code{4}lib JOURNAL

Issue 17, 2012-06-01 ISSN 1940-5758

Using Semantic Web Technologies to Collaboratively Collect and Share User-Generated Content in Order to Enrich the Presentation of Bibliographic Records–Development of a Prototype Based on RDF, D2RQ, Jena, SPARQL and WorldCat's FRBRization Web Service

In this article we present a prototype of a semantic web-based framework for collecting and sharing user-generated content (reviews, ratings, tags, etc.) across different libraries in order to enrich the presentation of bibliographic records. The user-generated data is remodeled into RDF, utilizing established linked data ontologies. This is done in a semi-automatic manner utilizing the Jena and the D2RQ-toolkits. For the remodeling, a SPARQL-construct statement is tailored for each data source. In the data source used in our prototype, user-generated content is linked to the relevant books via their ISBN. By remodeling the data according to the FRBR model, and expanding the RDF graph with data returned by WorldCat's FRBRization web service, we are able to greatly increase the number of entry points to each book. We make the social content available through a RESTful web service with ISBN as a parameter. The web service returns a graph of all user-generated data registered to any edition of the book in question in the RDF/XML format. Libraries using our framework would thus be able to present relevant social content in association with bibliographic records, even if they hold a different version of a book than the one that was originally accessed by users. Finally, we connect our RDF graph to the linked open data cloud through the use of Talis' openlibrary.org SPARQL endpoint.

by Ragnhild Holgersen, Michael Preminger, and David Massey

Preface

This article is based on a development project by Ragnhild Holgersen in the Digital Library course of the Master's Program in Library and Information Science at the Oslo and Akershus University College of Applied Sciences. Assistant professors Michael Preminger and David Massey have assisted. We thank the Stockholm Public Library for letting us use the Öppna bibliotek data in our prototype.

Introduction

Public libraries are competing for their users' attention against highly attractive, state of the art, commercial and community-based websites. There is a growing expectation for interactivity and integrated "social content," such as reviews, ratings, folksonomy tags and easy sharing through established social networks such as Facebook and Twitter.

Many social features require a critical mass of users and user-generated content in order to be useful. If such a feature were to be offered based on limited user activity, individual users' preferences would create a lot of noise, making the recommendations arbitrary and unhelpful.

Most public libraries have neither the resources nor the user mass necessary to provide successful social features in their own OPACs. It could therefore be interesting to develop a common repository of social data with an API that makes it easy for each library to integrate the desired features. This is the idea behind the Swedish project called "Öppna bibliotek" (English: "The Open Library") (Anderson 2010).

RDF is one of the W3C standards underpinning the Semantic Web (W3C 2004). Briefly stated, RDF is used to model a domain by making statements about resources in the form of simple subject – predicate – object triples. RDF is a very simple, yet extremely powerful standard, with serializations suitable for machine consumption and reasoning. Despite its simplicity, most data models can successfully be mapped to RDF.

Basing the social data service on RDF would thus make it possible to integrate data from many different sources, such as relational databases, XML documents and web services, and make these available in a seamless way. This would allow each library to continue maintaining their existing dataset, instead of forcing everyone to store their data in one common database. Some libraries may then choose only to contribute with folksonomy tags, while others may contribute with assessments, ratings and so on.

This article describes a prototype that extracts social data stored in a relational database, transforms the data to RDF and adds relevant data from other sources. The data can then be used to enrich OPACs and similar services. We start by describing the environment of the prototype, then provide a technical description of the prototype itself and the software framework used to create it, and its intended use. Finally, we show how we connect our RDF data to the linked open data (LOD) cloud and discuss future work.

The Prototype

In the prototype, an RDF graph of user-generated data from just one data source is generated. However, as Figure 1 shows, the suggested solution can be extended in order to incorporate additional data sources.

