XML Prague 2012

Conference Proceedings

University of Economics, Prague
Prague, Czech Republic

February 10–12, 2012

XML Prague 2012 – Conference Proceedings

ISBN 978-80-260-1572-7

<oXygen/> XML Editor

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.

- **Visual XML Authoring**
- **DITA and DocBook Ready**
- **Single Source Publishing**
- **CMS Integration**
- **Highly Configurable and Extensible**

XML Development

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

- **Intelligent XML Editing**
- **Visual Schema Modeling**
- **XSLT and XQuery Debugging**
- **XML Databases**
- **Integrated Tools**

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 64 USD.
All editions can run as a standalone application or as an Eclipse IDE plugin, on Windows 7, Vista, XP, 2000, Mac OS X, Linux and Solaris.

Table of Contents

General Information

Date

Friday, February 10th, 2012 (preconference day)
Saturday, February 11th, 2012
Sunday, February 12th, 2012

Location

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

Organizing Committee

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

Programm Committee

Robin Berjon, *freelance consultant*
Petr Cimprich, *Ubiqway*
Daniela Florescu, *Oracle*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *University of Economics, Prague*
Uche Ogbuji, *Zepheira LLC*
Petr Pajas, *Google*
Adam Retter, *freelance consultant*
Felix Sasaki, *German Research Center for Artifical Intelligence*
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

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

Sponsors

oXygen (http://www.oxygenxml.com)
Mercator IT Solutions Ltd (http://www.mercatorit.com)

Preface

This publication contains papers presented at XML Prague 2012.

XML Prague is a conference on XML for developers, markup geeks, information managers, and students. In its seventh year, XML Prague focuses especially on new advances in XQuery and on integration of XML with new web technologies. The conference provides an overview of successful XML technologies, with the focus being more towards real world application versus theoretical exposition.

XML Prague conference takes place 10–12 February 2012 at the campus of University of Economics in Prague. XML Prague 2012 is jointly organized by the XML Prague Organizing Committee and by the Faculty of Informatics and Statistics.

The full program of the conference is broadcasted over the Internet (see http://xmlprague.cz) – XML fans from around the world are encouraged to take part on-line. Remote and local participants are visible to each other and all have got a chance to interact with speakers.

This is the seventh year we have organized this event. For this year we have changed location to the larger venue in order to satisfy raising interest in XML Prague. For the first time we have extended the conference by additional pre-conference day—in three parallel tracks we have provided space for various XML community meetings.

We hope that all introduced changes are going in the right direction and that you will enjoy XML Prague 2012.

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

The eX Markup Language?

Eric van der Vlist
Dyomedea
<vdv@dyomedea.com>

Abstract

Revisiting the question that was the tag line of XML Prague last year: "XML as new lingua franca for the Web. Why did it never happen?", Eric tries to answer to other questions such as: "where is XML going?" or "is XML declining, becoming an eX Markup Language?".

1. XML as new lingua franca for the Web. Why did it never happen?

This was the tagline of XML Prague 2011, but the question hasn't really been answered last year and I'll start this talk to give my view on that question.

1.1. Flashback

February 1998 is a looong time ago, a date from another century and for those of you who were not born or don't remember, here is a small summary of what did happen in February 1998:

February

- *The United States Senate*[4] *passes Resolution 71, urging U.S. President Bill Clinton*[5] *to "take all necessary and appropriate actions to respond to the threat posed by Iraq*[6]*'s refusal to end its weapons of mass destruction programs."*
- *February 3*[7] *– Cavalese cable car disaster*[8]*: a United States Military*[9] *pilot causes the deaths of 20 people near Trento*[10]*, Italy, when his low-flying plane severs the cable of a cable-car.*

[4] http://en.wikipedia.org/wiki/United_States_Senate
[5] http://en.wikipedia.org/wiki/Bill_Clinton
[6] http://en.wikipedia.org/wiki/Iraq
[7] http://en.wikipedia.org/wiki/February_3
[8] http://en.wikipedia.org/wiki/Cavalese_cable_car_disaster_%281998%29
[9] http://en.wikipedia.org/wiki/United_States_Military
[10] http://en.wikipedia.org/wiki/Trento

