve the platform so scaling is possible. Ideally, this should not make the platform too complex to install either. Especially for the non-scaled repository, it should not require multiple man-days to install.

Different YDS components may require different views on the same data. We should try to minimize the footprint of the APIs we offer. Maintaining a large code-base can be costly. However, with the widely varying components being offered, we should foresee that at least some of these components will require some custom views on the data. It is tempting to only foresee altering views on the data, but we should also support server-side computations and caching for some components. A way to install and run plugins on the repository would help in supporting the data output for new components.

3 General architecture

The architecture of the Open Data Repository is based on user-facing microservices which store their data in a shared triplestore. The architecture has grown together with the mu.semte.ch project [3, 4]. Each of the microservices are said to be user-facing, in that they offer a service aimed directly at the end-users. The effort of installing all these microservices is minimized by bundling them in a Docker Compose [9] configuration. This allows the whole system to be downloaded and installed through a single command.

3.1 Layers

A request for the Open Data Repository passes through 4 layers (Figure 1). The first layer, *mu-identifier* [11], identifies the device that is accessing the service. The second layer, *mu-dispatcher* [10], ensures the request is dispatched to the right microservice. The third layer is one of the microservices. It communicates with the triplestore to return a response to the user. Except for a possible cookie set to identify the current user, the response is left untouched. This makes reasoning on the services easy. Each of these phases will be described in the following sections.

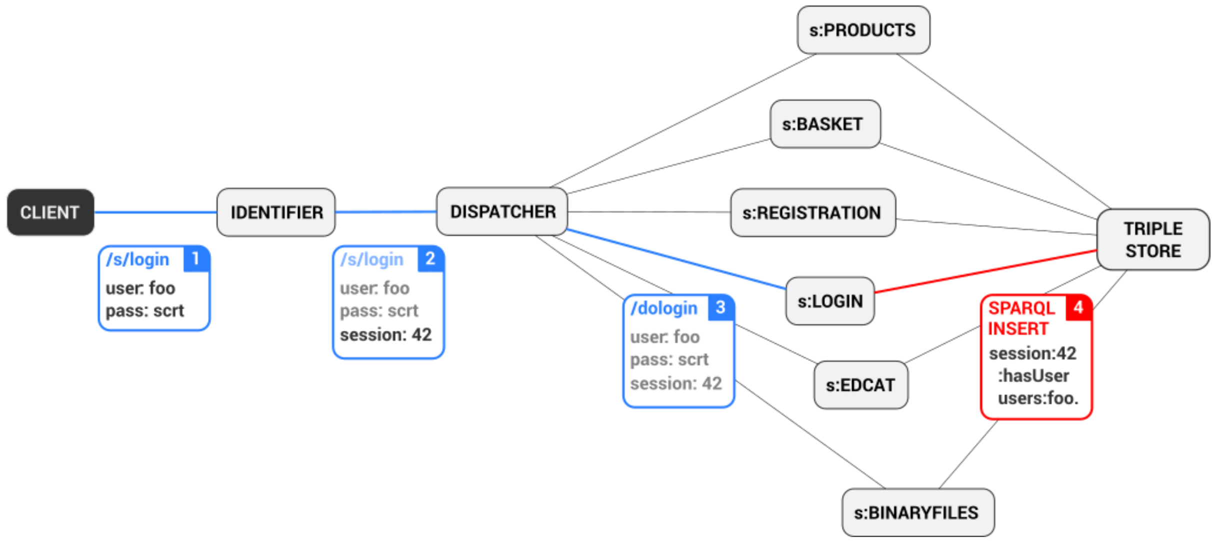

Figure 1: General Architecture

3.1.1 Identifier

The identifier layer is a trivial microservice which sets a cookie in the user's browser. This cookie is used to identify the current device which is accessing the service. When a user logs in through a login service, this service can store in the database that a user has logged in on the system with that device. Other services can then use that information if they need to support the currently logged in user. Logging out of the system would consist of removing the triple which connects the device identifier to the session. The identifier identifies a device and augments the request with a custom HTTP header containing an identification URI for the current device.

3.1.2 Dispatcher

The dispatcher contains a configuration to select the right microservice to handle a specific request. A specific microservice could be hosted on a variety of URLs. Many microservices will not care about the path on which they are hosted. A service which returns the total amount of currently logged in users would not need to adapt whether it is hosted on /stats/logged-in-users or /info/users/logged-in-count. The dispatcher decides which service is available on which path and forwards an incoming request to the correct microservice. Note, however, that some microservices may care about the location on which they are hosted.

3.1.3 Microservices

The microservices contain the specific implementations for the YDS components. One specific microservice is *mu-cl-resources*, which maps the contents of the triplestore to a JSON API [2] compatible REST API. The microservice receives the request and uses a connection to the triplestore to show and/or alter the contents of the triplestore. The response generated by the microservice is sent back to the end-user. The microservices can cooperate without much synchronization, as they all work on the same standardized data-model. The semantic model in the triplestore is therefore key for the correct operation of the architecture.