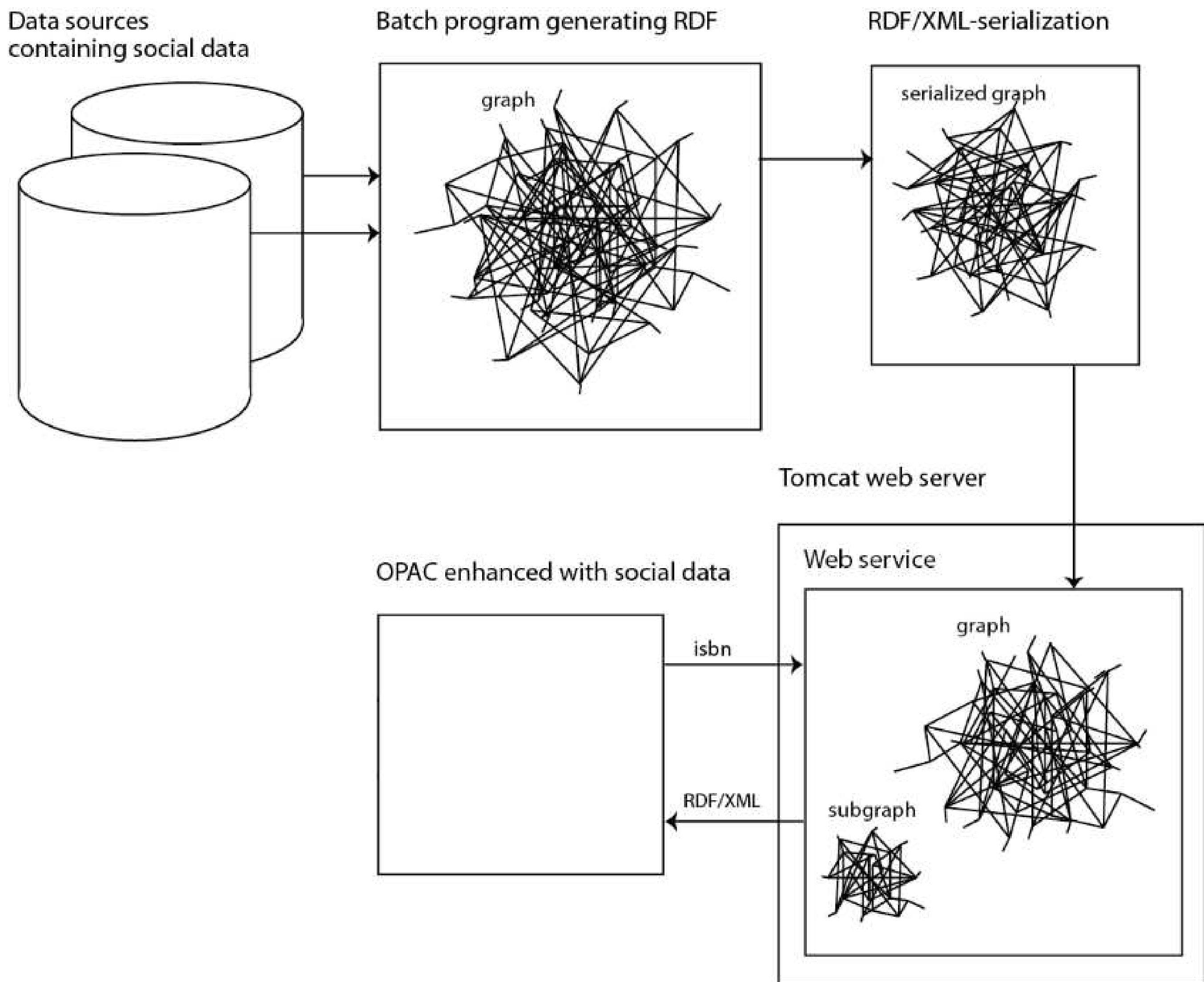

Figure 1. Conceptual Sketch of the Presented Social Data Service

In true Linked Data spirit, the graph only provides the content that it specializes in, the user-generated content, and relies on better-suited bibliographical databases as sources of high-quality bibliographical data.

The “Öppna bibliotek” database

The data source we are using is extracted from an early version of the Swedish Öppna bibliotek (open library) project database. Öppna bibliotek was initiated by the Stockholm public library (Anderson, 2010), with the purpose of gathering user-generated book-related content such as tags, ratings and reviews, and sharing those among Swedish libraries. In practice data is collected in a relational database, and an API is defined so that different library systems can both insert and update data into the database as well as extract data from it.

Integrating the “Öppna bibliotek” database schema into our prototype

Database schemata are seldom fully normalized. Schemata may be de-normalized for performance purposes (e.g. introducing redundancies to enhance speed) or simply normalized only to fit certain contexts. This may create challenges when trying to integrate a database schema into a new context.

A brief inspection of the database dump we received revealed that the data structure represented some challenges for our approach such as:

- The table of all books and their corresponding titles also contained the first name and last name of the book’s author
- The ISBN field in the *edition* table contained many values that couldn’t possibly be interpreted as ISBNs

Table 1. Excerpt of the Original Data Structure

Table Book			
Book_ID	Title	Author_firstname	Author_lastname
0001	Hitchhikers guide to the galaxy	Douglas	Adams
0002	Semantic web programming	John	Hebeler

These and many other issues are recurring real life problems that solutions such as the one presented should be able to cope with. We therefore decided not to further normalize the database, but rather incorporate it “as is” into the prototype.

Conversion of relational databases to RDF

There are several different approaches to converting relational databases to RDF (W3C RDB2RDF Incubator Group 2009; Malhotra 2009). One approach is to export the database to XML, a feature provided by most database management systems, and subsequently transform the data to RDF/XML using XSLT. Another is using the database structure directly, which was the choice for the presented prototype.

Prototype components

The Jena framework

Our Java program utilizes an open source framework called *Jena* to create and manipulate RDF. Jena provides interfaces to represent RDF *graphs* (called *models*), *statements*, *resources*, *properties* and *literals*. It provides both in-memory and persistent storage, and can read and write RDF in several different formats. Among many other features, Jena has a built-in query engine called *ARQ*, which supports execution of *SPARQL queries*. Jena can be integrated with other Semantic Web tools such as Pellet, Protegé and Virtuoso in order to optimize performance and/or scalability. For a more thorough introduction to the Jena framework, please refer to the tutorial by McBride (2009) or the documentation available from http://jena.sourceforge.net.