- *February 4*[11] *– An earthquake measuring 6.1 on the Richter scale*[12] *in northeast Afghanistan*[13] *kills more than 5,000 people.*
- *February 7*[14]*–February 22*[15] *– The 1998 Winter Olympics*[16] *are held in Nagano*[17]*, Japan.*
- *February 16*[18] *– China Airlines Flight 676*[19] *crashes into a residential area near Chiang Kai-shek International Airport*[20]*, killing 202 people (all 196 on board and 6 on the ground).*
- *February 20*[21] *– Iraq disarmament crisis*[22]*: Iraqi President Saddam Hussein*[23] *negotiates a deal with U.N. Secretary General Kofi Annan*[24]*, allowing weapons inspectors to return to Baghdad*[25]*, preventing military action by the United States and Britain.*

—Wikipedia

While the Iraq disarmament crisis was raging, the World Wide Web Consortium waited until the third day of the Winter Olympics held in Nagano to make the following announcement:

> *Advancing its mission to lead the Web to its full potential, the World Wide Web Consortium (W3C)*[27] *today announced the release of the XML 1.0 specification*[28] *as a W3C Recommendation. XML 1.0 is the W3C's first Recommendation for the Extensible Markup Language, a system for defining, validating, and sharing document formats* ***on the Web***
>
> *—W3C Press Release (February 1998)*

People curious enough to click on the second link of the announcement could easily double check that beyond the marketing bias XML was something to be used over the Internet:

[11] http://en.wikipedia.org/wiki/February_4
[12] http://en.wikipedia.org/wiki/Richter_magnitude_scale
[13] http://en.wikipedia.org/wiki/Afghanistan
[14] http://en.wikipedia.org/wiki/February_7
[15] http://en.wikipedia.org/wiki/February_22
[16] http://en.wikipedia.org/wiki/1998_Winter_Olympics
[17] http://en.wikipedia.org/wiki/Nagano
[18] http://en.wikipedia.org/wiki/February_16
[19] http://en.wikipedia.org/wiki/China_Airlines_Flight_676
[20] http://en.wikipedia.org/wiki/Chiang_Kai-shek_International_Airport
[21] http://en.wikipedia.org/wiki/February_20
[22] http://en.wikipedia.org/wiki/Iraq_disarmament_crisis
[23] http://en.wikipedia.org/wiki/Saddam_Hussein
[24] http://en.wikipedia.org/wiki/Kofi_Annan
[25] http://en.wikipedia.org/wiki/Baghdad
[27] http://www.w3.org/pub/WWW/
[28] http://www.w3.org/TR/1998/REC-xml-19980210

The design goals for XML are:

1. ***XML shall be straightforwardly usable over the Internet.***
2. ***XML shall support a wide variety of applications.***
3. *XML shall be compatible with SGML.*
4. *It shall be easy to write programs which process XML documents.*
5. *The number of optional features in XML is to be kept to the absolute minimum, ideally zero.*
6. *XML documents should be human-legible and reasonably clear.*
7. *The XML design should be prepared quickly.*
8. *The design of XML shall be formal and concise.*
9. *XML documents shall be easy to create.*
10. *Terseness in XML markup is of minimal importance.*

—W3C Recommendation (February 1998)

And the point was reinforced by the man who had led the "Web SGML" initiative and is often referred to as the father of XML:

> *XML arose from the recognition that key components of the original web infrastructure -- HTML tagging, simple hypertext linking, and hardcoded presentation -- would not scale up to meet the future needs of the web. This awareness started with people like me who were involved in industrial-strength electronic publishing before the web came into existence.*
>
> *—Jon Bosak*

This has often been summarized saying that XML is about "putting SGML on the Web".

Among the design goals the second one ("XML shall support a wide variety of applications") has been especially successful and by the end of 98, Liora Alschuler[31] reported that the motivations of the different players pushing XML forward were very diverse:

> *The big-gun database vendors, IBM and Oracle, see XML as a pathway into and out of their data management tools. The big-gun browser vendors, Netscape and Microsoft, see XML as the e-commerce everywhere technology. The big-gun book and document publishers, for all media, are seeing a new influx of tools, integrators, and interest but the direction XML publishing will take is less well-defined and more contingent on linking and style specs still in the hands of the W3C.*
>
> *—Liora Alschuler for XML.com (December 1998)*

