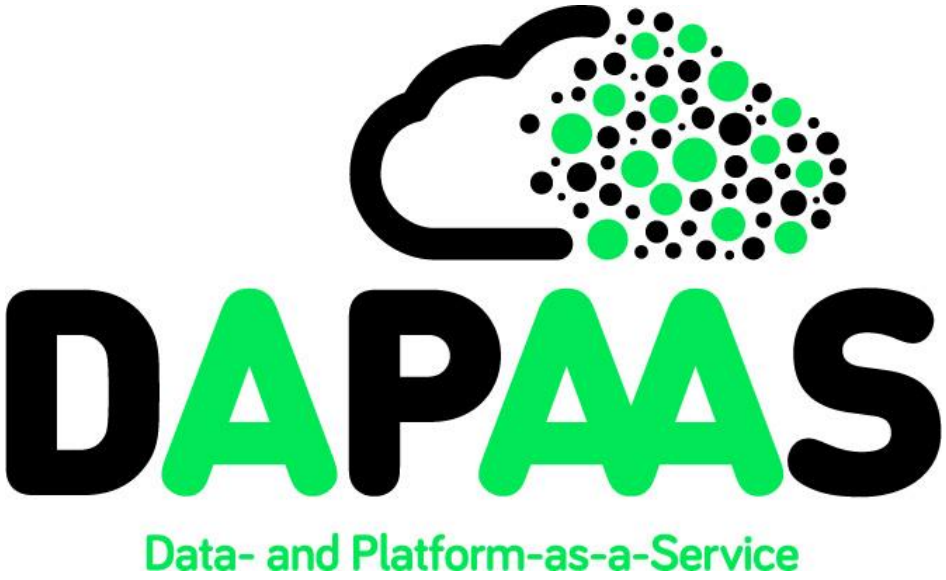
Small or medium-scale focused research project (STREP)

ICT SME-DCA Call 2013
FP7-ICT-2013-SME-DCA

Data Publishing through the Cloud: A Data- and Platform-as-a-Service Approach to Efficient Open Data Publication and Consumption

DaPaaS



Deliverable 1.2

Open DaaS prototype, v.1

Date:	31.10.2014
Author(s):	Marin Dimitrov (Ontotext), Alex Simov (Ontotext), Dumitru Roman (SINTEF), Brian Elvesæter (SINTEF)
Dissemination level:	PU
WP:	WP1
Version:	1.0

Document metadata

Quality assurors and contributors

Quality assuror(s)	Bill Roberts (Swirrl) Seonho Kim (Saltlux)
Contributor(s)	DaPaaS Consortium

Version history

Version	Date	Description
0.1	30.09.2014	Initial outline and Table of Contents (TOC)
0.2	15.10.2014	Revised outline
0.3	24.10.2014	APIs, components descriptions
0.4	27.10.2014	Requirements addressed sections
0.5	30.10.2014	Internal review comments
0.6	30.10.2014	Updates according to internal review
1.0	31.10.2014	Final version

Executive Summary

The main goal of the DaPaaS project is to provide an integrated Data-as-a-Service (DaaS) and Platform-as-a-Service (PaaS) environment, together with associated services, for open data, where 3rd parties can publish and host both datasets and data-driven applications that are accessed by end-user data consumers in a cross-platform manner.

This document describes the first version of the DaaS prototype developed by M12 of the project. The DaaS prototype consists of a set of software components that were developed based on the requirements, design and architecture specification from Deliverable D1.1.

The development has been performed in parallel with the activities on Deliverable D2.2 “Open Data PaaS prototype, v.1” in order to ensure seamless integration and components reuse.

The software components developed and integrated for the first version of the prototype are:

- **Data Warehouse** – a set of storage and data access facilities covering the core functional requirements of the DaaS layer. The main focus of the component is oriented towards linked data management in RDF.
- **Data Import/Export** services – a set of adapters enabling the users to load different types of data by transforming them into RDF as well as exporting the original uploaded data files or the results from the RDF transformation.
- **Data access & querying** – a set of APIs allowing different types of access to the data in the repositories. This includes accessing directly RDF data, querying the data via SPARQL expressions, updating the data via SPARQL 1.1 Update.
- **Datasets Catalog** services are means for providing metadata for the (linked) data stored in the DaaS platform.

These components are available through the DaPaaS platform and accessible for data publishers, application developers and data consumers through REST APIs and graphical front-ends. User guides and API documentation are included in the Appendices of this document.

Table of Contents

List of Acronyms

API	Application Programming Interface
CSV	Comma Separated Values (format)
DaaS	Data-as-a-Service
DCAT	Data Catalog Vocabulary
FOAF	Friend of a Friend (vocabulary)
JSON	JavaScript Object Notation (format)
PaaS	Platform-as-a-Service
R2RML	Relational to RDF Mapping Language
RDF	Resource Description Framework
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SPARQL	SPARQL Protocol and RDF Query Language
VoID	Vocabulary of Interlinked Datasets

List of Figures

List of Tables

1 DaaS Requirements & Architecture

This document represents supporting documentation for Deliverable D1.2 (Prototype) and addresses requirements outlined in Deliverable D1.1[1]. An online integrated version of the DaPaaS platform[2], including all available components from WP1-WP3 (D1.2, D2.2 and D3.2) is available to all consortium members. A publicly accessible version will be provided at the beginning of year 2, according to the plan.