SPARQL

SPARQL is a recursive acronym for ***S****PARQL* ***P****rotocol* ***a****nd* ***R****DF* ***Q****uery* ***L****anguage*. Being an official W3C recommendation (W3C 2008), it is supported by many different Semantic Web tools and platforms. SPARQL queries are based on pattern matching, and have a basic structure similar to SQL. There are four different query types; *select*, *construct*, *ask* and *describe*, of which the first two are used in our Java program. Whereas a SPARQL *select* query (normally) returns a *table* of results, a SPARQL *construct* query returns an *RDF graph*.

The D2RQ platform

D2RQ is a platform that can be used to extract and convert data from relational databases (RDBs) to RDF directly, without first “dumping” the database as XML (Bizer et al. 2009). It can be used either in a fully automated or semi-automated manner.

Fully automated conversion

In the fully automated mode the D2RQ shell script *dump-rdf* launches a program that extracts and converts the whole database in one step.

Calling the dump-rdf script from the command line:

```
sh dump-rdf -u username -p password -d com.mysql.jdbc.Driver -b http://openlibraryproject.no/
-j jdbc:mysql://localhost:3333/obtest_development -o dump.n3
```

The program first retrieves information about the data structure of the source database. By studying the log file from the database server, this appears as a series of *SQL-requests*. Based on this information, the program generates a default *mapping* from the original data structure to RDF *classes* and *properties*. Each *table* in the database is thus mapped to a *class*, while each *column* of the table is mapped to a *property*.

Example class generated by D2RQ:

```
<http://localhost/vocab/book>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/2000/01/rdf-schema#Class> .
<http://localhost/vocab/book> <http://www.w3.org/2000/01/rdf-schema#label>     "book" .
```

Example property generated by D2RQ:

```
<http://localhost/vocab/book_title>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#Property> .
<http://localhost/vocab/book_title>
<http://www.w3.org/2000/01/rdf-schema#label> "book title" .
```

The *data* is then extracted and converted by creating a unique RDF *resource* for each *record* of each table. Each *resource* is defined as an *instance* of the appropriate *class*. Each *field* in the record is then mapped to an RDF *statement*, where the *resource* is the *subject* of the statement, the *property* corresponding to the *column* is the *predicate*, and the *value* from the database field is the *object* of the statement. Values may be mapped to either *literals* or, if the foreign keys in the source database are properly defined, *other RDF resources*.

Example data triples generated by D2RQ:

```
<http://example.com/book/0001> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://localhost/vocab/book> .

<http://example.com/book/0001> <http://localhost/vocab/book_id> "0001"^^<http://www.w3.org/2001/XMLSchema#int> .

<http://example.com/book/0001> <http://www.w3.org/2000/01/rdf-schema#label> "book #0001" .

<http://example.com/book/0001> <http://localhost/vocab/book_title> "Hitchhikers guide to the galaxy" .

<http://example.com/book/0001> <http://localhost/vocab/book_authorlastname> "Adams" .

<http://example.com/book/0001> <http://localhost/vocab/book_authorfirstname> "Douglas" .
```

All *class* and *property* names are generated automatically based on the names of *tables* and *columns* in the source database. However, these names may not be very intuitive, and may thus be ill-suited for re-use as linked open data. Using this fully automatic method, we would miss the opportunity to map the data model to *established ontologies*, which is a crucial point in the Linked Data philosophy. Moreover, there is no selection of which data is extracted, so we may get a lot of irrelevant and bad quality data.

Semi-automated conversion

The *semi-automated* conversion strategy chosen for the prototype is based on another D2RQ shell script named *generate-mapping* and a customized Java program.

Calling the generate-mapping script from the command line:

```
sh generate-mapping -u username -p password  -o mapping.n3 jdbc:mysql://localhost:3333/obtest_development
```

The script runs a program that automatically generates a mapping file as in the automated conversion, but without extracting the actual *data*. The mapping file can then be used as input to a Java program (RDFGenerator), which builds a customized RDF graph using a *SPARQL construct query*. This method requires some manual effort for each new data source, but in return provides the opportunity to reuse established ontologies and select only the relevant and high quality data. The idea for the basic program structure was taken from an example in the textbook *Semantic Web Programming* by Hebeler et. al. (2009 chap. 9 p. 337-346).

Programmatically generating a graph of user-generated content

We developed a batch program in Java called RDFGenerator.java (http://journal.code4lib.org/media/issue17/holgersen/RDFGenerator.java.txt). The program generates an RDF graph of social data about books from the Öppna bibliotek relational database, and adds value to the data by increasing the number of access points through FRBRization and linking resources to equivalent resources in the "linked open data cloud." An illustration of the central components of the batch program can be seen in Figure 2.

The *RDFGenerator* class can be executed either via the *Menu* class, which provides a simple user interface, or separately, in which case the conversion from the relational database to RDF and the subsequent steps enriching the data model will be run successively until completion. If the user is interested in studying the results of any intermediate step, the program should be run via the menu class, which enables the user to select which steps to run. Here, we will explain the RDFGenerator as it is run without using the menu class.

