Python

Nov 06, 2018

Contents

Note: This documentation refers to the Alpha version of the DAF (released in October 2017) and it is daily updated and improved. For comments and enhancement requests about the documentation please open an issue on GitHub.

The Data & Analytics Framework (DAF, in short) is an open source project developed in the context of the activities planned by the Italian Three-Year Plan for ICT in Public Administration 2017 - 2019, approved by the Italian Government in 2017.

The DAF project is an attempt to establish a central Chief Data Officer (CDO) for the Government and Public Administration. Its main goal is to promote data exchange among Italian Public Administrations (PAs), to support the diffusion of open data, and to enable data-driven policies. The framework is composed by three building blocks:

- **A Big Data Platform**, to store in a unique repository the data of the PAs, implementing ingestion procedures to promote standardization and therefore interoperability among them. It exposes functionalities common to the Hadoop ecosystem, a set of (micro) services designed to improve data governance and a number of end-user tools that have been integrated with them.
- **A Team of Data Experts** (Data Scientists and Data Engineers), able to manage and evolve the platform and to provide support to PA on their analytics and data management activities in a consultancy fashion.
- **A Regulatory Framework**, that institutionalizes this activity at government level, and gives the proper mandate to the PA that will manage the DAF, in compliance with privacy policy.

This documentation is focused on the **Big Data Platform**, and we'll refer to it as DAF for the sake of simplicity.

The Italian instance of the DAF is developed and maintained by the DAF Team (part of the Digital Transformation Team of the Italian Government), composed by data scientists and data engineers, which uses and evolves the framework: to analyze data, to create machine learning models and to build data applications/visualization products.

The DAF is designed to be easily re-usable in other countries and other application domains. It exposes the following data management and analytics functionalities:

- a **Public Dataportal**, a Web user interface providing:
  - a catalog of open-data datasets based on CKAN;
  - a content management system for *data stories*, which are a kind of blog post that integrates interactive charts (made using the DAF) with a narrative description of the analysis made;
  - community tools to collaborate and learn how to use the platform;
- a **Private Dataportal**, a web application with the following features:
  - a catalog of all datasets the user can access;
  - an ingestion form to govern (insert, edit, delete) datasets information and setup ingestion procedures;
  - data visualization and dashboard tools;
  - a data science notebook;
- a **Hadoop Cluster** with typical applications to centralize and store, manipulate and standardize and re-distribute data and insights;
- a **Multi-tenant** architecture, based on Kerberos and LDAP.

The DAF is under development. This is a snapshot of the roadmap:

- By October 2017: Alpha release.
- By December 2017: Alpha 2 release.
- By January 2018: Alpha 3 release.

All contributions are welcome!

Contents

CHAPTER 1

Overview

The DAF is an open source project meant to manage the data of a country's Public Administration.

Consequently, it is designed to:

- be highly scalable and configurable;
- promote data and knowledge sharing across several organizations;
- promote diffusion of open-data and insights of public interest;
- manage privacy and security issues;
- manage both batch and streaming ingestion and e-gestion processes.

The DAF is intended to support Public Administrations in developing:

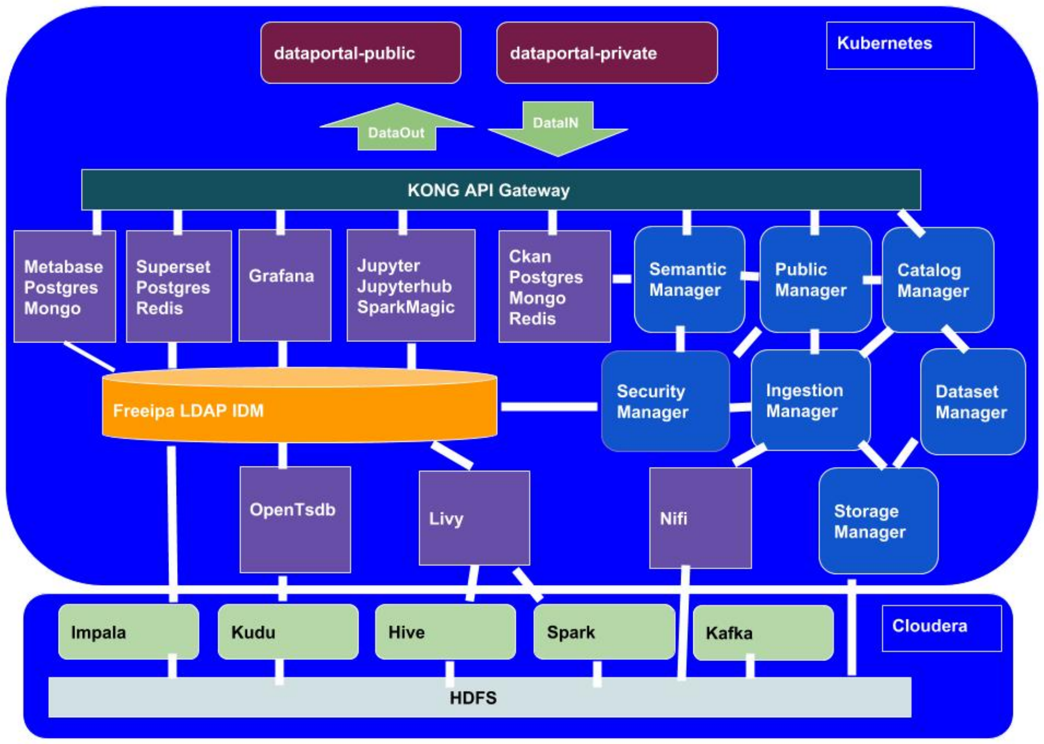
- data-driven policies;
- data applications to improve public services and internal processes;
- innovative non-critical services for citizens and businesses.

Other users of the DAF are:

- data journalists looking for information to support their journalistic theses;
- citizens looking for information regarding the Public Administration;
- community of hackers, developers, and companies that use the tools to create value-added applications and services;
- the world of research and innovation. Thanks to the DAF, it is possible to promote initiatives to involve the world of research and innovation on issues of public interest.

The DAF is based on a security system, access management and data separation that allows you to manage data access permissions. In this way, all DAF features will be exposed to all types of users mentioned above. Users will only have access to the data for which the user profile they belong to has been granted access rights.

The following image provides an architectural snapshot of the DAF architecture:



In a few words, the DAF platform integrates:

- front-end applications (dataportal-public and dataportal-private);
- front-end open source platforms, developed by third-parties (e.g. Metabase, Superset, Jupyter, CKAN);
- back-end open source big data platforms and technologies (e.g. Hadoop ecosystem, Livy, Nifi, etc.);
- microservices to manage all underlying DAF mechanisms (e.g. Catalog Manager, Ingestion Manager, etc.).

Everything is deployed on a Kubernetes cluster and relies on a Cloudera cluster.

1.1 Dataset Concept

The main aim of DAF is to provide a framework to manage data, regardless on their dimension and nature (from small vocabulary tables to big unstructured data). That's why we designed the DAF around an abstraction of the concept of *dataset*: potentially interconnected logical entities, made of metadata, data, storage options and "interactivity" capabilities. We tried to shape it to be generic enough to model both batch and streaming, structured, semi-structured and unstructured content.

1.1.1 Dataset lifecycle

Datasets follow a standard life cycle path, regardless on their nature and typology.

- **Dataset entity creation**: a dataset entity is created via a metadata management form, where the user specifies information about the dataset (following the required items of the DCATAP_IT standard), its data structure, properties and annotations of its fields/attributes, as well as operational information to govern the management of dataset in the platform (where to store the dataset, where to be listening for new data coming, who can have access to it, etc.).

- **Ingestion**: after the dataset entity has been created, a microservice activates an Apache NiFi pipeline that is listening for new data to ingest.DAF is currently ready to ingest data coming from SFTP (default option for batch data), pull and push from an external webservice.
- **Transformation and enrichment pipelines**: before being stored into the appropriate storage engine, the data goes through several pipelines that add information to the incoming data. We are currently developing the following two pipelines:
  - **Normalization pipeline**, to apply DAF internal conventions to raw incoming data, such as format (UTF-8), management of null entries, refactoring of codified fields like date and URL, and so on.
  - **Standardization pipeline**, to make sure that fields marked as bound to a controlled vocabulary (via semantic annotation made during the dataset entity creation phase) are actually using the terms present in the vocabulary.

All pipelines are thought to enrich the incoming raw data, so not to modify the original content: the steps described above add new fields with the result of the transformation applied and, when applicable, an information about the "goodness" of the transformation applied.

- **Storage**: after the dataset is ready, it will be persisted using one or more storage mechanism indicated during the dataset entity creation phase. We are currently working to support parquet files in HDFS, HBase, Kudu, MongoDB and ElasticSearch.
- **E-gestion**: data are consumed by the end user via API (implemented within the *dataset manager* microservice), Spark (accessible via Livy and a Jupyter notebook) and data analytics & visualization application (currently we integrated with Metabase and Superset).

1.1.2 Types of datasets

From a logical point of view, the DAF manages two types of datasets:

- **Standard datasets** describe phenomena that are common nationwide. These datasets are thus defined as datasets with national relevance and supported by the highest level of information/metadata. They follow a detailed set of rules and standardization mechanisms that make them homogeneous across data sources (there may be multiple data sources describing the same phenomena, e.g., the bike sharing phenomena can be analyzed using data coming from Milan, Turin, Rome, etc…).
- **Ordinary datasets** have "owner" relevance, in the sense that they are defined and generated by a specific owner for its specific usage. They do not obey to a standard nationwide schema (data model), but the owner needs to specify metadata and information about the dataset before ingesting the data into the Big Data Platform of the DAF.

1.2 Interoperability, Standardization and Semantics

One of the biggest issues DAF wants to solve is interoperability/integrability between datasets. This is achieved by a combined use of a deep metadata architecture together with semantic tagging, controlled vocabularies and ingestion transformation procedures. At the ingestion time, every dataset get transformed and enriched so that the output can be easily "linked" to other datasets.

At the moment, we are working to cover the following standardization use cases:

- **Data format & conventions**: incoming data are treated to follow coherent internal conventions, e.g. date, URL format, address format, text reformatted to account for normalization vocabularies, etc.
- **Data enrichment**: adding new information to the incoming dataset in order to improve the informative value of the dataset and ensure internal interoperability, e.g. terms standardization using controlled vocabularies, addresses components, global unique id for rows, etc.

1.2.1 Metadata, Semantic Tagging and Controlled Vocabularies

Datasets are associated to concepts and attributes described into domain ontologies via semantic tagging (done during the ingestion step). This mechanism consists in linking columns/features of the datasets to corresponding ontologies. This is done to specify the concepts and the attributes the datasets contain.

> For example, let's consider a dataset containing the list of Italian companies and their headquarters. Here all the columns related to personal information (e.g. name, year of establishment, etc.) will be tagged with the corresponding semantic tag contained in the "Organizations ontology". Furthermore, the information related to the address of the headquarter will be tagged with the semantic tag of *the address of* from the "Places Ontology", and also associated to the concept of "company". In this way the address is the one of a company (as opposed, for example, to the address of a citizen). Finally, by having linked the address to the related concepts contained in the "Places ontology", we will be able to use all info contained in the mentioned ontology, included the existence of controlled vocabularies for the properties used.

The advantage of using ontologies and linking them to the ingested dataset are the following:

- easier establishment and implementation of a standard dataset at national level;
- possibility to logically link datasets based on their content and concepts. This feature is specifically useful in exploratory analysis and model munging;
- usage of the controlled vocabulary information so to provide standardization procedures to ensure the correct usage of the vocabulary in the dataset, and therefore, improve interoperability;
- possibility to build meaningful and rich knowledge graphs to be exploited for further analysis.

1.3 End-user features: Dataportal & API

DAF offers functionalities to its users (policy makers, analysts, data scientists, civic hackers, data journalists and informed citizens) via a web application (that we call Dataportal) and an ecosystem of APIs to access the data and other core functionalities. All DAF users will be exposed to almost the same functionalities, but they will be able to access only the datasets they have been granted access to.

1.3.1 Dataportal

The Dataportal is the DAF user interface, a place where users can have access to most of DAF functionalities and contents, based on their role and access grants. It is a web applications offering a *public area* where users can have access to contents that can be released publicly, and a *private area* where authenticated users can manage and use all data they have been granted access to based on their role and organization, and functionalities like data visualization, business intelligence and data science.

Here are some basic content and features accessed and managed via the dataportal:

- **Dataset metacatalog**: a catalog where to search for datasets, their info according to DCAPAP_IT profile, correlated datasets, related information and datastories made by using the dataset, the list of API with which using the dataset, and other operational info useful to use the dataset with the integrated tools.
- **Data stories**: blog posts that integrate data visualizations and scripts to tell the whys and the results of a data analysis. Data stories are design to stimulate discussion around an data related themes, and allows the community to comment an propose further analysis.
- **Data applications**: software with a user interfaces that expose functionalities based on data and models trained on them. Data applications can be done by the DAF team, public administration or can be proposed by the community, and will be organized in a related section of the dataportal.

- **News & blog**: blog post containing news related to the data world, together with how-to and tutorial on how to use the DAF and dataportal.
- **Data visualization & Business Intelligence**: we integrated two open source tools, Metabase and Superset, where user can explore data managed by DAF and create graphs and dashboards.
- **Data science**: a JupyterHub notebook has been integrated and connected to the Hadoop cluster to allow for data manipulation, analysis and data science. Users can take advantage of the capability of Apache Spark and its libraries to run SQL queries on top of Hadoop, integrate different datasets and training machine learning models using MLlib library.