The goals of this deliverable are to provide:

- A more detailed architecture specification and identification of the main components of the DaaS layer
- An overview description of the first prototype implementation
- Detailed APIs specification and documentation for consumption of the core functionalities of the DaaS layer

Deliverable D1.1 outlined a set of requirements for the DaPaaS platform. The tables below summarize how the current version of the prototype as of M12 addresses these requirements.

Table 1: Data Publisher (DP) requirements addressed

ID	Name	Brief description of how the current version of the prototype addresses the requirement
DP-01	Dataset import	The Data Publisher is provided with a set of functionalities to import data from various formats into the data warehouse as RDF. The first prototype supports: • direct RDF data import • transform tabular CSV data into RDF • transform relational data into RDF • arbitrary complex transformations based on Grafter
DP-02	Data storage & querying	Each dataset provides SPARQL endpoint for querying and updates. Additionally, and API is provided for bulk loading of RDF data.
DP-03	Dataset search & exploration	A Catalog service based on DCAT vocabulary is available. Search is implemented on top of metadata fields like titles, descriptions, keywords.
DP-04	Data interlinking	The semi-automatic interlinking requires means (UI) for creating simple mappings which is not available by M12; support of this functionality by the DaaS layer is planned for year 2.
DP-06	Dataset bookmarking & notifications	This requirement is not considered as core functionality for the first prototype implementation and will be addressed during year 2.
DP-07	Dataset metadata management, statistics & access policies	The Catalog service takes care of management of metadata. The access control policies are also part of the meta descriptions.
DP-08	Data scalability	The current load of the system does not cause scalability issues. Further investigations on this feature will be considered with the increase of data that will be made available on the platform during year 2.

[1] http://project.dapaas.eu/dapaas-reports/deliverable-11-open-daas-requirements-design-architecture-specification
[2] http://dapaas.ontotext.com/demo/

DP-09	Data availability	The current prototype does not cause data availability issues. Further investigations on this feature will be considered with the increase of data that will be made available on the platform during year 2.
DP-11	Secure access to platform	Security aspects are not considered a priority for the first prototype. The public access to the platform will be secured by HTTPS and considered during year 2.

Table 2: Application Developer (AD) requirements addressed

ID	Name	Brief description of how the current version of the prototype addresses the requirement
AD-01	Access to Data Publisher services (DP-01 – DP-13)	The Application Developer has the same access to the APIs as the Data Publisher has (see table above).
AD-02	Data export	The Application Developer has the ability to export RDF data (repository level export); sub-graphs export using SPARQL CONSTRUCT queries; and export of raw non-RDF data.

Table 3: End-Users Data Consumer (EU) requirements addressed

ID	Name	Brief description of how the current version of the prototype addresses the requirement
EU-02	Search & explore datasets and applications	The end-users can search and browse the metadata of both datasets and applications as long as they are allowed by the access control mechanisms.
EU-03	Datasets and applications bookmarking and notifications	This requirement is not considered as core functionality for the first prototype implementation and will be addressed during year 2.
EU-05	Data export and download	Refer to AD-02.
EU-06	High availability of data and applications	This feature will be investigated once significant amounts of data apps will be made available on the platform during year 2.

Table 4: Instance Operator (IO) requirements addressed

ID	Name	Brief description of how the current version of the prototype addresses the requirement
IO-02	Platform performance monitoring	Partial support for measuring the resources consumption of the system is in place (various metrics from AWS Cloud Watch[3]), a centralized component for collecting the various metrics and being able to represent it in adequate way is part of plans for year 2.
IO-03	Statistics monitoring (users, data, apps, usage)	The first version of the prototype will be accessible to limited number of users; this requirement is not considered as a high priority issue and will be reconsidered with the public version of the DaPaaS platform.

[3] http://aws.amazon.com/cloudwatch/

IO-05	Policy/quota configuration and enforcement	Quota enforcement will be addressed in the next version of the platform by adoption of Docker[4] containers for both datasets and applications isolation.

[4] https://www.docker.com/

2 Prototype Design and Implementation

Figure 1 depicts the main software components, their relationships and associated APIs of the Data Layer as defined in D1.1. The software components of the Data Layer provide the following main capabilities:

- Open Data Warehouse (Storage & Access)
- Data & Metadata Catalog
- Query & Update
- Import & Export
- Interlinking
- Notifications & Statistics

