OD: Checkpoint 2

Franz Bermeo, Maryam Pashmi, Adrian Vogelsgesang

Our project is hosted under https://github.com/vogelsgesang/upc-od. We are using the wiki in order to organize our information. If one of the descriptions here is too superficious, more details with regards to the topics of this checkpoint can be found under https://github.com/vogelsgesang/upc-od/wiki/Checkpoint-2. If necessary, the provenance of the parts of this documentation can be tracked using the history function of this wiki.

Data sources

Name	Description	License	Format
Harvard Library Cloud	Bibliographic records of the Harvard university	CC0	JSON API
North Rhine-Westphalian Library Service Center's (hbz)	Linked data from Germany	CC0	JSON API
Cambridge University Library	Access to library catalogues	mixed	XML/JSON API
DBLP	On-line reference for bibliographic information on major computer science publications	ODC-BY 1.0 license	XML
CERN data set	Bibliographic Data from CERN	CC0	XML/Marc21
Open Library JSON API	spinoff of The Internet Archive	CC0 1.0	JSON
Cambridge Library	Bibliography Service	GPL	SPARQL
British Library	Bibliography Service	CC0 1.0	SPARQL

Cern Bibliography data (XML)

We decided to integrate the bibliography data of CERN available for bulk download (CC0 license). This data is coded as XML/Marc21, i.e. a recoding of the old Marc21 format to XML. The original Marc21 field names are still contained in the transcoded XML. Documentation for the meaning of these field names can be found under http://www.loc.gov/marc/bibliographic/ecbdhome.html. In our wiki we have a composition of all the relevant Marc21 field names.

In reference to data repository, we have chosen eXist-db as our native XML database. It provides a robust and efficient indexing to manage amounts of unstructured data, documents or collection. Thus we can take advantage of its support to XML queries and reduce the number of data transformations.

As technique to perform querying, we will use XQuery and its powerful features to manipulate XML-based object databases and in combination with XPath expressions to do much easier the surf through syntax. To achieve manage of the best way our data collections, we propose to use a well-known schema to represent the bibliography data and so to accelerate the evaluation of path expressions.

We are using eXistDB for accessing the XML sources. We can write XQuery-Documents and store them in the database. The results of evaluating these XQuery documents on the database are returned when sending a GET request to

```
<existDb-server>/exist/rest/<path of the document>?parameters
```

We can use placeholders in the XQuery document whose values can be specified as GET parameters. Hence, we can retrieve the data by querying the correct documents using the corresponding parameters. Alternatively, we can send a XQuery document to the relevant XML-document using a POST request. eXistDb will return the results of evaluating this Xquery against the corresponding XML-document.

In both cases, the access to the database is achieved by sending HTTP requests to the eXistDB server. We favored the first possibility (justification can be found in the wiki) The Xquery document can be found in the repository. F.e., by issuing the query

`http://localhost:8080/exist/rest//db/od/marc21_search.xq?100a=Horowitz,%20Ellis&limit=20&offset=0`

all records containing a Marc21 field 100a (the author field) with the content "Horowitz, Ellis" are returned. The results are sent as XML by the eXistDb server. Hence, accessing the XML contents can be broken down to the task of accessing an XML API.

The Harvard LibaryCloud API (Json API)

We decided to use the Harvard LibaryCloud API which is offered under the Creative Commons Zero license (CC0) . This repository contains information from the Harvard Library Bibliographic Dataset, which is provided by the Harvard Library under its Bibliographic Dataset Use Terms and includes also data made available by, among others, OCLC Online Computer Library Center, Inc. OCLC Online Computer Library Center, Inc. and the Library of Congress.. This dataset contains over 12 million bibliographic records. The data is also available for bulk download. A more in depth documentation of the original data is available here.

Request to this API are sent as GET request offers a search functionality on top of these records and renames Marc21 fields into human readable names. In addition, the original Marc21 record is returned. We can execute 3 queries per second from a single IP address. Example query:

`http://librarycloud.harvard.edu/v1/api/item/?filter=keyword:internet`

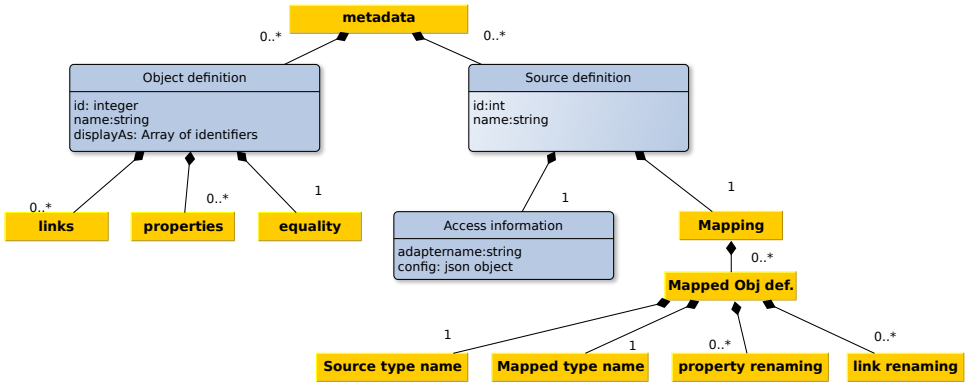
The following excerpt shows some of the returned fields. Complete documentation is available on the web.

Field name	Field description
keyword	Keywords.
id	The identifier given to the item here in LibraryCloud. Exact matching.
title	The title and/or subtitle of the item. Exact matching.
title_keyword	The title and/or subtitle of the item. Keyword matching.