1.3.2 API

The DAF exposes automatically an API to interact with datasets. Users can use standardized interfaces to access the datasets to:

- get statistical and metadata info;
- download the dataset (up to a certain limits);
- run SQL like queries on datasets.

The use of APIs will allow users to build their applications easily and with the guarantee to access the most updated data available.

1.4 Open Data in SaaS

The DAF manages the national open data catalog via its dataportal, by centrally integrating all metadata information of the open dataset provided by the Public Administration, and rearrange them using the DCATAP_IT profile. Furthermore, it also offers a Open Data as a Service tool: Public Administrations may choose to manage their open data using DAF functionalities and not having to take care of yet another open data catalog by themselves. They will have an always updated platform, and can provide to their users all current and new functionalities developed centrally for the dataportal.

The advantages of using DAF to manage Open Data can be summarized as follows:

- Open data managed directly via DAF will be DCATAP_IT compliant by design, and will follow its evolution once and for every PAs.
- Open Data that can be derived directly from PAs datasets managed into DAF will be automatically created and updated as new information is ingested.
- The usage of semantic tagging will allow for the creation of RDF and linked open data, exposed also via SPARQL endpoints.
- Open Data Website as a Service: the PAs can choose to use use DAF to manage their open data website with a simple console that will allow them to customize the look and feel, and have a new website on the fly.

CHAPTER 2

Data Management

This section is about dataset concept and management in DAF. You'll find info about the type of datasets managed, the logics behind datasets in DAF, ingestion pipelines and conventions used and much more.

2.1 More on Datasets: A Deep dive

The notion of datasets is the main concept around which DAF architecture has been build. In fact, DAF provides advanced features for data governance, analysis and interoperability designed to solve typical problems faced by the public administrations and large companies. In this section we expand on what has been presented in the Overview section, to better describe how datasets are managed in DAF. What will follow is common to all datasets managed in DAF, except for some aspects related to data streams, that will be covered into an ad-hoc section.

Formally, we define a dataset *as a combination of data and metadata.*

- **Data** is the actual content of the dataset, and can be organized into tabular, json and text formats.
- **Metadata** are all the information about the dataset that describe and give context to its contect. Metadata will be treated in details in an appropriate section.

2.1.1 Ordinary Dataset

Every datasets that are ingested by an organization (PA) are treated as **ordinary dataset**. An ordinary dataset does not have a pre-defined structure it needs to follow, it is ingested as it comes, besides the standardization, normalization and enrichment processes defined in the ingestion pipeline.

It will have the following data structure:

- standardized & normalized original columns
- raw original columns
- enrichment columns, among which (most of them will depend on the content of the original columns):
  - `__ROWID`: global unique row id

   - `__dtcreated`: date and time when the info has been added
   - `__dtupdated`: date and time when the info has been updated

Ordinary datasets can be created from data coming from outside DAF, or by transformations applied to already existing DAF datasets. For example, a public administration can decide to create an open data version of a private dataset, by indicating the columns that can be released publicly and an optional aggregation policy. In this way, everytime the private dataset receives updates, they will be automatically be reflected into its open data version (that will be another dataset in the DAF world).

Storage & other conventions

Ordinary datasets are stored by default in HDFS using the Parquet data format and exposed as Hive and Impala tables. Data will be physically stored in the following HDFS meta-directory:

`/daf/ordinary/{organization}/{domain}_{subdomain}/{dataset name}/{version}`

where:

- `{organization}` is the name of the organization owner of the dataset, e.g. 'Comune_Milano'
- `{domain}` and `{subdomain}` are the code of the categories used to group the data. They are mutuated from DCATAP_IT.
- `{dataset name}` is the unique identifier for the dataset
- `{version}` is the name of the version of the dataset that is stored into that folder. It can typically be `landing` for the raw data ingested, `final` for the output of the ingestion pipeline. The final version will be the one exposed via API by DAF.

As convention, DAF manages a system of logical uri by which a dataset can be uniquely identified. In the case of ordinary dataset, it is built as follows:

`daf://dataset/ordinary/{organization}/{domain}/{subdomain}/{dataset name}`

2.1.2 Standard Datasets

Standard Datasets are those datasets that describe concepts and phenomena valid nationwide, following a strict data structure and semantics rules. Standard datasets are said to follow 'standards' that are defined for all PAs, so to guarantee that a given phenomena can be described in the same way and following the same conventions nationwide. Data will be physically stored in the following HDFS meta-directory:

They will typically have the following data structure:

- standardized and normalized mandatory columns, grouped under the `mand.{colname}` struct field
- standardized and normalized optional columns, grouped under the `opt.{colname}` struct field
- enrichment columns from the ingestion procedures, grouped under the `enr.{colname}` struct field
- operational enrichment columns, with info needed for internal DAF operations and grouped under the `ops.{colname}` struct fields. These fields are:
   - `__dtcreated`: date time when the info has been ingested into the standard dataset
   - `__dtupdated`: date time when the info has been updated
   - `__srcorg`: the code name of the organization that originated the data (e.g. 'Comune_Milano')
   - `__dsname`: the unique name of the ordinary dataset from which the data come from.

Dataset standards can be created directly by a PA, if it is the only contributor to the national standard, or by 2 or more PAs in cases when every PA contribute to the national standard with the piece of info it manages. In the latter case, the standard dataset is created starting from 2 or more ordinary datasets, and in this case can be considered a 'derived' dataset. In this case, the ordinary dataset will specify that they contribute to a standard and the mapping model needed to map the ordinary dataset into the standard one.

Storage & other conventions

Standard datasets are stored by default in HDFS in parquet format and exposed as Hive and Impala tables. Data will be physically stored in the following HDFS directory:

`/daf/standard/{domain}__{subdomain}/{dataset name}/{version}`

If not specified differently, it is partitioned based on `__srcorg` and, in case the dataset is of type 'last update' (meaning it is composed by append of snapshot updates, as opposed to 'time series' type), it is also partitioned by `__dtcreated`

2.1.3 Raw Open Data

Finally, we collect and store raw open data coming from the national catalog. They are automatically metadated based on the info available (in 'at best' fashion), and stored in HDFS in the following path:

`/daf/opendata/{organization}/{dataset name}`

Once ingested, they can be used with all other tools managed by DAF.

2.2 Dataset MetaCatalog: a deep dive into metadata for datasets

The MetaCatalog, managed by the Catalog Manager microservice (see Architecture Section), are detailed info describing a dataset. They are data about data, therefore metadata. Metadata are heavily used in DAF to help discoverability of dataset, allows for multi-system interoperability, perform internal automatic operations. According to their function, metadata are divided into 3 macro categories: dataset level (DCATAPIT), data structure level, operational level metadata.

2.2.1 Dataset level metadata (DCATAP_IT)

These metadata are meant to describe the dataset's info such as its name, owner, category, etc. DAF implemented the mandatory fields of the DCATAP_IT profile, according to AGID regulations. Following a subset of the DCATAPIT metadata (see the link below for a complete list of required info)

- `name`: dataset unique name
- `title`: title of the dataset
- `identifier`: dataset unique identifier
- `alternate_identifier`:...
- `author`: ...
- `theme`: thematic domain that characterize the dataset
- `license_id`:...
- `resources`: a list of resources from where to download the dataset and other related data. Dataset managed in DAF will have here API endpoints to download and access the dataset.

- `license_title`:
- `frequency`:
- `publisher_name`: name of the organization that publish the dataset
- `publisher_identifier`. . . .
- `organization`:. . . .
- `owner_name`: name of the organization that owns the dataset
- `holder_name`:. . .
- `holder_identifier`:. . .
- `tags`: tagging system connected to vocabulary.
- `relationshiops_as_subject`:. . .
- `notes`: additional information
- `modified`: date of last modification to the dataset

2.2.2 Data structure level metadata

Data scructure metadata contains information about the internal structure of a dataset, at column or field level. It manages info about format, content, semantics that are contained into a single column of a tabular dataset, for example. Here we store two datastructure metadata: an avro schema of the dataset, and a 'flatschema' where we store additional information than the one provided by the avro schema, specifically thought to enrich the information expressed about the content and the semantics of the columns. Below, you'll find the list of info contained in the 'flatschema' part, we'll skip the avro schema part as are using the standard avro schema definition.

- `name`: name of the field/column. This name needs to obey to formatting rules that are necessary for it to be used as the name of a column in a database, therefore it may not be human readable.
- `type`: data format of the column, such as 'string'.
- `metadata.title`: human readable name for the column.
- `metadata.desc`: description of the content of the column.
- `metadata.field_type`: it tells if the column is a dimension, a metric (numeric attribute) or a descriptive attribute.
- `metadata.required`: it tells if the field is mandatory or optional.
- `metadata.uniq_dim`: checked if the column is part of the list of dimensions that make the row unique, such that there will not be two rows with the same values for the columns checked as `uniq_dim`.
- `metadata.is_createdate`: boolean, checked if the column contains the date when the row was created.
- `metadata.is_updatedate`: boolean, checked if the column contains the date when the row was updated.
- `metadata.cat`: category that can better represent the content of the field. This is controlled by a vocabulary.
- `metadata.tag`: list of tags that can better represent the content of the field. All tags are saved into an evolving vocabulary.
- `metadata.constr`: a list of objects (made by `type` and `param` arguments) to set contraints on the content of the field.
- `metadata.semantics`: an object containing semantic info to link hte column to related ontology and controlled vocabulary, if any. In particular:

  - `id`: this is the semantic tag that links the column with a given attribute of a concept described into an ontology.
  - `context`: this info gives context info on the semantic tag.
  - `rdf_subject`: it is used to give a better context to the info contained in the column. Technically, it is a tag for a concept described into an ontology. In most cases, it can be seen as the subject that makes an action, derived from the id attribute.
  - `rdf_predicate`: the action that the subject perform on the content of the column.
  - `rdf_object`: the target of the action performed by the subject.
  - `uri_voc`: It is a unique identifier for the vocabulary. It matches with the `dsname` field of the dataset in DAF.
  - `uri_property`: it is the uri associated to the element epressed in the column. It is used, among other things, to link the column of the dataset to the column of the vocabulary which controls it, if any.
  - `property_hierarchy`: it is of type array, and it gives info about the hierarchy, if any, to which the property/column belongs to.
- `metadata.personal`: this objects contains info whether the data are of personal kind. In particular:
  - `ispersonal`: boolean, to tell whether or not the info contained in the column is a personaltype of information.
  - `cat`: category of personal information
- `metadata.format_std`: this is an object that gives info about a format standard the data follows when ingested in DAF. It is useful to help the system identify such standard and transform into the DAF choosen standard. The object has the following two attributes:
  - `name`: name of the format standard, e.g. date, credit card, address, etc.
  - `param`: depending on the type of format, it is a string giving the exact formatting order and composition of the information contained. E.g. for the date example, it may be 'YY/MM/DD'. This will help the normalization procedure to refactor in the right order and format the information, such to follow the DAF internal conventions.
- `metadata.field`: it has info on indexing in SearchEngine and profiling of the field, plus other Kylo specific information on standard and validation.
  - `is_index`: it tells to create an index based on this field in the SearchEngine.
  - `is_profile`: it tells to create a profile for the field that will be displayed as result of the SearchEngine.
  - `validation`: contains info on the validation rules to be used for the field.
  - `standardization`: contains info on the standardization procedure to be performed on the field.

  Operational level metadata

These metadata are used to manage the dataset within DAF logics and conventions, from input sources to storage options to ingesiton pipelines mechanics.

- `inactive`: optional boolean, true if the dataset entry has been created as inactive (that is, no effects on the system has been created, e.g. no ingestion pipeline has been started for the dataset yet).
- `theme`:..
- `subtheme`:...
- `logical_uri`:

- `physical_uri`:..
- `is_std`:..
- `group_own`:...
- `group_access`: (name, role)
- `std_schema`:...
- `georef`:..
- `input_src`: It is an object containing information about input feeds, which can be of the following types:
  - `sftp`: it contains info on how to access the sftp plus specific info on the feed characteristics (i.e. data format and related options) * `name` * `url` * `username` * `password` * `param`: this is a json string containing info related to the specific input type not already codified. An important info contained here is the type of file to be ingested (e.g. csv, json, xml) and option related to the file format.
  - `srv_pull`: , srv_push, daf_dataset. Each of them,
- `ingestion_pipeline`:..
- `storage_info`: (hdfs, kudu, hbase, mongo, textdb)
- `dataset_proc`: It has info about how to process and store internally the dataset. Such info includes partitioning, merge strategy, etc.
  - `read_type`: update vs timeseries
  - `dataset_type`: batch vs stream
  - `partitions`: It contains info on how the dataset is partitioned in DAF.
    - `name`: name of the partition, given by the user.
    - `field`: name of the field to be used for partitioning. It must correspond to one of the 'name' of the dataschema.
    - `formula`: the formula to be applied to the field to get the partition value.
  - `merge_strategy`: It tells how new data should be ingested into the existing dataset. User must choose among the following options. 'SYNC' to replace the existing content with the new one; 'MERGE' to append the data into the target partitions; 'DEDUPE_AND_MERGE' to insert into the target partition but ensure no duplicate rows are remaining; 'PK_MERGE' to insert or update existing rows matching the same primary key; 'ROLLING_SYNC' to overwrite target partitions only when present in source.
- `opendata`: it is used to tell the system to create an open data version of the dataset. If valued, it will create a new derived dataset entry, precompiled with info taken from the original dataset, and put into 'inactive' state so it can be valued and confirmed by the user. It is an object with the following info: * `create_opendata`: boolean, valued as true if user wants to create a derived open data dataset. * `sql`: SQL query with the final data structure of the open data dataset.
- `service_layer`: it is used to put the dataset (or its transformation) into the service layer (Kudu?) * `transfer_mode`: it tells whether the dataset will be put as is in the service layer or it needs to be transformed via derived dataset. It takes two values: `direct`, `derived`. * `sql`: optional, used in case `transfer_mode` is valued at `derived` and user wants to specify ex-ante the transformation query.