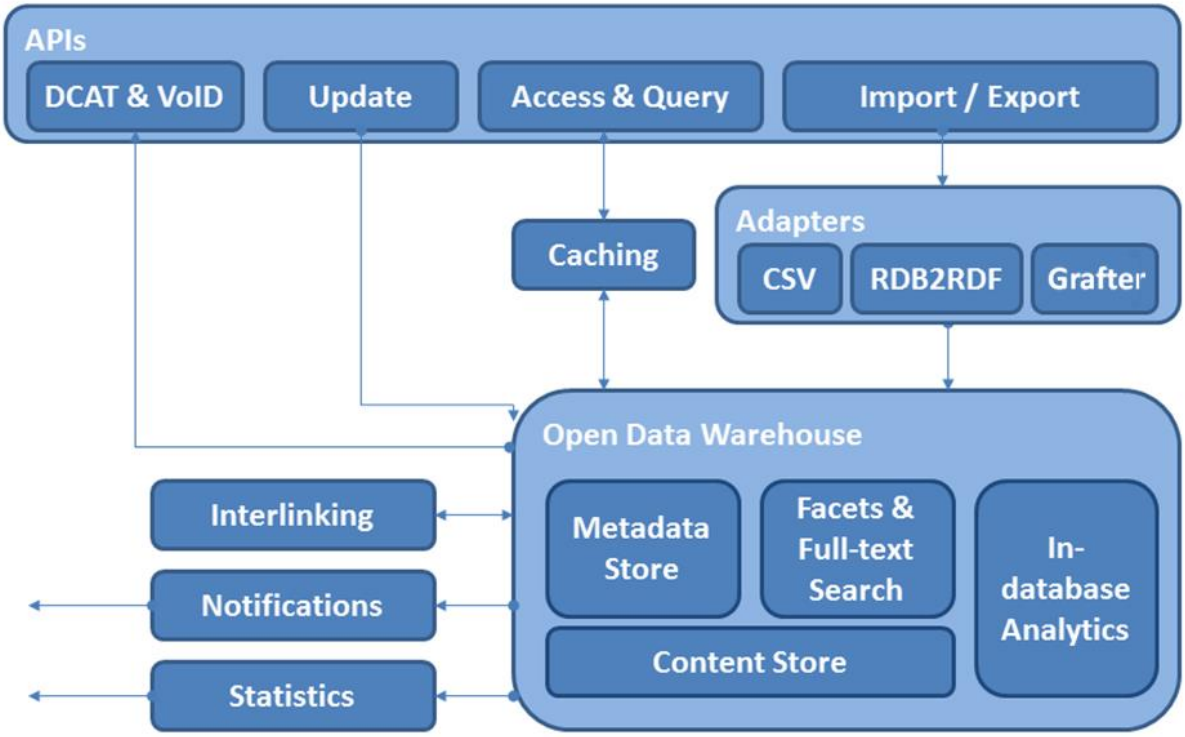


Figure 1. Data Layer Architecture

2.1 Open Data Warehouse

The implementation of the warehouse comprises two major components responsible for hosting the various types of data. The primary focus of the first prototype is oriented towards working with RDF data and therefore the metadata store is the key component in the platform. Any other types of data can only be stored and retrieved.

2.1.1 Metadata Store

The purpose of this component is to provide storage and querying capabilities for linked data represented as RDF (i.e. *Linked Data* store). The underlying implementation is supported by the GraphDB (formerly known as OWLIM) engine, accessible via the OpenRDF Sesame framework. Extended overview of the architecture and functionalities of the two systems was provided in D1.1. Here we focus on the actual deployment and the Sesame APIs we utilize for the prototype.

2.1.1.1 Multi-tenant model

To ensure isolation of the data between the users of the platform our strategy is to instantiate separate repository for each user so that no two users share the same container. The users can have more than one repository but each repository can have exactly one owner. Collaboration between different users on the same data still can be achieved by sharing the repository by using the access control mechanisms of the system. Of course, only the repository owner can grant or restrict access to the data.

2.1.1.2 Security & access control

On creation of a new repository, the user is provided with unique access URL to the repository. The access control mechanism guarantees that only this user can reach the URL unless he or she granted access to another user. The owner of the repository controls the access type for other users: read-only, read-write or none.

2.1.2 Content Store

The primary purpose of the content store in the first prototype is to keep track of the non-RDF files being uploaded into the system. Such files are usually inputs for the data import adapters where their content passes certain transformations before it reaches its final destination into the repository.

To be able to restore the original content and formatting of the data it is necessary that each file is stored in its original form into the content store prior to any subsequent processing. The content store itself is based on Amazon S3[5] service, providing secure, durable, highly-scalable object storage.

The mechanism for retrieval of original data is via the *Raw Data Export* API (see Section 5.1.2).

2.1.3 Full-Text Search

For full-text search (FTS) support we reuse the one already integrated into the metadata store (GraphDB). It is based on Apache Lucene[6], a high-performance, full-featured text search engine written entirely in Java.

GraphDB supports full text search capabilities using Lucene with a variety of indexing options and the ability to simultaneously use multiple, differently configured indices in the same query. The query language which is also used for index management is based on extension of the SPARQL and thus delivering the FTS functionality to any SPARQL component straight forward.

In order to use Lucene full-text search in GraphDB a Lucene index must first be computed. Before being created, each index can be parameterised in a number of ways using SPARQL 'control' updates. This provides the ability to:

- select what kinds of nodes are indexed (URIs/literals/blank-nodes)
- select what is included in the 'molecule' (entities relevant text) associated with each node
- select literals with certain language tags
- choose the size of the RDF 'molecule' to index
- select alternative analysers
- select alternative scorers

2.2 Import & Export Adapters

2.2.1 CSV import

The CSV import service uses generic algorithm for representing tabular data into RDF. Each CSV row is transformed to a single RDF entity, having its properties extracted from the corresponding row cells. The presence of a file header is required for generating property names properly. The implementation

[5] http://aws.amazon.com/s3/
[6] http://lucene.apache.org/core/

is based on the *Apache Any23*[7] toolkit. The service accepts as input either CSV content directly or URL where the actual data is accessible. The result data is loaded directly into the data warehouse in a RDF repository specified in advance.

Complete API specification is provided in Section 5.1.1.

2.2.2 RDB import

This service provides data import functionality from RDBMS. The result is RDF data loaded in the data warehouse. The service is compliant with the two leading RDB-to-RDF W3C standards for mapping relational data into RDF. As input the service requires the parameters of the source RDBMS, the result RDF repository and the mapping strategy to be applied. In the simpler case a *Direct Mapping*[8] transformation can be applied without any additional resources required. In the case of *R2RML*[9] application, the actual transformation document should be provided as well. For the latter there is no *default* mapping so the transformation document is expected to be created in advance.

