xmlprague

XML Prague 2013

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 8–10, 2013

XML Prague 2013 – Conference Proceedings
Copyright © 2013 Jiří Kosek

ISBN 978-80-260-3872-6

The Complete XML Development and Authoring Solution

XML Authoring

<oXygen/> XML Author is the most efficient solution for implementing single source publishing and content reuse.

XML Development

<oXygen/> XML Developer is specially tuned for XML development, providing the best coverage of today's XML technologies.

- **Visual XML Authoring**
- **Single Source Publishing**
- **Multiplatform Desktop and Web**
- **Highly Customizable**
- **Supported by Major CMSs**

- **Intelligent XML Editing**
- **Visual Schema Modeling**
- **XSLT and XQuery Debugging**
- **XML Databases Support**
- **Web Services Analyzer**

www.oxygenxml.com

Availability

<oXygen/> is available in three editions: **<oXygen/> XML Author** for content authors, starting from 349 USD, **<oXygen/> XML Developer** for XML developers, starting from 349 USD and **<oXygen/> XML Editor** for XML developers and content authors, starting from 488 USD.
For Academic/Non-Commercial use **<oXygen/> XML Editor** is available at a discounted price of 98 USD.
All editions can run as a standalone application or as an Eclipse IDE plugin, on Windows 8, Windows 7, Vista, XP, 2000, Mac OS X, Linux and Solaris.

ITS 2.0: Upcoming metadata & tools for Localization, LT, and the Multilingual Web

ITS = Internationalization Tag Set. See also
Showcases at MultilingualWeb workshop,
Rome 12-13 March
http://www.multilingualweb.eu/register

W3C MultilingualWeb-LT Working Group

Adobe, Baidu, CNR, Cocomore, DCU
DERI, EMI, Enlaso, DFKI, JSI
Linguaserve, Logrus, Lucy, Microsoft
Moravia, NCSR, Opera, SAP, TCD
UEP, UPM, UL, VistaTEC

Example: standard "translate" attribute In HTML5

```
<!doctype html><html>
... <p>Click the Resume button on the
Status Display or the
<span class="panelmsg"
translate="no">CONTINUE</span> button on
the printer panel.</p>
</body></html>
```

(Online) MT system,
human translation,
(hybrid) localization workflows, ...

```
<!doctype html><html>
... <p>Klicken Sie die Resume-Taste auf
der Status-Anzeige oder die
<span class="panelmsg"
translate="no">CONTINUE</span> Taste auf
dem Bedienfeld des Druckers.</p>
</body></html>
```

Reference Implementations

See use cases and tools:
http://tinyurl.com/its2-use-cases
Simple machine translation (MT), Translation package creation, Quality check, HTML5+ITS2 processing + validation, CMS<>TMS integration, online MT system, XLIFF+ CMS-LION and SOLAS, Text analysis annotation, Terminology enrichment, ... your implementation here ☺

Final: ITS 1.0

Translate, Localization Note, Terminology, Directionality, Ruby, Language Information, Elements within Text

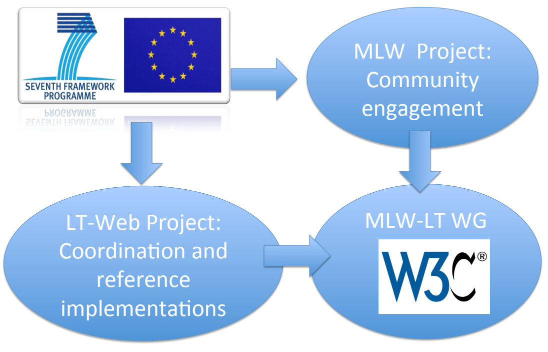
Proposed: ITS 2.0 new data categories

Domain, Disambiguation, Quality issues, Provenance, IdValue, Locale filter, TargetPointer, ...

We need your feedback!

GET INVOLVED!

- Provide comments about the ITS 2.0 draft http://www.w3.org/TR/its20/
- Join the Working Group
- Create reference implementations
- Attend upcoming workshops
- Timeline: Dec. 2013



SEVENTH FRAMEWORK PROGRAMME
MLW Project: Community engagement
LT-Web Project: Coordination and reference implementations
MLW-LT WG

MultilingualWeb
http://multilingualweb.eu

Node background by See-ming Lee 李思明
http://www.flickr.com/photos/seeminglee/2060090675/

Table of Contents

General Information

Date

Friday, February 8th, 2013 (preconference day)
Saturday, February 9th, 2013
Sunday, February 10th, 2013

Location

University of Economics, Prague (UEP) – Vencovského aula
nám. W. Churchilla 4, 130 67 Prague 3, Czech Republic

Organizing Committee

Petr Cimprich, *Xyleme*
James Fuller, *MarkLogic*
Vít Janota, *Xyleme*
Jirka Kosek, *xmlguru.cz & University of Economics, Prague*
Pavel Kroh, *pavel-kroh.cz & Macness.com*
Mohamed Zergaoui, *ShareXML.com & Innovimax*

Programm Committee

Robin Berjon, *freelance consultant*
Petr Cimprich, *Xyleme*
Daniela Florescu, *Oracle*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *University of Economics, Prague*
Uche Ogbuji, *Zepheira LLC*
Adam Retter, *freelance consultant*
Felix Sasaki, *DFKI / W3C Fellow*
John Snelson, *MarkLogic*
Eric van der Vlist, *Dyomedea*
Priscilla Walmsley, *Datypic*
Norman Walsh, *MarkLogic*
Mohamed Zergaoui, *Innovimax*

Produced By

XMLPrague.cz (http://xmlprague.cz)
Faculty of Informatics and Statistics, UEP (http://fis.vse.cz)
Ubiqway, s.r.o. (http://www.ubiqway.com)

Sponsors

Gold Sponsors

The FLWOR Foundation (http://www.flworfound.org)
Mark Logic Corporation (http://www.marklogic.com)

Sponsors

oXygen (http://www.oxygenxml.com)
Mercator IT Solutions Ltd (http://www.mercatorit.com)
MultilingualWeb-LT (http://www.w3.org/International/multilingualweb/lt/)
eXist Solutions GmbH (http://www.existsolutions.com)

Preface

This publication contains papers presented during the XML Prague 2013 conference.

In its eighth year, XML Prague is a conference on XML for developers, markup geeks, information managers, and students. XML Prague focuses on new advances in XQuery, digital books and publishing toolchains. The conference provides an overview of successful XML technologies, with a focus on real world application versus theoretical exposition.

The conference takes place 8–10 February 2013 at the campus of University of Economics in Prague. XML Prague 2013 is jointly organized by the XML Prague Organizing Committee and by the Faculty of Informatics and Statistics, University of Economics in Prague.

The full program of the conference is broadcasted over the Internet (see http://xmlprague.cz)—allowing XML fans, from around the world, to participate on-line.

In the previous year we moved to a new venue and added pre-conference day which provides space for various XML community meetings in three parallel tracks. Both changes were well received by attendees so we will continue with this tradition going into the future. Additionally, we coordinate, support and provide space for W3C XSLT and XQuery working group meetings collocated with XML Prague.

Last but not least—XML Recommendation was published on February 10th, 1998 and on the second conference day we will celebrate XML's 15th anniversary. We wish at least another fifteen years of success to XML.

We hope that you enjoy XML Prague 2013.

— Petr Cimprich & Jirka Kosek & Mohamed Zergaoui
XML Prague Organizing Committee

Multi-user interaction using client-side XSLT

O'Neil Delpratt
Saxonica
`<oneil@saxonica.com>`

Michael Kay
Saxonica
`<mike@saxonica.com>`

Abstract

We describe two use-case applications to illustrate the capabilities of the first XSLT 2.0 processor designed to run within web browsers. The first is a technical documentation application, which permits browsing and searching in a intuitive way. We then present a multi-player chess game application; using the same XSLT 2.0 processor as the first application, it is in fact very different in purpose and design in that we provide multi-user interaction on the GUI and implement communication via a social media network: namely Twitter.

1. Introduction

One of the original aims of XSLT was that it should be possible to use the language to convert XML documents to HTML for rendering on the browser. This aim has largely been achieved, but it took a long time before XSLT processors with a sufficient level of conformance and performance were available across all common browsers; and while that was happening, the landscape changed. It changed in several ways:

- XSLT 2.0 came along, raising the level of capability of the language and making the limitations of XSLT 1.0 even more obvious
- XML processing in the browser went out of fashion, in good measure because of the absence of universal XSLT support (and because, for users writing in Javascript, handling JSON was a lot easier)
- Microsoft's near-monopoly on browser installations was broken, with the result that web applications could only adopt new technologies when there was consensus to implement them across all the browsers; this added further to the disinclination of browser vendors to improve the level of XML support
- Web 2.0 came along: the web was no longer about producing pretty renditions of static documents, but about generating interactive applications. So XSLT as

originally conceived was only capable of doing half the job. Why would anyone want to use a mix of two languages (XSLT and Javascript) when Javascript could tackle it all?

- Mobile devices became as common a client platform as the traditional desktop, increasing the need for content repurposing to meet different device capabilities

Javascript has become a mature and powerful language, and there are good reasons for its popularity. However, in the areas where XSLT is strong, Javascript is at its weakest. Simple document transformation tasks, like sorting or grouping the rows of a table, or generating a table of contents, are painfully tortuous. So there are good reasons for feeling that the user community has a right to expect something better; developers writing web applications where document manipulation plays a significant role are currently using tools that deliver very poor productivity.

But if XSLT is to be viable in this space, it needs to be able to do far more than simple XML to HTML conversion; it also needs to handle the user interaction, and the other tasks that a modern web application is expected to perform, such as "behind-the-scenes" interaction with the server (still sometimes known by the inappropriate name of AJAX).

Although the rule-based processing model of XSLT was designed primarily with document rendition in mind (it lends itself well to handling documents with unpredictable or variable structure), it turns out that the same model is well suited to handling the other asynchronous and unpredictable tasks that arise in a web application. The way in which XSLT "push-mode" stylesheets are written to handle events from the XML parser is not at all dissimilar to the kind of event-based programming used in many graphical user interface toolkits.

This talk will examine how the first implementation of XSLT 2.0 on the browser, Saxon-CE, addresses this opportunity. This will be done by demonstrating example applications in which it is deployed.

We will look in particular at two applications.

The first is an application for browsing and searching technical documentation. This is classic XSLT territory, and the requirement is traditionally met by server-side HTML generation, either in advance at publishing time, or on demand through servlets or equivalent server-side processing that invoke XSLT transformations, perhaps with some caching. While this is good enough for many purposes, it falls short of what users had already learned to expect from desktop help systems, most obviously in the absence of a well-integrated search capability. Even this kind of application can benefit from Web 2.0 thinking, and we will show how the user experience can be improved my moving the XSLT processing to the client side and taking advantage of some of the new facilities to handle user interaction.

The second application is a classic Web 2.0 use case, an interactive multi-player game, where the communication between the players takes place over an underlying Twitter feed. Although the presentation of this application will no doubt be entertaining to the conference audience, the purpose is a serious educational one: the design of the application will be studied to show how real benefits are obtained by coding it in a high-level declarative language like XSLT, with a push-based event-driven approach at the heart of its processing model. It also gives an opportunity to examine some of the security issues associated with client-side processing, proxy authentication, cross-site scripting constraints, and the like.

XSLT 2.0 on the browser has been demonstrated at XML Prague in previous years, but only with very simple applications. Since last year's conference, Saxon-CE has become a fully-released product, and its capabilities have considerably increased. In particular, the interaction with the rest of the browser environment has been greatly strengthened, and these demonstrations will illustrate what can be achieved by taking advantage of these new capabilities.

2. Browsing and searching technical documentation

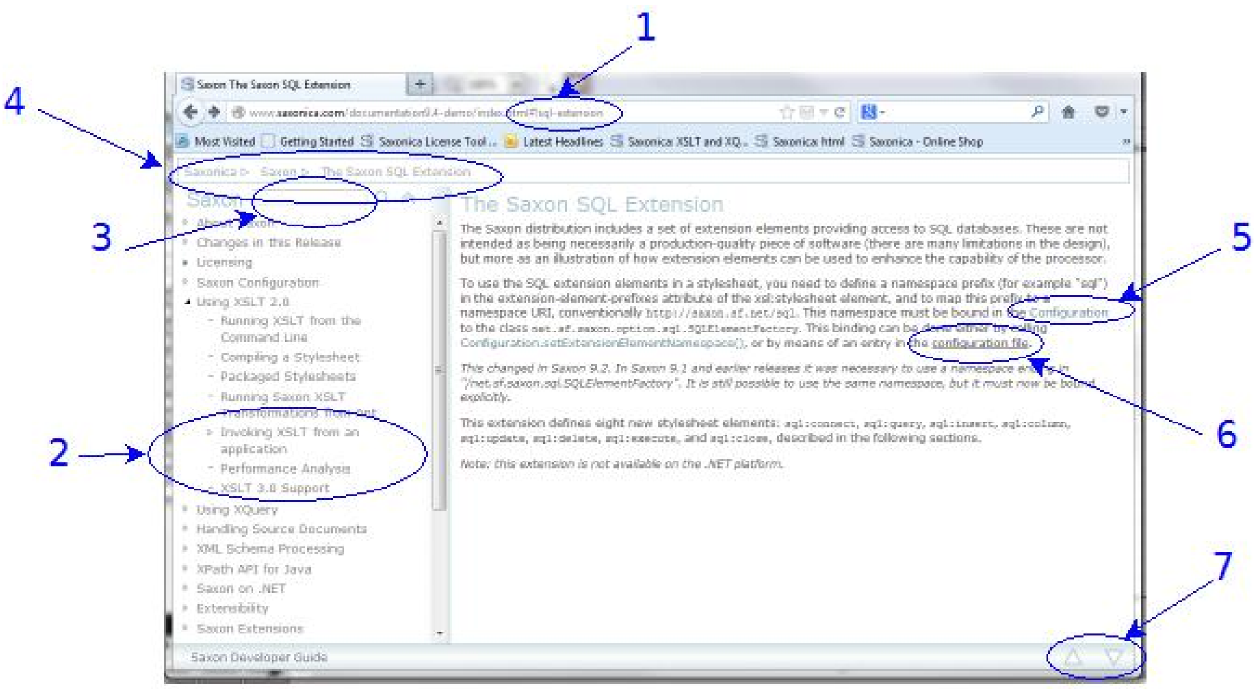
The first application we examine is an application for browsing technical documentation: specifically, it is used for display the Saxon 9.4 documentation on the Saxonica web site. (It is described as a demonstration, but it has been running successfully for several months, and we intend to cut over to this as the primary means of displaying the documentation at the next release.)

The documentation for Saxon 9.4 can be found at:

- http://www.saxonica.com/documentation9.4-demo/index.html

When you click on this link for the first time, there will be a delay of a few seconds, with a comfort message telling you that Saxon is loading the documentation. This is not strictly accurate; what is actually happening is that Saxon-CE itself is being downloaded from the web site. This only happens once; thereafter it will be picked up from the browser cache. However, it is remarkable how fast this happens even the first time, considering that the browser is downloading the entire Saxon-CE product (800Kb of Javascript source code generated from around 100K lines of Java), compiling this, and then executing it before it can even start compiling and executing the XSLT code.

The documentation is presented in the form of a single-page web site. The screenshot in Figure 1 shows its appearance.



The Saxon SQL Extension

The Saxon distribution includes a set of extension elements providing access to SQL databases. These are not intended as being necessarily a production-quality piece of software (there are many limitations in the design), but more as an illustration of how extension elements can be used to enhance the capability of the processor.

To use the SQL extension elements in a stylesheet, you need to define a namespace prefix (for example "sql") in the extension-element-prefixes attribute of the xsl:stylesheet element, and to map this prefix to a namespace URI, conventionally `http://saxon.sf.net/sql`. This namespace must be bound in the Configuration to the class `net.sf.saxon.option.sql.SQLElementFactory`. This binding can be done either by calling Configuration.setExtensionElementNamespace(), or by means of an entry in the configuration file.

This changed in Saxon 9.2. In Saxon 9.1 and earlier releases it was necessary to use a namespace ending in "/net.sf.saxon.sql.SQLElementFactory". It is still possible to use the same namespace, but it must now be bound explicitly.

This extension defines eight new stylesheet elements: `sql:connect`, `sql:query`, `sql:insert`, `sql:column`, `sql:update`, `sql:delete`, `sql:execute`, and `sql:close`, described in the following sections.

Note: this extension is not available on the .NET platform.

Screen-shot of the Technical documentation in the browser using Saxon-CE

Figure 1. Technical documentation application in the browser

Note the following features, called out on the diagram. We will discuss below how these are implemented in Saxon-CE.

1. The fragment identifier in the URL
2. Table of contents
3. Search box
4. Breadcrumbs
5. Links to Javadoc definitions
6. Links to other pages in the documentation
7. The up/down buttons

2.1. XML on the Server

This application has no server-side logic; everything on the server is static content.

On the server, the content is held as a set of XML files. Because the content is fairly substantial (2Mb of XML, excluding the Javadoc, which is discussed later), it's not held as a single XML document, but as a set of a 20 or so documents, one per chapter. On initial loading, we load only the first chapter, plus a small index document listing the other chapters; subsequent chapters are fetched on demand, when first referenced, or when the user does a search.

Our first idea was to hold the XML in DocBook form, and use a customization of the DocBook stylesheets to present the content in Saxon-CE. This proved infeasible: the DocBook stylesheets are so large that downloading them and compiling them gave unacceptable performance. In fact, when we looked at the vocabulary we were actually using for the documentation, it needed only a tiny subset of what DocBook offered. We thought of defining a DocBook subset, but then we realised that all the elements we were using could be easily represented in HTML5 without any serious tag abuse (the content that appears in highlighted boxes, for example, is tagged as an `<aside>`). So the format we are using for the XML is in fact XHTML5. This has a couple of immediate benefits: it means we can use the HTML DOM in the browser to hold the information (rather than the XML DOM), and it means that every element in our source content has a default rendition in the browser, which in many cases (with a little help from CSS) is quite adequate for our purposes.

Although XHTML 5.0 is used for the narrative part of the documentation, more specialized formats are used for the parts that have more structure. In particular, there is an XML document containing a catalog of XPath functions (both standard W3C functions, and Saxon-specific extension functions) which is held in a custom XML vocabulary; and the documentation also includes full Javadoc API specifications for the Saxon code base. This was produced from the Java source code using the standard Javadoc utility along with a custom "doclet" (user hook) causing it to generate XML rather than HTML. The Javadoc in XML format is then rendered by the client-side stylesheets in a similar way to the rest of the documentation, allowing functionality such as searching to be fully integrated.

The fact that XHTML is used as the delivered documentation format does not mean, of course, that the client-side stylesheet has no work to do. This will become clear when we look at the implementation of the various features of the user interaction.

For the most part, the content of the site is authored directly in the form in which it is held on the site, using an XML editor. The work carried out at publishing time consists largely of validation. There are a couple of exceptions to this: the Javadoc content is generated by a tool from the Java source code, and we also generate an HTML copy of the site as a fallback for use from devices that are not Javascript-enabled. There appears to be little call for this, however: the client-side Saxon-CE version of the site appears to give acceptable results to the vast majority of users, over a wide range of devices. Authoring the site in its final delivered format greatly simplifies the process of making quick corrections when errors are found, something we have generally not attempted to do in the past when republishing the site was a major undertaking.

2.2. Implementing the User Interface

This section discusses how the various aspects of the user interface are implemented. The implementation is done almost entirely in XSLT 2.0, with a few helper functions (amounting to about 50 lines) of Javascript. The XSLT is in 8 modules totalling around 2500 lines of code. The Javascript code is mainly concerned with scrolling a page to a selected position, which in turn is used mainly in support of the search function, discussed in more detail below.

2.2.1. The URI and Fragment Identifier

URIs follow the "hashbang" convention: a page might appear in the browser as:

- *http://www.saxonica.com/documentation9.4-demo/index.html#!configuration*

For some background on the hashbang convention, and an analysis of its benefits and drawbacks, see Jeni Tennison's article at [11]. From our point of view, the main characteristics are:

- Navigation within the site (that is, between pages of the Saxon documentation) doesn't require going back to the server on every click.
- Each sub-page of the site has a distinct URI that can be used externally; for example it can be bookmarked, it can be copied from the browser address bar into an email message, and so on. When a URI containing such a fragment identifier is loaded into the browser address bar, the containing HTML page is loaded, Saxon-CE is activated, and the stylesheet logic then ensures that the requested sub-page is displayed.
- It becomes possible to search within the site, without installing specialized software on the server.
- The hashbang convention is understood by search engines, allowing the content of a sub-page to be indexed and reflected in search results as if it were an ordinary static HTML page.

The XSLT stylesheet supports use of hashbang URIs in two main ways: when a URI is entered in the address bar, the stylesheet navigates to the selected sub-page; and when a sub-page is selected in any other way (for example by following a link or performing a search), the relevant hashbang URI is constructed and displayed in the address bar.

The fragment identifiers used for the Saxon documentation are hierarchic; an example is

- *#!schema-processing/validation-api/schema-jaxp*

The first component is the name of the chapter, and corresponds to the name of one of the XML files on the server, in this case `schema-processing.xml`. The subsequent components are the values of `id` attributes of nested XHTML 5 `<section>` elements within that XML file. Parsing the URI and finding the relevant subsection is therefore a simple task for the stylesheet.

2.2.2. The Table of Contents

The table of contents shown in the left-hand column of the browser screen is constructed automatically, and the currently displayed section is automatically expanded and contracted to show its subsections. Clicking on an entry in the table of contents causes the relevant content to appear in the right-hand section of the displayed page, and also causes the subsections of that section (if any) to appear in the table of contents. Further side-effects are that the URI displayed in the address bar changes, and the list of breadcrumbs is updated.

Some of this logic can be seen in the following template rule:

```
<xsl:template match="*" mode="handle-itemclick">
    <xsl:variable name="ids"
                  select="(., ancestor::li)/@id"
                  as="xs:string*"/>
    <xsl:variable name="new-hash"
                  select="string-join($ids, '/')"/>
    <xsl:variable name="isSpan"
                  select="@class eq 'item'"
                  as="xs:boolean"/>
    <xsl:for-each select="if ($isSpan) then .. else .">
        <xsl:choose>
            <xsl:when test="@class eq 'open' and not($isSpan)">
                <ixsl:set-attribute name="class" select="'closed'"/>
            </xsl:when>
            <xsl:otherwise>
                <xsl:sequence select="js:disableScroll()"/>
                <xsl:choose>
                    <xsl:when test="f:get-hash() eq $new-hash">
                        <xsl:variable name="new-class"
                                      select="f:get-open-class(@class)"/>
                        <ixsl:set-attribute name="class"
                                            select="$new-class"/>
                        <xsl:if test="empty(ul)">
                            <xsl:call-template name="process-hashchange"/>
                        </xsl:if>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:sequence select="f:set-hash($new-hash)"/>
                    </xsl:otherwise>
                </xsl:choose>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:for-each>
</xsl:template>
```

Most of this code is standard XSLT 2.0. A feature particular to Saxon-CE is the `ixsl:set-attribute` instruction, which modifies the value of an attribute in the HTML DOM. To preserve the functional nature of the XSLT language, this works in the same way as the XQuery Update Facility: changes are written to a pending update list, and updates on this list are applied to the HTML DOM at the end of a transformation phase. Each transformation phase therefore remains, to a degree, side-effect free. Like the `xsl:result-document` instruction, however, `ixsl:set-attribute` delivers no result and is executed only for its external effects; it therefore needs some special attention by the optimizer. In this example, which is not untypical, the instruction is used to change the `class` attribute of an element in the HTML DOM, which has the effect of changing its appearance on the screen.

The code invokes a function `f:set-hash` which looks like this:

```
<xsl:function name="f:set-hash">
   <xsl:param name="hash"/>
   <ixsl:set-property name="location.hash" select="concat('!',$hash)"/>
</xsl:function>
```

This has the side-effect of changing the contents of the `location.hash` property of the browser window, that is, the fragment identifier of the displayed URI. Changing this property also causes the browser to automatically update the browsing history, which means that the back and forward buttons in the browser do the right thing without any special effort by the application.

2.2.3. The Search Box

The search box provides a simple facility to search the entire documentation for keywords. Linguistically it is crude (there is no intelligent stemming or searching for synonyms or related terms), but nevertheless it can be highly effective. Again this is implemented entirely in client-side XSLT.

The initial event handling for a search request is performed by the following XSLT template rules:

```
<xsl:template match="p[@class eq 'search']" mode="ixsl:onclick">
    <xsl:if test="$usesclick">
        <xsl:call-template name="run-search"/>
     </xsl:if>
</xsl:template>

<xsl:template match="p[@class eq 'search']" mode="ixsl:ontouchend">
    <xsl:call-template name="run-search"/>
</xsl:template>
```

```
<xsl:template name="run-search">
    <xsl:variable name="text"
                  select="normalize-space(ixsl:get($navlist/div/input, ▶
'value'))"/>
    <xsl:if test="string-length($text) gt 0">
        <xsl:for-each select="$navlist/../div[@class eq 'found']">
            <ixsl:set-attribute name="style:display" select="'block'"/>
        </xsl:for-each>
        <xsl:result-document href="#findstatus" method="replace-content">
            searching...
        </xsl:result-document>
        <ixsl:schedule-action wait="16">
            <xsl:call-template name="check-text"/>
        </ixsl:schedule-action>
     </xsl:if>
</xsl:template>
```

The existence of two template rules, one responding to an `onclick` event, and one to `ontouchend`, is due to differences between browsers and devices; the Javascript event model, which Saxon-CE inherits, does not always abstract away all the details, and this is becoming particularly true as the variety of mobile devices increases.

The use of `ixsl:schedule-action` here is not so much to force a delay, as to cause the search to proceed asynchronously. This ensures that the browser remains responsive to user input while the search is in progress.

The template `check-text`, which is called from this code, performs various actions, one of which is to initiate the actual search. This is done by means of a recursive template, shown below, which returns a list of paths to locations containing the search term:

```
<xsl:template match="section|article" mode="check-text">
    <xsl:param name="search"/>
    <xsl:param name="path" as="xs:string" select="''"/>
    <xsl:variable name="newpath" select="concat($path, '/', @id)"/>
    <xsl:variable name="text" select="lower-case(
        string-join(*[not(local-name() = ('section','article'))],'!'))"/>
    <xsl:sequence select="if (contains($text, $search))
            then substring($newpath,2)
            else ()"/>
    <xsl:apply-templates mode="check-text" select="section|article">
        <xsl:with-param name="search" select="$search"/>
        <xsl:with-param name="path" select="$newpath"/>
     </xsl:apply-templates>
</xsl:template>
```

This list of paths is then used in various ways: the sections containing selected terms are highlighted in the table of contents, and a browsable list of hits is available, allowing the user to scroll through all the hits. Within the page text, search terms are highlighted, and the page scrolls automatically to a position where the hits are visible (this part of the logic is performed with the aid of small Javascript functions).

2.2.4. Breadcrumbs

In a horizontal bar above the table of contents and the current page display, the application displays a list of "breadcrumbs", representing the titles of the chapters/sections in the hierarchy of the current page. (The name derives from the story told by Jerome K. Jerome of how the *Three Men in a Boat* laid a trail of bread-crumbs to avoid getting lost in the Hampton Court maze; the idea is to help the user know how to get back to a known place.)

Maintaining this list is a very simple task for the stylesheet; whenever a new page is displayed, the list can be reconstructed by searching the ancestor sections and displaying their titles. Each entry in the breadcrumb list is a clickable link, which although it is displayed differently from other links, is processed in exactly the same way when a click event occurs.

2.2.5. Javadoc Definitions

As mentioned earlier, the Javadoc content is handled a little differently from the rest of the site.

This section actually accounts for the largest part of the content: some 11Mb, compared with under 2Mb for the narrative text. It is organized on the server as one XML document per Java package; within the package the XML vocabulary reflects the contents of a package in terms of classes, which contains constructors and methods, which in turn contain multiple arguments. The XML vocabulary reflects this logical structure rather than being pre-rendered into HTML. The conversion to HTML is all handled by one of the Saxon-CE stylesheet modules.

Links to Java classes from the narrative part of the documentation are marked up with a special class attribute, for example `<a class="javalink" href="net.sf.saxon.Configuration">Configuration</a>`. A special template rule detects the `onclick` event for such links, and constructs the appropriate hashbang fragment identifier from its knowledge of the content hierarchy; the display of the content then largely uses the same logic as the display of any other page.

2.2.6. Links between Sub-Pages in the Documentation

Within the XML content representing narrative text, links are represented using conventional relative URIs in the form `<a class="bodylink" href="../../extensions11/saxon.message">saxon:message</a>`. This "relative URI" applies, of course, to the hierarchic identifiers used in the hashbang fragment identifier used to identify the subpages within the site, and the click events for these links are therefore handled by the Saxon-CE application.

The Saxon-CE stylesheet contains a built-in link checker. There is a variant of the HTML page used to gain access to the site for use by site administrators; this displays a button which activates a check that all internal links have a defined target. The check runs in about 30 seconds, and displays a list of all dangling references.

2.2.7. The Up/Down buttons

These two buttons allow sequential reading of the narrative text: clicking the down arrow navigates to the next page in sequence, regardless of the hierarchic structure, while the up button navigates to the previous page.

Ignoring complications caused when navigating in the sections of the site that handle functions and Javadoc specifications, the logic for these buttons is:

```
<xsl:template name="navpage">
  <xsl:param name="class" as="xs:string"/>
  <xsl:variable name="ids" select="tokenize(f:get-hash(),'/')"/>
  <xsl:variable name="c" as="node()"
              select="f:get-item($ids, f:get-first-item($ids[1]), 1)"/>
  <xsl:variable name="new-li"
              select="if ($class eq 'arrowUp') then
                         ($c/preceding::li[1] union
                         $c/parent::ul/parent::li)[last()]
                      else ($c/ul/li union $c/following::li)[1]"/>
  <xsl:variable name="push" select="string-join(($new-li/ancestor::li union
                                $new-li)/@id,'/')"/>
  <xsl:sequence select="f:set-hash($push)"/>
</xsl:template>
```

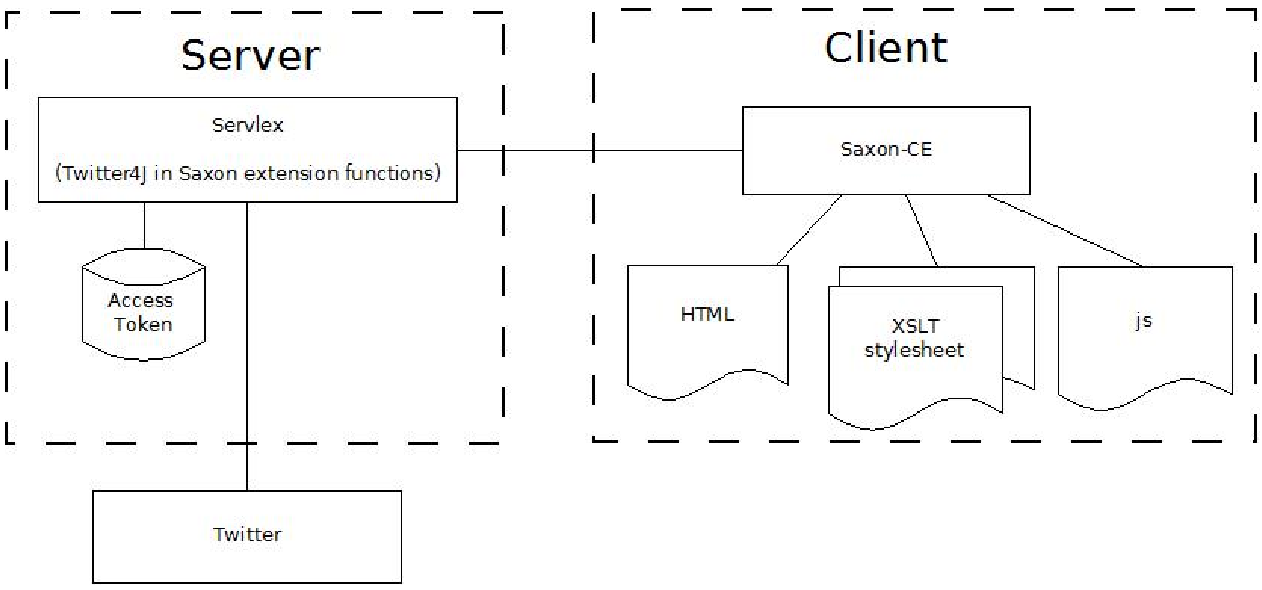
Here, the first step is to tokenize the path contained in the fragment identifier of the current URL (variable `$ids`). Then the variable `$c` is computed, as the relevant entry in the table of contents, which is structured as a nested hierarchy of `ul` and `li` elements. The variable `$new-li` is set to the previous or following `li` element in the table of contents, depending on which button was pressed, and `$push` is set to a path containing the identifier of this item concatenated with the identifiers of its ancestors. Finally `f:set-hash()` is called to reset the browser URL to select the subpage with this fragment identifier.

3. Chess Application

In this section we discuss the second of the two Saxon-CE applications: namely the multi-player chess game, which has been written primarily in XSLT and makes use of a proxy server to communicate via Twitter. There are three interesting and reasonable self-contained parts to the application, which we will examine in turn: the communication mechanism, the interactive user interface, and the chess game logic. But we will start by describing the overall application architecture.

3.1. Architecture

The multi-player game over Twitter consists of two components as shown in Figure 2: The server-side component containing tooling to handle Twitter communication and a client-side component containing the Saxon-CE application.



The architecture of the chess game application on Saxon-CE

Figure 2. Architecture of Chess Game application

3.1.1. Client-side Component

The client-side component is core. It consists of the Saxon-CE XSLT 2.0 processor for the browser, a single HTML file, two stylesheets and a Javascript file. The HTML file contains a skeleton webpage; the invariant parts of the display are recorded directly in HTML markup, and the variable parts are marked by empty <div> elements whose content is controlled from the XSLT stylesheets. The first stylesheet handles the rendering of the chess board, response to user input, and calls to the Javascript to perform Twitter support. The second stylesheet handles the chess game validation and general game play moves. Development with Saxon-CE often eliminates the need for Javascript, but at the same time it happily can be mixed with calls from XSLT. In this case we find it useful to abstract the Twitter http GET requests to a Javascript layer.

3.1.2. Server-side Component

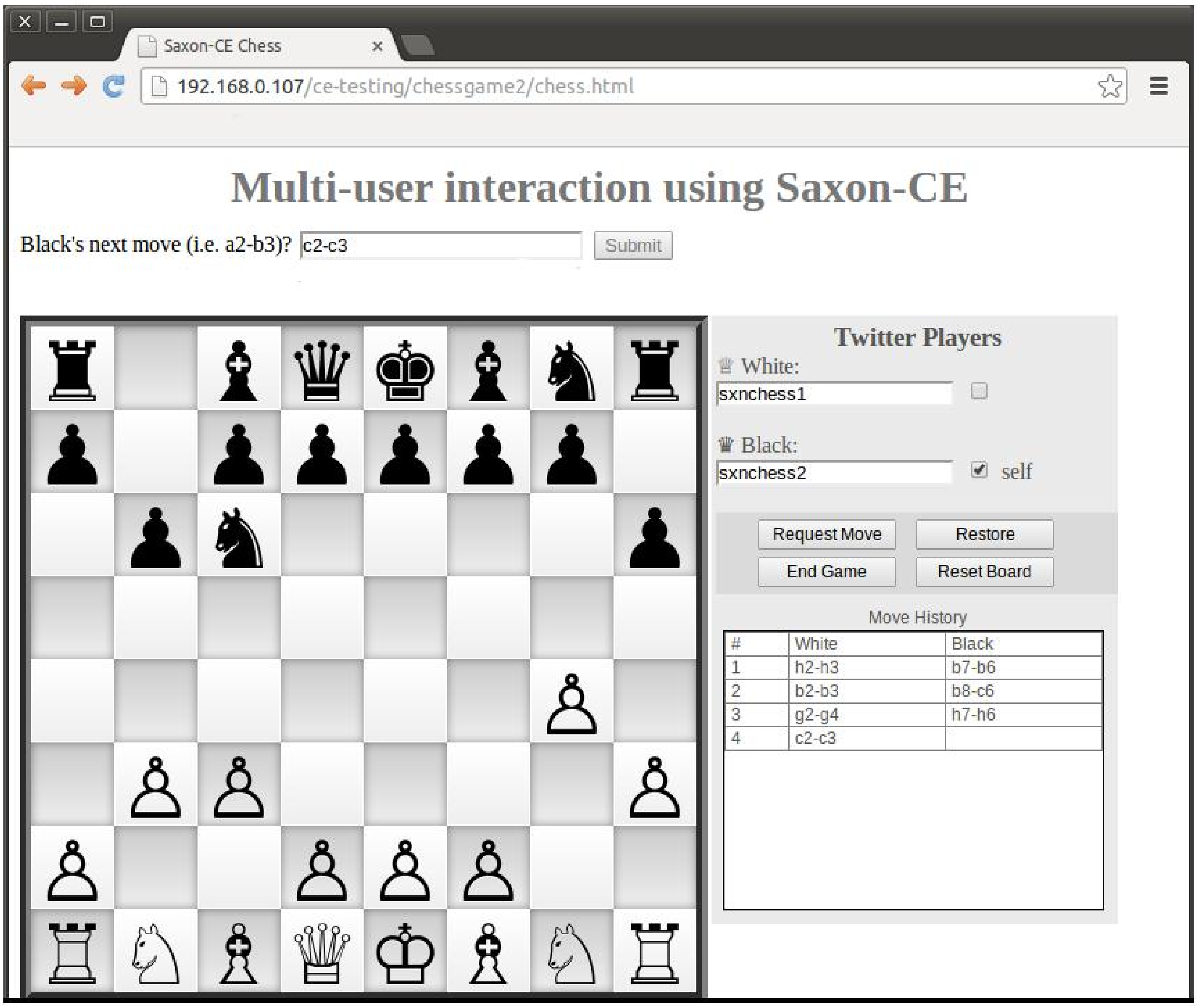
The purpose of the server-side component is to proxy communication with Twitter both to receive and send tweets. The original aim of this project was to develop an application where all functionality including Twitter communication was done client-side. However there are security and configuration concerns with sending tweets directly from the client-side. As discussed by [7], Javascript is a powerful language with good browser support, but the fundamental problem with using Javascript is that the source code is viewable. This means the consumer keys required in the Twitter authentication mechanism (OAuth) are exposed, which would compromise the application; and hence a server-side approach is appropriate.

We use Twitter's OAuth protocol [10] in favour of the Basic Auth facility, which is now deprecated and not supported. The OAuth authentication protocol allows a user to approve an application to act on their behalf without disclosing password credentials; security involves the use of consumer keys held by the application.

The API calls of OAuth are achieved using the Twitter4J API [8]. Twitter4J is an unoffical Java library for the Twitter API, which we integrete in the Servlex webapp [9] as Saxon extension functions called from within XSL. Servlex provides a way to write web applications directly in XSLT, XQuery and/or XProc. The request URIs are mapped to XSLT functions alongside variables which allow passing of data between the server and the client.

It should be emphasized that the server-side components are used only to proxy communication between the client and Twitter. The only data retained on the server is the Twitter authentication credentials. It is therefore not necessary for the two players to use the same server. Indeed, since the protocol for exchanging moves in Twitter messages is very simple, there is no real need for both players to be using the same client-side application. One could even envisage one of the players entering moves (as tweets) directly from the keyboard.

3.2. GUI interaction and Twitter Communication



Multi-user interaction using Saxon-CE

Black's next move (i.e. a2-b3)? c2-c3

Twitter Players

White: sxnchess1

Black: sxnchess2 ✓ self

Move History

#	White	Black
1	h2-h3	b7-b6
2	b2-b3	b8-c6
3	g2-g4	h7-h6
4	c2-c3	

Screen-shot of the chess board application in a browser.

Figure 3. Chess board application in browser

The chess game application in multi-player mode can played over the internet in the browser. The appearance of the graphical user interface (GUI) illustrated in Figure 3. The main components are:

- The display of the current state of the board
- The display of the history of the game
- Input fields and buttons allowing moves to be made (or other actions, such as resigning). The normal way of making a move is by drag-and-drop in the intuitive way. Support for this varies a little bit by platform, and on some devices; users may find it easier to enter moves from the keyboard in traditional chess notation.

When a user has made a move, we rely on the user to wait or watch for the opponent's reply (which might be after a few seconds or after a day or more, depending on the style of play). We don't poll for the move within the browser, or "push" the move from the server to the browser when it arrives. Instead, users have many tools available to alert them to tweets from their opponent, and when a tweet arrives they can click the "Request Move" button to accept their opponent's reply. Alternatively, if they have closed the browser in the meantime, they can reopen the application by loading the HTML page, and clicking "Restore", which restores the current state of play by reference to the Twitter timeline.

Figure 4 shows a flow diagram of the states and actions in the chess game application. The diagram is split into four four main sections: user interactions; Client which controls the user actions and interfaces with the server-side; Server which interfaces client actions with Twitter using published APIs; and the final section is Twitter itself.

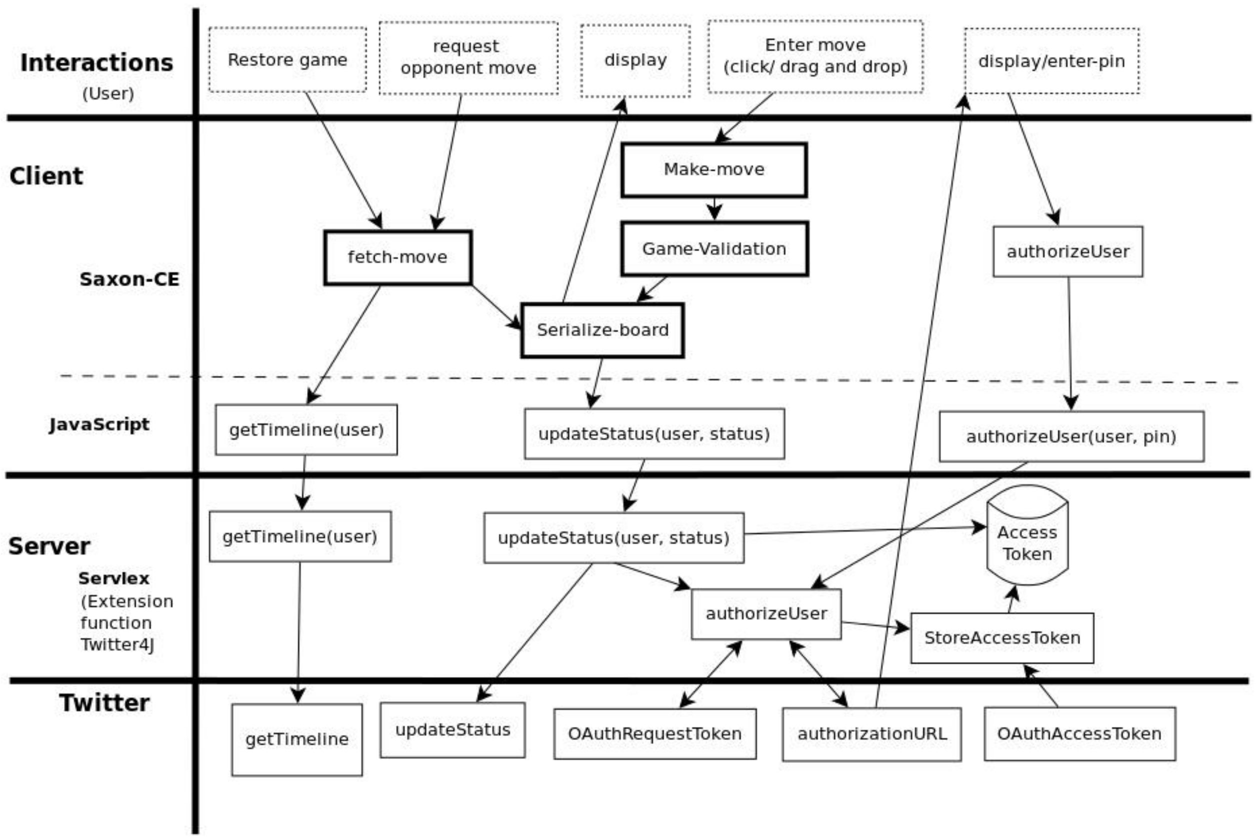


Diagram shows the interaction between user, client, server and Twitter.

Figure 4. Data flow diagram of the Chess game application

The server is required to communicate data between the chess game (on the client) and on Twitter. Servlex works as a dispatching and routing mechanism to components (implemented as XSLT stylesheets), applying a configuration-based mapping between the request URI and the component used to process that URI. The container communicates with the components by means of an XML representation of the HTTP request, and receives in turn an XML representation of the HTTP response to send back to the client. There are three XSLT functions to handle the Twitter operations: update Status, get timeline and authenticate user. These functions are written in Java using the Twitter4J library and made available as Saxon extension functions called from the XSLT.

1. *updateStatus*: At the start of a new game we assume the user is already authenicated. The player attempts a move on their chess board, which activates a Twitter submission. A request is made to servlex with the following URI pattern:

   http://192.168.0.2:8080/servlex/chess/updateStatus?user=johnWhite&status=@maryBlack%20RNBQKBNRPPPPPPPP____________________p___________pppp_ppprnbqkbnr%20e2-e4%20p:1

   We observe the URI pattern *updateStatus* and the parameter data after *?* is used to route the XSLT function and to supply the values required in the function. There are two parameters: the name of the user (in this case the user making the move), and the message to be sent. The message in this case includes the name of the other player, the current state of the board (represented as a simple string of 64 characters), and the move itself in algebraic chess notation (e2-e4 indicating an advance of the Kings Pawn by two squares). The message also includes the ply number (in chess terminology, a move consists of a ply by white followed by a ply by black). This we consider as the move number or incremented count in the current session.

   When an updateStatus request is processed there are two possible outcomes: success or failure. Success means the player's Twitter timeline status has been updated as a result of successful verification of the user's access token, in which case the client receives a "Ok" response, and the tweet is sent, hopefully reaching the other player.

   If the updateStatus request fails it means the player has not been granted authorization to the application. In this case we require the player to authenicate the application via Twitter. We achieve Twitter authentication using the PIN-based OAuth flow. Using Twitter4J we request an access token by requesting, from Twitter, a URL for the user to login and grant authorization. This URL is returned back to the client. Upon receipt, the client application allows the user the option to launch the URL in a new window. We discuss this further in step 2.

2. *authenicateUser*: To authenticate a user to play the chess game the client application asks the player to authenicate the application in Twitter. The client receives a URL from Twitter with a generated PIN number. The user will see a PIN code, with instructions to return to the application and enter this value in a form. The

value is then sent back from the client as HTTP GET request in the following pattern:

http://192.168.0.2:8080/servlex/chess/authenticateUser?user=johnWhite&pin=12345[1]

This is submited to Twitter via the servlex with consumer and user request tokens. If authorization is successful, an access token is sent in a callback by Twitter to the servlex code, and the server retains the details in persistent storage for future verification.

3. *getTimeline*: This function is used to request an opponent's move or restore a game from two players' Twitter feed. The URL pattern is as follows:

   http://192.168.0.2:8080/servlex/chess/timeline?user=maryBlack

To start play, players are required to enter their own Twitter screen-name and that of the opponent whom they will play. (Without this, the game is still playable in stand-alone mode without Twitter functionality, but we will ignore this option). The user has three main forms of action:

1. *Enter-move*: When it's the player's turn to make a move, this is done by a simple drag and drop or click on the piece you want to move and then click on the square you want to move it to. An event is then triggered and fired by the browser. Saxon-CE handles this event and fires an XSLT template rule for the <div> element representing the target square of the move. This is done in the ChessGame XSLT stylesheet. These template rules do basic checks on the syntax of the move and generate an XML view of the board, which is then passed through with the move data to the game-validation template. The game validation we discuss later, but upon successful validation of the move, the logic board is serialized and updated on the DOM view of the HTML page, which leads to it being visually represented on the screen. If the player attempts an invalid move, there is similar visual feedback, and the board state remains unaltered.

   A player's move is communicated to the opponent via Twitter. This is achieved as follows: we make an HTTP get request via a Javascript method called from the client-side XSLT. On the server the Servlex web application manages the HTTP requests.

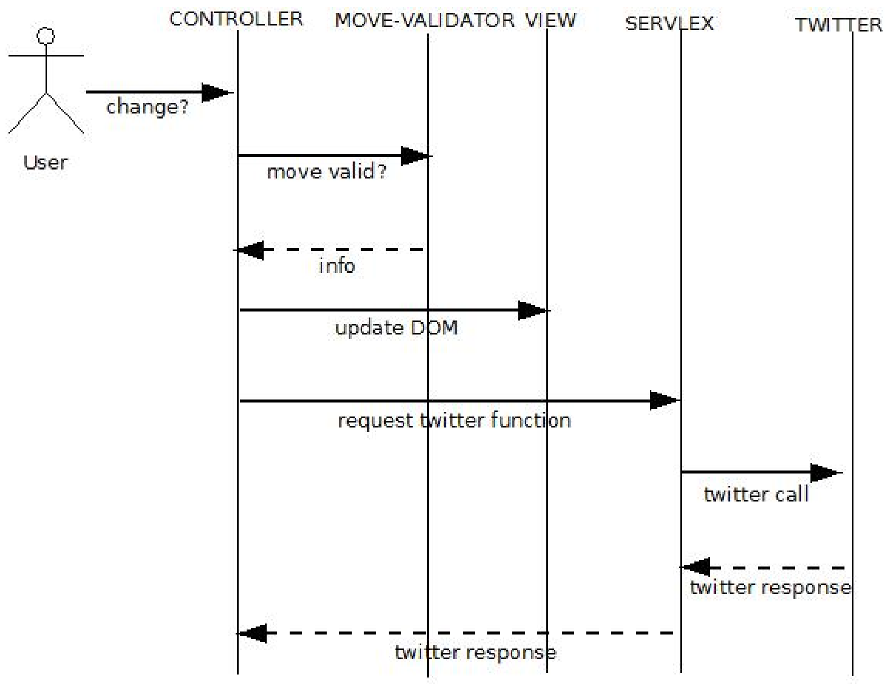
2. *Request opponent move*: When an opponent has made their move, a Twitter message is sent which appears on their Twitter timeline, it mentions the opposing player's screen-name allowing the player to pick up the move when they press request-move.

3. *Restore game*: Sometimes players will interact in real time, but on other occasions they may prefer to play a game slowly, with hours or days elapsing between moves. It is therefore not necessary to keep the browser open between moves. The state of the game can be restored at any time by reference to the Twitter timeline; it does not need to be retained in the client application, and it is not

[1] http://192.168.0.2:8080/servlex/chess/authenticateUser?user=johnWhite&pin=12345

retained in the server part of the application. Each tweet contains a representation of the state of the board, and if necessary (though we don't do it today) we could reconstruct the full game history by searching back through the time-line.

Figure 5 shows the sequence of operations in the chess game from end-to end.



Sequence Diagram which shows interaction between a player and various processes in the application.

Figure 5. Sequence Diagram of the Chess game application

3.3. Chess game logic

The final part of the application is concerned with what programmers often call "business logic", though in this case "game logic" would be more appropriate. The application, of course, does not include any chess strategy (though we could envisage one of the players being replaced with a robot); all it does is to verify that moves are legal, and update the state of the game in response to each move. This requires only enough look-ahead to ensure that a move does not leave a player in check, since such a move is illegal. In fact, our application does not currently have 100% coverage of all the laws of chess; we don't handle all the subtleties of pawn promotion, en-passent captures, or the rules about when castling is and is not permitted. Some of these rules in fact require additional information about the state of the game that cannot be inferred from knowing only the position of the pieces (castling is only allowed, for example, when the king has never moved in the history of the game; and a draw may be claimed if the same position occurs three times in the history of the game or if no pawn move or capture is made for 50 moves).

The XSLT logic of this part of the application is not especially noteworthy, and the same code could be used for a conventional server-side application. Keeping it compact, however, is important, because the size of a stylesheet has a material influence on the perceived responsiveness of Saxon-CE applications. The logic is encapsulated as a set of template rules, one matching each kind of chess piece, taking the state of the board and proposed move as parameters, and returning an indication of whether the move is valid. Here is the relevant rule for testing a knight's move:

```
<xsl:template match="div[@data-piece='knight']"
              mode="is-valid-move"
              as="element(move-test)">
    <xsl:param name="moveFrom" as="xs:integer"/>
    <xsl:param name="moveTo" as="xs:integer"/>
    <xsl:param name="board" as="element(div)+"/>
    <xsl:variable name="destinationAvailable"
                  select="not($board[$moveTo]/@data-colour = @data-colour)"/>
    <xsl:variable name="rowDistance" as="xs:integer"
                  select="f:row($moveTo) - f:row($moveFrom)"/>
    <xsl:variable name="columnDistance" as="xs:integer"
                  select="f:column($moveTo) - f:column($moveFrom)"/>
    <xsl:variable name="is-valid" as="xs:boolean"
                  select="$destinationAvailable and abs($rowDistance) *
                          abs($columnDistance) = 2"/>
    <move-test is-valid="{if ($is-valid) then 'yes' else 'no'}"/>
</xsl:template>
```

It relies on the simple principle that a knight's move is valid if and only if the product of the number of rows moved and the number of columns moved is 2. It omits the general rule that the target square must be either vacant or occupied by the opponent, because that rule is true for all pieces and can therefore be factored out.

As a first approximation, we can think of this template simply returning a boolean to indicate whether the move is valid or not. In practice, we also want to say why it's invalid (for example, because the target square is occupied), and for complex moves like castling or en-passent captures we also want to return sufficient information for the calling code to actually make the requested move by applying changes to the state of the board. So instead of a simple boolean, we return a constructed XML element containing this information.

We are primarily using template rules here as a polymorphic despatch mechanism, so that different rules apply to each kind of chess piece. The polymorphic templates are invoked using an <xsl:apply-templates> call contained in the logic of a stylesheet function, which returns the success/failure result to the caller, like this:

```
<xsl:function name="f:isValidMove" as="element(piece-move)">
        <xsl:param name="piece" as="element(div)"/>
        <xsl:param name="moveFrom" as="xs:integer"/>
        <xsl:param name="moveTo" as="xs:integer"/>
        <xsl:param name="board" as="element(div)*"/>

        <piece-move>
            <xsl:choose>
                <xsl:when test="$moveFrom = $moveTo">
                    <xsl:attribute name="is-valid" select="'no'"/>
                    <xsl:attribute name="description" select="'not moved'"/>
                </xsl:when>
                <xsl:when test="$board[$moveFrom][self::empty]">
                    <xsl:attribute name="is-valid" select="'no'"/>
                    <xsl:attribute name="description"
                                   select="'no piece at start position'"/>
                </xsl:when>
                <xsl:when
                    test="not($board[$moveTo][div/@data-piece eq 'empty' or
                          div/@data-colour != $piece/@data-colour])">
                    <xsl:attribute name="is-valid" select="'no'"/>
                    <xsl:attribute name="description"
                      select="'target sqare is occupied by your own colour'"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:variable name="move-test" as="element(move-test)">
                        <xsl:apply-templates select="$piece" ▶
mode="is-valid-move">
                          <xsl:with-param name="moveFrom" select="$moveFrom"/>
                            <xsl:with-param name="moveTo" select="$moveTo"/>
                            <xsl:with-param name="board" select="$board"/>
                        </xsl:apply-templates>
                    </xsl:variable>
                    <xsl:copy-of select="$move-test/@*"/>
                </xsl:otherwise>
            </xsl:choose>
        </piece-move>
</xsl:function>
```

For some of the more complex moves, making the move involves more than simply vacating the square where the piece started and occupying the square where it ends. Castling causes two pieces to move; en-passent capture removes a piece that is on neither the starting square nor the ending square; pawn promotion leaves a different kind of piece on the target square (and also involves user input, because the laws allow a pawn to be promoted to something other than the usual queen).

Good design practice suggests a model-view-controller architecture in which the model (the state of the board) is represented by a data structure independent of the view (the visualization of the board), where the controller ensures that the model and the view remain in step with each other. In fact our code holds the model implicitly as part of the HTML page (the view), which in some ways is a useful short-cut, but also reduces flexibility. For example, it makes the logic for deciding whether a player is in check more complicated, because special measures are needed to make trial moves without them being visible on the screen.

The entire logic for verifying and applying chess moves is 400 lines of XSLT coding.

4. Acknowledgement

Many thanks to Philip Fearon for his contribution in the development of the Saxon-CE project. In particular, he wrote the browser based technical documentation and helped to develop the Chess game application, both driven by Saxon-CE.

References

[1] *XSL Transformations (XSLT) Version 1.0. W3C Recommendation.* 16 November 1999. James Clark. W3C. `http://www.w3.org/TR/xslt`

[2] *XSL Transformations (XSLT) Version 2.0. W3C Recommendation.* 23 January 2007. Michael Kay. W3C. `http://www.w3.org/TR/xslt20`

[3] *Google Web Toolkit (GWT)*. Google. `http://code.google.com/webtoolkit/`

[4] *The Saxon XSLT and XQuery Processor*. Michael Kay. Saxonica. `http://www.saxonica.com/`

[5] *CXAN: a case-study for Servlex, an XML web framework*. Florent Georges. XML Prague. March, 2011. Prague, Czech Republic. `http://archive.xmlprague.cz/2011/files/xmlprague-2011-proceedings.pdf`

[6] *Twitter*. Twitter . `https://twitter.com/`

[7] *How-to: Secure OAuth in JavaScript*. Derek Gathright. 21 October 2010. Yahoo. `http://derek.io/blog/2010/how-to-secure-oauth-in-javascript/`

[8] *Twitter4J*. `http://twitter4j.org`

[9] *Servlex*. Florent Georges. `http://code.google.com/p/servlex/`

[10] *Twitter OAuth*. Twitter. `https://dev.twitter.com/docs/auth/oauth`

[11] *Hash URIs*. Jeni Tennison. `http://www.jenitennison.com/blog/node/154`

Efficient XML processing with XSLT 3.0 and XPath higher order functions

Abel Braaksma
Abrasoft
<abel@abrasoft.net>

Abstract

XSLT 3.0 has become a fully functional language with the embrace of functions as first class citizens. This paper introduces you to the new concepts of higher order functions, including function items, anonymous functions and partial function application. It explains the new syntax and shows new features of XSLT and XPath that are tightly integrated with function items, like maps, accumulators and the new XPath functions like fold-left/right and that take functions as arguments.

Keywords: XML, XSLT, XPath, HOF, higher-order-functions

1. Introduction to higher order functions

This section serves as an introduction to higher order functions. You can safely skip it when you are already well acquainted to the concepts.

Higher order functions, often abbreviated as HOF, are an essential part of functional programming languages. They have now become an integral part of XPath, XQuery and XSLT 3.0, pushing these languages to finally become fully-fledged functional languages.

A function is considered to be of *higher order* when it takes another function as an argument or when it returns a function. Any function not taking functions as arguments or as return types is called a first order function.

Consider for instance this pseudocode fragment:

```
function(int, int) f_add = getAddFunction(); // returns function
int x = f_add(10,30);                        // calls function
```

Here we assume that there is a function named *getAddFunction* which returns a function that takes two arguments of type *int*, and returns the sum of the two arguments. Here, the function *getAddFunction* is a higher order function and the variable *f_add* is assigned a function that takes two parameters. This variable can subsequently be used to call the function.

A typical use-case for higher order functions is sorting. Here is another example in pseudocode, suppose you have a function prototype for sorting such as:

```
string[] sort(string[], int function(string, string))
```

It takes an array of strings and a sort-function and returns a sorted array of strings. The sort function takes two strings and returns an integer that determines whether the two strings are equal, or the whether the first one is larger than the other one.

This has the advantage that you can now separate the string comparing from the sort function. A programmer can create a compare-function for, say, Arabic sorting, numeric sorting, sorting that takes special characters into account and so on. All it takes is writing a function that compares two strings. Here is a pseudocode example that compares strings as integers:

```
int compareStringsAsNumeric(string first, string second)
{
    // get integer value from string
    int i_first = int.Parse(first);
    int i_second = int.Parse(second);
    if(i_first == i_second)
        return 0;    // equal strings
    if(i_first > i_second)
        return 1;    // first comes before second
    if(i_first < i_second)
        return -1    // second comes before first
}
```

In normal string comparison, the string "200" will be sorted after the string "1000". With our numeric sorting example above, the input is treated as numbers and 200 will end up before 1000 as it should. In pseudocode this looks as follows:

```
string[] input = { "400", "120", "55", "503", "1234"};
string[] output = sort(input, compareStringAsNumeric);
print output;     // order is now 55, 120, 400, 503, 1234
```

The line of interest here is the second line. We call the sort-function, pass it the input array of numbers and as second argument we pass the function *compareStringAsNumeric*. The sort-function itself will now call our compare function for each pair in the input.

This concludes this little introduction. There's much more to say about higher order functions, but we'll see some of its uses and applications in the later examples in XPath and XSLT.

2. Higher order functions in XPath 3.0

Since Xpath 3.0, which is currently in Candidate Recommendation phase, the language supports higher order functions. A few highlights of what the syntax provides:

- Functions are a type, function item
- Inline higher order functions

- Functions as parameters and return value of other functions
- Functions that create functions and capture closures
- Memoization and determinism (XSLT only)
- XSLT patterns that match functions
- Accumulators: special purpose functions that collect data in streaming scenario's (XSLT only)
- Maps: special purpose functions that act as a dictionary (XSLT only, but part of the Xpath 3.1 requirements)

2.1. A simple example of a trivial higher order function

To start with a minimalistic example, here is an anonymous function in XPath that calculates the power of two of any number:

To start with a minimalistic example, here is an anonymous function in XPath that calculates the power of two of any number:

The first few lines define a function that takes an integer and returns the input multiplied by itself, in other words, the power of two. This function is then bound to the variable *$pow*.

The second line shows how to call a variable that contains a function which in turn calls the function defined previously.

Let's look at this in more detail:

- The first line contains a typical XPath let expression[1] that binds a function item to a variable
- The function signature is the part to the right of the assignment expression ":=" and it is formally defined as:

  ```
  "function" "(" ParamList? ")" ("as" SequenceType)? FunctionBody
  ```

- The *parameter list*, in the example "$num as xs:integer " can contain any number of parameters, where each parameter name must be unique and the optional typedeclaration defines the expected type. Formally:

  ```
  "$" EQName  ("as" SequenceType)?
  ```

- The *function return type*, in the example "as xs:integer" is the return type of the function.
- The *function body* is the part within curly brackets. It can contain any legal XPath expression. Functions do not have a "return" statement. The return value of the function is the result value of the XPath expression. Formally:

  ```
  "{" Expr "}"
  ```

- The last line contains the *return clause* of the let expression. It calls the variable we just bound to, which is a function. The syntax of calling a function that is bound to a variable is equal to the syntax of calling a function created with xsl:function. Here, it returns the power of two of the number 42, which is 1764.

2.2. Using anonymous functions in expressions

While the previous example may seem a bit contrived, as its only use is in the current scope of the let-expression, anonymous functions can be very powerful. You can use them in any place where you would otherwise be able to use a first order function, and you can pass them on to functions that take other functions as parameters.

Here are some examples:

```
(: (1) calculating the VAT of prices  :)
books/book/function($price) {$price * 1.21} (@price)

(: (2) calc VAT and show title and both prices  :)
books/book/
  function($price) { concat (
      $price/parent::book/title, ": ",
      $price, ", incl. VAT: ",
      $price * 1.21, "
")
  } (@price)


(: (3) finding the length of the longest word in a sequence :)
let $max := function($accumulator, $item)
{
    if($accumulator > string-length($item))
    then $accumulator
    else string-length($item)
}
return fold-left($max, 0, ('a', 'quick', 'yellow' 'fox'))
```

The first example (1) shows how to use an inline function inside an XPath select expression. It selects all the prices of books and calculates the VAT by multiplying the price with a factor of 1.21. Note that the function does not have specific types for return types and parameters; by default this means that all types are of type *item()*.

The second example (2) shows the same VAT example, but this time selecting a sequence of strings that contains the book title, the normal price and the VAT price. For readability, the inline function could be split into a separate let-binding expression.

The third example (3) introduces one of the standard XPath 3.0 higher order functions that take a function as a parameter, fn:fold-left. It processes the supplied sequence from left to right, applying the supplied function to each item and keeping an accumulated value from one call to the other. The first parameter in the anonymous function contains the accumulator (in this example, it holds the length of the longest word so far), the second contains the next item of the sequence, starting with the first.

It can be hard to visualize what happens underneath during folding. By writing down the individual calls to the function, it becomes easier to understand what is going on:

```
(: first item, returns 1 :)
$max(0, 'a')
(: second item, returns 5 :)
$max(1, 'quick')
(: third item, returns 6 :)
$max(5, 'yellow')
(: fourth item, returns 6 again :)
$max(6, 'fox')
```

The first invocation has the initial value for the accumulator, as given to the fn:fold-left function, and the first item of the sequence, it will return 1, because that's the length of the string 'a'. The next invocation uses the previous output, 1, as input for the accumulator argument, and will compare it to the length of the next string in the sequence, 'quick'. Because that string is larger then 1, the function will return 5. And so on, until the last item in the sequence, which is shorter than the largest item so far, so the anonymous function will return the accumulator value itself.

2.3. Closures in inline function items

One of the earliest languages to support closures was Smalltalk, more specifically Smalltalk-72. In the more current Smalltalk ANSI standard 1.9 a closure is defined as:

"Each block object is an independent closure that captures the current bindings for any enclosing functions' arguments or temporaries that are referenced from within the block's <block constructor>"

The history of closures goes further back. In 1964, Peter J Landin introduced the term closure in a paper on lambda calculus. His more generic definition still applies today:

"Also we represent the value of a λ-expression by a bundle of information called a "closure," comprising the λ-expression and the environment relative to which it was evaluated."

Today, closures find their usage mostly in functional languages like Lisp, Haskell, F#, but imperative object oriented languages like C# support it too (note: Java plans to support it for version 8, which is not yet published). In XPath, closures are a fundamental part of higher order functions: each local inline function that you define captures its environment as a local dynamic context bound to the function.

```
let $current = .
return function() { $current }
```

Here the variable *$current* is outside the scope of the function. However, because it is used inside the function body, the function will retain the value of *$current* that it had on the moment the function was created. Suppose you would use the function in the following XPath expression:

```
paper/para/(let $current = .
return function() { $current })
```

It returns now a sequence of function items, which each function body holding a different paragraph of the paper. Suppose you use the following input document:

```
<paper>
   <para>higher order functions</para>
   <para>are fun!</para>
</paper>
```

And the following XSLT instruction:

```
<xsl:template match="paper">
  <xsl:variable name="para" select="para/
     (let $current = .
     return function() { $current })" />
  <xsl:sequence select="$para()/text()" />
</xsl:template>
```

Then the output will be a sequence of text nodes, "higher order functions" and "are fun!". This works, because when the function was defined, it captured the context node inside its closure. It doesn't matter where and how you would use the returned function items, they will retain the information in the function closure.

In the section "Putting it all together" I will expand on this example by applying the function items to other functions and by selecting them through xsl:apply-templates. Using closures, it becomes possible to dynamically create sequences of sequences.

2.4. Schönfinkeling, currying and partial function application

It is occasionally handy to be able to just use a function without all its parameters. The concept of cutting up a function with multiple parameters into several functions that each take one parameter , is called *currying*, after Haskell Curry (who actually took the idea of Moses Schönfinkel, who first introduced it in 1920, later published in 1924, hence it should have been called *schönfinkeling* instead).

The method of currying available in XPath 3.0 is slightly different from the currying available in functional languages like Haskell. Instead, XPath supports *partial function application*. Where currying is the process of atomizing a function recursively into functions that each take one argument, partial function application is the process of removing several arguments and replacing them with fixed values.

Using partial function application in XPath is trivial. Using the special syntax with a question mark replacing the arguments you want to keep, you return a function that defaults to the other arguments that you applied. An example clarifies this:

```
let $format := format-date(?, '[D] [MN] [Y]', 'es', 'AD', ())
return $format(current-date())
```

Here we assign to $format a version of the build in format-date function that only takes one argument. This is convenient, because this function can either take two or five arguments.

Suppose you want to create a format date function that takes a date and a language specifier, you can define one in XSLT as follows:

```
<xsl:variable name="format-date"
     select="format-date(?, '[D] [MN] [Y]', ?, 'AD', ())" />
```

We can now use the variable $format-date as a function, for instance:

```
books/book/$format-date(@publish-date, 'es')
```

Another example where currying is of great use, is fn:concat. Because just using a question mark as a parameter changes the function into a function that returns the same function with its other parameters predefined, we can write the following:

```
let $format := concat("Price of '", ?, "': € ", ?, "&#xA;")
return book/$format(title, @price)
```

This returns output similar to:

```
Price of 'Huckleberry Finn': € 16.80
Price of 'La vie en rose': € 23.60
```

2.5. Getting function references of predefined and named overloaded functions

The syntax for passing a predefined or named function to another function is done using the hash-syntax. Take the function name, add a hash-sign "#"and add the parameter count. For example: format-dateTime#5 returns the five-parameter function of *fn:format-dateTime*.

Formally:

```
NamedFunctionRef ::= EQName "#" IntegerLiteral
```

This definition implies that it is also possible to use the new Expanded QName syntax for specifying a function reference. I.e., the following will return the three-argument function of the *mysort* function in the namespace "http://example.com/func":

```
let $functionref := Q{http://example.com/func}mysort#3
```

The expanded QName syntax can also be used in other areas where QNames are allowed, for instance in the definition of the parameters of an anonymous function. The expanded QName is introduced to make machine-generation of syntax easier and is not meant for use in literal stylesheets.

Note that in XPath and XSLT it is not possible to overload functions only by the parameter type. Thus the resulting syntax does not need to provide a way to distinguish between a function that takes one parameter of type string and another function of the same signature that takes type integer, simply because there is no way to define such functions.

2.5.1. Caveats with predefined function items

When working with function items there are a few things to notice:

- When not supplying the hash-sign, the zero-argument function is returned. In many cases, this is the function overload that operates on the current context item, however, it is not allowed to use a function item that relies on the context. It is not required for a processor to raise a static error here, which can result in hard-to-diagnose runtime errors.
- The special function fn:concat has a variable number of arguments. By using the hash-syntax, you create a function item with a specific number of arguments.
- It is not possible to get the function items that belong to operators. I.e., op:numeric-multiply#2 will result in an error.
- The syntax for XSLT functions is equal to the syntax for build-in functions.
- It is not possible to get the function item of an anonymous function this way, because an anonymous function does not have a name.
- You can use this syntax for getting the function item of a constructor function, i.e. xs:integer#1 returns the constructor function of xs:integer.