2.3 DAF Dataset Conventions & Ingestion Pipeline

The logic behind DAF highly relies on internal conventions when it comes to dataset ingestion, e-gestion and usage. These conventions aims to improve the interoperability between datasets and to make possible the enrichment of the original data with other useful info DAF already knows. Below, you find a list of conventions used, the ingestion steps, and an example that shows how these conventions are used in practice.

2.3.1 DAF Conventions List

- **Data format** (Normalization): all data is transformed into UFT-8 format
- **Data conventions** (Normalization): we make use of the following internal conventions
  - *null value* are saved as `NULL`. Therefore, all other external conventions used in the raw incoming data (i.e. empty string) are transformed to `NULL`
  - *date* are transformed using the ISO8601 in `YYYY-MM-DD hh:mm:ss`
  - *url* are formatted as following `http[s]://www.yoururl.com`
  - *normalization vocabularies* are used to normalize special cases that will be continuously added to ad-hoc vocabularies, such as accented letters, specific names, etc. The terms in the vocabularies are substituted with the appropriate translation, and this will be done both for all dataset (general normalization vocabularies), or for owner/dataset (source normalization vocabularies and dataset normalization vocabularies).
  - *address* written in a single cell, is interpreted (when possible) and rearranged as follows: *{address type} {address name}, {civic number}, {zip code} {City} ({Provincia/State}), Country*. The system recognize it as address, via semantic tag.
- *new columns from enrichment* are named as following: `__{enrichment type}[_{column name}]`, where text in `{}` is mandatory, and text in `[]` is optional and depends on the type of transformation (some are not connected to an existing column)
  - a *unique row id* (Enrichment) is added via a new column called `__ROWID` to ensure global uniqueness of the id within DAF. It is made as following: `{dataset name space}_{row hash}` [TBD]
  - a *generated datetime* (Enrichment) is added with the timestamp of the ingestion in a column called `__dtcreated`
  - a *updated datetime* (Enrichment) is added with the timestamp of the last update for that row in a column called `__dtupdate`
  - a *controlled vocabulary standardization* (Standardization) columns is added every time there is a feature/column that is tagged to have a controlled vocabulary associated (the tagging happens in the Catalog Manager, at metadata level). In this case, a procedure is activated to check whether the values contained in the column obey to the vocabulary. This will result in the addition of two columns: `__std_{col name}` will have the value of the original column in case they are found in the vocabulary, or the closest value (according to some distance metrics like the Levenshtein distance) in case the value is not exactly found; `__stdtat_{col name}` containing the distance between the original value and the closest one found in the vocabulary. Be aware that the final dataset will not contain the `__std_{col name}` column, as its value will be put in a column named with the original name `{col name}`. The `__std_{col name}` column is used in intermediate steps of the ingestion process.
  - a *unique identifier / code* (Enrichment) for the standardized columns, in case the controlled vocabulary has a `code` associated to the standard label. This column will be called `__stdcode_{column name}`
  - *address components* (Enrichment): every time there is a column/field tagged as address, this enrichment step will add the following columns inferred from it:
    - `__address@placetype_{column name}`: it contains the type of address (i.e. via, piazza, viale, ecc.) contained in the address from the column `{column name}`
    - `__address@placename_{column name}`: it contains the name of the address (i.e. 'del Corso') contained in the address from the column `{column name}`

      * `__address@placecode_{column name}`: it contains the code (unique identifier) of the address from its controlled vocabulary.
      * `__address@cityname_{column name}`: it contains the name of the city of the address from its controlled vocabulary.
      * `__address@citycode_{column name}`: it contains the code (unique identifier) of the city from its controlled vocabulary.
      * `__address@provname_{column name}`: it contains the name of the 'provincia' or state of the address from its controlled vocabulary.
      * `__address@provcode_{column name}`: it contains the code (unique identifier) of the 'provincia' or state from its controlled vocabulary.
      * `__address@countryname_{column name}`: it contains the name of the country of the address from its controlled vocabulary.
      * `__address@countrycode_{column name}`: it contains the code (unique identifier) of the country from its controlled vocabulary.
      * `__address@lat_{column name}`: it contains the latitude of the address.
      * `__address@lon_{column name}`: it contains the longitude of the address.

2.3.2 Ingestion pipeline steps

The following steps describe the ingestion pipeline applying the above conventions to the incoming new data.

1. New data are captured from the incoming source and stored into a landing area into DAF HDFS.
2. **Normalization procedures** are applied to the incoming data so to apply DAF conventions. The result will be a dataset with new features/columns added named `__norm_{col name}`, containing the result of the normalization applied to all features/columns of the original dataset.
3. **Standardization procedures** are applied to the normalized columns of the previous step. The result will be a dataset generated a the previous step, with the addition of the standardized columns (only when the standardization can be applied, so the number of added columns may be less then the total number of columns in the original dataset (step 1.) named `__std_{col name}`.
4. **Enrichment procedures** are applied to the dataset at step 3, resulting in a new dataset build starting from the one generated in step 3 and adding enrichment columns named as follows `__{enrichment tuype}[_{col name}]`.
5. **Finalization procedure** takes as input the dataset in step 4 and rearrange its columns as follows: it contains the normalized and standardized columns, named as the original columns in the incoming dataset (step 1); the original raw data columns named `__raw_{col name}`, the list of enrichment columns from step 4.

2.3.3 Final Dataset Structure

Based on the conventions and ingestion pipelines described above, a final dataset will have the following structure:

- **'ROWID'** column, with a unique identifier for the row.
- **List of dataset columns**: the list of columns of the original raw dataset ingested, at which all the procedures described above have been applied. These columns are named with the original column names provided at the ingestion time.
- **List of original raw columns**: this part reproduce the original data as provided for the ingestion. These columns will be named as follows: *__raw_{col name}*.

- **List of enrichment columns**: those are columns that add additional information to the dataset, extracted at ingestion time. They will be named according to the following convention: *__{enrichment type}[_{col name}]*.

As an example, consider the following dataset to be ingested into DAF.

Step 1: Ingest raw incoming data into DAF, after creation of dataset instance in the catalog manager.

id	company_name	industry_type	address	city	state	num_employees	website	description	year_foundation
1	Fiat	Automobili	via Gabriele Chiabrera, 20, 10126, Torino	Torino	Italia	1000	fiat.it	Fiat e’ stata fondata nella citta’ di torino	“”

Among other things, let’s suppose that the following metadata has been associated to the dataset:

- *industry_type* has been associated with the ATECO contolled vocabulary
- *address* has been linked to the semantic tag associated to address, and to data type ‘address’
- *city* has been linked to the semantic tag associated to the city and connected controlled vocabulary
- *state* has been linked to the semantic tag associated to the state and connected controlled vocabulary
- *website* has been associated to the data type *url*

Step 2: Apply normalization procedures

id	company_name	industry_type	address	city	state	num_employees	website	description	year_foundation	__norm_id	__norm_company_name	__norm_industry_type	__norm_address	__norm_city	__norm_state	__norm_num_employees	__norm_website	__norm_description	__norm_year_foundation
1	Fiat	Automobili	via Gabriele Chiabrera, 20, 10126, Torino	Torino	Italia	1000	fiat.it	Fiat e’stata fondata nella citta’ di torino	“”	1	Fiat	Automobili	via Gabriele Chiambrera, 20, 10126, Torino	Torino	Italia	1000	http://www.fiat.it	Fiat è stata fondata nella città di Torino	NULL

Step 3: Apply standardization procedures

id	company	industry	address	city	state	num_employees	website	description	year_foundation	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	stat_industry	[illegible]	[illegible]	[illegible]	stat_state
1	Fiat	Automobili	via Gabriele Chiabrera, 20, 10126, Torino	Torino	Italia	1000	fiat.it	Fiat "" e'stata fondata nella citta' di torino		1	Fiat	Automobili	via Gabriele Chiambrera, 20, 10126, Torino	Torino	Italai	1000	http://www.fiat.it	Fiat è stata fondata nella città di Torino	NULL	Fabbricazione di Autoveicoli	45	Torino	0	Italia	1

Step 4: Apply enrichment procedures

id	company	industry	address	city	state	num_employees	website	description	year_foundation	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	stat_industry	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
1	Fiat	Automobili	via Gabriele Chiabrera, 20, 10126, Torino	torino	Italai	1000	fiat.it	Fiat "" e'stata fondata nella citta' di torino		1	Fiat	Automobili	via Gabriele Chiambrera, 20, 10126, Torino	Torino	Italai	1000	http://www.fiat.it	Fiat è stata fondata nella città di Torino	NULL	Fabbricazione di Autoveicoli	45	Torino	0	Italia	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	Gabriele Chiambrera	[illegible]	[illegible]	[illegible]	[illegible] 'Torino'}	[illegible]	[illegible]

Step 5: Finalization

RAW DATA | FINALIZED DATA | ENRICHMENT COLUMNS |

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	company	industry_type	address	city	state	num_employees	website	description	year_foundation	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
1	Fiat	Automobili	via Gabriele Chiabrera, 20, 10126, Torino	torino	Italai	1000	fiat.it	Fiat "" e'stata fondata nella citta' di torino		1	Fiat	Fabbricazione di Autoveicoli	via Gabriele Chiambrera, 20, 10126, Torino	Torino	Italia	1000	http://www.fiat.it	Fiat è stata fondata nella città di Torino	NULL	45	0	1	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	Gabriele Chiambrera	[illegible]	[illegible]	[illegible] 'Torino'}	[illegible]	[illegible]	[illegible]	[illegible]

2.4 Storage Engines

TBD

2.5 Security & Permissions

TBD

CHAPTER 3

Data & Analytics Framework Architecture

[TBD] The DAF Big Data platform is an environment offering capabilities for:

- *storing and managing datasets*: users can register and load datasets on the platform, specifying the ingestion model (e.g batch, streaming), the serialization formats (e.g. Avro, Parquet), the desired serving layers (e.g. HBase, Impala), metadata, etc;
- *processing and analysing datasets*: the platform supports several Hadoop-based technologies. Users can not directly use these technologies, since they are mediated by user-friendly applications provided by the Dataportal (e.g. Superset, Jupyter);
- *managing of access rights for each dataset*: the adopted security approach allows the platform administrators to set the proper access rights for each dataset.

The DAF Big Data platform also enables *redistributing datasets, developing data applications, publishing insights* by means of the above mentioned tools provided by the Dataportal: by these tools, data scientists and analysts can perform analysis on data, run statistical and machine learning models, and produce data visualizations and reports.

For more information, continue your tour with the following sections.

3.1 Big Data platform Architecture

The DAF Big Data platform has been originally designed to gather and store data coming from different Italian Public Administrations. As a consequence, it provides efficient and easy to use ingestion mechanisms for allowing external organisations to simply ingest their data into the platform with minimal human intervention.

The DAF platform should not only provide support for data at rest and fast data (streaming), but also for storing and managing collections of unstructured data, textual documents. Besides providing those storing capabilities, the next main goal is to provide a powerful mechanism for data integration, i.e. a way for integrating data that traditionally reside on separate silos. Enabling the correlation of datasets normally residing on different systems/organizations can become a very powerful enabling factor for discovering new insights on the data. The platform should allow the data scientists to access its computational power for implementing advanced analytics algorithms.

The Big Data architecture underlying the DAF is described by the following views:

3.1.1 Logical View

The DAF platform is ultimately an implementation of the "data lake" concept. Assembling a data lake involves a sequence of unavoidable steps meant to gather, organise and publish the data in an efficient and secure way.

The most important aspect to take into account in a data lake being set up is the data governance. Data governance means data organizations and metadata management. Being able to catalog the datasets together with their metadata is the prerequisite for implementing any further steps in the data lake set up such as data ingestion/egestion and data security.

Implementing the Dataset Abstraction

The main abstraction the DAF platform is based upon is the dataset. From a technical point of view, a dataset is a collection of records described by a data schema. A dataset is identified by a *logical URI* and it is associated to a *physical URI* that identifies the medium and location where the data is actually stored.

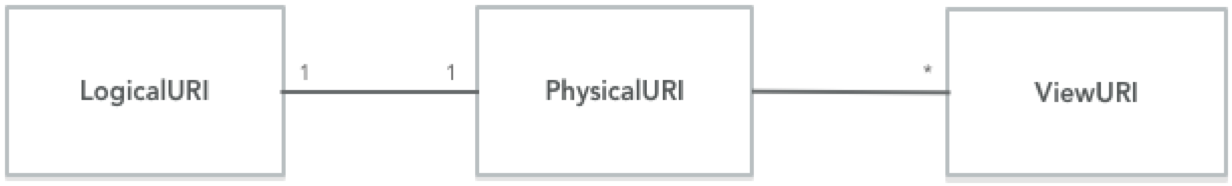


Fig. 1: URIs relationships

A *LogicalURI* must be associated to one and only one *PhysicalURI* that can be associated to zero or more *ViewURIs*. Let's explain this with an example.