Complete API specification is available in section 5.1.1.

2.2.3 Grafter import

This adapter is a wrapper of the Grafter execution engine intended to be applied repeatedly using transformations created in advance. Each transformation is a bundled artefact of transformation instructions, registered in the system and having a unique identifier. At runtime the service expects an identifier of the transformation to be applied, the source data (CSV or XLS) and the location of the repository to store the final RDF result. The adapter exposes several additional API methods for managing the transformations life cycle (registration, listing, unregistration).

Complete API specification is available in section 5.1.1.

2.2.4 RDF export

The RDF export adapter is based on the OpenRDF Sesame framework[10] APIs with a thin layer on top for user access control management. The API itself is exposed without any functional restrictions to ensure maximal interoperability with other systems or components.

The export RDF functionality comes in two flavours:

- Exporting a complete dataset or graph from a dataset;
- Exporting a sub-graph using SPARQL constructs

In either case the result is RDF serialized in user provided format.

Complete API specification is available in section 5.1.2.

2.2.5 Raw data export

Any non-RDF data loaded into the system via the import adapters is also preserved in its original form as an archive. This data is available for download at any time and it's identical to the version published (any subsequent modifications on the data in the data warehouse are not reflected).

Complete API specification is available in section 5.1.2

2.3 Access, Query & Update

2.3.1 SPARQL Query & Update

SPARQL is a query language and protocol for RDF. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware.

[7] https://any23.apache.org/index.html
[8] http://www.w3.org/TR/rdb-direct-mapping/
[9] http://www.w3.org/TR/r2rml/
[10] http://rdf4j.org/

SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports aggregation, subqueries, negation, creating values by expressions, extensible value testing, and constraining queries by source RDF graph. The results of SPARQL queries can be result sets or RDF graphs.

SPARQL Update is an update language for RDF graphs. It uses a syntax derived from the SPARQL Query Language for RDF. Update operations are performed on a collection of graphs in a Graph Store. Operations are provided to update, create, and remove RDF graphs in a Graph Store.

2.3.2 OpenRDF Sesame APIs

There are two approaches for accessing the RDF data from the data warehouse: direct data access and via SPARQL based communication.

The direct data access APIs works on the level of RDF triples (or quads if graphs are used). All requests go to the same URL (**/repositories/{id}/statements**), but using different parameters and different HTTP methods depending on the operation required:

- GET – retrieval of the data from the repository. Optionally the result can be filtered by certain sub-components of the RDF statement: subject, predicate, object or graph. Any combination of filters is acceptable and valid restrictions in the filters are concrete values rather than wildcards. The absence of a filter means unrestricted value.
- POST – addition of RDF data to the repository. The new data is supplied in the request in valid RDF format. Optionally a 'context' parameter can be specified to restrict the operation to a specific graph.
- PUT – update of RDF data in the repository. This operation removes any old data contained in the repository and loads the new one supplied in the request. Context parameter can be specified to restrict the operation to a specific graph.
- DELETE – removal of statements from the repository. Restriction parameters as the ones from the GET operation are valid here as well. A request without filter parameters removes all the data from the repository.

The SPARQL based data query and update is supported by two separate API endpoints (URLs). The SPARQL querying is done by sending the query expression and the desired format of the results. Depending on the query type the nature of the result can be either RDF data (CONSTRUCT) or variable bindings set (SELECT).

The SPARQL Update service is an extension of the direct data access API (POST) but instead of providing RDF data, an additional 'update' parameter is supplied. This parameter contains the *SPARQL Update* expression which describes the required updates (insertion or/and removal).

Complete API specification is available in Section 5.3.

2.4 Catalog Services

The Catalog services play an important role for the integrity of the various components of the DaaS layer. It is an integration component between data, metadata, user interface, user management and access control. The protocol for data exchange is based on the emerging W3C standard for datasets descriptions. All catalog service APIs conform to the DCAT vocabulary wherever applicable.

2.4.1 DCAT

DCAT is an RDF vocabulary designed to facilitate interoperability between data catalogs published on the Web. By using DCAT to describe datasets in data catalogs, publishers increase discoverability and enable applications easily to consume metadata from multiple catalogs. It further enables decentralized publishing of catalogs and facilitates federated dataset search across sites.

DCAT incorporates terms from pre-existing vocabularies, where stable terms with appropriate meanings could be found, such as foaf:homepage and dct:title. Informal summary definitions of these terms are included here for convenience, while complete definitions are available in the provided authoritative

references. Changes to definitions in those references, if any, will supersede the summaries given in this specification.