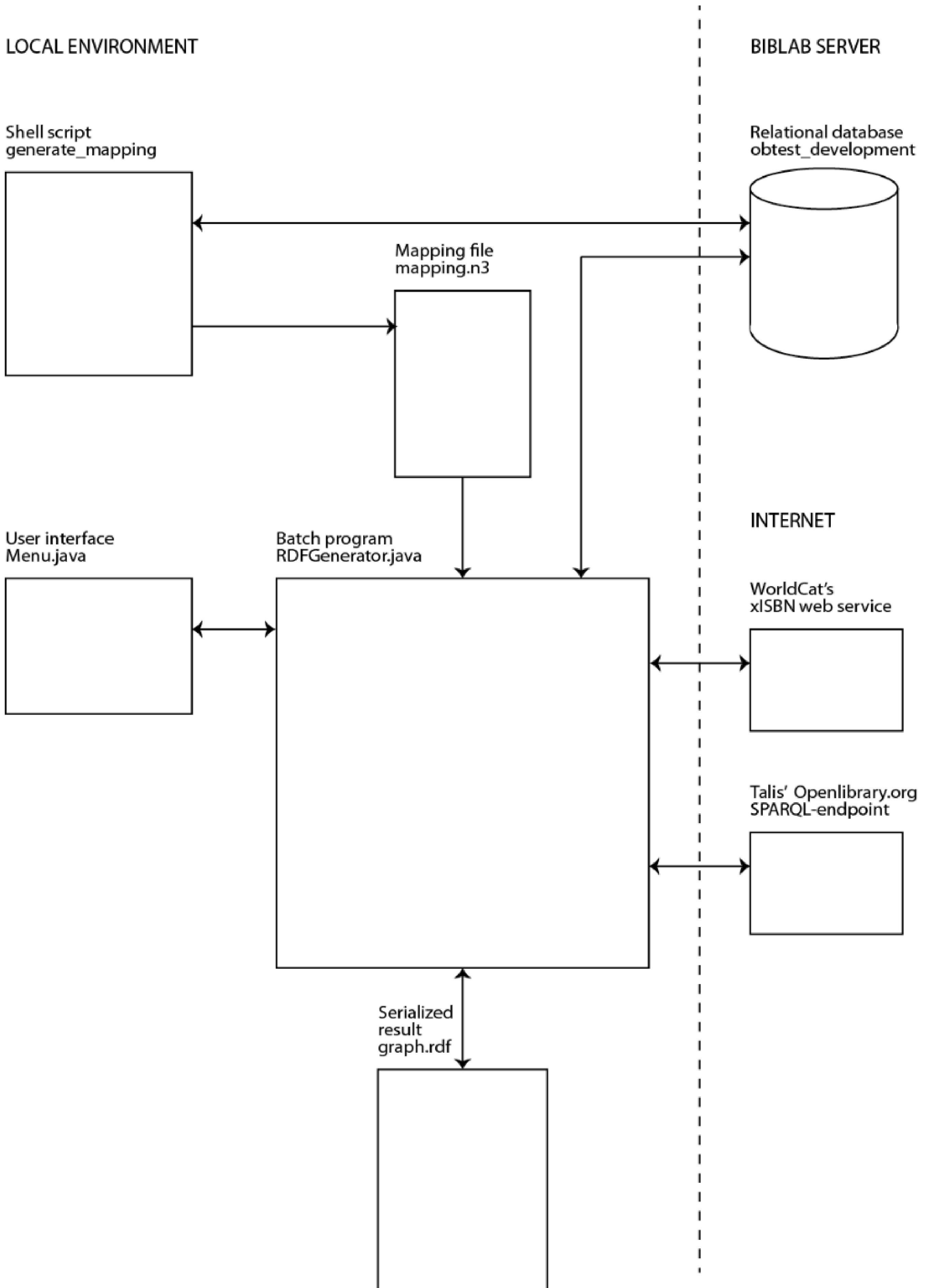

Figure 2. Central Components of the Batch Program

Command line parameters

The RDFGenerator can be called with command line parameters, as illustrated below, telling the program where to look for the SPARQL *query file*, the *name* of the query to be executed, where to look for the D2RQ *mapping file*, where to store the resulting RDF graph and which RDF format to use.

Calling RDFGenerator from the command line:

```
java RDFGenerator conf/queries.txt main conf/mapping.n3 graph.rdf RDF/XML
```

If the command line parameters are omitted, the program will run using *default parameters*. In the following, we will assume that input parameters have been submitted. All the information the program needs about the relational database to be converted is included in the auto-generated mapping file.

Importing data from the relational database

The method *importDataFromRDB()* in RDFGenerator starts by creating an object of the class *ModelD2RQ,* with the mapping file generated by the *generate-mapping* script as input. This object holds a virtual read-only graph representation of the whole relational database. Since it is the user-generated content (descriptions, assessments, ratings, tags etc.) we are after, rather than the bibliographical data, we run a *SPARQL construct* query against this graph, in order to extract the former.

The SPARQL construct query

A SPARQL construct query consists of three main blocks. The first block lists all the namespace prefixes used in the query, and maps them to the URIs of the ontologies and vocabularies they represent. The query itself consists of two blocks; a *construct clause*, which specifies the *graph template* of the resulting graph, and a *where clause*, which binds variables in the construct clause by specifying a *graph pattern* to be matched against the graph being queried.