Let's define a *LogicalURI*, for example:

`daf://ordinary/comune_milano/mobilita/sharing/bike`

this can be bound to the following *PhysicalURI*

`dataset:hdfs:/daf/ordinary/comune_milano/mobilita/sharing/bike`

and eventually to a *ViewURI* like

`dataset:hive://comune_milano/mobilita/sharing/bike`

In other words, while a *PhysicalUri* represents the actual location on the Hadoop storage behind, a *ViewURI* represents the fact that a dataset can be also exposed/view through a different platform.

As an example, a Hive/Impala external table created on top of a directory on HDFS represents a view of the same data stored in HDFS. This approach should allow modeling the mechanism of publishing datasets with low latency SQL engines like Impala/Presto.

All the metadata about datasets including their URIs are collected and organised in a catalog. This catalog is an essential component of the DAF platform: all the data ingestion steps and all the data manipulations' steps that we allow on the data will be driven by it.

DAF Big Data Architecture Layers

The high-level view of the architecture is pretty simple. It is based on the following layers:

- -*Service Layer*: it contains all the services needed to implement the platform functionalities. It also contains the catalog manager -service (*CatalogManager*) which is responsible to manage all the datasets metadata.

- *Ingestion Layer*: it is responsible for all the ingestion tasks. It is be based on tools for data ingestion automation like NiFi. It's strongly integrated with the *CatalogManager* because, as already said, all the incoming data is listed in the catalog: this implies all the ingestion supporting tools is integrated with the *CatalogManager*.
- *Hadoop Computational Layer*: it contains all the typical computational platforms part of the extended Hadoop stack. The most important platform which is going to be used extensively by the platform is Spark. The -service (in the -service layer) uses the computational layer for tasks like data access and data manipulation/transformation. The ingestion layer uses the computational layer for implementing tasks like data conversion/transformation.
- *Hadoop Storage Layer*: it contains all the storage platform provided by Hadoop: HDFS, Kudu and HBase. As described above the physical URIs contain the information for accessing the data as stored on those storage platforms.

The following image summarizes the logical view of the DAF architecture:

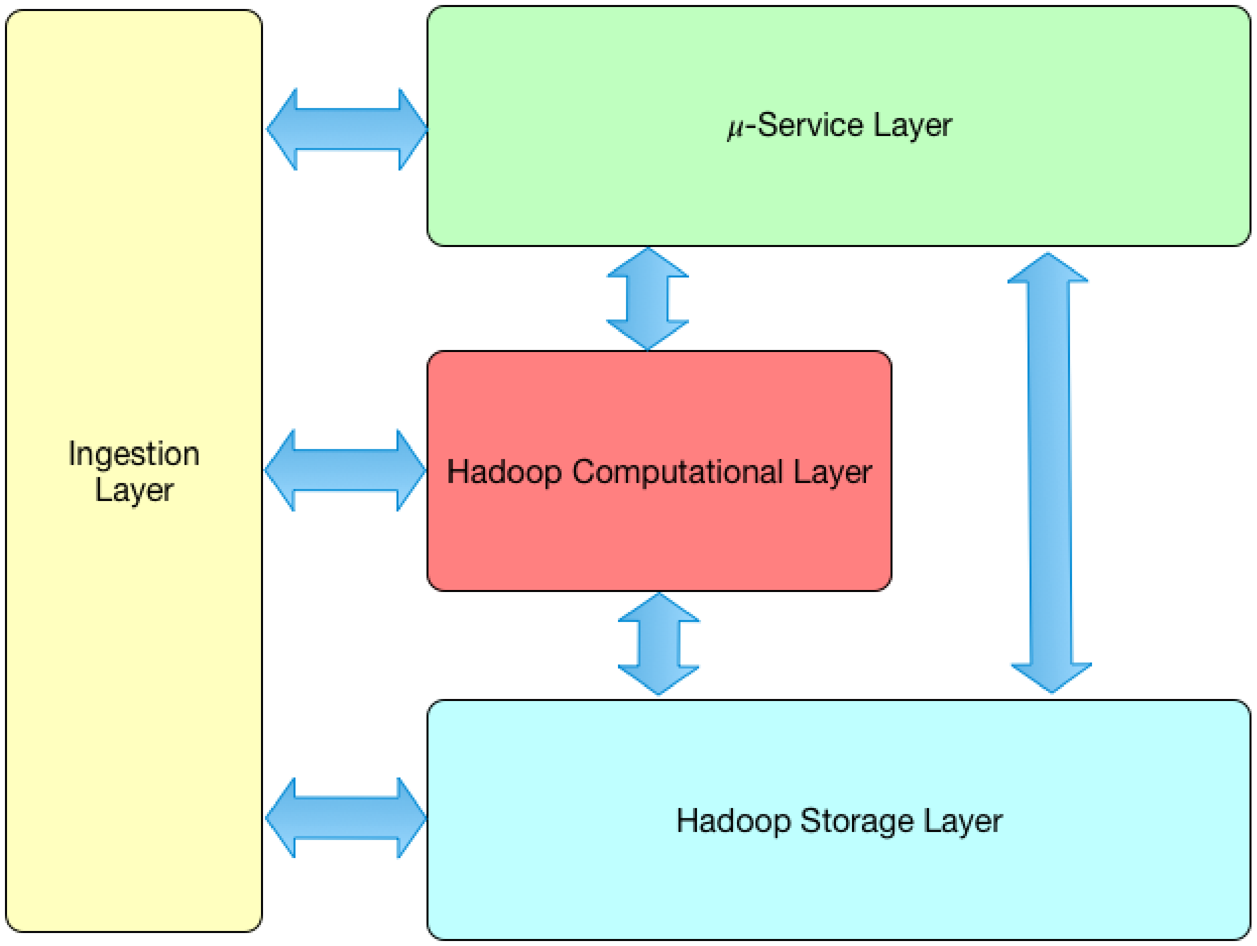


Fig. 2: Logical View

3.1.2 Component/-Service View

The main components/-services of the DAF platform are:

- *CatalogManager*
- *IngestionManager*

- *StorageManager*
- *DatasetManager*

The following image shows these components/-services and their mutual relationships.

CatalogManager

The *CatalogManager* is responsible for the creation, update and deletion of datasets in DAF. Furthermore, it takes care of the metadata information associated to a dataset.

The CatalogManager provides a common view and a common set of APIs for operating on datasets and on all related metadata information and schemas (see the CatalogManager API & endpoints).

The CatalogManager is based on the services provided by the CKAN service. In fact, one of the most relevant architectural decisions is to reuse as much as possible the metadata and catalog features provided by the CKAN service. The idea behind is simple: treating the data managed by the DAF platform similarly to what CKAN does with the open data. Part of the metadata are managed by the CKAN catalog and additional metadata information are managed by the CatalogManager.

The CatalogManager is also responsible to store all the schemas associated to the datasets: these schemas are saved as AVRO schemas.

IngestionManager

The *IngestionManager* manages all the data ingestion activities associated to datasets.

The IngestionManager collaborates with the CatalogManager to associate the proper metadata to the ingested data.

The IngestionManager provides an API to ingest data from a datasource into the DAF platfom (see the IngestionManager API & endpoints). In particular, the IngestionManager takes as input data and info needed to identify the dataset to which the data needs to be associated with. Before actually storing the data in DAF, the IngestionManager performs a set of coherence checks between the metadata contained in the catalogue and the data schema implied in the input data. There are two scenarios:

1. The catalog entry for the dataset has been already set up. In this case the *IngestionManager* will check if the incoming data and schemas are congruent with what has been configured in the catalog.
2. There is no catalog entry for the dataset. In this case the *IngestionManager* will automatically create an entry in the catalog checking that all the relevant information are provided during the ingestion phase.

The *IngestionManager* is also responsible for scheduling the ingestion tasks based on the information associated to the datasets. The ingestion for static data (data at rest) is based on a pull model. The dataset catalog entry should contain information about where and when the data should be pulled from.

StorageManager

The *StorageManager* is responsible for abstracting the physical medium where the data is actually stored (see the StorageManager API & endpoints).

The StorageManager is based on the Spark dataset abstraction for hiding the details of the specific storage platform. In fact, Spark provides a very powerful mechanism for describing a dataset source regardless of its actual physical place. We leverage this powerful mechanism for defining the physical URIs as described before, that is:

- `dataset:hdfs://` for HDFS,
- `dataset:kudu:dbname:tablename` for Kudu,
- `dataset:hbase:dbname:tablename` for Hbase.

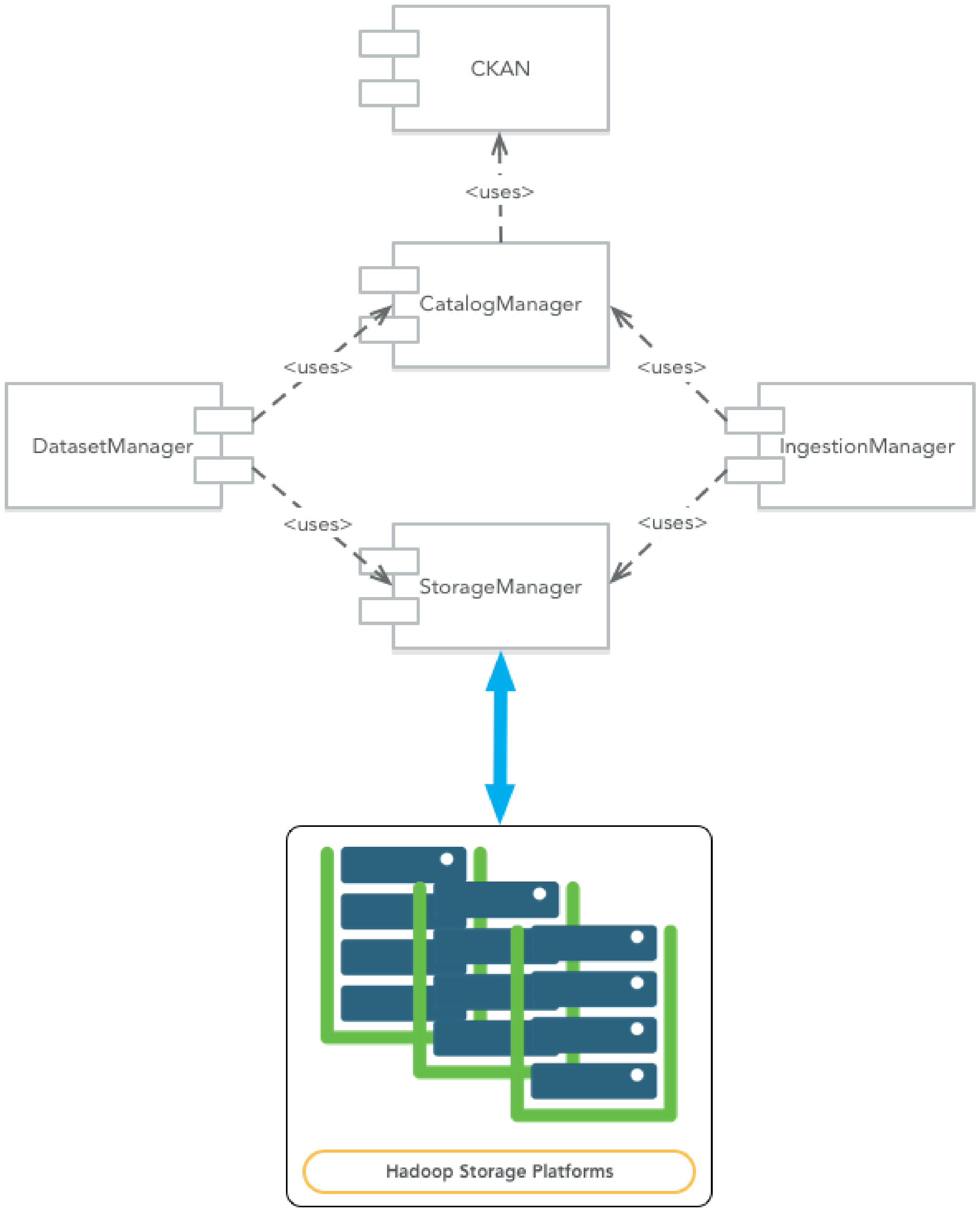


Fig. 3: Component View

The only restriction we have to impose for making this Spark based mechanism working is to always have a dataset per HDFS directory.

DatasetManager

The DatasetManager manages operations several related to the dataset, such as:

- to return the data of the dataset (or a sample of it) in a specified format:
- to create a specific view on top of a dataset,
- to get the dataset schema in a given format (e.g. AVRO);
- to create a new dataset based on an existing one but saved into a different storage mechanism or based on a transformation of the existing dataset, etc.

For a list of endpoints and functionalities currently provided by the DatasetManager see the DatasetManager API & endpoints.

Technically speaking, the DatasetManager is responsible for all the tasks on top of the datasets, indicated by the logical URIs. For example tasks like format conversion, AVRO to Parquet, dataset import/movement, from HDFS to Kudu will be managed by this -service.

The DatasetManager will interact with the CatalogManager for updating the information about the dataset is interacting with. For example, a format conversion means triggering a Spark job that creates first a copy of the source dataset in the target format. Then the catalog dataset is updated for taking into account the new dataset format.

The DatasetManager is also responsible for publishing the dataset into a proper serving layer. For example, a dataset operation could create an Impala external mapped on the dataset directory sitting on HDFS. This publishing operation will provide the user with the JDBC/ODBC connection informations for connecting an external tool to that table.