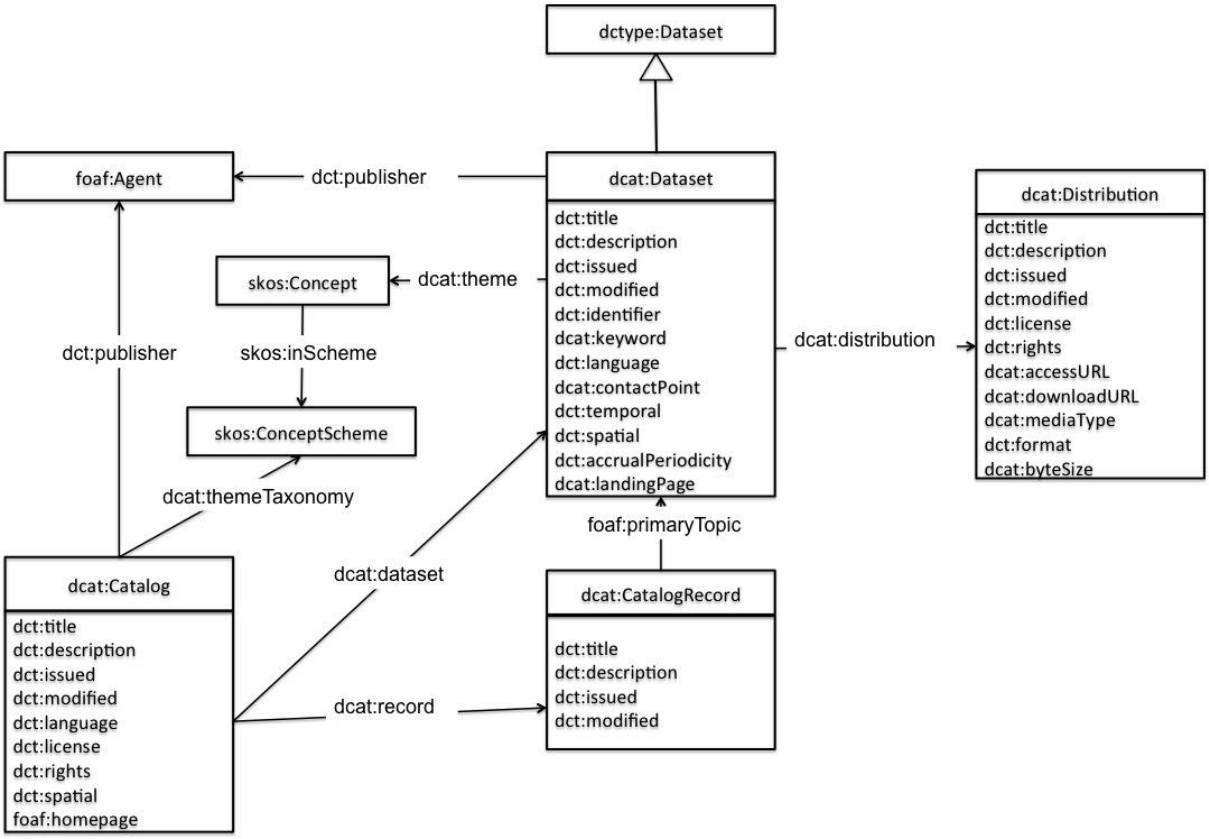


Figure 2. DCAT model

DCAT defines three main classes:

- **dcat:Catalog** represents the catalog
- **dcat:Dataset** represents a dataset in a catalog.

  dcat:Distribution represents an accessible form of a dataset as for example a downloadable file, an RSS feed or a web service that provides the data.

Dataset in DCAT is defined as a "collection of data, published or curated by a single agent, and available for access or download in one or more formats". A dataset does not have to be available as a downloadable file. For example, a dataset that is available via an API can be defined as an instance of *dcat:Dataset* and the API can be defined as an instance of *dcat:Distribution*. DCAT itself does not define properties specific to APIs description. These are considered out of the scope of the vocabulary.

2.4.2 Catalog APIs

The APIs support the complete set of CRUD operations for the different types of entities in the system. Following the approved RESTful design patterns we have the following action to http methods mapping:

- GET – retrieve an entity description
- POST – create a new description
- PUT – update an existing description
- DELETE – delete a description

The supported input/output data formats for the various API methods are in accordance with the DCAT vocabulary - any standard RDF serialization or JSON-LD.

The following tables list the main functionalities of the catalog APIs. Complete description of the API is provided in the Annex of this document.

Table 5. Datasets management

Method	URL	Description
GET	/users/{user-id}/catalog	List a catalog of datasets for the user (owned and shared)
GET	/users/{user-id}/datasets	Get details about certain dataset (meta data)
POST	/users/{user-id}/datasets	Create a new dataset (meta data)
PUT	/users/{user-id}/datasets	Update an existing dataset (meta data)
DELETE	/users/{user-id}/datasets	Delete a dataset
GET	/users/{user-id}/datasets/search	Search for datasets based on meta-data (titles, descriptions, keywords)

Table 6. Distributions management

Method	URL	Description
GET	/users/{user-id}/distributions	Get description of a distribution
POST	/users/{user-id}/distributions	Create a new distribution (description)
PUT	/users/{user-id}/distributions	Update an existing distribution
DELETE	/users/{user-id}/distributions	Delete distribution

The following example demonstrates a simple data catalog retrieval invocation and the result in JSON-LD.

Request via cURL[11] from the command line:

$ curl -H "Accept:application/ld+json" http://dapaas.ontotext.com/catalog/users/Alex/catalog

And the corresponding response:

```
{
   "@context": { ... }
   "@type": "dcat:Catalog",
    "dct:publisher": "Alex",
    "@id": "http://dapaas.eu/users/Alex/datasets",
    "dcat:record": [
        {
            "@type": "dcat:CatalogRecord",
            "foaf:primaryTopic": "http://eu.dapaas/dataset/1",
            "dct:issued": "2014-09-16",
            "dct:modified": "2014-09-17",
            "dct:title": "My first DaPaaS dataset"
        },
        {
```

[11] http://en.wikipedia.org/wiki/CURL

```
            "@type": "dcat:CatalogRecord",
            "foaf:primaryTopic": "http://eu.dapaas/dataset/2",
            "dct:issued": "2014-09-15",
            "dct:modified": "2014-09-17",
            "dct:title": "My second DaPaaS dataset"
        }
    ]
}
```

The complete catalog API documentation can be found in the Annex (Section 5.2).