The API allows to filter on multiple criteria, and supports pagination and sorting the results.

Parameter name	Parameter description
filter	Filters. Syntax: fieldname:filter Multiple filter parameters can be provided.
limit	Number of records to return. Default is 25. Max is 250.
start	The starting point in the result set. Default is 0.
sort	Specifies the sort order. Default: “shelfrank desc”. *This parameter is undocumented and not officially supported!*

Meta data



Our internal schema is based on the multi graph model: We have a set of entities. Every object is an instance of a specific type and can have a set of properties. These entities can be linked by relations which are directed edges.

Properties on relations are not supported. Inheritance is not supported neither.

The specification of our schema is saved as a set of object type definitions. Next to a name and an enumeration of properties and links, a definition of equality is stored which is used for the elimination of duplicates. In addition, we save information about our sources (how can the data be accessed? Which mapping must be done?) in machine readable form. All this information will be stored in a local MongoDB instance.

Currently we are facing the following unsolved problems:

- Determining the equality of field values might be non-trivial. F.e the two author names "Rowling, J.K." and "Joanne K. Rowling" are referring to the same persons but a simple check for string equality will not unveil this equality. Even the usage of a Levenshtein distance does not solve this problem.
- Fields with different granularities. F.e.: some sources split the authors name in first name and surname, others only provide a field which contains both informations concatenated.
- Differences in the graph structure: Some sources store the author's name as a property of a book while others are adding a link to an author and store his name as a property of this linked node

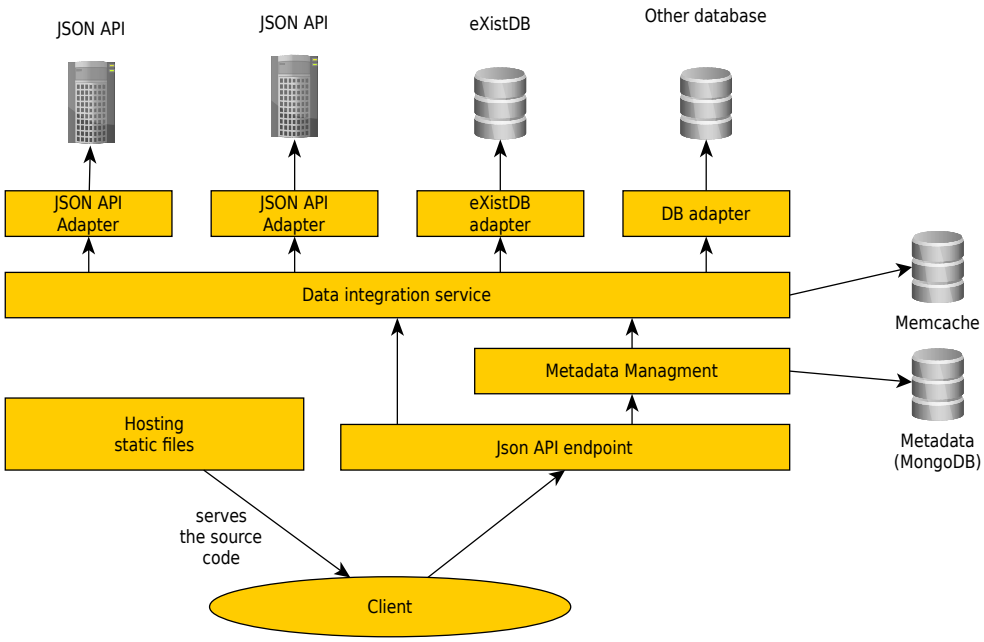
Some thoughts about possible solutions for these problems and our mapping into a Mongo document can be found under https://github.com/vogelsgesang/upc-od/wiki/Meta-data

System architecture

Since most data will be retrieved from sources which are based on web technologies, our system will be based on web technologies, as well. It will be divided into a client and a server. The server is responsible for creating a consistent view of the data contained in the data sources. It provides access to this data by means of a JSON Api. Thanks to this, our software can be reused and integrated into other systems. The client makes calls to this Api in order to retrieve the data and display it to the end user in a more intuitive way. Additional Implementation details are provided in a separate wiki page.

The client

The client is a one-page-webapp which provides a GUI for the JSON Api of our server. It is served by our server statically, i.e. the server will not embed any type of information into the delivered Html/Js files. The JS code of the client is responsible for querying the relevant information using the server's Json API. It does not take care of any type of data integration.



The server

The server has four responsibilities:

- Deliver the clients source code
- Provide an API for modifying the meta data repository
- Build a consistent view of the data and deliver this view through the API
- Providing recommendations (no concrete plans so far. Maybe we will use Random Walks)

The meta data is stored in a MongoDB instance. Access is provided using a REST Api.

Consolidated data is served by constructing this data on the fly. For this purpose, all sources configured in the meta data repository are queried for relevant informations. This returned information is filtered and if its is actually fitting our initial query, we add it to the set of acknowledged information. Every time when new information is acknowledged, all other sources are checked again if additional informations can be found using the new set of available base information. This process is repeated until no source is able to contribute additional information.

All sources are connected with our core through adapters (See Interface specification for source adapters). These modules are responsible for speaking to the relevant databases/web services. Every module delivers the data in a common format which still contains the field names of the source. The core of our data integration service takes care of mapping and restructuring these fields.

All sources are queried asynchronously, i.e. in parallel. This is done in order to avoid unnecessary idle times and for constructing the consolidated data faster. In order to further improve the response time, consolidated data is cached using MemCache. In addition, the unaggregated data returned by each source is cached in order to avoid the necessity of reloading the data from all sources if the configuration of one source is changed.