This whole setup of microservices is maintained through the Docker ecosystem. A Docker container can be considered to be a lightweight virtual machine, ideally hosting a single application. In our case, each of the microservices will be run in a single Docker container. The whole system is packaged in a *docker-compose* file. This file orchestrates a set of Docker containers and how they should work together. You can start the orchestration by running a single command, which will download and start all necessary Docker containers.

The microservices in the mu.semte.ch project can all be scaled up, aside from the database. Much of the architecture is therefore rather attractive to use at scale. Replication and load balancing would work for most of the components. The database is a different story. Most of the microservices talk to the database and triplestores aren't known for their superior performance. Approaches for scaling up the database are considered future work and are discussed in Section 6.

Our approach of using microservices makes it easy to implement a new microservice for a specific need. If there is a component in the YDS platform which needs a different approach than the standardized JSON API based responses, it can be implemented in a microservice. In contrast to other approaches, much of the complexities of working with microservices are hidden due to the use of the shared model in the triplestore.

4 Implementation of the model and extensions

The most generic JSON responses are offered by mu-cl-resources. This component offers a JSON API compatible API for the content in the triplestore. It also allows for the updating of the contents in the triplestore if the SPARQL endpoint allows for write operations.

The mu-cl-resources microservice needs to be configured so that it knows how to map the contents of the triplestore to a JSON API compatible response. Configuration occurs through a paren-based syntax. In order to support a catalog resource of the DCAT standard, one would use the following snippet:

```
(define-resource catalog ()
  :class (s-prefix "dcat:Catalog")
  :properties `((:title :string ,(s-prefix "dct:title"))
              (:description :string ,(s-prefix "dct:description"))
              (:issued :string ,(s-prefix "dct:issued"))
              (:modified :string ,(s-prefix "dct:modified"))
              (:language :string ,(s-prefix "dct:language"))
              (:license :string ,(s-prefix "dct:license"))
              (:rights :string ,(s-prefix "dct:rights"))
              (:spatial :string ,(s-prefix "dct:spatial"))
              (:homepage :string ,(s-prefix "foaf:homepage")))
  :has-one `((publisher :via ,(s-prefix "dct:publisher")
                    :as "publisher")
            (concept-scheme :via ,(s-prefix "dcat:themeTaxonomy")
                      :as "theme-taxonomy")
            (catalog-record :via ,(s-prefix "dcat:record")
                      :as "record"))
  :has-many `((dataset :via ,(s-prefix "dcat:dataset")
                    :as "datasets"))
  :resource-base (s-url "http://your-data-stories.eu/catalogs/")
  :on-path "catalogs")
```

This ensures there is a catalog resource hosted on /catalogs which contains entries of type dcat:Catalog. The entries have 9 different properties, and 4 relationships. If the YDS data model changes, we would alter this snippet and the system would consume and produce a different model. This approach allows us to be flexible with regards to the model.

The mu-cl-resources microservice offers support for declaratively specifying resources, and it offers a full JSON API compliant API for the specified resources.

5 Available APIs

The current version of the Open Data Repository supports both a SPARQL endpoint as well as a JSON API endpoint. Linked data aficionados as well as people that expect full control over the returned data will likely prefer the SPARQL endpoint, whereas future driven web developers will likely use JSON API as their tool of choice.

All content can be retrieved through the SPARQL endpoint. It can return the content as a JSON string. The data model has been defined in T3.2 and the SPARQL query language is fully described in SPARQL 1.1 [1]. We don't go in more depth on the use of SPARQL in this deliverable as it is mostly suited for linked data experts who are already familiar with the technology.

5.1 JSON API

The JSON API compliant specification supports all resources of the current data model. It also supports pagination and experimental filtering. We supply some examples so an initial impression can be formed on the responses, but urge the interested reader to read up on [2].

