An XML Pipeline-Based System Architecture for Managing Bibliographic Metadata

Johannes Textor[1] and Benjamin Feldner[2]

[1] Institut für Theoretische Informatik
Universität zu Lübeck, Ratzeburger Allee 160, 23538 Lübeck, Germany
textor@tcs.uni-luebeck.de
[2] Institut für Multimediale und Interaktive Systeme
Universisät zu Lübeck, Ratzeburger Allee 160, 23538 Lübeck, Germany
feldner@imis.uni-luebeck.de

Abstract. In our knowledge-based economy, bibliographic metadata is everywhere: Companies, research institutes, schools, museums, and many other institutions nowadays have to deal with large quantities of bibliographic information. Although several established standards are available for such data, home-grown ad hoc solutions are still widespread in small to medium enterprises. This paper presents a framework for storing, indexing, and browsing bibliographic metadata that is designed to lower the barrier for adoption of metadata standards by facilitating import of legacy data and integration into existing environments. This goal is achieved using XML pipelines as a central design paradigm. As a practical use case, we describe how the system was implemented at a research institute in our university, where it is used for managing publication lists and the local library.

1 Introduction

There are twelve institutes in the computer science section of our university, all of which operate their own web sites. Among other things, all of these websites contain lists of publications. Four institutes use Typo3 modules to manage these lists, and one uses a Drupal module; another one uses bibtex2html, and the remaining six rely on different home-grown solutions. These numbers might not be statistically significant, but they illustrate that there is no standard solution to a standard problem faced by, at least, all research institutions: managing bibliographic metadata. Certainly, this is not caused by a lack of choice: Several well-established standards for bibliographic metadata are available – MARC 21, RIS, EndNote, and BibTeX are just a few of them. Plenty of software uses or at least supports these formats, much of it Open Source: from reference management systems like RefDB (refdb.sourceforge.net) to whole integrated library systems such as Koha (koha.org).

One reason might be the abundance of field-specific bibliographic conventions: Every discipline, every journal has certain formatting rules for literature

references, and software like BibTeX that is capable of following all these conventions is often very complex – not many people are able to write custom BibTeX styles, for instance. Another important problem is legacy data: If an institution's library, consisting of several thousands of books and journals, has been managed for several years by the secretary using a Microsoft Excel document, an existing software product is unlikely able to import this data "out of the box" – clearly, this raises the barrier of adoption. Hence, one might decide to just continue using the Excel file, or implementing an custom solution.

In other words, two important factors that inhibit the spread of bibliographic metadata standards in small to medium enterprises are: (1) a lack of user-friendly and easily customizable data im- and export facilities in current software, resulting in (2) a prohibitively high cost of adoption. In this paper, we describe the design and implementation of a bibliographic metadata management system with the following main design goal: making import of legacy data and integration into existing environments as easy as possible. We demonstrate how this goal is achieved using an XML-centric approach, based on the Dublin Core and OpenURL standards (Sec. 2). Next, we introduce the system architecture (Sec. 3), which is built around XML transformations and pipelines as provided by the framework Apache Cocoon. The pipeline pattern is the most important architectural aspect of our solution, since it makes it easy to adapt the system talk to existing data sources and presentation targets. Finally, we describe the steps that were necessary to put the system into productive use in our own institution (Sec. 4), and conclude with perspectives for further research.

2 XML metadata format

We adopt the *Dublin Core* (DC) Metadata Initiative's metadata element set [1] as our main semantic framework. The main reasons not to use a more specialized format like EndNote or the DBLP XML format [2] are the DC's ease of use and its broader scope. However, it is not straightforward to describe complex bibliographic objects like articles in conference proceedings directly in DC. There was a Citation Working Group formed at the DCMI to address this problem, which in 2006 published a concluding guideline for encoding citation information that recommends to embed third-party formats within a refined DC *description* element [3]. Following this recommendation, we use the *OpenURL* framework standardized by ANSI/NISO [4] to encode citation information of more complex resources.

As is common practice, we encode the DC terms using RDF/XML syntax. However, because the XML documents are used for communication between our system's internal components, we simplify the syntax by omitting all namespace declarations. This considerably eases writing stylesheets to transform the data and saves processing time. For data export, an output filter that adds the correct namespaces can be implemented (see below).