3.1.3 Deployment View

The DAF platform is designed to be deployed on two disjoint clusters of machines, as shown in the next figure:

1. *Kubernetes Cluster* - this cluster is composed by nodes with the role of edge nodes from the Hadoop cluster standpoint. The edge nodes are configured to have access to all the Hadoop platforms as client. Moreover, these nodes are hosting a kubernetes cluster where all the -services will be deployed. Being deployed on nodes that are also Hadoop edge nodes provides the -services with the capabilities to interact with Hadoop out of the box.
2. *Hadoop Cluster* - this is the cluster of machines where Hadoop has been deployed.

From a deployment perspective, other essential points regard the integration with:

- an Identity Management System, in order to centralize the user account management and to enable the implementation of all security issues;
- tools supporting the access, the manipulation and the analysis of datasets.

IMS integration

An important piece is the integration with an external Identity Management System (currently a FreeIpa instance). All the information regarding users and user groups willing to access the platform are centrally listed on this system. This is the base for implementing all the authentication and authorization mechanisms the DAF platform will require for securing the data access.

Any user that will access the platform shall be registered in the Identity Management System and any access to the data will be tracked allowing the auditing of data accesses for security purposes.

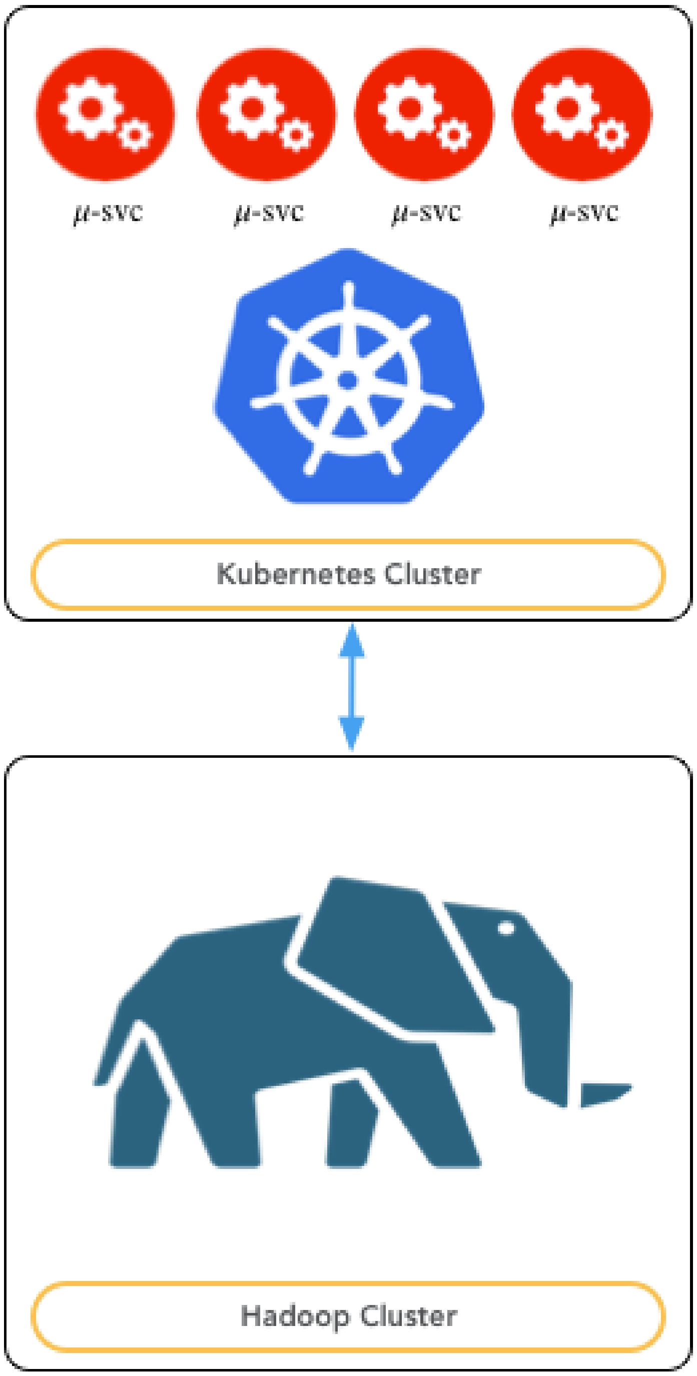


Fig. 4: Deployment View

As shown in the following figure, both the Kubernetes Cluster and the Hadoop Cluster refer to the same IMS. Consequently, it is possible to map user accounts created on the two cluster, improving the security of the entire system.

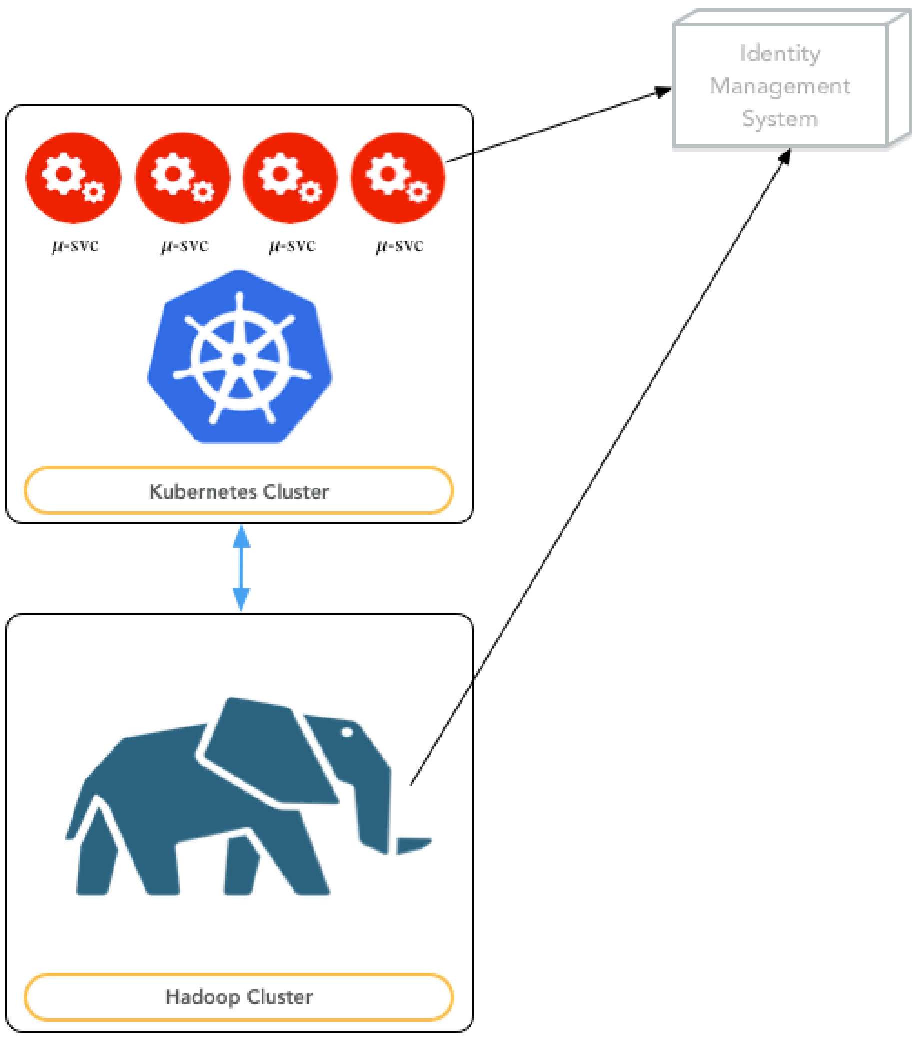


Fig. 5: Deployment View

Notebook support

The platform will support the usage of notebooks for accessing and manipulating the data. The platform will provide access to the Hadoop computational resources through proper services that avoid the user to access the Hadoop cluster directly.

A possible approach could be the combination of a REST service like livy with a tool like Sparkmagic for giving access from a Jupyter notebook to Spark.

The platform will provide special libraries for directly accessing the data sets from the notebooks just by knowing their URIs.

3.2 DAF - Security & Privacy Issues

3.2.1 Data at rest security policies

The default storage platform is HDFS, so once the data has been put on HDFS must be protected using the proper permissions. Moreover, since the data need to be accessible through Impala, a set of proper permissions should be provided to Sentry to open the data to the authorized users.

Regardless the data is accessed either from HDFS or through Impala the security policies should be the same. That means that the security rules defined regardless the particular data acces and should be compiled into prpper permissions rule either on HDFS or Sentry.

Security rules

1. **Data Ownerhip**

Any data set should be owned by an identified principal. Even in case of an organisation a specific user should be identified as the owner of that specific dataset.

2. **Singler User Group**

Any user should have a corresponding group named with the same username, i.e. a user David Greco with a username `david` should have a group called `david` containing only the username `david`.

This is the default in the POSIX world, in case of an integration with Active Directory this policy needs to be enforced.

The reason of this rule lies in the fact that Sentry doesn't allow to grant a privilege to a user but only to a group. Since, we want to maintain the single owner principle with to resort to this policy to maintain our security approach coherent.

3. **Access Delegation**

Only the dataset owner is allowed to provide access to it.

For example, if a dataset `D` is owned by `david`, only `david` can give read access to D to another user `paul`.

Implementing this rule means appliying to the dataset HDFS directories additional HDFS ACLs and to grant specific roles at the Sentry level as we will show later.

The same rule apply in case a user wants to provide access to a group. In general the dataset owner can give access to the dataset to a combination of users and groups.

4. **Access Rights**

The access rights are only two: READ and WRITE, these rights as said before have to be mapped on proper HDFS ACLs and Sentry roles and grants.

Security Rules and mapping on HDFS and Sentry

Let's start with ordinary data first. So, we know that the ordinary data HDFS layout is defined in the following way:

`/daf/ordinary/`

The content of this directory is organized adopting the following rules:

`sourceOrg/domain/subdomain/datasetName.stage.datasetFormat`

where:

- `sourceOrg` is the name of the organization owning the dataset. This name is specified as `organizationType_organizationName`
- `domain` is the parent category to which the dataset belong (e.g. "mobility")

- `subdomain` is the sub-category to which the dataset belong (e.g. "traffic")
- `datasetName` is the name of the dataset
- `stage` is the particular stage of the dataset in the transformation pipeline, ex. `landing`, `stage1`
- `datasetFormat` specifies the serialization format. At the moment the Allowed format are: `csv`, `json`, `avro`, `parquet`.

So, let's suppose that a representative user for `sourceOrg` is `dataowner` then the following rules should be applied to HDFS:

1. `hdfs dfs -chown -R dataowner:dataowner /daf/ordinary/sourceOrg/domain/subdomain/datasetName.stage.datasetFormat`
2. `hdfs dfs -chmod -R go-rwx /daf/ordinary/sourceOrg/domain/subdomain/datasetName.stage.datasetFormat`
3. `hdfs dfs -setfacl -R -m user:impala:rwx /daf/ordinary/sourceOrg/domain/subdomain/datasetName.stage.datasetFormat`

1. This fix the ownership to the orgamnisation's representative user.
2. Only the dataset owner has full access to that dataset.
3. The impala user need access to all the datasets since Impala doesn't support secure impersonation and all the daemons run under the impala user.

All the directories rooted under `/daf/ordinary/sourceOrg` should follow the same rules.

Once the HDFS rules has been applied the next step is to apply a set of rules at the Sentry side to implement logically the same security rules we described before:

First of all a proper database should be created to hold the dataset, the database should be named accordingly with the following naming convention:

`domain__subdomain`

in Impala SQL DDL commands:

`CREATE DATABASE domain__subdomain`

Then an external table should be created pointing to the dataset directory, something like:

```
CREATE EXTERNAL TABLE  domain__subdomain.dataset_stage_format
(
   ...
)
...
LOCATION '/daf/ordinary/sourceOrg/domain/subdomain/dataset.stage.format';
```

Then proper Sentry roles and rights should be granted:

This is for the database:

```
CREATE ROLE db_domain__subdomain_role;
GRANT ALL ON DATABASE domain__subdomain TO ROLE db_domain__subdomain_role;
GRANT ROLE db_domain__subdomain_role TO GROUP dataowner;
INVALIDATE METADATA;
```

where the dataowner group is actually a group containing only the user `dataowner` as dictated by the security rule 2.

Then, this is for the table:

```
CREATE ROLE table_domain__subdomain__dataset_stage_format_role;
GRANT SELECT ON TABLE domain__subdomain.dataset_stage_format TO ROLE table_domain__
↪subdomain__dataset_stage_format_role;
GRANT ROLE table_domain__subdomain__dataset_stage_format_role TO GROUP dataowner;
INVALIDATE METADATA;
```

CHAPTER 4

Installation Guide

This section provides installation information for all the components of DAF, except for the Big Data Platform. Based on your development needs, you will be able to install only the individual components you require for your task. Almost every component has dependencies on other components–this will be documented in the installation guide of each component.

In general, components developed internally are available via GitHub repositories, meanwhile external ones have been dockerized with all needed dependencies and configurations.

Several components are dependent on LDAP and/or FreeIPA. In this case, we offer you three alternatives: a dockerized LDAP, a remote FreeIPA test server, or a dockerized FreeIPA (working with Linux only at the moment). All of them will have test accounts already created for you.

The best way to have everything installed and properly configured is to use the Virtual Machine.

See the Local Installation guide to know how to configure the Virtual Machine or, if you want to run only a few components, please follow the component installation guide you find at the links below.

4.1 Local Installation

This guide explains how to use the Virtual Machine to create a test environment.

The procedure will soon be migrated to Vagrant. If you want, you can contribute with a pull request to the upstream repository.