[31] http://web.archive.org/web/19991011215212/http://www.xml.com/pub/au/Alschuler_Liora

One thing these "big-gun" players that were pushing XML to different directions did achieve has been to develop an incredible hype that rapidly covered everything and in 2001 the situation had become hardly bearable:

> *Stop the XML hype, I want to get off*
>
> *As editor of XML.com, I welcome the massive success XML has had. But things prized by the XML community — openness and interoperability — are getting swallowed up in a blaze of marketing hype. Is this the price of success, or something we can avoid?*
>
> *—Edd Dumbill (March 2001)*

Marketers behind the hype being who they were, the image of XML that they promoted was so shiny that the XML gurus didn't recognize their own technology and tried to fight against the hype:

> *I've spent years learning XML / I like XML / This is why www.XmlSuck.com is here*
>
> *—PaulT (January 2001)*

The attraction was high and people rushed to participate to the W3C working groups:

> ***Working Group size** - so many people means it is difficult to gain consensus, or even know everyone's face. Conference calls are difficult.*
>
> *—Mark Nottingham, about the SOAP W3C WG (May 2000)*

Huge working groups with people pushing to different directions is not the best recipe to publish high quality standards and even though XML itself was already baked, the perception of XML depends on the full "stack":

> *This is a huge responsibility for the Schema Working Group since it means that the defects of W3C XML Schema will be perceived by most as defects of XML.*
>
> *—Eric van der Vlist on xml-dev (April 2001)*

The hype was so huge that XML geeks rapidly thought that they had won the war and that XML was everywhere:

> *XML is now as important for the Web as HTML was to the foundation of the Web. XML is everywhere.*
>
> *—connet.us (February 2001)*

Why this hype? My guess is that the IT industry had such a desperate need for a data interchange format that any one of them could have been adopted at that time and that XML happened to be the one that went through the radar screen at the right moment:

> *When the wind is strong enough, even flatirons can fly.*
>
> *—Anonymous (February 2012)*

The W3C had now to maintain:

- XML, a SGML subset
- HTML, a SGML application that did not match the XML subset

Technically speaking, the thing to do was to refactor HTML to meet the XML requirements. Given the perceived success of XML, it seemed obvious that everyone would jump into the XML wagon and be eager to adopt XHTML.

Unfortunately from a web developer perspective the benefits of XHTML 1.0 were not that obvious:

> *The problem with XHTML is :*
> *a) it's different enough from HTML to create new compatibility problems.*
> *b) it's not different enough from HTML to bring significant advantages.*
> *—Eric van der Vlist on XHTML-DEV (May 2000)*

It is fair to say that Microsoft had been promoting XML since the beginning:

> ***XML, XML, Everywhere***
> *There's no avoiding XML in the .NET world. XML isn't just used in Web applications, it's at the heart of the way data is stored, manipulated, and exchanged in .NET systems.*
> *— Rob Macdonald for MSDN (February 2001)*

However, despite their strong commitment to XML, Microsoft had frozen new developments on Internet Explorer. The browser has never been updated to support the XHTML media type, meaning that the few web sites using XHTML had to serve their pages as HTML!

By 2001, the landscape was set:

- XML had become a dominant buzzword giving a false impression that it had been widely adopted
- Under the hood, many developers were deeply upset by this hype even among the XML community
- Serving XHTML web pages as such was not an option for most web sites

The landscape was set, but the hype was still high and XML was still gaining traction as a data interchange format.

In the meantime, another hype was growing...

Wikipedia has tracked the origin of the term Web 2.0 back to 1999:

> *The Web we know now, which loads into a browser window in essentially static screenfuls, is only an embryo of the Web to come.*
> *.../...*
> *Ironically, the defining trait of Web 2.0 will be that it won't have ant visible characteristics at all. The Web will be identified only by its underlying DNA structure-- TCP/IP (the protocol that controls how files are transported across the Internet);*

HTTP (the protocol that rules the communication between computers on the Web), and URLs (a method for identifying files).

.../...

The Web will be understood not as screenfuls of text and graphics but as a transport mechanism, the ether through which interactivity happens.