2.6. Function items

Every language that supports higher order functions, requires some kind of a datatype to pass functions around. Typically, in "pure" functional languages, everything is a function, even constants. Other functional languages, like F# take another approach, where a function is a special type of variable. This approach is also used for XPath, where a function item is a special type that holds a function.

Functions are first class citizens. To make it possible for a language to pass on functions from one function to another, and to return functions, the XDM needed something new: function items. Whenever you create a function, it has the type of a function item. The general form of this type is *function(*)* which is a special syntax for any function with any amount of parameters and any return value.

The specific syntax looks much like the function declaration itself, except for the named arguments, you only specify the type of the arguments, not the parameter names. For instance, the function $pow from an earlier example in this paper has the type

```
function(xs:integer) as xs:integer
```

This defines a function that takes an integer and returns an integer.

Note that when specifying an inline function, you do not have to specify the type of the parameter or the return type. When you don't, the matching function signature is *item()** for the missing type specifications. For instance, suppose you have the following function definition:

```
let $f := function($a as xs:string, $b){...}
```

Then the discrete type of the function item of *$f* is *function(xs:string, item()*) as item()**.

2.6.1. Selecting function items

Using sequence type matching, you can select functions in a sequence that have a certain type, or filter for only the function items in a sequence. Some examples:

```
(: selects function items in $sequence :)
$seq[. instance of function(*)]
(: selects function items with one parameter :)
$seq[. instance of function(item()*)]
(: selects function items returning xs:string and one param :)
$ seq[. instance of function(item()*) as xs:string]
(: selects function items of the specific type :)
$seq[. instance of function(xs:double, xs:double) as xs:integer]
```

2.6.2. Matching function items

As "first class citizens" function items really are part of the data model and as a consequence, they can be queried. For instance, one can have a tree that contains function items and apply a transformation to it. One can even create a pattern for a function item, such as:

```
<xsl:template match=".[.  instance of function(*)]">
```

which will match any function.

Once inside the matching template, you have a matching function item, but there's relatively little you can do with it. You can call the function, but only when you know the number of arguments statically, unless you resort to the new xsl:evaluate, which evaluates an XPath expression on the fly.

The few things you can find out are the name of the function using *fn:function-name*, unless the function is anonymous in which case the name is absent and the arity of the function using *fn:function-arity*, which returns an integer about the arity, but no information about the parameter types or return types is returned.

2.6.3. Covariance and contravariance

With matching, selecting or passing function items, functions can have non-matching return and parameter types. Typically, function items are covariant for their return types and contravariant for parameter types.

For return types it means that a function item with a more specialized return type (for instance, *xs:integer*) can be converted to a function item of a less specialized return type (for instance, *item()**).

For parameter types it means that a function item with a more generic parameter type (for instance xs:integer) can be converted (coerced) to a function with a more specialized parameter type (for instance, xs:positiveInteger).

Every function item is a subtype of *function(*)*, which itself is a subtype of *item()*.

2.7. Predefined functions

XPath has several functions that take functions as parameters. Of interest are *fn:map*, *fn:filter*, *fn:fold-left* and *fn:fold-right*. These functions are concisely described in the Recommendation, but it is beyond the scope of this paper to give a deep overview of them.

2.8. Putting it all together: a naive prime number filter

Applying higher order functions, partial function application and two of XPath 3.0's predefined higher order functions, namely fn:filter and fn:fold-left, we can create a tiny prime number filter that operates on sequences of numbers:

```
let $is-prime := function($x)
{
    $x gt 1 and (
        every $y in 2 to math:sqrt($x) cast as xs:integer
        satisfies $x mod $y ne 0)
}
return fold-left(
     concat(?, ',', ?),
     2,
     filter($is-prime, 3 to 50))
```

This example returns a sequence of primes concatenated with *fn:concat*, for which we used partial function application to combine the input:

```
2,3,5,7,11,13,17,19,23,29,31,37,41,43,47
```

A brief explanation of what is happening here:

- The first line binds the variable $is-prime to an anonymous function
- The second to fifth line are the function body. This function returns true when none of the numbers under the current number, excluding 1 and itself, can divide the current number. We only need to find the divisors under the square root of the current number.
- The sixth line uses fn:fold-left, which has been explained earlier in this paper
- The seventh line is the function applied to each item in the sequence by fn:fold-left and is the function item of fn:concat that inserts a comma between the accumulator and the current item.
- The eight line is the initial accumulator for fn:fold-left.
- The nineth line introduces the fn:filter function, which takes a sequence and returns all items of the sequence for which the supplied function item returns true. The supplied function item is $is-prime which is the function that determines whether the supplied number is indeed a prime or not.

The implementation for *$is-prime* is pretty naive. Most notably, for each number, the whole sequence needs to be re-evaluated up to the square root of the current number, which can take up a lot of resources for larger numbers. Luckily, XSLT 3.0 offers a way of caching function results by means of memoization. This is explained in the section on memoization further on in this paper.

3. Higher order functions in XSLT 3.0

XSLT 3.0 embraces and expands on the concepts of higher order functions in several ways. These are:

- xsl:param and xsl:variable can now hold function items. You can use the syntax described under "Function items" to define the function item types in the as-clause of the variable
- Named functions through xsl:function can accept parameters of function items and they can return function items.
- Maps are essentially function items that take one parameter, the key, and return the value associated with it. Maps are also known as associated arrays, hash tables, lookup tables, or dictionaries.
- Accumulators augment the input tree with a call to a function when a particular node is matched, and accumulate the result of this call to the next time the accumulator is called when another node is matched against to the same pattern.

This is particularly useful during streaming, when multiple downward expressions are not allowed.

- Information hiding through xsl:package. While the concept of information hiding is wider than just functions, it allows functions to be either private to the package, abstract (with a signature, but not with an implementation) or public.
- Matching and applying templates against function items, this is described previously under "Function items" above.
- Memoization (caching) of functions

3.1. Using higher order functions in XSLT 3.0

If we take the example we used with anonymous functions and create a named function of the *$max*-function, it would look something like this, which simply returns the longest word:

```
<xsl:function name="my:max>
   <xsl:param name="a" />
   <xsl:param name="b" />
   <xsl:sequence select="
          if($a > string-length($b))
          then $a
          else string-length($b)"/>
</xsl:function>
```

This function can be used using the #-syntax to call it with the *fn:fold-left* function:

```
fold-left($my:max#2, 0, ('a', 'quick', 'yellow' 'fox'))
```

This is not highly flexible. We are fixed to the counting function being string-length all the time. However, what if we were interested in the word with the most vowels? Or what if we are not interested in the string length, but more in the actual word length? I.e., a string that contains "Help!" has a length of 5, but the word length is 4. To accommodate for all these wishes, we should parameterize the string-length function and replace it with any function we want:

```
<xsl:function name="my:max">
    <xsl:param name="count-function" />
    <xsl:param name="a" />
    <xsl:param name="b" />
    <xsl:sequence select="
            if($a > $count-function($b))
            then $a
            else $count-function($b)" />
</xsl:function>
```

But is this enough? How do we use this with the *fn:fold-left* function? The fold function requires a function that takes two parameters. Remember the syntax of partial function application. We can write the following: $my:max(string-length#1, ?, ?). This will return the my:max function as a function item with two parameters and the first parameter already filled in. Putting it together the fold statement now becomes:

```
fold-left($my:max(string-length#1, ?, ?), 0, ('a', 'quick', 'yellow' 'fox'))
```

Suppose instead that we want the users of our function to be using a more relaxed syntax without the question marks as placeholder. Could we pull it off? Instead now we need to create a function. Here is how. The following function takes only one parameter, the count function and returns a function that takes two parameters. This also revisits the story on closures described in the XPath part. After the function returns a function, the supplied function, *$count-function*, remains in existence even when the my:max is assigned to a variable and called at a later stage. Note that the as-clause on the function is not required, but it helps clarify what the xsl:function statement does.

```
<xsl:function name="my:max" as="function(*)">
    <xsl:param name="count-function" />
    <xsl:sequence select="
        function($a, $b)
        {
            if($a > $count-function($b))
            then $a
            else $count-function($b)
         }" />
</xsl:function>
```

Applying this to our fold-left example, the statement has become simpler again:

```
fold-left($my:max(string-length#1), 0, ('a', 'quick', 'yellow' 'fox'))
```

So far I've refrained from using specific types for parameters and function items. The final version of our example is the discretely typed version. By specifying precise types for each parameter and return type, our code becomes less prone to errors:

```
<xsl:function name="my:max"
    as="function(xs:integer, xs:string) as xs:integer">
    <xsl:param name="count-function"
         as="function(xs:string) as xs:integer" />
    <xsl:sequence select="
        function($a as xs:integer, $b as xs:string) as xs:integer
        {
            if($a > $count-function($b))
            then $a
            else $count-function($b)
         }" />
</xsl:function>
```

Finally, here is the full example where the previous code is used. It defines a way of counting the maximum of an input sequence, by applying our own maximize function. The example has a function for counting the word length of only letters and one of counting all consonants:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:math="http://www.w3.org/2005/xpath-functions/math"
    xmlns:my="http://exselt.net/local"
    exclude-result-prefixes="xs"
    version="3.0">

    <xsl:output method="text"/>

    <xsl:variable name="input">Welcome everyone, in cold XMLPrague!</▶
xsl:variable>
    <xsl:variable name="words" select="tokenize($input, ' ')" />

    <xsl:template match="/">
        <xsl:sequence select="
            let $max := my:max(my:word-length#1)
            return fold-left($max, 0, $words)" />
    </xsl:template>

    <xsl:function name="my:word-length" as="xs:integer">
        <xsl:param name="word"  as="xs:string"/>
       <xsl:sequence select="string-length(replace($word, '[^a-z]', '', 'i'))" ▶
/>
    </xsl:function>

    <xsl:function name="my:count-consonants" as="xs:integer">
        <xsl:param name="word" as="xs:string" />
        <xsl:sequence select="string-length(replace($word, '[^aeiou]', '', ▶
'i'))" />
    </xsl:function>

    <xsl:function name="my:max" as="function(xs:integer, xs:string) as ▶
xs:integer">
       <xsl:param name="count-function" as="function(xs:string) as xs:integer" ▶
/>
        <xsl:sequence select="
            function($a as xs:integer, $b as xs:string) as xs:integer
            {
                if($a > $count-function($b))
                then $a
                else $count-function($b)
            }" />
    </xsl:function>

</xsl:stylesheet>
```

3.2. Accumulators put XSLT on steroids

In streaming, it is not allowed to have multiple downward selections, or to have any upward selections. This makes it hard to create any expression that requires previous information in the document, for instance counting all paragraphs (however deeply nested) or a word count, together with other kinds of processing.

There are ways of fixing this with tunneled parameters, but this is relatively cumbersome and error prone. The working group decided to provide a new declaration, xsl:accumulator, which defines a function that can accumulate data when a matching node is encountered. As a byproduct, accumulators proof to be very handy as a tool with non-streaming data.

In essence, using accumulators, you create a *sticky function* that is virtually attached to the input tree when the pattern of the accumulator matches the function. Then, when the node is visited, both before visiting and when leaving the node, a specific function is called. This is called the accumulator function, the *sticky one*.

This match happens regardless of whether there is a template match for the node.

Leaving the node. This is a new concept in XSLT and begs for a little extra explanation. In XSLT, nodes are visited in document order, and there's no way you can specify that you're at the beginning of a node, or at the end of a node. However, in the <chapter><para> example, one chapter can have several paragraphs. In streaming, it is not possible to visit all paragraphs (for instance for counting them) and have the result while at the chapter element, because multiple downward selections are not allowed. By specifying an accumulator as *post-descent* you can access the value of the accumulator while at the chapter element and the processor will only output the result *after* it visited all children of chapter.

Let us look at an example:

```
<xsl:accumulator post-descent="acc:count-para" initial-value="0">
   <xsl:accumulator-rule match="para" new-value="$value + 1" />
</xsl:accumulator>
```

This example will match each <para> element and will add 1 to the current count. The current value of the accumulator is retained (accumulated) through the *$value* variable, which is a special variable that is only available inside an accumulator rule. You can access the accumulator value by simply calling it as a function:

```
<xsl:template match="chapter">
    <xsl:apply-templates select="para" />
    <xsl:value-of select="
       concat('Total paragraphs so far: ', acc:count-para())" />
</xsl:template>
```

Accumulators can be very powerful and warrant a full paper on their own. Because they are not strictly higher order functions, but typically a new kind of function that does not take any parameter and cannot be used themselves as a function item, i.e., acc:count-para#0 is not allowed, I will not go into them any deeper here.

3.3. Memoization and determinism of functions

Quite recently, the working group decided to allow programmatic memoization of functions. Memoization is the process of caching the result of a function by remembering the function and the input parameters. When the same function is called again, the memoization mechanism returns the previously calculated value. This is particularly useful with recursive functions, like fibonacci number calculation, or the previous prime number calculation.

As a typical example of a recursive function, let us have a look at fibonacci numbers:

```
<xsl:function name="f:fibonacci">
    <xsl:param name="n" as="xs:integer" />
    <xsl:sequence select="
        if (($n = 1) or ($n = 2))
        then 1
        else f:fibonacci($n - 1) + f:fibonacci($n - 2)" />
</xsl:function>
```

This is very expensive, especially for large $n. Because the fibonacci has two recursive calls, it cannot be tail-recursive optimized by the compiler. In functional programming languages, one optimization that can be applied here is continuations, another, simpler one, is using memoization. While continuations can be achieved using XSLT 3.0 and higher order functions, they are relatively hard to master and it is uncertain whether the XSLT processor will actually tail-optimize your function. But with memoization, each call to the function with a given parameter will be cached and reused. Instead of running in O(2^n) time, this function will now run in O(n) time, which is a dramatic improvement:

```
<xsl:function name="f:fibonacci" memoize="full">
    <xsl:param name="n" as="xs:integer" />
    <xsl:sequence select="
        if (($n = 1) or ($n = 2))
        then 1
        else f:fibonacci($n - 1) + f:fibonacci($n - 2)" />
</xsl:function>
```

By defining that a function is memoizable, you also define the function to have no side effects. While a typical function in XSLT will not have side effects, a function that returns nodes will have the side effect that each node has a new id. By enabling memoization you actually tell the processor that you are not interested in this new node's identity and that the processor can optimize this identity away at its own discretion.

A warning is in place about the latter. The *memoize* attribute is a strong hint for the processor, but if the processor has no way of memoizing it, for instance because of the memory vs performance tradeoff, it has the option not to follow the hint. As a result, memoization is a power tool that may make your stylesheet incompatible between processors, especially when your code would behave differently when the node identity side effects are enabled.

3.4. Other areas for higher order functions in XSLT 3.0

A prime candidate for higher order functions is xsl:sort. As explained in the introduction of this paper, separating the algorithm from the compare algorithm is one that applies to many use cases. Currently it is not easy in XSLT to come up with ones own sorting order and having a compare function as an extra option would be very beneficial

Another area where XSLT could improve is by generalizing the memoization concept. Currently it is not applicable to inline anonymous functions, to function items or to accumulators. One way to achieve generic memoization is by allowing lazy values and leaving it to the programmer to define a memoization algorithm using lazy sequences that hold their data. Another one is by taking the current concept, but expanding it with a memoize function that takes another function as its argument and returns the same function as a memoized function.

4. Conclusion

XPath 3.0 and XSLT 3.0 have become a "real" functional programming language with the addition of higher order functions. The syntax is relatively straightforward and the invention of partial function application makes using concat and other complex existing functions much nicer. It touches almost every aspect of the language. Creating functions that return functions or functions that take functions as an argument has become trivial. Powerful, yet simple functions that come prepackaged with XPath, like map, fold-left/right and filter, make many common tasks a lot easier to accomplish.

As we have seen, accumulators are very powerful, and maps makes life a lot easier with regards to JSON integration, dealing with input parameters or dealing with error codes.

The Working Group of XSLT 3.0 could take this even a small step further and augment existing instructions with attributes that accept functions, like xsl:sort.

Bibliography

[1] *The CUCH as a formal and description language*. Formal Language Description Languages for Computer Programming. 13. 179-197. C. Böhm. 1966. North-Holland Co., Amsterdam.

[2] *Christopher Strachey—Understanding Programming Languages*. Higher-Order and Symbolic Computation. 13. 52. Rod Burstall. 2000.

[3] *History of Lambda-calculus and Combinatory Logic*. Online edition at Swansea University: http://maths.swan.ac.uk/staff/jrh/papers/JRHHislamWeb.pdf. Felice Cardone and J. Roger Hindley. 2006. Swansea University Mathematics Research Report. [No. MRRS-05-06.]

[4] *Grundlagen der kombinatorischen Logik*. American Journal of Mathematics. p512. H. B. Curry. 1930.

[5] *The Scheme Programming Language*. K. Dybyg. 1996. Prentice-Hall.

[6] *The mechanical evaluation of expressions*. The Computer Journal. 6. 308–320. P.J. Landin. 1963.

[7] *Recursive functions of symbolic expressions and their computation by machine.*. Communications of the ACM. 3. 184–195. J. McCarthy. 1960.

[8] *Arithmetices Principia, Nova Methodo Exposita.*. Giuseppe Peano. 1889. Fratelli Bocca, Torino, Italy.

[9] *Über die Bausteine der mathematischen Logik*. Mathematische Annalen. 92:305–316, 1924. M. Shönfinkel. 1924.

[10] *XQuery and XPath Data Model 3.0, W3C Candidate Recommendation 08 January 2013*. http://www.w3.org/TR/2013/CR-xpath-datamodel-30-20130108/. Norman Walsh, Anders Berglund, and John Snelson.

[11] *XML Path Language (XPath) 3.0, Latest Version*. http://www.w3.org/TR/xpath-30/. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson.

[12] *XML Path Language (XPath) 3.0, W3C Candidate Recommendation 08 January 2013*. http://www.w3.org/TR/2013/CR-xpath-30-20130108/. Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson.

[13] *XSL Transformations (XSLT) Version 3.0, Latest Version*. http://www.w3.org/TR/xslt-30/. Michael Kay.

[14] *XSL Transformations (XSLT) Version 3.0, W3C Working Draft 1 February 2013*. http://www.w3.org/TR/2013/WD-xslt-30-20130201/. Michael Kay.

Building a Personalized Communication Platform using Open Standards

Nick van den Bleeken
Inventive Designers
<nick.van.den.bleeken@inventivegroup.com>

Abstract

Communicating with customers in the manner they prefer, with the information they find interesting, is getting ever more challenging with rise of mobile and social media. This paper will discuss how we have built a Personalized Multi-channel Communication Platform using open standards, what extensions to those standards were required and what challenges we faced creating the platform.

Keywords:

1. Introduction

With the rise of mobile and social media (e.g.: Facebook, Twitter, LinkedIn, Pintrest), the number of channels over which to contact people keeps growing rapidly. More traditional output channels like print, email and sms keep their value, but depending on the addressee(s) preferences, the content and context, the best communication channel will vary. The importance of creating unique and engaging conversations increases the need for a full overview of all your communications (what is sent, when was it sent, was the communication delivered and/or opened by the addressee, did the addressee(s) interact with communication and how did they interact). Due to the great differences between these types of communication, the platform should be able to generate and manage high-volume batch, on-demand and interactive communications.

To build such a Personalized Communication Platform we used a lot of different open standards. We chose XML as the backbone of the platform because XML isn't limited to a fixed grammar; a lot of services return XML (if not, the data can easily be converted to XML and back); and there are rich query and transformation languages for XML. Additionally most open XML standards allow extensions, which are required to reach our goal. Most of those extensions are already scheduled for a future version of the standards or are being discussed in the appropriate working group.

This paper will discuss the standards that we used to build the platform, how they are combined to create a Personalized Multi-channel Communication Platform, what extensions to those standards were required, and what our challenges were building the platform. The platform is used today as a multi-channel communication platform that helps enterprises worldwide to improve their customer communications, leading to increased customer engagement and loyalty. This paper is absolutely not intended to be commercial in nature. We thought it would be interesting for people to see how technologies they work on for a large part of their life can all be combined to create an enterprise communication platform, and what the challenges were in using and combining these technologies.

2. Platform Overview

During the complete communication process a lot of decisions have to be made (e.g.: what channel to use, when to send the communication, fallback when the communication is unsuccessful, communication approval) based on the personal preference of the addressee, the available data and other context information. This orchestration is done using State Chart XML (SCXML) [1].

The data used throughout the communication process is XML. The data can be pushed into the platform (folder drop, SOAP/REST endpoint), pulled by the platform (XQuery, SOAP/REST call), or a combination of both. If manual data entry is required we use XForms to collect the data. We are also using XSLT, in case different data sources need to merged, transformed or enriched.

For the output generation we are using XSLT+XSLFO. We embed SVG, Open Office Charts, and XForms in those formats when appropriate. Those templates are created using a WYSIWIG editor. The output generation part of the platform uses a lot of NON-XML open standards, because certain channels don't support XML formats (e.g.: AFP, PS, PDF, PCL, … for printing/press). An exception to this are 'interactive documents', for which we use XHTML+XForms, which can be used in both desktop and mobile environments. The current work in the Open Web Platform and the rise of mobile are going to make it possible to interact with documents in ways we never imagined. In the delivery part of the platform we don't use a lot of XML because services like Print, FTP, SMTP, Facebook, Twitter, and the Apple Push Notification Service don't use XML as their data/communication format. Most of the modern channels have a REST service, but they either encode all data as parameters or use JSON as a data format. There are exceptions like the Windows Push Notification Services, which use an XML grammar as a data format.

Tracking how people interact with your communication is a really important part of the platform. Did the addressee(s) open the communication, did they respond to it, did they share it? All this information is collected, tracked using XQuery Update Facility, and fed back into the SCXML instance handling the communication, if requested.

To be able to get a full overview of all your communications, we are tracking all individual steps of the communication process using XQuery and XQuery Update Facility. The User Interface to manage, consult and monitor the complete system is written in XForms in which we use XQuery to retrieve the data and XQuery Update Facility if modifications to the data are required (e.g.: approval, set channel/communication on hold).

3. XML Standards used

In this section we will discuss for each standard:

- Why we have chosen that particular standard
- What extensions to the standard were needed to meet our needs
- What the challenges were using that standard
- The implementation of the standard (either off-the-shelf or custom-made)

3.1. SCXML

State Chart XML (SCXML) [1], a general-purpose event-based state machine language that combines concepts from CCXML and Harel State Tables. It is used in the platform to define the communication process at a high level. In the state chart you can define for example the approval process, the channel(s) to use for the communication, fallback states used when the message couldn't be delivered, delivery schedules, ...

3.1.1. Extensions to SCXML

We registered one custom invoker to send a communication.

3.1.1.1. Send communication invoker

This invoker sends a communication over the specified channel (print, email, SMS, ...). You can optionally specify the delivery intervals (on which days and between which hours) and how long the message is valid for delivery. When the message is sent, can't be send, or times out events are send to the state machine, which can be used to trigger specific behavior in the state machine. Depending on the output channel extra events can be sent (e.g.: Delivery notification for SMS)

3.1.2. SCXML challenges

The state machines for for the communication process typically stay alive for multiple days or even weeks. The Apache Commons SCXML implementation by default keeps all running state machine in memory and does not have support for multiple servers handling the same state machines.

We have implemented a high available event dispatcher that supports persisting running state machine instances and clustering of multiple servers which automatically distributes the work over all servers. When the load of one server gets to high, the running state machine instances are automatically distributed over the other servers.

3.1.3. SCXML implementation

We are using the Apache commons SCXML implementation [11].

3.2. XQuery and XQuery Update Facility

XQuery [3] is an XML query language, capable to extract and manipulate the information content of diverse data sources including structured and semi-structured documents, relational databases, and object repositories.

XQuery is used for the platform to retrieve the personalization data and obtain the data to represent in the management UI. With personalization data we mean the data that is used to create the communication, but also the data that is used to decide which communications to send, when and how.

In the platform we also need to be able to update that data. We have chosen XQuery Update Facility for this. XQuery Update Facility [4] is an extension to XQuery, which adds support for making persistent changes to instances of the XQuery and XPath data model. It supports operations like inserting, deleting and changing nodes.

3.2.1. XQuery challenges

Customers prefer to store their data in a relatational database because they already have the infrastucture and expertise to manage the relational database. We developed an XQuery engine which supports a subset of XQuery and store its data in a relational database. We store every element as a row in a table specific for every element type.

Working around database quirks is another challenge. An example of such a quirk is that an empty string on Oracle becomes NULL in the database. Since we consider a NULL value as the absence of the attribute on the element this poses a problem. We chose to prepend every string with a space character in oracle, but this of course complicates all operations on string in orcale because we have to counter the extra space character.

For performance reasons you want to map one XQuery expression to ideally one SQL query. The hierarchical structure of XML makes it really challenging to only use one SQL query, with the current feature set of XQuery that we support, we are able to map one XQuery expression to one SQL query.

It is an ongoing effort to support more of the XQuery specification. Mapping everything to SQL is hard (or maybe even impossible), some parts of the spec are difficult to implement without sacrificing performance in either the retrieval or insert/update process.

XQuery knowledge is not that common, but because XPath is used as a selection language in XQuery we have seen that the learning curve isn't as steep as we feared. Nevertheless we created a wizard to construct XQuery expressions for the most common use cases. This allows business users to create the queries.

3.2.2. XQuery implementation

The query engine and type of the data store is pluggable. Currently we have written our own XQuery engine which supports a subset of XML and store its data in a relational database. It is possible to use an off-the-shelf XML database and use its XQuery engine.

3.3. XSLT

XSLT [6] is a language for transforming XML documents into other XML documents. We use it both for transform and merge personalisation data, and for conditional structural formatting of the contents of the communication.

3.3.1. Extensions to XSLT

Customers can plug-in their own custom function library containing business and project specific extension functions. Customers use this to abstract complex expressions and business logic. The functions in the custom function library become available in our XPath query builder, which simplifies the creation of XPath expressions.

It is a common practice to re-use XSLT+XSL-FO blocks in multiple XSLT+XSL-FO documents. There are use cases in which you don't know upfront which XSLT+XSL-FO blocks you want to re-use in specific XSLT+XSL-FO document, because they are selected by the input data (e.g.: Creating a contract based on standardized paragraphs. The number of paragraphs changes over time, and you don't know all the paragraphs while designing the contract template.). To facilitate this use case, we have created an extension element which returns the ouput of another XSLT+XSL-FO stylesheet, and supports passing in arbitrary parameters.

3.3.2. XSLT challenges

Memory consumption while processing large XML source documents is currently our main challenge. The introduction of the streaming mode in XSLT 3.0 is going to make life a bit easier for us. In order to use this new feature in the output generation part of the product we will need to make adjustments to our WYSIWYG XSL-FO editor, and we are not yet completely sure how to integrate this in our product without introducing extra complexities for the document author.

3.3.3. XSLT implementation

We use Saxon PE [14] as our XSLT engine.

3.4. XSL-FO

XSL Formatting Objects [5], or XSL-FO, is a markup language for XML document formatting designed for paged media with features like advanced page layouts with citations, cross-references, running headers/footers, and running totals. In combination with XSLT it allows powerfull conditional structural formatting.

When we generate output (PDF, PS, AFP, HTML5, ...) we always generate XSL-FO first, then format the content and finally render to the specific output format. We currently have 18 different output formats, ranging from mainstream print and press formats, over web and mobile to less common formats like ZPL and TCPL.

3.4.1. Extensions to XSL-FO

For creating more graphical and interactive documents, we mix-in other open standards like SVG, XForms and Open office charts. We've also added some extensions to support features like rounded corners on all block level elements and a barcode markup language to represent barcodes in XSL-FO.

Because most output formats support meta-data (e.g.: TLE's in AFP or PDF annotations) we also added an extension for meta-data. This extension was added before RDFa existed, which we would have used if it existed when we added the extension.

We also added extensions for specifying finishing features like duplex printing, input and output trays.

Some output specific features are so important that we created extensions that are only applicable for one output format. An example of this is the AFP overlays extension. It allows a user to specify the position and the AFP page or medium overlay (resource and library) to use when printing the page.

At XSLT time pagination information is not available, therfore we created an extension called 'Page Data'. It creates an XPath data model, while formatting the XSL-FO, on which XPath expressions can be excuted. This extension can for example be used to calculate the sum of all item prices on the current page.

3.4.2. XSL-FO challenges

When we started writing our own XSL-FO formatter in 2002 there were no other implementations we could use that met our performance, memory and quality requirements. We have to be able to generate small on-demand documents but also large batch documents with millions of pages. In those large batch documents we have to support nested tables of which the cells of the outer table span hunderds of pages for example. This required us to serialize parts of the XSL-FO tree because we can't keep the complete table row in memory while rendering it. There are of course a lot of other XSL-FO related challenges like resolving the correct page masters, keeps/orphans/widows, and cross-references.

We have chosen to separate the XSLT and XSL-FO formatting processes. This means a lot of the markup is repeated in the generated XSL-FO document, which introduces performance and memory challenges. To work around this challenge we are first optimizing the XSLT+XSL-FO template by removing superfluous XSL-FO attributes and afterwards replacing groups of attributes with attribute group references. The second optimization is optional because other XSL-FO formatters won't know this extension.

3.4.3. XSL-FO implementations

We created our own XSL-FO processor see Section 3.4.2 for which we created our own processor.

3.5. XForms

XForms [2] is a cross device, host-language independent markup language for declarativly defining a data processing model of XML data and its User interface. It uses a model-view-controller approach. The model consists of one or more XForms models describing the data, constraints and calculations based upon that data, and submissions. The view describes what controls appear in the UI, how they are grouped together, and to what data they are bound.

We use XForms for interactive communications as wel as the management User Interface. For the interactive communications we first embed XForms in XSL-FO and convert it to XForms in XHTML. Until a couple of years ago we had a native client that accepted XSL-FO + XForms, but we discontinued it in favour of using only XHTML+XForms. Due to the recent changes in web browsers, driven by the HTML5 effort, there were no advantages anymore of having a native client.

3.5.1. Extensions to XForms

We used a lot of extensions which are standardized in XForms 1.1 or are in the XForms 2.0 specification which is about to go to last call. Examples of those are Attribute Value Templates on both XForms elements and on host langague elements, variables, multiple binds for the same MIP on the same node, XPath 2.0 expressions, and repeats over sequences of atomic values and nodes.

For creating the management UI and interactive communications we use a lot of custom components. We are using components that abstract a paging system to browse through large sets of data, auto complete controls, tab controls, ...

The custom components are using an enhanced version of XBL, but we never managed to standardize a custom control framework for XForms in the Forms WG.

3.5.2. XForms challenges

There are a lot of different web browsers which support different features and all have their own quirks, which makes creating an XForms processor that correctly works in all different browsers challenging. Recent versions of all browsers are implementing the standards better, which makes life easier for those, but we have to still support older versions too.

3.5.3. XForms implementations

We are using off-the-shelf XForms implementations. When we added XForms support to the platform we decided to use Chiba [12]. We contributed a lot back to the chiba project. For the management UI we are using Orbeon [13], because they have better custom components support and they do some clever performance enhancements.

4. Conclusion

Using open standards where possible and discussing the rough edges and 'missing' features in the current standards with their standardization bodies, ensures that the platform is future proof and easy to integrate in/with other systems. We collaborated in the XSL and XForms working groups at the W3C and other non-XML standards like AFP to ensure this. Working together with other people on those standards helped us a lot in better understanding the best practices and what is possible with those standards.

Other advantages of using open standards are:

- Documentation: there already is a lot of good documentation written on the internet and in books on how to use the standard.
- Re-use of off-the-shelf components: we have re-used a lot of open source implementations and commercial implementations. This decreased our time to market and allowed us to focus on the platform.
- Prevents vendor lockin: Our customers are free to replace one (or all) of the components with there own.

The platform is used today as a multi-channel communication platform that helps enterprises worldwide improve their customer communications, leading to increased customer engagement and loyalty.

References

[1] http://www.w3.org/TR/scxml/

[2] http://www.w3.org/TR/xforms20/

[3] http://www.w3.org/TR/xquery/

[4] http://www.w3.org/TR/xquery-update-10/

[5] http://www.w3.org/TR/xslfo20/

[6] http://www.w3.org/TR/xslt20/

[7] http://www.afpcinc.org/site/assets/files/1120/modca08.pdf

[8] http://www.json.org/

[9] http://www.adobe.com/products/postscript/pdfs/PLRM.pdf

[10] http://www.iso.org/iso/iso_catalogue/catalogue_tc/
catalogue_detail.htm?csnumber=51502

[11] http://commons.apache.org/scxml/

[12] http://chiba.sourceforge.net/

[13] http://www.orbeon.com/

[14] http://www.saxonica.com/

An XML Solution for Legal Documents

George Bina
Syncro Soft / oXygen XML Editor
`<george@oxygenxml.com>`

Abstract

All companies deal with legal documents. They are generally maintained in unstructured formats that do not allow reuse while most of these legal documents share common parts that should stay the same in all documents. We discovered that we have many end user license agreements, with very similar content and keeping them synchronized and making sure everything is up to date quickly become a challenge. This presentation shows the XML based solution we adopted to solve this problem that allows us to write once and publish in every place we need that information to be. Then we extended this to cover also the SDK agreement as well as our reseller agreements.

Keywords: XML, legal, documents, review, publishing, single source, reuse, XML vocabularies

1. Introduction

We are a software company that produces one product, oXygen, that is made available under a few profiles: oXygen XML Author - for content authors, oXygen XML Developer - for XML developers and oXygen XML Editor which includes both the oXygen Author and the oXygen Developer.

These products have different editions: Academic, Professional and Enterprise and some of these editions may be licensed to apply not only to a single user but also to a range of users like all the people from a classroom or from a department.

As expected the end user license agreement is mostly the same but for particular products or editions or ranges there may be particular terms that apply only to that specific case. More, the end user license agreements should be available both online on our website as well as integrated inside the products.

Along with the products themselves there is also a product SDK and that comes with an SDK agreement. There are also partners that can either simply resell product licenses or they may integrate our products in a solution and resell them as part of that solution - value added resellers (VARs). The relation with these partners is defined again in legal agreements that again are very similar but they should also capture the difference between a normal reseller and a VAR.

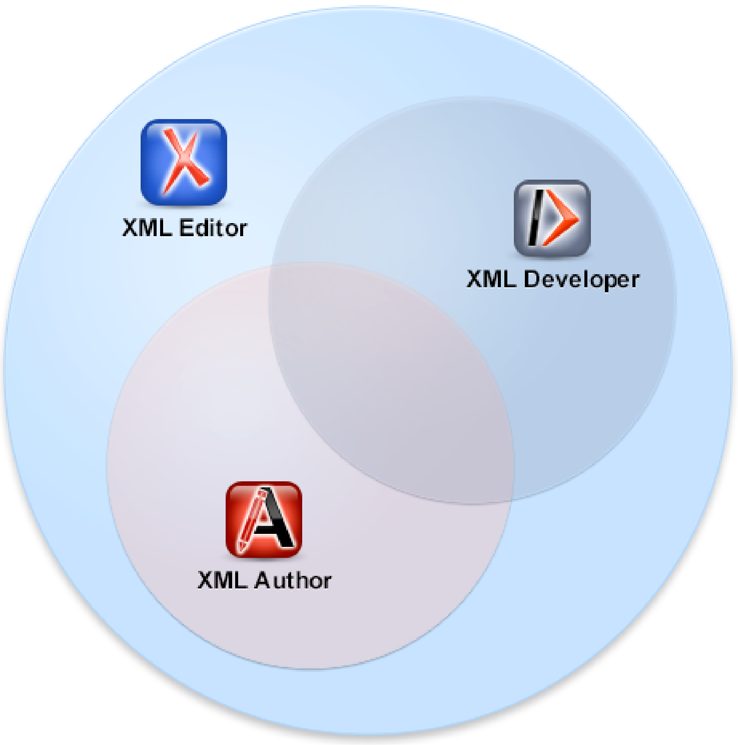


Figure 1. oXygen family of products

These numbers get multiplied and you end up easily with a large number of similar documents that you need to maintain.

As we noticed this problem and having experience with XML solving similar problems in technical documentation the next step was to think on how we can take some of the single sourcing concepts that are in use in technical documentation solutions and apply them for legal documents.

2. Possible solutions

One possibility is to base the legal documents on an existing framework like DocBook or DITA and define the structure, constrains and processing requirements of legal document under the customization capabilities of these frameworks. The other possibility will be to develop an XML language from scratch.

We decided to take the second approach, defining our own schema due to a number of reasons:

- We needed a very simple vocabulary with elements easily understandable by people working with legal documents
- While DITA allows defining your own elements through specialization we had only very basic needs that may not justify the effort to base this on DITA

- We did not have all the requirements from the start and we wanted to be able to easily evolve the system as we needed while working within another framework automatically comes with some constraints

3. Legal documents schema

The schema has only 10 elements so it is very easy to understand by anyone.

These elements are:

1. `eula` - marks an end user license agreement document.
2. `addendum` - marks an addendum to an existing agreement or to another addendum.
3. `info` - contains information about the document, initially it holds only the document title.
4. `title` - marks a title.
5. `important` - this marks a special section that contains very important information that needs to be emphasised.
6. `section` - marks a regular section.
7. `item` - contains an paragraph or idea in the agreement.
8. `list` - groups a set of items together.
9. `phrase` - allows to set profiling attributes to a specific piece of content.
10. `ref` - allows to make a reference to a section or to an item identified by an xml:id identifier.

The main schema structure is described in the following two diagrams.

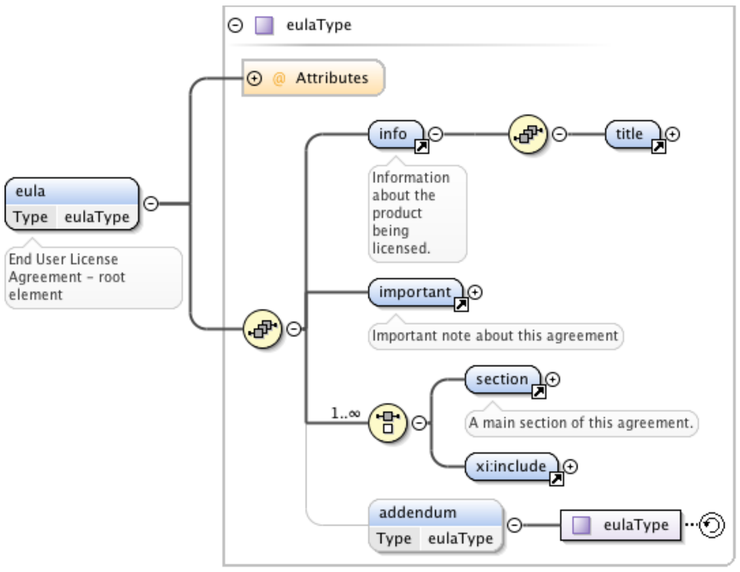

eulaType
Attributes
eula
Type eulaType
End User License Agreement – root element
info
Information about the product being licensed.
title
important
Important note about this agreement
1..∞
section
A main section of this agreement.
xi:include
addendum
Type eulaType
eulaType

Figure 2. The `eula` element

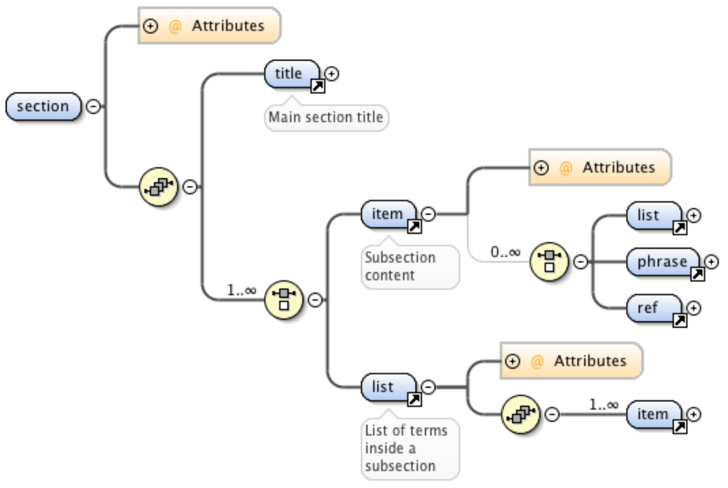
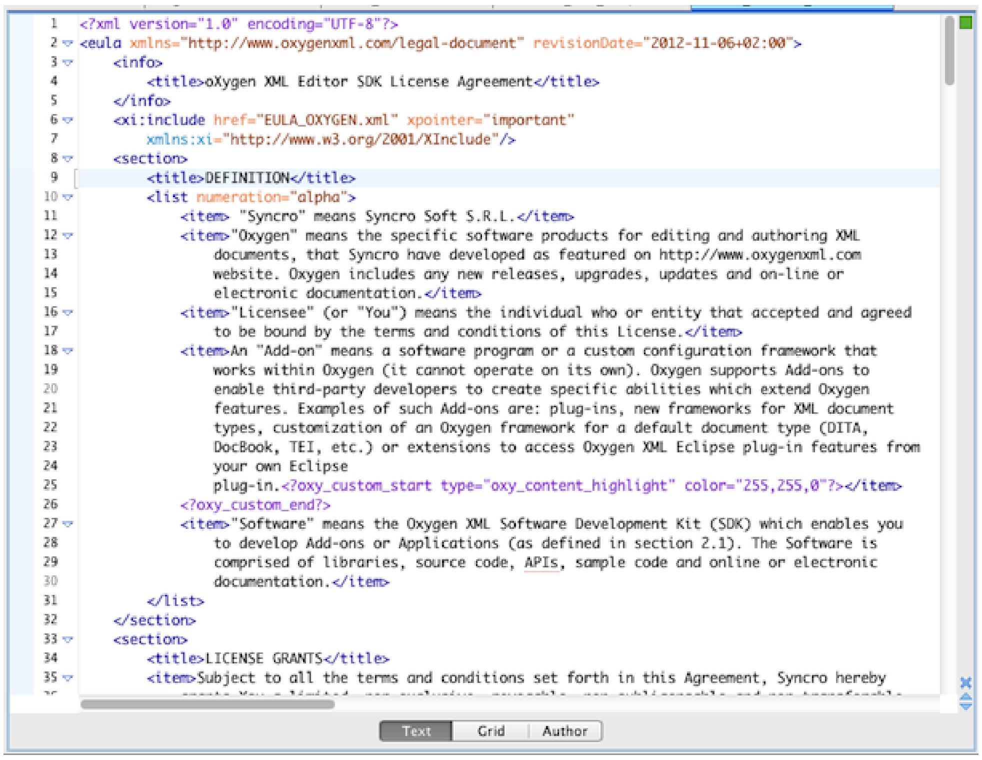


Figure 3. The `section` element

4. Authoring and review

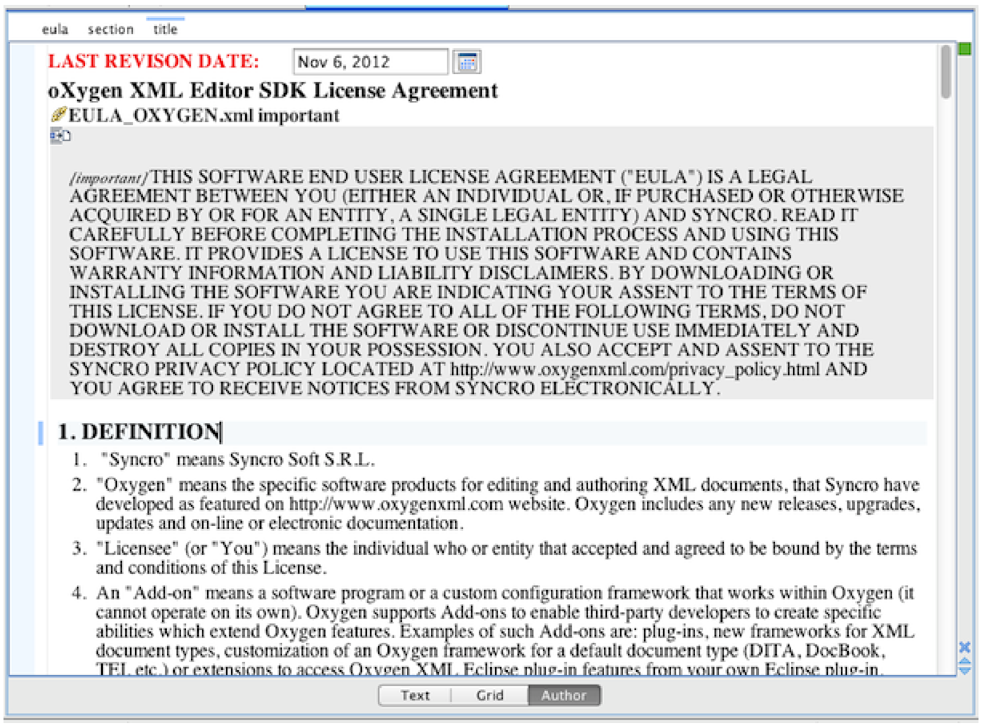
As XML is text, authoring can be done with any text editor at the XML source level. However, a real solution needs to take into account that people interacting with legal documents are generally non technical users and may be intimidated by the XML markup.



```
<?xml version="1.0" encoding="UTF-8"?>
<eula xmlns="http://www.oxygenxml.com/legal-document" revisionDate="2012-11-06+02:00">
    <info>
        <title>oXygen XML Editor SDK License Agreement</title>
    </info>
    <xi:include href="EULA_OXYGEN.xml" xpointer="important"
        xmlns:xi="http://www.w3.org/2001/XInclude"/>
    <section>
        <title>DEFINITION</title>
        <list numeration="alpha">
            <item> "Syncro" means Syncro Soft S.R.L.</item>
            <item>"Oxygen" means the specific software products for editing and authoring XML
                documents, that Syncro have developed as featured on http://www.oxygenxml.com
                website. Oxygen includes any new releases, upgrades, updates and on-line or
                electronic documentation.</item>
            <item>"Licensee" (or "You") means the individual who or entity that accepted and agreed
                to be bound by the terms and conditions of this License.</item>
            <item>An "Add-on" means a software program or a custom configuration framework that
                works within Oxygen (it cannot operate on its own). Oxygen supports Add-ons to
                enable third-party developers to create specific abilities which extend Oxygen
                features. Examples of such Add-ons are: plug-ins, new frameworks for XML document
                types, customization of an Oxygen framework for a default document type (DITA,
                DocBook, TEI, etc.) or extensions to access Oxygen XML Eclipse plug-in features from
                your own Eclipse
                plug-in.<?oxy_custom_start type="oxy_content_highlight" color="255,255,0"?></item>
            <?oxy_custom_end?>
            <item>"Software" means the Oxygen XML Software Development Kit (SDK) which enables you
                to develop Add-ons or Applications (as defined in section 2.1). The Software is
                comprised of libraries, source code, APIs, sample code and online or electronic
                documentation.</item>
        </list>
    </section>
    <section>
        <title>LICENSE GRANTS</title>
        <item>Subject to all the terms and conditions set forth in this Agreement, Syncro hereby
```

Figure 4. A document in Text editing mode

Different tools can render XML based on CSS, some limited form of XSLT, custom configurations, etc. As oXygen uses CSS for rendering XML we developed a simple CSS file to render the legal XML documents. The CSS mainly identifies the main block elements, the lists and items adding counters to number them. The rendered document looks now similar with a word processor document.



LAST REVISON DATE: Nov 6, 2012

oXygen XML Editor SDK License Agreement

EULA_OXYGEN.xml important

[important] THIS SOFTWARE END USER LICENSE AGREEMENT ("EULA") IS A LEGAL AGREEMENT BETWEEN YOU (EITHER AN INDIVIDUAL OR, IF PURCHASED OR OTHERWISE ACQUIRED BY OR FOR AN ENTITY, A SINGLE LEGAL ENTITY) AND SYNCRO. READ IT CAREFULLY BEFORE COMPLETING THE INSTALLATION PROCESS AND USING THIS SOFTWARE. IT PROVIDES A LICENSE TO USE THIS SOFTWARE AND CONTAINS WARRANTY INFORMATION AND LIABILITY DISCLAIMERS. BY DOWNLOADING OR INSTALLING THE SOFTWARE YOU ARE INDICATING YOUR ASSENT TO THE TERMS OF THIS LICENSE. IF YOU DO NOT AGREE TO ALL OF THE FOLLOWING TERMS, DO NOT DOWNLOAD OR INSTALL THE SOFTWARE OR DISCONTINUE USE IMMEDIATELY AND DESTROY ALL COPIES IN YOUR POSSESSION. YOU ALSO ACCEPT AND ASSENT TO THE SYNCRO PRIVACY POLICY LOCATED AT http://www.oxygenxml.com/privacy_policy.html AND YOU AGREE TO RECEIVE NOTICES FROM SYNCRO ELECTRONICALLY.

1. DEFINITION

1. "Syncro" means Syncro Soft S.R.L.
2. "Oxygen" means the specific software products for editing and authoring XML documents, that Syncro have developed as featured on http://www.oxygenxml.com website. Oxygen includes any new releases, upgrades, updates and on-line or electronic documentation.
3. "Licensee" (or "You") means the individual who or entity that accepted and agreed to be bound by the terms and conditions of this License.
4. An "Add-on" means a software program or a custom configuration framework that works within Oxygen (it cannot operate on its own). Oxygen supports Add-ons to enable third-party developers to create specific abilities which extend Oxygen features. Examples of such Add-ons are: plug-ins, new frameworks for XML document types, customization of an Oxygen framework for a default document type (DITA, DocBook, TEI, etc.) or extensions to access Oxygen XML Eclipse plug-in features from your own Eclipse plug-in.

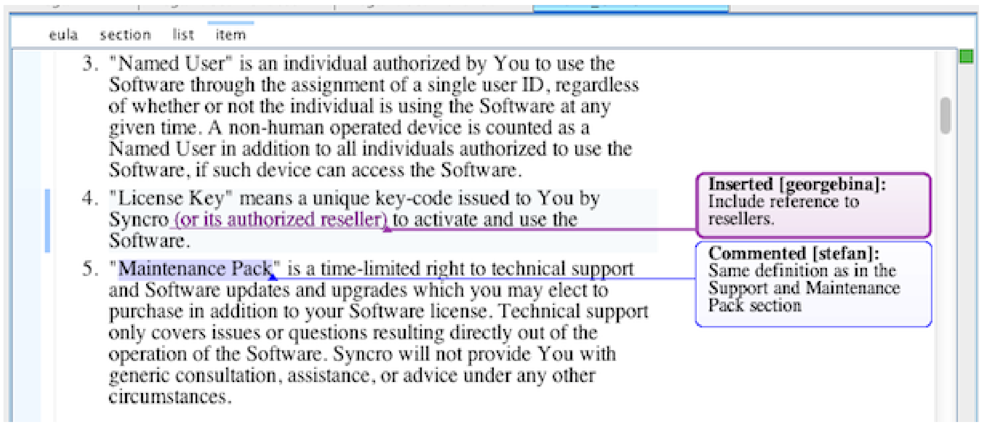
Figure 5. A document in visual editing mode

Having a very simple schema with a total of 10 elements, the choices an author has at any moment are very limited, at most 3 elements can be validly inserted at any moment. This coupled with the visual editing of the document and annotating the elements with descriptions to specify their purpose simplifies the authoring process. This can be improved further by adding actions to insert specific elements and make them available as buttons in the toolbar or in the contextual menu.

Once legal documents are created the next step is to review them. The review support is essential for legal documents. This includes not only annotations or comments on specific parts of the document but proposed document changes should be tracked as well.

XML track changes is not standardised although there is an initiative in this direction (this is actually the subject of one of the XML Prague presentations this year). So, until there will be a standard the track changes are preserved usually using product specific processing instructions. The main advantage of using processing instructions is that the XML vocabulary does not need to provide special elements or attributes for tracking changes or storing annotations or comments. This means that we get this functionality for free for any XML format, we only nee to choose a tool that supports track changes and annotations stored as processing instructions.

Here it is an example of how the changes and comments look like in oXygen XML editor:



3. "Named User" is an individual authorized by You to use the Software through the assignment of a single user ID, regardless of whether or not the individual is using the Software at any given time. A non-human operated device is counted as a Named User in addition to all individuals authorized to use the Software, if such device can access the Software.
4. "License Key" means a unique key-code issued to You by Syncro (or its authorized reseller) to activate and use the Software.
5. "Maintenance Pack" is a time-limited right to technical support and Software updates and upgrades which you may elect to purchase in addition to your Software license. Technical support only covers issues or questions resulting directly out of the operation of the Software. Syncro will not provide You with generic consultation, assistance, or advice under any other circumstances.

Inserted [georgebina]: Include reference to resellers.

Commented [stefan]: Same definition as in the Support and Maintenance Pack section

Figure 6. Comments and change tracking

In addition to comments, insertions and deletions there are also highlights. These are useful for individual review - for example one can mark, during an initial read of an agreement, with orange the parts he should come back to, with red the critical sections he should exclude, and so on.

The authoring support presented so far together with a simple schema provide an equivalent solution to a word processor.

5. Save time with reuse

Let's explore now how we can improve on basic authoring and review support adding in new concepts. Some sections stay the same in many documents so instead of writing them again we should be able to write them once and just refer to an instance of that section to include it in another legal document.

One evident advantage is that we do not have to write the same section twice. Also when we need to make a change we do that in only one place and we know that it the change will be propagated in all places. Another advantage is related to reviewing - if that section was already reviewed then when we need to review a new document then we can skip the reused sections that we know were already reviewed.

Some documents implement their own inclusion mechanism, for example DITA provides content references, but others rely on the XInclude standard. In our case we decided to take advantage of XInclude. We based this decision on the fact that the XInclude support can be found already implemented by authoring tools as well as by processing tools.

For adding XInclude support we need to take XInclude into account in the schema definition. First, if we need the XInclude elements to be available for authoring then the schema needs to define the XInclude elements. After the XInclude processing is applied on a document, some attributes may appear in the document due to the base URI fixup and language fixup, These attributes are the standard `xml:base` and `xml:lang` attributes and the schema should specify them on the elements that can be included using XInclude.

We grouped these two attributes in an attribute group called `xmlAttributes` and we made them available on the `section`, `important`, `addendum`, `eula`, and `item` elements - only these elements can be reused though XInclude.

In our end user license agreement documents we actually reuse in the SDK end user license agreement 7 sections from the oXygen end user license agreement out of a total of 16 sections. This is a very good reuse percentage!

Coming back to the review advantage, included content is marked in the user interface with a gray background so it is easy to visually recognize it and skip those parts during a review.

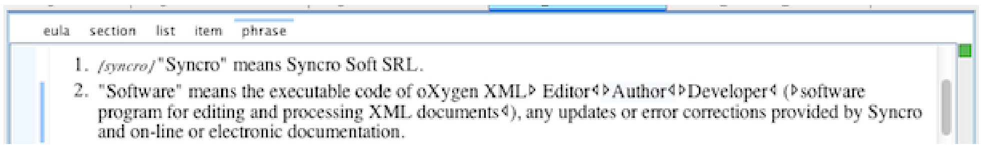
6. Conditional content

One of the problems we tried to solve was that we have very similar end user license agreements for similar products. While the reuse helps us to avoid duplicating sections or items between two documents it is not the perfect solution when only a couple of words are different in a section. Here we have another concept that solves this problem - conditional content also known as profiling.

We can have one document and different condition sets and by applying each condition set to the document we can produce different deliverables. That means also that we need to mark content with a specific condition that can then be checked against the current condition set.

We identified three characteristics for which we want to conditionally include/exclude content: *product*, *edition* and *license range*. For each of these we defined attributes to allow us to flag different parts of the document as belonging to specific products, editions or license ranges, respectively. These attributes were grouped together in an attribute group called `profillingAttributes` and we made them available on `section`, `addendum`, `eula`, `list`, `item` and `phrase` elements. The possible values for these attributes are defined in enumerations so we can both constrain the values that appear in the document to a specific list but also to be able to assist the user by proposing the possible values for these profiling attributes.

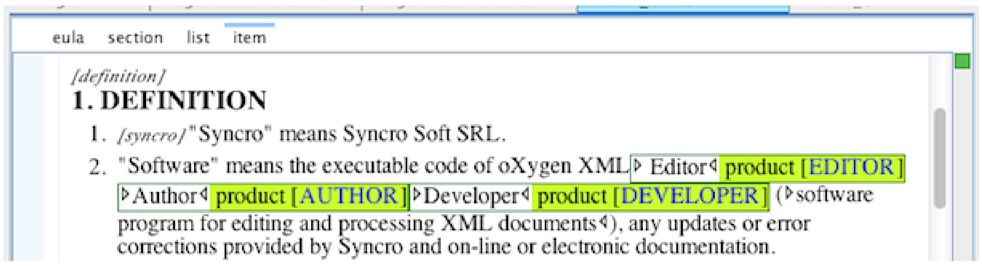
Conditional content may get difficult to work with without specific support from the authoring tool. Here we have an example where we use the `product` profiling attribute to specify the product name for each product.



1. [syncro] "Syncro" means Syncro Soft SRL.
2. "Software" means the executable code of oXygen XML▷ Editor◁▷Author◁▷Developer◁ (▷software program for editing and processing XML documents◁), any updates or error corrections provided by Syncro and on-line or electronic documentation.

Figure 7. Conditional support disabled

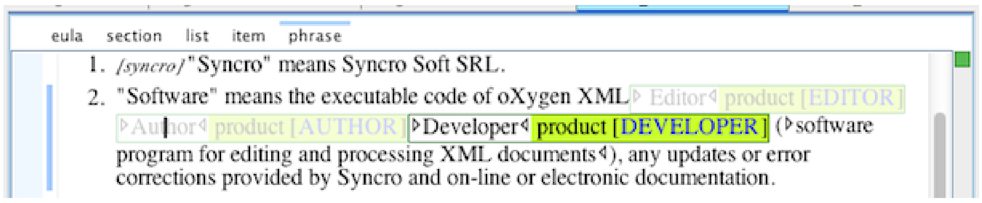
The conditional content can be marked on the screen to allow the author see the condition under which a specific content will appear.



[definition]
1. DEFINITION
1. [syncro] "Syncro" means Syncro Soft SRL.
2. "Software" means the executable code of oXygen XML▷ Editor◁ product [EDITOR] ▷Author◁ product [AUTHOR] ▷Developer◁ product [DEVELOPER] (▷software program for editing and processing XML documents◁), any updates or error corrections provided by Syncro and on-line or electronic documentation.

Figure 8. Conditional support enabled

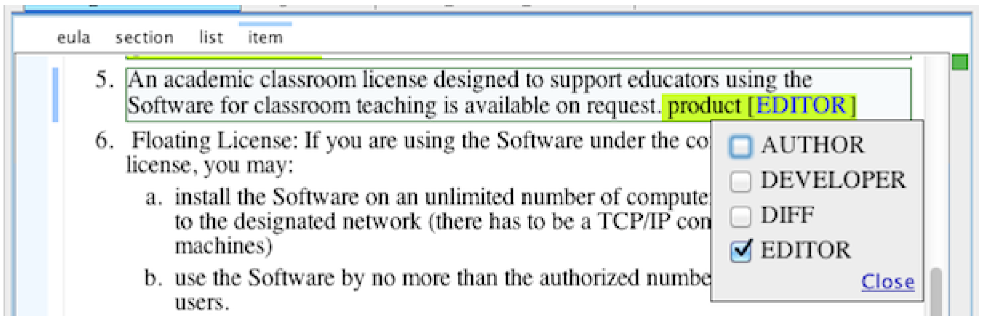
One aspect the authoring tool should support is related to being able to focus on the content that is included under a specified condition set. Here we activated a condition set that matches DEVELOPER product value and the other products will fade out allowing us to focus on the content that will be matched by the condition set.



1. [syncro] "Syncro" means Syncro Soft SRL.
2. "Software" means the executable code of oXygen XML▷ Editor◁ product [EDITOR] ▷Author◁ product [AUTHOR] ▷Developer◁ product [DEVELOPER] (▷software program for editing and processing XML documents◁), any updates or error corrections provided by Syncro and on-line or electronic documentation.

Figure 9. Faded out parts that do not match the current conditions

Another usability aspect is related to managing the conditional content, how we can easily change the conditions for a specific content. One possibility is to change directly the profiling attribute values but providing a more user friendly support helps with non technical people. For example allowing to change the profiling attribute values by presenting them using a standard set of check boxes is easier for a non technical person.



eula section list item

5. An academic classroom license designed to support educators using the Software for classroom teaching is available on request. product [EDITOR]

6. Floating License: If you are using the Software under the co license, you may:
   a. install the Software on an unlimited number of compute to the designated network (there has to be a TCP/IP con machines)
   b. use the Software by no more than the authorized numbe users.

AUTHOR [ ]
DEVELOPER [ ]
DIFF [ ]
EDITOR [✓]

Figure 10. Setting values for the product profiling attribute

7. References

References to other sections or items can be made textually in the document, but then if you make a change in the document that may result in a different number for that section or item then all those references need to be updated manually to reflect the new value. More, if we use conditional content and add a section only for a specified condition then that section will appear or will disappear depending on the current condition set. Thus the reference needs to use similar conditions to provide different numbers as the profiling condition set deliverables will contain.

To avoid all these problems we added support for referring a section or an item based on an identifier defined on that section or item and then the output processing will take care of generating the correct section or item number according with that deliverable, that means taking into account the reused content and the conditional content.

8. Publishing

The publishing is based on XSLT. We split this is a few stages. First we have a stylesheet that triggers the processing for each condition set we are interested in, obtaining a deliverable for each. The processing is a micro-pipeline that triggers different stages of processing: resolve XInclude elements (if not resolved already by the XML parser), applies conditional processing, resolve references and applies formatting rules.

Because we need these documents to appear together with the product, on the website, etc. we orchestrated everything using Ant. This allows us to issue a single command and have everything pushed in the product repository, in the website repository.

9. Conclusions

This shows a real use case for an XML based solution that improves the way we handle our legal documents - now all the information is kept in one place and when we change and publish we know that all our changes go automatically in all versions and in all places each version is used.

Although the example shows oXygen, similar support is available also in other XML tools.

The current state of XML tools allow you to use XML instead of a word processors providing comparable features and ease of use with the bonus of single source to avoid duplicates and markup to enable validation and automatic processing.

This project is made available under Apache 2.0 license on GitHub at `https://github.com/georgebina/legal`.

Conveying Layout Information with CSSa

XML and HTML Cross-Pollination (cont.)

Gerrit Imsieke
le-tex publishing services GmbH
`<gerrit.imsieke@le-tex.de>`

Abstract

This paper introduces the concept of CSS properties as attributes (CSSa) for transporting layout information in XML document conversion pipelines. CSSa may be regarded as an orthogonal layer for presentation[1]*, analogous to RDFa as an orthogonal layer for semantics.*

The paper sketches components of a formal CSSa specification, particularly a Relax NG schema for CSSa property names and permitted values, as per the underlying CSS specs. It then presents some CSSa applications: a CSS→CSSa parser for HTML documents; "Hub XML," a rather low-level interchange format for formatted documents, based on flattened DocBook and CSSa; a property mapping for IDML; device-specific CSS validation profiles (for example, checking Kindle compatibility of an EPUB file) that fit in the RNG/Schematron-based epubcheck infrastructure; a conversion pipeline where CSSa plays a crucial role in several regards: Word manuscript → Hub XML → custom TEI → HTML preview → IDML (InDesign).

Keywords: CSS, XML, presentation semantics, Schematron, OOXML, IDML, HTML, EPUB

1. Introduction

In recent years, publishers have been increasingly adopting so-called media-neutral XML workflows. These are workflows where, at some stage of the process, the content is converted to or authored as XML. The XML dialects in use range from generic vocabularies such as DocBook, TEI, or NLM/JATS to bespoke, publisher-specific vocabularies. While all of these vocabularies are suited to model the content structure (section hierarchy, lists, …), they vary in supporting two other orthogonal aspects: whether and if so how to capture semantic and layout information.

[1]more exactly: for representing presentational information of formatted documents

1.1. Representing Semantics and Layout in Orthogonal Layers

We see three principal ways to capture these dimensions:

1. Specified in the grammar, often as self-explanatory element names, with certain documented expectations regarding interpretation or rendition of the element's content. Semantic example: in a DTD for history books, we saw <battle> and even <person god="yes">. Layout example: in JATS and other DTDs, there's <bold>, <italic>, and <underline>.
2. Specified as free-text attribute values (of a @role, @type, @content-type or @class attribute), interpretation or rendering conventions are established within a possibly narrow user group. Ad-hoc processing rules (instead of community-wide conventions or machine-applicable logic) convey the intended ontological and presentational semantics.
3. Specified in a formal, machine-readable language that is orthogonal to the structure markup. For semantics, there is RDFa. CSSa may be regarded as a machine-processable analog to RDFa in the layout domain.

1.2. Central Styling and Local Overrides

Common wisdom stipulates that layout information stay out of the content and rather be applied by means of an external stylesheet. Most vocabularies follow path 2 of the list above: provide a generic attribute that a style may be bound to.[2] While it is easy to apply a wholesale style to such layout-agnostic XML documents, fine-grained layout control is a thing not anticipated in these XML vocabularies.

The selector/query languages of CSS, DSSSL and other languages both allow selecting elements by ID or by another attribute value, which in principle allows for individual styling of content while keeping content and styling separated. In addition, CSS, or rather HTML plus CSS by virtue of the @style attribute, allows local styling overrides to be inlined. Although this doesn't increase the expressiveness of CSS (both variants, local @style overrides and ID-referenced external rules, can be transformed into each other without loss of information), it frees the document author from the need to assign IDs and maintain a list of overrides in a separate document.

While the requirement for local overrides may not arise for many classes of publications that will be automatically rendered from XML, book publishers want both: media-agnostic structural markup of their content and automatic creation from one layouted edition of their books to another (e.g., print to EPUB). If EPUB is created from a media-neutral XML file, stripped of all formatting information, it needs to be styled by a stylesheet. That makes it difficult to convey deviating

[2]TEI is a notable exception; in addition to the generic @type attribute, it allows for formally attaching rendering information to elements, in CSS and other vocabularies [1].

formatting specifically for a single book (highlight colors, background colors, extra vertical spacing, …) and locally within a book.

One of the strengths of HTML+CSS is that it enables both central styling and local overrides while keeping the styling sufficiently separate in style elements and attributes.

1.3. CSSa's Purpose

Of the three main CSS constituents, namely selectors, properties, and priorities, CSSa chiefly adopts the *property* part. It utilizes the vocabulary, value space, and presentation semantics of established and upcoming CSS specifications. It aims at providing a common vocabulary for layout information interchange between XML-based document formats such as InDesign's IDML, Word's Office Open XML, and HTML[3]/EPUB. CSSa introduces *named* CSS rules as a means to capture the concept of named paragraph, character, etc. styles found in DTP and word processor applications. Just like in these applications, the use of named rules is optional in CSSa, and named rules' declarations may be overridden locally.

When confronted with an HTML attribute such as style="color:red; font-weight:bold; font-family: Verdana, sans-serif", XML processing tools need to parse this compound string in order to be able to apply Schematron or grouping by common styling to the content. In addition, if there is an additional class="foo bar" attribute, how can a checking application determine the font size? Maybe with Javascript. But if we were to adopt this CSS approach for the XML workflows and vocabularies that publishers are heavily invested in, the information must be readily available to XPath.

The main purpose of CSSa is to convey layout information in XML conversion workflows. CSSa serves as a common, application-agnostic vocabulary for expressing large subsets of the layout information found in document formats such as Office Open XML (MS Word), IDML (Adobe InDesign), or EPUB.

Apart from being application-neutral, CSSa's main advantages are:

- Rules and properties expressed in CSSa are trivially serializable to CSS rules or HTML style attributes;

  CSSa lends itself better to XML processing and checking tools than unparsed plain text CSS;

  Living in a namespace of their own, CSS attributes may easily be added to and stripped off arbitrary XML document types.

CSSa is for describing what is known in Word or InDesign as *named styles* in terms of CSS properties. These declarations can easily be separated from the content, so this alone does not justify carrying the layout information alongside with the content.

[3]We use the term HTML as a shorthand for an XML-based format such as XHTML 1.1 or HTML5's XML syntax.

In addition, and this is where the attribute part comes into play, CSSa can be used to describe local deviations from a centrally defined layout.

2. CSSa Specification

Currently, CSSa consists of

- a Relax NG schema [2] for attributes, permitted values, and named rules[4],
- a proposed XML namespace (http://www.w3.org/1996/css),
- (as of this writing) approx. 50 properties whose names and values are modeled by and large after the current CSS3 draft specifications,
- the concept of *named rules*[5],
- a mechanism to bind an attribute of the host vocabulary[6] (e.g., @role, @type, or @class) to named rules.

There is no written specification yet. This paper and the discussions at XML Prague 2013 may serve as a starting point.

In addition to attributes with the css: namespace prefix, additional attributes are permissible in the named rules: @xml:lang, @name (mandatory; permitted values: subset of the CSS class name value space [3]), @native-name (the original application's internal style name, possibly with characters that are not allowed in CSS class names) and @layout-type (para | inline | table | cell | object).

2.1. Extensibility

There are two entry points for extensions in the Relax NG schema: `custom.css-like.rule.attributes` for use with both named rule declarations and local overrides, and `custom.css_element_model` for use with named rule declarations. In Section 5, the custom model consists of a tabs element that may hold tab declaration elements. In Hub XML, the same tabs declaration may occur at the beginning of para content.

Another Hub extension is a linked-style declaration where the default para style for a cell style may be specified. It is yet an open discussion topic whether the basic CSSa specification should deal with this kind of style linking.

[4]A CSS rule is a selector plus a set of CSS declarations, as in

```
.dedication {
  text-align: center;
  font-size: 12pt;
}
```

[5]There is no such thing as a named rule in CSS. A named rule in CSSa is the declaration part of a CSS rule, plus a name. A CSSa rule with the name "foo" corresponds to a CSS selector `.foo` in HTML.

[6]The CSSa schema does not permit standalone CSSa documents. It always needs a host markup language.

2.2. Shorthand Properties

No attempt has been made so far to support CSS shorthand properties such as `border` or `margin`. These should be represented as individual properties. Otherwise, too much parsing would have to be done by processing tools or in Schematron rules.

2.3. Style Inheritance vs. Style Composition

For the sake of processing simplicity, CSSa does not adopt style inheritance, as found in most DTP and word processing programs. It would be easy to add a @based-on attribute (or, since style names are not necessarily unique, an element such as <based-on layout-type="para" name="Standard"/>) to a named rule. However, in order to find out what styling is in force at a given content element, not only the properties of the named rule and of local overrides would have to be considered. The named rule inheritance chain would have to be processed recursively, which puts too much of an onus on a Schematron processor and/or schema designer.[7]

Although style inheritance in rules is unsupported, style *composition* in the content is supported. Style composition happens when multiple space-separated values are used in the bound host vocabulary attribute (class, role, etc.).

If two named rules declare the same CSS property, precedence should be given to the property that is declared last in document order, as it is in CSS [5]. Local overrides should have precedence over properties from named rules. Consider the following example:

```
<para role="foo bar">Text</para>
```

This situation will not arise when converting from DTP / word processor formats (where only a single style may be attached to a paragraph, a character range, etc.) but when converting from CSS, translating class-based styles to named rules. The original CSS looks like this:

```
.bar {
  font-weight: bold;
  font-style: italic;
}
.foo {
  font-weight: normal;
}
```

and will be converted to

[7]Although this may be one of the rare cases where Rick Jelliffe might accept XSLT functions in Schematron [4], we think that it's better to do the recursion when converting to CSSa, rather than when checking Schematron assertions. Otherwise the Schematron processor would have to support XSLT2 function declarations, which doesn't work out of the box with oXygen or with XProc's `p:validate-with-schematron` step.