In the construct query used to extract data from the D2RQ model, the graph template only recreates selected fragments of the original Öppna bibliotek data model. That is, most bibliographical data and all administrative data are excluded. The graph pattern only matches books and users who have *at least one* associated description, review or tag.

The SPARQL CONSTRUCT query:

```
prefix olp:
prefix dc:
prefix frbr:
prefix foaf:
prefix bibo:
prefix rev:
prefix tag:
prefix rdf:
prefix rdfs:
prefix vocab:

CONSTRUCT
{

        ?work a frbr:Work ;
           dc:title ?title ;
           frbr:realization ?edition ;
         rev:hasReview ?bookReview .

        ?edition a frbr:Expression ;
                bibo:isbn ?isbn ;
                frbr:realizationOf ?work ;
                rev:hasReview ?editionReview ;
                olp:hasTagging ?tagging ;
                olp:hasDescription ?description .

        ?user a foaf:Person ;
                foaf:givenName ?userGivenName ;
                foaf:familyName ?userLastName ;
                olp:hasWritten ?description ;
                olp:hasWritten ?review ;
                olp:hasMadeTagging ?tagging .

        ?tagging a tag:Tagging ;
                tag:taggedResource ?edition ;
                tag:taggedBy ?user ;
                tag:associatedTag ?tag .

        ?tag a tag:Tag ;
                olp:associatedTagging ?tagging ;
                tag:name ?tagName .

        ?description a olp:Description ;
                olp:describes ?edition ;
                olp:hasText ?descriptionText ;
                olp:writtenBy ?user .

        ?bookReview a rev:Review ;
                rev:rating ?rating ;
                rev:reviewer ?user ;
                olp:reviews ?work ;
                rev:title ?reviewHeader ;
                rev:text ?reviewText .

        ?editionReview a rev:Review ;
                rev:rating ?rating ;
                rev:reviewer ?user ;
                olp:reviews ?edition ;
                rev:title ?reviewHeader ;
                rev:text ?reviewText .

}

WHERE
{

        OPTIONAL
        {
                ?tagging
                        vocab:taggings_tag_id ?tagId ;
                        vocab:taggings_user_id ?userId ;

                        {
                        ?tagging vocab:taggings_book_id ?workId .
                        }
                        UNION
                        {
                        ?tagging vocab:taggings_edition_id ?editionId .
                        }

                ?tag
                        vocab:tags_id ?tagId ;
                        vocab:tags_name ?tagName .
        }

        OPTIONAL
        {
                ?description vocab:descriptions_edition_id ?editionId ;
                        vocab:descriptions_text ?descriptionText .
```

```
                OPTIONAL
        {
                        ?description vocab:descriptions_user_id ?userId .
                }
    }

    OPTIONAL
    {
            ?editionReview vocab:assessments_id ?assessmentId ;
                    vocab:assessments_grade ?rating ;
                    vocab:assessments_user_id ?userId ;
                    vocab:assessments_edition_id ?editionId ;
                    vocab:assessments_comment_header ?reviewHeader ;
                    vocab:assessments_comment_text ?reviewText .
    }

    OPTIONAL
    {
            ?bookReview vocab:assessments_id ?assessmentId ;
                    vocab:assessments_grade ?rating ;
                    vocab:assessments_user_id ?userId ;
                    vocab:assessments_book_id ?workId ;
                    vocab:assessments_comment_header ?reviewHeader ;
                    vocab:assessments_comment_text ?reviewText .
                    FILTER (!bound(?editionReview))
    }

    OPTIONAL
    {
            ?user vocab:users_id ?userId .

                    OPTIONAL
                    {
                            ?user vocab:users_firstname ?userGivenName ;
                                    vocab:users_lastname ?userLastName .
                    }
    }

    OPTIONAL
    {
            ?edition
                    vocab:editions_isbn ?isbn ;
                    vocab:editions_id ?editionId ;
                    vocab:editions_book_id ?workId   .
    }

    {
            ?work vocab:books_id ?workId ;
               vocab:books_title ?title  .
    }

    FILTER  (regex(?isbn, "^([0-9X][- /.]?){10,10}$", "i") ||
            regex(?isbn, "^([0-9X][- /.]?){13,13}$", "i") )
}
```

Filtering the results

In a Linked Data setting, it is crucial that all resources are *identifiable*; otherwise any kind of inter-linking or re-use would be very difficult. We therefore decided to discard all book editions for which it was impossible to obtain a (potentially) valid ISBN. This was implemented by including a *filter* in the *where* clause of the construct query, containing a *regular expression*. The filter is based on the simplified assumption that, given this context, if the ISBN candidate contains the correct number of digits, it is likely to be a valid ISBN. The regular expression filters out any ISBNs with too few or too many digits.

Limiting the aspects of the original data model recreated by the graph template, in conjunction with reducing the number of graph solutions by applying a strict graph pattern, greatly reduces the size of the resulting graph.