—Darcy DiNucci (1999)

The term became widely known with the first Web 2.0 conferences in 2003 and 2004 and XML was an important piece of the Web 2.0 puzzle through Ajax (Asynchronous JavaScript and XML), coined and defined by Jesse James Garrett in 2005 as:

Ajax isn't a technology. It's really several technologies, each flourishing in its own right, coming together in powerful new ways. Ajax incorporates:

- *standards-based presentation*[42] *using XHTML and CSS;*
- *dynamic display and interaction using the Document Object Model*[43]*;*
- *data interchange and manipulation using XML and XSLT*[44]*;*
- *asynchronous data retrieval using XMLHttpRequest*[45]*;*
- *and JavaScript*[46] *binding everything together.*

—Jesse James Garrett (February 2005)

This definition shows how, back in 2005, some of us still thought that XML could dominate the Web and be used both to exchange documents (in XHTML) and data.

Unfortunately, this vision defended by the W3C, has been rapidely torpedoed by Ian Hickson[47] and Douglas Crockford[48].

Founded in 1985 for that purpose, the W3C had been the place where HTML had been normalized. Among other things, the W3C had been the place where the antagonists of the first browser war could meet and discuss in a neutral field.

In 2004, Netscape had disappeared, Microsoft had frozen the development of their browser and browser innovation moved into the hand of new players: Mozilla, Apple/Safari and Opera who was starting to gain traction.

Complaining that the W3C did not meet their requirements and that HTML needed to be updated urgently to meet the requirements what would be soon known as Web 2.0, they decided to fork the development of HTML:

[42] http://www.adaptivepath.com/publications/essays/archives/000266.php
[43] http://www.scottandrew.com/weblog/articles/dom_1
[44] http://www-106.ibm.com/developerworks/xml/library/x-xslt/?article=xr
[45] http://www.xml.com/pub/a/2005/02/09/xml-http-request.html
[46] http://www.crockford.com/javascript/javascript.html
[47] http://en.wikipedia.org/wiki/Ian_Hickson
[48] http://en.wikipedia.org/wiki/Douglas_Crockford

> *Software developers are increasingly using the Internet as a software platform, with Web browsers serving as front ends for server-based services. Existing W3C technologies — including HTML, CSS and the DOM — are used, together with other technologies such as JavaScript, to build user interfaces for these Web-based applications.*
>
> *However, the aforementioned technologies were not developed with Web Applications in mind, and these systems often have to rely on poorly documented behaviors. Furthermore, the next generation of Web Applications will add new requirements to the development environment — requirements these technologies are not prepared to fulfill alone. The new technologies being developed by the W3C[50] and IETF[51] can contribute to Web Applications, but these are often designed to address other needs and only consider Web Applications in a peripheral way.*
>
> *The Web Hypertext Applications Technology working group therefore intends to address the need for one coherent development environment for Web Applications. To this end, the working group will create technical specifications that are intended for implementation in mass-market Web browsers, in particular Safari, Mozilla, and Opera.*
>
> *—WHATWG (June 2004)*

The W3C was behind a simple choice: either push XHTML recommendations that would never be implemented in any browsers or ditch XHTML and ask the WHATWG to come back and continue their work toward HTML5 as a W3C Working Group. The later option was eventually chosen and HTML work resumed within W3C in 2007.

JSON was around since 2001. It took a few years of Douglas Crockford's energy to popularize this JavaScript subset but around 2005, JSON rapidly became a technology of choice as a "Fat-Free Alternative to XML[52]" in Ajax applications.

There is no direct link between HTML5 and JSON but the reaction against XML, its hype and its perceived complexity is a strong motivation in both cases.

1.2. Why?

A number of reasons can be found for this failure:

- Bad timing between the XML and HTML specifications (see Adam Retter's presentation at XML Amsterdam 2011[53]).
- Lack of quality of some XML recommendations (XML Namespaces, XML Schema, ...).

[50] http://www.w3.org/TR/
[51] http://www.ietf.org/rfc.html
[52] http://www.json.org/fatfree.html
[53] http://www.adamretter.org.uk/presentations/xml-and-web-technologies_xml-amsterdam_20111026.pdf

- Lack of pedagogy to explain why XML is the nicer technology on the earth.
- Dumbness of Web developers who not use XML.
- ...

There is some truth in all these explanations, but the main reason is that from the beginning we (the XML crowd) have been arrogant, over confident and have made a significant design error.