You can download the OVA image at the following link: download (8.6 GB)

Warning: In order to use the Virtual Machine, you must have at least 20 Gb of free space in your hard drive.

4.1.1 Virtual Machine Account

USER user

PASSWORD password

Check the IP address assigned to the Virtual Machine (also check bridge of virtual machines before start if you use wireless or Ethernet adapter on your PC) using the command:

```
> ip a | grep enp0s3
```

Access the Virtual Machine via ssh:

```
> ssh user@xxx.xxx.xxx.xxx (Virtual Machine IP)
```

4.1.2 Docker image

In the Virtual Machine, to access a folder with container docker image:

```
> sudo -i
> cd /root/docker
```

All the containers start automatically when the Virtual Machine starts.

4.1.3 Configuration

In your PC Hosts file, add the following lines:

```
x.x.x.x ipa.example.test superset.daf.test.it metabase.daf.test.it ckan.daf.test.it␣
↪mongodb ckan metabase supersetd
127.0.0.1 datipubblici-private.daf.test.it
```

where x.x.x.x is the Virtual Machine IP address.

Access Docker Services

When the Virtual Machine is running, you can access to the docker services from your browser.

FreeIPA

Use the following link https://ipa.example.test to access to FreeIPA.

The credentials are:

USER admin

PASSWORD adminpassword

or

USER ldap

PASSWORD ldap

The user `ldap` is used to bind docker system. Every user in `ldap` has the same user name and password.

If you are using Google Chrome, do not use the modal login on your browser, because it doesn't work.

Use the login in the web page.

Actual users:

- raffaele
- alssandro
- andrea
- pierpaolo
- david
- alberto

CKAN

Use the following link http://ckan:5000 to access the CKAN in the Virtual Machine.

Use only the user `ldap` to login.

METABASE

Use the following link http://metabase:3000 to access the metabase in the Virtual Machine, with credentials:

USER/MAIL	admin@admin.it
PASSWORD	admin01

or login with the user `ldap`.

SUPERSET

Use the following link http://supersetd:8088 to access the superset in the Virtual Machine.

USERNAME	superadmin
PASSWORD	password1

4.1.4 Services

In the host, run the following command to clone the DAF project:

```
> git clone https://github.com/italia/daf.git
```

In case sbt is not found, install it:

```
> echo "deb https://dl.bintray.com/sbt/debian /" | sudo tee -a /etc/apt/sources.list.
↪d/sbt.list
> sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv␣
↪2EE0EA64E40A89B84B2DF73499E82A75642AC823
> sudo apt-get update
> sudo apt-getinstall sbt
```

Common

On the host PC, go to the folder daf/common and run the following commands:

```
> sbt
> clean
> compile
> publishLocal
```

Security Manager

In your daf/security_manager folder, run:

```
> sbt
> clean
> compile
> run -Dconfig.resource=svil.conf -Dhttp.port=9002
```

Catalog Manager

On the host PC, go to the folder dat/catalog_manager and run the commands:

```
> sbt
> clean
> compile
> run -Dconfig.resource=svil.conf -Dhttp.port=9001
```

Dataportal

Clone the project daf-dataportal-backend from GitHub using the following command:

```
> git clone  https://github.com/italia/daf-dataportal-backend
```

In your daf-dataportal-backend project, run the following commands:

```
> sbt
> clean
> compile
> run -Dconfig.resource=local.conf
```

Front-end

Clone the project daf-dataportal from GitHub:

```
> git clone  https://github.com/italia/daf-dataportal
```

In your daf-dataportal project, add the following lines in . . . /src/config/serviceurl.js:

```
apiURLSSOManager: "http://localhost:9002/sso-manager",
apiURLDatiGov: "http://localhost:9000/dati-gov/v1",
apiURLCatalog: "http://localhost:9001/catalog-manager/v1",
```

(continues on next page)

(continued from previous page)

```
apiURLIngestion: "http://localhost:9002/ingestion-manager/v1",
apiURLSecurity: "http://localhost:9002/security-manager/v1",
urlMetabase: 'http://metabase.daf.test.it',
urlSuperset: 'http://superset.daf.test.it',

domain:".daf.test.it"
```

In your . . . /package.json edit the line in the section scripts

```
"start": "PORT=80 react-scripts start"
```

You can run the FE in the following modality:

Start in Debug Mode:

```
npm install
npm start
```

Start in Production Mode:

```
npm run build
npm install -g serve
serve -s build
```

For each configuration, the application should be reached through the following URL:

http://datipubblici-private.daf.test.it

When you access for the first time, click on the button "Registrati" to sign up. After the registration, access the FreeIpa, search for your account and add it to your user groups "daf_admins". Now, log out and log in again to DAF - Dataportal to see the admin features.

4.2 Catalog Manager

Catalog Manager is the microservice responsible for storing and retrieving metadata associated to the datasets stored in the DAF. It uses a Docker Compose CKAN based storage layer as a backend. It is developed following the OpenApi specification and contract-first design pattern.

Every microservice can run as a standalone module using mock data. Not all endpoint can retrieve mock data but all endpoints are described using https://swagger.io/ at localhost:9000/catalog-manager

See here for more details.

4.2.1 Local Installation

- Pre-requisites: JDK 8, SBT, GIT client
- git clone https://github.com/italia/daf
- cd daf/common
- sbt publishLocal
- cd ../catalog_manager
- sbt

- run
- connect to http://localhost:9000/catalog-manager

You should see a swagger UI with all endpoints described. Nevertheless, the authorization is not required by the UI you should pass at least a Basic authorization token made by an equal user name and password.

To test the endpoints, we suggest to use a tool like Postman

4.2.2 DAF integration note

[TBD]

4.2.3 Endpoints

[TBD]

4.3 Ingestion Manager

The **IngestionManager** manages all the data ingestion activities related to the datasets. The IngestionManager provides an API to ingest data from a data source into the DAF platform.

See here for more details.

4.3.1 Local Installation

- Pre-requisites: JDK 8, SBT, GIT client
- git clone https://github.com/italia/daf
- cd daf/common
- sbt publishLocal
- cd ../ingestion_manager
- sbt
- run
- connect to http://localhost:9000/ingestion-manager

You should see a swagger UI with all endpoints described. Nevertheless, the authorization is not required by the UI you should pass at least a Basic authorization token made by an equal user name and password.

To test the endpoints we suggest to use a tool like Postman

4.3.2 DAF integration note

[TBD]

4.3.3 Endpoints

[TBD]

4.4 Security Manager

Security Manager is the microservice responsible to manage security of the web application and the REST API developed within DAF. Its APIs verifies user's credential, produces JWT tokens needed to access DAF services and handles the SSO on the solutions integrated in the Data Portal.

4.4.1 Local Installation

- Pre-requisites: JDK 8, SBT, GIT client
- git clone https://github.com/italia/daf
- cd daf/common
- sbt publishLocal
- cd ../security_manager
- sbt
- run
- connect to http://localhost:9000/security-manager

You should see a swagger UI with all endpoints described. Nevertheless, the authorization is not required by the UI you should pass at least a Basic authorization token made by an equal user name and password.

To test the endpoints we suggest to use a tool like Postman

4.4.2 DAF integration note

[TBD]

4.4.3 Endpoints

[TBD]

4.5 Dataset Manager

The **DatasetManager** manages several operations related to the dataset, such as:

- to return the data of the dataset (or a sample of it) in a specified format;
- to create a specific view on top of a dataset;
- to get the dataset schema in a given format (e.g. AVRO);
- to create a new dataset based on an existing one but saved with a different storage mechanism or based on a transformation of the existing dataset, etc.

See here for more details.

4.5.1 Local Installation

- Pre-requisites: JDK 8, SBT, GIT client
- git clone https://github.com/italia/daf
- cd daf/common
- sbt publishLocal
- cd ../dataset_manager
- sbt
- run
- connect to http://localhost:9000/dataset-manager

You should see a swagger UI with all endpoints described. Nevertheless, the authorization is not required by the UI you should pass at least a Basic authorization token made by an equal user name and password.

To test the endpoints we suggest to use a tool like Postman

4.5.2 DAF integration note

[TBD]

4.5.3 Endpoints

[TBD]

4.6 Storage Manager

The **StorageManager** is responsible for abstracting the physical medium where the data is actually stored.

See here for more details.

4.6.1 Local installation

- git clone https://github.com/italia/daf
- cd daf/common
- sbt publishLocal
- cd ../daf/storage_manager
- sbt
- run
- connect to http://localhost:9000/storage-manager

4.6.2 DAF integration note

[TBD]

4.6.3 Endpoints

[TBD]

4.7 Frontend Manager

4.7.1 What is it?

The **Frontend Manager** is the layer that serves functionalities to the frontend web applications (Dataportal-public and Dataportal-private). It integrates external applications (e.g. CKAN) and exposes REST API to the web apps.

It has been developed using the following technologies:

- playframework 2.5.12
- scala 2.11.8
- sbt 0.13
- Zalando's api-first-hand

4.7.2 Install

- `$ git clone git@github.com:italia/dati-frontendserver.git`
- `$ sbt compile`
- `$ sbt run`
- Connect to `http://localhost:9000`

4.7.3 Setup

…

4.8 Dataportal-public

4.8.1 What is it?

The Dataportal-public is the web app that allows access to the open data catalog and other content that can be exposed publicly.

4.8.2 Install

Before proceeding with the installation steps, you need to install and run the following external components:

Basic Dependencies

There are no basic dependencies needed.

Features Enabling Dependencies

Connection with DataStories:

- daf-dataportal-backend

Installation Steps

First of all, clone the following GitHub repository:

```
> $ git clone https://github.com/italia/daf-dataportal-backend
```

Start in Debug Mode:

```
> npm install
> npm start
```

Start with mock server:

```
> npm run mock
```

Start in Production Mode:

```
> npm run build
> npm install -g serve
> serve -s build
```

4.9 Dataportal-private

4.9.1 What is it?

The Dataportal-private is the web app that allows access to the functionalities of DAF, like:

- **Ingestion** form to add dataset with metadata
- **Business Intelligence** with Superset (AirBnB)
- **Graphs** with Metabase
- **Data science** with Jupyter + Sparkmagic
- Ontologies and Controlled Vocabularies repository

4.9.2 Install

Before proceeding with the installation steps, you need to install and run the following external components:

Basic Dependencies

- daf-dataportal-backend
- FreeIPA
- CatalogManager

- SecurityManager

Features Enabling Dependencies

- Superset
- Metabase
- JupyterHub
- CKAN

Installation Steps

First of all, you need to clone the following GitHub repository:

```
> git clone https://github.com/italia/daf-dataportal
```

Start in Debug Mode:

```
> npm install
> npm start
```

Start with mock server:

```
> npm run mock
```

Start in Production Mode:

```
> npm run build
> npm install -g serve
> serve -s build
```

4.10 Semantic microservices

The semantic part of the DAF consists of the following microservices:

4.10.1 OntoNetHub

OntoNetHub is a microservice meant to deal with the management of ontology networks. This include the upload, deletion, storage, and indexing of an ontology part of a network.

OntoNetHub is designed as an extension of Apache Stanbol and released as a Docker component. Hence, users need Docker to build and run OntoNetHub.

Local Installation

- Pre-requisites: JDK 8, SBT, GIT client
- git clone https://github.com/teamdigitale/ontonethub
- docker-compose build
- docker-compose up

- connect to http://localhost:8000/stanbol/ontonethub

To test the endpoints it's possible to use a tool similar to Postman

DAF integration note

This microservice currently provides functionalities to the semantic_manager, which takes care of integrating them to DAF and the public dataportal:

- with the ingestion form of DAF, providing suggestions for the "semantic" annotations of dataset fields.
- with the public dataportal, providing a list of available ontologies and "core" vocabularies.

Endpoints

There is a list of available endpoints:

- */stanbol/ontonethub/ontology* : can be used to add a new ontology using a POST request.
- */stanbol/jobs/{job_id}* : provides informations about the status of a job associated with the upload of an ontology.
- */stanbol/ontonethub/ontology/{ontology_id}* : can be used with a GET request to access the information about the specific ontology.
- */stanbol/ontonethub/ontology/{ontology_id}* : can be used with a DELETE request for deleting an existing ontology.
- */stanbol/ontonethub/ontology/{ontology_id}/source* : can be used with a GET request for obtaining a representation of the ontology in JSON-LD.
- */stanbol/ontonethub/ontologies/find* : can be used for querying the OntoNetHub and retrieving OWL entities from the ontologies managed by it.

Detailed informations about the service can be found here

4.10.2 Semantic Manager

Semantic Manager is the microservice designed to provide a central access point for the so-called "semantic" functionalities, involving the usage of ontologies and core vocabularies supporting both DAF processes and the catalog front-end for users.

Local Installation

- Pre-requisites: JDK 8, SBT, GIT client
- git clone https://github.com/italia/daf-semantics
- cd daf-semantics/semantic_manager
- sbt docker:publishLocal
- sbt run
- connect to http://localhost:9000

You should see a swagger ui with all endpoints described.

To test the endpoints I suggest to use a tool similar to Postman

DAF integration note

This microservice is currently integrated:

- with the ingestion form of DAF, providing suggestions for the "semantic" annotations of dataset fields. Those annotations are saved into the schema for the imported dataset, and act as references for the standardization of fields.
- with the public dataportal, providing a list of available ontologies and "core" vocabularies.

Endpoints

There are two endpoints:

- */kb/v1/ontologies* : provides a list of the available ontologies
- */kb/v1/ontologies/properties/find* : enable searching by terms and language for properties which may be used for a simple annotation of dataset fields in the ingestion form (and later for standardization).