Mapping the data to established ontologies and vocabularies

In the Linked Data paradigm, reusing established ontologies and vocabularies is emphasized, in order to facilitate querying across different data sets. The construct query replaces all the automatically generated classes and predicates used in the D2RQ model with classes and predicates from established ontologies and vocabularies. We used a specialized search engine named *Swoogle* (http://swoogle.umbc.edu/) to identify appropriate candidates. Since we were not able to find a good match for all the aspects of the domain we wanted to model, we had to develop our own limited ontology as well, covering the “holes” in the established ontologies and vocabularies.

Querying the D2RQ model

An object of the class *QueryReader* is created, which is responsible for fetching the SPARQL queries, based on the name of the file where the queries are stored and the name of the desired query within the file. The QueryReader class originates from the textbook example (Hebeler et.al 2009 chap. 9 p. 337-346), and is reused without any modifications on our part.

The Jena classes *QueryFactory* and *QueryExecution* are used to generate a *Query* object from the query string, and execute the query against the D2RQ model. The resulting subgraph is stored in a Jena *Model* object, used throughout the rest of the program.

Serializing the model

The method *writeModelToFile()* is used to serialize the model to a file in the preferred RDF format, using Jena’s built-in methods.

Adapting the graph to the FRBR model

As we mentioned earlier, it can be difficult to find established ontologies and vocabularies that fit the data model well. The original “Öppna bibliotek” data model contains a table of *books* and a table of *editions*. The *book table* contains a *primary key* uniquely identifying each *book record*, and information about the book’s

title and *author*. The *edition table* contains a *primary key* uniquely identifying each *edition record*, a *foreign key* identifying each edition's associated *book*, and an optional *ISBN* field. The edition table is used for other media types as well as books, and thus contains several other optional fields. The tables containing user-generated content (*assessments*, *descriptions* and *tags*) reference either *books* or *editions* (or both) as *foreign keys*.

Rationale for modeling with FRBR

It is assumed that user-generated content might have relevance across different editions (expressions / manifestations, see below) of the work represented by a book. If a library user searches for a book and the library owns and presents information about a specific edition, the user still *might* find social content added about *other editions* of the same book relevant. In order to be able to identify and aggregate data about different editions of each book, we found it reasonable to use the *FRBR vocabulary* (Davis & Newman 2009).

FRBR is an acronym for **F**unctional **R**equirements for **B**ibliographical **R**ecords, and is a relational model that can be used to model the book domain. The RDF implementation of the model contains classes representing *works*, *expressions*, *manifestations* and *items*. For a thorough introduction to the FRBR model, please refer to the *Final report* by the IFLA Study Group on Functional Requirements for Bibliographical Records (2009).

Unfortunately, the "Öppna bibliotek" data model didn't correspond perfectly to the FRBR model, so we had to make a work-around.

Mismatch between "Öppna bibliotek" data model and the FRBR model

The *book* entity in the database matched well with the *work* class. However, we were unsure which FRBR class corresponded best to the database's *edition* entity, since the edition table contains data that may be connected to either the *expression* or the *manifestation* level of the FRBR model. It was important for us to be able to identify *alternative editions* of each work, and connect the resources to equivalent resources in other graphs in the linked open data cloud. The ISBN would play a critical role in this process, being a broadly adopted ISO standard. Since the ISBN is generally associated with the *manifestation* level of the FRBR model, and we didn't intend to keep the bibliographical information that might be relevant to the expression level in any case, we decided to map the *edition* entities to the *manifestation* class.

The RDF representation of the FRBR model (http://vocab.org/frbr/core.html) defines relationships between *work* and *expression*, and between *expression* and *manifestation*, but not directly between *work* and *manifestation*. Moreover, the four classes are disjoint, meaning that a resource can't be assigned to more than one of these classes.

To work around this problem, we decided to let the SPARQL construct query executed inside the *importDataFromRDB()* method make a provisional mapping, where *editions* are temporarily mapped to the *expression* class, and the *ISBN* and user-generated content are temporarily linked to the *expression* level instead of the *manifestation* level.

After the provisional model is made, a method named *adaptDataToFRBROntology()* is called. This method adds a new instance of the *manifestation* class for each *expression* resource in the Jena model. The resource URIs are built on the same URI pattern as the resources generated by the D2RQ mapping [1], but using *ISBN* instead of a *primary key* to achieve uniqueness of each resource identifier.

Example resources illustrating the URI patterns:

```
http://example.com/resource/work/32411

http://example.com/resource/expression/34265

http://example.com/resource/manifestation/8205329184
```

All statements linking ISBNs and user-generated content to the *expression* level are replaced by statements linking the same literals and resources to the *manifestation* level.

Since it's not possible to both iterate a Jena model and make changes to it at the same time, all the new statements are temporarily stored in a buffer model. The buffer model is *added* to the main model before the *adaptDataToFRBROntology()* method finishes. The outdated statements linking user-generated content to the expression-level are similarly added to another temporary model, which is eventually *subtracted* from the main model.

Adding alternative manifestations

WorldCat offers a RESTful web service called xISBN that, given a specific ISBN, returns a list of associated ISBNs. The web service is based on the result of an FRBRization algorithm developed by OCLC, and is an excellent tool to obtain information about alternative manifestations of a work. (WorldCat n.d.)