Every DC element can carry the attributes *type* to encode DC element refinements (e.g. the subtype *created* of the *date* element [5]), *scheme* to define

content encoding schemes (e.g. *HTML* for the *description* element or *ISBN-13* for *identifier*), and *lang* to indicate the content language of elements such as *title* and *rights*. Each element can be omitted or used several times. As recommended by the DCMI, a controlled vocabulary is used for the Element *type* (Fig. 1).

```
<RDF>
  <description about="235">
    <creator type="AUTHOR">Johannes Textor</creator>
    <creator type="AUTHOR">Juergen Westermann</creator>
    <date type="PUBLISHED" scheme="W3C_DTF">2007</date>
    <description type="BIBLIOGRAPHIC_CITATION"
        scheme="OPEN_URL">
      <proceeding>
        <volume>4628</volume>
        <series>Lecture Notes in Computer Science</series>
        <btitle>6th International Conference on Artificial
            Immune Systems (ICARIS 2007)</btitle>
        <pages>228-239</pages>
      </proceeding>
    </description>
    <identifier
        scheme="URI">http://www.springerlink.com/content/
        [snip]</identifier>
    <publisher>Springer</publisher>
    <title lang="en_EN">Modeling Migration,
        Compartmentalization, and Exit of Naive T Cells in
        Lymph Nodes Without Chemotaxis</title>
    <type>IN_PROCEEDINGS</type>
  </description>
</RDF>
```

Fig. 1. Description of an article in a conference proceedings in our simplified XML syntax. Authors, date of publication, URL, publisher, title, and type are directly expressed in terms of Dublin Core elements, while the information about the proceedings volume and the page numbers are encoded using OpenURL.

3 System architecture

The central design paradigm of our architecture is the XML pipeline as implemented in *Cocoon* [6], the Apache Software Foundation's Java-based XML presentation framework. Cocoon applications are built from a toolbox of reusable pipeline components that can be plugged together in a Lego-like approach: Generators, transformers, and serializers (Fig. 2). Data is sent through the pipeline

in XML format, although the XML is not actually serialized within the pipeline, but kept in form of SAX events for performance reasons. This makes it somewhat cumbersome to write custom generators and serializers, since the SAX API is more optimized for efficiency than for ease of use – however, a more programmer-friendly API such as DOM would cause performance problems when processing very large amounts of data. On the other hand, transformers are very easy to write using the XSL stylesheet language. Cocoon's philosophy is to provide a large set of generators and serializers out of the box, such that end users themselves usually only write transformations. Accordingly, our system provides custom Cocoon components that take care of storing, indexing, searching, and retrieving XML resource descriptions in the format described above. Indexing and searching capabilities are provided by Lucene, a framework for building search engines [7]. Our components manage the Lucene index and keep it synchronized with the backend XML store (Fig. 3).

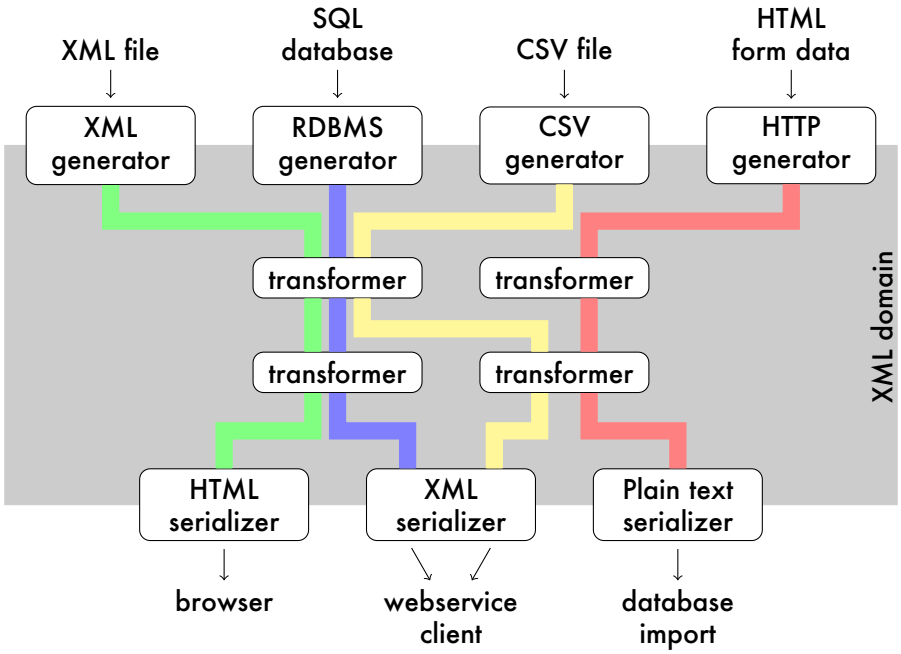

Fig. 2. Illustration of the Apache Cocoon framework: Data from different external sources is fed into XML pipelines by *generators* and subsequently modified, extended, and aggregated by *transformers*. Finally, a *serializer* converts the stream to an output format such as HTML or plain text and delivers the result to the client. Data is passed between the application's components in XML form (SAX events).