```
<css:rule name="bar"
  css:font-weight="bold"
  css:font-style="italic"/>
<css:rule name="foo"
  css:font-weight="normal"/>
```

An example for a typical Schematron assertion when checking hierarchized Hub (or DocBook proper for that matter, since any namespaced foreign elements / attributes such as css:rule / @css:font-weight are permitted there) is that there must be no all-boldface paragraphs. This is because all-boldface paragraphs are typically headings that should have been transformed into section or formalpara titles during hierarchization. There are two options for a Schematron author then: either preprocess the document so that the rules' and local overrides' attributes will be properly consolidated, or keep the document as is and consider the precedence rules in the @test attribute:

```
<rule context="dbk:para">
  <let name="roles" value="tokenize(@role, '\s+')"/>
  <let name="rules" value="/*/dbk:info/css:rules/css:rule[@name = $roles]"/>
  <report test="matches(
                 ('', ($rules | .)/@css:font-weight)[last()],
                 '^(bold|[6-9]00)$'
               )">Is this all-boldface paragraph a heading?</report>
```

The expression `('', ($rules | .)/@css:font-weight)[last()]` makes sure that the last @css:font-weight in document order wins (or the empty string, if there is no font weight attribute). In the example above this is the `@css:font-weight` of the last rule (foo), which is 'normal'.

If the example read

```
<para css:font-weight="900" role="foo bar">Text</para>,
```

a Schematron validation error would be raised because of the local font weight override.

3. Application 1: A CSS→CSSa Parser

The CSS declarations of an HTML file are being parsed to an XML representation. The precedence rules of the CSS cascade are translated to priorities, and the CSS selectors are translated to XPath (or rather, XSLT patterns). An XSLT stylesheet is generated out of this information and applied to the content. Then all styling is available locally, for Schematron checks as described in the next section, or for translating table cell widths from a styled HTML preview document to InDesign table cell widths.

By supplying the parser with the parameters `path-constraint` and `prop-constraint`, it is possible to restrict expansion to certain contexts or properties.

For example, the docx→idml converter in Section 7 uses `path-constraint='[parent::*:tr]'` to restrict expansion to table cells and `prop-constraint='width'` to further restrict expansion to @css:width attributes.

The parser is available as an XProc step [6].

4. Application 2: Device-specific EPUB Checking Rules

An application of the CSS expander is an augmented EPUB checker. Many issues with e-books are caused by limited CSS support of the devices. In book production workflows, it is desirable to detect these incompatibilities automatically, rather than visually checking each book on each relevant device. Schematron rules are a natural choice for CSS-expanded (Section 3) contents of EPUB files (or the other way round: in order to be able to use Schematron rules for CSS checks, the CSS needs to be more XPath-friendly).

These checks go beyond what, e.g., the W3C validator [7] can check. In a pure HTML/CSS environment, these *business rules* may be checked with custom Javascript programs. But in XML conversion pipelines or in the epubcheck [8][8] application, standardized declarative checking facilities (namely, Relax NG and Schematron) are available; in Javascript, not (yet) so much.

Given this input

```
<body>
  <section class="frontmatter TableOfContents" epub:type="frontmatter toc">
    <header>
      <h1>Brief Contents</h1>
    </header>
```

the CSS expansion yields:

```
<body css:font-size="1em" css:line-height="1.33em"
  css:font-family="'Stix', serif" css:font-variant-numeric="oldstyle-nums">
  <section class="frontmatter TableOfContents" epub:type="frontmatter toc">
    <header css:padding-top="3em" css:padding-right="0"
      css:padding-bottom="2em" css:padding-left="0">
      <h1 css:font-size="1.5em" css:line-height="1.33em"
        css:text-align="center" css:padding-bottom="0em"
        css:text-transform="uppercase" css:font-weight="normal"
        css:letter-spacing="4px">Brief Contents</h1>
    </header>
```

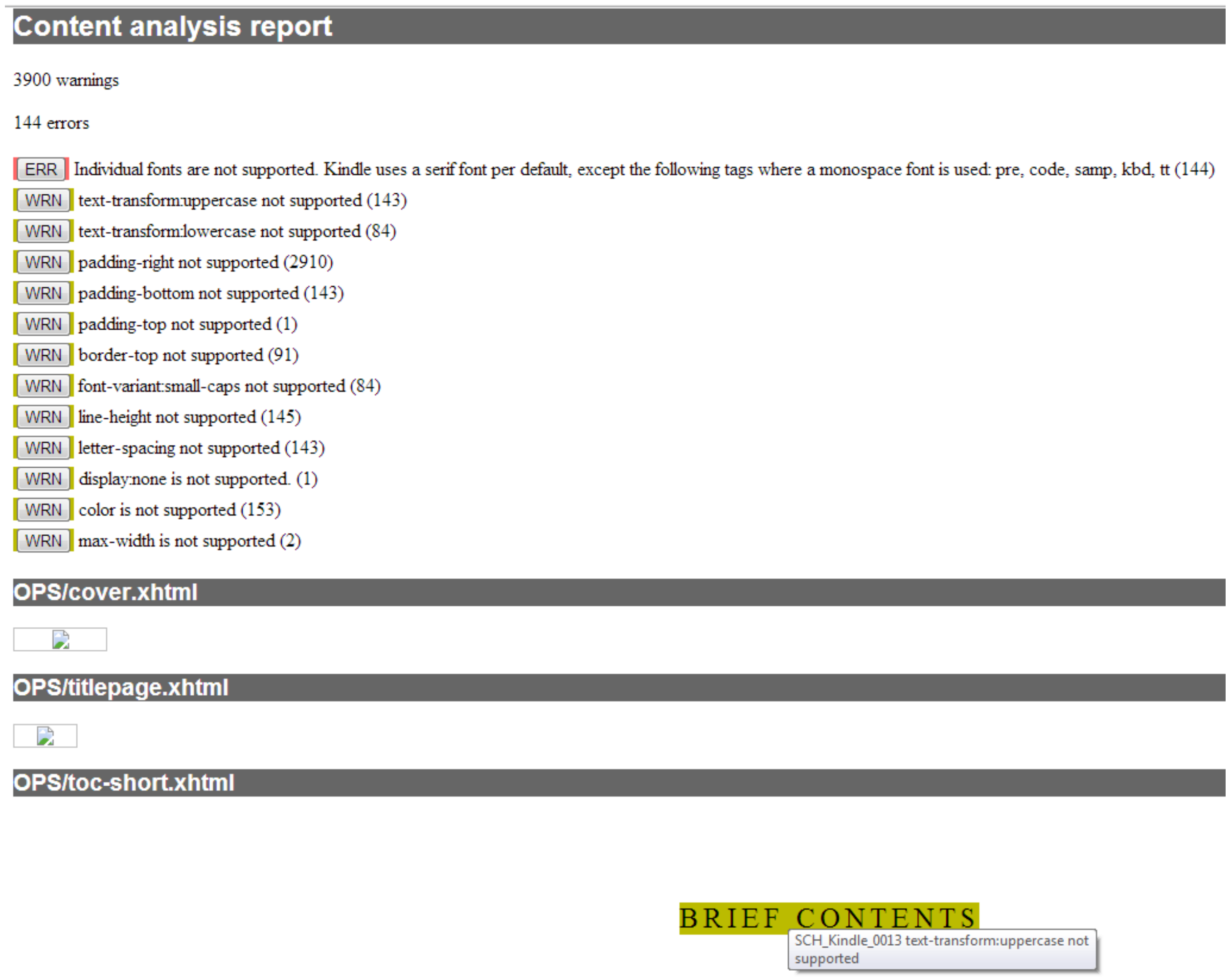
Applying the Schematron rule

```
<s:pattern id="css-text">
  <s:rule context="*[@css:text-transform]">
```

[8]However, in the current version of epubcheck, inclusion of custom Schematron rules is not supported (neither is the CSS→CSSa parser).

```
<s:report test="@css:text-transform eq 'uppercase'">
  <s:span class="severity">WRN</s:span>
 <s:span class="msgid">SCH_Kindle_0013</s:span> text-transform:uppercase
   not supported </s:report>
```

gives an HTML report (content documents with SVRL messages patched to the error/warning locations):



Content analysis report

3900 warnings

144 errors

ERR Individual fonts are not supported. Kindle uses a serif font per default, except the following tags where a monospace font is used: pre, code, samp, kbd, tt (144)
WRN text-transform:uppercase not supported (143)
WRN text-transform:lowercase not supported (84)
WRN padding-right not supported (2910)
WRN padding-bottom not supported (143)
WRN padding-top not supported (1)
WRN border-top not supported (91)
WRN font-variant:small-caps not supported (84)
WRN line-height not supported (145)
WRN letter-spacing not supported (143)
WRN display:none is not supported. (1)
WRN color is not supported (153)
WRN max-width is not supported (2)

OPS/cover.xhtml

OPS/titlepage.xhtml

OPS/toc-short.xhtml

BRIEF CONTENTS

SCH_Kindle_0013 text-transform:uppercase not supported

Figure 1. HTML report of epubcheck-xproc with a Mobipocket checking profile

5. Application 3: Hub XML

Hub XML [9] is an XML format derived from DocBook 5.1, with CSSa and publishing-specific elements such as tabs and line breaks added, and with the need for a chapter/section/… hierarchy removed.

A typical conversion workflow from manuscripts or typeset documents to publisher-specific XML looks like this:

Word (.docx), InDesign (.idml) or other complex XML document formats are converted to a flat Hub file.

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-model href="http://www.le-tex.de/resource/schema/hub/1.1/hub.rng"
  type="application/xml" schematypens="http://relaxng.org/ns/structure/1.0"?>
<?xml-model href="http://www.le-tex.de/resource/schema/hub/1.1/hub.rng"
  type="application/xml" schematypens="http://purl.oclc.org/dsdl/schematron"?>
<hub xmlns="http://docbook.org/ns/docbook"
     xmlns:css="http://www.w3.org/1996/css"
     version="5.1-variant le-tex_Hub-1.1"
     css:version="3.0-variant le-tex_Hub-1.1"
     css:rule-selection-attribute="role">
   <info>
      <keywordset role="hub"> ①
         <keyword role="source-basename">muster1</keyword>
         <keyword role="source-dir-uri">file://…/muster1.idml.tmp/</keyword>
         <keyword role="source-paths">false</keyword>
         <keyword role="formatting-deviations-only">true</keyword>
         <keyword role="source-type">idml</keyword>
      </keywordset>
      <css:rules>
         <css:rule name="p_Musterseite_p_column_left"
                   native-name="p:Musterseite:p_column_left"
                   layout-type="para"
                   css:font-size="9pt"
                   xml:lang="de"
                   css:text-align="center"
                   css:text-align-last="center">
            <tabs> ②
               <tab align="center"
                    alignment-char="."
                    leader=""
                    horizontal-position="170pt"/>
            </tabs>
         </css:rule>
         <css:rule name="p_boxes_p_box1_list"
                   native-name="p:boxes:p_box1_list"
                   layout-type="para"
                   css:font-size="8pt"
                   css:margin-top="0pt"
                   css:text-align="justify"
                   css:text-align-last="left"
                   css:font-weight="normal"
                   css:margin-left="8.5pt"
                   css:text-indent="-8.5pt"
```

```
                css:display="list-item"
                css:list-style-type="dash"
                css:font-family="Helvetica"
                css:pseudo-marker_content="'-'"/>③
      <css:rule name="p_tables_p_tab_leg"
                native-name="p:tables:p_tab_leg"
                layout-type="para"
                css:font-weight="normal"
                css:font-size="9pt"
                css:margin-left="42.5pt"
                css:text-indent="-42.5pt"
                css:margin-top="14.15pt"
                css:margin-bottom="5.65pt"
                css:text-align="justify"
                css:text-align-last="left"
                css:font-family="Times"/>
  …
      <css:rule name="Tabellenzelle"
                native-name="Tabellenzelle"
                layout-type="cell"
                css:padding-top="5.65pt"
                css:padding-left="5.65pt"
                css:padding-bottom="6.2pt"
                css:padding-right="5.65pt">
         <linked-style layout-type="para" name="p:tables:p_tab_text"/>④
      </css:rule>
    </css:rules>
  </info>
  <para role="p_tables_p_tab_leg" css:margin-top="11.3pt"> ⑤
    <anchor xml:id="id_HyperlinkTextDestination_Tab__3"/>
    <phrase role="ch_Tabellen_z_tab_leg_bold">Tabelle 3:</phrase>
    ⑥<tab> </tab>Klassifikation …</para>
  <para role="NormalParagraphStyle">
    <informaltable>
      <tgroup cols="2">
        <colspec colname="c1"/>
        <colspec colname="c2"/>
        <tbody>
          <row>
            <entry colname="c1" role="Tabellenkopf">
              <para role="p_tables_p_tab_head">DSM-IV-TR</para>
```

① Optional Hub-specific keyword section (whether there are formatting deviations only in the content; whether the section hierarchy and lists have been nested; input file type; …)

② Tabulator declaration (may also be included with individual paragraphs)
③ Representing pseudo selectors `li::marker { content: '\2012'; }` as CSSa. String has to be in quotes because there may also be keywords in lieu of literals.
④ *Linked style* doesn't mean that one style inherits properties from another. It rather specifies the default paragraph style for table cells with this cell style.
⑤ Local override (margin-top). (Please note that, other than in the real document, the XML source code within paragraphs is indented in this listing.)
⑥ A tabulator in the content (same element name as `tabs/tab` in the tab declaration, but different content model – this one contains just a &#9; character).

This file, within the confines of the Hub schema, is converted towards hierarchized DocBook, with proper lists, images/tables associated with their captions, etc. In the course of this operation, typically the <tab>s disappear (converted to list items, section numbering, etc.).

The Almost-DocBook-with-some-local-CSSa-overrides file will be converted to the publisher's XML format. Some publishers choose formats that include CSSa. We have implemented this with DocBook (trivially), TEI, and BITS (the upcoming NLM/JATS format for books [10]). BITS/JATS is an interesting case because some of the CSSa has to be mapped to actual elements: <bold>, <italic>, <underline>, …

So far, Hub has proven as the right choice for an intermediate format: since all upconversion heuristics is applied at a later stage, the conversion seldom fails; it may be validated; it serves as common ground for Word and InDesign (and other applications, in principle), so we need only one upconverter from flat Hub (of course with configuration options as to the style names, and with the possibility to override conversion steps for a given production line).

6. Example: Mapping InDesign Properties to CSSa

Not all of InDesign's more than 350 layout properties map to CSS in a linear fashion. Some don't map at all. For converting OOXML (.docx) and IDML properties to CSSa, we've established a property map mechanism with standard handling for lengths, colors, alignment options, etc. Although the propmap syntax is the same, the property types and their implementations (in XSLT 2) differ between Word and InDesign.

The mapping mechanism works by processing the properties (attributes and Properties/* elements) that occur in the style declarations and and content of IDML documents [11]. For each property, the corresponding prop element is looked up in the propmap, and then the prop element is processed with the context node (the property attribute/element) as parameter. The processing doesn't immediately create the CSSa attributes but intermediate attribute declarations. The styles will be processed recursively: properties of ancestor styles will be created before the properties of derived styles. Derived styles may not only alter but also switch off properties that have been set by ancestor styles. In a subsequent XSLT pass, these intermediate attribute declarations will be consolidated to form the final CSSa attribute.

The `Position` attribute is special in the effect that it isn't transformed to a CSS attribute. Both DocBook, on which Hub is based, and HTML don't describe position as properties but as elements (superscript etc.). This property→markup conversion is being taken care of by processing special intermediate declarations for that purpose.

Sometimes a single document format property maps to multiple CSS properties, and sometimes multiple document format properties constitute a single CSS property, as seen in the following excerpt from our IDML property map.

```
<propmap>
  <prop name="Name" />
  <prop name="NextStyle" />
  <prop name="Self" />
  ……
  <prop name="aid:cstyle" type="passthru"/>①
  <prop name="aid:pstyle" type="passthru"/>
  <prop name="aid5:cellstyle" type="passthru"/>
  <prop name="aid5:tablestyle" type="passthru"/>
   ……
  <prop name="AppliedFont" type="linear" target-name="css:font-family"/>②
  <prop name="AppliedLanguage" type="lang" target-name="xml:lang" />
  <prop name="BottomInset" type="length" target-name="css:padding-bottom" />③
  <prop name="BulletChar" target-name="css:pseudo-marker_content"
    type="bullet-char"/>④
  <prop name="BulletsAndNumberingListType" target-name="list-type"
    type="linear" hubversion="1.0"/>⑤
  <prop name="BulletsAndNumberingListType" target-name="css:list-style-type"
    type="list-type-declaration" hubversion="1.1"/>⑥
  <prop name="BulletsFont" target-name="css:pseudo-marker_font-family"
    type="linear"/>
  <prop name="BulletsFontStyle">
    <val match="(^|\W)Bold" target-name="css:pseudo-marker_font-weight"
      target-value="bold" />
    <val match="SemiBold" target-name="css:pseudo-marker_font-weight"
      target-value="600" />
    <val match="Italic" target-name="css:pseudo-marker_font-style"
      target-value="italic" />
    <val match="Oblique" target-name="css:pseudo-marker_font-style"
      target-value="oblique" />
    <val match="Medium" target-name="css:pseudo-marker_font-weight"
      target-value="normal" />
    <val match="Regular" target-name="css:pseudo-marker_font-weight"
      target-value="normal" />
    <val match="Roman" target-name="css:pseudo-marker_font-weight"
      target-value="normal" />
  </prop>
  <prop name="Capitalization">
    <val eq="SmallCaps" target-name="css:font-variant"
      target-value="small-caps"/>
    <val eq="AllCaps" target-name="css:text-transform"
      target-value="uppercase"/>
    <val eq="CapToSmallCap" target-name="css:text-transform"
      target-value="uppercase"/><!-- ? -->
  </prop>
  <prop name="aid:ccolwidth" type="length" target-name="css:width"/>
  <prop name="CharacterDirection" target-name="css:direction">
    <val eq="DefaultDirection" target-value="ltr"/>
    <val eq="LeftToRightDirection" target-value="ltr"/>
    <val eq="RightToLeftDirection" target-value="rtl"/>
  </prop>
  <prop name="FillColor" type="color" target-name="css:background-color">
    <context match="Para|Char" target-name="css:color"/>⑦
  </prop>
  <prop name="FillTint" type="percentage" target-name="fill-tint"/>⑧
  <prop name="FirstLineIndent" type="length" target-name="css:text-indent" />⑨
  ……
  <prop name="Hidden" target-name="css:display">
    <val eq="true" target-value="none"/>
    <val eq="false"/>
  </prop>
  <prop name="Justification">
    <val match="LeftAlign" target-name="css:text-align"
      target-value="left" />
    ……
    <val match="CenterAlign" target-name="css:text-align"
      target-value="center" />
    <val match="CenterAlign" target-name="css:text-align-last"
      target-value="center" />⑩
    <val match="Justified" target-name="css:text-align"
      target-value="justify" />
    <val match="LeftJustified" target-name="css:text-align-last"
      target-value="left" />
    ……
  </prop>
  <prop name="LeftBorderStrokeColor" type="color"
    target-name="css:border-left-color"/>⑪
  <prop name="LeftBorderStrokeWeight" type="length"
    target-name="css:border-left-width"/>
  <prop name="LeftIndent" type="length" target-name="css:margin-left" />
  <prop name="LeftInset" type="length" target-name="css:padding-left" />
  <prop name="ListItem/Position" type="length"
    target-name="horizontal-position" /><!-- for tablists -->
  <prop name="ListItem/Alignment">
    <val match="LeftAlign" target-name="align" target-value="left" />
    <val match="CenterAlign" target-name="align" target-value="center" />
    <val match="RightAlign" target-name="align" target-value="right" />
  </prop>
  <prop name="ListItem/AlignmentCharacter" type="linear"
    target-name="alignment-char" />
  <prop name="ListItem/Leader" type="linear" target-name="leader" />
  ……
  <prop name="PointSize" type="length" target-name="css:font-size" />
  <prop name="Position" type="position" />⑫
  <prop name="RightIndent" type="length" target-name="css:margin-right" />
  <prop name="RightInset" type="length" target-name="css:padding-right" />
  <prop name="ShadowColor" type="color" target-name="shadow-color" />
  <prop name="SpaceAfter" type="length" target-name="css:margin-bottom" />
  <prop name="SpaceBefore" type="length" target-name="css:margin-top" />
  <prop name="StrikeThru" target-name="css:text-decoration-line">
    <val eq="true" target-value="line-through"/>
    <val eq="false" target-value="none"/>
  </prop>
  <prop name="TabList" type="tablist"/>⑬
  <prop name="TintValue" type="percentage" target-name="fill-value"/>
  <prop name="TopInset" type="length" target-name="css:padding-top" />
  <prop name="Underline" target-name="css:text-decoration-line">
    <val eq="true" target-value="underline"/>
    <val eq="false" target-value="none"/>
  </prop>
  <prop name="UnderlineType"
    implement="use text-decoration-style must look at Stroke/..." />⑭
  <prop name="VerticalJustification">
    <val match="TopAlign" />
    <val match="CenterAlign" target-name="css:vertical-align"
      target-value="middle" />
    <val match="BottomAlign" target-name="css:vertical-align"
      target-value="bottom" />
  </prop>
  ……
  <prop name="NumberingApplyRestartPolicy" implement="maybe later" />
  <prop name="BulletsAlignment" implement="maybe later" />
  <prop name="NumberingAlignment" implement="maybe later" />
  ……
</propmap>
```

① @aid:pstyle, @aid:cstyle will be mapped to @role attributes later on in the process. type="passthru" signifies that the attribute will be reproduced verbatim.

② `linear` mapping means that the attribute will be renamed to what is given in @target-name.

③ `length` mapping converts the dimensionless input (which is always in pt in InDesign) to a 'XY.Zpt' value, and renames the attribute to @target-name.

④ `bullet-char` mapping is not acting on an attribute, but on an element Properties/BulletChar (elements below Properties are used by InDesign when there is no simple name/value relation). Converts the decimal value of its @BulletCharacterValue to a Unicode character, iff @BulletCharacterType equals `UnicodeOnly`. Raises a warning for other types.

⑤ `list-type` mapping selects one of the CSS3 predefined list style types [12] (currently only implemented for the most common types).

⑥ Since there are different iterations (versions) of the Hub format that different subsequent converters rely upon, the XSLT stylesheet that processes the property map needs to be able to generate different versions. Here it can be seen that from version 1.0 to 1.1 the list style became a proper css property.

⑦ The `FillColor` property is processed differently, depending on context. In paragraph and inline contexts, it specifies the font color, while in tables and objects, it specifies the background color.

⑧ The `FillTint` property does not map to a CSSa property. Instead, it creates an intermediate property whose percent value will be multiplied with the CSSa property value that is derived from `FillColor` above.

⑨ The combined presence of, e.g., css:margin-left="4mm" and the negative value css:text-indent="-4mm" is used in a Hub list recognition mode that does not rely on the presence of proper lists (that InDesign typesetters don't always use).

⑩ `Justification` is an example for a single IDML property that creates two distinct CSSa properties (at least with CSS 3 [13]).

⑪ Table cell border properties (color, width, style) will all be mapped to a distinct CSSa property for each of the 4 borders, so there are up to 12 CSSa attributes in lieu of "`border: 1px solid red`". We think that this is acceptable because one can use, for example, `every $att in @*[matches(name(), 'css:border-.+-width')] satisfies …` in Schematron rules.

⑫ The `Position` property will not create CSSa attributes but superscript / subscript elements, as discussed above.

⑬ The `TabList` property will not create CSSa attributes but tabs/tab elements – tab stops. They are permitted in Hub but need to be eliminated when upconverting to common XML vocabularies. Usually they are dissolved in lists or tables.

⑭ The @implement attribute contains implementation notes, for example, "maybe later."

7. Application 4: Converting from .docx to .idml via Hub, TEI and HTML

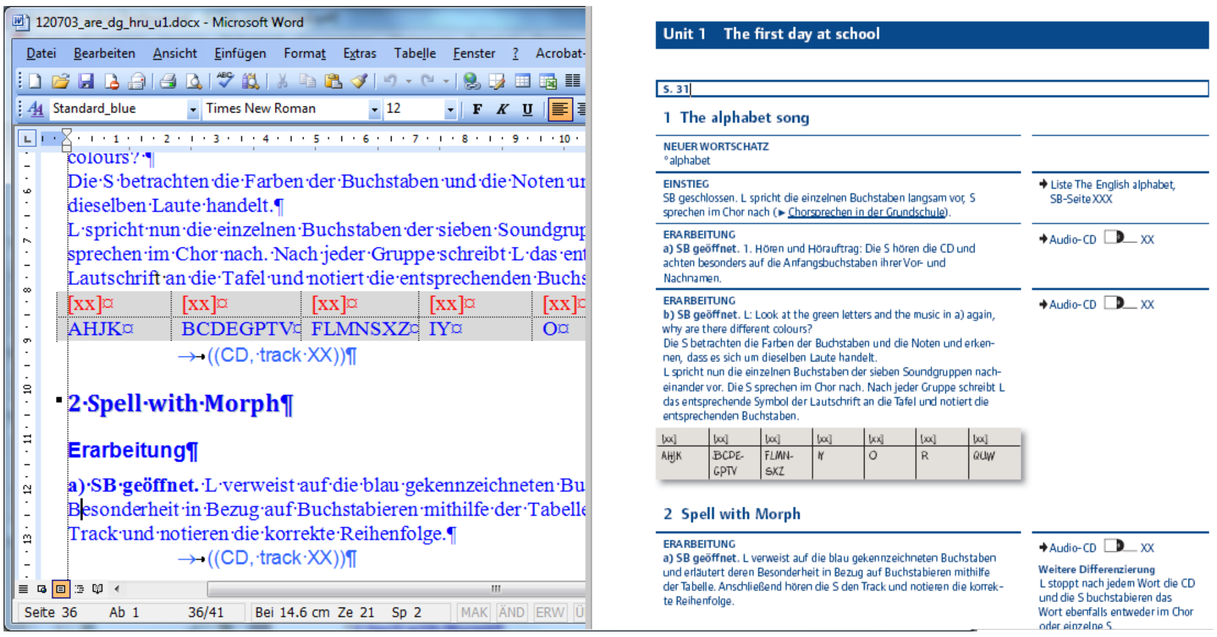
We set up a self-service preview workflow for a major German textbook publisher, with the aim of reducing turnaround times from authors to typesetters to authors, and with the ultimate aim of saving costs, of course.

In this workflow, the content is edited in .docx files and is ultimately converted to IDML (Figure 2) so that a typesetter can refine the page makeup later, or instantly create a coarsely paginated PDF, however with the final layout parameters. This is particularly useful for estimating the page count. Alternatively, in the absence of a typesetter, a desktop InDesign application or an InDesign server, the author can have an HTML preview whose print rendering comes sufficiently close to the InDesign rendering, at least for the purpose of page count estimates. Setting up the HTML preview does not come at extra cost. The workflow (XProc pipeline with WebDAV or HTML form upload as frontends) is structured as follows:

1. Convert from .docx to Hub XML.
2. Convert from Hub XML to a custom TEI format (this also serves as the media neutral archive format for these publications),
   - enriched with Schematron for the specific section hierarchy,
   - semantic markup for educational content,
   - and CSSa for ad-hoc formatting (e.g., red color in the "[xx]" cells, as can be seen in electronic renderings of this paper).
3. Convert from custom TEI to HTML:
   - implementing the two-column layout as tables, widths specified by ordinary CSS rules such as `table.main td { width: 112mm };`
   - proportionally scaling proper tables (the actual tables already present in the Word manuscript) to the layout column widths;
   - rendering ad-hoc formatting (CSSa attributes) as HTML style attributes.
   - InDesign needs pt values for the widths of all cells. The cell widths are calculated from either css:width attributes of the Hub XML (for proper tables) and the selectively CSS-expanded (see Section 3) HTML for the layout (outer) tables.
   - Also enrich the HTML with @data-rownum and @data-colnum attributes for proper tables (see Figure 3). This is difficult for tables with joined cells. We have adopted Andrew Welch's table normalization algorithm for this task [14]. IDML needs to know these absolute cell position numbers.

4. Convert the data- and width-enriched HTML to IDML.

   Although in principle, all InDesign styles could be synthesized from the HTML's CSS, we are pursuing a hybrid approach: Patch the synthesized IDML content into an IDML file *with predefined styles*. Only use certain layout properties from HTML, such as the calculated table cell widths and some local CSSa overrides for styling. As one can see in the right half of Figure 2 (at least if reading a color version of this paper), @css:color is not one of the attributes that is passed to IDML.[9]


120703_are_dg_hru_u1.docx - Microsoft Word
Datei Bearbeiten Ansicht Einfügen Format Extras Tabelle Fenster ? Acrobat-
Standard_blue
Times New Roman
12
F K U
Seite 36 Ab 1 36/41 Bei 14.6 cm Ze 21 Sp 2 MAK ÄND ERW Ü

colours?·¶
Die·S·betrachten·die·Farben·der·Buchstaben·und·die·Noten·u
dieselben·Laute·handelt.¶
L·spricht·nun·die·einzelnen·Buchstaben·der·sieben·Soundgru
sprechen·im·Chor·nach.·Nach·jeder·Gruppe·schreibt·L·das·en
Lautschrift·an·die·Tafel·und·notiert·die·entsprechenden·Buch

[xx]¤	[xx]¤	[xx]¤	[xx]¤	[xx]¤
AHJK¤	BCDEGPTV¤	FLMNSXZ¤	IY¤	O¤

→((CD,·track·XX))¶

2·Spell·with·Morph¶

Erarbeitung¶

a)·SB·geöffnet.·L·verweist·auf·die·blau·gekennzeichneten·Bu
Besonderheit·in·Bezug·auf·Buchstabieren·mithilfe·der·Tabell
Track·und·notieren·die·korrekte·Reihenfolge.¶
→((CD,·track·XX))¶

Unit 1 The first day at school

S. 31

1 The alphabet song

NEUER WORTSCHATZ
°alphabet

EINSTIEG
SB geschlossen. L spricht die einzelnen Buchstaben langsam vor, S sprechen im Chor nach (► Chorsprechen in der Grundschule).

→ Liste The English alphabet, SB-Seite XXX

ERARBEITUNG
a) SB geöffnet. 1. Hören und Hörauftrag: Die S hören die CD und achten besonders auf die Anfangsbuchstaben ihrer Vor- und Nachnamen.

→ Audio-CD XX

ERARBEITUNG
b) SB geöffnet. L: Look at the green letters and the music in a) again, why are there different colours?
Die S betrachten die Farben der Buchstaben und die Noten und erkennen, dass es sich um dieselben Laute handelt.
L spricht nun die einzelnen Buchstaben der sieben Soundgruppen nacheinander vor. Die S sprechen im Chor nach. Nach jeder Gruppe schreibt L das entsprechende Symbol der Lautschrift an die Tafel und notiert die entsprechenden Buchstaben.

→ Audio-CD XX

[xx]	[xx]	[xx]	[xx]	[xx]	[xx]	[xx]
AHJK	BCDE-GPTV	FLMN-SXZ	IY	O	R	QUW

2 Spell with Morph

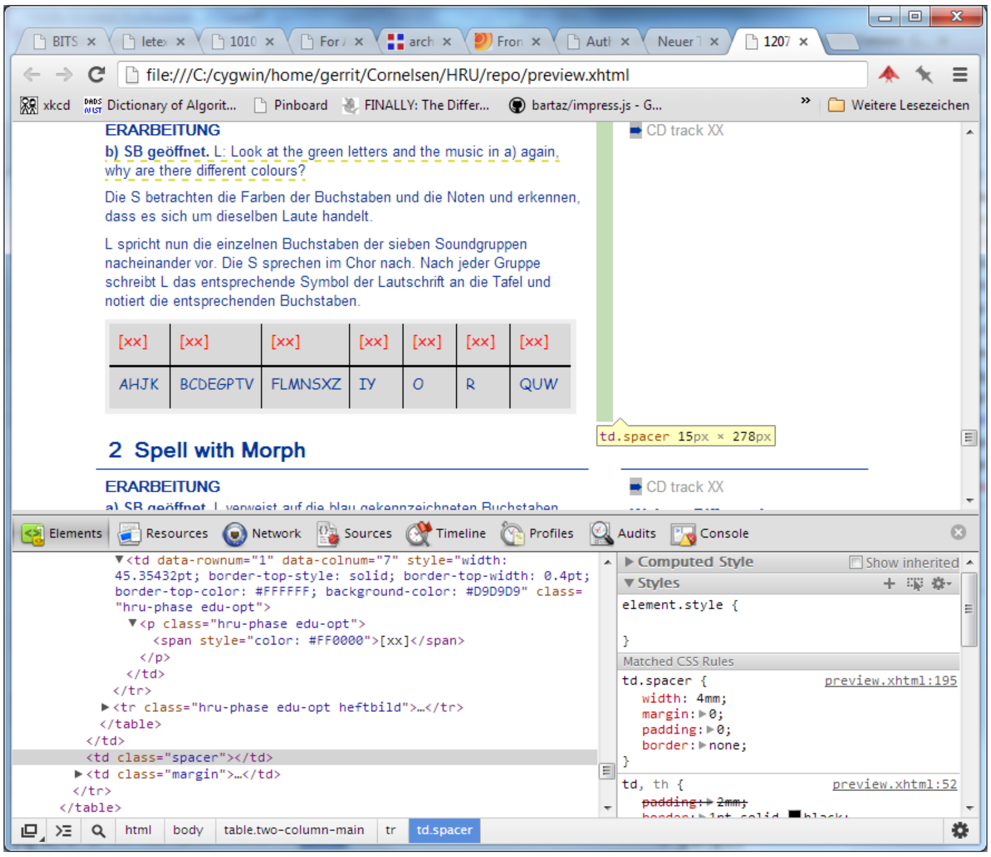
ERARBEITUNG
a) SB geöffnet. L verweist auf die blau gekennzeichneten Buchstaben und erläutert deren Besonderheit in Bezug auf Buchstabieren mithilfe der Tabelle. Anschließend hören die S den Track und notieren die korrekte Reihenfolge.

→ Audio-CD XX

Weitere Differenzierung
L stoppt nach jedem Wort die CD und die S buchstabieren das Wort ebenfalls entweder im Chor oder einzelne S.

Figure 2. From .docx to .idml without using Word or InDesign

[9]It will be an interesting endeavour to create an HTML+CSS→IDML converter that is only configured with CSS, rather than with an IDML template and some CSS. Currently, CSS and its rendering engines (a.k.a. browsers) don't offer all the advanced layout options available in InDesign. But given the rapid advances in the browsers' layout (and MathML, and SVG) capabilities and in the expressiveness of CSS itself, it is only a matter of time until dedicated typesetting systems become obsolete, even for print products.


file:///C:/cygwin/home/gerrit/Cornelsen/HRU/repo/preview.xhtml

ERARBEITUNG

b) SB geöffnet. L: Look at the green letters and the music in a) again, why are there different colours?

Die S betrachten die Farben der Buchstaben und die Noten und erkennen, dass es sich um dieselben Laute handelt.

L spricht nun die einzelnen Buchstaben der sieben Soundgruppen nacheinander vor. Die S sprechen im Chor nach. Nach jeder Gruppe schreibt L das entsprechende Symbol der Lautschrift an die Tafel und notiert die entsprechenden Buchstaben.

[xx]	[xx]	[xx]	[xx]	[xx]	[xx]	[xx]
AHJK	BCDEGPTV	FLMNSXZ	IY	O	R	QUW

CD track XX

td.spacer 15px × 278px

2 Spell with Morph

ERARBEITUNG

CD track XX

```
▼<td data-rownum="1" data-colnum="7" style="width:
45.35432pt; border-top-style: solid; border-top-width: 0.4pt;
border-top-color: #FFFFFF; background-color: #D9D9D9" class=
"hru-phase edu-opt">
  ▼<p class="hru-phase edu-opt">
     <span style="color: #FF0000">[xx]</span>
   </p>
  </td>
 </tr>
▶<tr class="hru-phase edu-opt heftbild">…</tr>
</table>
</td>
<td class="spacer"></td>
▶<td class="margin">…</td>
</tr>
</table>
```

```
element.style {

}
Matched CSS Rules
td.spacer {                     preview.xhtml:195
   width: 4mm;
   margin:▶0;
   padding:▶0;
   border:▶none;
}
td, th {                        preview.xhtml:52
   ~~padding:▶ 2mm;~~
```

Figure 3. HTML preview

It is obvious that both input- and output-wise, the pipeline can easily be enhanced. For example with Schematron checks that allow only certain colors for local overrides in the Hub XML, or with an EPUB creation step that operates on the preview HTML. Word file editing, upload and unzipping may be replaced with browser-based HTML/XML editors and XML database storage, accessed via RESTXQ. Long live the XML stack.

8. Finally

8.1. Future Work

A couple of things remain to be done:

- Deal with relative dimensions: express width and lengths as an absolute unit if they are measured in em or ex and if they may be tracked back to an absolute unit.
- Library (XSLT function, XProc step) to convert all lengths to integer values of a certain unit, e.g., twips (1/20 of a pt) and to store them in attributes with the same local names, but with a different namespace prefix. This will make length

comparisons a lot easier, because length attribute values don't need to be parsed and normalized any more.

- Further process font-family if it contains a comma-separated list.
- Generate styles from local (per-element) CSS properties (need an algorithm that minimizes both the number of styles and local overrides). Similar problem: gather common properties from partly redundant named rules and derive composable named rules from them.
- Modularize the CSSa Relax NG schema and create distinct front-end schemas for different CSS versions (CSS 2.1, EPUB 2.0.1, EPUB 3, CSS 3, …).
- Get feedback and act upon it.

8.2. Have You Considerd XYZ?

Is CSSa the only viable approach? Probably not. There are established contenders within the XML family and movements in the opposite direction (towards CSS-like syntax, e.g., selectors in jQuery) in Web tech.

8.2.1. Where X = XSL, Y = dash and Z = FO

Mapping DTP and word processor formats to XSL-FO (instead of host vocabulary + CSSa) may have the advantage that more constructs of the original layout information may be expressed, while most of the properties share their name and semantics with their CSS counterparts.

On the other hand, XSL-FO clearly fails to add itself as an orthogonal layer onto other vocabularies. It is not suited for the common scenario that a given host vocabulary needs to be annotated with rendering hints.

8.2.2. CAS

One day before the final version of this paper was due, I learned of "Cascading Attribute Sheets" [15]. It's a proposal to express HTML attributes in CSS syntax. Example: `#content video { preload: auto; }`. Obviously, for CSS selectors to work without circularity, attaching classes and IDs to elements is not a use case for CAS.

CAS is the opposite of what this paper is proposing. The Web people seem to love the concept. I think this exemplifies the different cultures of Web and XML people. CSS vs. XPath selectors, DOM vs. XDM, curly braces vs. angle brackets. But there's at least some hope for reconciliation: both approaches are declarative. And they aren't really antagonistic because they serve different purposes: CAS is syntactic sugar that may add maintanability, while CSSa makes CSS palatable to the (still-alive) XML toolchain.

9. Conclusion

CSSa is a simple yet powerful way to describe layout information in XML workflows. It facilitates advanced content checking, XSLT processing, and HTML preview / e-book generation. Its vocabulary, together with a host vocabulary such as DocBook, is (almost – think of tabs and line breaks) sufficient for capturing relevant layout information of common DTP / word processing tools, providing a common ground for all kinds of XML processing.

By virtue of its origin in Web technology, CSSa lends itself particularly well to converting these applications' documents to and from styled HTML or to e-books. By virtue of its node representation (one property/value pair per attribute instead of chunks of unparsed text) and its simplicity (no inheritance among named rules, straightforward calculation of the effective style attributes on a given element), it lends itself well to XML processing tools such as Schematron validation or XSLT grouping.

Bibliography

[1] Style declarations in TEI P5: http://www.tei-c.org/release/doc/tei-p5-doc/en/html/ref-styleDefDecl.html.

[2] A snapshot as of this writing can be found on http://www.le-tex.de/resource/schema/hub/1.1/css/css.rng.

[3] Rules for CSS identifiers: http://www.w3.org/TR/CSS21/syndata.html#characters.

[4] Jelliffe, Rick: Do you need to make your own XSLT2 function definitions when using Schematron? http://broadcast.oreilly.com/2010/09/do-you-need-to-make-your-own-x.html.

[5] CSS document order: http://www.w3.org/TR/css3-cascade/#cascade-order.

[6] CSS expansion as an XProc step: https://subversion.le-tex.de/common/css-expand/xpl/css.xpl .

[7] W3 CSS validator: http://jigsaw.w3.org/css-validator/.

[8] epubcheck: https://code.google.com/p/epubcheck/.

[9] Hub XML schema: http://www.le-tex.de/resource/schema/hub/1.1/hub.rng.

[10] Book Interchange Tag Suite: jats.nlm.nih.gov/extensions/bits/tag-library/.

[11] IDML File Format Specification: https://www.adobe.com/content/dam/Adobe/en/devnet/indesign/cs55-docs/IDML/idml-specification.pdf.

[12] CSS Lists and Counters Module Level 3 (predefined counter styles): http://www.w3.org/TR/css3-lists/#ua-stylesheet.

[13] CSS Text Module Level 3 (text-align-last property): http://www.w3.org/TR/css3-text/#text-align-last0.

[14] Andrew Welch: Table Normalization in XSLT 2.0. http://andrewjwelch.com/code/xslt/table/table-normalization.html.

[15] Tab Atkins, Jr.: Proposal for "Cascading Attribute Sheets". http://lists.w3.org/Archives/Public/public-webapps/2012JulSep/0508.html.

XProc at the heart of an ebook production framework

The approach of the DAISY Pipeline project

Romain Deltour
DAISY Consortium
<rdeltour@gmail.com>

Abstract

Using XProc, the XML Pipeline Language, is the natural choice when implementing XML-centric single source ebook production workflows. With proper extensions and by encapsulating it in a component model, XProc can lie at the heart of a truly modular and extensible conversion framework. This paper describes the approach and lessons learned in the DAISY Pipeline project, an open source conversion platform for the production of accessible digital content.

Keywords: XML, XProc, XSLT, EPUB, ebook

1. Introduction

The rapid growth of the ebook industry is a real challenge for digital content producers who have to keep up with an ever increasing set of formats and reading devices. It is also a great opportunity to give equal access to information and knowledge to everyone, regardless of disability, whether we read with our eyes, ears or fingers.

Promoting an inclusive publishing ecosystem around the world has been a key mission of the DAISY Consortium for many years. To help creators of accessible publications be more effective, the DAISY Consortium notably coordinates the development of the DAISY Pipeline project: an XML-centric open source cross-platform framework for the automated processing of various digital formats.

Using XProc, the XML Pipeline Language, is a natural choice when choosing to implement document conversion workflows - especially given the overwhelming majority of XML-based document formats. XProc offers many benefits out of the box: it is a platform-neutral open standard, it is capable of describing complex processing workflows, and it is natively scalable and extensible. On the other hand, using XProc in the context of ebook production sometimes requires a particular mindset; conversions not only have to manipulate single XML documents but also entire file sets, sometimes including non-XML content. XProc's core concepts of input ports, output ports, options and parameters can be difficult to expose to end users who are not XProc or XML experts. However, XProc can be extended. It is possible to base a real component architecture on bundled XProc and XSLT scripts, enabling the creation of truly reusable software components. It is possible to annotate XProc documents with machine-readable user-oriented metadata to allow applications to expose the functionality of XProc scripts in user-friendly interfaces.

This article presents the approach taken in the DAISY Pipeline project to use XProc at the heart of a modular and extensible ebook production framework. After giving more information on the context of the project and its primary objectives in Section 2, we present in Section 3 how we achieve a truly modular architecture by integrating XProc and XSLT documents in a component model. In Section 4 we describe what we identified as best practices or useful patterns when using XProc to process publication file sets, before describing in Section 5 a list of features missing in today's XProc that would make our life easier if they were supported in future versions.

2. Background

The DAISY Consortium is a non-profit global consortium of like-minded organizations committed to working towards creating the best way to read and publish. Our vision is that people should have equal access to information and knowledge regardless of disability; a right confirmed by the UN Convention on the Rights of Persons with Disabilities. To support this vision, one of the DAISY Consortium's key missions is to help producers of accessible content to be more effective. The DAISY membership is composed primarily of non-profit organizations specializing in re-publishing material in a variety of formats readable with eyes, ears (audio books) or fingers (Braille). The DAISY Pipeline project [1] has been initiated as a collaborative effort to provide a generic conversion framework that can be easily adapted and extended to support the various production workflows in this heterogeneous community.

The overarching objectives of the projects are to:

- provide functionality to produce, maintain, and validate accessible digital formats
- embrace good practices for the creation of quality accessible content
- support the single source publishing approach - i.e. converting a master XML document into a variety of output formats - where applicable
- minimize overlap and duplication, notably via the development of reusable components

Earlier versions of the system relied on a custom workflow description syntax, but with the release of XProc [2] as a W3C Recommendation in 2010 coupled with the powerful features of XPath 2.0 [3] and XSLT 2.0 [4], we decided to fundamentally redesign the framework and embrace XProc as one of the keystones of the new architecture.

3. A modular architecture

A component-oriented design is the obvious approach to achieve an adaptable and extensible framework. Cohesive modules with the right level of encapsulation are usually highly reusable, which consequently enables flexibility. In the DAISY Pipeline, the modules can be of different nature; some modules are made only of Java classes, or only of XML documents, or a mix of Java and XML, and even possibly other types of code or content.

As the core of the middleware is developed in Java (notably relying on Norman Walsh's Calabash XProc processor [6] and Saxonica's Saxon XSLT/XQuery processor [7]), we opted to use OSGi [5] as the basis of our runtime component system. OSGi is the de-facto standard for a dynamic component model for Java, coupled with a flexible service-oriented approach. An OSGi module (called *bundle* in the local lingo) is basically a ZIP archive (a jar) with metadata stored in a *manifest* document (stored in the `META-INF/MANIFEST.MF` entry of the archive). OSGi notably specifies the way to declare the module's unique identifier, its version, as well as its runtime dependencies.

For the XML components themselves, technologies like XProc (and also XSLT, XQuery) already have good native modularity properties: XProc notably allows the declaration of re-usable steps via the `p:declare-step` element which can be imported with `p:import` statements and grouped in libraries with the `p:library` element. However, basic principles of a true component model — e.g. encapsulation — involve slightly more advanced mechanisms: some components should be usable by other modules without requiring relative linking (i.e. import statements with relative URIs), while other internal components should be hidden to the outside.

At the early stage of the development, we had a look at the promising EXPath Packaging System [8], which deals with some of the same concerns. However, at that time the proposed specification was still very young and not widely adopted; parts of it were not deemed mature enough (e.g. the versioning system; or the very nature of the spec — did it describe a mere packaging format or an entire provisioning or runtime system?). Additionally, parts of the EXPath Packaging System overlapped with an OSGi-based approach (e.g. the module's identifier and version, the dependencies declaration) and some parts were not critical to our use case (e.g. the distribution of components amongst "spaces"). These considerations led us to adopt a simpler alternative approach based on OASIS XML Catalogs [9].

An XML-based module in the DAISY Pipeline framework is essentially a bundled collection of XML documents (see Example 1). An OASIS XML Catalog is used to associate public URIs to the components that are accessible from outer modules (see Example 2), whereas other components will be de-facto invisible to outer modules. A shared URI resolver keeps track of the respective catalogs of all the modules in the system to let modules call each others via public URIs. A module's catalog is converted to OSGi metadata at build time to make the module a good citizen in an OSGi system; this notably allows to implement the dynamic hooks required to implement hot-deployment of modules in the system. The use of XML catalogs coupled with OSGi metadata also enables the definition of complex modules that for instance contain both the Java-based implementation of a processor-specific XProc step associated with its XProc signature declaration (e.g. held in a `p:library` element), all runtime dependencies being precisely defined as OSGi metadata.

Example 1. Structure of a DAISY Pipeline module

```
+ META-INF/
    - catalog.xml
+ public/
    - html-to-epub3.xpl
+ internal/
    - html-to-nav-doc.xsl
    - html-to-epub-xhtml.xsl
```

Example 2. The OASIS catalog of a DAISY Pipeline module

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
   <uri
     name="http://www.daisy.org/pipeline/modules/daisy3-to-epub3/▶
daisy3-to-epub3.xpl"
     uri="../xml/xproc/daisy3-to-epub3.xpl"/>
</catalog>
```

4. Extending XProc

The XProc recommendations specifies various ways in which XProc can be extended to caters for implementation-specific needs — this ranges from namespaced extension attributes to `p:pipeinfo` or `p:documentation` elements. This section presents the approaches we adopted to answer our requirements.

First, one of the characteristics of the DAISY Pipeline framework is that we need to differentiate on the one hand top-level, user-oriented XProc scripts and on the other hand the utility XProc scripts used only as building blocks from other modules. Some scripts hold the logic for the main conversion steps and are rightfully exposed to the users of the system ; other scripts are merely used as internal utility steps and should not be exposed as runnable components. To differentiate the two set of scripts, we had to augment XProc with custom metadata stored in extension attributes in a custom namespace.

A top-level XProc script intended to be executable by end-users (we call them "Pipeline Scripts") must be identified as such in its corresponding catalog entry (see Example 3) and should additionally contain descriptive information that is both human-readable and machine-readable. We use HTML documentation blocks, augmented with `px:role` attributes to convey machine-readable semantics (see Example 4)

Example 3. Identifying a module component as a top-level script

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog"
         xmlns:px="http://www.daisy.org/ns/pipeline">
   <uri
     name="http://www.daisy.org/pipeline/modules/daisy3-to-epub3/▶
daisy3-to-epub3.xpl"
     uri="../xml/xproc/daisy3-to-epub3.xpl"
     px:script="true"/>
</catalog>
```

Example 4. A Pipeline script documentation block

```
<p:documentation xmlns="http://www.w3.org/1999/xhtml">
    <h1 px:role="name">DAISY 3 to EPUB 3</h1>
    <p px:role="desc">Transforms a DAISY 3 publication
       into an EPUB 3 publication.</p>
    <p px:role="author maintainer">Romain Deltour</p>
    <p>See the
       <a href="http://code.google.com/p/daisy-pipeline/wiki/DAISY3ToEPUB3"
          px:role="homepage">online documentation</a>
    </p>
</p:documentation>
```

Additionally, the XProc step signature is augmented with typing information to further refine the format of documents expected on input ports and produced on output ports as well as the type of the XProc options, as showed in Example 5. This typing information notably allows user interfaces to validate the user input and to present accurate input widgets, for instance in an HTML form used to create a new execution job.

Example 5. Typed XProc signature

```
<p:input port="source" px:media-type="text/xml"
         primary="true" sequence="false">
    ...
</p:input>

<p:option name="output-dir" px:type="anyDirURI"
          required="true">
    ...
</p:option>

<p:option name="mediaoverlay" px:type="boolean"
```

```
            required="false" select="'true'" >
        ...
    </p:option>
```

Metadata is also provided to better integrate XProc scripts within the application and "hide" XProc idiosyncrasies. For instance, consider a script that produces an XML document; the user ultimately needs to have this document written to some location on a file system. The XProc script could declare an output port and the user would declare the path to the location where the XProc engine should write the document produced on this port (see Example 6). Alternatively, the script could declare an option representing a location on the file system (as a URI) and the document would be serialized to this location directly within the script using a `p:store` step (see Example 7). Adopting one or the other implementation pattern should be transparent to the end user who eventually just needs to invoke a script and have its results written somewhere. The intent of the option used in the second approach is therefore declared via an extended attribute, `px:output="result"`.

It shall be noted that this additional information is particularly useful in the scenario where the application is deployed in a server and accessed remotely. In that case, the user will typically not be asked to specify the path where to store a script's results on the server's file system; the application will rather control where it stores the results and make them available for later download. The `px:output` metadata is used to differentiate the options that are controlled and set by the application from the options that should be specified by the user.

Example 6. XProc script producing an XML document on an output port

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="1.0">
    <p:input port="source"/>
    <p:output port="result" sequence="false"/>
    ...
</p:declare-step>
```

Example 7. XProc script serializing an XML document at a location specified as an option

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="1.0">
    <p:input port="source"/>
    <p:option name="result-uri" required="true" px:output="result"/>
    ...
    <p:store>
        <p:with-option name="href" select="$result-uri"/>
    </p:store>
</p:declare-step>
```

Another use of extension metadata attributes is the differentiation of the "nature" of a script's results. Some results are intended to be directly rendered to the user, whereas others are intended to be stored on disk. For instance, when invoking a validation script the user will typically want to directly see the content of the validation report; on the contrary, the user would not be interested in seeing the result of a DocBook to HTML conversion but would rather have it stored in a file. A typical XProc command line tool would offer the flexibility to decide on one or the other approach. However, sometimes the application needs take control in place of the user: for example the result page of a Web-based user interface could in one case render the full validation report and in the other case just display a download button to access the stored result files. We also use the `px:output` extension attribute for this purpose, with a value set to either `result` or `report`.

5. Tips and best practices

Using XProc to process not only single XML documents but also entire publication file sets can require a particular mindset. The following sections describe what we found out to be good practices in our use of XProc and XSLT for ebook production ; these approaches can practically be generalized to any workflow involving the processing of collections of documents.

5.1. Representing File Sets

Although an ebook publication (for instance an EPUB [10]) is primarily composed of XML documents (XHTML content documents, OPF package document, Navigation Document, etc), it usually also includes a variety of non-XML data, like images and audio files. Producing content and converting between formats requires at least the ability of moving the files in the file system, and sometimes even processing them (e.g. transcoding audio, compressing image files, etc). Manipulating non-XML data is one of the major limitations of XProc, which has been designed to only let XML documents flow between steps. Some interesting approaches to support non-XML data processing have been proposed (see [11]) and may hopefully be supported in a future version of the language. In any case, we also needed the ability to handle a file set as a cohesive collection through the processing pipeline, and we found that we could generally do that by describing the file set as XML.

In order to describe a file set, we opted for a flat structure (see Example 8) rather than a recursive structure like the `c:directory` document described in XProc. It is mostly a matter of preference, and both structures could be easily converted one to another, but our experience was that flat structures were simpler to manipulate as we always consider files relative to the root of the publication file set.

Example 8. A file set XML representation

```
<d:fileset xmlns:d="http://www.daisy.org/ns/pipeline/data"
           xml:base="file:///Users/me/my-epub">
    <d:file href="mimetype"/>
    <d:file href="META-INF/container.xml"/>
    <d:file href="Content/package.opf"
            media-type="application/oebps-package+xml" />
    <d:file href="Content/toc.xhtml" media-type="application/xhtml+xml"/>
    <d:file href="Content/chap1.xhtml" media-type="application/xhtml+xml"/>
    <d:file href="Content/chap2.xhtml" media-type="application/xhtml+xml"/>
    <d:file href="Content/img1.jpg" media-type="image/jpeg"/>
    <d:file href="Content/img2.jpg" media-type="image/jpeg"/>
</d:fileset>
```

Based on this file set XML representation, we implemented generic XProc utility libraries to manipulate file sets virtually (only changing the XML representation) or physically (moving files on disk). Virtual file set operations include creating a new file set, adding an entry to a file set, re-basing a file set, joining a sequence of file sets, etc. Physical operations include storing in-memory documents on the file system, moving non-XML files from one location to another, etc. For these latter operations, we use Calabash's implementation of the EXProc file utilities [12].

5.2. Relying on base URIs

One of the interesting specifics of XProc is that XML documents flowing through XProc steps usually have a base URI (as part of the Infoset properties), regardless whether they actually represent physical resources. Our conversion modules rely heavily on this base URI information, notably to link documents to the XML file set representation described above.

For instance, consider an EPUB 3 production workflow where at the end of the processing we have XHTML content documents, the Package Document (OPF) and the Navigation Document readable through output ports of our XProc pipeline, as well as the XML representation of the file set as shown in Example 8. To actually store the result EPUB on disk, we simply send our in-memory documents and file set representation to a generic `px:fileset-store` step, which iterates over the element of the file sets, copies non-XML resources from their original locations, and otherwise uses `p:store` if a document with a matching base URI is found in the readable ports. This allows us to focus on the actual XML conversion logic and reuse the same storing code in all our converters.

Another useful pattern relying on base URIs is based on the `p:xslt` step's support for input sequences. When a sequence of documents is provided on the `source` port, it is available as the default collection in the XSLT. This allows us to perform multi-document look-ups directly in XSLT, using the base URI of the input documents to load them with the `collection` XPath function. An example of this approach is shown in Example 9, which is part of an XSLT that creates Media Overlays (SMIL-based synchronization of text and audio narration) for an EPUB XHTML Content Document.

Example 9. Using the default collection in XSLT

```
<xsl:template match="h:*[@id]">
    <xsl:variable name="clip"
         select="key('audio-clips',@id,collection()[/d:audio-clips])"/>
    <xsl:choose>
        <xsl:when test="exists($clip)">
            <par>
                ... SMIL audio element from the $clip info ...
            </par>
        </xsl:when>
        <xsl:otherwise>
            <seq epub:textref="{concat($content-doc-rel,'#',@id)}">
                <xsl:apply-templates/>
            </seq>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

5.3. Modularizing the workflow

One of the key benefit of a modular framework as described in Section 3 is the ability to reduce code duplication by creating re-usable components. Finding the right level of modularization mostly depends on the underlying use case, but we found that in general we want to split a single conversion module in at least three main steps: load, convert, then store.

- In the load step we basically load any input document passed as an option URI, and we also create the XML representation of the input file set by collecting all the relative URIs of external resources referenced in the master XML input documents.
- The conversion step receives in-memory documents along with an XML file set representation, applies all the core conversion logic, and produces in-memory documents and the XML representation of the output file set.
- The store step makes sure that the output file set is physically stored on disk at the required location.

Splitting the conversion logic in at least this three steps means that the core conversion step can be reused in other workflows by being chained to other such conversion steps, without having to store the files on disk and load them again in-between.

5.4. Keeping XProc as an orchestration technology

XProc comes with a rich set of standard steps and it can be tempting to rely on XProc as much as possible to implement the core processing logic. However, XProc should not be considered a hegemonic all-purpose solution. XProc shines at the expressive declaration of processing workflows. We found that the most readable and maintainable pipelines were those which were using XProc only to orchestrate processing steps otherwise implemented with tailored technologies like XSLT; the XML stack is rich and varied!

6. XProc wish list

This section presents some features or extensions of XProc that could generally improve its usability. The following list is not exhaustive and comes from one real-world experience of developing with XProc in the context of ebook production; other usages may reveal different needs. It is also worth noting that the requirements for the next version of XProc (v2) are — at the time of writing — being drafted by the W3C's XML Processing Working Group.

6.1. Ability to define XPath functions

The only way to extend the list of the XPath functions available to XProc is to register processor-specific extension functions, which typically have to be implemented in the programming language used by the processor. XPath functions are notably useful to post-process XProc options or to define case conditions in `p:choose` blocks. While it is possible to delegate the function definition and invocation to XQuery or XSLT (as shown in Example 10), the process is rather cumbersome. It would be much easier if there was a way to define native function libraries in XProc, and even better if XSLT-defined or XQuery-defined functions could be directly imported and callable from XProc.

Example 10. Invoking an XSLT-defined XPath function

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" version="1.0">

    <p:option name="uri" required="true"/>

    <p:xslt name="is-absolute-call">
        <p:with-param name="uri" select="$uri"/>
        <p:input port="source">
            <p:inline><irrelevant/></p:inline>
        </p:input>
        <p:input port="stylesheet">
            <p:inline>
```

```
                <xsl:stylesheet version="2.0"
                  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                  xmlns:functx="http://www.functx.com">
                    <xsl:import
                     href="http://www.xsltfunctions.com/xsl/▶
functx-1.0-nodoc-2007-01.xsl"/>
                    <xsl:param name="uri" required="yes"/>
                    <xsl:template match="/*">
                        <irrelevant
                           res="{functx:is-absolute-uri($uri)}"/>
                    </xsl:template>
                </xsl:stylesheet>
            </p:inline>
        </p:input>
    </p:xslt>
    <p:sink/>

    <p:group>
        <p:variable name="is-absolute" select="/*/@res">
            <p:pipe port="result" step="is-absolute-call"/>
        </p:variable>
        ...
    </p:group>
</p:declare-step>
```

6.2. Ability to access readable ports form XPath

It would sometimes be convenient to be able to access readable ports directly within an XPath expression. Example 11 shows how it is currently possible to use a variable computed from the content of an input port (in this case, a count of the number of readable documents). The pipeline would be more readable if the documents readable from the input port could be somehow accessed from the XPath expression, for instance via a well-known document collection (a naïvely proposed alternative is shown in Example 12).

Example 11. Counting documents on an input port

```
<p:declare-step name="main">
    <p:input port="source" sequence="true"/>

    <!-- Count if there are some docs -->
    <p:count name="count" limit="1">
        <p:input port="source">
            <p:pipe step="main" port="source"/>
        </p:input>
    </p:count>
    <p:sink/>

    <!--Repipe the primary source port-->
    <p:identity>
        <p:input port="source">
            <p:pipe port="source" step="main"/>
        </p:input>
    </p:identity>
    <p:choose>
        <p:xpath-context>
            <p:pipe port="result" step="count"/>
        </p:xpath-context>
        <p:when test="/c:result = '0'">
            ...
        </p:when>
        <p:otherwise>
          ...
```

Example 12. Counting documents on an input port - using XPath

```
<p:declare-step name="main">

    <p:input port="source" sequence="true"/>

    <p:choose>
        <p:when test="count(collection('main#source')) eq 0">
            ...
        </p:when>
        <p:otherwise>
          ...
```

6.3. Implicit connections based on port media types.

It is not uncommon in our use of XProc to have several "sets" of documents flowing through a sequence of steps, for example the XML description of a file set on the one hand and a sequence of in-memory XML documents on the other hands. In XProc version 1.0, a primary input port can be implicitly connected to the primary output port of the preceding step, but we do have to explicitly connect all the other ports. If XProc ports were to be annotated with media type information, as proposed for instance in [11], it would be possible to extend the implicit connection mechanism to automatically connect input and output ports of the same media type, provided there are no conflicting candidates.

6.4. Improved base URI manipulation

We've seen in Section 5.2 that the base URIs of the documents flowing through a pipeline can be highly useful in the processing workflows. However, the manipulation of base URIs has some limitations in XProc 1.0: it is not possible to set a document's base URI; furthermore, most of the document-producing steps do not allow to configure the base URI of their result.

The situation could be improved with the following extensions:

- a step to set the base URI of a document (see Example 13).
- a `base-uri` option on steps that produce new documents (e.g. `p:wrap-sequence`, `p:wrap`, etc).
- a mechanism to set the base URI of documents created with `p:inline`.

Example 13. A step to set a document's base URI

```
<p:declare-step type="p:set-base-uri">
    <p:input port="source"/>
    <p:output port="result"/>
    <p:option name="base-uri" required="true"/>
</p:declare-step>
```

7. Conclusion

This article is based on a concrete experience of using XProc at the heart of an ebook production tool. XProc's inherent modularity can be leveraged in a truly flexible component-based design. In addition, XProc provides all the extensibility hooks required to integrate it in a broader system, transparently to the end user. A set of techniques and best practices can be adopted to improve the XProc developer's productivity and make the resulting code more readable and maintainable. While several additions and usability improvements would certainly make future versions of XProc even more likeable, XProc is already a very powerful, mature and shining orchestration technology for the XML stack.

Bibliography

[1] *DAISY Pipeline 2. An open source framework for automated production of accessible digital publications.* The DIAISY Consortium. http://www.daisy.org/pipeline2.

[2] Norman Walsh, Alex Milowski, and Henry S. Thompson. *XProc: An XML Pipeline Language*. W3C Recommendation. World Wide Web Consortium. 11 May 2010. http://www.w3.org/TR/xproc/.

[3] Anders Berglund, Scott Boag, Don Chamberlain, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language (XPath) 2.0 (Second*

Edition). W3C Recommendation. World Wide Web Consortium. 14 December 2010. http://www.w3.org/ TR/xpath20/.

[4] Michael Kay. *XSL Transformations (XSLT) Version 2.0*. W3C Recommendation. World Wide Web Consortium. 23 January 2007. http://www.w3.org/TR/xslt20/.

[5] *OSGi Service Platform Core Specification*. Release 4, Version 4.3. The OSGi Alliance. April 2011. http://www.osgi.org/Specifications/HomePage.

[6] Norman Walsh. *XML Calabash*. *An XProc Processor.* http://xmlcalabash.com/.

[7] *The Saxon XSLT and XQuery Processor*. Saxonica Limited. http://saxonica.com/ products/products.xml.

[8] Florent Georges. *EXPath Packaging System*. Candidate Recommendation. EXPath Community Group. May 9 2012. http://expath.org/spec/pkg.

[9] Norman Walsh. *XML Catalogs*. OASIS Standard, V 1.1. OASIS Entity Resolution TC. 7 October 2005. https://www.oasis-open.org/committees/ documents.php?wg_abbrev=entity.

[10] Markus Gylling, William McCoy, and Matt Garrish. *EPUB 3.0*. IDPF Recommended Specification. International Digital Publishers Forum. 11 October 2011. http://www.idpf.org/epub/30/spec/.

[11] Vojtěch Toman. *XProc: Beyond application/xml*. XML Prague. 2012.

[12] Norman Walsh. *EXProc extensions for XProc*. EXProc Initiative. http:// exproc.org/.

Fully automatic database publishing with the speedata Publisher

XSL-FO++

Patrick Gundlach
speedata
<gundlach@speedata.de>

Abstract

XSL-FO is the current standard for creating paged media with XML. Due to the complete lack of built in capabilities to dynamically optimize the layout, XSL-FO is only suitable for simple documents.

We at speedata have build an Open Source replacement for XSL-FO that allows arbitrary complex documents. It has proven its usefulness in various commercial projects.

Keywords: XSL-FO, database publishing, PDF, typesetting

1. About the speedata Publisher

The speedata Publisher is an Open Source software for unattended XML to PDF typesetting workflows. It is used in commercial environments for the production of product catalogs (b2b), data sheets and offerings and others. Due to its usage of LuaTeX as a base, it offers highest quality output and great speed (up to 300 pages/second, 0.2 seconds for the first page), while being up to date with current technologies (UTF-8, r-t-l typesetting, OpenType fonts). It is available for download (both source and compiled for windows/linux/mac) on `http://speedata.github.com/publisher/`.

2. Static layout with XSL-FO

When you want to convert XML data to PDF, the first thing that comes into your mind is most likely XSL-FO. XSL-FO is a W3C standard to describe layout elements on (paged) media. It gives detailed information how formatters, also called renderers, display these elements on a page.

In practice, this usually works by enriching given XML data with an XSLT transformation to include layout specifications and pass the resulting data stream to a renderer. This works fine in many simple cases, but can become complex and tedious in more advanced layout requirements. In demanding layout requirements we usually deal with optimizations that cannot be done by XSL-FO. Reorganizing or rearranging elements on a page with respect to some given parameters or constraints can only be solved in a very complex manner.

This is due to the static nature of the XSL-FO workflow. With XSL-FO, the layout is mostly finished, before it gets to the formatter. You cannot "ask" the formatter how big a layout element will be when rendered, because the formatter is run after the FO elements (and thus the layout) are finalized. Take for example the simple case: does a given text fit into one line without a line break? There are two solutions to this case. The first is an approximation. We can count the number of letters in a text and guess if it will fit in a line. This is obviously an inferior solution. The second approach would be a multi pass solution to find out how the formatter has typeset the text and write that output into an intermediate file. This information can be used in a second run to optimize layout. This approach works reasonable with small documents and small optimization steps. The optimizations we encounter in day to day typesetting tasks are much more demanding and would not work feasibly with one of these approaches.

3. Dynamic layout

The solution to this problem is to closely integrate the layout description language and the formatter. If the layout description language is powerful enough (when it has control structures like a programming language such as Java), one can dynamically "ask" the renderer how large page elements are and act on the resulting information. With the example above, a simple approach could be like this (pseudo code):

```
typeset the text on a virtual page
(the size of the virtual page is now exactly the size of the text)
if height of this virtual page is larger than a given amount
  .. do this
else
  .. do something else
end
```

With this approach there are no approximations and no multi-pass solution involved. The result of the size is available immediately during the layout generation and can be used to determine the future layout of the page. This only works because the layout instructions are interpreted by the renderer which knows the exact dimensions of every element in the output media.

The more common dynamic optimization problems are among others:

- Place text on a page. If the page is overfull, decrease the font size of the text until it fits on the page. Issue a warning message if this happens.

- In a catalog of consumer products: rearrange products on a page such that one more product can fit on the page. The parameters that are allowed to change are: image size, amount of additional information, order of the products.
- Typeset a text in many languages. The amount of space the text occupies is determined by the "largest" text. That way color images can be placed at the same position in every catalog variant, thus saving money if you only need to change the black color plate in printing while the other colors can be the same in all languages.