2.5 Component Workflow

The following diagram (Figure 3) outlines the major components of the DaaS platform layer and their dependencies to other modules. Some of the components are common for the data and the platform layer like user management and access control and catalog services.

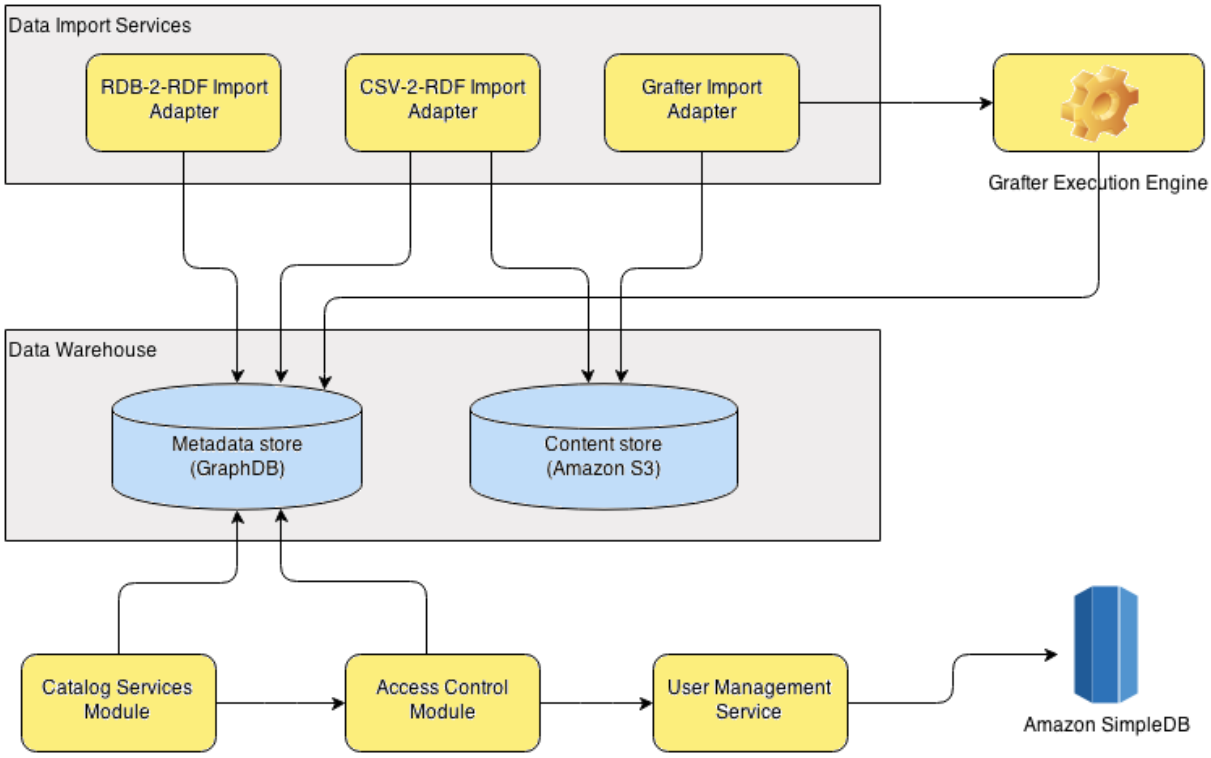


Figure 3. Prototype Components Overview

The import adapters receiving tabular data files as input store the original files in the *Content store* and then perform the RDFization step which loads the result in the *Metadata store*. Additionally the *Grafter Import Adapter* needs the content store to retrieve an existing transformation to apply it on the input data.

The *RDB-2-RDF Import Adapter* receives parameters to external RDBMS rather than actual data so it needs only the metadata store to load the results produced.

The *User Management Service* is quite self-contained component requiring only persistence mechanism to store user data. This information is potentially sensitive containing account passwords and e-mail so we cannot reuse the Data Warehouse for that. So we chose to use Amazon SimpleDB[12] service which is a highly available and flexible non-relational data store.

[12] http://aws.amazon.com/simpledb/

The *Access Control* module matches datasets and applications and with users and allowed operations so it needs access to both - the metadata store and the user management service.

The *Catalog Services* keep track of the content of the whole data warehouse so its major counterpart on the storage layer is the metadata store. To ensure adequate content exploration the service needs support by the *Access Control* module.

3 Deployment Details

3.1 Hardware, OS, etc

The first version of the DaaS platform layer is hosted on two machines and a set of external cloud services. The metadata store which is part of data warehouse requires significant amount of RAM memory and intensive CPU utilisation. Therefore it occupies an entire machine with 64GB RAM[13]. The rest of the data warehouse (that is, the *Content store*) which contains data organized in files is hosted on Amazon Simple Storage Service[14] (S3).

The second machine required to run the DaaS layer is a moderate performance one hosting various services which are either used rarely or does not require computational resources (management services, import adapters, catalogs).

Concerning the operating system requirements, all the components are based on Java so they can run on any server platform. The current deployment is based on Ubuntu 12.04.2 LTS.