When we read this quote:

> *XML arose from the recognition that key components of the original web infrastructure --* ***HTML tagging, simple hypertext linking, and hardcoded presentation -- would not scale up to meet the future needs of the web****. This awareness started with people like me who were involved in industrial-strength electronic publishing before the web came into existence.*
>
> *—Jon Bosak*

We all understand what Jon Bosak meant and we probably all agree that HTML is limited and that something more extensible makes our lives easier, but we must also admit that we have been proven wrong and that HTML has been enough to scale up to the amazing applications we see today.

Of course, the timing was wrong and everything would have been easier if Tim Berners-Lee had came up with a first version of HTML that would have been a well formed XML document but on the other hand, the web had to exist before we could put SGML on the web and there had to be a prior technology.

In 1998 it was already clear that HTML was widespread and the decision to create XML as a SGML subset that would be incompatible with HTML has been a bad one:

- Technically speaking because that meant that millions of existing pages would be non well formed XML ("the first Google index in 1998 already had 26 million pages[55]").
- Tactically speaking because that could be understood as "what you've done so far was crappy, now you must do what we tell you to do".

To avoid this deadly risk, the first design goal of XML should have been that existing valid HTML documents were well formed XML documents. The result might have been a more complex format and specification, but this risk to create a gap between XML and HTML communities would have been minimized.

Another reason to explain this failure is that XML is about extensibility. This is both its main strength and weakness: extensibility comes at a price and XML is more complex than domain specific languages.

Remove the need for extensibility and XML will always loose against DSLs, we've seen a number of examples in the past:

[55] http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

- RELAX NG compact syntax
- JSON
- HTML
- N3
- CSS
- ...

2. Is it a time to refactor XML? Converge or convert?

Hmmm... It's time to address the questions asked this year by XML Prague!

We've failed to establish XML as **the** format to use on the web but we've succeeded in creating a strong toolbox which is very powerful to power websites and exchange information.

I don't know if it's to compensate the ecosystems that we are destructing on our planet, but one of the current buzzwords among developers is "ecosystem": dominant programming languages such as Java and JavaScript are becoming "ecosystems" that you can use to run a number of applications that may be written using other programming languages.

What we've built with XML during the past 14 years is a very strong ecosystem.

The XML ecosystem is based on an (almost) universal data model that can not only represent well formed XML documents but also HTML5 documents and (with an impedance mismatch that may be reduced in future versions) JSON objects.

> **Note**
> Notable exceptions that cannot be represented by the XML data model include overlapping structures and graphs.

On top of this data model, we have a unique toolbox that includes:

- transformation and query languages
- schema languages
- processing (pipeline) languages
- databases
- web forms
- APIs for traditional programming languages
- signature and encryption standards
- a text based serialization syntax
- binary serialization syntaxes

We can truly say that what's important in XML is not the syntax but that:

Angle Brackets Are a Way of Life

—Planet XMLHack

Rather than fighting fights that we've already lost we need to develop our ecosystem.

The number one priority is to make sure that our data model embraces the web that is taking shape (which means HTML5 and JSON) as efficiently as possible. Rather than converge or convert we must embrace, the actual syntax is not that important after all!

To grow our ecosystem, we could also consider embracing more data models, such as graphs (RDF), name/value pairs (NOSQL), relations (SQL), overlaps (LMNL).

I am more skeptical about refactoring XML at that stage.

It's always interesting to think about what could be done better, but refactoring a technology as widespread as XML is tough and needs to be either backward compatible or provide a huge benefit to compensate the incompatibilities.

Will we see a proposal that will prove me wrong during the conference?

XML and HTML Cross-Pollination: A Bridge Too Far?

Norman Walsh
MarkLogic Corporation
<norman.walsh@marklogic.com>

Robin Berjon
Robineko
<robin@berjon.com>

Abstract

W3C created a small Task Force to look at convergence paths between XML and HTML. One of the notions that it discussed was that the two technologies could perhaps not be aligned, but that they could cross-pollinate.

This talk will look at ways for this to happen. Can we use CSS Selectors in XPointer? Can we build something like XSLT using CSS + JavaScript and another syntax? Does HTML actually have some interesting approaches to distributed extensibility? Should SVG be in the HTML namespace and has it suffered from changing its syntax?