While these problems can be solved easily in manual workflow (such as InDesign), they are hard to do in a fully automatic way. With our product, the speedata Publisher, we can handle these problems easily while providing a 100% automatic workflow.

4. Implementation

The main problem we were facing implementing this approach is that there is no (W3C or similar) standard currently supporting this kind of dynamically optimized paged media output. For output styling, CSS seems to be useful, combined with some elements from HTML (especially the HTML table model seems easy yet powerful enough to handle all practical problems). For data access, a combination of XPath and some elements of XSLT are the natural choice these days. To handle the dynamic nature we implemented a simple but fully functional (Turing complete) language that is interpreted within the typesetting engine. Therefore it is possible to typeset material (text, tables, images and other elements such as barcodes) on a virtual page and find out the dimensions of this virtual page. You can then reorganize the material based on the exact dimensions and not on guesses simply by discarding the contents of the virtual page and recreating it if necessary. If the virtual page is acceptable for placement in the PDF, it can be placed on the main page.

We have based the implementation on the TeX typesetting system from the 1980s which is known for its highest typographic standards and stability. The relatively new implementation called LuaTeX outputs PDF and is fully unicode aware and handles OpenType fonts natively. In addition to that, LuaTeX is easily extensible by software written in the Lua language, a relatively simple to learn language that has similarities with Java, Ruby and other languages in this area. With LuaTeX, one can access TeX's internal datastructures and routines from Lua which is not possible in classic TeX, and therefore it is natural to interface these routines from Lua and thus bypassing all the macro programming that would be necessary in classic TeX.

The user of our system is not exposed to Lua or any other parts of the implementation. The interface to our publishing software is through an XML based language (the layout instructions) that has declarative and procedural parts. The declarative part could look like this:

```
<Layout
  xmlns:sd="urn:speedata:2009/publisher/functions/en"
  xmlns="urn:speedata.de:2009/publisher/en">

  <LoadFontfile name="sans" filename="Helvetica-Regular.otf"/>
  <LoadFontfile name="bold" filename="Helvetica-Bold.otf"/>

  <DefineFontfamily name="text" fontsize="12" leading="14">
    <Regular fontface="sans"/>
    <Bold fontface="bold"/>
  </DefineFontfamily>

  <Pageformat width="210mm" height="297mm"/>

  <DefineColor name="yellow" model="cmyk" c="0" m="0" y="100" k="0"/>

</Layout>
```

The declarative section of the layout defines the overall appearance of the document, including the definition of master pages based on arbitrary tests (which are XPath expressions). Pages can contain positioning areas that are sub areas on the page used for automatic text flow from one area to the next area of the same kind, introducing a page break if necessary. Thus you can divide your page into header, footer and a page body. During page initialization and page shipout (to PDF) you can run instructions that could produce a running head (including the current page number or title).

What follows is the part of the data handling itself. The layout instructions are kept separate from the data (also encoded in XML) but are interpreted at the same time. We start at the root element in the data file and find a matching "Record" element in the layout instructions. If the root element is called "data", the system looks for an entry in the layout file `<Record element="data">` and starts processing the instructions found in the child elements of `<Record>`. Surprisingly few building blocks are necessary to solve the typesetting tasks and optimizations:

`PlaceObject`	We can place any kind of object (image, text, table, barcode, lines, virtual pages) at a given position. The coordinate system we use is a user definable grid or absolute coordinates). If the user does not supply a position, the system tries to find the next available space, first from left to right, then from top to bottom. The exact outcome of the command depends on the type of the material. Tables for examples are automatically split across pages, including a user definable continuing header and footer.

	Text can be typeset in columns or broken across pages.
Variable assignment	As in other programming languages, variables can hold any kind of data. Variables can for example be used to accumulate table data or to store font sizes or positioning information. Variables can be set based on the current data, and therefore allows the clear separation of data and layout instructions
While loops / Switch, Case, Otherwise conditionals	These classical programming constructs can be nested and allow complex design decisions.
Virtual pages (so called "groups")	A "group" contains material that is not placed in the PDF but kept in a virtual page of a given name. All instructions in the child elements are executed inside this virtual page that starts with size of 0. The size of the virtual page extends to the necessary amount given by its contents.
Child elements	To access child elements in the data file one can call the routine `<ProcessNode select="...">`. The instructions in the corresponding `<Record>` element gets executed for each matching child element.
Datasets	Any kind of data can be read from and written to an intermediate XML file. This simple mechanism allows dynamic table of contents, indexing and other information that depends on the current page. Is is also useful to generate customized status reports.

The instruction set of the layout is defined in a "master" RelaxNG grammar. Together with a translation file this "master" is translated into different languages, currently English and German. The user can choose the favorite interface language for the layout instructions. The speedata Publisher uses the same translation file to translate the instructions written by the user ("`layout.xml`") back to the master language. So the user can decide if he or she prefers "`while`" or the German "`solange`". The requested language is given by the namespace of the layout elements: the prefix is always `urn:speedata:2009/publisher/functions/` and the suffix denotes the interface language ("en","de"). The interface language must be the same within a file.

5. The backend

As mentioned above, we use LuaTeX as our backend, but this could be exchanged by many other software, such as Adobe InDesign or any other self programmed backend. We use LuaTeX mainly for these features:

- PDF writing. This is the most visible part of LuaTeX that we use. TeX uses a two step process when creating the output: first, the user's input (high level macros such as `\section{...}` and low level primitives such as `\hbox{}`) is interpreted and converted to an intermediate data structure, unaccessible to the user. The second step is the conversion of this intermediate format to PDF. In the speedata Publisher we skip the first step and create the intermediate data structures ourselves and use TeX only for writing out the PDF. LuaTeX is the only TeX implementation that allows the skipping of the first step. The data structure is relatively easy to handle. It consists of so called *nodes* that are chained in linked lists and represent tree structures. There are approximately 10 different type of nodes that are used in actual implementation. A node represents a glyph, a stretchable space, a vertical or horizontal container (so that everything inside is stacked in that direction) or PDF instructions to implement arbitrary PDF elements.
- Paragraph breaking. TeX is excellent in breaking paragraphs into lines so we use internal TeX routines for that. This includes hyphenation. The resulting data structure can be used for the second step mentioned above.
- Image and font inclusion. With LuaTeX we can easily load images and fonts in different formats (png, pdf, jpeg, Type1, OpenType) and use them in resulting PDF.

The list above should make it clear that we could exchange LuaTeX by other software providing this functionality. To use other words (for those who know TeX, the typesetting system): we do not create any TeX macros to be interpreted by TeX or LaTeX. The macro language has many drawbacks that we wish to avoid.

It would be not too difficult (in theory) re implement our application in Adobe InDesign. InDesign has a rich API that can be accessed via JavaScript and it surely provides the three features listed above (PDF writing, paragraph breaking and font/image inclusion).

A. A sample application (hello world)

This sample application is the classical *hello world*. Note that every application consists of two files, a layout instruction file and a data source

A.1. The layout instruction file ("`layout.xml`")

```
<Layout
  xmlns="urn:speedata.de:2009/publisher/en">
  <Record element="data">
    <PlaceObject>
      <Textblock width="10">
        <Paragraph>
          <Value>Hello </Value><Value select="string(.)"/><Value>!</Value>
        </Paragraph>
      </Textblock>
    </PlaceObject>
  </Record>
</Layout>
```

A.2. The data file ("`data.xml`")

```
<data>
  world
</data>
```

If you have an installation of the speedata Publisher, you can run it by giving the command `sp` on the command line in a directory with the two files above.

The software starts by reading the layout instructions. There are defaults for paper size, grid width and font usage which you can change of course. Now the publisher starts reading the data file at the top element (here: `data`) and executes the instructions by the matching `<Record>`. The command `<PlaceObject>` places images, texts, tables and other material on a page. The position can be given by a row and column attribute that takes an integer number, which denotes the address of a grid cell or as an absolute position on the page. If no coordinate is given, the publisher tries to place the object at the *current* position if possible, taken the size of the object into account.

All material has to have a given size (width or height) if it doesn't have a natural size (such as an image for example). In textblocks for example, the line width is given by the `width` attribute. Textblocks can contain more than one paragraph and each paragraph consists of text and images.

In the sample above, the paragraph is build from three parts. The first and third part is a static text, the middle part uses XPath to address a node in the data. We could also use `@` to address attributes or `$` for variables that are set with `<SetVariable>`.

B. A simple optimizing application

The following application checks if an image exceeds a given width and if that is the case, places the image and a text below each other instead of horizontally adjacent. Note that this is a static optimization that could be performed my XSL-FO.

B.1. The layout instruction file ("`layout.xml`")

```
<Layout
  xmlns="urn:speedata.de:2009/publisher/en"
  xmlns:sd="urn:speedata:2009/publisher/functions/en">

  <Options
    show-grid="yes"
    show-gridallocation="yes"/>

  <Record element="data">
    <SetVariable variable="imagename" select="'myimage.pdf'"/>
    <SetVariable variable="imagewidth" select="sd:imagewidth($imagename)"/>
    <Switch>
      <Case test=" $imagewidth > 10">
        <PlaceObject>
          <!-- Place image above text -->
          <Image width="{$imagewidth}" file="{$imagename}"/>
        </PlaceObject>
        <NextRow/>
        <PlaceObject>
          <Textblock width="{sd:number_of_columns()}">
            <Paragraph>
              <Value>...</Value>
            </Paragraph>
          </Textblock>
        </PlaceObject>
      </Case>
      <Otherwise>
        <!-- place textblock right of image -->
        <PlaceObject>
          <Image width="{$imagewidth}" file="{$imagename}"/>
        </PlaceObject>
        <!-- the cursor is now in the next grid cell right of the image, so ▶
we don't need
             to provide a coordinate -->
        <PlaceObject>
          <Textblock width="{sd:number-of-columns() - $imagewidth}">
            <Paragraph>
              <Value>...</Value>
            </Paragraph>
          </Textblock>
        </PlaceObject>
      </Otherwise>
    </Switch>
  </Record>
</Layout>
```

We don't provide the data file, as we have only static content in this example. The publisher provides a command line switch (`--dummy`) that provides a simple data file without creating one.

C. Dynamically optimizing application

This sample application typesets a table onto a virtual page and depending on the height of the virtual page (which has the same height as the table) it either re-typesets the table using a different font size and width or it keeps the table.

C.1. The layout instruction file (“`layout.xml`”)

```
<?xml version="1.0" encoding="UTF-8"?>
<Layout
  xmlns="urn:speedata.de:2009/publisher/en"
  xmlns:sd="urn:speedata:2009/publisher/functions/en">

  <Record element="data">
    <!-- A group is a virtual page with unknown size
         (starts with width/height of 0) -->
    <Group name="a table">
      <Contents>
        <PlaceObject>
          <Table width="6">
            <ForAll select="entry">
            <!-- one table row for each data set 'entry' -->
              <Tr valign="top">
                <Td>
                  <Paragraph>
                    <Value>Entry: </Value>
                  </Paragraph>
                </Td>
                <Td>
                  <Paragraph>
                    <Value select="string(.)"/>
                  </Paragraph>
                </Td>
              </Tr>
            </ForAll>
          </Table>
        </PlaceObject>
      </Contents>
    </Group>
    <!-- At this point we know the exact height of the table
         (= the group) because we have typeset it onto the virtual
         page. There is no guessing involved. -->
    <Switch>
      <Case test="sd:group-height('a table') > 3">
        <!-- The table is too high for our purpose,
             we need to re-typeset it using different
             parameters (font size, width etc.) -->
        <Group name="a table">
          <Contents>
            <PlaceObject>
              <Table width="10">
            <!-- We leave out the actual table contents (as above) for brevity ▶
-->
              </Table>
            </PlaceObject>
          </Contents>
        </Group>
      </Case>
    </Switch>
    <!-- Now we know that the group called 'a table' contains
         a table that is valid for our purpose -->
    <PlaceObject groupname="a table"/>
  </Record>
</Layout>
```

C.2. The data file ("`data.xml`")

```
<data>
  <entry>A sample entry.</entry>
  <entry>More text for our database.</entry>
  <entry>We don't know how long this text is.</entry>
</data>
```

C.3. Explanation

The example above shows how we use the fact that the speedata Publisher interprets the layout instructions during the typesetting run. We can typeset material (a table in the example) and find out about the dimensions it takes. We can use the virtual page (the group) as if we were on an infinitely stretchable page and get the size of that page. This measured dimensions are what you would expect: if you place an object of width 5 and height 5 at position (6,6), the virtual page has the size width 10 and height 10. The publisher starts all coordinates at (1,1) at the top left corner.

D. Generating a table of contents

This example collects data during the typesetting run, writes it out to an external XML file and uses its content to generate a table of contents in the next run.

On the first run, the instructions in `<Record element="tocroot">` are not executed because the intermediate XML file is not written yet. On the subsequent runs the command `<LoadDataset>` reads the given file and executes the `<Record>` command corresponding to the root element of the toc file (`<tocroot>`) which just typesets the contents of the file in a table.

D.1. The layout instruction file ("`layout.xml`")

```
<Layout
  xmlns="urn:speedata.de:2009/publisher/en"
  xmlns:sd="urn:speedata:2009/publisher/functions/en">

  <DefineColor name="green" model="cmyk" c="22" m="0" y="55" k="0"/>
  <DefineColor name="blue" model="cmyk" c="36" m="2" y="1" k="0"/>

  <!-- The 'tocroot' element comes from the dataset 'toc'
       that was written on the harddrive with SaveDataset -->
  <Record element="tocroot">
    <PlaceObject row="10" column="5">
      <Table width="7" stretch="max">
        <Tr align="center" backgroundcolor="green">
          <Td>
            <Paragraph><Value>Title</Value></Paragraph>
          </Td>
          <Td>
            <Paragraph><Value>Page</Value></Paragraph>
          </Td>
        </Tr>
        <!-- The 'tocentry' element comes from 'Element'
             command that creates an XML element (similar to XSLT) -->
        <ForAll select="tocentry">
          <Tr backgroundcolor="blue">
            <Td>
              <Paragraph><Value select="@name"></Value></Paragraph>
            </Td>
            <Td>
              <Paragraph><Value select="@page"></Value></Paragraph>
            </Td>
          </Tr>
        </ForAll>
      </Table>
    </PlaceObject>
  </Record>

  <Record element="data">
     <!-- On the first run, the 'toc' dataset is empty, thus 'nothing happens'. ▶
On the
         subsequent runs, the dataset is nonempty and the root element is ▶
'tocroot',
         so the rule 'tocroot' above is executed. -->
    <LoadDataset name="toc"/>
    <NewPage/>
    <ProcessNode select="entry"/>
    <!-- This writes an XML file to the harddrive so that it can be used in the
         subsequent runs -->
    <SaveDataset filename="toc" elementname="tocroot" select="$tocstructure"/>
  </Record>

  <Record element="entry">
    <!-- Add the element 'tocentry' to the variable 'tocstructure' -->
    <SetVariable variable="tocstructure">
      <Copy-of select="$tocstructure"/>
      <Element name="tocentry">
        <Attribute name="name" select="@title"/>
        <Attribute name="page" select="sd:current-page()"/>
      </Element>
    </SetVariable>

    <!-- We place the dummy text from the database on each page -->
    <PlaceObject>
      <Textblock width="10">
        <Paragraph><Value select="."/></Paragraph>
      </Textblock>
    </PlaceObject>
    <NewPage/>
  </Record>
</Layout>
```

D.2. The data file (“`data.xml`”)

```
<data>
  <entry title="One">First text.</entry>
  <entry title="Two">Second text.</entry>
  <entry title="Three">Third text.</entry>
</data>
```

D.3. The intermediate dataset file generated from `<SaveDataset>`

```
<tocroot>
 <tocentry name="One" page="2" />
 <tocentry name="Two" page="3" />
 <tocentry name="Three" page="4" />
</tocroot>
```

Representing Change Tracking in XML Markup

Robin La Fontaine
DeltaXML Ltd
<robin.lafontaine@deltaxml.com>

Tristan Mitchell
DeltaXML Ltd
<tristan.mitchell@deltaxml.com>

Nigel Whitaker
DeltaXML Ltd
<nigel.whitaker@deltaxml.com>

Abstract

This paper presents work done over the past two years to provide an improved change tracking representation for documents in XML. The original intention was to provide improved change tracking for the OpenDocument format (ODF), but the approach is generic and is therefore potentially applicable to other XML document formats and even XML data.

A detailed specification was developed and prototype implementations developed in Abiword and KWord to demonstrate interoperability. However, developers of the main ODF office packages found the approach a challenge and were less keen to implement it and are currently looking at other options.

This paper presents the basic design principles behind the proposal, and how these are satisfied in the approach taken. Since the initial work, there has been interest from the wider XML community and new requirements relating to its use within XML editors have also been proposed. There is now a W3C Community Group formed specifically for change tracking markup, and a standard in this area could have significant benefits for the XML community as a whole.

1. Introduction

The ability to track changes made to text documents is commonly available in document editing systems. These are now moving to XML, e.g. OOXML and ODF [2]. At the same time, structured document formats all use XML and currently do not have any extensive capability to track changes. The change-tracking capability of

XML editors is fairly basic, for example many do not track attribute changes, and there is no common standard. The lack of a standard means that documents with changes tracked cannot be moved between XML editors.

A standard way of tracking changes in XML documents would provide many benefits:

- documents with tracked changes could be moved from one XML editor to another
- XML editors could track changes in any XML document type
- every XML document type could include a change history and the ability to roll back to previous versions
- software designed to handle change in XML could be applied to many different XML document types

The state-of-the-art at present is that every XML document type takes its own approach to change tracking. OOXML is built on the underlying binary model within Microsoft Word. ODF has only a very limited capability to track some changes. DITA uses rev and status attributes to indicate changes and DocBook similarly has a revisionflag attribute, but neither can track attribute changes or complex structural changes.

XML editors track changes either by additional markup or using Processing Instructions (PI). Additional markup has the advantage of being easily processed using standard XML tools but at the cost of modifying the underlying schema. PIs have the advantage of preserving the latest state of the document in valid XML markup but the PIs do not have structure and so are limited in the changes they can track.

The original purpose of the change tracking format described in this paper was to improve the change tracking within ODF. The proposal was known as GCT (Generic Change Tracking) and full details can be found in [1].

ODF is a large and complex XML format, including representation of textual documents, graphics, spreadsheets and presentations. Due to its complexity, it seemed sensible to take a generic approach so that any change to the underlying XML could be represented in a precise and unambiguous way. The scope of the actual change tracking within specific elements of ODF could then be constrained within a modified version of the RelaxNG schema.

It was generally agreed within the ODF community that the generic approach proposed would be applicable to other XML documents. However, the developers of ODF office packages felt that they would not be able to implement such a generic approach because their own internal data structures were very different from the XML representation. They felt that the ability to represent any change was beyond what was needed and too complex for ODF editing applications to handle. Prototype implementations were however completed for two such office packages and these demonstrated both that the proposal could be implemented and that there was interoperability between two independent implementations.

2. Outline of Paper

In this paper we first explore some of the requirements for change tracking within documents, and the basic principles behind the design of the change tracking format.

We outline the reasons behind the two implementation levels for change tracking: the simplest level allows any change to be represented, but in a non-optimal way; the second level allows structural change to be represented, at a cost of more complexity. Issues of validation of changes are discussed.

Changes need to be structured in some way in order to represent interdependency and to provide meta data including timestamps and author details.

Examples are given to show how attribute changes, element deletions and additions, and text changes are represented.

The issues around structural changes are discussed, and the way that these are represented is described.

The paper outlines how the generic ability to track changes can be used independently or integrated into a particular schema, in a way that provides some control over what can be changed.

Finally, the paper outlines some of the as yet unresolved issues.

3. Requirements for XML Change Tracking

3.1. Distinction between 'edit tracking' and 'revision tracking'

Opinions within the ODF group were divided on the subject of the ultimate purpose of change tracking, but the discussion was useful and raised some important issues. Some viewed change tracking as a record of the edits made by an application. Others viewed change tracking as a record of the changes between two revisions of a document. These can be characterized as an 'Application viewpoint' and a 'Document viewpoint', and they lead to two different interpretations of change tracking.

The Application viewpoint asserts that change tracking should support the features in editors, no more and no less. Therefore editor application programmers need to agree on what these edit operations are so that they can be unambiguously represented, in this case in the ODF document format. The Document viewpoint asserts that change tracking should support changes to the document, in this case the XML representation of the document in ODF, so it should be possible to roll back to any previous version.

The Application viewpoint did not want a Document viewpoint solution because it was considered hard to implement. Existing ODF implementations tended to read ODF files into internal editor specific data structures and not keep any association with between the XML representation and the internal one. The Document viewpoint regards the Application viewpoint solution as inadequate for other applications and a moving target: as editors add new features the standard will need to be changed. It was noted that a Document viewpoint solution must include all of the needs of the Application viewpoint.

This leads to a refinement of the term 'change tracking' to two variants:

'**Edit tracking**' is the ability to record edits made in an editing application in order that they can be viewed, accepted or rejected at a later date.

'**Revision tracking**' is the ability to record changes to a document such that these changes can be displayed in a document viewer, and the document rolled back to a previous version.

When the scope is widened from ODF to any XML, there are many different applications and therefore it would not be possible to have a single standard for edit tracking. If the revision tracking is based on changes to the underlying XML representation, then a single standard is possible and potentially useful across many different XML schemas.

3.2. ODF Requirements

These are the requirements that were discussed for ODF in the Advanced Document Collaboration subcommittee.

1. **Reversibility:** At its most basic level, change tracking, as its name suggests, is the ability to track or record a change to an XML document. The way that a change is recorded needs to be capable of being reversed or undone, typically to support undo and/or redo operations of an editor. Therefore by undoing changes that have been tracked, it is possible to move back to a previous version of a document. A standard for change tracking should include a very clear definition of how a particular change is reversed.
2. **Easy to ignore:** Change tracking information is additional information for a document, and there will always be applications that are not interested in it. Therefore it should be easy to ignore changes that have been tracked, and if the changes are ignored then the result should be the latest version of the document.
3. **Use markup:** Since this is an XML standard with associated schema, it makes sense that the changes should be recorded in standard XML markup.
4. **Granularity:** It is preferable that changes should be tracked at a low level of granularity. For example, it should be possible to represent the deletion of a single word or a single character within a larger PCDATA segment and not have to delete all of the textual content, and then add new text without the word or character.
5. **Grouping:** Although some changes are quite simple, other changes, such as a global search and replace operation or the deletion of a column in a table, are more complex although they still need to be considered as a single reversable

change operation. This implies that the ability to group simple changes together into more complex units is necessary.

6. **Dependency:** As mentioned above, it is fundamental that the tracked change can be reversed. The reversal of changes does however introduce a new issue, which is that it is not always possible to reverse one change without first reversing, or accepting, some other change. Consider for example the case of a spelling correction to some text inside a paragraph which is them deleted entirely.
7. **Deleted content outside document body:** In an ODF document, all of the text within the main body of the document is deemed to be a part of the document. The advantage of this is that an application that does not understand particular markup can ignore it and just process the content. This means that any deleted item will be represented by a marker, and the actual deleted content will be elsewhere in the document. This requirement was to maintain compatibility with previous versions of ODF.
8. **Change tracking markup integrated with schema:** The markup for change tracking needed to be part of the schema (RelaxNG) for ODF.

3.3. General XML Requirements

Most of the above requirements were established in the initial work with ODF. When looking more generally at requirements for XML editors and authoring tools some of the requirements may be adjusted from those for ODF. Note that these reflect our view which does not necessarily reflect the views of others involved in the W3C community group.

The scope of a change tracking standard also needs to be considered. For example, it would probably be sensible to exclude the representation of changes to a DTD, and probably also exclude changes to entities. Changes to processing instructions and comments would ideally be included, but they are likely to introduce an extra degree of complexity.

A common practice for XML editors is to use Processing Instructions (PIs) to represent changes, and therefore it seems highly desirable to have a representation that uses processing instructions. The reason that XML editors use processing instructions is so that the document itself remains valid relative to its schema. The ability to swap between a markup representation and a processing instruction representation, in a way that is completely lossless, would make a change tracking standard more versatile.

A processing instruction representation was not provided in the ODF work but a simple approach to this would be to convert the outermost element into a processing instruction, and its content becomes the processing instruction content. This approach needs to be validated. Initial experimentation using the oXygen editor suggests that it would work subject to some constraints.

Another implication of the use of processing instructions is that the deleted content can be represented in situ, and does not need to be moved to another part of the document. Therefore the requirement noted above that deleted content should be outside the document body is not necessarily a requirement for uses in other XML document formats.

3.4. Validation of changes

It is certainly necessary to define whether or not a particular change is valid. This is potentially very complex. However, XML provides many mechanisms for validating an XML document, whether it is against a schema or with additional Schematron rules or NVDL validation. In order to validate changes, the validation must take account of the document that is being changed.

We can circumvent this complex issue by saying that if the document before the change is valid, and if the document after the change is valid, then the change is valid. This is simple and intuitive to understand, and completely removes the need for more complex validation of changes.

4. Levels of Complexity

It is always good to strive for the simplest possible solution to a problem. But simple solutions can be limited, and increasing complexity is sometimes needed to provide a more useful solution.

It is possible to represent any change to an XML document using just the deletion and addition of elements. However, this is not particularly useful, because it would mean that an entire subtree would need to be deleted and added in order to represent the change to a single attribute. Similarly, it is evident that the ability to represent changes to textual content is necessary. This leads to a basic level (Level 1) of change tracking ability which would include the addition and deletion of elements, the addition and deletion of text, and the addition, deletion and modification of attributes.

As an aside, it may seem odd that only attributes can be modified, and not text or elements. It turns out to be convenient to have a single operation to modify the value of an attribute, whereas there is little to be gained by having a special operation for modification of text: modify-text has no advantage over deletion and addition. Regarding elements, modification of an element involves some change to its attributes or content, so there is no need for a special modify-element operation.

Although this basic level of complexity seems to be adequate at first sight, it soon becomes clear that it is not always sufficient. This is certainly the case when the XML is used to represent documents. This is simply because XML documents tend to use structural changes to XML in order to represent changes that a viewer might consider to be only aesthetic, for example the addition of text decoration.

Another example of this is when an editor inserts a newline in the middle of a paragraph, i.e. splits it into two paragraphs. Level one represents this as a paragraph with deleted content and another added paragraph, however an editor does not expect to see change to those paragraphs, but rather the insertion of a new line. Such changes do not always fit well with the underlying XML structure.

In order to avoid the need to delete and add potentially large amounts of content in order to show such changes, we need to introduce the ability to add and delete structural information, i.e. XML tags. This does make the change tracking much more precise, though at a cost of additional complexity. It may also be argued that a typical XML editor will allow the addition of structural markup inserted around existing content, so again there is a need for a more precise representation of this.

Therefore, **Level 1** provides the ability to modify attributes, add and delete elements, and add and delete text. It also enables changes to be grouped into transactions where a single transaction moves the document from one valid state to another.

Level 2 adds to this the ability to add or delete element structure around existing content and to split and merge elements in more complex ways.

In the following sections we will describe these two levels, and show examples of how changes are represented. They are not intended to be a formal definition of how the change tracking format works, but rather an introduction.

5. Level 1

5.1. Change Transaction (CT) Structure

This structure provides a place for metadata and a structure to define any dependencies between changes.

There must be a position in the document where the change transactions are defined, each being identified by an ID. Each will have some associated meta information such as the name of the author who made the change, and the time and date.

The ordering of the change transactions is important. If a user wishes to undo the changes one by one, then this can be achieved by undoing the last change transaction, i.e. it is a stack of transactions.

It is also possible to group CTs in a change transaction group (CT group). This will have similar meta information to a CT. All the members must be previously-defined CT or CT groups. The effect of undoing a CT group will be to undo a number of CTs.

A CT group may be ordered (a CT stack) or unordered (a CT set). The members of a CT set can be accepted or rejected in any order. The members of a CT stack must be accepted or rejected in the defined order, i.e. undo last member first. An example of a CT set would be a global textual replace operation, the user may wish to accept all of the replacements as one operation, or accept/reject them individually in any order. A change that depends on another change would be represented together as a stack.

5.2. Changes to Attributes

Attribute changes are tracked within new attributes. The reason for doing it this way is to make the minimum structural changes to the document. Typically, there will only be one or two attribute changes within an element, although if there were a large number of changes then this would not be very readable.

An element will always contain the latest version of its attributes. This means that if an attribute is added, we only need to record the fact that it has been added, and not its value because the value will be specified in the element. When we change the value of an attribute, then we need to keep a record of the previous value so that the change can be undone. Similarly, for attribute deletion we need to keep a record of the original value.

Each attribute change will reference a change transaction, and this and other information is encoded in a new attribute which is in a defined namespace, but the actual name of the attribute is generated. The information that is encoded within the attribute value is as follows:

1. The change transaction (CT) ID. This is a reference to the ID.
2. The type of change: insert, remove, modify
3. The name of the attribute that is changed
4. The old value of the attribute – this is not needed for an added attribute because the value will either be in the element or, if the attribute is later deleted it will be recorded there.

Example 1. Attribute addition: the outline-level attribute has been added

```
<text:p text:style-name="Standard" text:outline-level="3"
   ac:change001="ct1,insert,text:outline-level">
   How an attribute is added
</text:p>
```

Example 2. Attribute deletion: the outline-level attribute has been deleted

```
<text:p text:style-name="Standard"
  ac:change001="ct1,remove,text:outline-level,3">
  How an attribute is deleted
</text:p>
```

Example 3. Attribute modification

```
<text:p text:style-name="Code"
  ac:change001="ct1,modify,text:style-name,Standard”>
```

```
    The style on the paragraph has been changed from Standard to Code
  </text:p>
```

5.3. Changes to Elements

An element is marked as inserted with an attribute, delta:insertion-type="insert-with-content" (we will discuss the other insertion types later). There will also be a reference to a CT which will have all the meta-data associated with this change.

Example 4. Element insertion

```
<text:p delta:insertion-type="insert-with-content"
        delta:insertion-change-idref='ct1234'>
        This paragraph is inserted.
</text:p>
```

When an element or other content is deleted, it is wrapped in an element <delta:removed-content/> to indicate it is no longer part of the document. This element can contain mixed content, ie one or more elements and/or text that was removed as part of the same editing operation.

Example 5. Element deletion

```
<delta:removed-content delta:removal-change-idref='ct456'>
                <text:p>    This paragraph is deleted.   </text:p>
</delta:removed-content>
```

Note that a deleted item may contain changes within it, but the changes must all be before its deletion.

5.4. Changes to Text

Text addition uses the conventional method of setting a marker at the beginning of the addition and a corresponding marker at the end. These markers are empty elements, and they are linked using an ID.

Example 6. Simple text insertion

```
<text:p>
How text is
   <delta:inserted-text-start delta:inserted-text-id="it632507360"
     delta:insertion-change-idref=”ct1”/>
very easily
   <delta:inserted-text-end delta:inserted-text-idref="it632507360"/>
added.
</text:p>
```

Additions may not always be within a single element, but the delta:inserted-text-start and delta:inserted-text-end must both have the same parent element when they are created, and the content between them must be PCDATA only. Therefore when a second paragraph is added as per the example below, the first atomic change terminates and the paragraph is added in the normal way. The CT reference provides a link to indicate these occur at the same time as a single addition. This avoids having two ways to add an element and avoids the need to track across the element hierarchy to find the corresponding end of an addition.

Example 7. Text insertion that flows into a new paragraph

```
 <text:p>
How text is
   <delta:inserted-text-start delta:inserted-text-id="it123"
      delta:insertion-change-idref=”ct3”/>
very easily added.
   <delta:inserted-text-end delta:inserted-text-idref="it123"/>
</text:p>
<text:p delta:insertion-type="insert-with-content"
      delta:insertion-change-idref=”ct3”>
And the addition is into a second paragraph.
</text:p>
```

Additions must therefore always be non-overlapping and the start and end of a change must be within a single element, when they are formed. Of course they may not be within a single element at some later stage due to other changes, but in this case it would not be possible to 'undo' it. This rule adds clarity at the slight cost to the writer application, i.e. the application creating the change, and the considerable gain for the reader application, i.e. the application consuming the change. Since any number of atomic changes can be associated with a single CT, there is no loss of information.

Text is marked out as deleted in exactly the same way as an element is marked as deleted, i.e. it is wrapped within a change tracking element.

Example 8. Simple text deletion

```
 <text:p>
How text is
  <delta:removed-content delta:removal-change-idref=”ct2”>
   deleted or </delta:removed-content>
removed from a paragraph.
</text:p>
```

If the content is not simple text, but mixed content, it is handled in the same way.

Example 9. Mixed content deletion

```
 <text:p>
How text is deleted
   <delta:removed-content delta:removal-change-idref=”ct2”>
     or <text:span text:style=”bold”>removed</text:span> like this
   </delta:removed-content>
from a paragraph.
</text:p>
```

6. Level 2

6.1. Add an element around some existing content (insert-around-content)

In document editing, it is common to add text decoration or structural information to existing content. As the content itself is not changed, we wish to reflect just the addition or change of the structure.

Example 10. Addition of a <span> element around some text

```
 <text:p> This text will be made
   <text:span text:style-name="bold-style"
     delta:insertion-type='insert-around-content'
     delta:insertion-change-idref='ct1234'>
   bold
   </text:span>
  . </text:p>
```

Since this tag has been added around the content, when the change is undone its content will remain in place.

6.2. Delete an element but not its content (remove-leaving-content)

This is the opposite of the previous example, where the tags around an element are removed but the content remains.

Example 11. Removal of a <span> element leaving the text

```
<text:p> This text will be made
  <delta:remove-leaving-content-start delta:removal-change-idref='ct345'
     delta:end-element-idref='ee888'>
   <text:span text:style-name="bold-style" />
  </delta:remove-leaving-content-start>
unbold
  <delta:remove-leaving-content-end delta:end-element-id='ee888'/>
. </text:p>
```

Since the element has been deleted, but the content remains, it is split it into a start and end element so that the content remains in position at the correct level. The split element is linked by an ID so that it can be reconstructed. The splitting of a wrapper element into its start element and end element means that deleted wrapper elements do not contribute to the hierarchical structure of the document. This is important because over time they may be split across element boundaries. When created, the start and end elements must have the same parent element.

6.3. Split an element into two elements (split)

The classic example of this is when a paragraph is split into two by the insertion of a new line. Similarly, a list item might be split into two list items. The element that is split is known as the parent of the split and the element that is created is known as the child of the split.

Example 12. Two text:p elements formed from splitting a single text:p element

```
<text:p split:split01='sp1'>
This paragraph will be split into two.
</text:p>
<text:p delta:split-id='sp1'
  delta:insertion-type='split' delta:insertion-change-idref='ct1' >
This will be in the second paragraph.
</text:p>
```

This facility allows the representation of quite a common editorial action. Note that there may be elements between the split paragraph elements but these would all have been added in the same or a later CT. Therefore there is an attribute value pair to link the start and end of a split.

6.4. Merge two sibling elements into one (merge)

This merge change is the opposite to a split, and is used to capture a number of common editing operations, for example moving the cursor to the start of a paragraph and pressing the delete key to join it to the previous paragraph. This would create a single paragraph as shown in Example 13

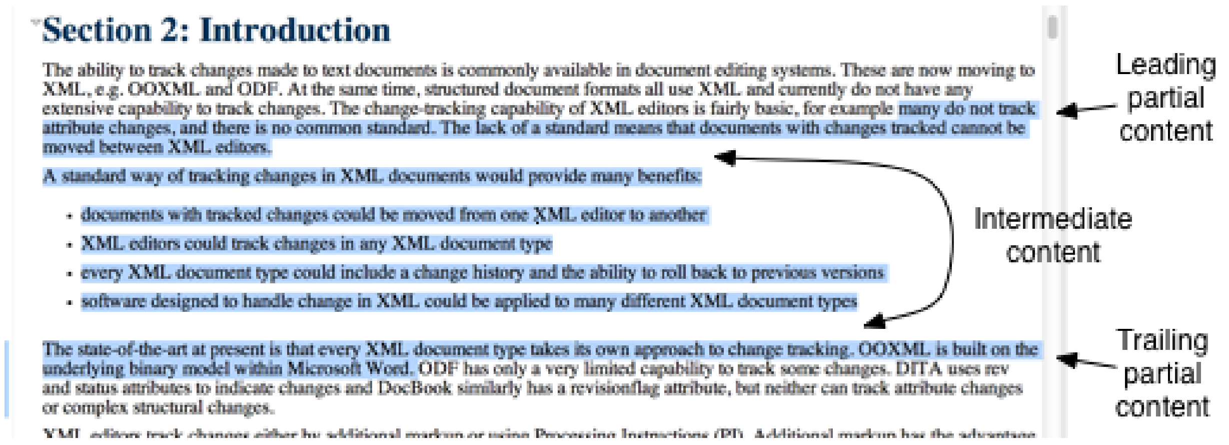
Example 13. A single text:p element formed from merging two text:p elements

```
<text:p text:style-name="Standard">
These paragraphs will be merged into one.
 <delta:merge delta:removal-change-idref='ct2'>
   <delta:leading-partial-content/>
   <delta:intermediate-content/>
   <delta:trailing-partial-content>
```

```
        <text:p text:style-name="Code"/>
      </delta:trailing-partial-content>
    </delta:merge>
   This was in the second paragraph.
   </text:p>
```

The delta:leading-partial-content element is in this case empty, but it could contain content from the first text:p element. Similarly, the delta:intermediate-content element is also empty, but it could contain any number of elements or content that lay between the two text:p elements. The delta:trailing-partial-content always contains one element, i.e. the element that forms the end of the merge operation. This element may or may not contain content. Any content within it would have been removed from the start of the final element in the merge. The element in delta:trailing-partial-content could be of a different type to the one that encloses the delta:merge; this allows elements of different types to be merged.

The above is only a simple example, but the structure allows for more complex merge operations to be represented and to be reversible. Such a use-case is depicted in Figure 1 which is a screenshot made during the editing of this paper. The region highlighted in blue has been selected (by holding and dragging the mouse). Pressing the delete key would logically complete the merge operation and the highlighted text would be replaced by a merge element containing the three children with the content indicated.



Section 2: Introduction

The ability to track changes made to text documents is commonly available in document editing systems. These are now moving to XML, e.g. OOXML and ODF. At the same time, structured document formats all use XML and currently do not have any extensive capability to track changes. The change-tracking capability of XML editors is fairly basic, for example many do not track attribute changes, and there is no common standard. The lack of a standard means that documents with changes tracked cannot be moved between XML editors.

A standard way of tracking changes in XML documents would provide many benefits:

- documents with tracked changes could be moved from one XML editor to another
- XML editors could track changes in any XML document type
- every XML document type could include a change history and the ability to roll back to previous versions
- software designed to handle change in XML could be applied to many different XML document types

The state-of-the-art at present is that every XML document type takes its own approach to change tracking. OOXML is built on the underlying binary model within Microsoft Word. ODF has only a very limited capability to track some changes. DITA uses rev and status attributes to indicate changes and DocBook similarly has a revisionflag attribute, but neither can track attribute changes or complex structural changes.

XML editors track changes either by additional markup or using Processing Instructions (PI). Additional markup has the advantage

Figure 1. Merge regions

The merge operation could be represented using remove-leaving-content and insert-around-content but this leads to a more complex structure. Therefore the merge element provides a special representation for this common editing action.

7. Integration of Change Tracking with a Host Schema

7.1. Stand-alone use

The format can be used as an independent addition to an existing XML host format. In this scenario no changes are made to the schema of the host format, but the track change elements and attributes are used to represent changes and edits to a document.

This is the simplest way to use change tracking, but of course the document cannot be validated against a schema. However, the latest version of the document can easily be extracted and checked, and each change can be individually rolled back and the resulting document checked. This could be handled using NVDL validation.

A Schematron checker can also validate the entire document with change tracking against certain rules, for example that references to change transactions are correct.

7.2. Schema integrated use

In this scenario there will be a RelaxNG schema which specifies the host format with change tracking schema integrated with it. The stand-alone testing mentioned above would still be valid and work, but as well as that the change-tracked document could be checked against a schema.

The steps outlined below show how to change a schema to allow changes to be tracked throughout the schema. By restricting the application of each individual step, it would be possible to restrict the tracking of changes to certain areas within the schema.

Integration of Level 1 is simpler than integration of Level 2.

Schema Integration Level 1

The following steps provide a way to perform the integration.

Step 1: An element containing the change meta data (change transactions and their grouping) must be allowed at one point in the document.

Step 2: Any element in the host format that has one or more attributes which can be added, deleted or values changed, need to allow attributes in the ac: namespace.

Step 3: All elements that can be added or deleted with their content (including any element that allows no content, i.e. is always empty) need to allow the attribute delta:insertion-type with value 'insert-with-content' and be permitted as a child of delta:removed-content. Note that this is not necessarily all elements, for example an element that is only used as a required item and never in a choice would not be in this category. Some modification to choice element structure may be needed to allow changes.

Step 4: All elements that allow element content must have their content model modified so that they allow delta:removed-content to appear anywhere as a child element.

Step 5: All elements that allow PCDATA content, including elements that allow mixed content, need to allow for PCDATA content to be added (Step 4 allows text to be deleted).

Schema Integration Level 2

Step 1, 2, 3 and 5 are as Level 1. Step 4 is replaced with Step 8.

Step 6: Any element that can be added as a wrapper around existing content, or removed as a wrapper (in this situation it is often true that the content model of the element is a subset of the content model of its parent): These elements need to allow delta:insertion-type='insert-around-content' and supporting attributes.

Step 7: For any Step 6 element, if any content is required then the content model must be changed to make an empty element (no content) allowed when its parent is delta:remove-leaving-content-start.

Step 8: This is an extension to Step 4: Any element that allows content must allow as child elements delta:remove-leaving-content-start, delta:remove-leaving-content-end, delta:removed-content and delta:merge to appear zero or more times anywhere.

Step 9: Elements where it is useful to represent a split or merge. Typically these will have mixed content, though this is not a condition. These elements need to allow delta:insertion-type='split' (and supporting attributes).

8. Conclusions

The description provided in this paper gives a flavour of how change tracking could be achieved generically for any XML format. Initial implementation and validation of this has been performed for the ODF format, but it has not been tested on other XML formats. The processing instruction representation has also not been worked through in detail, nor implemented.

One area of change tracking that has not yet been considered is the representation of conflicting changes. These might result from real-time collaboration or the merging of concurrent edits.

This initial work will be submitted to the W3C community group for their consideration and potential development into a recommendation. As outlined above, there are potentially significant gains for the XML community in having a robust and generic standard in this area, and it is hoped that the new community group will be able to achieve this.

Bibliography

[1] Robin La Fontaine: XML Change Tracking ODF Proposal submission http://www.deltaxml.com/attachment/481-dxml/XML-change-tracking.pdf

[2] Open Document Format for Office Applications (OpenDocument) Version 1.2 29 September 2011, OASIS Standard. http://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2.pdf

Local Knowledge for *In Situ* Services

R. Alexander Milowski
ILCC, School of Informatics, University of Edinburgh
`<alex@milowski.com>`

Henry S. Thompson
ILCC, School of Informatics, University of Edinburgh
`<ht@inf.ed.ac.uk>`

Abstract

Many Semantic Web applications rely on the idea that all data is globally known, readily accessible, encoded using some RDF mechanism, and translatable into graph structures. This model ignores the reality of local knowledge and data processing by local services (e.g. Web applications). This paper explores the usefulness of encoding local knowledge with RDFa for Web applications and demonstrates local services via an implementation of the RDFa API.

1. Linked Data or Not?

On the Web, there are many definitions of term "linked data" [1]. On one popular website (linkeddata.org), the website opens with:

> *Linked Data is about using the web to connect related data that wasn't previously linked, ...*

and in [Linked Data: Evolving the Web into a Global Data Space] [2], the authors state that RDF links things, not just documents, and that RDF links are typed. The authors assert that, in contrast to the HTML documents on the Web, RDF data alone links between objects' representations using predicates.

Both of these viewpoints seem to ignore the irrefutable fact that non-RDF documents containing data do exist on the Web and they do link to each other. In fact, these links, their link relations, and their associated media types and representations returned via retrieval are sufficient for a lot of data to be processed on the Web. The success of search companies in their ability to get structured information about specific entities or places shows that data is already out there and linked by regular HTML documents.

To be fair, the proponents of Linked Data, as described above, want more explicit links at a finer granularity. Rather than having to put large search engine machinery in place to discern relationships between entities or determine their facets, these advocates are pushing for data to be more explicitly annotated. With explicit typing of these links, the relationships are easily detectable and transferable off the Web into graph-oriented computation engines and databases.

While a matter of opinion, time has so far shown that the average publisher of documents or services on the Web will not annotate their data unless there is both a direct benefit (the selfish principle) and that doing so is a proportionally easy task relative to the benefit. In relation to this observation, many consumer services have been able to expand their market penetration by providing a useful API that provides individuals access to their data. This allows the individual developer to create their own derivatives for their own purposes which, in turn, often provide useful services other than those provided by the original offering.

The point of these services accessible via an API is to enable developers to create new derivative products that they will then share or integrate into their own offerings. The hope for the originating provider of the services is to extend their development at minimal cost while increasing their market share. The intent is rarely to publish their data in an open and consumable format that participates in a global data space.

In contrast, government and non-profit agencies sometimes have more altruistic goals for their data. Their goal, often under mandate from laws or directives, is to share their data openly. Yet, when it comes to putting this data on the Web, the common outcome is either direct access to raw data files or tightly controlled Web applications that limit the views on the data. Rarely is there an API provided for interaction with a data set as there is for consumer services on the Web.

2. The Local Knowledge Problem

The *Local Knowledge Problem* is an information problem from economics first stated by Friedrich A. Hayek. Hayek observed that certain important knowledge necessary for macroeconomics is restricted to local actors who provide this information in often unorganized and unscientific forms [3]. As such, to gain the knowledge, an information seeker must interact with the local actor on their terms. This causes problems for macroeconomic theories which rely on globally uniform access to all economically relevant information.

Similarly, when a service interacts with Web resource representations, often in HTML, the service must interact with what the publisher of those pages has provided. The publisher is the local actor and the structure of their pages, often unorganized and unscientific, is all the service has to work upon. As such, local knowledge of the page structures is essential for understanding the contained information in a larger context.

As in economics, the Local Knowledge Problem is a counter-argument to global data theories. Essentially, to build a global data space, every purveyor of local knowledge would need to agree on their terms, methods, and syntax. While information is widely available to people on the Web, systems find locating, aggregating, and disambiguating information difficult due to wide-scale lack of agreement on representations.

Fortunately, all is not lost for local knowledge on the Web. Large interests of commercial search engine providers can entice these local purveyors to publish their information in common structures using annotations such as RDFa Lite 1.1 [5]. That may provide some agreement on common entities like people, places, etc. Similarly, other industry groups may find they can influence their communities of information purveyors. In the end, this may result in "islands of interoperability" where information can be exchanged or extracted at a higher level of understanding.

3. Annotating for *In Situ* Services

Web documents are identified by URI (a name) and have resource representations that are retrieved by user agents on the Web--most commonly as HTML [13] to be rendered by Web browsers. In HTML, these documents contain references to scripts that are expected to be executed by the user agent and their execution often provides additional services to the local Web page representation (e.g. a map or other visualization widget). These scripts are packaged by service providers but they often require integration with the page that can only be done by the website purveyor. That is, the website purveyor must indicate to the service provider's script the information in the Web page to act upon and exactly how to do so.

This is mostly achieved today by the website purveyor writing specialized scripts to interface between his/her pages and the service provider-specified API. But there is a better way. Instead, the purveyor can consider the service semantics as data that they must provide and then require the service to use this data directly. For example, a Web page for a business can annotate their address and the map widget service then looks within the page for information annotated as an address. The intent is to avoid the potentially fragile method of the purveyor having to "program" the integration of the document and the service provider's API.

The way this is accomplished is by annotations within the HTML document. In the past, services have relied on different methodologies (e.g. class or id attributes, embedded script data, etc.) but now these annotations can be encoded as RDFa annotations of types and properties [4]. This method uses regular HTML elements and a few standard attributes that can be added to any element (i.e. `typeof`, `property`, `resource`) to encode the additional information.

Often, the website purveyor and the author are different individuals. The author may not have the permissions to invoke any form of scripting. As such, all the author can do is provide information upon which the infrastructure provided by the website purveyor will act. It is in this role that the author can enhance the information they are providing to enable services that will enhance the reader's experience.

For example, an author may want to provide more information about an individual they reference on a Web page. Certainly the author can provide a sidebar or footnote detailing this information. If they do so, this information can be encoded via the types available at `schema.org` [10]. An example of such information encoded in RDFa with its Turtle output graph [7] is shown in Example 1.

Example 1. A Person Encoded in RDFa

```
<div vocab="http://schema.org/" typeof="Person">
  <span property="name">
     <span property="givenName">Alex</span>
     <span property="familyName">Milowski</span>
  </span>
  <span property="email">alex@milowski.com</span>
  <span property="organization" typeof="Organization">
     <span property="name">ILCC, School of Informatics,
         University of Edinburgh</span>
     <span property="url">
        http://www.inf.ed.ac.uk/research/ilcc/</span>
  </span>
  <span property="url">http://www.milowski.com/</span>
</div>

@prefix s: <http://schema.org/>
<_:1> rdf:type s:Person ;
  s:name "Alex Milowski";
  s:givenName "Alex";
  s:familyName "Milowski";
  s:email "alex@milowski.com";
  s:organization <_:2>;
  s:url "http://www.milowski.com/" .
<_:2> rdf:type s:Organization
  s:name> "ILCC, School of ..." ;
  s:url "http://www.inf.ed.ac.uk/..." .
```

Two important things should be noted about Example 1. First, only the structure of information is shown in the example. An author may choose to layout the information in HTML differently for visual presentation or include other text that is not annotated by RDFa. That flexibility allows the author to control how the information is presented without being restricted by the type structure.

Second, there is no resource identified by the encoded information. That is, the author has made no assumption about the preferred URI used by the person to represent themselves. Instead, they have just made assertions about the relationships between the “facts” presented on the page. In RDFa terms, the result is that a blank node is generated and the graph uses a generated URI.

In this scenario, the author is a third party making assertions about the person they are referencing. The author shouldn't invent a URI for this person because the author is not, generally, a definitive source of information about the person they are referencing. Inventing a URI, especially a publically accessible URI, might cause others to mistake that URI for a location on the Web at which they should be able to retrieve a representation and triples associated with the referenced person.

This still leaves the practical problem of how an author refers to content about a person encoded locally on the page. Our answer is direct and simple: use the local knowledge (e.g. a name or email) as a reference key. That is, the author can pick some property that matches between the reference and the full person information. As long as that information is both locally unique and matches, the reference can be matched by some *in situ* service provided by the website purveyor. The consequence is that the relationship between the reference and the referred is implied and not represented in the graph.

As there is no type in the `schema.org` type hierarchy for references to a person, we'll extend the type using the recommended method of a path structure (a slash and name) to denote a more specific type [11]. The encoded information will use the same properties of the `Person` type and a simple example is shown in Example 2.

Example 2. Person Reference Example

```
<p>An author is
  <span typeof="Person/Ref">
    <span property="name">Alex Milowski</span>
  </span>.
</p>

@prefix s: <http://schema.org/>
<_:1> rdf:type s:Person/Ref ;
  s:name "Alex Milowski" .
```

It may be the case that an author wants to reference a person by a more reliable property (e.g. e-mail) that should not be shown to the reader. In this case, the `content` attribute can be used to encode the e-mail. Use of this method is shown in Example 3.

Example 3. Person Reference by Facet

```
<p>An author is
  <span typeof="Person/Ref"
        property="email" content="alex@milowski.com">
     Alex Milowski</span>
</p>

@prefix s: <http://schema.org/>
```

```
<_:1> rdf:type s:Person/Ref ;
  s:email "alex@milowski.com" .
```

Again, it should be noted that the markup and content structure is mostly under the control of the author. They only have to add the RDFa annotations to the markup surrounding the reference to the author. As such, regardless of the display of the reference, the graph structure will be equivalent.

4. Implementing Local Services

The impetus for the author to encode local knowledge goes beyond better search. Authors and website purveyors provide local services to readers to engage them beyond just reading. That is, they want to enhance the user experience with useful interactive services.

In the case of the person example, a website purveyor may want all local knowledge of people accessible to the reader when the person is referenced. When the reader encounters a reference, they may want to follow associated links or search for the user on various social networks. This can be implemented as a local service that provides the reader with the ability to find out more information about the person at the reference by displaying a menu of data and actions (e.g. search on the Web). As such, the website purveyor engages an *in situ* service, which uses the RDFa annotations to augment the user experience, without the website purveyor having to do much more than include a single script.

The onus to use local information is on the service provider and not on the website purveyor or author. Fortunately, for the service provider there is an API available for RDFa [6] that simplifies processing and using the RDFa annotations in the Web browser. As such, all the service developer must do is provide a single script that will use this API to find relevant local information and act upon it as appropriate.

In the case of the person reference example, the script must find each Person-reference in the current Web page, locate the referenced Person node in the RDFa-derived graph, and use the properties of that node to augment the user interface (UI) (e.g. responding to hovering over the reference by opening a popup offering the Person's properties or offering a specialized search). This process is straightforward:

1. Locate all references to people by type.
2. For each reference, find the annotation graph node of type Person than matches the reference's properties
3. For each match, enhance the UI behavior of the reference.

The first two steps are accomplished by using the RDFa API in combination with the DOM and are shown in Example 4. The last step (3) depends on the UI behavior desired and other browser or user profiles. As such, the example focuses on how the RDFa API accomplishes the first two steps.

Example 4. Person Reference Service Steps

```
// Step 1: Locate references (DOM Elements)
var refs = document.getElementsByType(
   "s:Person/Ref"
);

// Step 2: iterate the references (DOM Elements)
for (var i=0; i<refs.length; i++) {
   // Use the subject node id of the reference
   // to get the the e-mail or name of the referenced
   // person from the graph (strings)
   var refProperty = "s:email";
   var refValues = document.data.getValues(
      refs[i].data.id,
      refProperty
   );
   if (refValues.length==0) {
      refProperty = "s:name";
      refValues = document.data.getValues(
         refs[i].data.id,
         refProperty
      );
   }
   if (refValues.length==0) continue; // No referent found.

   // Find subject node URIs in the graph that have
   // the same value for the same property as found on the
   // reference. (URI strings)
   var matchSubjects = document.data.getSubjects(
         refProperty,
         refValues[0]
   );

   // Determine which subjects are persons and not other
   // references by type (URI string)
   var personSubject = null;
   for (var j=0; j<matchSubjects.length; j++) {
      var types = document.data.getValues(
         matchSubjects[j],
         "rdf:type"
      );
      if (types.indexOf("http://schema.org/Person")
          >= 0) {
         personSubject = matchSubjects[j];
         break;
      }
   }

   // Step 3:
   if (personSubject) {
      // personSubject can be used to get more
      // information from the graph
      // refs[i] holds the DOM element for the
      // reference.
   }
}
```

5. Image Annotation

A common user interaction on the Web is to “tag” portions of an image with annotations so that when a user drags the mouse over portions of the image, the Web browser displays additional information and behaviors over specific locations. This is often used to identify people or objects in an image by highlighting a box in the image and then displaying a description for some interaction (e.g. a mouse over) while allowing a click-through to more information.

This method of adding interface semantics to an image has four major different parts:

1. The image annotation position and shape (local knowledge).
2. The annotation description (local knowledge).
3. Expected behaviors (integration - interface semantics).
4. Display style (integration - interface styling).

The first two items are the local knowledge provided by the author who has both the location of the annotation in the image and the metadata about the annotation. This metadata may contain facets such as a person's name, an address, or other descriptions but also may contain Web-oriented data such as links to other resources on the Web. This information is as essential to the page as the other text normally placed within the page.

The third and forth items have a lot to do with the design of the website or application, often are under control of the website purveyor and not the author, and directly relate to the form factor of the device receiving the Web page. As such, this information may be embedded in the annotation or may be embedded in how the application is programmed to behave. While annotations can be used to control facets like color, other Web-oriented mechanisms could easily be used for that as well (e.g. CSS) and may be more appropriate.

In contrast to the person example of the previous section, annotating an image is much more straightforward as the image is a resource that exists on the Web. As such, the image has a resource URI to which annotations can be associated. In this example, the first two parts of image annotations are easily encoded using the `schema.org` type hierarchy and two extra properties: `annotation` and `region`.

An example of such an annotation is shown in Example 5 that starts with a wrapper for the vocabulary. The first child element is a regular `img` element with an extra `typeof` attribute that indicates the image is a "creative work" in the `schema.org` type hierarchy. Note that this image reference could appear anywhere within the document but is shown next to the annotation for compactness.

The second child element `div` following the image is a container for annotations of the image and identifies the image resource being annotated by the `resource` attribute whose value must be the same URI as the image resource. An `annotation` property makes an association between the image being annotated and the annotation encoded as a `Comment` in the `schema.org` type hierarchy. This `annotation` property is an extension to `schema.org` that is not currently available.

The comment itself is the annotation and contains a number of different properties to provide short and long descriptions, links, and other possible facets from the `Comment` type. The single extension to this type is the `region` property that encodes the region of the image for placement of the annotation. The region is defined as a shape type, position, and size facets. The intended result is a rectangular annotation around the salamander in the image that associates a title (description), long description (text), and a click-through link to the Wikipedia page for the species.

Example 5. An Image Annotation

```
<div vocab="http://schema.org/">
<img src="Salamander.jpg"
      typeof="ImageObject" />

<div resource="Salamander.jpg">
<div property="annotation" typeof="Comment">
<h2 property="description">
California Slender Salamander
</h2>
<p property="text">
The California Slender Salamander is a lungless
salamander ...
</p>
<p>Link:
<a property="url"
   href="http://en.wikipedia.org/wiki/...">Further reading...</a>
</p>
<p property="region" typeof="Rectangle">
```

```
Position: <span property="positionX">84</span>,
          <span property="positionY">114</span>
Size: <span property="width">90</span> by
      <span property="height">75</span>
</p>
</div>
</div>

</div>
```

The triples, represent in Turtle notation [7], are shown in Example 6. It should be noted how the use of the resource container associates the `annotation` property with the image resource and the value is a blank node. Again, the local knowledge of the annotation is, at minimum, just a blank node with a generated URI. The comment has not been required to be promoted to a Web resource just to allow the annotation to take place.

Example 6. Image Annotation Triples

```
@prefix s: <http://schema.org/>
<Salamander.jpg> rdf:type s:ImageObject;
  s:annotation <_:1> .
<_:1> rdf:type s:Comment;
  s:description "California Slender...";
  s:text "The California Slender ...";
  s:url "http://en.wikipedia.org/wiki/...";
  s:region <_:2> .
<_:2> rdf:type s:Rectangle;
  s:positionX "84";
  s:positionY "114";
  s:width "90";
  s:height "75" .
```

Even though the image annotation is encoded in the RDFa, the image annotation will not show up on the image unless some local service acts upon the data. This is another place where the RDFa API [6] can be used by a service to find the local knowledge encoded in the document by simple iteration in JavaScript as show in Example 7.

The process is a simple navigation of the underlying triple graph produced by the RDFa processor. First, the elements that have been annotated are retrieved by type. This returns the actual `img` element that has the link to `Salamander.jpg` image and has the `typeof` attribute. With that element, the service can decide how to augment the UI and add additional behaviors for the annotation.

For each of these target elements, the subject is used to retrieve any annotations associated. This is accomplished by getting a list of the subjects who are in the `s:annotation` relationship to the image. The result is a list of subjects that can be used to get the annotation properties.

Finally, for each of these annotations, the specific annotation properties are retrieved. In Example 7, the example code retrieves the short and long descriptions of a region in the image. Other properties are retrieved similarly and the service then uses this information to produce an overlay for the image.

Example 7. Iterating Over Annotations

```
// Step 1: Get annotated images (DOM elements)
var targets =
   document.getElementsByType("s:ImageObject");

// Step 2: Get annotation subjects (URI strings)
var annotations = document.data.getValues(
   targets[0].data.id,
   "s:annotation");

// Step 3: Iterate over annotations
for (var i=0; i<annotations.length; i++) {

   // Only use annotations that are of type s:Comment
   var types = document.data.getValues(annotations[i],"rdf:type");
   if (types.indexOf("http://schema.org/Comment")>=0) {

      // Get specific property values
      var shortdesc = document.data.getValues(
         annotations[i],
         "s:description")[0];
      var longdesc = document.data.getValues(
         annotations[i],
         "s:text")[0];
      ...
   }
}
```

6. Implementation Experience

Both examples have been implemented as local services that only require inclusion of a single script reference in an HTML document. The website purveyor must both include the local service script and provide the appropriate RDFa annotations. The local services then detect whether the RDFa API is present, provide it if it is missing, and then operate on the local document to provide additional services.

These implementations rely on the RDFa API being available and browsers do not currently implement RDFa. As such, the local scripts attempt to detect whether the RDFa API is present through some other means. If the document does not contain the RDFa API, it is dynamically loaded by injecting a script. Regardless, processing resumes once the `rdfa.loaded` document event occurs, which signals that the RDFa processor has completed processing the document.

The RDFa API implementation used by the local services is the open-source Green Turtle RDFa Processor [9]. This processor is an RDFa 1.1 compliant processor that also provides the RDFa API and several useful extensions. The processor is relatively efficient and local services execute without noticable or annoying delays for the user of the website.

An example of using a local service for image annotation is shown in Figure 1. The annotation has been activated by placing the mouse (not shown) over the Salamander which shows the description text. A subsequent click would direct the browser to the Wikipedia page for the California Slender Salamander [14].

All these services are available on the Google Code website `http://code.google.com/p/rdfa-in-action/`. Examples and documentation for the use of both the scripts and RDFa annotations are provided on that website.

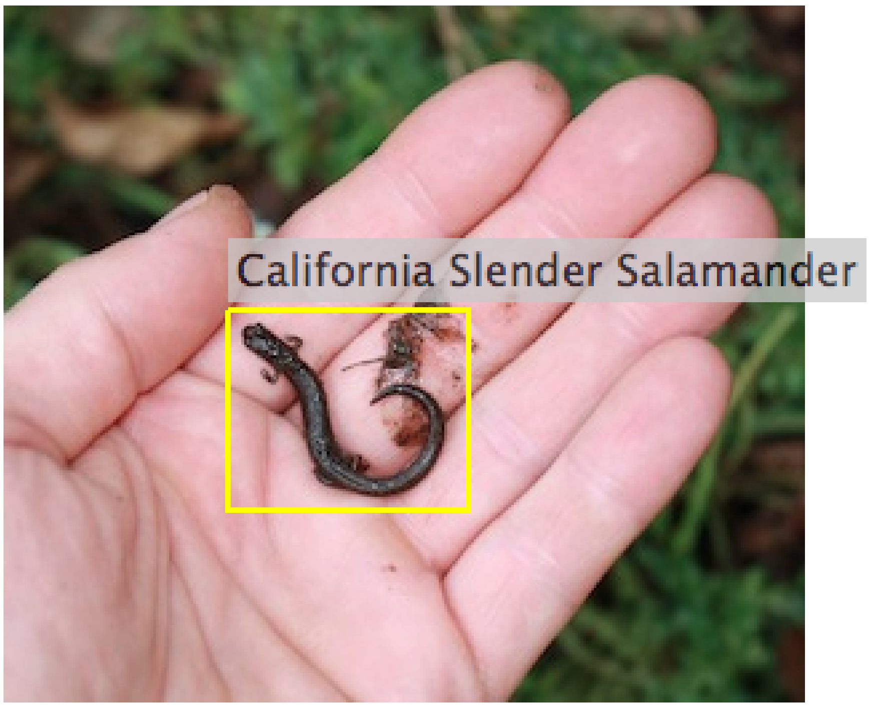


Figure 1. Image Annotation Example

7. Conclusion

The encoding of local knowledge is an altruistic goal. Authors and website purveyors need to be encouraged to spend the time to do so. While baiting them with "better search" carrots may work for some, enticing them with direct benefits to their organization and consumers will surely be more attractive. The only roadblock is well designed and solid local services that can be simply used by the website purveyor.

For this approach of local services to succeed, the essential next steps are for the `schema.org` and other common vocabularies to become both stable and complete enough to support encoding local knowledge for local services. In the case of `schema.org`, they encourage extension via properties and sub-typing. As such, the ability to fill in the gaps, as shown in both the examples of person references and image annotations, is completly within the expected rules. That allows developers to innovate, standards developers to interact with innovations, and allows the right set of properties and types to evolve along side their uses.

In the end, the most attractive benefit is the simple idea that local knowledge can be encoded and used by local services without necessarily resulting in the website author having to use and direct a lot of scripting. Instead, they declare the knowledge they have and invoke local services that will do something useful with that knowledge. Admitably, they must include scripts that implement these local services but that won't necessarily always be the case. The knowledge they encode can be used by any tool or browser extension. As such, the author has taken a step towards future-proofing their information and the services that they provide on the Web.

Bibliography

[1] Linked Data -- The Story So Far, Bizer, Christian and Heath, Tom and Berners-Lee, Tim , International Journal on Semantic Web and Information Systems vol. 5, issue 3, pp. 1-22 http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf

[2] Linked Data: Evolving the Web into a Global Data Space (1st edition), James Hendler and Frank van Harmelen , Morgan & Claypool 2011 ISBN 9781608454303 linkeddatabook.com/editions/1.0/

[3] The Use of Knowledge in Society, Friedrich A. Hayek , American Economics Review vol. XXXV, issue 4, pp. 519-30 09 1945 http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf

[4] RDFa Core 1.1, Ben Adida and Mark Birbeck and Shane McCarron and Ivan Herman, 2012-03, W3C http://www.w3.org/TR/rdfa-core/

[5] RDFa Lite 1.1, Manu Sporny, 2012-03, W3C http://www.w3.org/TR/rdfa-lite/

[6] RDFa API, Manu Sporny and Benjamin Adrian and and Nathan Rixham and Ivan Herman, 2011-04-19, W3C http://www.w3.org/TR/rdfa-api/

[7] Turtle: Terse RDF Triple Language, David Beckett and Tim Berners-Lee and Eric Prud'hommeaux and Gavin Carothers, 2012-07, W3C http://www.w3.org/TR/turtle/

[8] Deep Web, , 2012-05, Wikipedia http://en.wikipedia.org/wiki/Deep_web

[9] Green Turtle, , 2012-, R. Alexander Milowski http://code.google.com/p/green-turtle

[10] schema.org http://schema.org

[11] Extension Mechanism [schema.org] http://www.schema.org/docs/extension.html

[12] Citizen Weather Observation Program (CWOP) http://www.wxqa.com/

[13] HTML5: A vocabulary and associated APIs for HTML and XHTML, Ian Hickson, W3C, March 2012 http://www.w3.org/TR/webarch/

[14] Wikipedia: California slender salamander http://en.wikipedia.org/wiki/California_slender_salamander

Quo vadis XML?

History and possible future directions of the Extensible Markup Language

Maik Stührenberg
Bielefeld University
`<maik.stuehrenberg@uni-bielefeld.de>`

Abstract

XML has been a success story for nearly 15 years as a technical basis for a large variety of markup languages. However, during its ongoing development XML and its accompanying specifications have grown in complexity. While there are projects that try to reduce this complexity by addressing at least one of XML's building blocks, that is the linearisation, the formal model, the validation, or the differentiation between level and layer, other voices complain that some of XML's building blocks are still not complex enough. In this paper we will present the components of an XML-based markup language (including the respective limitations) and some alternative notation formats that try to address specific limitations. The goal of this paper is not to present new findings but to give a survey about the current state.

1. Introduction

XML has been a success story for nearly 15 years. The meta language has been the technical basis for a large variety of markup languages in different fields such as document management, web, multimedia and business applications. However, during the last years voices have been raised that XML has become too complex. Others argue that XML in all its complexity still lacks some features that other (newer) notation formats provide. In this paper we will discuss the components of XML-based markup languages as building bricks for XML's complexity and their respective limitations. We will afterwards present some alternative notation formats that address some of the before-mentioned limitations as possible marker for future developments. But before going into details we will start with a brief overview of XML's history.

2. XML's origin

Historical background

In 2013 XML will celebrate its 15th birthday. Since then it has become a widely adopted standard for the structured storage of data of many kinds (especially for semi-structured data). But the roots of the specification (and therefore, elementary parts of its syntax and data model) go back to the 1960s, when William Tunnicliffe of the Graphic Communications Association (GCA) presented the idea of separating content and formatting at a meeting of the Canadian Government Printing Office in September 1967. This resulted in the GenCode concept [Goldfarb 1991]. In 1969 the *Text Description Language* (TDL) was developed at IBM by Charles F. Goldfarb, Ed Mosher, and Ray Lorie, followed by the *Integrated Textual Information Management Experiment* (InTIME) prototype [Goldfarb et al. 1970]. In 1971 TDL was renamed as *Generalized Markup Language* (GML) [Goldfarb 1973], introducing a simple syntax, including angle brackets < and > as delimiter strings between text and markup, and the concept of start and end tags. In 1978 both GenCode and GML combined forces in the newly established ANSI Committee on Computer Languages for the Processing of Text which became the ISO committee that developed the Standard Generalized Markup Language (SGML) in the 1980's and which published the final version of the standard in 1986 [ISO 8879:1986], establishing the concept of generic markup:

> *The characteristics of being declarative, generic, non-proprietary, and contextual make the Standard Generalized Markup Language "standard" and "generalized".*
>
> *—[Maler and El Andaloussi 1995, p.12]*

However, SGML brought a bunch of optional features resulting in a huge complexity. Apart from the possibility to use another than the reference and core concrete syntax, features such as CONCUR made implementing the standard as a whole really tough. As a result, SGML's use was mainly limited to publishing houses and government-related information processing. Therefore, Sperberg-McQueen and Bray 1997 presented a proposal for an SGML subset, called Extensible Markup Language (XML).[1] This stripped-down version of SGML was initially intended to conquer to the World Wide Web [Rubinsky and Maloney 1997]. It only took until 1998 for the specification to be published as W3C recommendation. From this point onwards, XML has been the great success story that we all know today.