Detailed informations about the service can be found here

4.10.3 Semantic Repository

Semantic Repository is the microservice designed to provide basic functionalities for managing ontologies/vocabularies (and data, in the future) using the the well-know [RDF4J](http://rdf4j.org/) interface as an abstraction over triplestores. The idea is to have a list of core functionalities for a catalog service of queryable ontologies, which can be implemented over an external triplestore, and it will evolve accordingly.

Local Installation

- Pre-requisites: JDK 8, SBT, GIT client
- git clone https://github.com/italia/daf-repository
- cd daf-semantics/semantic_repository
- sbt docker:publishLocal
- sbt run
- connect to http://localhost:9000

You should see a swagger ui with all endpoints described.

To test the endpoints I suggest to use a tool similar to Postman

DAF integration note

[TBD]

Endpoints

There is a list of basic endpoints:

- */kb/v1/ontologies* : using this endpoint it's possible to add new ontologies using HTTP POST with parameters. By conventions the user should assign a prefix / context pair (which willbe unique in the underlying repository catalog).

- */kb/v1/ontologies/remove* : the endpoint is dedicated to remove an existing ontology, using the context where it was published.
- */kb/v1/contexts* : a list of the existing context can e retrieved: by convention ontologies are published on assigned contexts, which will be different from the ones used for data.
- */kb/v1/prefixes* : this endpoints returns the full list of used prefix / namespace pairs, where the namespace usually coincide with an assigned contexts on the underlying repository.
- */kb/v1/prefixes/lookup* : this endpoint can be used for retrieving the namespace associated to a given prefix.
- */kb/v1/prefixes/reverse* : this endpoint can be used for retrieving the prefix associated to a given namespace.
- */kb/v1/triples* : provides the amount of available triples.
- */kb/v1/triples/{prefix}* : provides the triples count for a given context.

Detailed informations about the service can be found here

4.10.4 Semantic Validator

Semantic Validator is the microservice designed to provide a simple way for validating RDF metadata dataset against a specific Ontology on an underlying triplestore. The validator is currently based on a set of queries (about 150 for DCAT-AP_IT) returning a record of information for the rules broken by the dataset, the most important infos are:

- Class name: the class involved in the rule (ex: Organization for DCAT-AP_IT)
- Rule ID: the broken rule id (ex: 207 for DCAT-AP_IT)
- Error description: the problem description (ex: “vcard:hasURL should be a resource” for DCAT-AP_IT)

Local Installation

- Pre-requisites: JDK 8, SBT, GIT client
- git clone https://github.com/italia/daf-semantics
- cd daf-semantics/semantic_validator
- sbt docker:publishLocal
- sbt run
- connect to http://localhost:9000

You should see a swagger ui with all endpoints described.

To test the endpoints I suggest to use a tool similar to Postman

DAF integration note

This microservice will be integrated with the front-end component “semantic_frontend” in the block called “Public Manager” as you can see in the DAF architecture main schema.

Endpoints

There are two endpoints:

- */validator/validate* : in order to validate a document

- */validator/validators* : in oder to ghe the list of available validators

Detailed informations about the service can be found here

4.11 Freeipa LDAP version: 4.4.0

4.11.1 Docker

FreeIPA server can be run in a Docker container for testing or demo purposes. It makes it possible to run all the processes comprising the server in an isolated way, leaving the host free to run other software, not clashing with the FreeIPA server.

This installation is done on Ubuntu 16.04. FreeIPA is focused on Linux (and other standards compliant) systems. Therefore, in our knowledge, you cannot run a container of a FreeIPA server on **Mac OS** or **Windows**. However, any help in this direction is very welcomed!!

Follow these steps to run our FreeIPA server docker:

1. Create a directory which will hold the server data:

```
> mkdir /var/lib/ipa-data
```

2. Edit */etc/hosts* and ensure that the IPA server address is listed. This is required for Apache to work properly. You have to change IPA_SERVER_IP with the IPA server IP:

```
127.0.0.1      localhost localhost.localdomain localhost4 localhost4.localdomain4
::1            localhost localhost.localdomain localhost6 localhost6.localdomain6
IPA_SERVER_IP    ipa.example.test
```

3. Finally, You run the container:

```
> docker run -it -p 389:389 -p 443:443 -p 636:636 --name freeipaldap --cap-add SYS_
↪ADMIN --security-opt seccomp:unconfined -v /sys/fs/cgroup:/sys/fs/cgroup:ro --tmpfs␣
↪/run --tmpfs /tmp -v /var/lib/ipa-data:/data:Z -h ipa.example.test italia/freeipa-
↪server --ds-password=The-directory-server-password --admin-password=The-admin-
↪password
```

where:

- *–cap-add SYS_ADMIN*, performs a range of system administration operations (see here for more details).
- *–security-opt seccomp:unconfined* to run a container without the default seccomp profile (see here for more details).
- *-v /var/lib/ipa-data:/data:Z* to store data and configurations in the folder */var/lib/ipa-data/*

Answer to the question:

Do you want to configure integrated DNS (BIND)? [no]: –> press "Enter"

Server host name [ipa.example.test]: –> press "Enter"

Please confirm the domain name [example.test]: –> press "Enter"

Please provide a realm name [EXAMPLE.TEST]: –> press "Enter"

Continue to configure the system with these values? [no]: –> type "y" and press "Enter"

Wait some time until FreeIPA server is completely configured and started. The server is ready when on the shell the following message appears:

```
> FreeIPA server configured.
```

Note: You need to answer the previous questions only the first time you build the image and run docker.

- You can connect to FreeIPA Server from a web interface:

  https://IPA_SERVER_IP:443

  USER: admin

  PW: adminpassword

- You can also connect with an LDAP client with Server IP address IPA_SERVER_IP
- The container can then be started and stopped with the following commands:

```
> docker stop freeipaldap
> docker start freeipaldap
```

4.11.2 References

[1] FreeIpa docker-hub documentation.

[2] Using Free Ipa for user authentication.

[3] FreeIpa website.

4.12 LDAP Installation

This docker container allows you to start a simple LDAP server (OpenLdap) and a client (phpLDAPadmin). In particular, the Docker Compose downloads an initial database having domain *daf.test.it* and containing the user *bob* with password *password*.

Clone the git project:

```
> git clone git@github.com:italia/daf-recipes.git
```

Run the docker container:

```
> cd ./daf-recipes/ldap
> docker-compose up -d
```

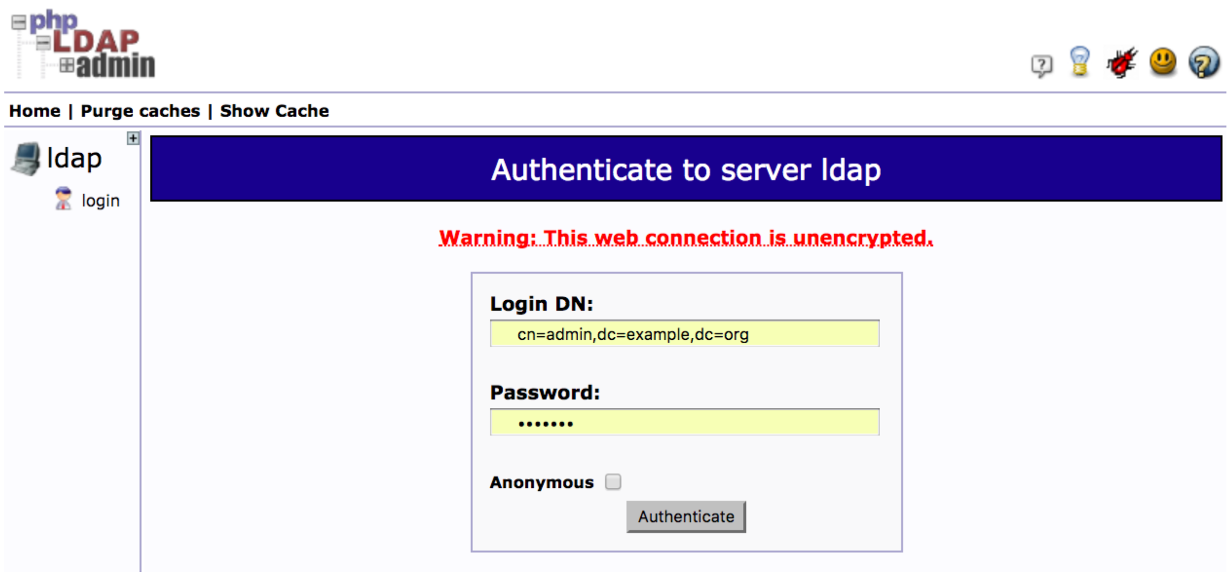
Check whether dockers are running:

```
> docker ps
e8ff9611aeff  osixia/openldap  "/container/tool/r..."  17 minutes ago  Up 17 minutes ␣
↪0.0.0.0:389->389/tcp, 0.0.0.0:636->636/tcp  ldap
6a0d0d6c3b9a  osixia/phpldapadmin  "/container/tool/run"  17 minutes ago  Up 17␣
↪minutes  0.0.0.0:80->80/tcp, 443/tcp  phpldapadmin
```

Note

The Docker Compose requires that ports 80, 636 and 389 are available. If not, change them.

Now, open your favorite browser and type *http://localhost*.



Authenticate to server ldap

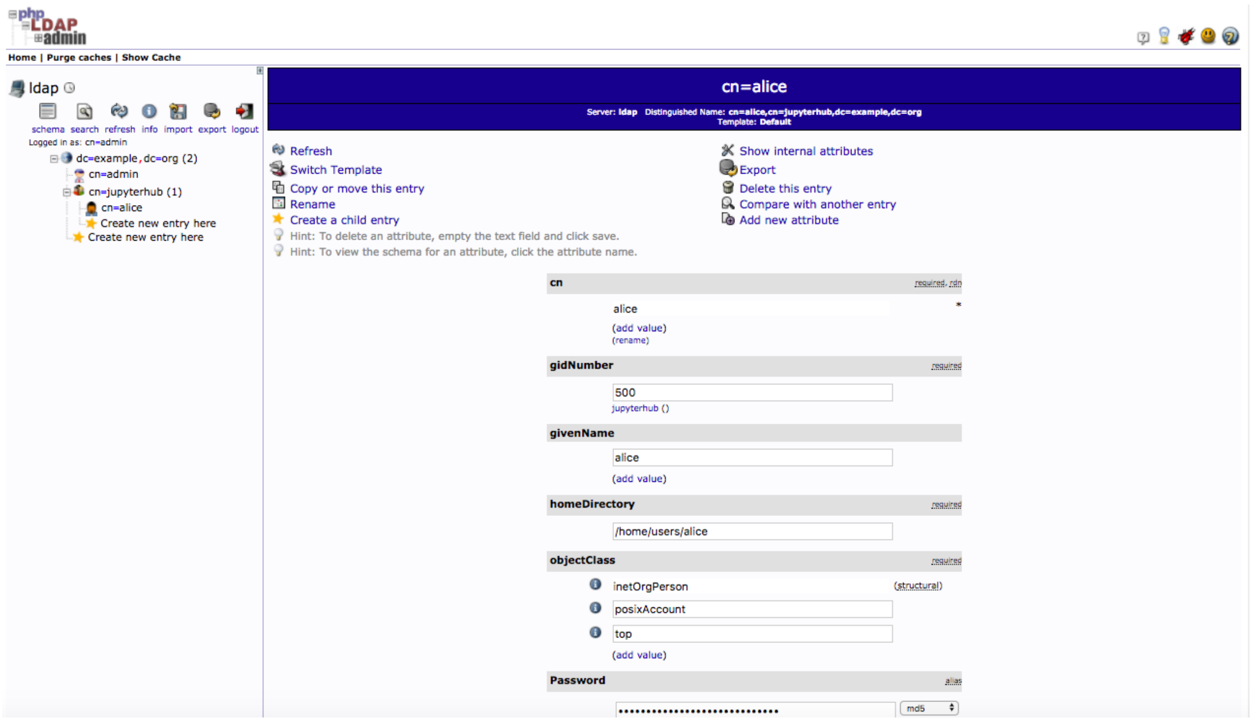
Warning: This web connection is unencrypted.

Login DN: cn=admin,dc=example,dc=org

Password: •••••••

Anonymous

Login as *cn=admin,dc=example,dc=org* and password *admin* to navigate inside.



Logged in as: cn=admin

```
dc=example,dc=org (2)
  cn=admin
  cn=jupyterhub (1)
    cn=alice
    Create new entry here
  Create new entry here
```

cn=alice

Server: ldap Distinguished Name: cn=alice,cn=jupyterhub,dc=example,dc=org
Template: Default

Hint: To delete an attribute, empty the text field and click save.
Hint: To view the schema for an attribute, click the attribute name.

cn — required, rdn
alice *
(add value)
(rename)

gidNumber — required
500
jupyterhub ()

givenName
alice
(add value)

homeDirectory — required
/home/users/alice

objectClass — required
inetOrgPerson (structural)
posixAccount
top
(add value)

Password — alias
••••••••••••••••••••••••••••• md5

4.12.1 FreeIpa Instance

We installed a FreeIpa server which can be used for test purposes. It can be reached at the address *91.206.129.245*.

4.13 CKAN

CKAN is an open-source DMS (data management system) for powering data hubs and data portals. CKAN makes it easy to publish, share and use data. It powers datahub.io, catalog.data.gov and data.gov.uk, among many other sites.

This guide will show you how to use Docker Compose to set up and run a CKAN instance which uses LDAP credentials to authenticate users. In particular, you can use an openLDAP Docker container or a FreeIpa instance.