Keywords: XML, HTML, W3C

1. Introduction

Over a year ago, the W3C put together a small XML-HTML Task Force that was asked to look into the convergence between these two technology families. This paper is derived from the experience of two of the task force's participants in looking at the two ecosystems in order to figure out what they could share.

While it appears that complete alignment between the HTML and XML families may not be achievable (or in fact desirable), there are nevertheless areas in which cross-pollination between those two stacks could help improve either or both. Despite the important design differences that exist between HTML and XML, their goals are not as divided as some of the stormier rhetoric suggests. There are plenty of areas of common experience despite significant differences in the details.

This paper will therefore navigate a number of cross-over options. Some of these work today but may not be in common use, at times because they could in fact be detrimental, but in a number of cases possibly because they have been overlooked by those who could benefit from them. Other parts may not work because of a

technological gap. Yet others may seem no less than utter madness, but could nevertheless constitute an interesting avenue for exploration.

2. XML in Today's Browser

XML is used on the Web only in moderation, but this does not prevent it from being available as a technology in browsers. In this section we look at what can be done with XML on the Web, what's missing and how that could be addressed, and at the overall worthiness of the notion.

2.1. XML, CSS, Javascript

It is readily possible, using today's browsers, to simply ship XML + CSS on the Web. The style sheets that one will have to craft will be more complex than those often used for HTML since they will have to define the basic properties of every single element, but that does not constitute a major undertaking (unless the given language is huge, naturally).

A good starting point for the above would be one of the many CSS reset style sheets that can be found at large. The goal of these is to reset the default styles that browsers apply (which differ) so as to start from a common stylistic baseline. Adapting one of those to work for an XML vocabulary should prove straightforward. CSS can be applied to an XML document using an `xml-stylesheet` processing instruction.

For example:

```
<?xml-stylesheet href="xmltest.css"?>
<doc>
<title>Title</title>
<para xml:id="test">Testing.</para>
</doc>
```

Where `xmltest.css` contains:

```
doc {
    display: block;
}

title {
    display: block;
    font-size: 18pt;
    font-weight: bold;
}

para {
    display:block
```

```
    margin-before: 1em;
}
```

Inlined images are slightly trickier than the rest, but it is at least theoretically possible to address that issue with a combination of several `background` properties and the `attr()` function which can reuse the value of an attribute inside a CSS property. (Though this does not appear to be supported today.)

Of course, once you've styled your document there isn't much that you can do with it beyond read it. In a number of cases that may be sufficient, but the interactivity that one has grown accustomed to on the Web will not be directly possible. For that, one needs support for forms and Javascript.

XForms might spring to mind as a good option for the inclusion of forms in your XML document, but given that it is not natively supported in browsers a client-side Javascript implementation will be required to support it (or to implement form-like functionality based on another vocabulary), which brings us back to scripting.

There is no equivalent to `xml-stylesheet` for scripting, but a `script` element in the XHTML namespace will do the trick:

```
<?xml-stylesheet href="xmltest.css"?>
<doc>
<title>Title</title>
<para xml:id="test">Testing.</para>

<script xmlns="http://www.w3.org/1999/xhtml">
//<![CDATA[
var text = document.createTextNode("Script testing");
var para = document.getElementsByTagName("para")[0];
para.removeChild(para.firstChild)
para.appendChild(text);
//]]></script>
</doc>
```

If using XML documents in this fashion were to become more widespread, the idea of am `xml-script` processing instruction might become interesting (it should not prove difficult to specify).

Once this is set up, most of the moving parts ought to be functional. Since the DOM is properly live, just like any HTML DOM, modifications to the tree will have the effect on the rendering that you would expect. By and large, popular libraries like jQuery will (mostly) work. One thing worth noting though is that since the XML DOM has a number of difference with the HTML DOM (if only that the elements won't be properly specialised to a given interface), a number of tasks may prove more complex (and common libraries may stumble on some cases).

Support for `xml:id` attributes in particular would make locating elements in the XML DOM easier.