3. What is XML?

Building bricks of markup languages: syntax, data model, grammar formalism
When we talk about XML, we usually talk about the meta language that allows us to create markup languages. One often talks about XML with specific XML applications in mind, such as XHTML, DocBook or the TEI. These applications consist of the XML-based markup language itself and an accompanying toolset of XSLT stylesheets or similar satellite applications. But let's start with the markup language

[1]Technically speaking, XML has been a true subset only after [ISO 8879 TC 2] was published.

itself: In [Sperberg-McQueen and Huitfeldt 1999] the authors describe the three building blocks of a markup language as the linearisation, the formal model (the data structure), and a validation mechanism (via a constraint language or grammar).[2] [Huitfeldt and Sperberg-McQueen 2004] coined the metaphor of a *tripod* for these three components. Following [Sperberg-McQueen et al. 2000] as well as [Witt 2004] we will add a fourth leg to this tripod by differentiating between the markup serialization and its interpretation, resulting in the metaphor of a chair.[3] In addition, we split up the first component, linearisation, into syntax and notation, resulting in a total of four or five (depending on your counting method) components of a markup language. The differentiation between four or five components is dependant on the meta language which is used to define a markup language firsthand.

3.1. Linearisation

3.1.1. Syntax

XML-based markup languages inherit their notation format (the syntax) from their meta language. XML, as successor and real subset of SGML [ISO 8879 TC 2 p. 111], uses the same reference concrete syntax for XML instances as its ancestor. An XML document (that is, an XML instance) is physically composed of “storage units called entities which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data and some of which form markup” [Bray et al. 2008, Section “Introduction”]. While the document entity serves as the root of the entity tree, the elements are of special interest, since “[e]ach XML document contains one or more elements, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its generic identifier (GI)” [Graham and Quin 1999 p. 161-162]. A start tag begins with the delimiter string opening angle bracket <, followed by the GI and the delimiter string closing angle bracket >. The end tag is constructed similar with the only difference that after the opening angle bracket the forward slash has to be written. The data between start and end tag is called the element content that can be either text, other elements or a mixture of both. Empty elements (that are elements without content) can be represented either via a start tag immediately followed by an end tag, or via an empty element tag. The differences between XML and its ancestor SGML are quite small: Apart from being case-sensitive, most of the latter's minimization features have been stripped in XML. While SGML allows for example that either the start tag or

[2]Although the authors describe these three components as fundamentals of an SGML based markup language, it is valid for XML based markup languages as well.
[3]We have chosen this metaphor since the last component can be seen as optional - and there are both three-legged and four-legged chairs.

the end tag (or both) could be declared optional in the document grammar, there is only a single possible minimization left in XML which corresponds to attributes declared with default values in the document grammar. It is not only possible to use the same element type more than once but also to nest elements into each other - as long as they nest properly, since overlapping start and end tags are not allowed.

Attribute specifications can occur in the start tag (and only in the start tag) of an element and consist of the attribute name and its value (enclosed in single or double quotes, see [Graham and Quin 1999, p. 166-167]). Each attribute name may only appear once in a start tag (although it is allowed to use the same attribute multiple times in different tags, because attributes are declared locally). The value of an attribute may not contain the opening angle bracket character. Attributes can neither contain other attributes nor elements, however, they can be of different types: "a string type, a set of tokenized types, and enumerated types" [Graham and Quin 1999 p. 187]. All of these types have to be declared in the attribute list declaration in the document grammar, but two tokenized types are of special interest: `ID` and `IDREF`. The value of an `ID` token type attribute "must uniquely identify the elements which bear them" [Graham and Quin 1999 p. 187]. Values of an `IDREF` token type attribute in contrast "must match the value of some `ID` attribute" [Graham and Quin 1999 p. 190]. That means that a pair of elements having an `ID` (respective `IDREF`) type attribute can be connected via their attributes' values (e. g. to link a footnote text to the corresponding footnote marker in the running text). There is an additional `IDREFS` tokenized type that allows to refer to one or more values of attributes of the `ID` type, thus establishing not only 1:1 (one-to-one) but 1:n (one-to-many) links. Both, the nesting of elements and the use of attributes supporting integrity features have direct consequences for the formal model of XML instances discussed in Section 3.2.

3.1.2. Notation

Typically, XML instances contain markup that is wrapped around the data that is annotated (the element content as described in the previous section). This inline annotation (or embedded markup) has many advantages: Both, annotation and corresponding text can be read as a single unit and changes to the primary data have little or no effect on the surrounding start and end tags at all. In addition, virtually every XML-processing software is capable of dealing with inline annotated data.

In contrast to this, standoff notation ("remote markup" called in [Corpus Encoding Standard]) separates the markup from the primary data [Larson et al. 2007], either outside of the text or in separate files.

The concept of standoff notation was first mentioned in documents describing the second phase of the TIPSTER project, TIPSTER II: [TIPSTER 1996 p. 19] "Markups may be embedded within the document or they may co-exist with it in the form of annotations, possibly containing pointers to specific locations in the document." [Thompson and McKelvie, 1997] describe three reasons for using standoff annotation:

1. The base material may be read-only and/or very large, so copying it only to introduce markup may be unacceptable;
2. The markup may involve multiple overlapping hierarchies;
3. Distribution of the base document may be controlled, but the markup is intended to be freely available.

Other reasons are technical issues such as non-digital primary data or non-textual primary data. Especially for linguistic resource it is a crucial point to leave the primary data untouched. Apart from that there is another aspect which can be used in favor for standoff markup and which was brought up by [Nelson 1997]: Inline annotation "limits the kinds of structure that can be expressed". We will talk about this aspect in the following section.

3.2. Formal Model

Both, SGML and XML, were developed with the task of structuring textual documents in mind. As stated by [Coombs et al. 1987] "[d]ocuments have a natural hierarchical structure: chapters have sections, sections have subsections, and so on, until one reaches sentences, words, and letters." This statement grounded the well-known OHCO thesis that text can be considered as an ordered hierarchy of content objects (OHCO), postulated by [DeRose et al. 1990] and refined later by [Renear et al. 1996]. Such an ordered hierarchy can be represented by a tree and therefore the formal model of an XML instance is considered to be a tree since elements have to nest properly. This model is widely accepted by authors such as [Ng 2002] or [Liu et al. 2003]. [Klarlund et al., 2003] proposes the following definition of an XML Tree (we have added the restriction that the order of the nodes should be strict):

An XML tree $t = (N, E, <, \lambda, v)$ consists of a directed tree (N, E), where N is the set of nodes, $E \subseteq N \times N$ is the set of edges, $<$ is a [strict] total order on N, $\lambda : N \rightarrow \Sigma$ is a partial labeling function, and $v : N \rightarrow D$ is a partial value function; moreover, the order must be depth-first: whenever x is an ancestor of y, it holds that $x < y$.

However, the tree model has some limitations when compared to the more general graph model: A tree has only a single root node, and crossing arcs and cyclic paths are not allowed. Since [Renear et al. 1996] already stated that text is in fact not an ordered hierarchy (and therefore not an ordered, rooted tree) by providing counter-examples to the then-called OHCO-1 thesis, we may challenge the formal model of an XML tree, following [Jettka and Stührenberg 2011]. Postulating the more general formal model of a graph provides some serious advantages when dealing with multiple or overlapping hierarchies.[4] Especially the combination of standoff notation and a graph-based formal model allows for heavily annotated XML instances. Examples can be found in [Stührenberg and Jettka, 2009] as well as in newer annotation standards such as [ISO 24612:2012]. A crucial point in serializing graphs in XML is the support for integrity features such as the already mentioned `ID`/`IDREF`/`IDREFS` token type attributes (see Section 3.1.1) or the corresponding `xs:ID`/`xs:IDREF`/`xs:IDREFS` data types or `xs:key` in XML Schema. Therefore, we will discuss the topic of validation next.[5]

3.3. Validation

An XML markup language is formally defined by means of a schema which itself is defined via a constraint language. Constraint languages for XML can be divided into two classes: grammar-based and rule-based languages. A definition for both classes can be found in [van der Vlist 2003] and [van der Vlist 2004] as well as in [Costello and Simmons 2008]:

> *A grammar-based schema language specifies the structure and contents of elements and attributes in an XML instance document. For example, a grammar-based schema language can specify the presence and order of elements in an XML instance document, the number of occurrences of each element, and the contents and datatype of each element and attribute. A rule-based schema language specifies the relationships that must hold between the elements and attributes in an XML instance document. For example, a rule-based schema language can specify that the value of certain elements must conform to a rule or algorithm.*
>
> *—[Costello and Simmons 2008]*

Following this definition, the three-most used document schema languages DTD, XSD, and RELAX NG are grammar-based constraint languages, while DSD [Møller 2002] and Schematron are rule-based constraint languages.[6]

While all grammar-based constraint languages support the validation of the structure of elements and attributes and the order of elements, some lack in data typing or integrity checking (for a comparison see [Møller and Schwartzbach 2006], amongst others). Apart from these differences regarding technical features (such as XML namespace support), schema languages can be classified according to their

[4]Overlapping hierarchies have often been treated as an esoteric phenomena. However, recent projects such the one discussed by [Marcoux et al. 2012] show a larger number of occurrences of overlapping hierarchies in natural languages.
[5]The XML Infoset specification contains the phrase that "there is no requirement that the XML Information Set be made available through a tree structure" [Cowan and Tobin 2004, Section 1, "Introduction"], however, the pre-dominant data model for XML instances is still considered to be at least tree-like (see the Section "Terminology" in [Berglund et al. 2010] for example).
[6]Readers familiar in the XML world may be aware that the schema languages named here reflect only a small part of the entirety of XML schema languages that have been developed until today.

expressive power by using a taxonomy (see [Murata et al. 2005] and [Stührenberg and Wurm, 2010]). The current version of XML Schema, 1.1, supports Schematron-like assertions, introducing a feature that was formerly reserved for rule-based constraint languages. However, compared to RELAX NG, XML Schema still limits the expressivity of content models via its *Unique Particle Attribution*, which itself can be traced back to the times of SGML when deterministic finite automata where used for context-checking of content models (see [Stührenberg 2012 and Stührenberg 2012a] for a discussion).

Regarding validation, a quite drastic change was introduced by XML: draconian error handling. This means, that a parser that processes an XML instance as part of the process of validating it (or just check for well-formedness) immediately gives up if it encounters an error.[7] This error handling makes it quite simple to implement XML parsers (both validating and non-validating), since XML's formal model (see Section 3.2) is based on relatively simple grammatical models (take the requirement for deterministic content models in XML DTD and XSD for example). However, on the other side, the author of XML-encoded content bears the full responsibility to ensure that his content is at least well-formed.

3.4. Distinguishing levels and layers

[Witt 2004] introduced the separation of the concept (the level of description) and its serialization (the layer). Differentiating between level and layer is usually not necessary when handling single (that is, inline) annotations where the concept and the notation correlate in a 1:1 relation. However, other relations are possible as well, such as 1:n, n:1, or n:m. Markup languages that support this differentiation can serialize a specific level (e.g. logical document structure) by using different serialization formats (that is, tag sets). On the other hand the differentiation between concept and notation opens up a whole new possibility to add semantics to a markup language. For example, the Graph Annotation Format (GrAF) defined in [ISO 24612:2012] combines a generic annotation approach by using feature structures similar to the ones defined in the TEI Guidelines (Chapter 18 of [Burnard and Baumann 2012]) and the international standard [ISO 24610:2006] with data categories that are stored in a data category registry according to the international standard [ISO 12620:2009].[8] Other examples of a level-layer distinction can be found in the feature structure serialization format that is part of the TEI and is standardized as two-part international standard [ISO 24610-1:2006] and [ISO 24610-2:2011].

[7]A decision has to be made between (ordinary) errors and fatal errors. While the latter result in the cancellation of the parsing process, the handling of non-fatal errors is dependent on the parser used (conforming software may report the error but may recover from it) [Bray et al. 2008, Section 1.2].
[8]For an implementation of [ISO 12620:2009] see ISOcat at http://www.isocat.org.

3.5. What should not be forgotten - XML's satellite standards

XML-based markup languages can exist on their own. Well-formed instances can be created by only using the syntax defined in the XML spec alone, without any need for a document grammar (and therefore without any other formal model than a tree and without any differentiation between level and layer). However, usually a markup language is at least formally defined by a document grammar which in turn is defined by a constraint language itself. In addition, a successful markup language often is accompanied by a toolkit consisting of XSLT stylesheets for transforming encoded content in other serialisation formats (such as TXT, (X)HTML, or PDF), XQuery scripts and other software. And this is one reason of XML's success: XML does not stand on its own but is accompanied by a huge family of specifications that deal with different aspects of markup-encoded information. Amongst these XSLT, XPath and XQuery are the most prominent representatives. However, these satellite specifications add up to the whole bunch of complexity of XML. Therefore, we will discuss some alternative notation formats in the next section.

4. XML compared to alternative meta languages and notation formats

Although the XML world as such is quite complex, XML has its advantages over other notation formats and meta languages which have added to its tremendous success. One of these advantages is, that XML-based markup languages can be designed to be not only machine-processable but human-readable, too, which makes it a lot easier for human readers to understand structured information. Another big advantage of SGML and XML is the possibility to include mixed content models. Since mixed content is a natural way of describing textual information found in typical document structures, this can be seen as one of the major reasons for XML's success.

On the other hand we have limitations that are inherent to XML-based markup languages: XML's data model is often interpreted as that of a tree (see Section 3.2), and although this does not have to be the case if a document grammar with certain features is involved, it is far from satisfactory for all possible applications. Especially when compared to newer notation formats such as the JavaScript Object Notation (JSON, [Crockford 2006]), XML's at least seems to be less expressive.[9] Although JSON itself is not a meta language, its simple notation format and type system were reason enough for a very quick adoption in web applications. As a result, new meta languages have been developed, that use either a different syntax and/or a different

[9]We have to point to XML-based markup and meta languages that support more formally expressive data models, such as GODDAG-like structures (see [Sperberg-McQueen and Huitfeldt 2004] for further information). An example for such an XML-based meta language would be XStandoff discussed in Section 4.1.2.

data model. We will therefore split the overview in two sections: alternatives that stick to the XML syntax and those that choose an alternative one.[10]

4.1. Alternative meta languages that use XML's syntax

4.1.1. MiroXML

In December 2010, James Clark made a blog post about XML's future [Clark 2010]. Based on the discussions on the XML-dev mailing list, there were three possible directions further developments could take:

1. XML 2.0, a new version of XML with a high-degree of backward compatibility;
2. XML.next, a replacement for XML, featuring a richer feature set but lacking backward compatibility;
3. MicroXML, a subset of XML 1.0, not meant as a full replacement for XML 1.0 but as a stripped-down alternative solution for those applications which do not need full XML.

In his proposal he addresses the third approach to tackle some of XML's limitations since both XML 2.0 and XML.next would have to cope with serious development problems (the first due to XML's already inherent complexity, the second due to the size of such a project). In 2012 several articles about MicroXML were published ([Kattau 2012, Ogbuji 2012, Ogbuji 2012a, and Cowan 2012]). In addition, first parsers such as MicroLark by John Cowan [Ogbuji 2012a] were developed and a W3C Community Group[11] was established. The current MicroXML draft [Clark and Cowan 2012] was published in October 2012.

MicroXML provides both a stripped-down data model and a much simpler syntax (compared to XML), and does not imply XML's draconian error handling (see [Ogbuji 2012]). The data model contains only three primitive types (character, list, and maps). It prohibits processing instructions, the XML declaration, entities others than the five built-in entities, the doctype declaration, CDATA sections, and encodings others than UTF-8, amongst other features that are present in full XML. Whitespace does not affect the data model at all and is permitted for increased readability (especially for human readers). In total, the current draft specification takes only seven pages compared to the 32 pages of XML's current recommendation (5th edition).

[10]The discussion of formats in the following sections is by no means exhaustive. Other formats such as JXON [Lee 2011] try to build a bridge between the XML and JSON worlds as well; XML5 [van Kesteren 2012], proposed at XML Prague 2012, addresses some issues regarding XML parsers. However, we have concentrated on the following approaches since these address specific limitations of XML's building blocks.

[11]See http://www.w3.org/community/microxml/ for further details.

4.1.2. XStandoff

XStandoff is the successor of the Sekimo Generic Format (SGF, see [Stührenberg and Goecke 2008]). As such it is a meta markup language that uses a generic annotation format and XML namespaces in combination with a standoff approach to support overlapping hierarchies. XStandoff's data model is that of a GODDAG (general ordered-descendant directed acyclic graph) and supports discontinuous elements, multiple parenthood and differentiation between dominance and containment (see [Stührenberg and Jettka, 2009] and [Stührenberg and Wurm, 2010] for further details) as well as a differentiation between level and layer (see Section 3.4).

An XStandoff instance contains at least the root element (either `corpusData` or `corpus`) and a `primaryData` child element containing either the unannotated string content of the primary data file or the reference to a external file containing it. Further elements underneath the root element are the `segmentation` and `annotation` elements. The former stores virtual ranges over the primary data (using either character positions, byte positions or similar schemas) by `segment` child elements while the latter consists of converted annotations over the primary data.

Example 1 and Example 2 show two possible annotations of a given dialog between Peter and Paul. Since Peter's second sentence is ended by Paul, we have overlapping markup if we try to annotate both utterance and sentence structure. XStandoff uses a standoff approach by instantiating segments that are referenced by the converted annotation of the primary data (the automatic conversion process contains the transformation of all elements containing textual data to empty elements or those with a mixed content model to element only content models). Example 3 shows the converted XStandoff instance.

Example 1. Inline annotation of utterances

```
<?xml version="1.0" encoding="UTF-8"?>
<div xmlns="http://www.xstandoff.net/peter/dialog"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.xstandoff.net/peter/dialog ../xsd/▶
peter_dialog.xsd" type="dialog" org="uniform">
  <u who="Peter">Hey Paul! Would you give me</u>
  <u who="Paul">the hammer?</u>
</div>
```

Example 2. Inline annotation of sentence structure

```
<?xml version="1.0" encoding="UTF-8"?>
<text xmlns="http://www.xstandoff.net/peter/text"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.xstandoff.net/peter/text ../xsd/peter_text.xsd">
  <s>Hey Paul!</s>
  <s>Would you give me the hammer?</s>
</text>
```

Example 3. The resulting XStandoff instance

```
<?xml version="1.0" encoding="UTF-8"?>
<xsf:corpusData xmlns:xsf="http://www.xstandoff.net/2009/xstandoff/1.1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.xstandoff.net/2009/xstandoff/1.1"
  xsi:schemaLocation="http://www.xstandoff.net/2009/xstandoff/1.1
  http://www.xstandoff.net/2009/xstandoff/1.1/xsf.xsd" xsfVersion="1.1"
  xml:id="peter_dialog">
  <xsf:primaryData start="0" end="39">
    <primaryDataRef uri="../pd/peter.txt"/>
  </xsf:primaryData>
  <xsf:segmentation>
    <xsf:segment xml:id="seg1" type="char" start="0" end="39"/>
    <xsf:segment xml:id="seg2" type="char" start="0" end="27"/>
    <xsf:segment xml:id="seg3" type="char" start="28" end="39"/>
    <xsf:segment xml:id="seg4" type="char" start="0" end="9"/>
    <xsf:segment xml:id="seg5" type="char" start="10" end="39"/>
  </xsf:segmentation>
  <xsf:annotation>
    <xsf:level xml:id="dialog">
      <xsf:layer xmlns:dialog="http://www.xstandoff.net/peter/dialog" ▶
priority="0"
        xsi:schemaLocation="http://www.xstandoff.net/peter/dialog ../xsd/▶
peter_dialog.xsd">
        <dialog:div xsf:segment="seg1" type="dialog" org="uniform">
          <dialog:u xsf:segment="seg2" who="Peter"/>
          <dialog:u xsf:segment="seg3" who="Paul"/>
        </dialog:div>
      </xsf:layer>
    </xsf:level>
    <xsf:level xml:id="text">
      <xsf:layer xmlns:text="http://www.xstandoff.net/peter/text" priority="0"
        xsi:schemaLocation="http://www.xstandoff.net/peter/text ../xsd/▶
peter_text.xsd">
        <text:text xsf:segment="seg1">
          <text:s xsf:segment="seg4"/>
          <text:s xsf:segment="seg5"/>
        </text:text>
      </xsf:layer>
    </xsf:level>
  </xsf:annotation>
</xsf:corpusData>
```

XStandoff instances are created by using a set of XSLT stylesheets[12] for converting inline annotations (such as the ones given in Example 1 and Example 2). Similar to LMNL (see Section 4.2.2), XStandoff can be used for visualizing overlapping hierarchies (the XStandoff Toolkit contains XSLT stylesheets for creating both 2D and 3D visualizations of XStandoff instances, see [Jettka and Stührenberg 2011]).

4.2. Alternative notation formats

4.2.1. FtanML

FtanML [Kay 2012], named after the Swiss village where it was created, is a alternative notation format that differs in both the syntax and the data model from both XML and JSON (although it combines components of both). It has a simple but powerful grammar and data model, consisting of values, strings, numbers, arrays, elements, attributes and content. Compared to JSON, it adds elements, attributes and content, but lacks the more generic JSON object (which in turn is equivalent to FtanML elements). A value is either an array, an element, a string, a number, a boolean (these are depicted by the string constants 'true' and 'false') or 'null'. Arrays are very JSON-like while elements are quite similar to XML elements (see Example 4 and Example 5).

Example 4. JSON-like array[13]

```
[1, 2, "abc", [1, 2]]
```

Example 5. XML-like element[14]

```
<para|Here is some <b|bold> text>
```

In Example 5 the string 'para' depicts the name of the element (the *generic identifier* in XML terms). The example element and its content can be rewritten as `<name="para" content=["Here is some", <name="b" content="["bold"]>, "text">`, or `<para content=["Here is some", <b content="["bold"]>, "text">` since both the `name` and the `content` attribute are treated specially in the surface syntax, but not in the data model [Kay 2012]. In the short form given in Example 5, the string left of the pipe character `|` is equivalent to an XML element's start tag, the part right of it is the (optional) content of this very element. The end tag of an element is reduced to the right angle bracket character. FtanML elements can exist without a name (since element names and attributes are optional in the data model). An anonymous empty element therefore is notated as <>.

[12]See http://www.xstandoff.net for further details about the XStandoff Toolkit.
[13]Example taken from [Kay 2012].
[14]Example taken from [Kay 2012].

One of the benefits of FtanML compared to JSON is the addition of XML's mixed content, which makes it suitable for typical textual document content. Attributes can contain not only strings (like in XML DTDs - we subsume the tokenized types and enumerations here, since these are based on strings, too), but numbers, booleans, arrays - and other elements as well. Much of XML's syntactic clutter such as processing instructions, entity references, and CDATA sections have been stripped from both the syntax and data model.

Up to now, FtanML is in the state of a proposal (although there are a parser for FtanML and serialization of the object model to FtanML, JSON and XML). But it presents some interesting ideas about addressing XML's limitations. It is not clear whether FtanML supports overlapping hierarchies and it currently lacks both a traversal/query language and a validation language.

4.2.2. LMNL

LMNL (Layered Markup and Annotation Language, see [Tennision 2002], [Piez 2004], [Cowan et al. 2006], and [Piez 2012]) addresses both the syntax and the formal model of XML. It is based on the Range Algebra developed by [Nicol 2002] and [Nicol 2002a] and supports free form markup including overlapping ranges (note, that LMNL does not work with hierarchies but with ranges and annotations) and structured annotations: Ranges can be annotated and annotations in LMNL are comparable to attributes in XML - with the exception, that LMNL annotations can be miniature LMNL documents, including other annotations, think of XML attributes that contain elements. In addition, annotations and ranges can be anonymous (such as elements in FtanML).

While LMNL supports a built-in syntax (the so-called sawtooth syntax), alternative syntaxes can be chosen as well (for example, the XML serialization called xLMNL). Example 6 demonstrates the sawtooth syntax while Example 7 shows the same annotation using the xLMNL syntax.

Example 6. LMNL sawtooth syntax[15]

```
[excerpt [source [book}1 Kings{book] [chapter}12{chapter]]}
[verse}And he said unto them, [q}What counsel give ye that we may answer this ▶
people, who have spoken to me, saying, [q}Make the yoke which thy father did ▶
put upon us lighter?{q]{q]{verse]
[verse}And the young men that were grown up with him spake unto him, saying, ▶
[q}Thus shalt thou speak unto this people that spake unto thee, saying, [q=i}Thy ▶
father made our yoke heavy, but make thou it lighter unto us;{q=i] thus shalt ▶
thou say unto them, [q=j}My little finger shall be thicker than my father's ▶
loins.{verse]
[verse}And now whereas my father did lade you with a heavy yoke, I will add ▶
```

[15]Example taken from http://cocoon.lis.illinois.edu:8080/lis590dpl/wapiez/LMNL/lmnl/1kings12.lmnl.

```
to your yoke: my father hath chastised you with whips, but I will chastise you ▶
with scorpions.{q=j]{q]{verse]
[verse}So Jeroboam and all the people came to Rehoboam the third day, as the ▶
king had appointed, saying, [q}Come to me again the third day.{q]{verse]
{excerpt]
```

Example 7. xLMNL syntax[16]

```
<?xml version="1.0" encoding="UTF-8"?>
<x:document xmlns:x="http://lmnl-markup.org/ns/xLMNL" ID="N.d2821e1"
  base-uri="file:/homei/users/wapiez/courseweb_html/cocoon_pub/LMNL/lmnl/▶
1kings12.lmnl">
  <x:content>
    <x:span start="0" end="1" layer="N.d2821e1" ranges="R.d2821e2"> </x:span>
    <x:span start="1" end="24" layer="N.d2821e1" ranges="R.d2821e2 ▶
R.d2821e12">And he said unto
      them, </x:span>
   <x:span start="24" end="108" layer="N.d2821e1" ranges="R.d2821e2 R.d2821e12 ▶
R.d2821e15">What
      counsel give ye that we may answer this people, who have spoken to me, ▶
saying, </x:span>
    <x:span start="108" end="163" layer="N.d2821e1"
     ranges="R.d2821e2 R.d2821e12 R.d2821e15 R.d2821e18">Make the yoke which ▶
thy father did put
      upon us lighter?</x:span>
   <x:span start="163" end="164" layer="N.d2821e1" ranges="R.d2821e2"> </x:span>
    <x:span start="164" end="234" layer="N.d2821e1" ranges="R.d2821e2 ▶
R.d2821e27">And the young men
      that were grown up with him spake unto him, saying, </x:span>
   <x:span start="234" end="303" layer="N.d2821e1" ranges="R.d2821e2 R.d2821e27 ▶
R.d2821e30">Thus
      shalt thou speak unto this people that spake unto thee, saying, </x:span>
    <x:span start="303" end="368" layer="N.d2821e1"
     ranges="R.d2821e2 R.d2821e27 R.d2821e30 R.d2821e33">Thy father made our ▶
yoke heavy, but make
      thou it lighter unto us;</x:span>
   <x:span start="368" end="400" layer="N.d2821e1" ranges="R.d2821e2 R.d2821e27 ▶
R.d2821e30"> thus
      shalt thou say unto them, </x:span>
    <x:span start="400" end="457" layer="N.d2821e1"
      ranges="R.d2821e2 R.d2821e27 R.d2821e30 R.d2821e39">My little finger ▶
shall be thicker than my
      father's loins.</x:span>
   <x:span start="457" end="458" layer="N.d2821e1" ranges="R.d2821e2 R.d2821e30 ▶
```

[16]Example taken from http://cocoon.lis.illinois.edu:8080/lis590dpl/wapiez/LMNL/lmnl/xLMNL/1kings12.xml.

```
R.d2821e39"> </x:span>
    <x:span start="458" end="621" layer="N.d2821e1"
      ranges="R.d2821e2 R.d2821e30 R.d2821e39 R.d2821e46">And now whereas my ▶
father did lade you
      with a heavy yoke, I will add to your yoke: my father hath chastised you ▶
with whips, but I
      will chastise you with scorpions.</x:span>
    <x:span start="621" end="622" layer="N.d2821e1" ranges="R.d2821e2"> </x:span>
    <x:span start="622" end="720" layer="N.d2821e1" ranges="R.d2821e2 ▶
R.d2821e54">So Jeroboam and
      all the people came to Rehoboam the third day, as the king had appointed, ▶
saying, </x:span>
    <x:span start="720" end="751" layer="N.d2821e1" ranges="R.d2821e2 R.d2821e54 ▶
R.d2821e57">Come to
      me again the third day.</x:span>
    <x:span start="751" end="752" layer="N.d2821e1" ranges="R.d2821e2"> </x:span>
    <x:span start="752" end="753" layer="N.d2821e1"> </x:span>
  </x:content>
  <x:range start="0" end="752" ID="R.d2821e2" sl="1" so="1" name="excerpt" ▶
el="6" eo="9">
    <x:annotation ID="N.d2821e3" sl="1" so="10" el="1" eo="58" name="source">
      <x:annotation ID="N.d2821e4" sl="1" so="18" el="1" eo="36" name="book">
        <x:content>
          <x:span start="0" end="7" layer="N.d2821e4">1 Kings</x:span>
        </x:content>
      </x:annotation>
      <x:annotation ID="N.d2821e7" sl="1" so="38" el="1" eo="57" name="chapter">
        <x:content>
          <x:span start="0" end="2" layer="N.d2821e7">12</x:span>
        </x:content>
      </x:annotation>
      <x:content/>
    </x:annotation>
  </x:range>
  <x:range start="1" end="163" ID="R.d2821e12" sl="2" so="1" name="verse" ▶
el="2" eo="188"/>
  <x:range start="24" end="163" ID="R.d2821e15" sl="2" so="31" name="q" el="2" ▶
eo="181"/>
  <x:range start="108" end="163" ID="R.d2821e18" sl="2" so="118" name="q" ▶
el="2" eo="178"/>
  <x:range start="164" end="457" ID="R.d2821e27" sl="3" so="1" name="verse" ▶
el="3" eo="325"/>
  <x:range start="234" end="621" ID="R.d2821e30" sl="3" so="78" name="q" el="4" ▶
eo="178"/>
  <x:range start="303" end="368" ID="R.d2821e33" sl="3" so="150" name="q" ▶
el="3" eo="224"/>
```

```
  <x:range start="400" end="621" ID="R.d2821e39" sl="3" so="257" name="q" ▶
el="4" eo="175"/>
  <x:range start="458" end="621" ID="R.d2821e46" sl="4" so="1" name="verse" ▶
el="4" eo="185"/>
  <x:range start="622" end="751" ID="R.d2821e54" sl="5" so="1" name="verse" ▶
el="5" eo="149"/>
  <x:range start="720" end="751" ID="R.d2821e57" sl="5" so="106" name="q" ▶
el="5" eo="142"/>
</x:document>
```

The xLMNL syntax is comparable to XStandoff already discussed in Section 4.1.2: Elements depict possibly overlapping ranges with start and end positions separated from the annotations. The main difference between both formats is that LMNL uses the flat range data model and does not prescribe a specific syntax. One of the disadvantages is that LMNL lacks the whole XML ecosystem consisting of validation languages (although there is the CREOLE validation language proposal by [Tennision 2007]), query and traversal languages. But as [Piez 2012] demonstrates, LMNL instances can be processed by XSLT 2.0 stylesheets and since xLMNL is straightforward XML, it can be validated with regular XML parsers.

5. What comes next?

Current trends in meta language development tend to be diverse: On the one hand we can observe specifications that maintain backwards-compatibility with XML (such as MicroXML and XStandoff) but try to either reduce or extend the syntax and/or the data model. On the other hand, specifications such as LMNL or FtanML tend to change the predominant data model (trees vs. graph vs. ranges) and XML's syntax as well, possibly replacing XML as basis for future markup languages[17]. One point that should be addressed in a possible future meta language should be the support for overlapping hierarchies, that is, we need a richer data formal or a selection of data models (see [van der Vlist 2012] for similar suggestions). Similar to MicroXML, the upcoming HTML5 [Berjon et al. 2012] addresses XML's current parsing model insofar that it tries to create a third way between the quite complex parsing model of SGML (that lead to the *tag soup* that nowadays fills web pages) and XML's draconian error handling by describing precise terms of error handling and recovery. For example, misnested elements will be inserted into the foster parent element in HTML5's DOM representation (which is different from XML's DOM; see [Berjon et al. 2012, Section 8.2.5.3, *Foster parenting*].

Future versions of XML (or whatever it is called then) should be designed modular to unify the current lines of development, that is, the data model and the syntax could be more adaptive to one's needs. In addition, a parsing model similar

[17]It has to be made clear at this point that neither LMNL nor FtanML have the goal to replace XML.

to that of HTML5 (but with included backwards compatibility to the current model) could be an interesting option as well (although it is a fine line between a more relaxed parser and a too forgiving one). Constraint languages should be separated from the base specification (one reason for MicroXML's development was to remove XML's DTD from the specification of the meta language itself). However, XML's greatest benefit, its large number of implementations and surrounding standards, tends to be the largest obstacle in creating a radically new approach such as the variety of current implementations of XML's data model (such as the XML Infoset [Cowan and Tobin 2004] or the XDM [Berglund et al. 2010]) or its schema languages. In general, people participating in the standardization process of a possible successor to XML are advised to have at least a close look at the specifications discussed in this article.

Bibliography

[1] Ahotra, Ashok and Marsh, Jonathan and Nagy, Marton and Norman Walsh (2010). XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition). W3C Recommendation, World Wide Web Consortium (W3C). http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/.

[2] Berjon, Robin, Leithead, Travis, Navara, Erika Doyle, O'Connor, Edward, and Silvia Pfeiffer (2012). HTML5. A vocabulary and associated APIs for HTML and XHTML. W3C Working Draft 17 December 2012, World Wide Web Consortium (W3C). http://www.w3.org/TR/2012/WD-html51-20121217/.

[3] Bray, Tim, Paoli, Jean, Sperberg-McQueen, C. M., Maler, Eve, and Franois Yergeau (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, World Wide Web Consortium (W3C). http://www.w3.org/TR/2008/REC-xml-20081126/.

[4] Burnard, Lou, and Syd Bauman (2012). TEI P5: Guidelines for Electronic Text Encoding and Interchange. Published for the TEI Consortium by Humanities Computing Unit, University of Oxford, Oxford and Providence and Charlottesville and Bergen. Version 2.2.0, 25th October 2012.

[5] Clark, James (2010). MicroXML. 2010-12-13, http://blog.jclark.com/2010/12/microxml.html.

[6] Clark, James, and John Cowan (2012). MicroXML. W3C Community Group Draft Specification, World Wide Web Consortium (W3C). http://dvcs.w3.org/hg/microxml/raw-file/tip/spec/microxml.html.

[7] Coombs, James H., Renear, Allen H., and Steven J. DeRose (1987). Markup Systems and the Future of Scholarly Text Processing. Communications of the ACM, 30(11):933–947.

[8] Costello, Roger L., and Robin A. Simmons (2008). Tutorials on Schematron. http://www.xfront.com/schematron/index.html.

[9] Cowan, John, and Richard Tobin. XML Information Set (Second Edition). W3C Recommendation, World Wide Web Consortium (W3C). http://www.w3.org/TR/2004/REC-xml-infoset-20040204/.

[10] Cowan, John, Tennison, Jeni, and Wendell Piez (2006). LMNL update. In: Proceedings of Extreme Markup Languages 2006, Montréal.

[11] Cowan, John (2012). Microxml: Who, what, where, when, why. In Proceedings of Balisage: The Markup Conference, Balisage Series on Markup Technologies vol. 8, Montréal.

[12] Crockford, Douglas (2006). JSON: The Fat-Free Alternative to XML. In Proceedings of XML 2006, Boston. http://www.json.org/fatfree.html.

[13] DeRose, Steven J., Durand, David G., Mylonas, Elli, and Allen H. Renear (1990). What is text, really? Journal of Computing in Higher Education, 1(2):3–26.

[14] Goldfarb, Charles F., Mosher, Edward J., and Theodore I. Peterson (1970). An online system for integrated text processing. In Proceedings of the 33rd Annual Meeting of the American Society for Information Science (ASIS Proceedings), volume 7, 1970.

[15] Goldfarb, Charles F. (1973). Design Considerations for Integrated Text Processing Systems. Technical Report 320-2094, IBM Cambridge Scientific Center, 5 1973.

[16] Goldfarb, Charles F. (1991). The SGML Handbook. Oxford University Press, Oxford, 1991.

[17] Graham, Ian S., and Liam R. E. Quin (1999). XML Specification Guide. Wiley Computer Publishing, 1999.

[18] Huitfeldt, Claus, and C. M. Sperberg-McQueen (2004). Markup Languages and Complex Documents (MLDC). Presentation given at the colloquium of the HKI (Historisch Kulturwissenschaftliche Informationsverarbeitung) of the University of Cologne, 10 Decembre 2004. http://old.hki.uni-koeln.de/material/huitfeldt.ppt.

[19] Ide, Nancy M., Priest-Dorman, Greg, and Jean Véronis (1996). Corpus Encoding Standard (CES). Expert Advisory Group on Language Engineering Standards (EAGLES), 1996. .

[20] ISO/IEC JTC 1/SC 34 (1986). Information Processing — Text and Office Information Systems – Standard Generalized Markup Language. International Standard ISO 8879, International Organization for Standardization, Geneva.

[21] ISO/IEC JTC 1/SC 34 (1997). ISO 8879 TC 2. Technical Corrigendum ISO/IEC JTC 1/WG 4 N1955, International Organization for Standardization, Geneva.

[22] ISO/TC 37/SC 4/WG 1 (2006). Language Resource Management — Feature Structures – Part 1: Feature Structure Representation. International Standard ISO 24610-1:2006, International Organization for Standardization, Geneva.

[23] ISO/TC 37/SC 4/WG 1 (2012). Language Resource Management — Linguistic Annotation framework (LAF). International Standard ISO 24612:2012(E), International Organization for Standardization, Geneva.

[24] ISO/TC 37/SC 3 (2009). Terminology and other language and content resources -- Specification of data categories and management of a Data Category Registry for language resources. International Standard ISO 12620:2009, International Organization for Standardization, Geneva.

[25] ISO/TC 37/SC 4/WG 1 (2006). Language Resource Management -- Feature Structures -- Part 1: Feature Structure Representation. International Standard ISO 24610-1:2006, International Organization for Standardization, Geneva.

[26] ISO/TC 37/SC 4/WG 1 (2011). Language Resource Management -- Feature Structures -- Part 2: Feature System Declaration. International Standard ISO 24610-2:2011, International Organization for Standardization, Geneva.

[27] Jettka, Daniel, and Maik Stührenberg (2011). Visualization of concurrent markup. From trees to graphs, from 2D to 3D. In: Proceedings of Balisage: The Markup Conference, volume 7 of Balisage Series on Markup Technologies, Montréal.

[28] Kattau, Suzanne (2012). MicroXML: The future of XML? SD Times, http://sdtimes.com/MICROXML_THE_FUTURE_OF_XML_/By_Suzanne_Kattau/About_MICROXML_and_W3C_and_XML/36778.

[29] Kay, Michael (2012). FtanML - A new markup language. http://dev.saxonica.com/blog/mike/2012/08/#000193.

[30] Klarlund, N., Schwentick, T., and D. Suciu (2003). XML: Model, Schemas, Types, Logics and Queries. In: Chomicki, J., R. van der Meyden, and G. Saake, eds., Logics for Emerging Applications of Databases, pages 1-41. Springer, Berlin, Heidelberg.

[31] Larson, Martha, Jijkoun, Valentin, Löffler, Jobst, and Erik Tjong Kim Sang (2007). Practical applications of stand-off annotation. SDV – Sprache und Datenverarbeitung, 31(01/02):115–129.

[32] Lee, David (2011). JXON: an Architecture for Schema and Annotation Driven JSON/XML Bidirectional Transformations. In: Proceedings of Balisage: The Markup Conference, Balisage Series on Markup Technologies, vol. 7. Montréal.

[33] Liu, Jixue, Vincent, Millist, and Chengfei Liu (2003). Local XML Functional Dependencies. In: Proceedings of the 5th ACM international workshop on Web

information and data management, WIDM ’03, pages 23–28, New York, ACM Press.

[34] Eve Maler and Jeanne El Andaloussi (1995). Developing SGML DTDs: From Text to Model to Markup. Prentice Hall PTR, Upper Saddle River.

[35] Marcoux, Yves, Huitfeldt, Claus, and C. M. Sperberg-McQueen (2012). The MLCD Overlap Corpus (MOC): Project report. In: Proceedings of Balisage: The Markup Conference 2012, Balisage Series on Markup Technologies, vol. 8. Montréal.

[36] Møller, Anders (2002). Document Structure Description 2.0. Technical report, BRICS (Basic Research in Computer Science, Aarhus University).

[37] Møller, Anders, and Michael I. Schwartzbach (2006). Schema languages. In An Introduction to XML and Web Technologies. Addison-Wesley, Harlow, 2006.

[38] Murata, M., Lee, D., Mani, M., and K. Kawaguchi (2005). Taxonomy of XML Schema Languages Using Formal Language Theory. ACM Transactions on Internet Technology, 5(4):660–704.

[39] Nelson, Theodor Holm (1997). Embedded Markup Considered Harmful. http://www.xml.com/pub/a/w3j/s3.nelson.html.

[40] Ng, Wilfred (2002). Maintaining consistency of integrated xml trees. In: Proceedings of the Third International Conference on Advances in Web-Age Information Management, WAIM ’02, pages 145–157, Springer.

[41] Nicol, Gavin Thomas (2002). Range Algebra. Presentation given at Extreme Markup Languages 2002.

[42] Nicol, Gavin Thomas (2002a). Attributed Range Algebra. Extending Core Range Algebra to Arbitrary Structures. http://www.mind-to-mind.com/library/papers/ara/attributed-range-algebra-07-2002.html.

[43] Ogbuji, Uche (2012). Introducing MicroXML, Part 1: Explore the basic principles of MicroXML. Learn about the possible future of XML. IBM Developerworks. http://www.ibm.com/developerworks/opensource/library/x-microxml1/index.html.

[44] Ogbuji, Uche (2012a). Introducing MicroXML, Part 2: Process MicroXML with MicroLark. Experimenting with the pioneering MicroXML parser. IBM Developerworks. http://www.ibm.com/developerworks/opensource/library/x-microxml2/index.html.

[45] Piez, Wendell (2004). Half-steps toward LMNL. In: Proceedings of Extreme Markup Languages 2004, Montréal.

[46] Piez, Wendell (2012). Luminescent: Parsing LMNL by XSLT Upconversion. In Proceedings of Balisage: The Markup Conference, Balisage Series on Markup Technologies vol. 8, Montréal.

[47] Renear, Allen H., Mylonas, Elli, and David G. Durand (1996). Refining our notion of what text really is: The problem of overlapping hierarchies. In: Nancy M. Ide and Susan Hockey, editors, Selected Papers from the 1992 ALLC/ACH Conference, Christ Church, Oxford, April 1992, volume 4 of Research in Humanities Computing, pages 263–280. Clarendon Press, Oxford.

[48] Rubinsky, Yuri, and Murray Maloney (1997). SGML on the Web: Small Steps Beyond HTML. Charles F. Goldfarb Series On Open Information Management. Prentice Hall, Upper Saddle River.

[49] Sperberg-McQueen, C. M., and Tim Bray (1997). Extensible Markup Language (XML). In Proceedings of ACH-ALLC '97. Joint International Conference of the Association for Computers and the Human- ities (ACH) and the Association for Literary & Linguistic Computing (ALLC), Kingston. Queen's University.

[50] Sperberg-McQueen, C. M., and Claus Huitfeldt (1999). Concurrent Document Hierarchies in MECS and SGML. In: Literary and Linguistic Computing 14.1:29–42.

[51] Sperberg-McQueen, C. M., Huitfeldt, Claus, and Allen H. Renear (2000). Meaning and interpretation of markup. Markup Languages – Theory & Practice, 2(3):215–234.

[52] Sperberg-McQueen, C. M., and Claus Huitfeldt (2004). GODDAG: A Data Structure for Overlapping Hier- archies. In Peter King and Ethan V. Munson, editors, Digital Documents: Systems and Principles, 8th International Conference on Digital Documents and Electronic Publishing, DDEP 2000, 5th In- ternational Workshop on the Principles of Digital Document Processing, PODDP 2000, Munich, Germany, September 13-15, 2000, Revised Papers, volume 2023 of Lecture Notes in Computer Science, pages 139–160. Springer.

[53] Stührenberg, Maik and Daniela Goecke (2008). SGF – an integrated model for multiple annotations and its application in a linguistic domain. In: Proceedings of Balisage: The Markup Conference, Balisage Series on Markup Technologies, vol 1.

[54] Stührenberg, Maik and Daniel Jettka (2009). A toolkit for multi-dimensional markup: The development of SGF to XStandoff. In: Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3. .

[55] Stührenberg, Maik and Christian Wurm. (2010).Refining the Taxonomy of XML Schema Languages. A new Approach for Categorizing XML Schema Languages in Terms of Processing Complexity. In: Proceedings of Balisage: The Markup Conference 2010. Balisage Series on Markup Technologies, vol. 5.

[56] Stührenberg, Maik (2012). Foundations of Markup Languages. In Mehler, Alexander, and Laurent Romary, editors, Handbook of Technical

Communication, Volume 8 of Handbooks of Applied Linguistics, pages 83–120. de Gruyter.

[57] Stührenberg, Maik (2012a). Auszeichnungssprachen für linguistische Korpora: Theroretische Grundlagen, De-facto-Standards, Normen. Dissertation, Bielefeld University. http://pub.uni-bielefeld.de/publication/2492772.

[58] Tennison, Jeni (2002). Layered Markup and Annotation Language (LMNL). In: Proceedings of Extreme Markup Languages 2002, Montréal.

[59] Tennison, Jeni (2007). Creole: Validating overlapping markup. In: Proceedings of XTech 2007: The Ubiquitous Web Conference, Paris, France.

[60] Architecture Committee for the TIPSTER Text Phase II Program (1996). TIPSTER Text Phase II Architecture Concept. Technical report, National Institute of Standards and Technology.

[61] Thompson, H. S., and D. McKelvie (1997). Hyperlink semantics for standoff markup of read-only documents. In: Proceedings of SGML Europe '97: The next decade – Pushing the Envelope, pages 227–229, Barcelona, 1997.

[62] van Kesteren, Anne (2012). XML5's Story. In: XML Prague 2012 Conference Proceedings, pages 23–26, Prague.

[63] van der Vlist, Eric (2003). XML Schema. O'Reilly. .

[64] van der Vlist, Eric (2004). ISO DSDL overview. In: Proceedings of XML Europe, Amsterdam.

[65] van der Vlist, Eric (2012). The eX Markup Language? In: XML Prague 2012 Conference Proceedings, pages 1–10, Prague.

[66] Witt, Andreas (2004). Multiple hierarchies: New Aspects of an Old Solution. In: Proceedings of Extreme Markup Languages, Montréal.

Embracing JSON? Of course, but how?

Eric van der Vlist
Dyomedea
`<vdv@dyomedea.com>`

Abstract

JSON is now widely used to exchange data over the web and it is crucial for the XML toolkit to seamlessly support JSON.

This is not new and a number of solutions have already been proposed for this purpose. Unfortunately even if the JSON data model is relatively simple it is nor directly compatible with the XML data model and none of these solutions seem to clearly dominate the landscape.

The purpose of this talk is to classify and review different proposals and briefly present a new proposal that will probably be considered as a total heresy by XML fundamentalists.

1. JSON is hot

> ***XML birthday***: *Do we need new syntax, data model or both or is XML just fine after 15 years?*
>
> —*XML Prague 2013 list of topics*

If there should be one single reason why we need to update XML syntax, data model or both, JSON would undoubtedly be this reason.

JSON has be a prominent topic at XML Prague last year:

- JSON was one of the four web formats covered by Jeni Tennison in her unmemorable opening keynote[3].
- Eric van der Vlist stated that the support of JSON was key to the XML ecosystem[4].
- Vojtěch Toman pleaded for JSON support in XProc[5].
- Jonathan Robie presented JSONiq, a query language for JSON, based on XQuery[6].
- Jason Hunter followed with Corona, a JSON friendly REST API to MarkLogic[7].

[3] http://archive.xmlprague.cz/2012/sessions.html#Opening-Keynote-%E2%80%93-Collisions-Chimera-and-Consonance-in-Web-Content
[4] http://archive.xmlprague.cz/2012/sessions.html#The-eX-Markup-Language
[5] http://archive.xmlprague.cz/2012/sessions.html#XProc-Beyond-application/xml7
[6] http://archive.xmlprague.cz/2012/sessions.html#JSONiq-XQuery-for-JSON-JSON-for-XQuery
[7] http://archive.xmlprague.cz/2012/sessions.html#Corona-Managing-and-querying-XML-and-JSON-via-REST

- Steven Pemberton described how the XForms Working Group plans to treat JSON as a subset of XML[8] (Michael Kay couldn't help exclaiming "Oh no! not yet another JSON to XML mapping proposal!").
- Adam Retter made it clear that his RESTful XQuery proposal[9] did support JSON serializations.
- Michael Kay presented the new XSLT 3.0 map datatype as a way to support JSON in XSLT 3.0[10].

JSON has also been a key topic at Balisage in August:

- Eric van der Vlist dared to dissent on the support of JSON in the current XSLT 3.0 proposal[11]
- Jonathan Robie gave a higher level view of JSONiq[12],
- Hans-Jürgen Rennau presented UDL (Unified Document Language), a proposal to extend the XDM to support JSON[13].
- John Cowan, in his MicroXML talk[14] described the coexistence with JSON as one of the project motivations.
- A nocturne has been held on the specific subject of bridging the JSON and XML data models.

Why such a flurry of proposals to solve a problem that is supposedly simple and how do they relate?

Is it too late to discuss that kind of things? One of the conference reviewers thought so:

> *Sorry, I disagree on the timing of this submission ... I think JSON has already been embraced ... we've moved on, for all the reasons that you state. Its hard for me to recommend to accept this but I would have liked to heard/seen you present it!*
>
> — *Assigned_Reviewer_3*

Overall, the reviews were more balanced but this talk is going to be challenging:

> *There's been an awful lot on this topic in the last couple of years and people may be getting a little bored of it. However, it remains an important topic and this proposed paper seems to make a useful contribution to the debate. Because we only have a*

[8] http://archive.xmlprague.cz/2012/sessions.html#Treating-JSON-as-a-subset-of-XML-Using-XForms-to-read-and-submit-JSON
[9] http://archive.xmlprague.cz/2012/sessions.html#RESTful-XQuery---Standardised-XQuery-3.0-Annotations-for-REST
[10] http://archive.xmlprague.cz/2012/sessions.html#Whats-New-in-XPath/XSLT/XQuery-3.0-and-XML-Schema-1.1
[11] http://balisage.net/Proceedings/vol8/html/Vlist01/BalisageVol8-Vlist01.html
[12] http://balisage.net/Proceedings/vol8/html/Robie01/BalisageVol8-Robie01.html
[13] http://balisage.net/Proceedings/vol8/html/Rennau01/BalisageVol8-Rennau01.html
[14] http://balisage.net/Proceedings/vol8/html/Cowan01/BalisageVol8-Cowan01.html

sketch of the paper, there is a tendency to skirt around the detail, and the quality of the final paper absolutely depends on the detail. By accepting it I think we're putting a lot of trust in the author to create something where the detail is worthwhile, because the abstract doesn't really show how the difficult problems will be solved, it only leaves placeholders for the solutions.

— *Assigned_Reviewer_4*

Let's see how I can take this challenge!

2. JSON and XML: the warring brothers

If we take a simple JSON document such as this sample borrowed from Wikipedia:

```
{
  "person": {
    "firstName": "John",
    "lastName": "Smith",
    "age": "25",
    "address": {
      "streetAddress": "21 2nd Street",
      "city": "New York",
      "state": "NY",
      "postalCode": "10021"
    },
    "phoneNumber": [
      {
        "type": "home",
        "number": "212 555-1234"
      },
      {
        "type": "fax",
        "number": "646 555-4567"
      }
    ]
  }
}
```

It is, bracket style apart, confusingly similar to this XML fragment:

```
<?xml version="1.0" encoding="UTF-8" ?>
<person>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <age>25</age>
    <address>
  <streetAddress>21 2nd Street</streetAddress>
  <city>New York</city>
```

```
    <state>NY</state>
    <postalCode>10021</postalCode>
   </address>
      <phoneNumber>
          <type>home</type>
          <number>212 555-1234</number>
      </phoneNumber>
      <phoneNumber>
          <type>fax</type>
          <number>646 555-4567</number>
      </phoneNumber>
 </person>
```

These snippets are so similar that we could think that a bunch of regular expressions is enough to convert one into the other...

Of course, the devil is in the details and that's not the case because:

1. These samples do not expose all the features of the two notations (no attribute, PI, namespace, ... in the XML example; no keys with spaces, binary values, ... in the JSON sample).
2. We look at them syntactically and their data models state that what we see is not what we get (for instance, the relative order between JSON object properties is not significant).

To understand the different proposals, how they relate and differ, we must thus compare the XML and JSON data models.

2.1. XML Data Model(s)

To make thinks worse, or more interesting, XML per see doesn't come with a data model. The XML and Namespaces in XML recommendations (1.0 or 1.1) are mostly about syntax and mainly define the class of documents which can be considered as XML documents.

In practice, three different data models are commonly used when working with XML documents:

- The XML Information Set[15] (aka XML infoset) "“defines an abstract data set called the **XML Information Set** (**Infoset**). Its purpose is to provide a consistent set of definitions for use in other specifications that need to refer to the information in a well-formed XML document [XML][16]”".
- XML Schema defines[17] a **post-schema-validation infoset**, or **PSVI** as " “the augmented infoset which results from conformant processing as defined in this

[15] http://www.w3.org/TR/xml-infoset/
[16] http://www.w3.org/TR/xml-infoset/#XML
[17] http://www.w3.org/TR/xmlschema11-1/#key-psvi

specification"". In other words, the PSVI is an XML infoset with added information such as type information.

- XPath/XSLT/XQuery define their own data models often called the **XPath Data Model** or **XDM**. These data models are basically composed of a subset of the XML infoset with some information from the PSVI (versions 2 and 3 only) and language specific information items (such as sequences, atomic values, functions and, for XSLT 3.0, maps). To make things worse, these data models are different between versions 1, 2 and 3 and can be extended by their host languages (XSLT and XQuery). We have thus different versions and flavors of XDM.

Tip

When you hear the words "XML Data Model", always ask "which one?". When you hear the word "XDM", always ask "which version? which flavor?".

The XML Data Model which we'll be considering in this paper is the subset of the XML Infoset used by the various versions of flavors of the XDM. This subset is reasonably stable between versions and consist of seven kind of nodes[18]:

- Document nodes
- Elements
- Attributes
- Text nodes
- Comments
- Processing Instructions (PIs)
- Namespace nodes

The XML Data Model can be summarized as a hierarchical structure composed of elements.

Each element has:

- A name.
- A map of attributes (key/value pairs) which values are textual.
- An (ordered) array of children element, text, comment and PI nodes. Text nodes have a specific behaviour in this array of children nodes:

Text Nodes that would be adjacent ***must*** *be combined into a single Text Node.*

A document node is the root of such a structure and namespace nodes are transversal and serve to "qualify" the names of elements and attributes.

XML and Namespaces in XML introduce two lexical restrictions:

[18] http://www.w3.org/TR/xpath-datamodel-30/#Node

- "Legal characters are tab, carriage return, line feed, and the legal characters of Unicode and ISO/IEC 10646". This excludes "control" characters below #x20 other than tab, CR and LF and requires to encode binary content.
- Names are qualified names which "local part" must follow the NCName production[19]. This exclude names starting with digits, containing spaces or more generally arbitrary text to be used as names. Additionally, XML names should not start with the string "xml".

2.2. JSON Data Model

JSON, the fat-free alternative to XML[20], is derived from the ECMAScript Programming Language Standard, Third Edition [ECMA[21]] and described in the IETF RFC4627[22].

JSON makes a distinction between primitive and structured types:

JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays).

The definitions of objects and arrays is straightforward:

An object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.

An array is an ordered sequence of zero or more values.

There no lexical restriction neither for strings:

A string is a sequence of zero or more Unicode characters.

nor for names:

a name is a string.

2.3. Comparison

2.3.1. Type system

JSON introduces four primitive types (strings, numbers, booleans, and null) while there is no notion of type in XML itself.

Types are usually introduced in XML through "type annotations" coming either from the PSVI or from `@xsi:type` attributes but then we run into a new difference:

[19] http://www.w3.org/TR/2009/REC-xml-names-20091208/#NT-NCName
[20] http://www.json.org/fatfree.html
[21] http://tools.ietf.org/html/rfc4627#ref-ECMA
[22] http://tools.ietf.org/html/rfc4627

the XML Schema datatype system introduces much more datatypes than JSON supports.

In other words,

- with "raw" XML, there is no way to tell for sure if `<foo>1</foo>` should be translated in JSON as `foo: 1` or `foo:"1"`.
- with "typed" XML, there is no way to tell for sure if `foo: "2012-02-10"` should be translated in XML as `<foo xsi:type="xs:date">2012-02-10</foo>` or `<foo xsi:type="xs:string">2012-02-10</foo>`.

2.3.2. Lexical space

The lexical restrictions of XML on legal characters and names do not exist in JSON.

2.3.3. Structure

We have both maps of key/value pairs and ordered arrays in JSON and XML, but that's where the similarities stop!

In JSON, maps of key/value pairs are called objects. Keys can be any string and values may be either primitive or structured types.

In XML, elements have a map of key/value pairs among their properties. This map is called attributes. Keys (ie attribute names) are subject to lexical restrictions and values cannot be structured types.

In JSON, ordered arrays are called arrays and their members may be either primitive or structured.

In XML, elements have an ordered array of children nodes among their properties. Their members can be elements, comments, PIs or text nodes. XML text nodes are the kind of nodes which is the more similar to JSON's primitive types. However, adjacent text nodes are concatenated (which of course is not the case of adjacent primitive values in a JSON array).

2.3.4. Encodings

This doesn't make any difference at a data model level and we won't discuss that further, but for completeness I need to mention that JSON is always expressed in UTF-8 while XML can use a number of different encodings.

2.3.5. So what?

Oh, no, not yet another JSON to XML mapping proposal!
—Michael Kay listening Steve Pemberton present JSON support in XForms 2.0 at XML Prague 2012

The data models are different, so what?

They are not only different but none of them can be considered as a superset of the other. The good news is of course that they are not disjoint either!

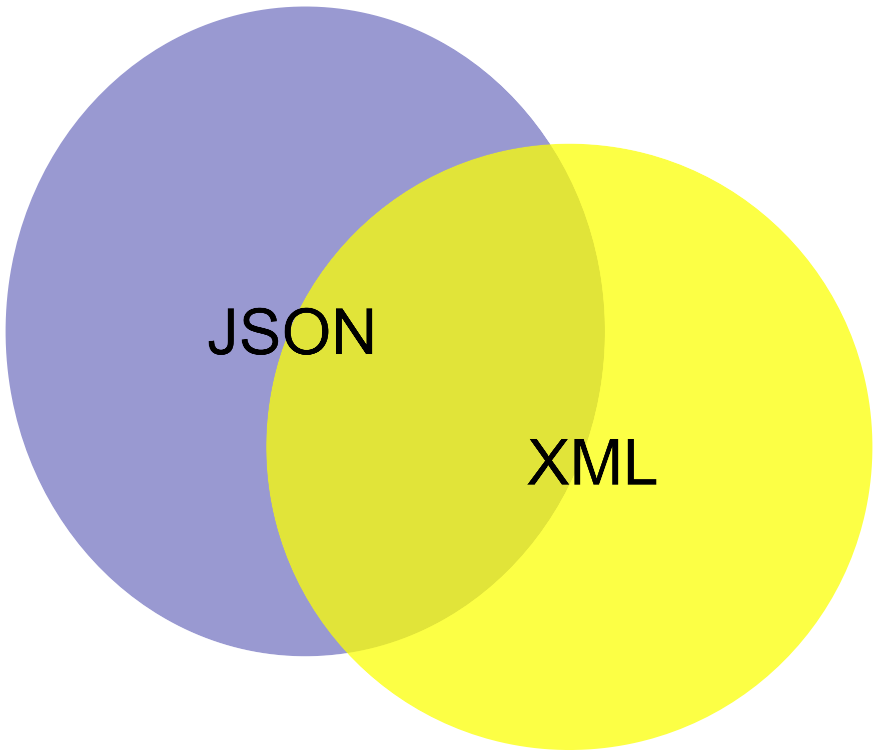


Figure 1. JSON and XML compared

We are in a situation where we have on one side JSON, a simple generic data model with a few structural and primitive types and on another side XML, a document markup specialized data model with more complex structural types and no primitive types.

JSON can be seen as simple, generic building blocks (maps, arrays, ...) and XML as building blocks slightly more complex which are specialized combinations and restrictions of JSON's building blocks.

The situation is somewhat similar to the relations between RDF and Topic Maps with RDF being the JSON and Topic Maps playing the role of XML.

With these differences between the data models, it is clear that if we don't update the XML data model there will be no perfect solution, that we'll have to accept restrictions, make compromises and these compromises will be influenced by the requirements.

The requirements may differ in a number of ways:

- **Direction**: do we need to bridge JSON to XML, XML to JSON or both?
- **Scope**: for each direction: which subset of source and target documents do we support?

- **Information loss**: for each direction: in this subset, which information can we afford to loose?
- **Round trip** (another way to look at information loss): for each direction, do weed need roundtrip?
- **Style**: how important is it that the JSON|XML side looks like "natural, good practiced" JSON|XML.
- **Query-ability**: how important is it to be able to query the JSON|XML side, with which language and features?

The approaches can be split into four main categories:

- **Bridges**: Information expressed in a source format is translating into something as similar as possible in the destination format.
- **Polyglotism**: Different languages or dialects are used to manipulate both formats and move information around from one to another.
- **Serialization**: One of the formats is used to serialize the data model of the other format.
- **Consolidation**: The target data model is extended to become a superset the source data model.

3. Use cases

3.1. Data translation

Acme Corporation has developed a set of RESTfull web services which sends and receives data oriented XML documents. Due to popular demand, they want to support JSON in addition to XML.

Acme Web Design has developed a set of RESTFull web services which sends and receives JSON documents. Their business users require the same information in XML.

In both cases we do not really care about the format which is used and what is important is the information to exchange. If these documents describe well known Acme anvils we do not care about XML or JSON subtleties but we must make sure that they give the same information and describe the same anvils in both formats.

ACME AMERICAN WROUGHT ANVILS

THEY RING LIKE A BELL. No anvil made, English or American, surpasses our Acme in shape, material or finish. It is solid forged of two pieces of best wrought iron, welded at waist; face is made of one piece of tool steel, electrically welded to the body and warranted not to come loose. Base has sufficient spread to insure stability and prevent tipping; has long perfectly shaped horn and heel; face is trued and shaped by a special machine so that there are no hollow or uneven places; edges are perfectly tempered and will not chip. Hardie holes are straight and true, so you will have no trouble on account of anvil tools sticking or not setting level.

WE HAVE THE EXCLUSIVE SALE OF THE ACME. We take the entire output of the factory that makes them, and get them so cheap we are enabled to sell them at a lower price than others pay for anvils not as good. We sell more anvils than any concern in the United States. We could not sell so many unless they were everything we claim for them.

ONLY $9\frac{1}{2}$C POUND.

Usual Price is 15c

You Save More Money

The low prices on hardware in our last catalogue are lower still in this book. Over 3,000 hardware items reduced.

Figure 2. An ad for an Acme anvil, source Wikimedia[23]

This is a situation where bridges are typically used.

These bridges map JSON object and properties to XML elements and attributes. There are a number of these, such as XML.ObjTree[24], a JavaScript implementation by Yusuke Kawasaki[25] that you can use online at http://jsontoxml.utilities-online.info/.

This library is following the same principles than Stefan Goessner has exposed on XML.com[26] to map XML attributes and mixed content to JSON:

- Attributes names are prefixed in JSON with a character which is not a valid character for beginning an element name is XML (Stefan Goessner had proposed to use a "`@`" which can be a problem for E4X and Yusuke Kawasaki is using a "`-`"): `<person name="smith"/>` is mapped into `"person": { "-name": "smith" }`.
- Text nodes within mixed content (or simple content elements with attributes) are translated into `"#text"` properties. This is not fully accurate since `<p>This <b>is</b> a text</p>` gives `{ "p": { "#text": [ "This ", " a text" ], "b": "is" } }` but that's probably the best that can be done without further complication.

[23] http://commons.wikimedia.org/wiki/File:Acme_anvil.gif
[24] http://www.kawa.net/works/js/xml/objtree-e.html
[25] http://www.kawa.net/xp/index-e.html
[26] http://www.xml.com/lpt/a/1658

- Namespaces attributes are treated like any other attribute.
- Comments and PIs are ignored.

Let's try to convert an anvil from XML to JSON with http://jsontoxml.utilities-online.info/ to illustrate our point.

The following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<anvil reference="acme-5103">
    <weight unit="pound">9.5</weight>
    <composition>best wrought iron</composition>
    <price currency="USD">.15</price>
</anvil>
```

produces this JSON object:

```
{
  "anvil": {
    "-reference": "acme-5103",
    "weight": {
      "-unit": "pound",
      "#text": "9.5"
    },
    "composition": "best wrought iron",
    "price": {
      "-currency": "USD",
      "#text": ".15"
    }
  }
}
```

Note

The round trip is perfect in that case and the JSON document can be converted back into the original XML format.

This JSON object is perfectly valid but the minus sign before some object keys and the "#text" key will look weird for JSON users.

If the description had been originally done in JSON the document would probably have looked more like:

```
{
  "anvil": {
    "reference": "acme-5103",
    "weight": {
      "unit": "pound",
      "value": 9.5
    },
    "composition": "best wrought iron",
    "prices": {
        "usd": 0.15,
        "eur": 0.12
    }
  }
}
```

which is the equivalent of this -fine but nonetheless different from and less XMLish than our original format- XML document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<anvil>
    <reference>acme-5103</reference>
    <weight>
        <unit>pound</unit>
        <value>9.5</value>
    </weight>
    <composition>best wrought iron</composition>
    <prices>
        <usd>0.15</usd>
        <eur>0.12</eur>
    </prices>
</anvil>
```

To reduce these impedance mismatches we need to know the structures (or schemas) of the documents which we will translate. When that's the case, we can design specific transformations or rely on annotations. For a well thought example of annotation driven transformation, see David Lee's JXON talk[27] at Balisage 2011.

3.2. Polyglotism

ACME is developing an application which gets anvil descriptions in XML and prices lists in JSON.

Descriptions come as:

```
<?xml version="1.0" encoding="UTF-8"?>
<anvils>
    <anvil reference="acme 5103">
        <weight unit="pound">9.5</weight>
        <composition>best wrought iron</composition>
    </anvil>
    <anvil reference="acme 5104">
        <weight unit="pound">15</weight>
        <composition>ultimate best wrought iron</composition>
```

[27] http://www.balisage.net/Proceedings/vol7/html/Lee01/BalisageVol7-Lee01.html

```
    </anvil>
</anvils>
```

and prices as:

```
{
"acme 5103":
    {
        "usd": 0.15,
        "eur": 0.12
    },
"acme 5104":
    {
        "usd": 0.20,
        "eur": 0.17
    }
}
```

In that case we'll use programming language features or libraries to read JSON and XML in a single application which can then consolidate both inputs.

There are a number of solutions for that in your favorite programming language and I'll give examples in a couple of them, showing how simple it would be to display the price and weight from one of these anvils.

JavaScript and its E4X has been a good example of a programming language treating both XML and JSON as first class citizens.

Note

Note that E4X has lost traction and its support has been dropped from Firefox 18.

With E4X, the exercise is as simple as:

```
var descriptions = <anvils>
    <anvil reference="acme 5103">
        <weight unit="pound">9.5</weight>
        <composition>best wrought iron</composition>
    </anvil>
    <anvil reference="acme 5104">
        <weight unit="pound">15</weight>
        <composition>ultimate best wrought iron</composition>
    </anvil>
</anvils>;

var prices = {
"acme 5103":
    {
        "usd": 0.15,
```

Figure 3. Tower of Babel (source Wikimedia [http://commons.wikimedia.org/wiki/File:Weltchronik_Fulda_Aa88_016r_detail.jpg])

```
            "eur": 0.12
        },
    "acme 5104":
        {
            "usd": 0.20,
            "eur": 0.17
        }
    };
```

```
var anvil = "acme 5103";

print("Weight: " + descriptions.anvil.(@reference == anvil).weight
    + ", price: $" + prices[anvil].usd);
```

This sample can be run using the rhino JavaScript interpreter:

```
vdv@ldlc:~/Documents/Dyomedea/presentations/en/xmlprague2013$ rhino anvil.js
Weight: 9.5, price: $0.15
vdv@ldlc:~/Documents/Dyomedea/presentations/en/xmlprague2013$ ▶
```

Note

See the difference in the syntaxes used to select information in JSON objects (which happen to be pure JavaScript objects) and in XML trees. Even if XML nodes become part of the JavaScript data model with E4X, they are still different beasts, there is no homogeneity between the XML world and the good old JavaScript objects but rather a brand new set of objects.

Using E4X to access XML is not transparent at all and programmers need to learn two different "sub languages".

Python is a good example of a programming language supporting both XML and JSON is its standard library. Using the ElementTree XML API -arguably not the best binding API but having the advantage to be part of the standard API-, the same exercise is as simple as:

```
import json

prices = json.loads('''
{
"acme 5103":
    {
        "usd": 0.15,
        "eur": 0.12
    },
"acme 5104":
    {
        "usd": 0.20,
        "eur": 0.17
    }
}''')

import xml.etree.ElementTree as ET

descriptions = ET.fromstring('''<?xml version="1.0" encoding="UTF-8"?>
<anvils>
    <anvil reference="acme 5103">
```

```
        <weight unit="pound">9.5</weight>
        <composition>best wrought iron</composition>
    </anvil>
    <anvil reference="acme 5104">
        <weight unit="pound">15</weight>
        <composition>ultimate best wrought iron</composition>
    </anvil>
</anvils>''')

anvil = "acme 5103"

print "Weight: " + descriptions.findall("anvil[@reference='" + anvil + "']/▶
weight")[0].text \
    + ", price: $%.2f" % prices[anvil]['usd']
```

This sample can be run using the python interpreter:

```
vdv@ldlc:~/Documents/Dyomedea/presentations/en/xmlprague2013$ python anvil.py
Weight: 9.5, price: $0.15
vdv@ldlc:~/Documents/Dyomedea/presentations/en/xmlprague2013$ ▶
```

Note

Here again, note the difference between the syntaxes used to access XML nodes and JSON objects. In both cases the documents have been parsed and stored in Python objects, but this has been done under the control of two different libraries which respect the differences between the XML and JSON data models and our programers still need to be polyglot to manipulate both sources.