4.13.1 Account Management Dependency

This configuration of CKAN needs an account management system to work with. We provide three different options, you will find more info on their respective sections:

- Local LDAP Docker
- Local FreeIPA Docker (works only with Linux)
- Remote FreeIpa Server

4.13.2 Ckan docker compose

Now that we have a LDAP server up we can run the CKAN Docker Compose. It will run an instance of Solr, Postgresql, Redis and Mongo.

First we have to build a custom image:

```
> cd ./daf-recipes/ckan
> ./build_local.sh
```

Then edit the file *ckan.ini*:

- If you are using our openLDAP server:

```
# LDAP Intergration with ldap and ip address
ckanext.ldap.uri = ldap://LDAP_IP:389
ckanext.ldap.auth.dn = cn=admin,dc=daf,dc=test,dc=it
ckanext.ldap.auth.password = admin
ckanext.ldap.base_dn = cn=users,cn=accounts,dc=daf,dc=test,dc=it
ckanext.ldap.search.filter = uid={login}
ckanext.ldap.username = uid
ckanext.ldap.email = mail
ckanext.ldap.ckan_fallback = True
```

where LDAP_IP is the IP of the LDAP docker. To know the LDAP IP, run:

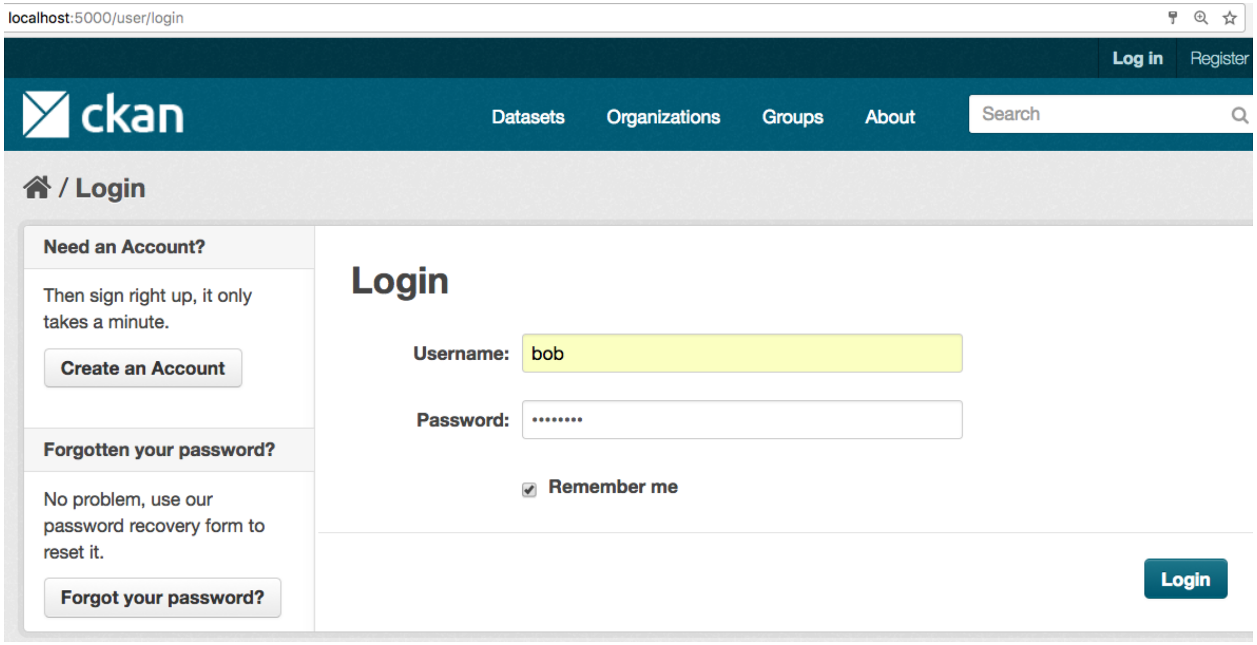
```
> docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' ldap
172.22.0.2
```

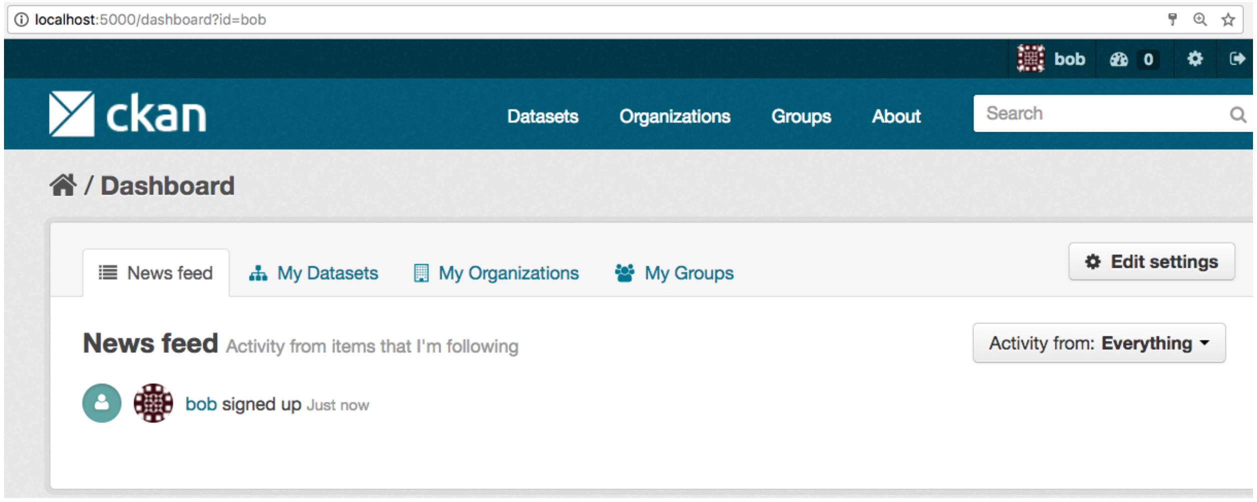
We know that this is not the best approach to connect containers among them (maybe it is the worst), we are using a deprecated compose file version (i.e. version 1 rather than using version 3), and we are using very heavy images. We will improve CKAN docker compose as soon as possible.

- **If you are using our FreeIpa server**

```
# LDAP Intergration with ldap and ip address
ckanext.ldap.uri = ldap://91.206.129.245:389
ckanext.ldap.auth.dn = uid=admin,cn=users,cn=accounts,dc=daf,dc=test,dc=it
ckanext.ldap.auth.password = aiyaiPh8
ckanext.ldap.base_dn = cn=users,cn=accounts,dc=daf,dc=test,dc=it
ckanext.ldap.search.filter = uid={login}
ckanext.ldap.username = uid
ckanext.ldap.email = mail
ckanext.ldap.ckan_fallback = True
```

Now that CKAN container is up, type *http://localhost:5000* on your browser and login as user *bob* (password *password*).





4.14 Superset

This guide presents how to configure and run a docker containing a Superset instance.

Clone the git project:

```
> git clone git@github.com:italia/daf-recipes.git
```

Go to the superset directory:

```
> cd superset
```

The Superset docker connects, by-default, to a test LDAP server provided by the DAF platform. If you need to configure another LDAP server, choose the proper configuration of the LDAP parameters inside the configuration file "superset_config.py":

- AUTH_TYPE = AUTH_LDAP
- AUTH_LDAP_SERVER = "ldaps://server:636"
- AUTH_LDAP_SEARCH = "cn=users,cn=accounts,dc=test,dc=example,dc=it"
- AUTH_LDAP_UID_FIELD = "uid"
- AUTH_LDAP_FIRSTNAME_FIELD = "givenName"
- AUTH_LDAP_LASTNAME_FIELD = "sn"
- AUTH_LDAP_EMAIL_FIELD = "mail"
- AUTH_LDAP_BIND_USER = "uid=admin,cn=users,cn=accounts,dc=test,dc=example,dc=it"
- AUTH_LDAP_BIND_PASSWORD = "password"
- AUTH_LDAP_ALLOW_SELF_SIGNED = True

Build the Superset image:

```
> ./build.sh
> docker-compose up -d
```

This will start Superset, Postgres and Redis servers. Wait some time to be sure all processes are up and running.

Run the `init` command to load into Superset some data for testing:

```
> ./init.sh
```

Connect to http://localhost:8088/ to browse the Superset web app.

4.15 Metabase

This guide explains how to run and execute a Metabase server.

Follow these steps to run the Docker images.

Clone the git project:

```
> git clone git@github.com:italia/daf-recipes.git
```

Go to the `metabase` directory, the images needed by docker-compose and run it:

```
> cd metabase
> ./build_local.sh
> docker-compose up -d    # it will run all the needed containers
```

Open the Metabase home at http://localhost:3000.

Go to GitHub to check how to set up Metabase.

4.16 Jupyter

This guide will show you how to use Docker Compose to set up and run a JupyterHub instance which uses LDAP credentials to authenticate users.

4.16.1 Account Management Dependency

This configuration of CKAN needs an account management system to work with. We provide three different options, you will find more info on their respective sections:

- Local LDAP Docker
- Local FreeIPA Docker (works only with Linux)
- Remote FreeIpa Server

4.16.2 JupyterHub

This Docker container runs a JupyterHub instance which is connected with a PostgreSQL database.

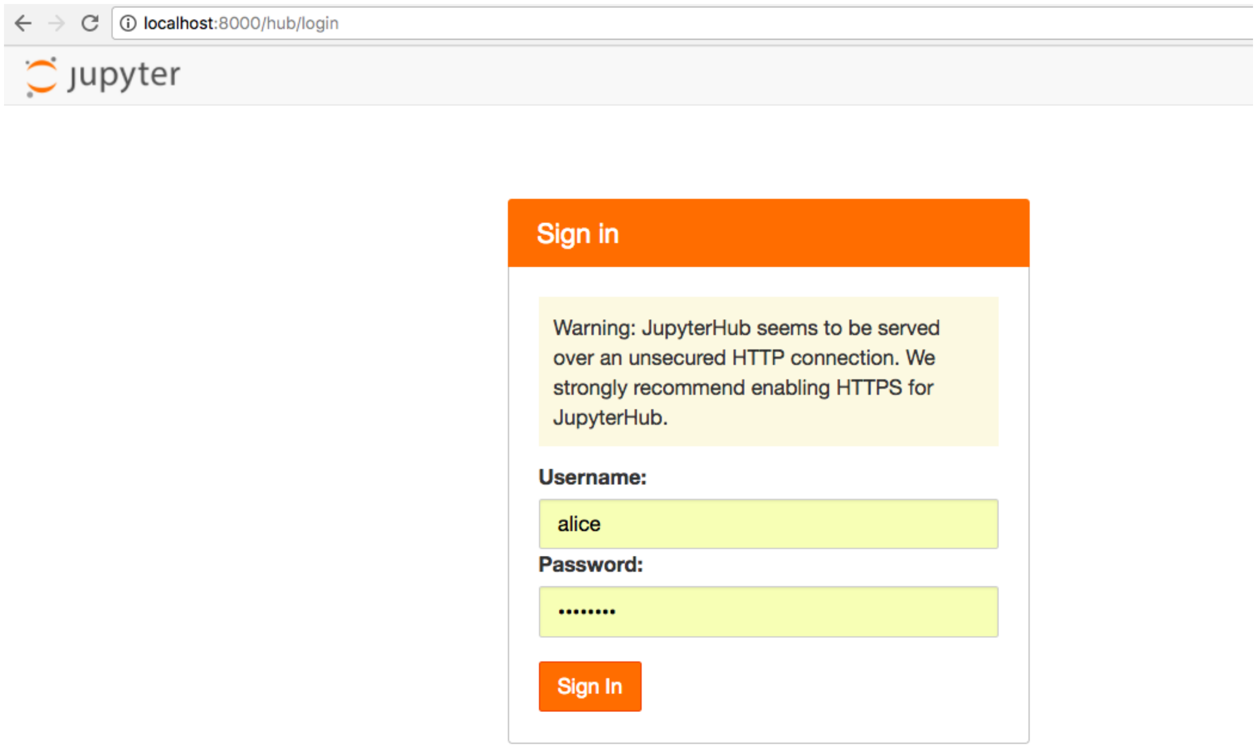
Run the Docker container:

```
> cd ./daf-recipes/jupyterhub
> docker-compose up -d
```

Check whether dockers are running:

```
> docker ps
8350963ac06c        jupyterhub_jupyterhub   "/wait_db_is_ready.sh"   16 minutes ago   ␣
↪    Up 16 minutes       0.0.0.0:8000->8000/tcp                    jupyterhub
6a0d0d6c3b9a        osixia/phpldapadmin     "/container/tool/run"    17 minutes ago   ␣
↪    Up 17 minutes       0.0.0.0:80->80/tcp, 443/tcp               phpldapadmin
e8ff9611aeff        osixia/openldap         "/container/tool/r..."   17 minutes ago   ␣
↪    Up 17 minutes       0.0.0.0:389->389/tcp, 0.0.0.0:636->636/tcp   ldap
cee2d35feaaf        postgres:9.6            "docker-entrypoint..."   2 hours ago      ␣
↪    Up 2 hours          0.0.0.0:5432->5432/tcp                   ␣
↪postgresjupyterhub
```

To open the interactive shell type *http://localhost:8000* and login as user *alice* (password *password*).



Warning: JupyterHub seems to be served over an unsecured HTTP connection. We strongly recommend enabling HTTPS for JupyterHub.

Username: alice

Password: ••••••••