All resources are hosted top-level. The related requests for a particular request are supplied in the top-level links set of the response. Each resource has a self, which can be used to retrieve that specific resource. If we retrieve a listing of catalogs, the following request/response could occur:

```
REQUEST /catalogs

RESPONSE
{
 "data": [
  {
   "attributes": {
    "title": "YDS example catalog",
    "description": "A catalog which is built as an example for D3.9"
   },
   "id": "4646989B-D91A-4685-AA2F-03EBEC06CE8F",
   "type": "catalogs",
   "relationships": {
    "datasets": {
     "links": {
      "self": "/catalogs/4646989B-D91A-4685-AA2F-03EBEC06CE8F/links/datasets",
      "related": "/catalogs/4646989B-D91A-4685-AA2F-03EBEC06CE8F/datasets"
     }
    },
    "publisher": {
     "links": {
      "self": "/catalogs/4646989B-D91A-4685-AA2F-03EBEC06CE8F/links/publisher",
      "related": "/catalogs/4646989B-D91A-4685-AA2F-03EBEC06CE8F/publisher"
     }
    },
      "theme-taxonomy": {
        "links": {
          "self": "/catalogs/4646989B-D91A-4685-AA2F-03EBEC06CE8F/links/theme-taxonomy",
          "related": "/catalogs/4646989B-D91A-4685-AA2F-03EBEC06CE8F/theme-taxonomy"
        }
      },
      "record": {
        "links": {
          "self": "/catalogs/4646989B-D91A-4685-AA2F-03EBEC06CE8F/links/record",
          "related": "/catalogs/4646989B-D91A-4685-AA2F-03EBEC06CE8F/record"
        }
      }
    }
  }
 ],
 "links": {
   "last": "/catalogs&page[number]=2",
   "first": "/catalogs",
   "next": "/catalogs&page[number]=1"
 }
}
```

Not all attributes were specified in the database. The attributes which are specified and described will be returned. If a links object is specified at the top-level, it contains references to related links. This could be for pagination, or for finding a (set of) related resource(s).

Search/filtering of responses is supported too, however it is experimental and support is subject to change. How filtering evolves, will depend on the uses as they show up in the developed YDS components. JSON API specifies filtering in an open-ended way. We provide a few examples in this follow up to make our choices simpler to digest.

The simplest form of filtering is to filter on an attribute. We could search for all catalogs with YDS in the title by querying the path /catalogs?filter[title]=YDS. Filter paths can be used too. Searching for all catalogs which contain a dataset which contains the word *"links"* in the description could be done by issuing /catalogs/filter[datasets][description]=links. You can supply more than one filter in a request, all conditions must hold in that case. Filtering can also occur on the identifier of a linked resource. In order to find all datasets which contain both the theme with identifier *"ABC"* as well as the theme with identifier *"DEF"*, one could search for /datasets?filter[themes]=ABC&filter[themes]=DEF.

The content which is offered through the JSON API is easy to discover once a start point has been found. An element of future work would be to auto-document this specification. This would yield a large document which could be used for discovering what is available in the repository. Reading the declarative specification used by mu-cl-resources will help until then.

The mu-cl-resources service provides a standardized JSON view on the content of the YDS repository. This provides us with a flexible and easy-to-use endpoint for simple requests. Given that this component is central to the use of the platform, we expect it to evolve over time as we gain more experience with the components of the YDS platform.

6 Future Vision

This is the first release of the Open Data Repository. Further releases will aid the scaling of the repository and will extend the supported calls.

As indicated through the text, we mainly need to tackle the scaling of the central store. Although more research is needed to find an optimal approach, we foresee two main approaches. In a first approach, we could replicate the triplestore. This approach would be easy if the content in the triplestores would never change. In case the content of the triplestore can change, we will need to find an approach which shares the changes amongst the triplestores. It is possible that the queries themselves become too heavy for the triplestore, regardless of the amount of stores we have. In that case, we should reach for alternative stores for some responses. We could export the content of the triplestore to a document store for specific views. We will work on finding good ways to scale this architecture.

Next on the list is supporting more calls. The main way in which we foresee this to happen is through the creation of new microservices. As these services are available to the users, they can offer custom views on known data. We also intend to investigate plugins for mu-cl-resources which would allow the requests or responses to be altered for specific calls.

The evolution of our approach will depend heavily on how we foresee the use of the Open Data Repository to evolve. As we gain more input from the YDS components, it becomes clearer what we can expect in the future. We will alter our approach as we see a need arise.

7 Conclusion

The current version of the Open Data Repository is based on user-facing microservices which store their data in a shared triplestore. It provides support for retrieving the contents either through JSON API, or through SPARQL. Both of these variants support a different audience. The architecture of the platform provides options for future scaling and offers sufficient flexibility should the need arise in the components.

8 References

[1] SPARQL 1.1 Overview <http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/>
[2] JSON API <http://jsonapi.org>
[3] Mu-semtech <http://github.com/mu-semtech>
[4] A. Versteden, E. Pauwels, and A. Papantoniou. *An Ecosystem of User-facing Microservices supported by Semantic Models*. The 5th International USEWOD Workshop, Portoroz, Slovenia. 2015. <http://usewod.org/files/workshops/2015/papers/USEWOD15_versteden_pauwels_papantaniou.pdf
>
[5]Mu-cl-resources <https://github.com/mu-semtech/mu-cl-resources>
[6] Open Link Virtuoso <http://virtuoso.openlinksw.com/>
[7] JSON <http://json.org>
[8] Docker <https://www.docker.com/>
[9] Docker Compose <https://docs.docker.com/compose/>
[10] Mu-dispatcher <https://github.com/mu-semtech/mu-dispatcher>
[11] Mu-identifier <https://github.com/mu-semtech/mu-identifier>