3.3. JSON support in XSLT

JSON is a popular format for exchange of structured data on the web: it is specified in [JSON[30]]. This section describes facilities allowing JSON data to be processed using XSLT.

—XSL Transformations (XSLT) Version 3.0 - Working Draft 10 July 2012[29]

XSLT developers should be able to process JSON data? This is mirroring the situation of JavaScript developing E4X to add support for XML and the solution which is proposed in the XSLT 3.0 latest Working Drafts is following the same principles: add a new layer of objects (item kinds in XDM terminology) to support a new feature.

This support has been presented by Michael Kay at XML Prague last year in more detail that I can do here.

[30] http://www.ietf.org/rfc/rfc4627.txt

[29] http://www.w3.org/TR/2012/WD-xslt-30-20120710/

The main addition to the XDM is the introduction of maps, a new item kind based on XDM 3.0 functions. Maps are a superset of JSON objects (map keys can be any atomic value and not only strings, map values can be any XDML item type including nodes, ...).

Maps are not considered as nodes and cannot be manipulated or matched using axis paths.

Overall JSON support in XSLT 3.0 will be similar to whet we've seen in the previous section and the equivalent of the two scripts presented there would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs" version="3.0">

    <xsl:output method="text"/>

    <xsl:variable name="descriptions">
        <anvils>
            <anvil reference="acme 5103">
                <weight unit="pound">9.5</weight>
                <composition>best wrought iron</composition>
            </anvil>
            <anvil reference="acme 5104">
                <weight unit="pound">15</weight>
                <composition>ultimate best wrought iron</composition>
            </anvil>
        </anvils>
    </xsl:variable>

    <xsl:variable name="json-prices" as="xs:string"><![CDATA[
 {
"acme 5103":
    {
        "usd": 0.15,
        "eur": 0.12
    },
"acme 5104":
    {
        "usd": 0.20,
        "eur": 0.17
    }
}
        ]]></xsl:variable>

    <xsl:variable name="prices" select="parse-json($json-prices)"/>
    <xsl:variable name="anvil">acme 5103</xsl:variable>

    <xsl:template match="/">
        <xsl:text>Weight: </xsl:text>
       <xsl:value-of select="$descriptions/anvils/anvil[@reference = $anvil]/▶
weight"/>
        <xsl:text>, price: $</xsl:text>
        <xsl:value-of select="$prices($anvil)('usd')"/>
    </xsl:template>

</xsl:stylesheet>
```

Note

We find the same kind of difference between axis based XPath expressions used to navigate in node trees and a different syntax based on function calls used to manipulate maps which we've seen in JavaScript and Python.

3.4. JSON support in XQuery

XQuery is well suited for hierarchical semi-structured data. Many implementations exist, and can easily be adapted to add JSON support.

—Jonathan Robie[31], XML Prague 2012

JSONiq[32] is "“a small and simple set of extensions to XQuery that add support for JSON”".

JSONiq differs from XSLT 3.0 maps and arrays in a number of terms (the JSON terminology is preserved and objects are called... objects) and design decisions (objects are not based on functions, arrays are not based on maps/objects, maps and arrays have identities and are not idempotent, ...) but the principle is still to add brand new item types into the XDM.

Axis navigation which is the basis of XPath navigation within XML documents have not been extended either and objects and arrays can only be accessed through selectors.

Like XSLT 3.0 maps, JSONiq can thus be classified in the category of polyglot approaches.

3.5. Serialization

Acme software needs to store JSON documents in their big XML database. These are arbitrary JSON documents which can be tough to map into XML. There is still

[31] http://archive.xmlprague.cz/2012/presentations/JSONiq_XQuery_for_JSON_JSON_for_XQuery/JSONiq-EMC-Brownbag.xhtml#%287%29
[32] http://jsoniq.org/docs/spec/en-US/html/index.html

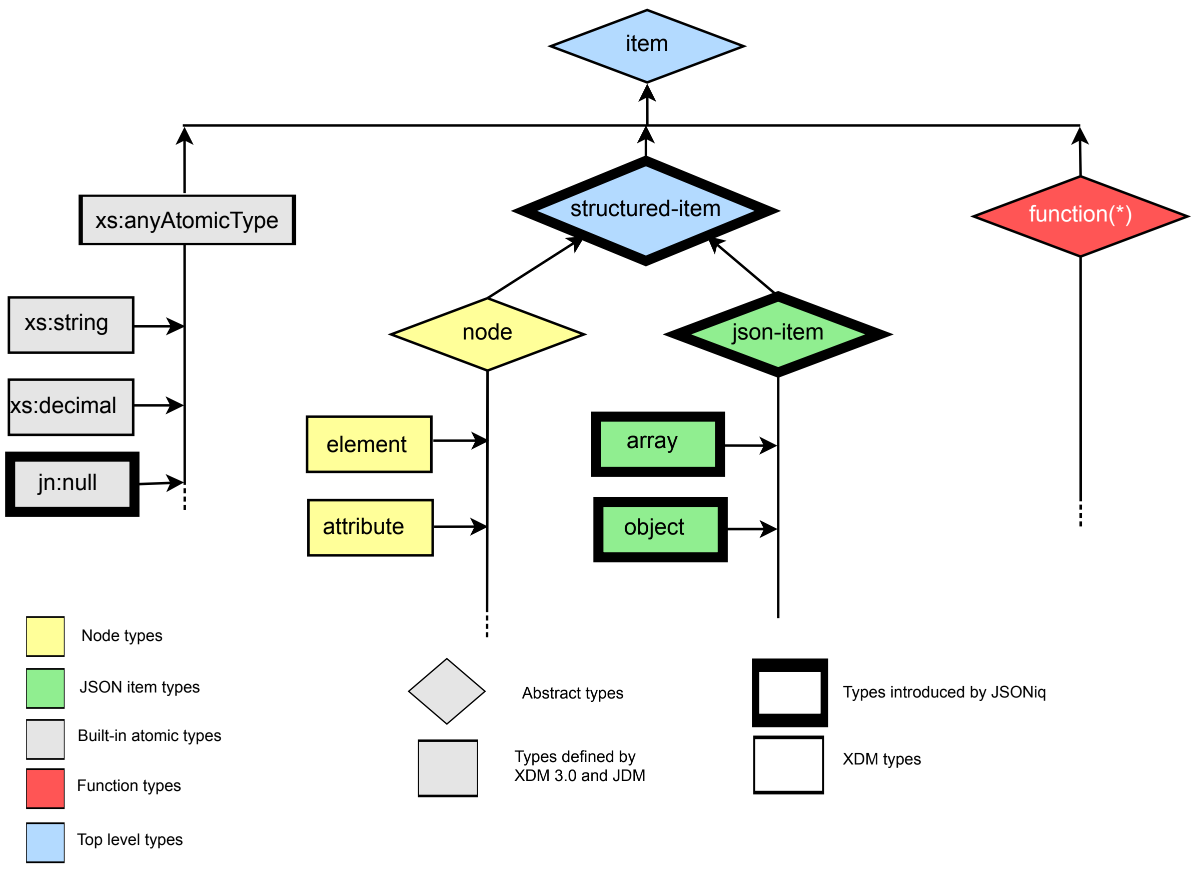


Figure 4. JSONiq item kinds hierarchy

> *a need to query the corpus and storing them as plain text in a root element i snot an option either.*

This is a typical case where a serialization makes sense. The basic idea there is to serialize the JSON document at a data model level like you'd serialize any data model in XML.

This approach is not new and John Snelson did propose something along these lines[34] a while ago which is used by Zorba as one of two different options to serialize JSON[35].

Back from XML Prague last year, I wanted to learn more about the XSLT 3.0 map feature and wrote a χίμαιραλ[36] (chimeral) an experimental XML serialization format for XDM 3.0. Since XDM 3.0 includes maps which are a superset of the JSON data model, this format can be used to serialize JSON documents.

The serialization of our JSON anvil:

[34] http://john.snelson.org.uk/parsing-json-into-xquery
[35] http://www.zorba-xquery.com/html/modules/zorba/data-converters/json
[36] http://χίμαιραλ.com/

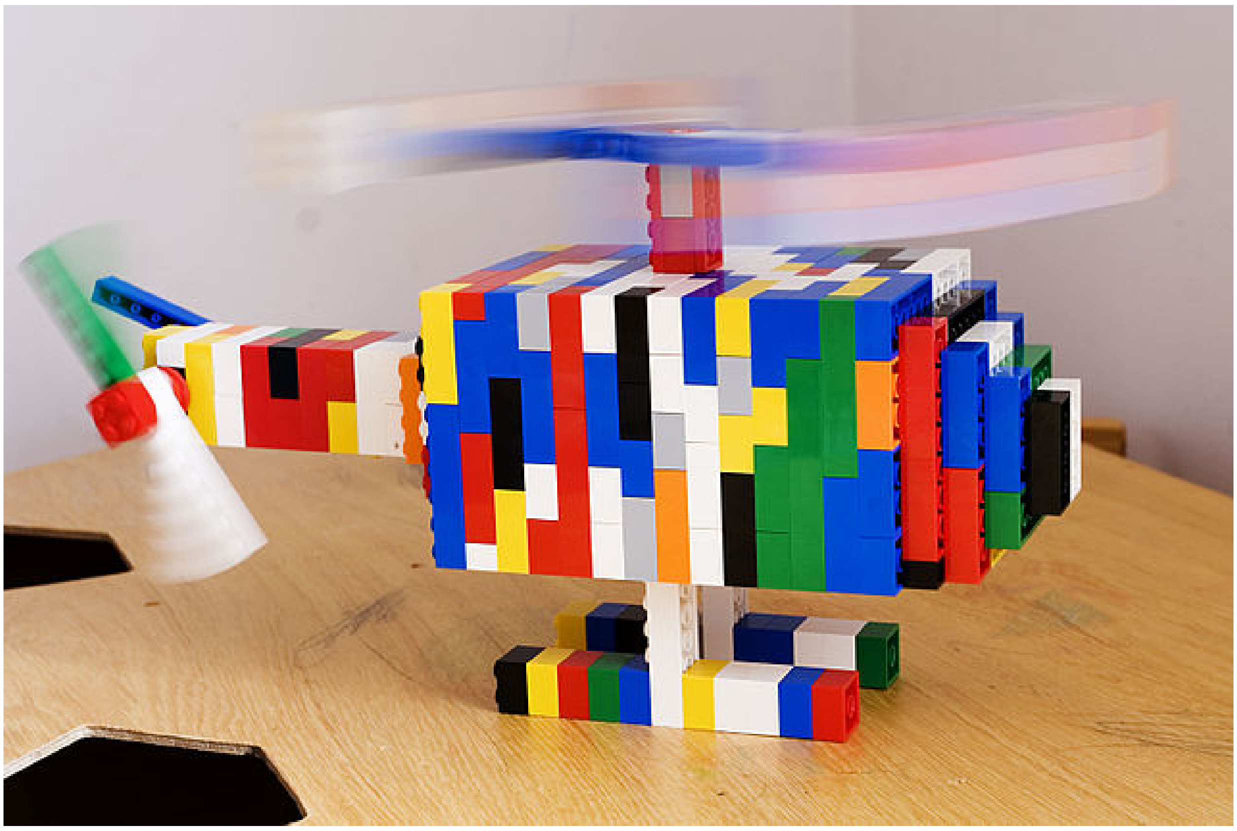


Figure 5. An helicopter serialized in legos (source Wikimedia [http:// commons.wikimedia.org/wiki/File:Lego_helicopter.jpg])

```
{
  "anvil": {
    "reference": "acme-5103",
    "weight": {
      "unit": "pound",
      "value": 9.5
    },
    "composition": "best wrought iron",
    "prices": {
        "usd": 0.15,
        "eur": 0.12
    }
  }
}
```

is serialized into:

```
<?xml version="1.0" encoding="UTF-8"?>
<χ:data-model xmlns:χ="http://χίμαιραλ.com#">
   <χ:map>
      <χ:entry key="anvil" keyType="string">
```

```
        <χ:map>
          <χ:entry key="weight" keyType="string">
            <χ:map>
              <χ:entry key="unit" keyType="string">
                <χ:atomic-value type="string">pound</χ:atomic-value>
              </χ:entry>
              <χ:entry key="value" keyType="string">
                <χ:atomic-value type="number">9.5</χ:atomic-value>
              </χ:entry>
            </χ:map>
          </χ:entry>
          <χ:entry key="composition" keyType="string">
           <χ:atomic-value type="string">best wrought iron</χ:atomic-value>
          </χ:entry>
          <χ:entry key="prices" keyType="string">
            <χ:map>
              <χ:entry key="eur" keyType="string">
                <χ:atomic-value type="number">0.12</χ:atomic-value>
              </χ:entry>
              <χ:entry key="usd" keyType="string">
                <χ:atomic-value type="number">0.15</χ:atomic-value>
              </χ:entry>
            </χ:map>
          </χ:entry>
          <χ:entry key="reference" keyType="string">
            <χ:atomic-value type="string">acme-5103</χ:atomic-value>
          </χ:entry>
        </χ:map>
      </χ:entry>
    </χ:map>
  </χ:data-model>
```

This syntax is verbose, but it is lossless, can be used with any JSON document which doesn't contain characters forbidden in XML and can be queried using axis based XPath expressions even if they grow rapidly verbose. As an example, to find the USD price of anvil reference, you'd write something like `//χ:entry[@key="anvil"]/χ:map[χ:entry[@key="reference"]/χ:atomic-value = "acme-5103" ]/χ:entry[@key="prices"]/χ:map/χ:entry[@key="usd"]/χ:atomic-value`.

3.6. JSON support in XForms 2.0

XForms 1.1 → XForms 2.0 .../... accept data in other formats than XML
An obvious data format widely in use on the web is JSON.
— Steven Pemberton[37] XML Prague 2012

[37] http://www.cwi.nl/%7Esteven/

XForms allows the initialization, processing and serialization of instances whose data come from a JSON source by transforming the JSON value into an XML instance, and serializing it back out as JSON. The XML version of JSON has been designed to be round-trippable, and to allow XPath selectors that resemble the equivalent Javascript selectors.

—*W3C XForms wiki*[38]

One of the questions I asked after Pemberton's was "[would you consider using XSLT 3.0 map extensions?]". Steven Pemberton answered that this was not possible due to timing constraints.

A year later I think that the key point is to choose between the different approaches which we've already seen:

- A bridge which will work most of the time with a few glitches.
- A serialization which will work all the time but imply lengthy XPath expressions hardly readable.
- A polyglot data model in which XSLT 3.0 maps or native JSON arrays and objects would have to be supported as well as XML nodes.

XForms has chosen to use a bridge and that's probably the safest thing to do!

3.7. Consolidation

A possible response to this perception is a new dream: the dream of one universal information language, backed by several syntax variants aka markup languages. If XML is not as universal as it looked a few years ago – might we extend the language (no pun intended), restoring the universality?

—*Hans-Jürgen Rennau*[39]*, Balisage 2012*

Bridges are not perfect and do not work all the time, polyglotism turns into ugly chimeras and serialization is verbose...

Can't we consolidate XML and JSON data models into a more generic one that would homogeneously include the features of XML and JSON?

UDL (Unified Document Language) has been proposed by Hans-Jürgen Rennau at Balisage 2012. UDL includes a new data model which is a superset of the XML data model. To embrace JSON, UDL adds two properties to element nodes:

- **model** which can either be "map" (the element should then be considered as a map) or "sequence" (the element is an array).
- **key** which can only be used by children of maps.

The benefit of this proposal is that the impact on the data model is somewhat minimal (only two properties have been added).

[38] http://www.w3.org/MarkUp/Forms/wiki/index.php?title=Json&oldid=3735
[39] http://balisage.net/Proceedings/vol8/html/Rennau01/BalisageVol8-Rennau01.html

UDL appears to me as a strict superset or collage of JSON and XML and the frontiers between both formats are still very visible: an element can either be a "traditional" element, a map or a sequence and in each case strict limitations apply.

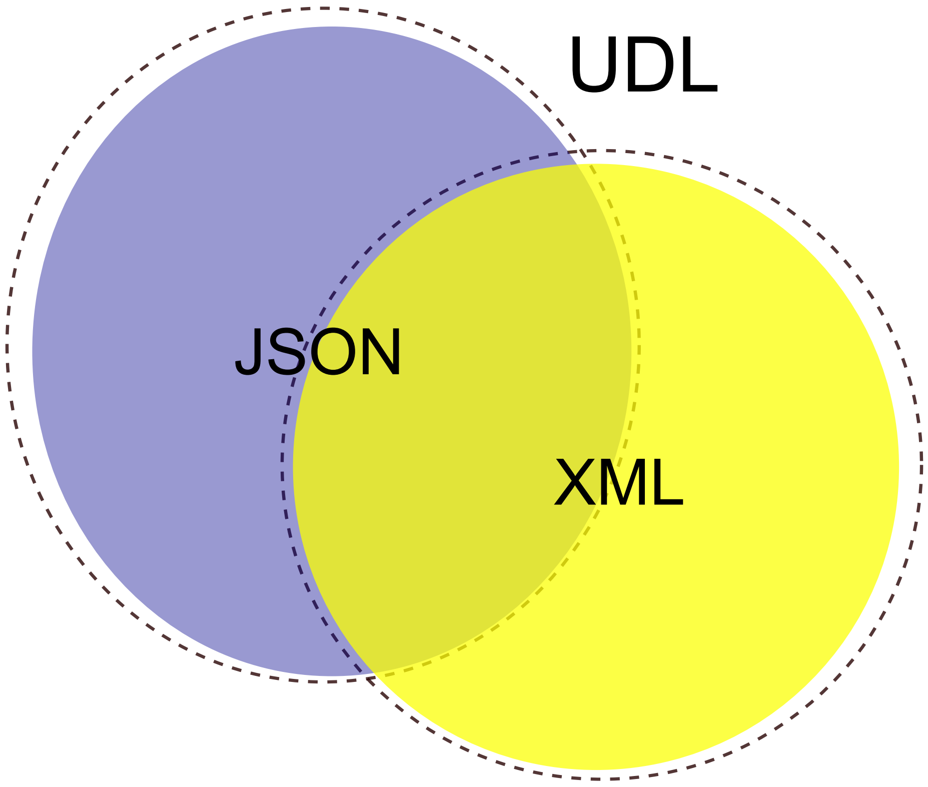


Figure 6. UDL as a strict superset of JSON and XML with apparent frontiers

Another approach would be to go back to the basics of the XML data model and erase the differences between the maps and arrays which are already part of XML and those of JSON. This is the purpose of the χίμαιραλ (chimeral) / superset[40] that I have drafted on my blog.

Like UDL, the goal of χίμαιραλ/superset is to allow to use XML elements as JSON arrays or maps. However, while UDL does so by adding new properties, χίμαιραλ/superset does so by removing constraints on elements, attributes and text nodes:

- Elements can be anonymous (JSON objects and arrays do not have names).
- Attributes names can be QNames, but also strings, numbers, booleans and null (to be used as JSON object keys).
- Attributes can have children attributes and nodes (to be used as JSON object values).

[40] http://eric.van-der-vlist.com/blog/2012/08/08/toward-%CF%87%CE%AF%CE%BC%CE%B1%CE%B9%CF%81%CE%B1%CE%BBsuperset/

With these modifications, JSON objects are anonymous elements with attributes and no children nodes and JSON arrays are anonymous elements without attributes and anonymous children element nodes.

χίμαιϱαλ/superset is thus a superset of JSON and XML but unlike UDL there is no frontier between what was previously possible in XML and what was previously possible in JSON and the superset can be used to create new structures that are currently impossible to model in XML or JSON.

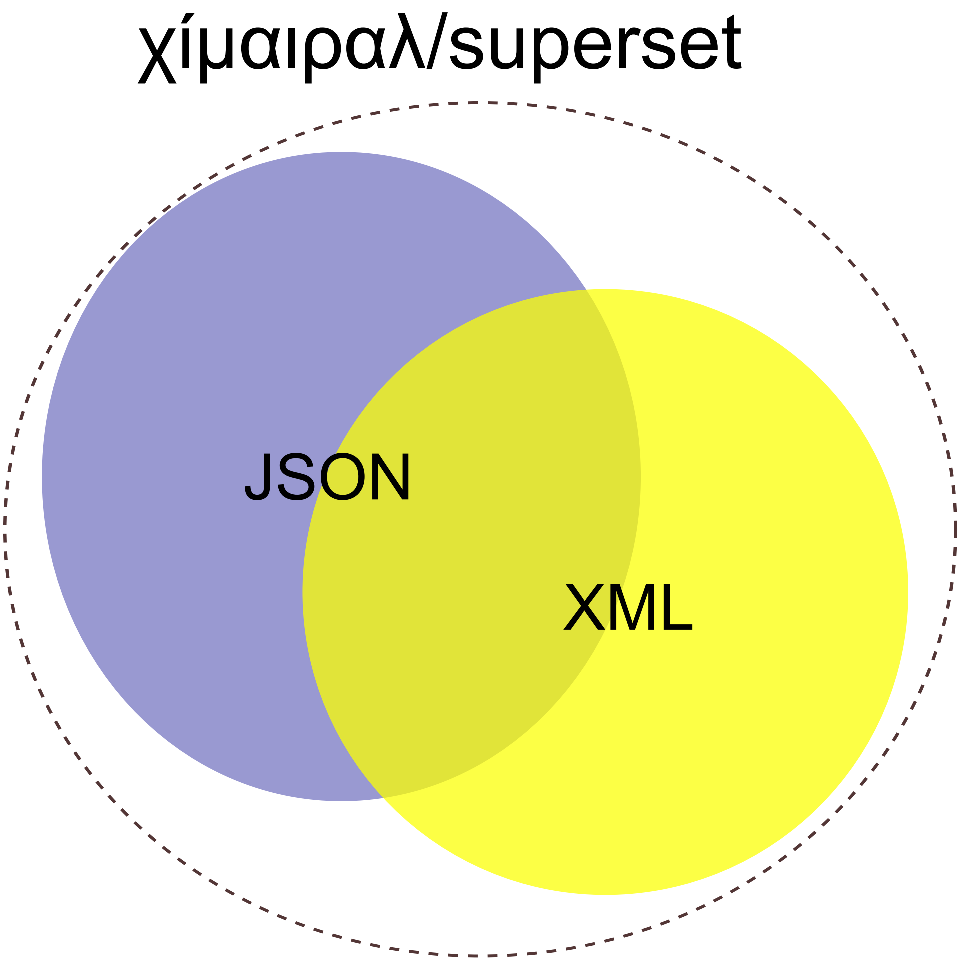


Figure 7. χίμαιϱαλ/superset creates new grounds beyond JSON + XML

One of the useful things that can be done with the new model is that it becomes possible to annotate attributes. How many of us have never dreamed they could do that?

The big issue with these proposals is of course that they are personal initiatives not endorsed by any official organisation and may well keep the status of beautiful utopias!

4. Conclusion

JSON and XML have similar hierarchical data models, JSON being more generic and XML more specialized with markup specific features. They are different enough to rise a number of issues when converting one into the other or just working with both.

Solutions can be split into four categories:

- **Bridges** work in most cases but can get tricky or dirty in corner cases. These difficulties can be reduced when the structure of the documents is known in advance. This is the solution chosen by XForms 2.0 to support JSON.
- **Polyglotism** works in all cases but requires developers (and features or libraries) skilled in both formats. The differences between JSON and XML are exposed and the end result looks more or less like an "ugly chimera". This is the way chosen by XSLT 3.0 and JSONiq to support JSON.
- **Serialization** works in all cases but gives verbose results difficult to read and query. It can be used to persist JSON objects in XML but isn't usually exposed to developers.
- **Consolidation** would be more elegant and really embrace JSON. This would require an updated data model and this approach currently remains an utopia.

XML and RDF Architectural Beastiary

Integration patterns for Documents and Graphs

Charles Greer
MarkLogic Corporation
<cgreer@marklogic.com>

Abstract

This paper presents some architectural patterns for leveraging XML documents and RDF graphs in enterprise database scenarios. Last year you heard about Chimeras. This year we'll craft a few practical beasts that combine two of the web's foundational technologies.

For the time being, document-oriented databases and those that store graphs are separate products; among them are, on the one hand XML databases, and on the other RDF triple stores. As we continue to refine what it means to serve data within post-relational enterprises, we'll find that the XML and RDF communities together understand the underpinnings of a tremendous variety of opportunities as yet unrealized. This paper helps enumerate architectures in which the use of RDF graphs and XML documents intersect.

Keywords: XML, RDF, Data Architecture, Systems Integration

1. Introduction

IT professionals and stakeholders understand that data is in the midst of a sea change. As market-speak would have us believe, we have thus far only learned how to work with "small data" using relational database technologies. Now, data is "big." And while SQL and the relational model may have been great for tightly-controlled and small datasets, the technical mandate behind "Big Data" is to manage petabytes of data and open-ended datasets. The schema-first approach of relational modelling is less compelling as it used to be.

As XML practitioners, we know we've already solved some of the challenges of this new era. XML was one invention that separated data modelling from database products, by providing a method to structure documents independently of their storage mechanism. Early post-relational technologies leveraged XML precisely because it manages structured data separately from schemas. XML provides a method for managing the most complex of human data structures, and XML databases can index these structures and serve them at scale. In the NoSQL vocabulary that has emerged, XML databases fall into the category of "document-orientated databases," which also includes databases that are more JSON-centric, such as [5] and [2].

But XML alone doesn't fit the bill for resilient data structures in the "Big Data" era. While XML databases broke the dependence on relational schemas, RDF triple stores go one step further and require no schema at all to store, aggregate, and query graph data. We've found that this incredibly flexible approach to data structures enables data management techniques at a level of abstraction appropriate for large applications and volatile datasets. Triple stores present a CRUD and query interface to graph data, which enables system-agnostic presentation of object state (RDF), object models (RDFS and OWL), and transformation scenarios (SPARQL queries).

To help frame this new era of data management, I've crafted a beastiary, a collection of imagined metaphoric animals that combine two of the web's foundational technologies. Some versions of these beasts I'm sure exist in the wild already, but as no two IT shops are the same and no two solutions are the same, so it serves us well to consider the various ways in which RDF and XML could be deployed in enterprises. The following "beasts" simply enumerate some architectural patterns for leveraging XML documents and RDF graphs in enterprise IT scenarios.

2. Beastiary

2.1. RDF in an XQuery single-tier architecture

The most familiar XQuery application bundles database and application logic into a single tier. Since XQuery excels both as a data transformation language and as an application platform, we sometimes forget that XQuery can just as easily leverage datasets external to the attached database.

2.1.1. Beast The First - The Consumer

Today's XQuery applications can already work with RDF. Perhaps the least intrusive way to imagine mixing XML and RDF data would be to extend an existing XQuery search application, by adding queries over RDF data, the results of which are incorporated into a user interface.

There are plenty of publicly-accessible SPARQL endpoints by now. SPARQL is a very straightforward language for accessing schema-agnostic objects, and it's suitable both for public endpoints and for enterprise business data services.

Here are just a few data services that you can incorporate into web applications on the internet today, just by mixing in the results of HTTP calls:

- DBPedia[1]. *http://DBpedia.org/sparql* A periodic semantic extraction of wikipedia. Latest one is based on a wikipedia dump from May/June 2012.

[1] http://dbpedia.org

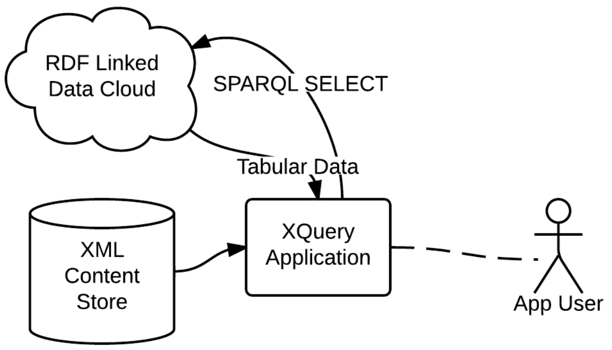


Figure 1. Beast The First: The Consumer

- MusicBrainz[2] *http://dbtune.org/musicbrainz/sparql* An open encyclopedia of music-related artifacts
- FreeBase[3] "An entity graph of people, places and things, built by a community that loves open data. "

This list could extend rather far actually. You'll find a list of public SPARQL end-points with uptime statistics at [SPARQL Endpoint Status]. Collectively this list and its extrapolation comprises the universe of LinkedData -- which is what happens when people begin to use RDF en-mass. Since triples have no impedance between systems that understand RDF, location of data is no longer an access issue, but a database management issue, and your XQuery apps already have all the tools they need to query and store RDF from these public endpoints and the ones which will soon be common in within enterprises.

To query RDF using XQuery, I'll demonstrate GRASP, a small library by Philip Fennell for MarkLogic applications[4] to query SPARQL endpoints:

```
let $qs := "http://dbpedia.org/resource/Linus_Torvalds"
let $result :=
    spq:query-get(
        'http://dbpedia.org/sparql',
        (),
        (),
        'SELECT * {?s ?p ?o. filter (?s = <'||$qs||'>)}',
        'application/sparql-results+xml'
    )
return $result
```

That single `spq:query-get` call contains options for the endpoint (first argument), the SPARQL query (argument 4) and the result mime type, in this case, an XML

[2] http://musicbrainz.org
[3] http://www.freebase.com/
[4]Philip has taken pains to separate MarkLogic implementation from the XQuery 1.0 code for this library, so feel free to use it for other XQuery containers.

format for encoding the results of SPARQL `SELECT` queries. The results of this call are the triples that match the subject URI `<http://dbpedia.org/resource/Linus_Torvalds>`, but as a SELECT, the format of the result is actually a tabular XML format that clearly indicates the bindings requested in the SPARQL query:

```
<response status="200" xmlns="http://www.w3.org/Protocols/rfc2616"><header ▶
name="date" value="Tue, 15 Jan 2013 17:01:45 GMT"/><header name="content-type" ▶
value="application/sparql-results+xml; charset=UTF-8"/><header ▶
name="content-length" value="86428"/><header name="connection" ▶
value="keep-alive"/><header name="server" value="Virtuoso/06.04.3132 (Linux) ▶
x86_64-generic-linux-glibc212-64  VDB"/><header name="accept-ranges" ▶
value="bytes"/><body content-type="application/sparql-results+xml"><sparql ▶
xsi:schemaLocation="http://www.w3.org/2001/sw/DataAccess/rf1/result2.xsd" ▶
xmlns="http://www.w3.org/2005/sparql-results#" xmlns:xsi="http://www.w3.org/▶
2001/XMLSchema-instance">
 <head>
  <variable name="s"/>
  <variable name="p"/>
  <variable name="o"/>
 </head>
 <results distinct="false" ordered="true">
  <result>
   <binding name="s"><uri>http://dbpedia.org/resource/Linus_Torvalds</uri></▶
binding>
   <binding name="p"><uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</▶
uri></binding>
   <binding name="o"><uri>http://www.w3.org/2002/07/owl#Thing</uri></binding>
  </result>
  <result>
   <binding name="s"><uri>http://dbpedia.org/resource/Linus_Torvalds</uri></▶
binding>
   <binding name="p"><uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</▶
uri></binding>
   <binding name="o"><uri>http://xmlns.com/foaf/0.1/Person</uri></binding>
  </result>
  <result>
   <binding name="s"><uri>http://dbpedia.org/resource/Linus_Torvalds</uri></▶
binding>
   <binding name="p"><uri>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</▶
uri></binding>
   <binding name="o"><uri>http://dbpedia.org/ontology/Person</uri></binding>
  </result>
  ...
  </results>
  </body>
  </results>
```

The SPARQL query above has three variables in the select list, and these are translated into the `@name` attribute on each `<binding/>` element. It's simple to see how one could transform this list of values into HTML for display in a browser:

```
let $vars := $result//sparql:head/sparql:variable/@name/data(.)
let $header :=
    <tr>
    {
        for $var in $vars
        return <th>{$var}</th>
    }
    </tr>
let $data :=
    for $tuple in $result//sparql:result
    return
        <tr>
        {
            for $var in $vars
            return
                <td>{$tuple//sparql:binding[@name=$var]//text()}</td>
        }
        </tr>
return
    (
            xdmp:set-response-content-type("text/html"),
            <html>
                <body>
                    <h1>Results</h1>
                    <table>
                    {$header}
                    {$data}
                </table>
            </body>
        </html>
    )
```

This pattern is of course using the XQuery application server as a proxy for the SPARQL endpoint. There are many variations on the basic pattern, but this simple one demonstrates one way in which existing XQuery applications can leverage an enterprise's RDF infrastructure, or the public LinkedData ecosystem [4].

2.1.2. Beast The Second - The Ingester

That result format above is helpful for another beast too. Let's say that the data I got from the external RDF service isn't simply extra information to display, but core business data in and of itself, say, data that contains product metadata from an authoritative RDF data source, along with a "BUY ME NOW" button and link. It might be data I want to keep around for a while.

In this case, the RDF data source is System of Truth for business data, but I want to treat it as native XML for the purposes of my XQuery application. I need to store the XML I've retrieved from another source, and then serve it up as part of the native application.

One could accomplish this by storing RDF triples natively (the result of a CONSTRUCT query) or as an XML transformation of the SPARQL results format.

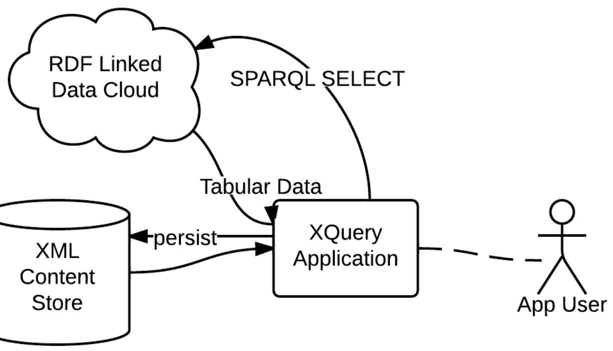


Figure 2. Beast the Second- The Ingester

This strategy has the benefit of incorporating external data into the XQuery application development environment. If you're interested in culling information and serving it as your own, this would be a good pattern. However, it does beg questions of data freshness for your app; you'll want to have a plan in place for updates to the persisted data.

2.1.3. Beast The Third - The Publisher

RDF data stores in the above scenarios are data services; the client issues a SPARQL query over HTTP and processes the results for display. In addition to consumers, a data service can have client publishers. So our XQuery application might also use the SPARQL graph protocol [11]to export data for publication. This third beast is therefore another way for existing XQuery applications to gain value, by publishing data to an external RDF triple store.

The XQuery application in this case is the System of Record for certain business domain objects, but the triple store (in this scenario) is a more effective query tool for enterprise-wise data services. So we pack up domain objects that were modelled in XML and transform them to RDF for publication. The triple store in this scenario may or may not be exposed to the public, so I've replaced the cloud with an honest-to-goodness database that exposes a graph store protocol endpoint. The GRASP library can also be used to export data to RDF stores:

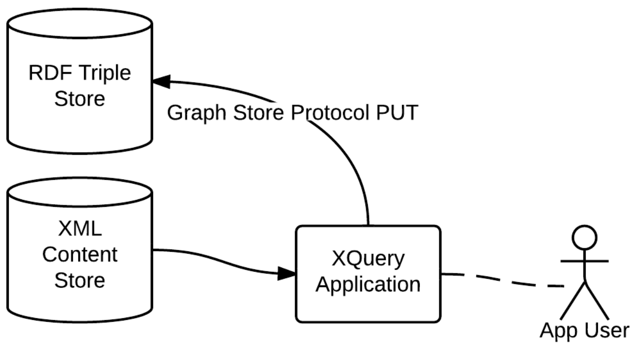


Figure 3. Beast the Third - The publisher

```
xquery version "1.0-ml";

import module namespace spg = "http://www.w3.org/TR/sparql11-protocol/" at
        "lib-spg.xqy";

let $rdf :=
    <rdf:Description @rdf:about="http://dbpedia.org/resource/▶
The_Brothers_Karamazov">
        <dc:title xml:lang="en">The Brothers Karamazov</dc:title>
        <dc:title xml:lang="cs">Bratři Karamazoviv</dc:title>
    </rdf:Description>

return gsp:add-named-graph(
    "http://my-endpoint/sparql",
    "http://dbpedia.org/resource/The_Brothers_Karamazov",
    $rdf)
```

The tricky part for this scenario is creating RDF data from your XML domain objects, if they're complex or if you've not modelled them in an RDF-friendly way. The example above has a dead-simple RDF/XML, but in real situations its the RDF-ification of your data that will pose the most challenge (but also the most reward).

2.2. RDF and XQuery as distinct services

Up to this point we've maintained an XML database-centric view of the intersections of XML and RDF. But few if any enterprise architects encourage XQuery applications as the bread-and-butter application server in the IT shop; other technologies have considerably more adoption for general-purpose integration and development. So the next few beasts concern issues in coordinating among data stores and application servers when XML and RDF live in separate systems, but neither of these systems provides end-user applications.

It's worth noting that, by shifting to an n-tier view, I've temporarily weakened what "XML Database" means, because as we all know, one advantage of an XML database is its flexibility in service creation. So mainly for this section, XML and "Search" are corresponding terms. We use an XML database for search services over large numbers of documents.

Today's RDF and XML architectures all are of the n-tier variety. Many organizations have recognized the power of coupling RDF and XML content, but as of yet there's no integration layer more suitable than the application tier.

2.2.1. Beast the Fourth - The Butcher

If there's some need to make sure that data is synchronized among databases, then a peculiar beast will arise that handles trans-system updates. This beast in my past was known as the "butcher." The butcher was an ETL process that extracted fresh content and System of Record product metadata, transformed to RDF as needed, and then updated both graphs in the RDF database and XML in the document database. Since downstream clients might query either system for current data, any shared data between the two systems had to be updated simultaneously.

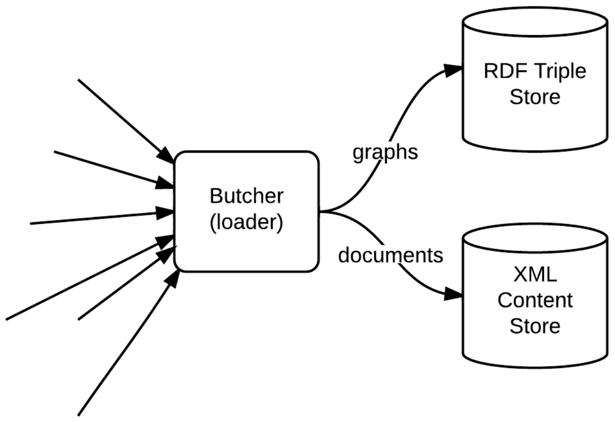


Figure 4. Beast the Fourth - the Butcher

The original "butcher" was implemented in Python. It fetched XML content and RDF/XML data from atompub collections [1], and product metadata from relational databases, and loaded the XML database, the triple store, and also Amazon S3 with coordinated Linked Data.

2.2.2. Beast the Fifth - The URI[5]

There's a little beast that's been hiding among all the other ones in this paper so far -- the URI. It's a miniscule data beast rather than a systems beast. The URI glues all of these scenarios together as the globally unique identifier for a set of data. The most significant distinguishing feature between XML documents and RDF graphs are that graphs are globally identified, intrinsically.

A document's identity is external to the document. That is to say, I cannot know what specific object a document describes unless I know where to find that identifier. From the outside, documents are akin to printed books - I can hold them and store them for later retrieval, but can't match that location to the information in the book without knowing where to find the book's printed metadata inside it.

RDF graphs on the other hand are only constituted by relationships of triples one to another, and every triple has an identifying URI as subject. The data in RDF triples are not subject to external qualification, but rather have a self-contained, internal structure within a global addressing space. In other words, the omnipresent URI in RDF data situates each triple of an RDF dataset within all the world's data.

```
<http://the/most/important/takeaway/is/the/uri>
```

Figure 5. Beast the Fifth - The URI

This exaltation of the URI can find its way into XML databases too, and the data architecture of the enterprise will benefit from it. If we tag a node set with a URI identifier, then we explicitly know how to look up correlating RDF data from a triple store, and hence how the XML document fits into a larger context. The managed URI is a beast that no enterprise will be without once it crosses the line into managed Linked Data.

2.2.3. Beast the Sixth - The n-tier Seeker

Now back to systems.

While we're all familiar with XQuery application servers in combination with an XML database, most enterprises see application servers and databases as distinct, and would more likely consider a beast that consumes both Search services and RDF services as needed.

[5]I know this should technically be the IRI. I think it's safe to say that, at least in my homeland, the typical developer have only an inkling of what I18N is all about, and hence don't tend to understand (yet) what IRI is. We'll just call it Юрий and have done with it.

This beast is the n-tier analogue of the first one. The only difference between a single-tier XQuery scenario and this one is that we've moved the application server out of the XML database container.[6]

An application executes a search against the XML content store and retrieves, say, a result set containing ten books. The metadata provided within each book (including theURI identifying the document) can be used to query a triple store and find out more about the book's context -- who bought it, who wrote it, how to contact the publisher, what retail outlets carry it, and so forth. This beast opens up the world of the data server to search results and allows an application server to enrich search results with RDF Linked Data.

2.2.4. Beast the Seventh - The Asker

The seventh beast is just like the sixth, but inverts the relationship between the RDF and XML databases with regard to normative data. In this case I have a data-heavy application which benefits from incorporating XML assets; one incarnation might be a storefront and personal library for electronic products. The XML data store holds things people want to buy: documents, audio files, video, applications. The RDF system contains product metadata, permissions, purchase histories. Combined, we have fine-grained access to a customer's mixed-content artifacts.

An initial query provides all of the framework for drawing a person's library of electronic book purchases:

```
PREFIX dc: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX : <http://my.organization/>

select ?isbn ?title ?author_name ?upsellLink
where
{
   ?book a :Product;
         dc:title ?title ;
         dc:identifier ?isbn ;
         dc:author ?author .
   ?author foaf:name ?author_name .
   ?special_offer :upsellLink ?upsellLink ;
         :appearsOn ?book .
   :me   :owns ?book .
}
```

[6]This external application could of course be written in XQuery. We love XQuery as an application development language and see no reason why it shouldn't be adopted universally in n-tier architectures.

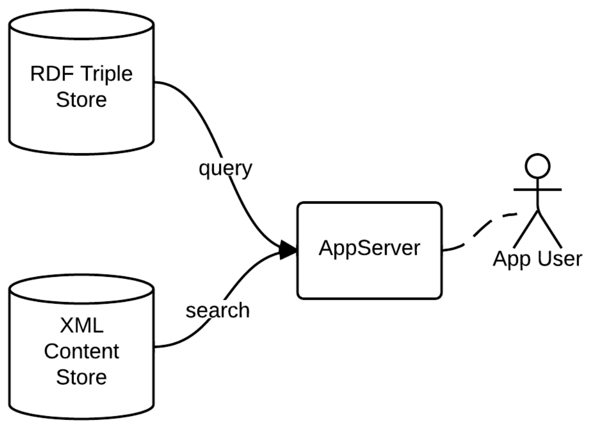


Figure 6. Beast the Sixth - The n-tier Seeker

Once I have a list of books in hand, I can use, say, the ISBN numbers of the books to invoke a lookup service on the XML side and get access to the electronic book text.

2.2.5. Beast the Eighth - Approaching a Monolith

I wonder now whether the application servers in the previous two beasts have to be too smart. Architects like to design data services, and it's easy to envision one which would wrap queries into a search interface. Does the client application need to know all about SPARQL in order to create the storefront? It seems that one search call, if not one data source, should be enough to get all the data needed.

So this beast replaces two separate service invocations with one. Rather than assembling the results of two queries in the application server, I'll construct a search request for all the books, with an embedded constraint that they must be owned by me. Maybe an end-user submits this search string:

```
topic:python published:2010 "list comprehension" sort:relevance
```

The application server could add on one more expression, say "`owned-by:http://uri-for-me`" The search runs on the XQuery backend, which generates a SPARQL query that filters results based on contents of the triple store. This SPARQL query might look like:

```
PREFIX : <http://my.organization/>
select ?book-uri
where
{
   ?book-uri a :Book.
```

```
    :me :owns :Book
}
```

The result is a search across books that I own.

So as in the last beast, this approach assumes that business data are in the triple store, while product artifacts are served from the XML database. But it provides a method to search and run stored business data queries within the same request.

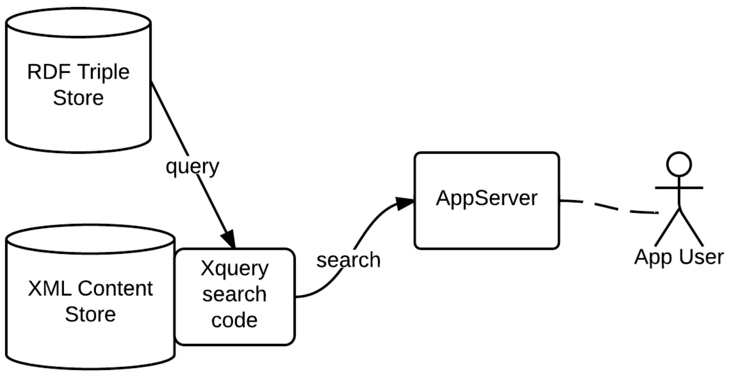


Figure 7. Beast the Eighth: Approaching a Monolith

2.3. A database that stores and queries documents and RDF data together

RDF systems excel at integrating data across multiple sources. True, XML databases also aggregate data; their indexes are aggregating constructs that provide access to the whole corpus of documents based on selected important data points. For example, an XML database that contains invoice records and purchase orders can search and return results from both kinds of documents based on coordinated keys. But RDF goes a step further by allowing derived data to live alongside original data, so that a particular object can be found both with its original vocabulary, but also with a transformed vocabulary better suited for integration.

What if we could dispense with the complex orchestration of "The Butcher" and just load RDF and XML content into the same database? What would that kind of solution look like? Would it simplify the n-tier approach and improve performance of the service integration scenario in "Beast the Eighth?"

2.3.1. Beast the Ninth: A Single Data Repository

Let's store SPARQL query within a search interface and make it work against a single data store that has both triples and XML.

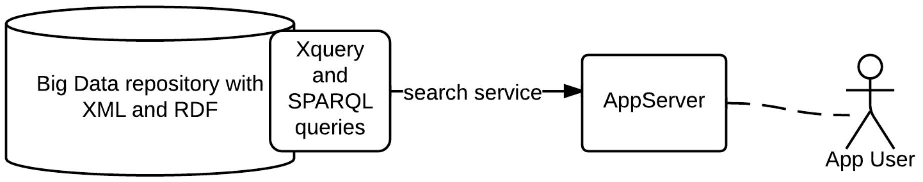

Big Data repository with XML and RDF
Xquery and SPARQL queries
search service
AppServer
App User

Figure 8. Beast the Ninth: A single Data Repository

This beast is remarkably similar to the last one, but assumes that the currently fantastic comes to be, and that we can store and query XML content and RDF together. This approach eliminates the need for complex loading and data management processes, but how might we accomplish the task of XML and RDF data coexisting together in a single database? We'll muse on these issues for the last beasts.

2.3.2. Beast the Tenth: XML Tripler

Now that we're considering a database which stores both content as XML and data as RDF, there are still some interesting options to consider for systems and data architecture. They all boil down to "Where do triples come from?"

Storing triples in XML is not hard; in fact, there are rather too many ways to do it. To create a performant triple store, however, it seems to boil down to storing triples such that the parts of a triple are hard-coded as positional elements, and the text nodes therein are the URIs or literals stored at those positions.

An early experiment for RDF in MarkLogic [9] modeled the triples this way:

```
<t>
  <s>http://dbpedia.org/resource/The_Brothers_Karamazov</s>
  <p>http://www.w3.org/2000/01/rdf-schema#label</p>
  <o xml:lang="en">The Brothers Karamazov</o>
</t>
<t>
  <s>http://dbpedia.org/resource/The_Brothers_Karamazov</s>
  <p>http://www.w3.org/2000/01/rdf-schema#label</p>
  <o xml:lang="cs">Bratři Karamazovi</o>
</t>
```

This approach, while incomplete and experimental,[7] illuminates how some of the simpler SPARQL queries could translate to XPath expressions given this kind of storage strategy. For example, a query for all triples about a particular subject is simply:

```
/t[s eq "http://dbpedia.org/resource/The_Brothers_Karamazov"]
```

[7]It lacks types and has unclear performance characteristics.

This approach provides a potential method for turning an XML database into a triple store, but it doesn't uncover some of the more interesting ramifications of putting triples together with XML.

2.3.3. Beast the Eleventh: The Gleaner

The above beast points out that it's possible to store RDF in queryable XML, but for a hybrid database to make any sense, it should provide some strategy for linking RDF data embedded in XML payloads to the mechanisms that back SPARQL queries.

This last beast then is one that can recognize RDF in XML and expose that RDF in the SPARQL interface. I can think of three useful ways that one might want to send RDF to the hybrid database:

- As RDF. You should be able to ingest common RDF formats and make sure they're indexed as triples. RDF would just go right in. Internally this could be some kind of transform to the hidden XML format.
- As embedded RDF/XML. RDF/XML, or a simplified version thereof, would be a suitable format for tagging RDF metadata while keeping it as maintainable XML. Ideally, the triples embedded in this document would simply be exposed in the hybrid's RDF interfaces:

```
<?xml version="1.0"?>
<View xmlns="http://marklogic.com/cgreer/ramp/arch/" xmlns:rdf="http://▶
www.w3.org/1999/02/22-rdf-syntax-ns#" xmlns:dc="http://purl.org/dc/terms/">
  <rdf:Description rdf:about="http://marklogic.com/cgreer/ramp/arch/Client">
    <dc:title>Pragmatic Semantic System - Client</dc:title>
    <version>0.1-SNAPSHOT</version>
    <dc:creator>Charles Greer</dc:creator>
    <dc:created>2011-10-07</dc:created>
    <dc:type>Component and Connector</dc:type>
  </rdf:Description>
  <overview>The client in the PSS is both a metadata client and a content ▶
client.  The typical scenario has the client querying for RDF data as ▶
metadata to be used within templates, and also the client displaying ▶
transforms of XML content as mapping layers, written documents, and so ▶
forth.</overview>
  <primary src="/images/client.jpg"/>
  <catalog>
    <elements>
</elements>
    <relations>
</relations>
..
</View>
```

The section that starts with the `rdf:Description` tag is valid RDF/XML [8] , which can be parsed by any RDF parser into triples.[8] The element name, plus inclusion of an `@rdf:about` attribute mark it as RDF. The value of `rdf:Description/@rdf:about` denotes the subject of the triples within that element. Each element name within the rdf:Description element are property names, and the values inside those elements are literal object values. In the ideal hybrid database, I'd want to automatically recognize, index, and expose these triples in the RDF corpus.

- Using an RDF-embedding technology like RDFa [RDFa-CORE] to mark up XML content with metadata. This example from the specification shows what properties I'd be able to parse from HTML content, properly marked up with RDFa attributes:

```
<body vocab="http://purl.org/dc/terms/">
   ...
   <div resource="/alice/posts/trouble_with_bob">
      <h2 property="title">The trouble with Bob</h2>
      <p>Date: <span property="created">2011-09-10</span></p>
      <h3 property="creator">Alice</h3>
      ...
   </div>
   ...
   <div resource="/alice/posts/jos_barbecue">
      <h2 property="title">Jo's Barbecue</h2>
      <p>Date: <span property="created">2011-09-14</span></p>
      <h3 property="creator">Eve</h3>
      ...
   </div>
   ...
</body>
```

This example contains a web page and six triples. Three triples are about subject `/alice/posts/trouble_with_bob` (which is relative to base URI of the document) and three about `/alice/posts/jos_barbecue`. The ideal hybrid store would ingest this document and be able to access the triples too.

- Facebook opengraph data. Here is the head of a web page scraped from Best Buy:

```
<!DOCTYPE HTML>
<head>
    <title>Zoom H4n Handy Recorder H4N - Best Buy</title>
```

[8]Arbitrary RDF/XML is terrible for processing with XQuery or other XML-centric tools. There are numerous XML representations of the same RDF graph using this standard, and so queries must address many possible structures in order to do proper semantic queries. There is a proposal in RDF-WG for a simpler RDF/XML that could be more XQuery friendly, but no work has been done in this area yet.

```
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <meta name="keywords" content="ZOOM, H4n Handy Recorder, H4N, Recorders, ▶
Instrument Accessories">
    <meta name="description" content="ZOOM H4n Handy Recorder: 4-channel ▶
simultaneous recording; built-in reference speaker; shock-resistant ▶
rubberized body">
    <meta property="og:title" content="Zoom - H4n Handy Recorder">
    <meta property="og:type" content="product">
    <meta property="og:url" content="http://www.bestbuy.com/site/▶
Zoom+-+H4n+Handy+Recorder/9281957.p?id=1218095840087&amp;skuId=9281957">
    <meta property="og:image" content="http://pisces.bbystatic.com/image2/▶
BestBuy_US/images/products/9281/▶
9281957_sa.jpg;canvasHeight=210;canvasWidth=210">
    <meta property="og:site_name" content="Best Buy">
    <meta property="fb:app_id" content="125188000891129">
    <meta property="og:description" content="ZOOM H4n Handy Recorder: ▶
4-channel simultaneous recording; built-in reference speaker; ▶
shock-resistant rubberized body">
    <meta property="og:upc" content="884354007959">
  ...
```

You can see clearly that facebook's OpenGraph [6] properties took their lead from RDF, and that it would be a simple matter to construct triples from this document. The opengraph parser would have in this case to know that the `og:` prefix corresponds to `http://ogp.me/ns#`.

So a most intriguing data beast would be one that looks something like this:

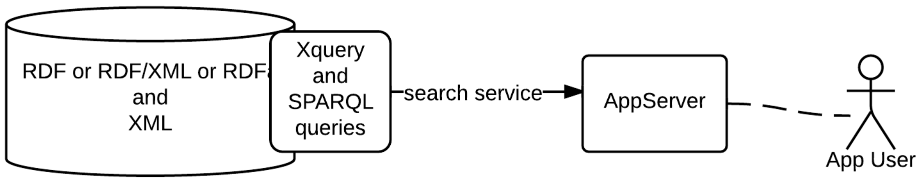


Figure 9. Beast the Eleventh: A Hybrid Database

This final beast represents a powerful new combination of database technologies that could very well be the platform for the next generation of database applications. Access to the database will either be through a search service with semantic extensions, or via SPARQL.[9]

[9]There are of course also hybrid languages like XSPARQL [13] but that's out of scope for today's paper.

3. Wrapping Up

This talk frames the complementary strengths of RDF and XML, to help us focus on how each technology contributes to the whole vocation of managing enterprise IT solutions. While organizations recognize the need for managing large heterogeneous datasets, database practitioners and solutions architects are only beginning to understand when one kind of database or data modelling technique might be used over another (or the old ones), and how to construct systems that decouple and secure enterprise data management from other business processes.

I hope that this beastiary has provided a few approaches that will enable this next generation of database architectures, and look forward to discussing it in Prague.

Bibliography

[1] Atom Publishing Protocol RFC 4287 protocol http://atompub.org/rfc4287.html

[2] http://couchdb.apache.org/

[3] Fennell, Philip GRASP Graph Store and SPARQL Protocol https://github.com/philipfennell/grasp

[4] Linked Data Tim Berners-Lee http://www.w3.org/DesignIssues/LinkedData.html

[5] http://www.mongodb.org/

[6] Open Graph Concepts https://developers.facebook.com/docs/concepts/opengraph/

[7] Shane McCarron; et al. RDFa Core 1.1 Syntax and processing rules for embedding RDF through attributes. 07 June 2012. W3C Recommendation. http://www.w3.org/TR/2012/REC-rdfa-core-20120607/

[8] Dave Beckett. RDF/XML Syntax Specification (Revised). 10 February 2004. W3C Recommendation. URL: http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210

[9] Semantic Storage and Retrieval with MarkLogic Server http://marklogic.github.com/semantic/

[10] http://labs.mondeca.com/sparqlEndpointsStatus/

[11] SPARQL 1.1 Graph Store HTTP Protocol W3C Candidate Recommendation 8 November 2012 http://www.w3.org/TR/sparql11-http-rdf-update/

[12] http://www.w3.org/TR/1998/REC-xml-19980210

[13] Bridging the RDF and XML worlds http://xsparql.deri.org/

XQuery meets SQL

Processing SQL data with XQuery

Rodolfo Ochoa
Oracle
<rodolfo.ochoa@oracle.com>

Luis Rodriguez
Oracle
<luis.g.rodriguez@oracle.com>

Abstract

This paper is intended to illustrate the way that SQL modules work with XQuery in order to provide the ability to query data from a SQL database and insert data to a SQL database. This storage is, at the time of creation of this paper, either SQLite[2] or JDBC[5]. These modules let the user, for example, execute a query, analyze the data, modify it and then send back the results to be stored again in the database.

All the input/output for these modules is managed in JSON format using the JSONiq[1] specification. JSONiq is a small and simple set of extensions to XQuery, allowing XQuery processors to work with JSON and XML natively and to convert data between both formats. JSONiq is implemented in the Zorba[4] XQuery Processor, giving it the option to use pure JSON, pure XML, or both depending on the application's needs.

1. Introduction

Developers need an easy way to interact between the SQL and NoSQL worlds. While it is nice to have a language (XQuery) that can process XML, the reality is that the world's data is mostly stored in SQL databases. So, many developers must use:

- SQL to store their structured data
- XQuery to process any XML data
- An intermediate language bind SQL and XQuery
- Some even XQuery over XML fields from SQL

To solve this problem we proposed a standard way to bind SQL to our XQuery/JSONiq processor, bringing the best of both worlds in a simple solution. In this way developers can:

- Continue storing data on their SQL databases
- Combine the power of SQL and XQuery/JSONiq processing to formulate complex queries over their data
- Avoid the complexity of an intermediate language.

2. SQL for XQuery

In this section we are going to focus on how SQL modules are used. The following examples use the SQLite module, but the API for the JDBC module is very similar.

The SQLite module has the following namespace:

http://www.zorba-xquery.com/modules/sqlite

For a complete list of functions available check Appendix A.

The following sections explain SQL for XQuery by example.

2.1. Querying your tables

For the first example, we are using the most basic methods: `connect()` and `execute-query()`:

```
import module namespace s =
  "http://www.zorba-xquery.com/modules/sqlite";

let $dbConnection := s:connect("D:/small2.db")
let $query := "SELECT * FROM smalltable"
let $result := s:execute-query($dbConnection, $query)

return $result
```

With these 4 lines of code we connect to the database, gather information and display such information.

Connections in the SQL modules are represented by a database key, which is an `xs:anyURI` with an opaque generated value. This key is passed to each other function. Multiple connections to the same or different databases may be open at any time.

All the resulting information will be a series of JSON objects, or an empty sequence in case that the query resulted in a NULL set. Each JSON object will contain key/value pairs where the key is the column name from the result set and the value is the corresponding value. (The mapping from SQLite data types to JSONiq data types is covered later in this paper.) So, for example, the result of the above query might be something like:

```
{ "id" : 1, "name" : "apple", "calories" : 80 }
{ "id" : 2, "name" : "orange", "calories" : 60 }
{ "id" : 3, "name" : "fried egg", "calories" : 92 }
{ "id" : 4, "name" : "cholate milk regular", "calories" : 210 }
```

2.2. Inserting data

In the following example, we insert information into the database through execute-update(). This function will execute a SQL statement that will result in a modification of the connected database.

```
import module namespace s =
  "http://www.zorba-xquery.com/modules/sqlite";

let $dbConnection := s:connect("D:/small2.db")
let $query :="INSERT INTO smalltable (name, calories) VALUES ('carrot', 25)"
let $result := s:execute-update($dbConnection, $query)

return $result
```

Now `$result` will hold the number of rows affected over the execution of the update (just 1 in this case). Note that the module has an auto-disconnect feature, so all connections opened will be closed automatically at the time the module is finished.

2.3. From XML to SQL

An important part of the SQL modules is the ability to use prepared statements in order to optimize time though the use of precompiled SQL statements that can include parameters. The user is able to vary the values assigned to each parameter when the statement is executed without having to recompile each time. The next example shows this functionality:

```
import module namespace s =
  "http://www.zorba-xquery.com/modules/sqlite";

let $xml :=
<root>
  <food><name>carrot</name><calories>25</calories></food>
  <food><name>tomato</name><calories>35</calories></food>
</root>
let $db := s:connect("")
let $inst := s:execute-update($db, "CREATE TABLE smalltable (id INTEGER primary ▶
key asc, name TEXT not null, calories INTEGER)")
let $prep-stmt := s:prepare-statement($db, "INSERT INTO smalltable (name, ▶
calories) VALUES (?, ?)")

for $e in $xml/food
let $name := data($e/name)
let $calories := data($e/calories) cast as xs:integer
return {
  s:set-string($prep-stmt, 1, $name);
  s:set-numeric($prep-stmt, 2, $calories);
  s:execute-update-prepared($prep-stmt)
}
```

In this example we are getting XML (`$xml`) as input. Similarly to connections, prepared statements in the SQL modules are modeled as opaque `xs:anyURI` keys which are passed to other functions. Through the use of a prepared statement (`$prep-stmt`), the parameters are changed between inserts according the data extracted from the XML input, so the database will hold the same information as the XML does. `set-string()` and `set-numeric()` let us change such parameters. In this way, we automatically dump the XML contents to the database.

Note that in this example, `connect()` is being called with an empty string; this tells the SQLite module that we want to use an in-memory database. Also, this example uses Zorba's implementation of XQuery scripting extensions[3] to provide a more natural imperative programming experience, as it is important that the functions be called in a particular order. Finally, the result of this query is not very interesting, as it is just a sequence of "1"s produced by the update statements.

2.4. From SQL to XML

In the next example we use very similar code, but in this case, it does the opposite: creates XML from data gathered from a SQL Database.