The task of the core system is to store and retrieve two simple kinds of objects: *Resources*, which are explicitly identified by their database IDs, and

lists, which are implicitly identified by a Lucene query (e.g. `creator:feldner AND date:[1990 TO 2005]`). To connect the system's XML-based core to the outside world, we need input and output adapters, called *importers* and *views*, respectively. Following Cocoon's philosophy, views are created by writing a custom XSL stylesheet that converts the simplified RDF/XML to the desired output format, and combining it with the corresponding serializer. Since existing pipelines can be used as input sources for new pipelines, one will typically not create a new view from scratch, but rather fine-tune an existing one by adding another stylesheet. For instance, our current prototype contains views for RDF/XML with namespaces, HTML, and BibTeX.

As mentioned at the beginning of this paper, importing bibliographic data from existing sources is a crucial function, even though it might well be used only once. Our approach is to split the import in two parts: A *syntactic* step, where the input data is converted to XML according to its internal syntactic structure; and a *semantic* step, where the elements of the resulting XML stream are rearranged, post-processed and converted to corresponding Dublin Core elements. While the first step is functionally trivial, it is usually hard to implement, as some seemingly simple formats like CSV and BibTeX are not well standardized and there are surprisingly many variants. The goal is to provide a large toolbox of Cocoon generators that can syntactically preprocess these common input formats out of the box. Once the data has been fed into an XML stream, the second step is again merely a matter of writing an XSL stylesheet, or adapting an existing one. Our prototype currently contains generators for CSV, RIS, and BibTeX.

The reason to use this two-step approach is that one cannot expect legacy data to follow certain semantics. This is obvious in the case of CSV tables, but even the semantics of more advanced formats such as BibTeX are often not clear at all (there is no definitive standard for the meaning of BibTeX keys). Hence, we cannot expect data import functions to work correctly in all cases. Our approach enables users to focus on getting the semantics right without having to deal with the parsing step.

4 Implementing the system in practice

To illustrate how the described concepts are applied in the real world, we describe the steps that were necessary to implement the system at our faculty's institute for theoretical computer science, where it is now used to manage the publication lists and the institute's internal library. For the publication lists, a view for HTML form-based editing of the database was first implemented. The view consists of some XSL stylesheets (for the forms themselves and a searchable list of existing publications) along with some static CSS and image files, which are stored in a subdirectory of the application's webapp folder. Some researchers had BibTeX files of their publications available, which were imported directly. To embed the bibliographic information into the institute's website, a view was written to generate HTML bibliographies, which was basically a small extension to the standard HTML fragment view to allow for both English and German

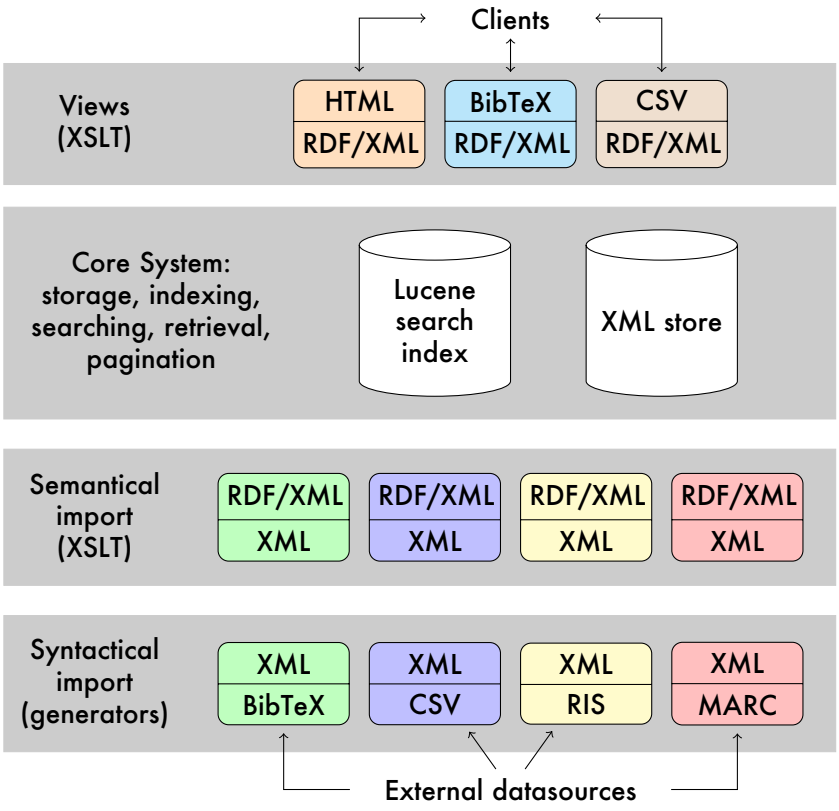

Fig. 3. Overview of the proposed system architecture. The core system uses RDF/XML as its only interface language, and is specialized on efficient storage, searching, and retrieval of documents in this format. Clients will usually not talk to directly to the core system, but rather use a corresponding view. Import of existing data sources is split in two stages: First, the data's internal structure is converted to XML (for instance, a CSV file results in a document containing *row* and *column* elements), which is then further processed by XSL stylesheets to produce semantically correct Dublin Core output.

output. The result can be seen at our website http://www.tcs.uni-luebeck.de, where the system feeds bibliographic information to several places.