We use the xISBN web service in order to extend the RDF graph, so that libraries wanting to enrich their library catalogue using the service need not rely on holding the exact same version of the book as a user has provided information about, in order to retrieve useful information.

The method *addAlternativeManifestationsOfEachExpression()* performs a SPARQL select query in order to identify all *manifestation resources* in in the Jena model and their related *ISBN values*. For each result, a request to the xISBN web service is made. The web service returns an XML document containing a list of related ISBNs. This document is parsed in order to fetch the ISBN values. For each new ISBN, a new manifestation resource is created and linked to the appropriate expression resource.

This process results in a huge increase in the model size, greatly extending the number of entry-points to retrieve user-generated content about the books. A drawback of this approach, however, is that there is a relatively low correspondence between the books included in the "Öppna bibliotek" database and the books covered by WorldCat's bibliographical database**.** This implies that some books will necessarily escape our attempt to FRBRize the RDF graph.

Intended Use of the Service

Libraries wanting to automatically enrich their OPAC with user-generated content can use our service to fetch content relevant to a certain book.

According to Linked Data principles, all resource URIs are supposed to return useful information in RDF when a HTTP request is made. We were not able to set up a proper SPARQL endpoint within the limited scope of this project, but we managed to make the data available through a simple REST-based web service (http://journal.code4lib.org/media/issue17/holgersen/WebServiceResource.java.txt) that returns RDF/XML. The overall structure of the web service is based on a tutorial by Vogel (2011).

Making our data available through a web service

Libraries wanting to enrich their OPAC with user-generated content can make calls to the web service using an ISBN as a parameter.

Example call to the web service [2]:

```
http://example.com/RestExample/isbn/8373910573
```

In the background, the web service makes a SPARQL construct query against the graph generated by the batch program, in order to create a *sub graph* of all user-generated content about all manifestations of all expressions of the requested work.

The sub graph is returned in RDF/XML, so that developers integrating the data may choose to treat it either as an RDF graph or parse it like any other XML.

The libraries may choose what kind of user-generated content they want to integrate in their OPAC (descriptions, reviews, ratings and/or tags) and whether they want to show information related to *any* manifestations of the work, or only the information related to the *requested* manifestation.

Participating in the LOD Cloud

Basing our service on RDF allows us to increase the value of our data by connecting to other data sets published openly as linked data. In order to facilitate re-use of our data, we chose to programmatically add links to equivalent resources in a central linked data source.

Openlibrary.org is a collaborative, open source project which aims to make one web page for each book ever published. The combination of universally unique resource identifiers for each book edition and the fact that metadata can be returned in RDF format makes this website an important player in the linked open data "cloud." That is, other RDF graphs are likely to link to equivalent resources in the Openlibrary.org graph. A buffered version of the Openlibrary.org graph can be queried through a SPARQL endpoint hosted by Talis (http://api.talis.com/stores/openlibrary/services/sparql).

The method *identifyEquivalentResourcesInExternalGraph()* starts by running a SPARQL select query in order to identify all *manifestation resources* and their corresponding *ISBNs* in the local graph. For each result, the method proceeds by running a remote SPARQL query through Talis' SPARQL endpoint in order to identify resources in the Openlibrary.org graph that share the ISBN in question. Every time a resource is returned, a new statement, claiming that the external resource is equivalent ("*same as*") to the local resource, is added to the local graph.

Again, there is the problem of low correspondence between the two data sets. However, since anyone is allowed to add new books to the Openlibrary.org dataset, this gap *may* be bridged over time.

Conclusion and Future Work

In this paper, we have described a simplified prototype of a collaborative service for storing and serving user-generated content that can be used to enrich libraries' OPACs, based on Semantic Web technologies.

The use of an external FRBRization service greatly increases the number of entry points available to retrieve user-generated content about books, based on ISBN. In the long run, we believe it would be beneficial for the actual library catalogues to adopt a relational model, but in the meantime, FRBRizing the value-adding information is a step in the right direction.

In order to turn the prototype into a working service, aspects such as scalability and buffering will need further consideration, and the data model should be extended in order to keep track of administrative information. Moreover, the data should be made available through a SPARQL endpoint.

In order to achieve a critical mass of social content, it would be necessary to add additional data sources. In order to do that, an extension of the program structure would be necessary. If a critical mass of user-generated content were achieved, it would be possible to add some simple reasoning functionality and calculation of average ratings, etc. By extending the model to keep track of individual users, it would also be possible to provide Amazon-like features such as "users who liked A also liked B," etc.

Not only will participating libraries gain access to a common repository of user-generated, social data, they will also be able to integrate other types of value adding content in their OPAC by utilizing the provided links to equivalent RDF resources in the LOD cloud.

Appendix: Code

Menu.java
RDFGenerator.java
Timer.java
queries
openlibraryproject_ontology.owl
GraphBeforeFRBRizing.n3
WebServiceResource.java
UsageResource.java
SubGraph.rdf
Graph.rdf
mapping.n3

Notes

[1] We changed the URI patterns in the D2RQ mapping file manually, so that the resource URIs generated by D2RQ reflect the chosen ontologies and vocabularies instead of the table names in the source database. That is, class names like *work* and *expression* are used instead of original table names. This is not strictly necessary, but makes the RDF data more comprehensible to humans.