3.2 Software (3rd party)

The following third party components are supporting the platform:

- Apache Tomcat 7.0 (applications container)
- OpenRDF Sesame framework (RDF management middleware)
- AWS SDK for Java (java libraries for working with the Amazon services)
- Apache CXF framework (web services development framework)
- Apache Any23 (transformation framework from various data formats to RDF)
- Antidot db2triples (open source RDF2RDF library implementing *DirectMapping* and *R2RML*)

3.3 DaPaaS components

The data layer of the platform is organized into several components, most of them packaged as web applications and exposed as RESTful web services:

- Data access services
- Data import services
- Catalog services
- User management and access control
- Grafter execution engine (WP4)

[13] These are rough estimates as by the time of writing this document there are still no real world datasets hosted by the platform.
[14] http://aws.amazon.com/s3/

4 Future Work

Functionalities not addressed in the first prototype (M12) of the platform:

- Linked Data Platform[15] (LDP)

  This feature was not originally planned, however compliance with emerging standards for Linked Data oriented applications interoperability is of great importance for the DaPaaS platform. The LDP specification describes a set of best practices and simple approach for a read-write Linked Data architecture, based on HTTP access to web resources that describe their state using the RDF data model.

- Automated and supervised interlinking between datasets

  This feature depends on the availability of visual (Web UI) means for specification of interlinking parameters and mappings. Both the visual environment and the interlinking discovery engine will be addressed in the second version of the platform.

- Notifications
- Statistics and monitoring
- Docker[16] based multi-tenancy for repositories

  The application of Docker containers technology to repositories will improve the processes isolation between the users as well as it will enable resources usage quota management.

- Database analytics

[15] http://www.w3.org/TR/ldp/
[16] https://www.docker.io/

5 Annexes

5.1 API Documentation

5.1.1 Data Import APIs

CSV-2-RDF Import

URL	/dapaas-import-services/csv2rdf
HTTP Method	POST
Description	Performs CSV-to-RDF transformation and loads the result in certain repository
Inputs (HTTP headers)	**publisher** - the publisher ID from the catalogs **repository-url** - target repository to store the result **repository-graph** - optional graph in the repository **sep** - field separator (optional) **base-uri** - base URI for the result RDF data Data supply: a) remote CSV file import: **source-url** - source to CSV **source-user** - user **source-pass** - password b) csv data upload as multipart attachment **file** - the CSV content **name** - the original file name of the csv file
Response	Operation completion status (HTTP response code)

RDB-2-RDF Import

URL	/dapaas-import-services/rdbv2rdf
HTTP Method	POST
Description	Performs RDB-to-RDF transformation and loads the result in certain repository
Inputs (HTTP headers)	**db-url** - source RDBMS **db-user** - source database user **db-pass** - source database password **db-name** - database name **db-drive**r - RDBMS driver **repository-url** - target repository to store the result **repository-graph** - optional graph in the repository **base-uri** - base URI for the result RDF data **mode** - one of '*direct*' and '*r2r*' r2rml mapping file (as body content)
Response	Operation completion status (HTTP response code)

Grafter Import

URL	/dapaas-import-services/grafter/apply
HTTP Method	POST
Description	Performs CSV-2-RDF data transformation with Grafter and loads the result in certain repository

Inputs (HTTP headers)	**publisher** - the publisher ID from the catalogs **repository-url** - target repository to store the result **repository-graph** - optional graph in the repository **transformation-id** - the id of the specific Grafter transformation to be used CSV data upload as multipart attachment **file** - the CSV content **name** - the original file name of the csv file
Response	Operation completion status (HTTP response code)

Grafter transformations management:

URL	/dapaas-import-services/grafter/register
HTTP Method	POST
Description	Registers a new transformation into the DaPaaS platform
Inputs (HTTP headers)	**transformation-id** - the id for the new Grafter transformation **description** - free text description of the transformation, what it does, what it requires, etc. jar binary as multipart attachment **file** - the jar content **name** - the original file name of the jar file (it is not necessary to be the same as the ID)
Response	Operation completion status (HTTP response code)

URL	/dapaas-import-services/grafter/list
HTTP Method	GET
Description	Lists all available (registered) transformations in the platform
Inputs	-
Response	JSON array of objects representing registered transformations (name and description). Examlpe: [{ "id' : "myGrafter1", "descr":"This is my transformation text" }, ...]

URL	/dapaas-import-services/grafter/unregister
HTTP Method	DELETE
Description	Unregisters a transformation from the platform
Inputs (HTTP headers)	**transformation-id** - the id of the specific Grafter transformation to be unregistered (removed)

Response	Operation completion status (HTTP response code)

5.1.2 Data Export APIs

RDF Export

URL	/repositories/<repository-id>/statements
HTTP Method	GET
Description	Exports the content (RDF statements) of a repository, optionally filtered by subject, predicate, object and graph
Inputs	**repository-id** - id contained in the distribution description Optional URL parameters: **subj** - filter by subject URI **pred** - filter by predicate URI **obj** - filter by object value (literal or URI) **context** - filter by graph (export of statements from a certain graph only) HTTP header: **Accept** - serialization format for the result. Valid formats are: *application/rdf+xml*, *text/plain*, *text/turtle*, *text/rdf+n3*, *text/x-nquads*, *application/rdf+json*, *application/trix*, *application/x-trig*, *application/x-binary-rdf*
Response	RDF data serialized in the desired format

Raw Data Export (non RDF data)

URL	/users/{user-id}/files/{id}
HTTP Method	GET
Description	Retrieves a single file by ID.
Inputs	**id** - the identifier of the file taken from the distributions catalog **user-id** - the owner of the file
Response	The original file imported into the platform

5.2 Data Catalog APIs

5.2.1 Catalog access API

URL	/users/{user-id}/catalog
HTTP Method	GET
Description	List user's datasets as well as public or shared ones with the user
Inputs	**user-id** - the owner id HTTP headers: **Accept** - serialization format for the result: • application/ld+json