```
import module namespace s =
  "http://www.zorba-xquery.com/modules/sqlite";

let $dbConnection := s:connect("D:/small2.db")
let $query := "SELECT id, name, calories FROM smalltable"
let $results := s:execute-query($dbConnection, $query)

for $e in $results
let $id := $e("id")
let $name := $e("name")
let $calories := $e("calories")
return
  <food>
   <id>{$id}</id>
   <name>{$name}</name>
   <calories>{$calories}</calories>
  </food>
```

Note that the notation `$e("id")` is a JSONiq extension to XQuery. It means that, from the JSON object `$e`, we want the value associated with the key `"id"`.

2.5. JDBC for XQuery

In addition to the SQLite module, Zorba also includes a JDBC module. The main advantage of JDBC is that no matter the source, if you have a JDBC connector, then you can get all your data and process it with XQuery/JSONiq. As JDBC is the standard method of connecting to databases in the Java world, virtually any RDBMS software will have a JDBC module. Any data in those databases can now be queried and manipulated using XQuery.

We have designed the SQLite and JDBC modules to have similar APIs, as the example below demonstrates:

```
import module namespace jdbc = "http://www.zorba-xquery.com/modules/jdbc";

let $connectionString := "jdbc:mysql://myhost/▶
human_resources?user=name&password=pass"
let $dbConnection := s:connect($connectionString)
let $query := "SELECT idEmployee, name, address FROM employees"
let $results := s:execute-query($dbConnection, $query)

for $employee in $results
  let $idEmployee := $employee("idEmployee")
  let $name := $employee("name")
  let $address := $employee("address")
return <employee idEmployee="{$idEmployee}">
  <name>{$name}</name>
  <address>{$calories}</address>
</employee>
```

2.6. Executing Prepared Queries

There is one place in which the APIs of the SQLite and JDBC modules differ: `execute-query-prepared()`, which allows you to execute a non-updating prepared query. In the SQLite module, this function directly returns a sequence of JSON objects, in the same way that the non-prepared `execute-query()` function does. However, the JDBC module works a little differently: executing a prepared query results in a DataSet, which is another opaque `xs:anyURI` key. With this key, you can call the `result-set()` function to obtain the actual results. There are also functions in the JDBC module to inquire DataSet-specific metadata. This API difference exists primarily to model the respective APIs of SQLite and JDBC more closely.

2.7. Getting metadata from queries

Through `metadata()` it is possible to get the associated metadata for a given query. The result from this function is a JSON object that contains an array of elements that represent the structure of each one of the columns associated to a specific query. This data is structured as follows:

```
{
  "columns" :
      [{
          "name"          : <column name>,
          "table"         : <table name>,
          "database"      : <database name>,
          "type"          : <type name>,
          "collation"     : [BINARY|NOCASE|RTRIM],
          "nullable"      : [true|false],
          "primary key"   : [true|false],
          "autoincrement" : [true|false]
      }*]
}
```

As an example we have:

```
import module namespace s = "http://www.zorba-xquery.com/modules/sqlite";

variable $db := s:connect("D:/small2.db");
variable $prep-statement := s:prepare-statement($db, "SELECT * FROM smalltable");
variable $meta := s:metadata($prep-statement);
for $i in 1 to jn:size($meta("columns"))
  return $meta("columns")($i}
```

This code will result in metadata for each of the three columns as follows:

```
{ "name" : “id”, "table" : “smalltable”, "database" : “main”, "type" : ▶
“INTEGER”, "collation" : BINARY, "nullable" : false, "primary key" : true, ▶
"autoincrement" : true }
{ "name" : “name”, "table" : “smalltable”, "database" : “main”, "type" : “TEXT”, ▶
"collation" : BINARY, "nullable" : true, "primary key" : false, "autoincrement" ▶
: false }
{  "name" : “calories”, "table" : “smalltable”, "database" : “main”, "type" : ▶
“INTEGER”, "collation" : BINARY, "nullable" : true, "primary key" : false, ▶
"autoincrement" : false }
```

3. Implementation Details

3.1. SQLite module

SQLite module make use of sqlite3 C++ library that is freely available in the sqlite3 homepage. Most of the XQuery functions are implemented with direct calls to its API. A few are not, such as `metadata()` that needs to find out the column name, table name, database name, type name, collation sequence, and associated constraints such as primary key, nullable and auto increment. This information comes from the SQLite API calls `sqlite3_coumn_database_name()`, `sqlite3_column_table_name()`, `sqlite3_column_origin_name()` and `sqlite3_table_column_metadata()`.

3.2. SQLite Data Iterator

Once a connection is successfully established and the desired query is executed, internally Zorba needs a `zorba::Iterator` implementation to handle the way each row is delivered in order to be consumed by the query. This Iterator make use of SQLite storage row-by-row API (`sqlite3_step()`). In this way the memory footprint is minimal since we only keep one row at any given time.

This memory saving does have a drawback: connections can't be closed before all necessary data has been read from the storage. Since XQuery is a functional language, and in particular because Zorba attempts to stream data (consume lazily), it is sometimes difficult to structure the query such that you are sure all data has been consumed before disconnecting. For this reason the current implementation does not allow explicit disconnections. Instead all connections are disconnected by Zorba at the end of query execution, along with the prepared statements and dataset results related to every connection. In practice, having a small footprint overcomes not being able to explicitly close a connection.

3.3. SQLite Data conversion

Once we obtain one row from the storage, we have to map from native data formats to those available for a XQuery/JSONiq query. In the case of SQLite module, having a small and simple set of native formats, it is reduced to the following table:

Table 1.

SQLite Native Type	**XQuery Type**
`SQLITE_NULL`	`jn:null`
`SQLITE_INTEGER`	`xs:Integer`
`SQLITE_FLOAT`	`xs:double`
`SQLITE_BLOB`	`xs:Base64Binary`
`SQLITE_TEXT`	`xs:string`

3.4. SQLite module code size

SQLite module is about 1350 lines of C++ code.

3.5. JDBC module

The JDBC module for Zorba is entirely written in C++. It uses JNI (Java Native Interface) to interact with a JVM (Java Virtual Machine), which Zorba starts and manages. This JVM will interact directly with any JDBC implementation the user provides. The module has nothing written in Java, so there is no extra jar the user needs to add other than the JDBC implementation from the target database.

3.6. JDBC implementation

The JDBC module closely models the way the JDBC classes are organized in Java. It allows obtaining data directly from a query in one step, as well as the preparation of queries with parameters and the use of DataSets (equivalent to ResultSets in JDBC). This approach gives a familiar environment for the user, and also helps the module to use the least amount of memory possible. Also, the DataSet works in such a way that it does not use much memory until the data is gathered from it. To manage large amounts of data we provide functions that release DataSets and Prepared Queries from memory.

3.7. Data type handling

The JDBC module converts data types between JDBC and XQuery/JSONiq as follows:

Table 2.

JDBC Types (java.sql.Types)	XQuery Type
BIGINT - INTEGER - ROWID - TINYINT - BIT - SMALLINT	xs:Integer
DECIMAL - DOUBLE - FLOAT - NUMERIC - REAL	xs:Double
BLOB - BINARY - LONGVARBINARY - VARBINARY - ARRAY - DATALINK - JAVA_OBJECT - OTHER - REF	xs:Base64Binary
CHAR - CLOB - LONGVARCHAR - LONGNVARCHAR - LONGVARBINARY- NCHAR - NCLOB - NVARCHAR - VARCHAR - DATE - TIME - TIMESTAMP	xs:string
NULL	jn:null

3.8. JDBC module code size

JDBC module is about 2200 lines of C++ code.

4. Conclusions

In this paper we have illustrated that Zorba's SQL modules give the ability to perform most of the functions that relational databases allow, in order to extend Zorba and XQuery's storage capabilities. In this way, these modules work as a bridge to communicate across the gap between the XML/JSON world and the Database world. This gap has existed since the beginnings of the XML, so this ability must not be taken lightly. Even though the modules can be extended and polished further, they already demonstrate the feasibility and usefulness of this ability.

References and Acknowledgments

[1] http://www.jsoniq.com/

[2] http://www.sqlite.org/

[3] http://www.w3.org/TR/xquery-sx-10/

[4] http://www.zorba-xquery.com/

[5] http://www.oracle.com/technetwork/java/javase/jdbc/index.html

A. Appendix A - SQLite API

```
module namespace s = "http://www.zorba-xquery.com/modules/sqlite"

(: Connection related functions :)

s:connect($db-name as xs:string ,$options as object()?) as xs:anyURI
(: Creates and returns a connection to a SQLite Database file denoted by ▶
$db-name. the options are passed as a JSON object of the form:
let $options := { "open-read-only" : true, "open-create" : false }
The options available are: open-read-only, open-create, open-no-mutex and ▶
open-shared-cache. :)

s:is-connected($conn as xs:anyURI) as xs:boolean
(: Returns whether or not the passed connection is still connected. :)

(: Transaction related functions :)

s:commit($conn as xs:anyURI) as xs:anyURI
(: Mute function since SQLite has atomic autocommit. :)

s:rollback($conn as xs:anyURI) as xs:anyURI
(: Mute function since if any operation failed an autorollback is executed. :)
(: Simple execution related functions :)

s:execute-query($conn as xs:anyURI, $sqlstr as xs:string) as object()*
(: Executes the SQLite query denoted by $sqlstr under a connection denoted by ▶
$conn. The result is a JSON sequence with the data gathered. :)

s:execute-update($conn as xs:anyURI, $sqlstr as xs:string) as xs:integer
(: Executes the SQLite update denoted by $sqlstr under a connection denoted ▶
by $conn. The result is an integer with the amount of rows affected. :)

(: Prepared Statement related functions :)

s:metadata($pstmnt as xs:anyURI) as object()
(: Returns the metadata associated with the prepared statement as a JSON object. ▶
The result is of the form:
{ columns:
    [{
      “name” : <column name>,
      “table” : <table name>.
      “database” : <database name>,
      “type” : <type name>,
      “collation” : [BINARY|NOCASE|RTRIM],
      “nullable” : [true|false],
      “primary key” : [true|false],
      “autoincrement” : [true|false]
    }*]
}:)

s:prepare-statement($conn as xs:anyURI, $stmnt as xs:string) as xs:anyURI
(: Returns a prepared statement representing the SQL statement $stmnt compiled ▶
against connection $conn. :)

s:set-value($pstmnt as xs:anyURI, $param-num as xs:integer, $val as item()) ▶
as empty-sequence()
(: Sets the value passed by $val in the position $param-num for prepared ▶
statement $pstmt. :)

s:set-boolean($pstmnt as xs:anyURI, $param-num as xs:integer, $val as xs:boolean ▶
) as empty-sequence()
(: Sets the boolean passed by $val in the position $param-num for prepared ▶
statement $pstmt. :)

s:set-numeric($pstmnt as xs:anyURI, $param-num as xs:integer, $val as ▶
xs:anyAtomicType) as empty-sequence()
(: Sets the numeric value passed by $val in the position $param-num for prepared ▶
statement $pstmt. :)
```

```
s:set-string($pstmnt as xs:anyURI, $param-num as xs:integer, $val as xs:string) ▶
as empty-sequence()
(: Sets the string passed by $val in the position $param-num for prepared ▶
statement $pstmt. :)

s:set-null($pstmnt as xs:anyURI, $param-num as xs:integer) as empty-sequence()
(: Sets a null value in the position $param-num for prepared statement $pstmt. :)

s:clear-params($pstmnt as xs:anyURI) as empty-sequence()
(: Clears all values for prepared statement $pstmt. :)

s:close-prepared($pstmnt as xs:anyURI) as empty-sequence()
(: Close and release all resources related to prepared statement $pstmt. :)

s:execute-query-prepared($pstmnt as xs:anyURI) as object()*
(: Executes the prepared statement denoted by $pstmnt. The result is a JSON ▶
sequence with the data gather. :)

s:execute-update-prepared($pstmnt as xs:anyURI) as xs:integer external
(: Executes the prepared statement denoted by $pstmnt. The result is an integer ▶
with the amount of rows affected by $pstmnt. :)
```

B. Apendix B - JDBC API

```
module namespace jdbc = "http://www.zorba-xquery.com/modules/jdbc";
(: CONNECTION :)
(: Opens a connection to a relational database. :)
 jdbc:connect(
                 $connection-config as object() ) as xs:anyURI
(: Opens a connection to a relational database with specified options. )
 jdbc:connect(
                 $connection-config as object(),
                 $options as object()?) as xs:anyURI
(: Closes an open database connection. :)
 jdbc:disconnect(
                 $connection-id as xs:anyURI)
(: Returns true if a connection is connected. :)
 jdbc:is-connected(
                 $connection-id as xs:anyURI) as xs:boolean
(: Returns the options set for a connection. :)
 jdbc:connection-options(
                 $connection-id as xs:anyURI) as object()

(: TRANSACTIONS :)
(: Commit current transaction of a connection. :)
```

```
 jdbc:commit(
                 $connection-id as xs:anyURI) as xs:anyURI
(: Rollback the current transaction of a connection. :)
 jdbc:rollback(
                 $connection-id as xs:anyURI) as xs:anyURI;

(: SIMPLE STATEMENTS :)
(: Executes any kind of SQL statement provided as $sql string. :)
 jdbc:execute(
                 $connection-id as xs:anyURI,
                 $sql as xs:string ) as xs:anyURI
(: Executes any read-only SQL statement provided as $sql string. :)
 jdbc:execute-query(
                $connection-id as xs:anyURI, $sql as xs:string) as object()*
(: Executes deterministic read-only SQL statements provided as $sql string. :)
 jdbc:execute-query-deterministic(
                 $connection-id as xs:anyURI,
                 $sql as xs:string ) as object()*
(: Executes only updating SQL statements provided as $sql string. :)
 jdbc:execute-update(
                 $connection-id as xs:anyURI,
                 $sql as xs:string) as xs:integer

(: PREPARED STATEMENTS :)
(: Creates a prepared statement for multiple executions. :)
 jdbc:prepare-statement(
                 $connection-id as xs:anyURI,
                 $sql as xs:string) as xs:anyURI
(: Set the value of the designated parameter with the given value. :)
 jdbc:set-numeric(
                 $prepared-statement as xs:anyURI,
                 $parameter-index as xs:integer,
                 $value as xs:anyAtomicType)
(: Set the value of the designated parameter with the given value. :)
 jdbc:set-string(
                 $prepared-statement as xs:anyURI,
                 $parameter-index as xs:integer,
                 $value as xs:string)
(: Set the value of the designated parameter with the given value. :)
 jdbc:set-boolean(
                 $prepared-statement as xs:anyURI,
                 $parameter-index as xs:integer,
                 $value as xs:boolean)
(: Set the value of the designated parameter with the given value. :)
 jdbc:set-null(
                 $prepared-statement as xs:anyURI,
                 $parameter-index as xs:integer)
(: Clears the current parameter values immediately. :)
 jdbc:clear-params(
                 $prepared-statement as xs:anyURI)
(: Retrieves the number, types and properties of the prepared statement ▶
parameters. :)
 jdbc:parameter-metadata(
                 $prepared-statement as xs:anyURI) as object()
(: Executes SQL statements prepared with 5.1 jsql:prepare-statement. :)
 jdbc:execute-prepared(
                 $prepared-statement as xs:anyURI) as xs:anyURI
(: Executes a non-updating SQL statement prepared with 5.1 ▶
jsql:prepare-statement. :)
 jdbc:execute-query-prepared(
                 $prepared-statement as xs:anyURI) as object()*
(: Executes an updating SQL statement prepared with 5.1 jsql:prepare-statement. ▶
:)
 jdbc:execute-update-prepared(
                 $prepared-statement as xs:anyURI) as xs:integer

(: DATASETS :)
(: This function returns a sequence of objects representing the rows of data
   from a non-updating query. :)
 jdbc:result-set(
                 $dataset-id as xs:anyURI) as object()*
(: Return the metadata of the result of a particular DataSet. :)
 jdbc:metadata(
                 $dataset-id as xs:anyURI) as object()
(: Return the number of affected rows of a particular DataSet. :)
 jdbc:affected-rows(
                 $dataset-id as xs:anyURI) as xs:integer
```

XQuery Development in the Cloud(9)

XQuery code anywhere, anytime.

William Candillon
28msec
<william.candillon@28msec.com>

Abstract

XQuery is a double-edged sword: On one side, it enables building extremely powerful data-intensive applications. On the other side, the entry-barrier to the technology is extremely high. This puts a tremendous amount of pressure on the development tooling. Tools need to have a deep semantic knowledge of the language while still being widely accessible. Our goal is to contribute a new XQuery development toolkit that is entirely browser-based and heavily relies on static code analysis in order to provide a rich editing experience.

Keywords: XQuery, Static code analysis, IDE, Cloud9

1. Introduction

When it comes to XQuery development, there are currently two kinds of tooling available: desktop-based and browser-based IDEs.

Desktop-based IDEs such as XQDT[1] or Oxygen[2] provide a rich editing experience based on static analysis of the source code. For instance, this analysis is used for semantic code highlighting, code completion, module outlines, and debugger support.

Browser-based IDEs such as eXide[3] enable XQuery developers to work entirely from the browser with no installation or setup required. The code is available anywhere at anytime. We see eXide as being a great proof of concept that the future of XQuery development will be browser-based. However, in order to enjoy the benefits of a browser-based experience from eXide, developers have to make two compromises:

- The development support is solely based on the analysis of source code as text and there is no deeper semantic analysis. The lack of static analysis very often leads to faulty code highlighting and irrelevant code completion proposals to the developer.
- eXide does not provide the usual expected features of a traditional desktop IDE. There is neither a plugin ecosystem nor integration with third-party toolkits. It also lacks of the following features: code versioning, terminal access, rich keyboard shortcuts, and integration with other programming languages.

In order to overcome these limitations, we have decided to contribute a full-fledged static XQuery code analyzer written in JavaScript to what is arguably the best open source browser-based IDE available on the market: Cloud9[4].

The positioning of our contribution is explained in Figure 1. Clearly, desktop-based IDEs have the richest level of features available for XQuery code editing. However these tools have a high entry barrier. Oxygen, for instance, is a commercial product that requires a complex setup in order to be integrated with a developer's XQuery toolchain. On the other side of the spectrum, browser-based IDEs are shattering traditional setup requirements: all you need is a browser. Unfortunately, the level of features provided by this type of tooling is far below to what a developer would expect from desktop-based IDEs. This observation is driving our contribution: we want to bring rich XQuery editing features to a full-fledged browser-based IDE.

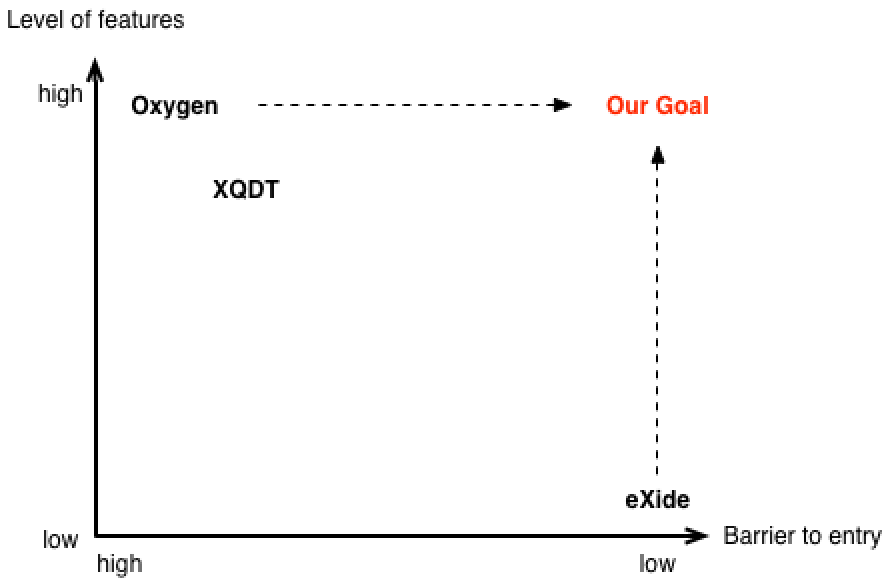


Figure 1. Positioning of our contribution.

The remainder of this paper is structured as follows. In Section 2, we describe the requirements and use-cases that we identified in order to contribute a new XQuery development toolkit. In Sections 3 and 4, we present the architecture of our approach and how we used it to implement our use cases. Furthermore, we present in Section 5 a benchmark of our static XQuery code analyzer written in JavaScript. Section 6 gives an outlook on future work.

2. Requirements & Use Cases

In order to contribute a new XQuery development toolkit, we identified the following five requirements and use cases.

1. **Browser Access & Plugins (see 4.1).** The XQuery toolkit needs to be available via the browser with no installation required. Moreover, it should easily integrate with third-party development tools such as git. Additionally, XQuery and XML developers should be able to extend the development environment to support their own tools by writing plugins.
2. **Semantic Highlighting. (see 4.2)** Syntax highlighting in the editor should be based on static code analysis, not just plain text. For instance, in the code snippet `(return, let)`, the words `return` and `let` need to be colored as NCNames, not keywords.
3. **Semantic Completion. (see 4.3)** Code completion needs to leverage the semantic context from which it is invoked in order to propose the most relevant possible completions.
4. **Code Navigation. (see 4.4)** Developers should have fast access to the outline of their code and be able to jump to the definition of a referenced variable or function.
5. **Quality Checks. (see 4.5)** The editor should display compiler errors as you type and provide warnings in order to improve code quality (e.g. unused variables or imports).

3. Architecture

In order to implement the use cases described in Section 2, we rely on two components: XQLint[5], an XQuery code analyzer written in JavaScript, and Cloud9, an open source browser-based IDE.

3.1. XQLint

XQLint is an open source XQuery static code analyzer written in JavaScript. It can be executed in a web browser or using node.js. The parser component of the analyzer supports the following XQuery specifications: XQuery 3.0[6], XQuery Update Facility 1.0[7], Full-Text 1.0[8], JSONiq[9], Zorba Scripting Extension[10], and the Zorba Data Definition Facility[11]. In order to ensure the compliance of the JavaScript parser, XQLint is tested against the same test suite as the one from the Zorba XQuery processor[12].

The parser is generated using the REx parser generator[13]. REx has two key benefits. Firstly, it uses the same grammar input format as the one used in W3C specifications: EBNF. This enables XQLint to perform code analysis while staying close to the specification documents. Secondly, REx provides good performance. A benchmark using the Zorba test suite has shown that the REx generated parser is about 500% faster than the equivalent parser generated with ANTLR (see Section 5).

On top of the abstract syntax tree (AST) generated by the parser, XQLint provides two visitors: *SyntaxHighlighter* and *Translator*. The *SyntaxHighlighter* visitor is responsible for providing semantic highlighting based on the AST. The code snippet below describes how this visitor can be used for code highlighting. In the first step, we parse the source code using the REx generated parser. The AST is visited by the SyntaxHighlighter object, which then returns a tokenized version of the source code.

```
var source = "(let, return)";
var parser = new XQueryParser(source);
parser.parse_XQuery();
var ast = h.getAST();
var highlighter = new SyntaxHighlighter(ast);
var tokens = highlighter.getTokens();
console.log(tokens);
// Returns [{ type: "text", value: "(" }, { type: "ncname", value: "let" } ...]
```

The Translator visitor is responsible for performing static code analysis. It pseudo-compiles the source code to produce a static context. During this analysis, static errors might be raised. In addition, the translator reports warnings for unused variables and unused namespace bindings. The code snippet below shows how it can be invoked. Similar to the *SyntaxHighlighter* visitor, we transform the source code into an AST that will then be visited by the Translator object using the `translate()` method. `translate()` returns a static context that can be introspected to provide deep semantic insights.

```
var source = "(:~ Test function :)\n" +
"declare function local:test($arg){ 1 };\n" +
" local:test(1)";
var parser = new XQueryParser(source);
ast = h.getParseTree();
var translator = new Translator(ast);
sctx = translator.translate();
sctx->getMarkers() //returns warning: unused variable $arg
sctx->getFunctions() //returns local:test with associated XQDoc
```

3.2. Cloud9

Cloud9 IDE is an open source browser-based IDE that supports several programming languages but mainly JavaScript. It is written in JavaScript, and uses Node.js on the backend. The architecture of Cloud9 is built such that the IDE can be easily extended by third-party plugins. For example, such plugins provide the following features:

- Integration with ssh, ftp, and git
- Code search and navigation
- Cloud deployment to third-party services

- VI and Emacs keyboard bindings
- Code folding and multiple editing cursors
- Customized graphical themes and "zen mode" editing

More importantly, Cloud9 provides a rich language framework that enables developers to write plugins to support their own programming language. In order to perform heavy computations in the background without blocking the UI, the language framework "lives" in a web worker[14]. This language framework provides the following interface:

- `complete(doc, fullAst, pos, currentNode)`. This function is invoked when the user requests code completion. `complete()` receives as parameters the document being edited, the entire AST of the current file, the current cursor position, and the AST node found at the cursor position. As a result of this function, an array of completion proposals is expected.
- `analyze(doc, fullAst)`. Triggers the static analysis of the AST. This function is expected to return an array of markers. A marker can be an error, a warning, or a simple notice. The list of the different marker types that are available can be extended.
- `getVariablePositions(doc, fullAst, pos, currentNode)`. This function is invoked when variable renaming is activated. This function is expected to return an object containing the main variable occurrence and an array of all occurrences of the selected element.
- `codeFormat(doc)`. This function is invoked when code formatting is requested. As a result, this function returns a formatted version of the source code.
- `jumpToDefinition(doc, fullAst, pos, currentNode)`. This function must return the position of the definition of the currently selected element.
- `outline(doc, fullAst)`. This function is invoked when the user requests the outline. It is expected to return an outline structure of the source code as JSON.

We implemented all the functions provided by the language framework interface in order to provide solid XQuery support in Cloud9 (Fig 2).

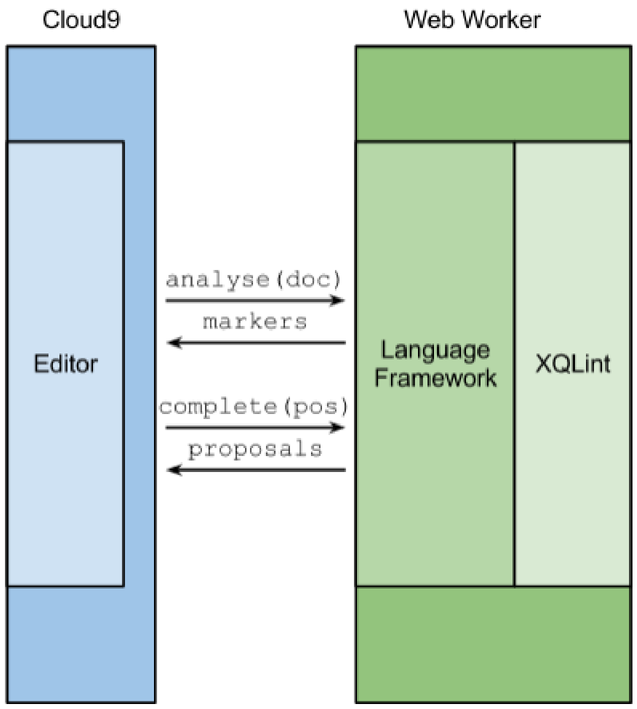


Figure 2. Integration of XQLint with Cloud9 via the language framework.

4. The Toolkit in Action

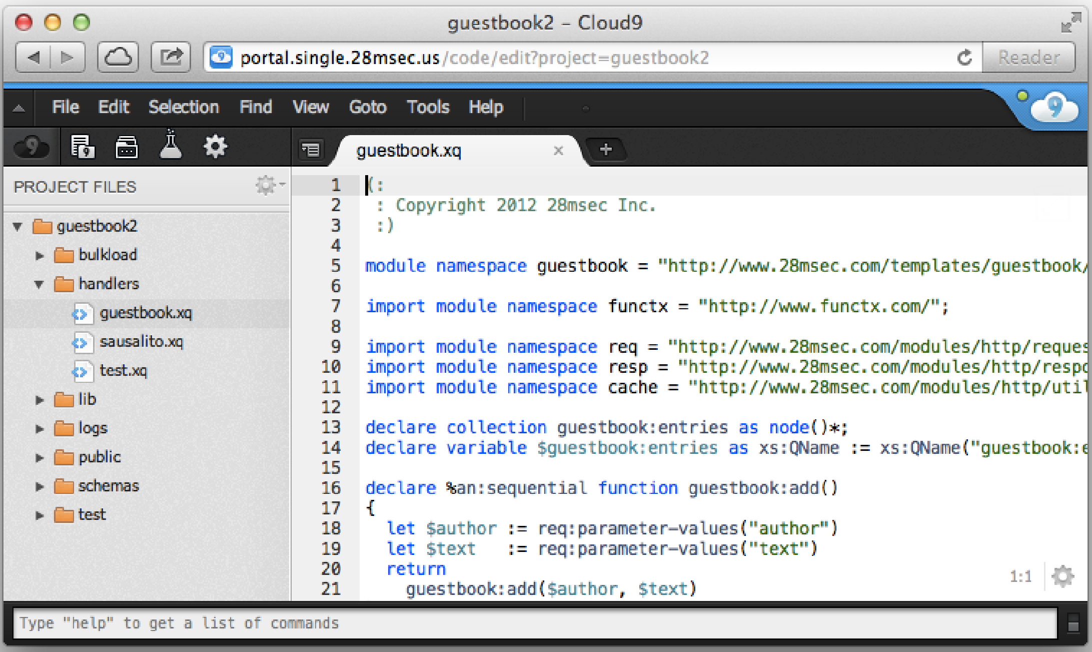
The goal of our contribution is to integrate the static analysis capabilities of XQLint into the Cloud9 language framework. In this section, we describe how each of the use cases described in Section 2 have been implemented using this approach.

4.1. Browser Access

Cloud9 is our web IDE of choice. Its server backend is built on top of node.js. The communication between the client and the server is done via two network protocols: WebDAV[16] and engine.io[15]. WebDAV is used for storing and editing files in the IDE workspace. Engine.io is used for realtime communication between the IDE and the server.

In order for XQuery vendors to easily integrate Cloud9 in their own infrastructure, we have contributed a backend implementation of Cloud9 written in XQuery. This backend, codenamed *Cloud9.xq*, relies on an implementation of the WebDAV and engine.io protocol which are entirely written in XQuery.

Figure 3 shows Cloud9 running directly on top of 28msec's XQuery application server.


guestbook2 – Cloud9
portal.single.28msec.us/code/edit?project=guestbook2
Reader
File Edit Selection Find View Goto Tools Help
PROJECT FILES
guestbook2
bulkload
handlers
guestbook.xq
sausalito.xq
test.xq
lib
logs
public
schemas
test
guestbook.xq
1:1
Type "help" to get a list of commands

```
(:
 : Copyright 2012 28msec Inc.
 :)

module namespace guestbook = "http://www.28msec.com/templates/guestbook/

import module namespace functx = "http://www.functx.com/";

import module namespace req = "http://www.28msec.com/modules/http/reques
import module namespace resp = "http://www.28msec.com/modules/http/respo
import module namespace cache = "http://www.28msec.com/modules/http/util

declare collection guestbook:entries as node()*;
declare variable $guestbook:entries as xs:QName := xs:QName("guestbook:e

declare %an:sequential function guestbook:add()
{
  let $author := req:parameter-values("author")
  let $text   := req:parameter-values("text")
  return
    guestbook:add($author, $text)
```

Figure 3. Cloud9.xq: Full XQuery backend for Cloud9

4.2. Semantic Highlighting

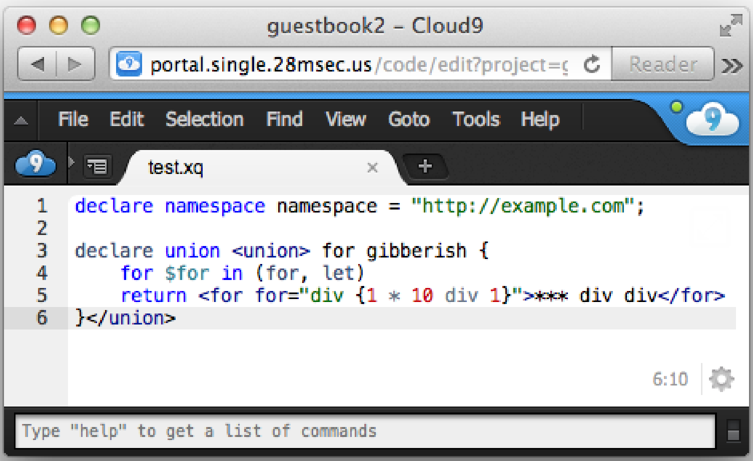
For each keystroke in the XQuery code editor, the AST is computed in a web worker in order for the operation to be non-blocking to the developer. Once the AST is built, the *SyntaxHighlighter* visitor computes the tokens to highlight and sends the result back to the editor. The editor caches each such token.

The code snippet in Figure 4 demonstrates the power of the semantic highlighter. It contains an unlikely, but legal, XQuery program. There is a couple of interesting things to note about the way the code is highlighted. For instance, each occurrence of the word namespace is correctly highlighted as a keyword or an NCname according to the semantic of the program. The same thing goes for occurrences of the word div which is rightfully highlighted as attribute string, operator, or text node.

4.3. Semantic Completion

When code completion is triggered from the editor, the completion proposals are filtered according to the context from which the completion is requested.

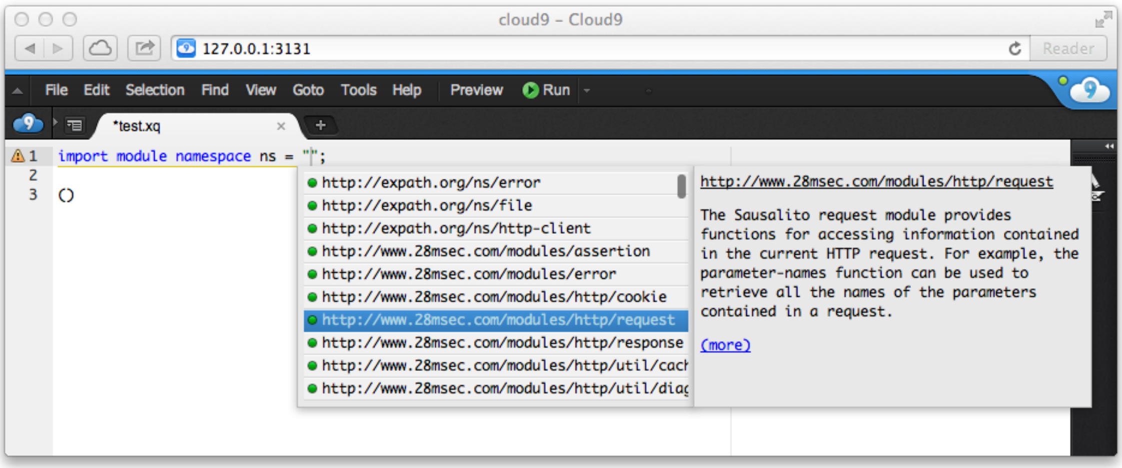
In Figure 5, the developer triggers code completion in the context of an import module declaration. Completion proposals therefore contain potential namespace URIs of modules to import including their documentation.


guestbook2 – Cloud9
portal.single.28msec.us/code/edit?project=
Reader
File Edit Selection Find View Goto Tools Help
test.xq
6:10

```
declare namespace namespace = "http://example.com";

declare union <union> for gibberish {
    for $for in (for, let)
    return <for for="div {1 * 10 div 1}">*** div div</for>
}</union>
```

```
Type "help" to get a list of commands
```

Figure 4. XQuery Semantic Highlighting



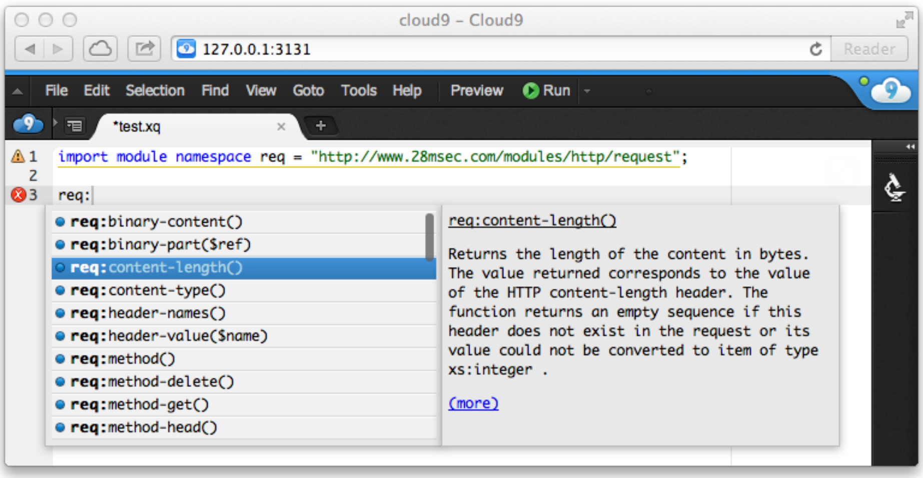
```
import module namespace ns = "";

()
```

```
http://expath.org/ns/error
http://expath.org/ns/file
http://expath.org/ns/http-client
http://www.28msec.com/modules/assertion
http://www.28msec.com/modules/error
http://www.28msec.com/modules/http/cookie
http://www.28msec.com/modules/http/request
http://www.28msec.com/modules/http/response
http://www.28msec.com/modules/http/util/cach
http://www.28msec.com/modules/http/util/diag
```

```
http://www.28msec.com/modules/http/request

The Sausalito request module provides
functions for accessing information contained
in the current HTTP request. For example, the
parameter-names function can be used to
retrieve all the names of the parameters
contained in a request.

(more)
```

Figure 5. Namespace Completion

The code completion also takes advantage of XQLint to request known namespace bindings in the source code. In Figure 6, the code completion correctly links the `req` prefix to its namespace, therefore proposing functions from the `http://www.28msec.com/modules/http/request` module.



```
import module namespace req = "http://www.28msec.com/modules/http/request";

req:
```

```
req:binary-content()
req:binary-part($ref)
req:content-length()
req:content-type()
req:header-names()
req:header-value($name)
req:method()
req:method-delete()
req:method-get()
req:method-head()
```

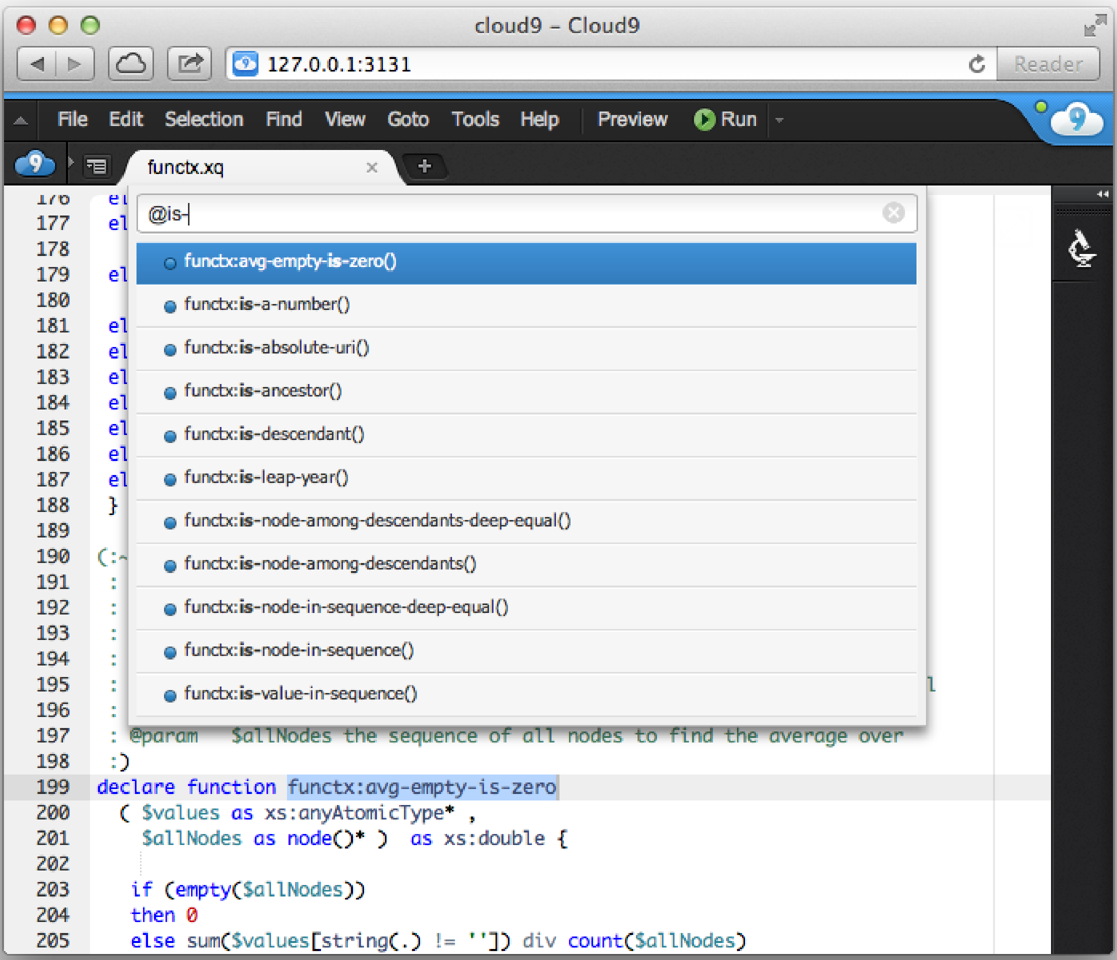
```
req:content-length()

Returns the length of the content in bytes.
The value returned corresponds to the value
of the HTTP content-length header. The
function returns an empty sequence if this
header does not exist in the request or its
value could not be converted to item of type
xs:integer .

(more)
```

Figure 6. Function Completion

The XQuery code completion framework has access to the static information and documentation content of all built-in modules that are provided by the XQuery processor (Zorba in this case). Moreover, in order to provide static information about all user-defined XQuery modules, the developer's workspace is continuously indexed in the background. The integration of other built-in modules is easily possible in order to allow extensions from other XQuery vendors. This can be done by feeding a JSON file that contains all static information about the desired library modules to XQLint.

4.4. Code Navigation

To navigate in a module outline, we simply use the computed static context from XQLint in order to get the list of declared variables and functions with their exact positions in the source code. Figure 7 shows the feature in action.



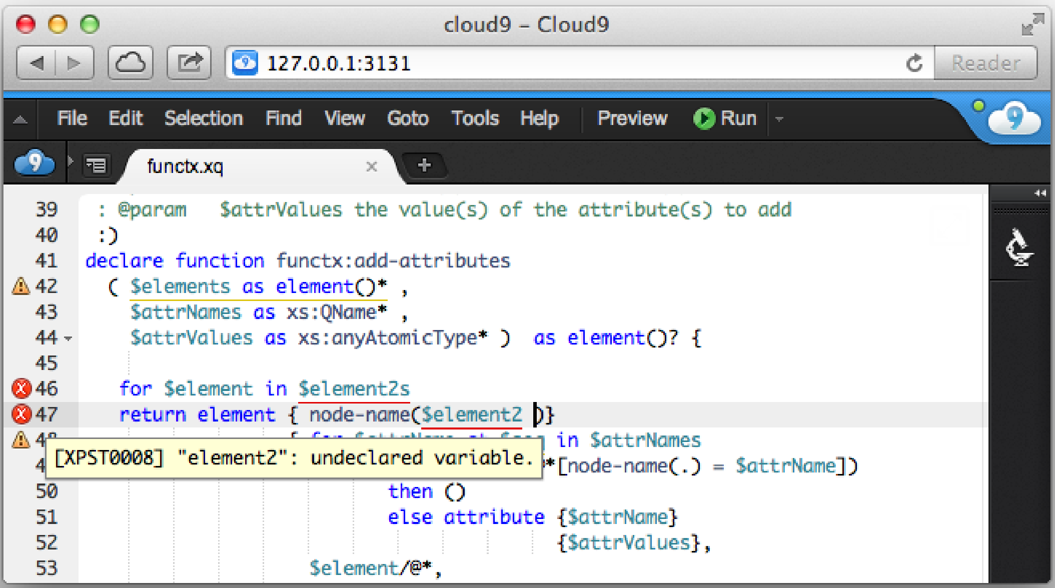
```
@is-
functx:avg-empty-is-zero()
functx:is-a-number()
functx:is-absolute-uri()
functx:is-ancestor()
functx:is-descendant()
functx:is-leap-year()
functx:is-node-among-descendants-deep-equal()
functx:is-node-among-descendants()
functx:is-node-in-sequence-deep-equal()
functx:is-node-in-sequence()
functx:is-value-in-sequence()
```

```
: @param   $allNodes the sequence of all nodes to find the average over
:)
declare function functx:avg-empty-is-zero
  ( $values as xs:anyAtomicType* ,
    $allNodes as node()* )  as xs:double {

   if (empty($allNodes))
   then 0
   else sum($values[string(.) != '']) div count($allNodes)
```

Figure 7. Module Outline

4.5. Code Quality Checks

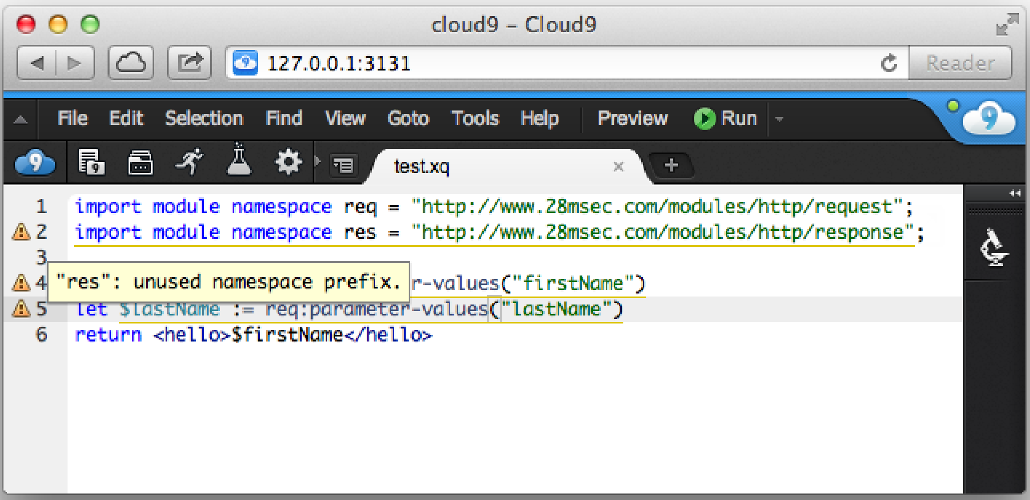
XQLint pseudo-compiles XQuery code to produce a static context. Where most XQuery processors return only one compilation error at a time, XQLint is capable of raising multiple errors at once. This level of responsiveness is key in order to improve the productivity of XQuery developers. Figure 8 shows how multiple compilation errors are displayed to the developer.



```
 : @param   $attrValues the value(s) of the attribute(s) to add
 :)
declare function functx:add-attributes
  ( $elements as element()* ,
    $attrNames as xs:QName* ,
    $attrValues as xs:anyAtomicType* )  as element()? {

    for $element in $element2s
    return element { node-name($element2 )}
                     { for $attrName at $pos in $attrNames
[XPST0008] "element2": undeclared variable.  *[node-name(.) = $attrName])
                            then ()
                            else attribute {$attrName}
                                           {$attrValues},
                       $element/@*,
```

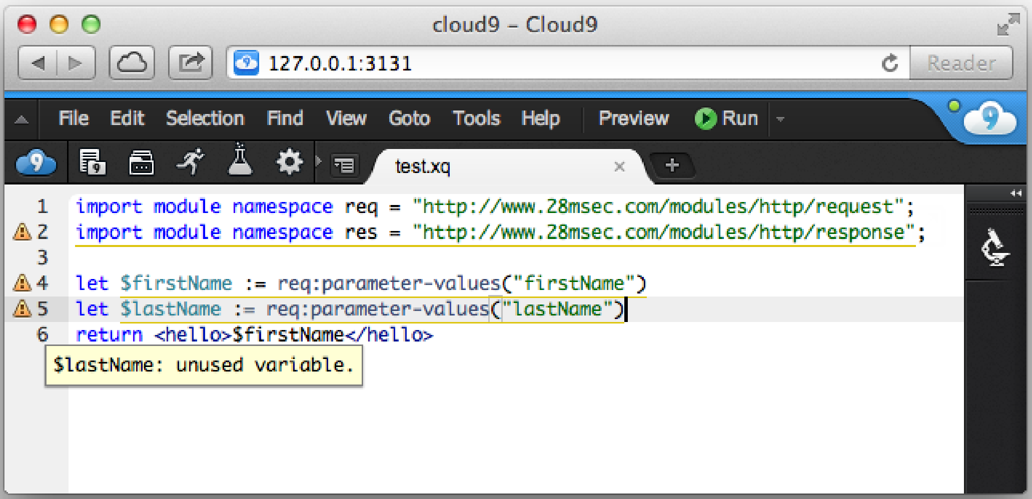
Figure 8. Reporting of multiple static errors

On top of compilation errors, XQLint also provides warnings for unused variables, functions and namespace bindings, as shown in the figures 9 and 10.



```
import module namespace req = "http://www.28msec.com/modules/http/request";
import module namespace res = "http://www.28msec.com/modules/http/response";

"res": unused namespace prefix. r-values("firstName")
let $lastName := req:parameter-values("lastName")
return <hello>$firstName</hello>
```

Figure 9. Unused namespace prefix



```
import module namespace req = "http://www.28msec.com/modules/http/request";
import module namespace res = "http://www.28msec.com/modules/http/response";

let $firstName := req:parameter-values("firstName")
let $lastName := req:parameter-values("lastName")
return <hello>$firstName</hello>
```

```
$lastName: unused variable.
```

Figure 10. Unused variable

5. Performance

When integrated with Cloud9, XQLint performs a lot of computationally demanding tasks such as parsing and pseudo-compiling the source code. In order to provide a smooth and responsive user experience, the performance of the toolkit needs to be considered.

To perform XQuery parsing using JavaScript, we considered two possible parser generators: REx and ANTLR. We benchmarked the two generated parsers using the functx module[17] and also compared the results to the parsing time of Zorba in order to get an idea of the penalty added by using JavaScript. Figure 11 shows the average parsing time when parsing functx 100 times using ANTLR, REx, and Zorba. This benchmark motivated us to use REx over ANTLR.

We also benchmarked the time XQLint takes to pseudo-compile XQuery code. With this benchmark, we tried to answer the following question: when a developer opens an XQuery file, how much time does it take for XQLint to take the module and make its outline and code analysis available in the editor? Figure 12 shows the average pseudo-compilation time of XQLint on the Zorba test suite. Because we consider it as very unlikely for a random XQuery module to be larger than the functx module, we also benchmarked the average pseudo-compilation time of the functx module (figure 13).

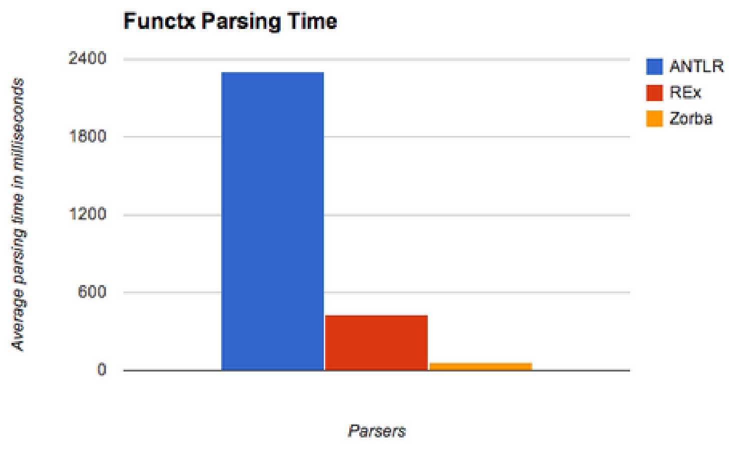


Figure 11. Functx parsing time in milliseconds

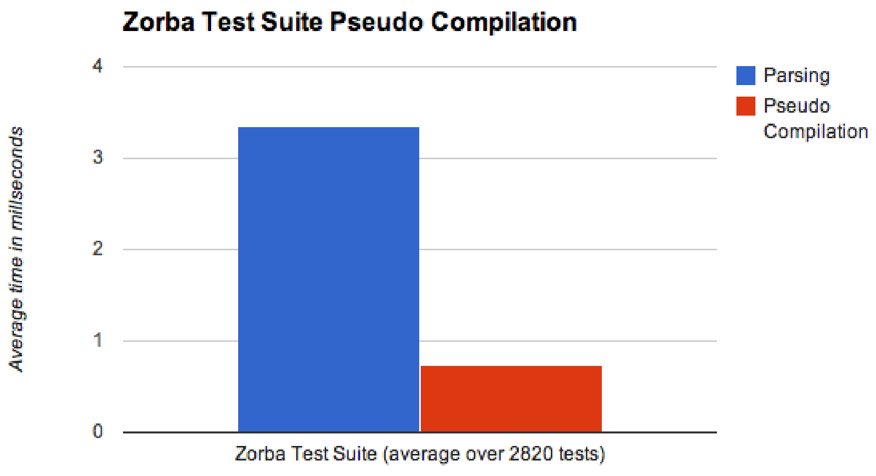


Figure 12. Zorba test suite pseudo-compilation time in milliseconds

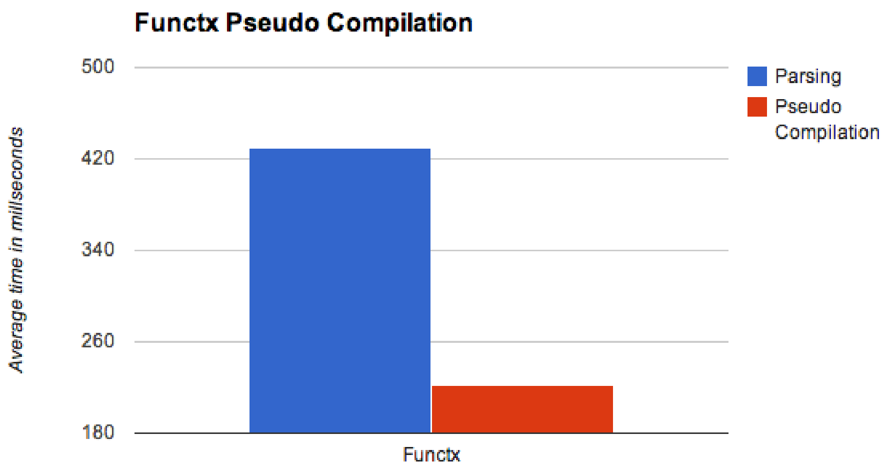


Figure 13. Functx pseudo-compilation time in milliseconds

6. Conclusion & Outlook

Development toolkits have a key role in the XQuery ecosystem. They are responsible for empowering developer's productivity (and fun!) and lowering the entry barrier to XQuery-related technologies.

In this paper, we introduced a new XQuery toolkit by integrating an XQuery static code analyzer written in JavaScript (XQLint) with a fully-fledged browser-based IDE, namely Cloud9. In order to bring a rich XQuery development experience to the browser, we identified and implemented five use cases from the everyday life of a typical XQuery developer.

The Semantic highlighter provides sharp XQuery code highlighting based on the information of an AST. This is the first browser-based XQuery semantic highlighter we are aware of. XQuery modules are automatically pseudo-compiled in the browser using XQLint. XQLint produces a static context given an XQuery source code as an input. This static context is used to build rich editing features. Code completion leverages this information to provide completion proposals that are highly relevant to the developer. Code navigation is also implemented using information from the static context, enabling developers to see their module outline and jump to variables or functions declarations. Last but not least, we use XQLint to report warnings allowing the developer to improve the quality of the code.

In order to make Cloud9 more accessible to the XQuery community, we introduced *Cloud9.xq*, a server backend for Cloud9 entirely written in XQuery. This implementation relies on two network protocols: WebDAV and engine.io.

The different components that were introduced in this paper (*Cloud9*, *xqlint*, and *Cloud9.xq*) are all open-source and free for the community to grab. Coincidentally, eXide and Cloud9 are using the same editor component: ACE. This means that the eXide project can directly benefit from our contributions.

This is just the beginning. We are looking to build a richer set of XQuery editing features into Cloud9. XQLint will provide more and more complex quality checks. Quick fixes, which are proposals for automatic error and warning resolution, are currently being developed. Integration with XQuery debuggers is a feature on our roadmap. We are also discussing adding type inference capabilities into XQLint.

Finally, we are looking forward to see XML and XQuery developers to extend this development environment and integrate it with their own tools such as the RESTful XQuery framework, EXPath packaging, XSpec, and more.

Bibliography

[1] Eclipse Foundation: XQuery Development Toolkit. http://www.xqdt.org

[2] SyncRO Soft SRL: Oxygen XML Editor. http://www.oxygenxml.com/

[3] eXist-db: eXide, a web-based XQuery IDE. http://atomic.exist-db.org/blogs/eXist/eXidePart1

[4] Cloud 9 IDE. http://c9.io

[5] XQLint - XQuery Code Quality Tool. https://github.com/wcandillon/xqlint

[6] Jonathan Robie – Don Chamberlin – Michael Dyck – John Snelson: XQuery 3.0: An XML Query Language. W3C Working Draft 13 December 2011 http://www.w3.org/TR/xquery-30/

[7] Jonathan Robie – Don Chamberlin – Michael Dyck – Daniela Florescu – Jim Melton – Jérôme Siméon: XQuery Update Facility 1.0. W3C Recommendation 17 March 2011 http://www.w3.org/TR/xquery-update-10/

[8] Pat Case – Michael Dyck – Mary Holstege – Sihem Amer-Yahia – Stephen Buxton – Jochen Doerre – Jim Melton – Michael Rys – Jayavel Shanmugasundaram: XQuery and XPath Full Text 1.0. W3C Recommendation 17 March 2011 http://www.w3.org/TR/xpath-full-text-10/

[9] Jonathan Robie – Ghislain Fourny – Matthias Brantner – Daniela Florescu – Till Westmann – Markos Zaharioudakis: JSONiq. XQuery for JSON, JSON for XQuery http://jsoniq.org/docs/spec/en-US/html-single/index.html

[10] Matthias Brantner – Daniela Florescu – Ghislain Fourny – Josh Spiegel: XQuery Scripting. Extension Proposal http://www.zorba-xquery.com/html/scripting_spec.html

[11] Matthias Brantner – Daniela Florescu – Markos Zaharioudakis: XQuery Data Definition Facility. http://www.zorba-xquery.com/html/documentation/latest/zorba/xqddf

[12] FLWOR Foundation: Zorba XQuery Processor. http://www.zorba-xquery.com

[13] Gunther Rademacher: REx Parser Generator. http://www.bottlecaps.de/rex/

[14] Ian Hickson: Web Workers W3C Candidate Recommendation 01 May 2012. http://www.w3.org/TR/workers/

[15] Enno Boland – Guillermo Rauch: Engine.IO SPEC https://github.com/LearnBoost/engine.io-client/blob/master/SPEC.md

[16] IETF: HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV) RFC 4918. http://www.ietf.org/rfc/rfc4918.txt

[17] Datypic: FunctX XQuery Function Library http://www.xqueryfunctions.com/

eXistential Issues in Farming

Ari Nordström
Condesign
<ari.nordstrom@condesign.se>

Abstract

The Federation of Swedish Farmers – LRF - provides its 170,000 members with a yearly printed checklist document used to indicate compliance with existing state and EU farming regulations. This document, also available as a web-based checklist form fed from a central SQL database, is out of date almost as soon as it leaves the printers.

The LRF wants to provide customised PDFs to their members, with member-specific checklists and corresponding help texts - facts - instead of today's 100+ page, fine print document. Most of this is available in the SQL database - the web form answers, with a member's answers and comments, are saved and could be used to generate such customised PDFs on request, but the SQL database is ill suited for the task.

This paper describes the solution currently being implemented. It consists of an external eXist-based publishing server, to which the SQL DB contents are exported nightly. The quality of the exported XML varies, with factual texts in HTML fragments, etc, requiring cleanup and conversions in stages before they are converted to a publishing XML format. The solution uses XProc pipelines and XSLT to handle most of this processing. A yearly volume of roughly 40,000 documents is expected.

Additional requirements now include writing everything but the checklist questions themselves in eXist. For this, an oXygen/eXist system using a DocBook subset is used, with more pipelines and XSLT for processing currently being implemented.

1. Introduction

The Federation of Swedish Farmers – LRF – is an interest and business organisation for the green industry with approximately 170,000 individual members. Together they represent some 90,000 enterprises, which makes LRF the largest organisation for small enterprises in Sweden.

Their operations range from small-scale farming to large operations handling crop, livestock, and more. They frequently rely on state or EU funds to subsidise their operations but in order to receive funding, they need to comply with relevant regulations in their various areas of operation, requiring inspection both by themselves and by appointed officials.

To ease the compliance-related tasks for its members, LRF provides a yearly checklist document with questions and forms, plus help texts, "facts", explaining the questions. The checklist is now also available on the web, where members must answer a potentially very large number of questions. The 5,000 LRF members that use the online services have all registered on the LRF web site. The registration data is used to filter the checklists and the associated facts and laws so that only the relevant contents are included. A dairy farmer will not have to answer questions about beekeeping, for example.

The LRF members that use the online services, currently more than 5,000 of them, have all registered on the LRF web site, including contact information, business details and their area(s) of operations (for example, livestock and/or crop). This data is used to filter the checklists and the associated facts and laws so that only the relevant contents are included. A dairy farmer will not have to answer questions about beekeeping, for example.

Three possible answers can be given. "Yes" and "N/A" answers indicate compliance and non-applicability, respectively, while "no" answers identify areas where actions are required before approval. Members can add comments, notes, completion dates and other information to every "no" answer to ease inspection and to keep track of their progress.

The checklist questions always include help texts to aid the members in completing the checklists. These help texts,called *facts* in LRF terminology, appear on the web as popups, and as a separate - and large - facts section in the yearly checklist printed document, a 100+ pages document based on the web contents as well as other sources but completed manually by LRF writers in *InDesign*. The InDesign version is printed in roughly 40,000 copies.

The web checklist data, with questions, member data (inluding the member partial or complete checklists) and related facts, is fed from a central SQL database. For the printed InDesign document, however, information is manually added through several other sources.

The facts (related to questions) are based on laws and regulations originating from a third-party legal publishing house that fetches the data from various government agencies, EU databases, and so on, in formats ranging from text to PDF and, after some processing, place it in their database. As the regulatory data is frequently updated, it is mirrored to LRF's database nightly.

Most members are not active within every area covered by the printed checklist document so much of the contents are irrelevant to them, making the document difficult to browse and find information in. Also, the PDF information is rarely up to date, as the requirements change more often than the document, causing additional problems. Nevertheless, the typical member is not computer-savvy and much prefers paper to web.

1.1. Requirement: Customised Checklists

The yearly printed checklist is not particularly useful or up-to-date, so the LRF wants to replace it with member-customised PDF checklist documents instead, including only the questions and facts relevant to each member but also the member's individual comments, completion dates, etc. This PDF will be available for download from the web once the web form is completed.

Basically, providing a customised PDF to replace the static *InDesign* document with should be a question of exporting the checklist data (questions and facts) and the user-specific data (personal data, answers, comments and dates) from the SQL database to XML in a suitable form, filtering the total data for each member, and then applying XSL-FO stylesheets to that.

1.2. Requirement: Cache Storage for the XML

The XML needs to be processed so only information common to all members and the member-specific information is included in the customised PDF. As this information comes from several different sources and is produced independently, at different times, it is very useful to store that XML in the same place, both the manually authored parts and the exported parts, and keep them "in synch" with their sources. Compiling them at runtime would be too time-consuming and complex.

Therefore, if publishing the combined XML in various configurations, a cache storage area is required. The publishing needs to be initiated from that storage area rather than the remote sources.

1.3. Requirement: Dynamic Processing and Publishing

The exported XML needs to be kept in sync with the sources, preferably with either a nightly mirroring operation or an incremental update when changes have been made. This XML will contain both "common" parts, applicable to all members, and "member-specific" parts applicable only to the users with relevant areas of operation *and* incomplete actions pertaining to that content[1].

When publishing a member-specific PDF, the processing needs to be initiated by the user (from, for example, the web form) but processed in the cache storage area. This entails excluding content not relevant to the user (that is, not registered and associated with that user's ID, including content ("facts") associated with the member's "no" answers, normalising the resulting XML into one big file, validating the normalised file, and finally converting the normalised XML to PDF via XSL-FO stylesheets.

[1]In other words, questions answered with "no".

This includes several different steps, best expressed in XProc pipelines: locating and fetching the required XML, including and excluding content based on the user ID (and including user metadata), normalising the resulting files, validating the result, and converting it to PDF.

The “cache”, then, requires XProc pipeline processing, XSLT transformations, validation, and XSL-FO processing. Hmm. I wonder; is there such a “cache storage” product?

2. Useful Stuff

Before I describe the actual solution, allow me to briefly describe some of the components and why they matter.

2.1. XProc and Friends

XProc is a given. It's probably the most useful spec I've come across in later years. For a non-programmer, implementing a pipeline comprising several steps of processing on an XML file, without having to know anything beyond XML, is now not only possible but realistic. Adding or reconfiguring processing steps is easy, for exampe, when pointing out a different stylesheet or reconfiguring an existing one with new parameters or options.

A pipeline can be more of a blueprint for those parameters and options when processing, rather than a single process. With enough variables, a single pipeline can be a very versatile tool when publishing, allowing for a multitude of stylesheets, stylesheet options, and so on.

Here lies a problem. It is often easy to reconfigure the behaviour of a single set of stylesheets in a pipeline by changing input parameters and options, etc, but to make these choices available to an end user is harder. A choice of parameters may result in new choices and new configurations, all of which make the act of selecting more complex because most users are not particulartly comfortable with configuring either a pipeline or the batch file that runs it. They require a user interface that then produces the batch file, based on the user's choices.

2.1.1. Process XML

Process XML is about adding an abstraction level to XML processing so that not only are the steps in XProc pipelines expressed in XML, but also the configuration of the XProc engine. Does this sound like an unnecessary evil to you?

Here's why I think it's not: an XProc pipeline of reasonable complexity can be run with many different parameters and options, with different input stylesheets and *their* input parameters, etc. The various tools used can be configured, as can the XProc engine itself. These various options present choices to the users, in effect making the pipeline and its various choices into a *blueprint* of available alternatives rather than a specific pipeline with its specific options. These choices need to be made available to a user, preferably in a user interface, allowing the user to select an input stylesheet and its parameters, options, etc. Also, while one stylesheet may result in only a few options (or none), another might result in further choices.

The point here is this: a pipeline and its associated alternatives, choices and options will almost always only be as flexible as the user interface that makes its features available to the user. In other words, a dynamic, adaptable GUI will stand a better chance than a static one.

The process XML describes all these available choices for a specific process, regardless of how many parameters and options and stylesheets that need to be selected before the actual pipeline is run. The XML can be converted to a GUI, perhaps presented using XForms.

But also, when the choices have been made and we are left with a specific pipeline, it still needs to be run with the selected options. The resulting XML, based on the GUI selections, describes the actual process instance. This can be converted to a batch file that runs the configured pipeline.

2.2. XQuery and eXist

Another very useful W3C spec but one that I have only recently started exploring in more detail, is XQuery. The databases I have had opportunity to use for my XML content management have all been SQL-based and with very limited XML processing to begin with. Therefore, most of my earlier work has naturally focussed on other areas than addressing XML as XML when it's stored in a database, and, needless to say, I've done most of my XML conversions using XSLT rather than XQuery, so far.

eXist has changed all that. I've been wanting a chance to implement it in a customer project ever since I first saw it in action at an XML Prague conference a few years ago. I could see that processing my XML as XML, using XML-based tools, directly in the database would not only lessen but eliminate a lot of the problems I was having. Using a combination of XQuery to locate the content I require, and XProc, XSL and an integrated FO engine to process and publish it[2] is, for me, a game changer.

It was a natural for the project described in this paper.

3. The Evolution of the Publishing Chain

This whitepaper is something of a travelogue; it has evolved with the publishing server and its publishing chain. Reading through the draft submmitted to XML Prague, I realised that much had changed and a reader would have to possess infinite

[2]Or some other combination of techniques starting with "X".

patience and a thick set of notes to wade through the details. Hence this section; it describes the basic concepts and evolution of the publishing chain, hopefully making the rest of the paper more readable.

The checklist document as printed, with its hundreds of questions, accompanying facts and supplementary information, was to be replicated by the pub server. There would have to be a root XML document that links to the various parts, some of them static and others dynamically produced based on the individual user's ID, in this paper called UID=XYZ. The total document, with everything, we call UID=0.

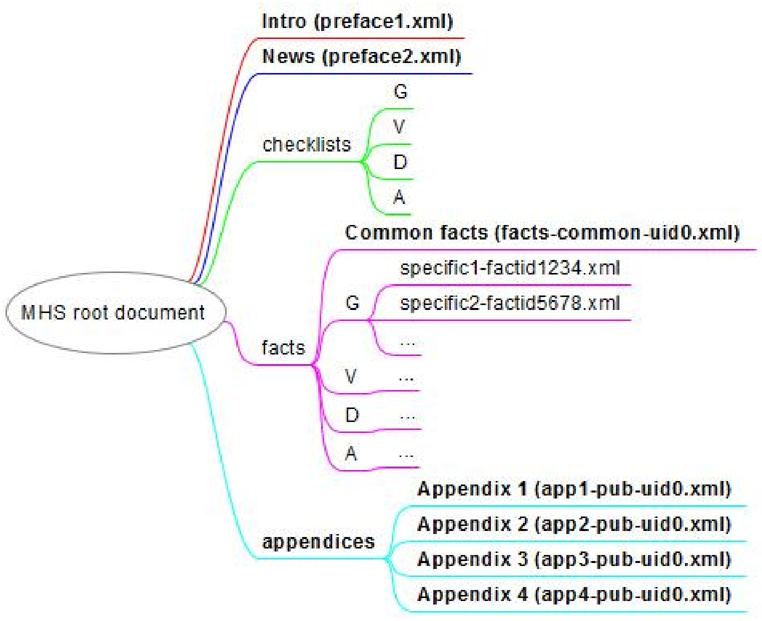

MHS root document
Intro (preface1.xml)
News (preface2.xml)
checklists
G
V
D
A
facts
Common facts (facts-common-uid0.xml)
G
specific1-factid1234.xml
specific2-factid5678.xml
...
V
...
D
...
A
...
appendices
Appendix 1 (app1-pub-uid0.xml)
Appendix 2 (app2-pub-uid0.xml)
Appendix 3 (app3-pub-uid0.xml)
Appendix 4 (app4-pub-uid0.xml)

Figure 1. The Basic Document Structure

Figure 1 shows the basic checklist document structure. The modules marked with bold text are “standard texts”, required in every published document. These were written specifically for the InDesign version and were not stored in the SQL database. The standard texts include appendices containing additional information (for example, about specific regulatory information concerning certain animals), but also include documentation explaining how to use the document, common facts, etc.

The initial assumption was to manually enter the information in XML files that would be stored as static texts in eXist, linked to from the customised PDFs but not edited or changed with any regularity.

The checklist structure, marked in green, consists of a total of roughly 600 questions, divided into four basic areas of operation. A version for UID=0, that is, the total, will include them all while a user-specific version, UID=XYZ, will only include those questions that are relevant to that particular user[3].

The facts section, marked in purple, consist of a common section[4] that includes information and help on how to answer the checklist questions correctly, additional information on laws and regulations, etc, and four basic facts sections corresponding with the checklist main areas of operation, above. There is a "fact" for each and every question, but also "facts" that correspond with groups of questions within each area.

Producing the UID=0 version of the checklist document might be represented with this:

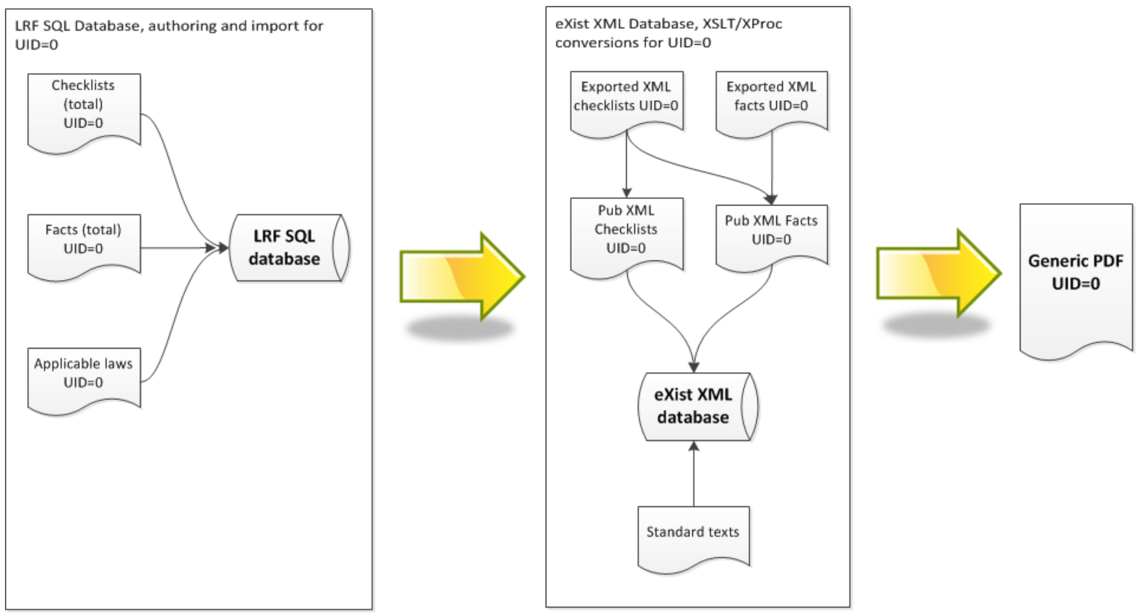


Figure 2. producing a UID=0 Document

producing a user-specific version would look like this:

[3]The SQL database includes logic to filter among the questions, based on the information from the user when the user registers as a new member.
[4]Also a standard text.

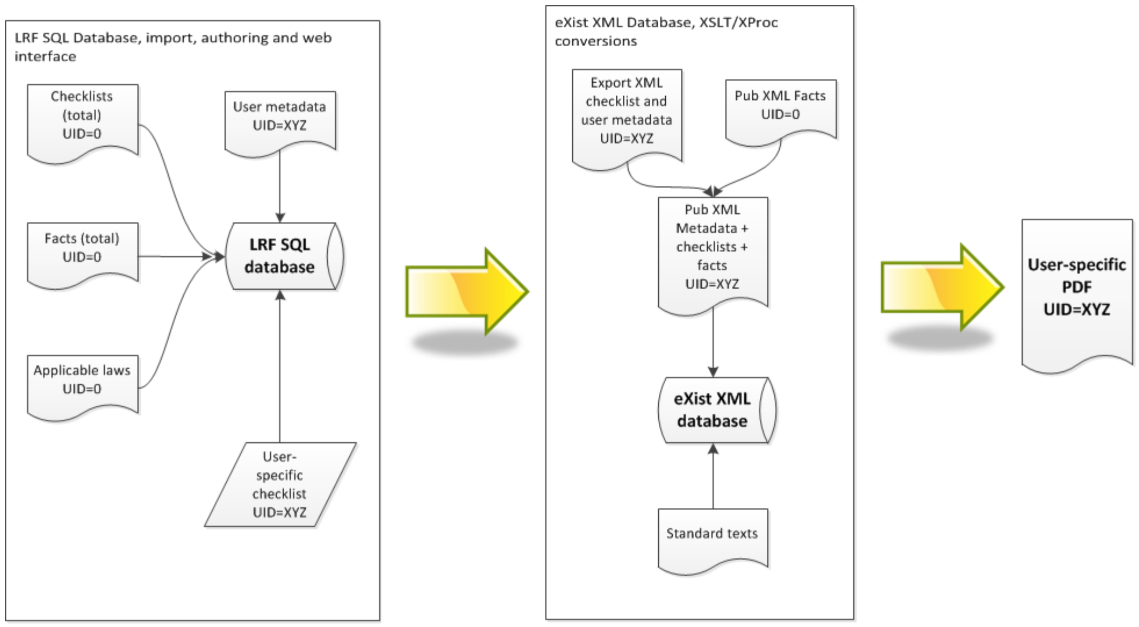

LRF SQL Database, import, authoring and web interface
Checklists (total) UID=0
User metadata UID=XYZ
Facts (total) UID=0
LRF SQL database
Applicable laws UID=0
User-specific checklist UID=XYZ
eXist XML Database, XSLT/XProc conversions
Export XML checklist and user metadata UID=XYZ
Pub XML Facts UID=0
Pub XML Metadata + checklists + facts UID=XYZ
eXist XML database
Standard texts
User-specific PDF UID=XYZ

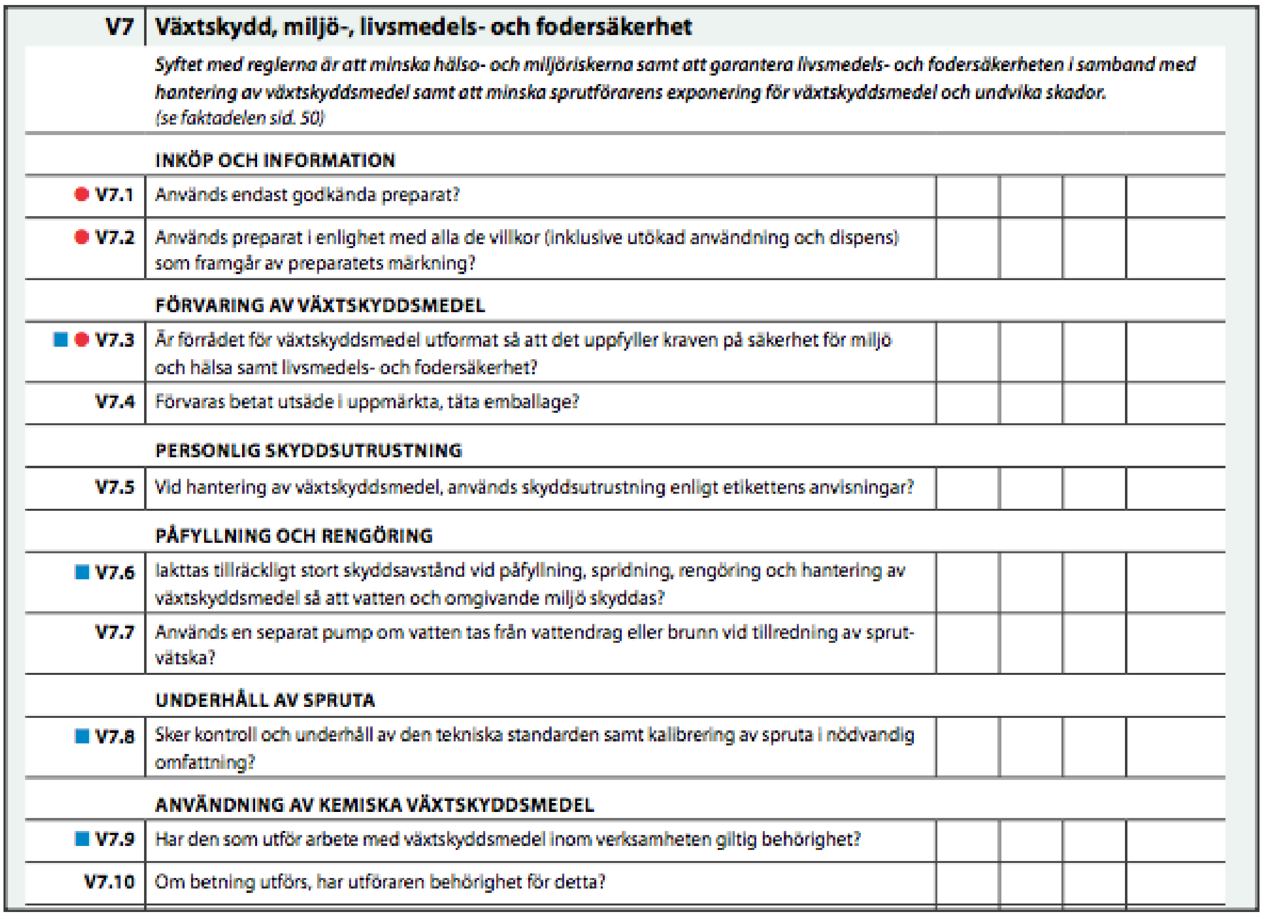
Figure 3. process-uidXYZ.png

When the project started, everything but the standard texts (that is, checklists, facts and user data[5]) were written and handled in the SQL database and were therefore to be exported to the publishing server nightly. The standrad texts would be converted to XML and stored on the pub server as static texts.

3.1. Checklist Filtering

The checklist questions can be answered with “yes”, “N/A” and “no”. The first two mean that the regulation covered by the question is complied with or is not applicable to the member. A “no”, however, means that the member does not yet comply with the regulation. In that case, he needs to set a completion date and add any relevant comments, both for his own use and that of the inspector's. Figure 4 shows a checklist fragment. The four empty columns to the right represent yes, N/A, no and comments field in the printed version.

[5]The user data is not shown in the illustration. It is included in a metadata page immediately after the cover.



V7	**Växtskydd, miljö-, livsmedels- och fodersäkerhet**				
	Syftet med reglerna är att minska hälso- och miljöriskerna samt att garantera livsmedels- och fodersäkerheten i samband med hantering av växtskyddsmedel samt att minska sprutförarens exponering för växtskyddsmedel och undvika skador. *(se faktadelen sid. 50)*				
	INKÖP OCH INFORMATION				
● **V7.1**	Används endast godkända preparat?				
● **V7.2**	Används preparat i enlighet med alla de villkor (inklusive utökad användning och dispens) som framgår av preparatets märkning?				
	FÖRVARING AV VÄXTSKYDDSMEDEL				
■ ● **V7.3**	Är förrådet för växtskyddsmedel utformat så att det uppfyller kraven på säkerhet för miljö och hälsa samt livsmedels- och fodersäkerhet?				
V7.4	Förvaras betat utsäde i uppmärkta, täta emballage?				
	PERSONLIG SKYDDSUTRUSTNING				
V7.5	Vid hantering av växtskyddsmedel, används skyddsutrustning enligt etikettens anvisningar?				
	PÅFYLLNING OCH RENGÖRING				
■ **V7.6**	Iakttas tillräckligt stort skyddsavstånd vid påfyllning, spridning, rengöring och hantering av växtskyddsmedel så att vatten och omgivande miljö skyddas?				
V7.7	Används en separat pump om vatten tas från vattendrag eller brunn vid tillredning av sprut-vätska?				
	UNDERHÅLL AV SPRUTA				
■ **V7.8**	Sker kontroll och underhåll av den tekniska standarden samt kalibrering av spruta i nödvändig omfattning?				
	ANVÄNDNING AV KEMISKA VÄXTSKYDDSMEDEL				
■ **V7.9**	Har den som utför arbete med växtskyddsmedel inom verksamheten giltig behörighet?				
V7.10	Om betning utförs, har utföraren behörighet för detta?				

Figure 4. A Checklist Example

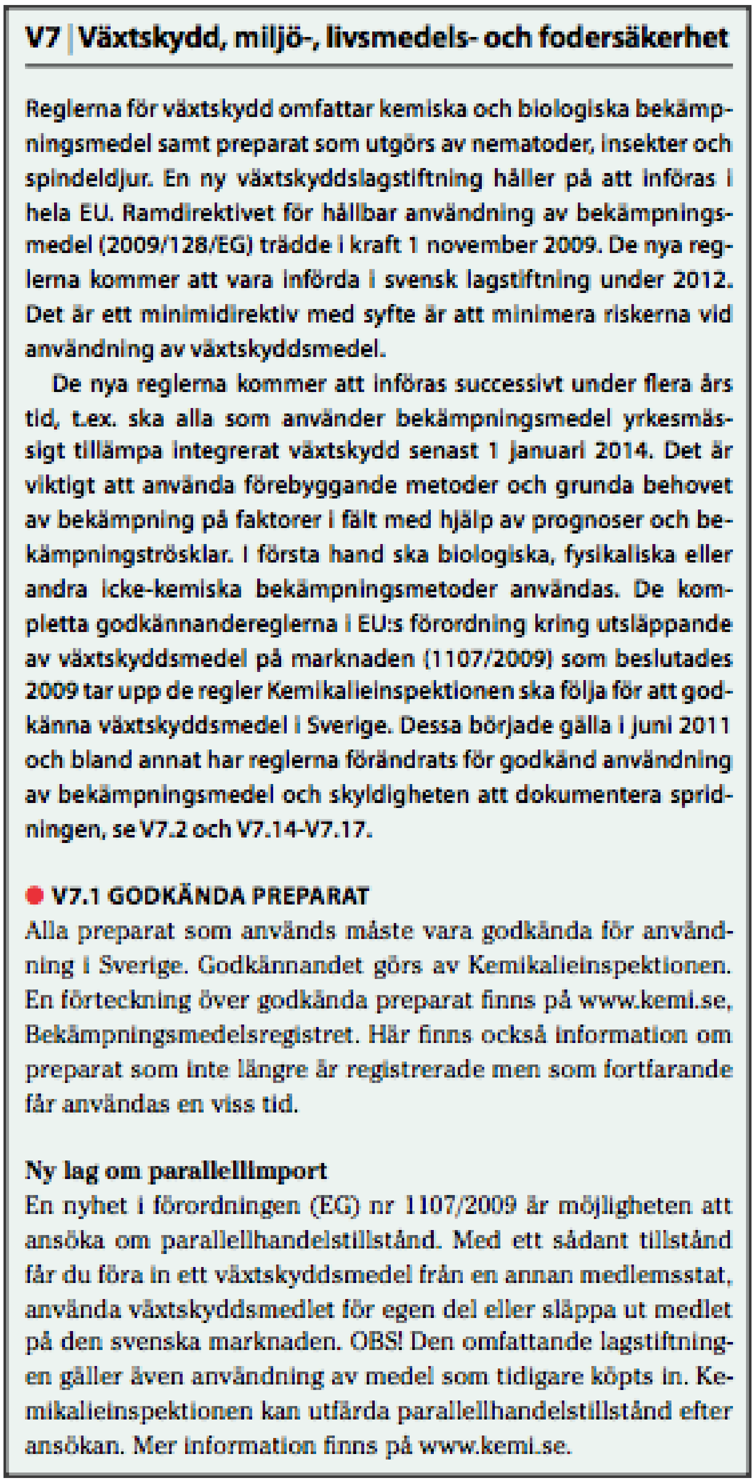
A user-specific checklist document must *only include the "no" answers and their associated facts* (and, of course, the standard texts), as the others are either not applicable or already in compliance with the regulations. It also needs to include the completion dates and the comments.

The observant reader will note the red and blue marks in the checklist and fact example illustrations. These marks indicate *cross-compliance requirements*, meaning that that they are required for compliance with certain state or EU regulations.

The user-specific data exported from the SQL database is an XML file identified using the UID and includes the checklist answers, dates and comments, in addition to the user metadata.

3.2. eXist Layout, Version 1

The first publishing server (eXist) had a simple layout, as everything but the standard texts were imported nightly from the SQL DB. Figure 6 shows the collections and file naming.



V7 | Växtskydd, miljö-, livsmedels- och fodersäkerhet

Reglerna för växtskydd omfattar kemiska och biologiska bekämpningsmedel samt preparat som utgörs av nematoder, insekter och spindeldjur. En ny växtskyddslagstiftning håller på att införas i hela EU. Ramdirektivet för hållbar användning av bekämpningsmedel (2009/128/EG) trädde i kraft 1 november 2009. De nya reglerna kommer att vara införda i svensk lagstiftning under 2012. Det är ett minimidirektiv med syfte är att minimera riskerna vid användning av växtskyddsmedel.

De nya reglerna kommer att införas successivt under flera års tid, t.ex. ska alla som använder bekämpningsmedel yrkesmässigt tillämpa integrerat växtskydd senast 1 januari 2014. Det är viktigt att använda förebyggande metoder och grunda behovet av bekämpning på faktorer i fält med hjälp av prognoser och bekämpningströsklar. I första hand ska biologiska, fysikaliska eller andra icke-kemiska bekämpningsmetoder användas. De kompletta godkännandereglerna i EU:s förordning kring utsläppande av växtskyddsmedel på marknaden (1107/2009) som beslutades 2009 tar upp de regler Kemikalieinspektionen ska följa för att godkänna växtskyddsmedel i Sverige. Dessa började gälla i juni 2011 och bland annat har reglerna förändrats för godkänd användning av bekämpningsmedel och skyldigheten att dokumentera spridningen, se V7.2 och V7.14-V7.17.

● V7.1 GODKÄNDA PREPARAT

Alla preparat som används måste vara godkända för användning i Sverige. Godkännandet görs av Kemikalieinspektionen. En förteckning över godkända preparat finns på www.kemi.se, Bekämpningsmedelsregistret. Här finns också information om preparat som inte längre är registrerade men som fortfarande får användas en viss tid.

Ny lag om parallellimport

En nyhet i förordningen (EG) nr 1107/2009 är möjligheten att ansöka om parallellhandelstillstånd. Med ett sådant tillstånd får du föra in ett växtskyddsmedel från en annan medlemsstat, använda växtskyddsmedlet för egen del eller släppa ut medlet på den svenska marknaden. OBS! Den omfattande lagstiftningen gäller även användning av medel som tidigare köpts in. Kemikalieinspektionen kan utfärda parallellhandelstillstånd efter ansökan. Mer information finns på www.kemi.se.

Figure 5. A Corresponding Fact Example

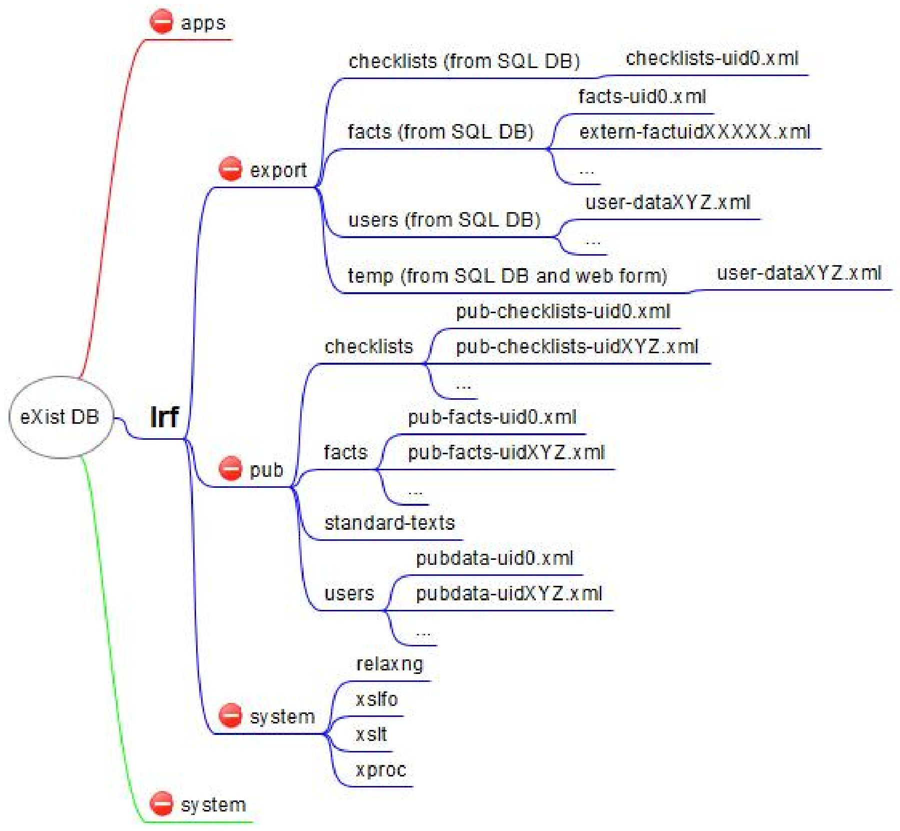


Figure 6. eXist Collection Layout, Version 1

There are two main collections[6], `export` and `pub`. The former stores the raw XML content imported from the SQL DB while the latter stores the same contents after they have been converted to the publishing XML format. The publishing chain only deals with content in the `pub` collections.

Here's the basic preprocessing chain for version 1, repeated nightly. Every item here represents an XProc pipeline that does things with the XML in eXist:

1. The user metadata is converted from `/db/lrf/export/user-uidXYZ.xml` to `/db/lrf/pub/users/pubdata-uidXYZ.xml`. These include name, address, company name, etc.
2. The total checklist (UID=0) is converted from `/db/lrf/export/checklists/checklists-uid0.xml` to `/db/lrf/pub/checklists/pub-checklists-uid0.xml`.

[6]The illustration also shows eXist's standard collection and the *system* collection for processing LRF documents.

These are the questions total, each accompanied by a reference to the corresponding fact.

3. The user-specific checklists are converted from `/db/lrf/export/user/user-uidXYZ.xml` to `/db/lrf/pub/checklists/pub-checklists-uidXYZ.xml`.

   This is a "no" answer with a comment ("test1") and a completion date ("2012-12-21T00:00:00"). The user-specific checklist still contains *all* answers but indicate the answers, comments, etc, but also show if a question is not yet answered and the particular checklist is incomplete.
4. The complete set of facts is converted, identifying which facts to convert using the fact IDs in the total checklist for UID=0, from the 600+ raw fact files (`/db/lrf/export/facts/extern-factidXXXX.xml`) to a single, large facts file, `/db/lrf/pub/facts/pub-facts-uid0.xml` that also links to common facts, fetched from `standard-texts`. The resulting file can be used as the complete facts section for UID=0.
5. The user-specific facts are filtered from the total facts file, `pub-facts-uid0.xml` in #4, using the answers given in the user-specific checklist, `pub-checklists-uidXYZ.xml`, converted in #3.

This worked well and produced nice, user-specific versions of the checklist document after adding the publishing steps (see Section 5), until LRF updated their requirements.

3.3. eXist Layout, Version 2

Quite a few of the InDesign texts (introductory texts, appendices, etc) were written specifically for it; they do not exist in the SQL database or elsewhere. These texts had to be included in the eXist publishing chain. The original plan was to simply convert and link to them from the root XML template, without providing a set method for editing them, but soon, LRF wanted some way to update them when needed. Would it be possible to write them directly on the publishing server?

This was a request we had anticipated. At first, then, the plan was to create a basic *oXygen*-based authoring environment for the publishing XML format, add linking functionality, for cross-references, images and content inclusions, and include some customisation and hiding of tags for the benefit of authors unaccustomed to XML. oXygen, as some readers may already be aware of, includes an eXist integration. This could easily be used to update the standard texts in `/db/lrf/pub/standard-texts`.

But very soon LRF also wanted to edit the facts related to the checklists on the publishing server rather than in the SQL DB's admin interface. Their problem was that when adapting the source material for laws and regulations provided by the third-party supplier, *they are required to do it on-site, even if they only need to edit or copy/paste content.* This, of course, is both cumbersome and expensive. LRF saw the need to move away from the third-party dependency, realising that they a) do not import as much as they thought, b) they can import the legal texts themselves, and c) they need to adapt the information in any case.

The oXygen environment now needed to handle several types of information and simply editing it in `/db/lrf/pub` would be impractical as it was part of the publishing chain. We decided to add a work area, `/db/lrf/work`, that would contain anything authored in oXygen and not be part of the publishing chain.

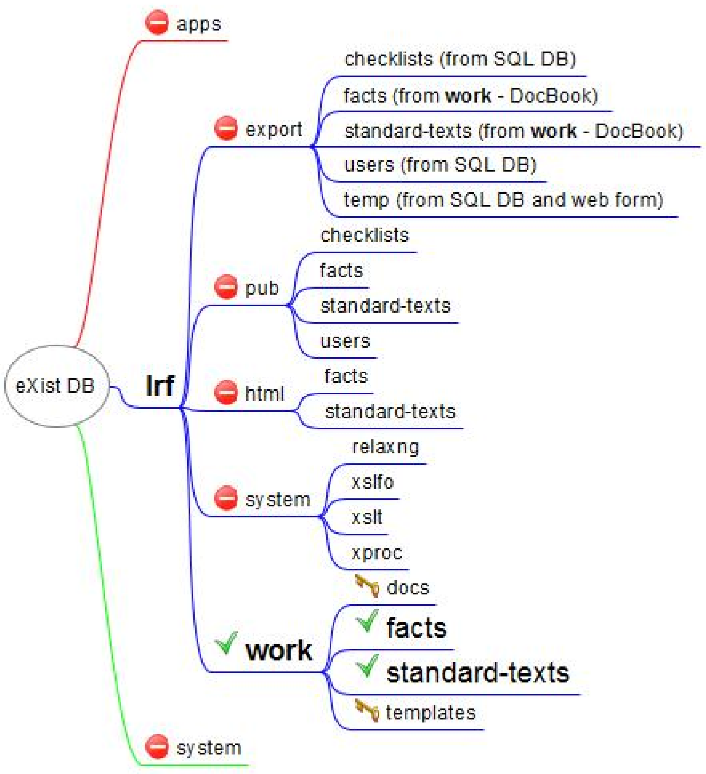


Figure 7. eXist Collection Layout, Version 2

In the new version, the checklists and the user data are still imported nightly (to `/db/lrf/export`) but the corresponding facts and the standard texts are simply copied from `/db/lrf/work` when approved for publishing. The preprocessing chain described above is nearly intact, however. The facts are authored in DocBook (see Section 6), placed in `/db/lrf/export/facts` and converted to `/db/lrf/pub/facts` nightly, requiring only the replacement of a single XSLT (from an HTML-to-publishing XSLT to a DocBook-to-publishing one). The standard texts are now also placed in `export` when approved and converted from DocBook format to the pub format, which required an additional pipeline to be performed after onverting the facts (step #4, above).

Since the facts are now no longer created and edited in the SQL DB, we also added a conversion from DocBook to HTML and an eXist collection from which HTML content is returned to the SQL DB nightly, to be used in the checklist web form.

The eXist collection layout for LRF is now divided into four main parts:

- *export* now contains both SQL and DocBook material. With oXygen added for authoring some of the information, user data and the checklist questions are still imported from the SQL DB every night, but their respective facts are edited in *work* and moved to *export* once they have been approved for publishing.
- *pub* contains *everything* required to publish the PDF. Some are converted from the raw XML data (checklist questions and user data) while others (standard texts and facts) are converted from the DocBook authoring format.
- *work* is for creating and editing standard texts and checklist facts. It is not part of the publishing workflow; to add a document edited in *work* to the publishing data in *pub*, it must first be moved to *export* (with the user running a local XQuery).
- *html* contains aproved facts in HTML format that are fed *back to* the SQL database for inclusion in the web checklist form. These HTML documents are converted form the approved DocBook.

Every main collection, sans *html*, has the same basic subcollections:

- *checklists*
- *facts*
- *standard-texts*
- *users*

These are needed to keep the main types of information apart for easier processing when importing, converting and authoring. For example, checklists need to be processed first because they decide which facts to include. Etc.

4. Processing in (and Outside) eXist

As eXist includes native XProc support through *xprocxq* as well as the *Calabash* module, implementing the processing outlined above as XProc pipelines was natural. Most processing steps are about moving and converting XML, and validating the result, so what could have been implemented in XQuery became pipelines.

After writing a number of XSLT stylesheets and the pipelines to call them from, I discovered that eXist's XProc support was effectively broken in the 2.0 Tech Preview we were using. A quick email to Jim Fuller, the person behind both XProc implementations in eXist, confirmed. Panic!

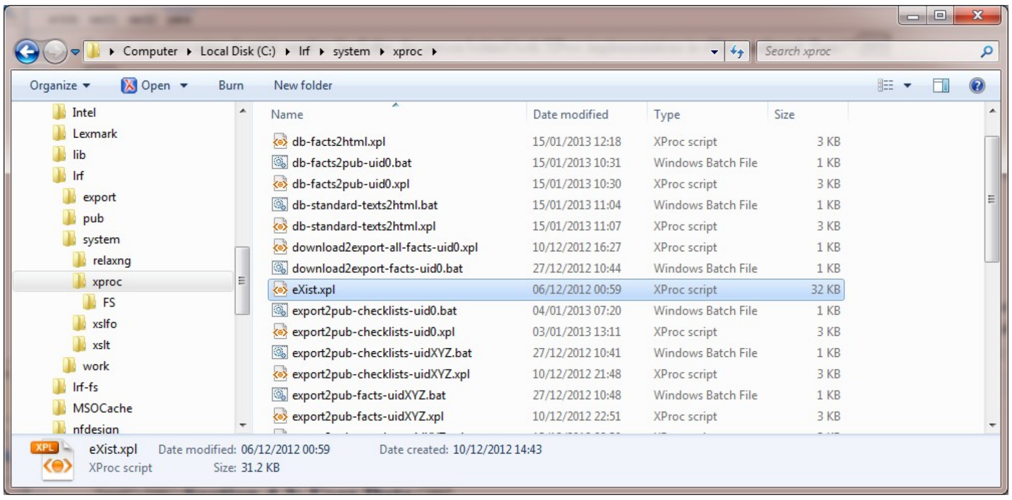
Desperately browsing the net for alternatives[7], I came across James Sulak's eXist XProc library that allows you to do a sort of reverse integration of Calabash with eXist, placing the Calabash locally but communicating with the database thorugh extension steps. It almost literally saved my bacon.

4.1. A Reverse Integration: the eXist XProc Library

James Sulak's eXist XProc library does this (see https://github.com/jsulak/eXist-XProc-Library):

A set of XProc extension steps for interacting with an eXist XML database from a client. Using these steps, you can conduct common eXist management tasks from XProc - loading resources, extracting resources, querying data, etc. They fill much the same role as the eXist Ant tasks. They are run from a client, and so fill a different (smaller) role than James Fuller's xprocxq project, which is an entire XProc implementation within eXist.

The library is placed on the server's file system so easiest was simply to replicate the planned eXist system collections (see Figure 6) with a local file system version:



Name	Date modified	Type	Size
db-facts2html.xpl	15/01/2013 12:18	XProc script	3 KB
db-facts2pub-uid0.bat	15/01/2013 10:31	Windows Batch File	1 KB
db-facts2pub-uid0.xpl	15/01/2013 10:30	XProc script	3 KB
db-standard-texts2html.bat	15/01/2013 11:04	Windows Batch File	1 KB
db-standard-texts2html.xpl	15/01/2013 11:07	XProc script	3 KB
download2export-all-facts-uid0.xpl	10/12/2012 16:27	XProc script	1 KB
download2export-facts-uid0.bat	27/12/2012 10:44	Windows Batch File	1 KB
eXist.xpl	06/12/2012 00:59	XProc script	32 KB
export2pub-checklists-uid0.bat	04/01/2013 07:20	Windows Batch File	1 KB
export2pub-checklists-uid0.xpl	03/01/2013 13:11	XProc script	3 KB
export2pub-checklists-uidXYZ.bat	27/12/2012 10:41	Windows Batch File	1 KB
export2pub-checklists-uidXYZ.xpl	10/12/2012 21:48	XProc script	3 KB
export2pub-facts-uidXYZ.bat	27/12/2012 10:48	Windows Batch File	1 KB
export2pub-facts-uidXYZ.xpl	10/12/2012 22:51	XProc script	3 KB

eXist.xpl Date modified: 06/12/2012 00:59 Date created: 10/12/2012 14:43
XProc script Size: 31.2 KB

Figure 8. Local Filesystem on the Server

In this hack - because it's what it feels like - the pipelines need to download the XML to be converted, apply the XSLT, validation and possibly other steps, and then upload the resulting XML to eXist. This works reasonably well. While slower than a purely eXist-based version would have been, the heavy conversions involving the facts are done nightly rather than during peak hours, and the remaining conversions (see Section 4.4) only involve small files[8].

[7]By then, it was too late to reimplement the processing with XQuery, even if I had been up to par with XQuery as it relates to eXist.

[8]XQuery-based conversions would have been faster but it was much faster to simply edit the pipelines written for eXist to use the reverse integration instead.

4.2. Checklists and User Data

The exported checklist is a single large XML file that includes 600+ questions, divided into four main areas, each of which are in turn further divided into smaller related groups. Each question or group of questions also includes information on specific requirements, applicable laws and related facts. That last bit is identified using a fact ID. A single question looks like this:

```
<question operation="G1.1" question_id="1409">
    <jp_node_id>72632</jp_node_id>
    <text>Om verksamheten är anmälningspliktig, har anmälan gjorts till ▶
kommunen?</text>
    <tvar>1</tvar>
    <xtvar>0</xtvar>
    <ny>0</ny>
    <rek>0</rek>
    <laws>
        <law law_id="208575" />
        <law law_id="208576" />
    </laws>
    <facts>
        <fact fact_id="183178" />
    </facts>
</question>
```

The `text` element contains the question itself. Most of the fragment should be self-explanatory (see, for example, the `law` and `fact` elements that identify the laws and facts, respectively, that are related to this specific question). The `tvar` and `xtvar` tags indicate cross-compliance requirements (additional requirements for certain regulations).

This raw XML is converted to the publishing format, better suited for outputting the checklist in the preferred table-like format. It is used when outputting the total, generic checklist, the one we call UID=0.

User metadata is exported in one XML file per member, with a simple file naming convention: `user-uidXYZ.xml`, where `XYZ` is the user's ID. This ID is used everywhere, is unique (for obvious reasons) and therefore easy to match against.

The user-specific questions contained in `user-uidXYZ.xml` consist of user metadata (name, address, etc) and answers to *relevant* checklist questions (the filtering is done before export, in the SQL DB). If the answers are current (as indicated by the timestamp that is compared to the last-changed date of the file when requesting a publication), the user XML file can be used when requesting a user-specific publication. Here's a single answer:

```
<answer operation="G1.1" answer_id="1991102">
    <qresponse>no</qresponse>
    <responsedate>2012-12-20T10:08:56</responsedate>
    <comment>test1</comment>
    <attended>2012-12-21T00:00:00</attended>
    <jp_node_id>72632</jp_node_id>
    <question>Om verksamheten är anmälningspliktig, har anmälan gjorts till ▶
kommunen?</question>
    <tvar>1</tvar>
    <xtvar>0</xtvar>
    <ny>0</ny>
    <rek>0</rek>
    <laws>
        <law law_id="208575" />
        <law law_id="208576" />
    </laws>
    <facts>
        <fact fact_id="183178" />
    </facts>
</answer>
```

This is converted to a user-specific checklist in the user-specific document, and includes comments and dates pertaining to each question, in addition to the question itself.

Note
The exported user checklist always includes “yes” and “N/A” answers, too, even though they are currently removed before the checklist is published as a PDF, since they might be needed in the future if a different presentation of the checklist is required. The user checklist also includes questions that do not yet have an answer. These are used when numbering questions and answers, and, of course, to indicate that the inspection has not yet been completed.

4.3. Facts and Standard Texts

Facts are basically simple sections of content containing paragraphs, lists, tables and an occasional subsection. At first, when imported to eXist from the SQL database, they were small HTML fragments, plus some plain text and various Office fragments. They were stored in the SQL DB as fragments and fed to the web form popups as such, relying heavily on the ability of a web browser to handle almost any kind of content[9].

Of course, when exported to eXist, the requirements were far more strict. For example, we could not rely on the fact IDs in the checklists (see the example in Section 4.2); these only indicated the presence of a registered object in the database,

[9]Which it didn't; many facts were simply missing.

not that there was, in fact, any usable content stored. We had to produce not only the exported facts in a raw XML format but also a listing of the ones that would convert successfully in a large facts list XML document. This was then used to convert the exported facts to the pub format. Several facts would be missing when compared with the checklist references, requiring us to produce a placeholder fact to be used in the place of each missing fact.

Producing the facts directly for eXist allowed us to create a more robust publishing chain, as the vast majority of problems with the first version had to do with handling the exported facts content. Here's the updated preprocessing chain, performed after the nightly import:

1. As before, the user metadata is converted from `/db/lrf/export/user-uidXYZ.xml` to `/db/lrf/pub/users/pubdata-uidXYZ.xml`.
2. As before, the total checklist (UID=0) is converted from `/db/lrf/export/checklists/checklists-uid0.xml` to `/db/lrf/pub/checklists/pub-checklists-uid0.xml`.

   New checklist questions and groups are still created and edited in the SQL DB, as all of the logic handling them is implemented in it. When a new item is created, it starts life in draft and is not meant to be published before there is an accompanying fact. The new item is therefore marked as such in the exported checklist; this is a simple boolean attribute in the wrapper element for the question or group.
3. As before, the user-specific checklists are converted from `/db/lrf/export/user/user-uidXYZ.xml` to `/db/lrf/pub/checklists/pub-checklists-uidXYZ.xml`.

Facts and standard texts are now both edited in oXygen and saved in a work area in eXist. Here are the additional preprocessing steps required to handle them:

1. A new checklist item always gets a fact ID from the SQL DB. When the checklist is exported to eXist, a placeholder fact generated form a DocBook template, named using the fact ID[10], is copied to `/db/lrf/export/facts/` and `/db/lrf/work/facts/`. The fact is then edited in the latter collection by the authors, while the former exists simply to allow publishing in draft mode.
2. Standard texts are produced in the work area. As with facts, there are placeholders for them in `export`.
3. When approved or released in draft, facts and standard texts are copied from their work areas to `/db/lrf/export/facts/` and `/db/lrf/export/standard-texts`, respectively, replacing their placeholders. No conversions take place; the texts remain in DocBook format.

[10]The filename always ends with the string `-factid<factID>.xml` where `<factID>` is the fact ID assigned to it in the SQL DB.

4. The complete set of facts used by the latest exported checklist is converted from the DocBook format in `/db/lrf/export/facts` to a single, large facts file, `/db/lrf/pub/facts/pub-facts-uid0.xml` in pub format, just as before. The only difference here is the stylesheet used to convert facts with.
5. As before, the user-specific facts are filtered from the total facts file, `pub-facts-uid0.xml`, using the answers given in the user-specific checklist, `pub-checklists-uidXYZ.xml`.
6. The standard texts are converted from `/db/lrf/export/standard-texts` to `/db/lrf/pub/standard-texts`.

Publishing can in principle now take place. All of the required modules illustrated in Figure 1 are now present.

4.4. Runtime Conversions

Often, the individual member will request a personalised PDF immediately after completing the web form[11], which means that the last few answers in the web form will not have been exported to eXist yet. They will have to be exported again before publishing the completed checklist.

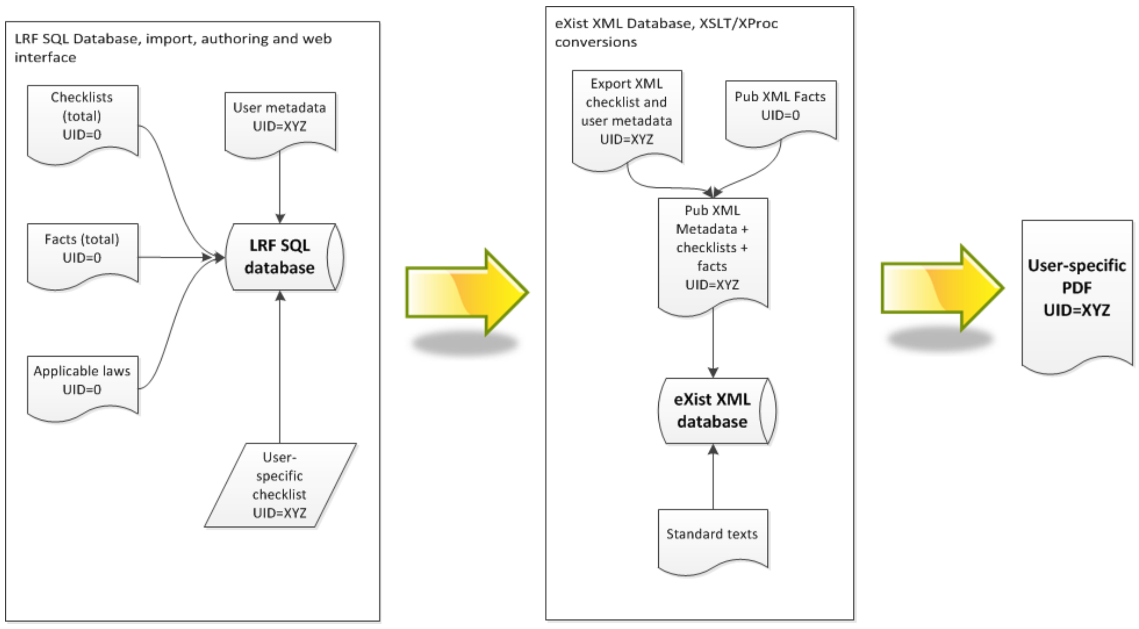

LRF SQL Database, import, authoring and web interface
Checklists (total) UID=0
User metadata UID=XYZ
Facts (total) UID=0
LRF SQL database
Applicable laws UID=0
User-specific checklist UID=XYZ
eXist XML Database, XSLT/XProc conversions
Export XML checklist and user metadata UID=XYZ
Pub XML Facts UID=0
Pub XML Metadata + checklists + facts UID=XYZ
eXist XML database
Standard texts
User-specific PDF UID=XYZ

Figure 9. Exporting and Publishing for UID=XYZ

As illustrated above, a few of the steps in Section 4.3 are repeated:

[11]The average LRF member does not visit the site often and will probably want to minimise any computer-related tasks.

1. A new user data and checklist file, `user-uidXYZ.xml`, is exported to a temp area in `export`.
2. The metadata in it is converted to a new `pubdata-uidXYZ.xml` in `/db/lrf/pub/users`.
3. The checklist answers are converted to a new user-specific checklist, stored in `/db/lrf/pub/checklists/pub-checklists-uidXYZ.xml`.
4. The new checklist in the publishing format is used to filter a corresponding set of facts from the main facts file, `/db/lrf/pub/pub-facts-uid0.xml`.

These steps all use the same pipelines and XSLT as the batch ocnversion that take place nightly, the difference being that they are here invoked with a `$uid` parameter to limit the run to the specific user and enable the pipelines to store the resulting files.

4.5. A Note Regarding Mirroring

When the project started, the fact texts were mirrored every night from LRF's legal text supplier's database and thus they could resonably be expected to change every night in the SQL DB, too. The checklist questions are thus also updated very frequently, changing with the changing regulations. In addition, the LRF members fill their checklists daily and so the user data changes often, too.

Mirroring the relevant SQL database contents to the publishing server on a nightly basis therefore seems reasonable. As a yearly volume of up to 40,000 member-specific generated PDFs is expected, this should be kept off business hours.

Similarly, converting the exported data and the DocBook-format facts and standard texts to the publishing format is done outside business hours to keep loads to a minimum. The only data transferred from the SQL DB to eXist is the user data completed by an individual member after the last mirroring and conversion operation.

5. Publishing

The publishing chain uses the user ID as an in parameter, `$uid="XYZ"`[12], when requesting a publication. Before publishing can take place, the preprocessing must be complete; the publishing pipeline assumes converted content where required in `/db/lrf/pub`.

[12]With `$uid="0"` representing the generic version.

5.1. Normalisation and Validation

The normalisation process uses a root document template found in `/db/lrf/pub/standard-texts` that basically links to the various modules (see Figure 1), including `$uid` in every "dynamic" link and thus allowing the normalisation stylesheets to modify the links with the current user ID and locate the right checklists and facts.

The UID=0 document, that is, the generic document, is generated in the same way. The XProc pipeline is simply given invoked with `$uid="0"`.

A complication with the authoring-enabled eXist solution is that no new questions exported from the SQL DB can be published before there are corresponding facts. On the other hand, the system will only generate a fact placeholder when there is a new question and a corresponding fact ID. Publishing the checklist without the placeholder would result in a non-publishable final document since a missing fact referenced from the checklist would break the publication, while including the placeholder would result in an empty fgact section for that question.

Therefore, the SQL database marks new questions (and new groups of questions) in the exported checklist with `new` attributes set to "1". During export, new facts are generated for every new fact ID. The `new` attribute is used to filter the publication, producing either a draft (by including everything, including the new items) with possibly missing or incomplete fact sections or an approved version where new items are excluded in the normalisation.

Note

Of course, the default publishing chain as invoked by an LRF member will only used the approved version.

Once normalised, the document is parsed against a RelaxNG schema for the publishing XML format. Any cross-references in the document are also checked, as some of the checklists will reference corresponding parts of the facts sections. Finally, there are also some content checks, for example, to make sure that any completion dates are realistic[13]. If the document is valid, it is passed on to the FO stylesheets. If not, the processing will stop, producing an error log and notifying the user.

5.2. Publishing

The FO stylesheet package is fairly big. It started out as a reasonably faithful reproduction of the InDesign layout, with many different master page layouts and conditions, but it has been considerably streamlined during the project. Having the LRF team review the test PDFs proved to be an excellent learning experience for

[13]The database erroneously reported 1890-01-01 for some answers, and for some reason, several of these dates are still in the database.

them; I was able to significantly simplify the layout once LRF realised that many of the InDesign tricks were in place not because they had to but because they could.

Finally, while *Apache FOP* is a competent XSL-FO publishing engine and available out-of-the-box in eXist, we needed both XSL-FO 1.1 compliance and higher performance with a smaller memory footprint. *RenderX's XEP* engine was our choice, mostly because it's more familiar to me.

The publishing XProc steps are, as everything else, currently implemented partly outside eXist. This does not pose a problem; quite the contrary, as the web server that includes eXist also stores the PDF output from XEP in a separate temporary download area and sends the PDF link in an email to the member that requested the publication. The member has 48 hours to download the PDF before it is automatically deleted.

5.3. The Silent Majority

While roughly 5,000 LRF members can now generate their individual checklists on the fly, it should be remembered that there are an additional 165,000 members. The complete checklist document is still every bit as important as before, the difference being that it can now be updated far more regularly than before.

In fact, the publishing server automatically produces an UID=0 version every night, immediately after completing the mirroring and conversion operations, and places a link to that document on the LRF web page.

6. Wait, We Want Authoring, Too!

At first, I thought implementing limited oXygen-based authoring based on my publishing XML format would be enough for producing the standard texts. The addition of facts, however, quickly changed my mind. While the first authoring system version will not implement significant reuse capabilities (such as linking to reusable modules or allowing the authors to define the root document template for the published document), the next probably will, meaning I would have to implement this in my for-publishing-only format AND the authoring environment.

George Bina at Syncro Soft suggested I use DocBook5 instead. Considering that everything from linking, reuse and image handling is already in place, a server-side conversion of DocBook to the publishing format would be a trivial matter.

Adding (DocBook) authoring on the server meant several things:

- First of all, a *work* collection (including facts and standard texts for editing) was needed. The *work* collection is not part of the publishing chain; instead, when a text is approved for publishing, it is moved to the *export* structure and thus included in the publishing chain.

- The publishing chain collection structures (*export* and *pub*) need to be hidden from view; authors must not change them.
- Approved facts and standard texts needed to be converted to the pub format. this was a mere in-place replacement of stylesheets in the publishing chain, converting DocBook rather than raw XML or HTML to the pub format.
- *Moving* the fact texts from *work* to *export* once they were approved proved to be something of a challenge, requiring a local XQuery to capture the currently open file's URL and use xmldb:copy in eXist. Eventually, this will be replaced by a process XML-generated GUI where the user is allowed more control when approving the file. For example, s/he might want to move the file to the publishing chain *without* approving it and instead adding conditional content for reviewers.
- *export*, of course, was designed for publishable content. The assumption is that everything that reaches *export* is already approved and so all related information must also be approved (and exist). Now, since new checklist questions are still authored in the SQL DB admin interface but there is a nightly export of the complete checklist from the SQL DB to eXist, any new questions will not yet have matching facts. These cannot be published; they have to be marked as "new", "draft" or similar and removed from the final document.
- Of course, the authors may want to publish a draft that includes the new, incomplete checklists, so any filtering of the new information must be conditional and user-selectable.

6.1. The Workflow and the Copy XQuery

As a somewhat interesting parenthesis, in the editing environment came finally the need for an XQuery script.

All of the processing is currently implemented as XProc pipelines that run XSLT stylesheets, validation and more. A lot of this is moving XML about, frequently converting it in the process. Some of these pipelines might have been better, and certainly faster, if implemented as XQueries, but that tells us more about how the project happened, with added requirements and setbacks involving the XProc implementations, than it does about poor design choices. Well, hopefully. It is what happened.

When implementing the working areas for oXygen, I considered what the authoring workflow would look like. The texts would be authored outside the publishing chain, meaning they'd have to be moved there and converted to the publishing format when included in the publication. This could only take place when a) there was a fact ID in the exported checklist to match the new text with[14] and b) the text was approved and suitable for publishing.

[14]If it was indeed a fact.

After briefly pondering a roundtrip conversion from DocBook to the publishing format and back, I realised that it was better to make that move a copy, and then update the working copy when a change was required, copying it again when ready. the copy would be the “approval” in the workflow.

Which only left me with the small matter of the copy function itself. The authors cannot access the `export` collections at all, so a direct *Ctrl+C* in oXygen's Data Source Explorer is out of the question. In eXist, of course, there are extension functions for the purpose: `xmldb:copy` does exactly that. It copies a resource from one collection to another. Perfect!

As I needed to provide the XQuery with the filename and URL of the fil to be copied, I ended up placing the XQuery on the client. There was no easy way to provide that information to a server-side script, I discovered.

6.2. The Return of the Process XML

The process XML I vowed to include in the project was left out for mostly practical reasons (there are no user-configurable pipelines in the preprocessing or publishing chains) is now set to make a comeback. The publishing chain will need user configuration, after all. There is the question of including or excluding newly added questions and facts in draft mode, which means that the author will have to be able to choose that feature if needed, but also author-selectable options for marking up that new content for reviews by non-authors.

The process XML is basically just a way to describe the various components, options and choices that I want to make available when using a specific pipeline. It is meant to be normalised (using XSLT) and then converted to a batch file (again, using XSLT) that is run by the system.. These two steps, of course, can be handled by a simple pipeline that initiates the larger (publishing, in this case) process as described by the process XML, like this:

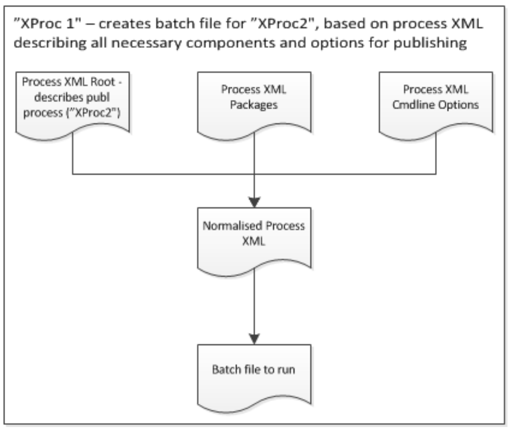

”XProc 1" – creates batch file for ”XProc2", based on process XML describing all necessary components and options for publishing
Process XML Root - describes publ process (”XProc2")
Process XML Packages
Process XML Cmdline Options
Normalised Process XML
Batch file to run

Figure 10. Initiating Publishing: XProc1

The basic publishing pipeline (“XProc2”) is slightly more complex:

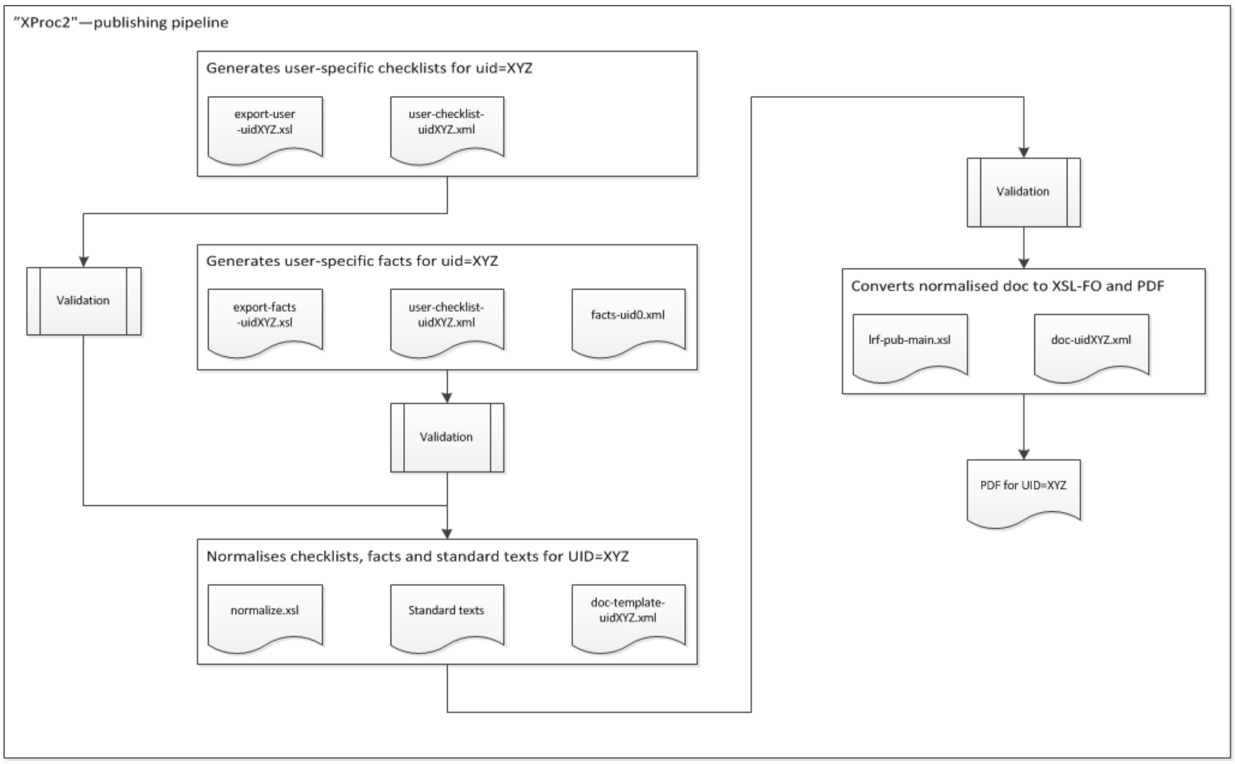

"XProc2"—publishing pipeline
Generates user-specific checklists for uid=XYZ
export-user -uidXYZ.xsl
user-checklist-uidXYZ.xml
Validation
Generates user-specific facts for uid=XYZ
export-facts -uidXYZ.xsl
user-checklist-uidXYZ.xml
facts-uid0.xml
Validation
Validation
Normalises checklists, facts and standard texts for UID=XYZ
normalize.xsl
Standard texts
doc-template-uidXYZ.xml
Validation
Converts normalised doc to XSL-FO and PDF
lrf-pub-main.xsl
doc-uidXYZ.xml
PDF for UID=XYZ

Figure 11. Publishing a User-specific PDF: Xproc2

The first pipeline configures the second one with specific options (such as how to mark up the draft questions and facts in the PDF or which ones to leave out in a particular review) and, once the user has made his choices using a GUI generated by the process XML describing the second pipeline, generates a batch file to run the configured pipeline with.

7. Some Final Comments

As this is being written, a first version of the authoring environment has been released to authors. The authors are currently busy updating facts, converted from the rather shaky HTML fragments to a basic DocBook format. The review functionality - being able to add drafts to the publishing chain and publish them for review - is not in place yet and probably won't be in time for XML Prague. instead I'm currently writing report scripts and stylesheets that aid authors in finding incomplete content.

The inclusion of authoring means a paradigm shift that probably wasn't planned. Editing was needed basically to have the means to produce the missing standard texts, but now, the editor also provides facts. In other words, most of the content is handled in eXist while the checklist questions and their logic is still in the SQL database. The dissatisfaction with the admin GUI and the cost involved in adapting it can easily result in them being moved to eXist as well. It would most likely be far more effective to handle everything in one place; now, two different interfaces are required when producing checklist content.

Bibliography

[1] James Sulak: eXist XProc Library https://github.com/jsulak/eXist-XProc-Library

[2] Ari Nordström: Processing XML Using XML http://www.balisage.net/Proceedings/vol8/html/Nordstrom01/BalisageVol8-Nordstrom01.html

Bringing NoSQL Datastores into an XQuery Playground

Juan Zacarias
Oracle
<juan.zacarias@oracle.com>

Matthias Brantner
28msec
<matthias.brantner@28msec.com>

Cezar Andrei
Oracle
<cezar.andrei@oracle.com>

Abstract

JSONiq is an extension of XQuery making JSON a first class citizen within XQuery. The paper proposes three JSONiq modules to interface with three major NoSQL key-value data stores - namely MongoDB, Couchbase Server, and Oracle NoSQL DB. Using JSONiq and those three modules NoSQL developers can use the power of XQuery to process JSON data in their favorite NoSQL Datastore.

1. Introduction

In the last few years, there has been a flurry of activity in the area of distributed data storage and processing going under the name of the "NoSQL" movement named after "Not Only SQL." It has two main goals: (1) being able to process flexible schema data (e.g., JSON or XML) and (2) provide scalability for data processing. In order to achieve those goals, different kinds of architectures have been proposed under this very name.

In this paper, we are focusing our attention on three major key-value NoSQL data stores: Mongo, CouchBase, and Oracle NoSQL DB, that allow the storage and access of binary data, or JSON in particular. JSON is currently used as a popular data format for data with flexible schema, and it is used side-by-side together with XML in current architectures. Unlike the XML databases with their use of XQuery as a sophisticated data processing language, the JSON data stores such as Mongo, CouchBase, or Oracle NoSQL DB, do not use a structured query language (or any query language) to process (query, index, update) JSON data. If they do, the languages they use are usually very limited in functionality, often limited to search by key and simple inserts and replace updates.

The lack of declarative query languages for JSON, and standard ones even less, is a major limitation in processing large amounts of JSON data productively and effectively. Last year the JSONiq query language has been proposed as a solution to this problem, and since then it is receiving increasing attention in the technical communities. JSONiq is an extension of XQuery, adding the JSON data model as first class citizen to the XML Data Model. Being based on XQuery, it has the advantage of exploiting decades of research, standardization, and implementation work around processing flexible data. Querying JSON data is also the primary focus for the W3C XQuery Working group for the XQuery 3.1 extension, and such an extension might be standardized as part of W3C in the near future.

In this paper, we propose three JSONiq modules to interact, extract, query, update JSON data stored on the three NoSQL stores that we considered. We report on the experience with integrating the NoSQL databases into an existing XQuery playground, namely Zorba. Given that JSONiq is capable of querying both XML and JSON seamlessly, the three APIs allow users to mix and match data extracted from NoSQL stores together with data stored in XML databases, effectively bringing JSON query processing as par with XML.

The paper is organized as follows. First, we will briefly describe JSONiq and its relationship with XQuery, mostly through examples that run seamlessly on the three data stores. Then, we will describe each store in particular. For each store, we will give a general description, an assessment of the current query capabilities, the XQuery/JSONiq module API that we propose that that respective module, and give some examples of how to interact with it (connect, disconnect, load data) from within XQuery.

2. JSONiq

JSONiq is a small and simple set of extensions to XQuery that add support for JSON. Being an extension of XQuery, JSONiq exploits the fundamental features of XQuery:

- the powerful FLWOR construct, that allows specification of selects, joins, outer-joins, group-by, windowing, and sorting in a compact, productive and optimizable manner.
- the functional nature of the language, extremely useful in processing hierarchical data with no a priori known structure. In particular the functions and recursive functions, as well as the high order functions, are essential for this purpose.
- the modules, that allow extensibility and integration with external environments.
- the declarative, snapshot based, data updates.
- the powerful full text search that uses sophisticated search capabilities (e.g. thesaurus).

- a sophisticated type system, using a complete atomic types set (e.g. dates or durations), and processing libraries on such types.

In order to be able to also query JSON data, JSONiq add the following extensions to XQuery:

- Extensions to the XQuery Data Model (XDM) to support JSON (i.e. objects and arrays).
- Support for JSON's datatypes (e.g null or boolean), with a mapping to equivalent XML Schema types.
- Navigation for JSON Objects and JSON Arrays.
- Constructors for JSON Objects and JSON Arrays using the same syntax as JSON.
- Update primitives against JSON items as well as updating expressions that produce them.
- New item types that extend sequence type matching to allow the type of JSONiq datatypes to be specified in function parameters, return types, and other expressions that specify XQuery types.

Please note that the three APIs proposed below, as well as the work of integrating NoSQL data stores into an XQuery playground, are relatively orthogonal to the JSONiq language per se. In the future, they can be easily adapted to any XQuery extension that the W3C community might standardize.

2.1. Example Dataset & Queries

For the purpose of demonstrating the JSONiq language, we are going to show example queries on a sample data set. This data set contains information about the location, population, city, and state for every zip code in the United States. It can be found at http://media.mongodb.org/zips.json. The data is given as a collection of 29469 JSON documents. One of them is shown in the following example:

```
{
  "_id": "10280",
  "city": "NEW YORK",
  "state": "NY",
  "pop": 5574,
  "loc": [ -74.016323, 40.710537 ]
}
```

Based on this dataset, we would like to showcase the power of JSONiq using three example queries. In all of the queries below, we assume that the find(“zips”) function call returns all the zip documents from the zips collection.

The first query selects all the states having a population over 10 million from the collection.

```
for $zip in find("zips")
let $state := $zip("state")
group by $state
let $pop := sum($zip("pop"))
where $pop > 10000000
order by $pop descending
return
  {
    "state" : $state,
    "population" : sum($pop)
  }
```

First, this uses the find function to retrieve all the objects from the collection. In later sections, we will show how this function can be implemented on top of any of the three NoSQL Datastores. Further, the program groups all objects by state and selects only the states whose overall population is bigger than 10 million. The result is returned as a sequence of JSON objects containing the state and population in descending order of the population.

The second query computes the average city population by state.

```
for $state-city in
  for $zip in find("zips")
  let $state := $zip("state")
  let $city := $zip("city")
  group by $state, $city
  return
    {
      "state" : $state,
      "pop" : sum($zip("pop"))
    }
group by $state := $state-city("state")
return
  {
    "state" : $state,
    "avgCityPop" : avg($state-city("pop"))
  }
```

Similar to the previous query, this one starts by retrieving all documents from the collection. Those documents are then grouped by state and city. For each group, an intermediate sequence of objects is constructed. Each object of the sequence contains the name of the state as well as the overall population of the cities in this state. To return the result, the intermediate sequence is grouped again - this time by state - in order to average the population of all cities in a particular state.

The third query selects the largest and smallest cities per state.

```
for $zip in find("zips")
group by $state   := $zip("state")
```

```
let $smallestPop  := min($zip("pop"))
let $smallestCity := $zip[.("pop") eq $smallestPop][1]
let $biggestPop   := max($zip("pop"))
let $biggestCity  := $zip[.("pop") eq $biggestPop][1]
return {
    "state": $state,
    "biggestCity": {
      "name": $biggestCity("city"),
      "pop": $biggestCity("pop")
  },
  "smallestCity": {
    "name": $smallestCity("city"),
    "pop": $smallestCity("pop")
  }
}
```

To select the largest and smallest city per state, the query starts by grouping the documents by state. For each group, the smallest and biggest cities are computed. The result is returned by constructing a new object containing all the extracted information.

3. MongoDB

MongoDB is a scalable, high-performance, open source NoSQL database. It allows for storing and querying collections of JSON documents. It uses BSON (short for Binary JSON) to store those documents. In addition to JSON's primitive types (i.e. string, boolean, null, and number), BSON also supports some other types such as byte, int, double, binary, datetime, or timestamp.

The database has it's own query language which generally works by creating "query documents" that indicate the patterns of the keys and values that are to be matched in the query.

3.1. JSONiq API for MongoDB

We developed a JSONiq module providing functions that expose the most common MongoDB operations, i.e. functionality that is available in supported host languages. In the following, we present some of the most important functions available in this module. You can find the full module including all the functions at http://my.zorba-xquery.com/tmp/XMLPrague2013/mongodb.xq.

- mongo:connect($options as object()) as xs:anyURI
- mongo:disconnect($db as xs:anyURI) as empty-sequence()
- mongo:collection-names($db as xs:anyURI) as xs:string*
- mongo:find($db as xs:anyURI, $coll as xs:string) as object()*

- mongo:find($db as xs:anyURI, $coll as xs:string, $query as object(), $options as object(), $projection as object()) as object()*
- mongo:save($db as xs:anyURI, $coll as xs:string, $doc as object()) as empty-sequence()
- mongo:update($db as xs:anyURI, $coll as xs:string, $query as object(), $update as object()) as empty-sequence()
- mongo:remove($db as xs:anyURI, $coll as xs:string, $remove as object()) as empty-sequence()

3.1.1. Connecting

Connecting to the database is a simple as passing the connection information (e.g. host and database) as JSON object to the connect function. The connect function returns an opaque identifier (as xs:anyURI) to the database. All other functions take such an identifier as their first parameter.

```
import module namespace m = "http://www.28msec.com/modules/mongodb";

let $db := m:connect( { "host" : "localhost", "port" : 27017, "db" : "my-db" }
return (: do something with the connection $db :)
```

3.1.2. Storing Data

The save function could be used to load the zips data from a text file into MongoDB.

```
for $zip in jn:parse-json(file:read-text("zips.json"))
return
m:save($db, "zips", $zip)
```

As already mentioned, MongoDB uses BSON to store data. Because BSON extends the JSON data model with some other primitive types (e.g. int, timestamp), the module defines a mapping from BSON types to types of the JSONiq data model (JDM). For example, BSON's 32bit integers are mapped to xs:int. BSON types that don't have a JDM counterpart, are represented using user-defined atomic types. One example, is the timestamp type which is a user-defined type derived from xs:long for representing BSON timestamp values. Such a mapping allows for round-tripping documents without losing the types of the data.

3.1.3. Accessing Data

In order to read data from a collection, the module's find function can be used. The two argument version of the function returns all the objects of the given collection. The five argument version shown above also allows for filtering and projecting the resulting documents. The filtering and projection parameter are JSON objects because MongoDB's uses JSON to express queries and projections.

Given the find function, the query to retrieve the states with the biggest population could look as follows:

```
import module namespace m = "http://www.28msec.com/modules/mongodb";

let $db := m:connect( { "host" : "localhost", "port" : 27017, "db" : "my-db" }
for $zip in m:find($db, "zips")
let $state := $zip("state")
group by $state
let $pop := sum($zip("pop"))
where $pop > 10000000
order by $pop descending
return
  {
    "state" : $state,
    "population" : sum($pop)
  }
```

In order to retrieve only the names of the cities of California, the find function could be used as follows:

```
db:find($db, "zips", { "state" : "CA" }, { "city" : 1}, {||} )
```

The last argument allows for providing options to MongoDB. In this example, the options object is an empty object.

Similar to the query shown above, all the other example queries can be modified to query data in MongoDB.

3.1.4. Implementation Notes

The implementation of the MongoDB module has been done in C++ based on Zorba's API for building external modules. It uses MongoDB's C++ driver for the communication with MongoDB. The implementation of all the functions is relatively straightforward and has approx. 1300 lines of code. However, this doesn't include the mapping from JDM items to BSON items and vice-versa. This mapping is relatively complex because there is no obvious one-to-one mapping between those types. Designing and implementing such a mapping was the hardest part and needed another 3100 lines of code.

4. Couchbase Server

Couchbase Server is a distributed key-value database. As you would expect from a key-value database, it allows for efficiently storing and retrieving arbitrary key-value pairs using get and put functions. In the latest version, Couchbase also allows for querying data using Views. Views are a way to extract particular fields and information out of your raw data and to produce an index of this information. This concept allows for iterating, selecting, and querying the data in the database and goes beyond basic key-value retrieval. Most importantly, this allows for efficiently storing and querying JSON documents.

4.1. JSONiq API for Couchbase Server

Similar to the MongoDB module, we also developed a JSONiq module that provides functions to interact with Couchbase Server. Again, please find some of the most important functions of this module below. The full module with all functions can be found at http://my.zorba-xquery.com/tmp/XMLPrague2013/couchbase.xq.

- cb:connect($options as object()) as xs:anyURI
- cb:get-text($db as xs:anyURI, $key as xs:string*) as xs:string*
- cb:get-binary($db as xs:anyURI, $key as xs:string*) as xs:base64Binary*
- cb:put-text($db as xs:anyURI, $key as xs:string*, $value as xs:string*) as empty-sequence()
- cb:put-binary($db as xs:anyURI, $key as xs:string*, $value as xs:base64Binary*) as empty-sequence()
- cb:remove($db as xs:anyURI, $key as xs:string*) as empty-sequence()
- cb:create-view($db as xs:anyURI, $doc-name as xs:string, $view-name as xs:string*) as xs:string*
- cb:create-view($db as xs:anyURI, $doc-name as xs:string, $view-name as xs:string*, $options as object()*) as xs:string*
- cb:view($db as xs:anyURI, $path as xs:string*) as xs:string*
- cb:view($db as xs:anyURI, $path as xs:string*, $options as object()) as xs:string*

4.1.1. Connecting

Just like the MongoDB module, the connect function can be used to connect to the database. It's as simple as passing the connection information as JSON object to the function.

```
import module namespace cb =
  "http://www.zorba-xquery.com/modules/couchbase";

let $db := cb:connect({
  "host": "localhost:8091",
  "username" : null,
  "password" : null,
  "bucket" : "default"})
return (: do something with the connection $db :)
```

4.1.2. Storing Data

To import the data from the zips.json file into Couchbase Server, each JSON object is stored as text using the zip code (i.e. value of the _id field) as key.

```
for $zip in jn:parse-json(file:get-text("zips.json"))
return
  cb:put-text($db, $zip("_id"), fn:serialize($zip))
```

4.1.3. Accessing Data

The simplest way to access the data is by using the get-text or get-binary functions. The former returns the value for a given key as xs:string. The latter returns the value as xs:base64Binary.

Additional to the basic get functions, we designed the API to allow for creating and querying Views. This is particularly important for JSONiq because Couchbase has no other way for listing all or a subset of the keys/objects.

A view can be created using the create-view function. Essentially, it allows the user to create a map using some simple options. In the following example, the key for the map is the name of the state and the value is the population.

```
cb:create-view($db, "document-name", "view-name", {"key" : "doc.state", "values" : "doc.pop"});
```

The view function can be used to return the contents of the view given the name returned by create-view:

```
jn:parse-json(cb:view($db, $view-name))("rows")
```

This would return a sequence of JSON objects as follows:

```
{ _id="94303", key="CA", value="24309" }
```

If multiple values where specified in the view definition, the values would be returned in an array.

Putting this together, the first example query could look as follows:

```
import module namespace cb =
  "http://www.zorba-xquery.com/modules/couchbase";

let $db := cb:connect({ "host": "localhost:8091", … })
let $view-name := cb:create-view(
  $instance,
  "document-name",
  "view-name",
  {"key" : "doc.state", "values" : "doc.pop"})
for $zip in jn:parse-json(cb:view($instance, $view-name))("rows")
```

```
let $state := $zip("key")
group by $state
let $pop := sum($d("value"))
where $pop > 10000000
order by $pop descending
return
  {
    "state" : $state,
    "population" : $pop
  }
```

The view definition for the third example query would also need to project the name of the city:

```
cb:create-view(
  $instance,
  "document-name",
  "view-name",
  {"key" : "doc.state", "values" : ["doc.pop", "doc.city"]})
```

4.1.4. Implementation Notes

The implementation of the Couchbase Server module has been done in C++ using the Zorba API for building external modules and the Couchbase Server's C++ API. The implementation of the module is a straightforward communication between Zorba's and Couchbase's APIs. The most complex part is the communication between JSONiq and Couchbase's View functions. However, the code remained simple and required less than than 1300 lines of code.

5. Oracle NoSQL DB

Oracle NoSQL DB is a distributed key-value database. It is built upon the proven Oracle Berkeley DB Java Edition high-availability storage engine. It is designed to provide highly reliable, scalable, and available data storage across a configurable set of systems that function as storage nodes. In addition to that it adds a layer of services for use in distributed environments. The resulting solution provides distributed, highly available key-value storage that is well suited to large-volume, latency-sensitive applications.

The database uses a composite key of multiple strings such that the multi-get and multi-delete functions can specify only the prefix part of the key or a range of keys. This gives more functionality to the user and also the application can be designed to make use of this processing power of the database.

5.1. JSONiq API for Oracle NoSQL DB

The JSONiq module defines the usual key value store functions: put, get, delete as well the connect and disconnect, but also the more advanced multi-get and multi-delete functions. Below, please find a summary of the most important functions. The full module is available at http://my.zorba-xquery.com/tmp/XMLPrague2013/nosqldb.xq.

- nosql:connect($options as object()) as xs:anyURI
- nosql:disconnect($db as xs:anyURI) as empty-sequence()
- nosql:get($db as xs:anyURI, $key as xs:string) as xs:string
- nosql:put($db as xs:anyURI, $key as xs:string, $value as xs:string) as xs:long
- nosql:delete($db as xs:anyURI, $key as xs:string) as xs:boolean
- nosql:multi-get-base64($db as xs:anyURI, $parentKey as object(), $subRange as object(), depth as xs:string, $direction as xs:string) as object()*
- nosql:multi-get-string($db as xs:anyURI, $parentKey as object(), $subRange as object(), $depth as xs:string, $direction as xs:string) as object()*
- nosql:multi-get-json($db as xs:anyURI, $parentKey as object(), $subRange as object(), $depth as xs:string, $direction as xs:string) as object()*
- nosql:multi-delete-values($db as xs:anyURI, $parentKey as object(), $subRange as object(), $depth as xs:string) as xs:int

5.1.1. Connecting

As previously presented modules, the connect function takes the address of the database as arguments and it returns a connection identifier. This identifier is used by all of the other functions to discriminate between multiple connections. For example:

```
import module namespace nosql =
  "http://www.zorba-xquery.com/modules/oracle-nosqldb";

let $db := nosql:connect({
  "store-name" : "kvstore",
  "helper-host-ports" : ["localhost:5000"] }
)
return (: do something with the connection $db :)
```

5.1.2. Storing Data

Since Oracle NoSQL DB supports composed key values, we chose to load the data having the major key part 'zips' and under minor key part the zip-code '_id'.

```
for $zip in jn:parse-json( file:read-text("zips.json") )
return
  nosql:put-json($db,
    { "major" : ["zips"],
    "minor": [{$zip("_id")}] },
    $zip )
```

5.1.3. Accessing Data

As for any key-value store, the simplest way to access data is using the get function. On top of that, there are more advanced functions that allow to return multiple objects at once. For example, in order to retrieve all zip documents from the database as JSON, the multi-get-json function can be used as follows:

```
import module namespace nosql =
  "http://www.zorba-xquery.com/modules/oracle-nosqldb";

let $db := nosql:connect({ "store-name" : "kvstore", … })
for $zip in nosql:multi-get-json($db,
  { "major" : "zips" }, {"prefix": ""},
  $nosql:depth-PARENT_AND_DESCENDANTS,
  $nosql:direction-FORWARD)("value")
let $state := $zip("state")
group by $state
let $pop := sum($zip("pop"))
where $pop > 10000000
order by $pop descending
return
  {
    "state": $state,
    "population" : $pop
  }
```

5.1.4. Implementation Notes

Oracle NoSQL module is implemented using NoSQL's Java client library. The module covers the most important aspects of the NoSQL API, leaving out many API details for corner cases. This makes the module simple to use while powerful with access to more advanced compound keys and multi get/delete functionality. The implementation takes only 2700 lines of code but it's making use of the Java library infrastructure that is available in Zorba.

6. Conclusion & Outlook

In this paper, we have presented three modules each of which allows interfacing with a NoSQL database using JSONiq. Particularly, we have shown modules for the NoSQL databases MongoDB, Couchbase Server, and Oracle NoSQL DB. The goal of those modules is to allow NoSQL developers to use the full power of JSONiq with data in their favorite NoSQL database.

In the future, we would like to define modules to other NoSQL stores (e.g. Amazon DynamoDB). Also, we want to share our insights and findings with the W3C working group in order to officially integrate JSON data processing into XQuery. In the meantime, we hope that the proposed connectors help NoSQL database vendors to get a feeling for the benefits of using a sophisticated data processing language. We believe that the expressive power of JSONiq/XQuery together with the scalability of NoSQL Datastores make a great/killer combination.

References

[1] MongoDB Module: http://my.zorba-xquery.com/tmp/XMLPrague2013/mongodb.xq and http://my.zorba-xquery.com/tmp/XMLPrague2013/mongodb.xsd.xml

[2] Couchbase Server Module: http://my.zorba-xquery.com/tmp/XMLPrague2013/couchbase.xq

[3] Oracle NoSQL DB Module: http://my.zorba-xquery.com/tmp/XMLPrague2013/nosqldb.xq

[4] JSONiq Language Specification: http://www.jsonig.org/

[5] MongoDB: http://www.mongodb.org/

[6] Couchbase Server: http://www.couchbase.org/

[7] Oracle NoSQL DB:http://www.oracle.com/technetwork/products/nosqldb/overview/index.html

[8] Zorba: http://www.zorba-xquery.org/

Jiří Kosek (ed.)

XML Prague 2013
Conference Proceedings

Published by
Ing. Jiří Kosek
Filipka 326
463 23 Oldřichov v Hájích
Czech Republic

PDF was produced from DocBook XML sources
using XSL-FO and XEP.

1st edition

Prague 2013

ISBN 978-80-260-3872-6