[2] See SubGraph.rdf for an excerpt of the RDF/XML document returned.

References

Anderson, D. (2010). Öppna bibliotek. Retrieved May 29, 2011, from http://www.oppnabibliotek.se/

Berners-Lee, T. (1998). Relational Databases on the Semantic Web. Retrieved from http://www.w3.org/DesignIssues/RDB-RDF.html

Berners-Lee, T. (2006). Linked Data. Retrieved from http://www.w3.org/DesignIssues/LinkedData.html

Berners-Lee, T., Hendler, J. & Lassila, O. (2001). The Semantic Web. In *Scientific American Magazine*, May 2001. Retrieved from http://www.scientificamerican.com/article.cfm?id=the-semantic-web

Bizer, C., Cyganiak, R., Garbers, J., Maresch, O., Becker, C. (2009). The D2RQ Platform v0.7: Treating Non-RDF Relational Databases as Virtual RDF Graphs. [User Manual and Language Specification]. Retrieved May 29, 2011, from http://www4.wiwiss.fu-berlin.de/bizer/d2rq/spec/20090810/

Davis, I. & Newman, R. (2009). Expression of Core FRBR Concepts in RDF. Retrieved May 29, 2011, from http://purl.org/vocab/frbr/core#

Heath, T., & Bizer, C. (2011). *Linked Data: Evolving the Web into a Global Data Space* (1st edition). Morgan & Claypool Publishers. doi:10.2200/S00334ED1V01Y201102WBE001 (COinS)

Hebeler, J., Fisher, M., Blace, R. & Perez-Lopez, A. (2009). *Semantic Web Programming*. Indianapolis, Ind. : Wiley. (COinS)

IFLA Study Group on the Functional Requirements for Bibliographic Records. (2009). *Functional Requirements for Bibliographic Records: Final Report* (Rev. ed.). Retrieved from http://www.ifla.org/files/cataloguing/frbr/frbr_2008.pdf

Jena: A Semantic Web Framework for Java. (n.d.). Retrieved May 29, 2011, from http://jena.sourceforge.net/

Jena Framework. [Documentation]. (n.d.). Retrieved May 29, 2011, from http://jena.sourceforge.net/javadoc/

Make Yahoo! Web Service REST Calls [Tutorial]. (n.d.). Retrieved May 29, 2011, from http://developer.yahoo.com/java/howto-reqRestJava.html

Malhotra, A. (Ed.). (2009). *W3C RDB2RDF Incubator Group Report*. Retrieved from http://www.w3.org/2005/Incubator/rdb2rdf/XGR-rdb2rdf-20090126/

McBride, B. (2010). An Introduction to RDF and the Jena RDF API. Retrieved May 29, 2011, from http://jena.sourceforge.net/tutorial/RDF_API/index.html

Parse Yahoo! Web Service REST Calls. [Tutorial]. (n.d.). Retrieved May 29, 2011, from http://developer.yahoo.com/java/howto-parseRestJava.html

Segaran, T., Evans, C. & Taylor, J. (2009). *Programming the Semantic Web*. Sebastopol, Calif. : O'Reilly. (COinS)

Vogel, L. (2011). REST with Java (JAX-RS) Using Jersey: Tutorial (Version 1.1). Retrieved May 29, 2011, from http://www.vogella.de/articles/REST/article.html

W3C. (2004). *RDF Primer: W3C Recommendation.* Retrieved from http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

W3C. (2008). *SPARQL Query Language for RDF: W3C Recommendation.* Retrieved from http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/

W3C RDB2RDF Incubator Group. (2009). *A Survey of Current Approaches for Mapping of Relational Databases to RDF*. Retrieved from http://www.w3.org/2005/Incubator/rdb2rdf/RDB2RDF_SurveyReport.pdf

WorldCat. (n.d.). ISBN (Web service). Retrieved May 29, 2011, from http://www.worldcat.org/affiliate/webservices/xisbn/app.jsp

About the Authors

Ragnhild Holgersen is a graduate student at the Department of Archivistics, Library and Information Science at Oslo and Akershus University College of Applied Sciences. Having a background in computer science and entrepreneurship, she is passionate about using insights gained within the LIS field in other contexts where people are striving to meet their information needs. She is currently working on her master thesis, using semantic web technologies to aid people with multiple food allergies in their search for safe food.

Michael Preminger is an Associate Professor at the Oslo and Akershus University College of Applied Sciences. He has Masters degree in IT and a PhD in Information Retrieval, and has been doing research and teaching within topics that combine library- and information science and information technology.

David Massey is an Assistant Professor at the Oslo and Akershus University College of Applied Sciences. He was the system librarian at the Oslo Public Library and a system consultant at a leading Norwegian library technology company before moving to the Department of Archivists, Library and Information Science.

Subscribe to comments: For this article | For all articles

This work is licensed under a Creative Commons Attribution 3.0 United States License.