	• application/rdf+xml or any RDF type **showShared** - includes public and shared datasets in the result as well. Valid values are 'y' and 'n'
Response	List of dataset catalog records using the DCAT vocabulary in RDF or JSON-LD

URL	/users/{user-id}/datasets/search
HTTP Method	GET
Description	Search for datasets by keywords on meta data (titles, descriptions, etc.)
Inputs	**user-id** - the owner id HTTP headers: **Accept** - serialization format for the result: • application/ld+json • application/rdf+xml or any RDF type URL parameter: **q** - the search query expression (plain text)
Response	List of dataset catalog records using the DCAT vocabulary in RDF or JSON-LD

5.2.2 Datasets descriptions access and management

URL	/users/{user-id}/datasets
HTTP Method	GET
Description	Get a dataset description (meta data) by id
Inputs	**user-id** - the owner id HTTP headers: **id** - URI of the dataset, taken from the catalog **Accept** - result serialization format
Response	Complete dataset description using the DCAT vocabulary in RDF or JSON-LD

URL	/users/{user-id}/datasets
HTTP Method	POST
Description	Create a new dataset description
Inputs	**user-id** - the owner id HTTP header: **Content-Type** - format of the data supplied dataset description as RDF or JSON-LD Note: if the description contains no @id, the system will generate one
Response	URI of the new dataset in the format: { "@id" : "http://dapaas.eu/dataset/4" }

URL	/users/{user-id}/datasets
HTTP Method	PUT
Description	Update an existing dataset description
Inputs	**user-id** - the owner id HTTP header: **Content-Type** - format of the data supplied dataset description as RDF or JSON-LD
Response	Operation completion status (HTTP response code)

URL	/users/{user-id}/datasets
HTTP Method	DELETE
Description	Delete a dataset description with all of its distributions
Inputs	**user-id** - the owner id HTTP header: **id** - URI of the dataset to be removed
Response	Operation completion status (HTTP response code)

5.2.3 Distributions management

URL	/users/{user-id}/distributions
HTTP Method	GET
Description	Get distribution description by id (contained in the corresponding dataset description)
Inputs	**user-id** - the owner id URL parameter: **distrib** - URI of the distribution, taken from the dataset description HTTP header: **Accept** - result serialization format
Response	Complete distribution description using the DCAT vocabulary in RDF or JSON-LD

URL	/users/{user-id}/distributions
HTTP Method	POST
Description	Create a new distribution description
Inputs	**user-id** - the owner id HTTP headers: **dset** - URI of the dataset containing the distribution **Content-Type** - data input format distribution description as RDF or JSON-LD

	Note: if the description contains no @id, the system will generate one
Response	URI of the new distribution in the format: { "@id" : "http://dapaas.eu/distrib/abc..." }

URL	/users/{user-id}/distributions
HTTP Method	PUT
Description	Update an existing distribution description
Inputs	**user-id** - the owner id HTTP headers: **dset** - URI of the dataset containing the distribution **Content-Type** - data input format distribution description as RDF or JSON-LD
Response	Operation completion status (HTTP response code)

URL	/users/{user-id}/distributions
HTTP Method	DELETE
Description	Delete a distribution description
Inputs	**user-id** - the owner id HTTP header: **distrib** - URI of the distribution to be deleted
Response	Operation completion status (HTTP response code)

5.3 Data Access APIs

5.3.1 RDF direct data access APIs

URL	/repositories/<repository-id>/statements
HTTP Method	GET
Description	Retrieve RDF data (statements) from a repository
Inputs	refer to 5.1.2 for details
Response	RDF data

URL	/repositories/<repository-id>/statements
HTTP Method	POST
Description	Add RDF data to a repository
Inputs	**repository-id** - id contained in the distribution description HTTP header: **Content-Type** - the format of the data supplied RDF data in the request

Response	Operation completion status (HTTP response code)

URL	/repositories/<repository-id>/statements
HTTP Method	PUT
Description	Replace the content of a repository
Inputs	**repository-id** - id contained in the distribution description HTTP header: **Content-Type** - the format of the data supplied RDF data in the request
Response	Operation completion status (HTTP response code)

URL	/repositories/<repository-id>/statements
HTTP Method	DELETE
Description	Delete RDF data (statements) from a repository
Inputs	refer to 5.1.2 for details
Response	Operation completion status (HTTP response code)

5.3.2 SPARQL Query & Update APIs

URL	/repositories/<repository-id>
HTTP Method	GET or POST
Description	Perform SPARQL query over the repository content
Inputs	**repository-id** - id contained in the distribution description HTTP header: **Accept** - the format for the expected result. Depending on the query type, valid formats are: • select: *application/sparql-results+xml*, *application/sparql-results+json* • construct: any standard RDF format (refer to 5.1.2) **query** - the SPARQL query expression provided either in the URL (GET) or in the request body (POST). The format in the latter case is: *query=<url encoded SPARQL expression>*
Response	Query evaluation results - variable bindings or RDF

URL	/repositories/<repository-id>/statements
HTTP Method	POST
Description	Apply SPARQL Update on the repository content
Inputs	**repository-id** - id contained in the distribution description HTTP header: **Content-Type** set to "*application/x-www-form-urlencoded*"

	Request body content: **update** - the SPARQL 1.1 Update expression in the form: *update=<URL-encoded SPARQL 1.1 Update expression>*
Response	Operation completion status (HTTP response code)