Similarly, a custom view was created to allow browsing the local library. The metadata was imported from a legacy CSV file, which was piped through the XSL stylesheet shown in Fig. 4 upon syntactic conversion to an XML stream consisting of *line* and *column* elements

5 Conclusions

Many small and medium enterprises nowadays deal with large quantities of bibliographic metadata. Adoption of metadata standards and metadata management software, however, progresses at a rather slow pace – even though it yields unquestionable benefits, such as improved searching capabilities and data consistency, easier data exchange among institutions, and increased data lifetime. In this paper, we presented an architecture for an XML-based metadata management system that lowers the cost of adopting metadata standards by facilitating data import and integration into existing environments. We implemented the system and put it in productive use.

There are two main directions for future work. On one hand, a prerequisite for widespread use of the described system would be to hide the fairly complex Cocoon framework, which is known to have a rather steep learning curve, from the end user. Currently, at least a basic knowledge of Cocoon's internals such as how to configure and connect generators and serializers is necessary to write new views and importers. On the other hand, the system is currently tied to our specific RDF/XML implementation of Dublin Core. In principle, the architecture described here is applicable to any XML-based metadata format, and a generalization would broaden its application potential substantially.

References

1. Dublin Core Metadata Initiative Usage Bord: DCMI metadata terms. http://dublincore.org/documents/dcmi-terms/ (2008)
2. Ley, M.: The DBLP computer science bibliography: Evolution, research issues, perspectives. In: String Processing and Information Retrieval. Volume 2476 of Lecture Notes in Computer Science., Springer (2002) 481–486
3. Dublin Core Metadata Citation Working Group: Guidelines for encoding bibliographic citation information in dublin core metadata. http://dublincore.org/documents/dc-citation-guidelines/ (2005)
4. ANSI/NISO: The OpenURL framework for context-sensitive services. Standard Nr. Z39.88-2004 (2004)
5. Johnston, P.: Element refinement in Dublin Core metadata. http://dublincore.org/documents/dc-elem-refine// (2005)
6. Langham, M., Ziegeler, C.: Cocoon: Building XML Applications. Sams (2002)
7. Hatcher, E., Gospodnetic, O., McCandless, M.: Lucene in Action. Manning (2009)

```
<xsl:stylesheet version="2.0"
  exclude-result-prefixes="xs"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xsl:template match="csv">
    <RDF><xsl:apply-templates /></RDF>
  </xsl:template>
  <xsl:template match="line">
    <xsl:variable name="title" select="field[position()=3]"/>
    <xsl:if test="$title!=''">
      <description>
        <xsl:variable name="isbn" select="field[position()=1]"/>
        <xsl:variable name="authors" select="field[position()=2]"/>
        <xsl:variable name="publisher" select="field[position()=5]"/>
        <xsl:variable name="year" select="field[position()=6]"/>
        <xsl:variable name="signature" select="field[position()=8]"/>
        <xsl:if test="$isbn!=''">
          <identifier scheme="ISBN_10"><xsl:value-of
              select="$isbn"/></identifier>
        </xsl:if>
        <xsl:for-each select="tokenize($authors,'/')">
          <xsl:if test=".!=''">
            <creator type="AUTHOR"><xsl:value-of select="."/></creator>
          </xsl:if>
        </xsl:for-each>
        <title><xsl:value-of select="$title"/></title>
        <xsl:if test="$publisher!=''">
          <publisher><xsl:value-of select="$publisher"/></publisher>
        </xsl:if>
        <xsl:if test="$year castable as xs:integer?">
          <date type="PUBLISHED" scheme="W3C_DTF"><xsl:value-of
              select="$year"/></date>
        </xsl:if>
        <xsl:if test="$signature!=''">
          <identifier><xsl:value-of select="$signature"/></identifier>
        </xsl:if>
        <type>BOOK</type>
      </description>
    </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

Fig. 4. XSL stylesheet used for the semantic import of a custom CSV file, which was previously converted to an XML stream consisting of *line* and *column* elements by a Cocoon generator.