2.2. What, no links?

One thing is missing from the above picture though: linking. XLink is not supported in any generally reusable manner (it is recognised on select SVG elements, but that's about it). Assuming Javascript and CSS, this can be emulated. For a DocBook document for instance, we would have the following CSS:

```
@namespace "http://docbook.org/ns/docbook";
@namespace xl "http://www.w3.org/1999/xlink";
link[xl|href] {
    cursor: hand;
    color:  #00c;
    text-decoration: underline;
}
```

And navigation would work using Javascript (assuming jQuery and its xmlns plugin):

```
$.xmlns("http://docbook.org/ns/docbook");
$.xmlns("xl", "http://www.w3.org/1999/xlink");
$("article").on("click", "link[xl|link]", function () {
    document.location = $(this)[0].getAttributeNS("http://www.w3.org/1999/▶
xlink", "href");
});
```

While this works, it is hardly the best for all link-based navigation to be handled in script. There have, in the past, been proposals that endeavoured to decorate documents in such a way that links would be recognised (for instance, using a style sheet that would identify link elements as such) but none ever gained much traction. If using XML on the Web were to be commonplace, this would likely be one of the first gaps that would require filling.

2.3. The Accessibility of XML

Theoretically, accessibility of XML should be at least as good, perhaps better than HTML because the opportunity exists for expressing richer semantics in the document. In practice, this is utterly wrong. Had XML become widespread on the web, languages for mapping accessibility onto XML documents could have been developed. Since it didn't, they weren't and the result is that HTML documents have much greater accessibility because so much is known in advance about the semantics of the elements.

Perhaps ARIA and CSS would provide a framework for building some accessibility into XML on the web, but it's not likely to be sufficient for the more complex cases.

2.4. Why bother?

You probably shouldn't. If you subscribe to the many worlds interpretation of quantum mechanics, you may be able to imagine worlds where support for XML on the web is widespread and robust. In those universes, deploying XML documents on the web, transforming, styling, and scripting them is as easy and straightforward as deploying HTML documents in our universe. But we don't live there.

In this universe, the benefits from deploying XML on the web are probably limited to closed environments where it's convenient to think of the documents that will be deployed on the web as being equally useful for additional processing inside the environment. Extracting linked data, performing entity enrichment, etc. using existing XML technologies on top of the web documents may have real value.

But actual deployment is probably limited to XML documents that happen to also be HTML documents or that generate the HTML DOM you expect when parsed with an HTML parser.

3. Born in XML, Live in HTML

There are a great many technologies developed for XML that could (and do!) apply equally well to HTML. If you move beyond the question of syntax to the level of an object model, an XML tree and an HTML tree are even closer together than documents written in their respective syntaxes.

It would be nice to be able to use them when they offer the easiest way forward.

3.1. XPointer and CSS

XPointer and CSS both describe mechanisms for selecting regions of a document. In fact, XPointer is really a framework for such mechanisms. An XPointer implementation that understood a “CSS scheme” would allow authors to reuse expressions across styling and selection.

See, for example, Using CSS Selectors as Fragment Identifiers[1] by Simon St. Laurent Eric Meyer.

3.2. Can we replace FO?

At least in some environments, CSS has made good progress in its support for print-related styles. Unfortunately, we don't yet seem to have reached the point where formats like PDF can fully be replaced on the Web.

A good source of gaps in the current HTML+CSS ecosystem could therefore be FO. The pdf.js project has shown that it is possible to parse and render PDF using

[1] http://simonstl.com/articles/cssFragID.html

only Web technologies. The time therefore seems ripe to try to do the same thing with FO.

Using such an approach it may be that we will be able to use FO directly in the browser, which at least could help solve PDF's many issues such as the difficulty in generating it easily, its various text-selection problems, and at least some of its more endemic accessibility problems (through judicious application of HTML and ARIA).

More likely though it could highlight both some priorities in the evolution of CSS, and the value that there could be for implementers to support them in order to progressively phase out PDF.

3.3. Compilation to JS

Another approach is simply to bring those tools directly into the browser. We've seen clear demonstrations of substantial success in these areas in the work of Saxonica[2]'s on "Saxon Client Edition" and Vojtěch Toman on XML Pipeline Processing in the Browser[3].

3.4. How SVG made the jump

SVG was born and raised an XML language. It has its own namespace fully defined with a proper namespace document. It has a RelaxNG as well as an NVDL schema; in fact at one point or another it has even had a DTD and an XML Schema. It uses XLink, and version 1.2 even adopted `xml:id`. Over the years it has routinely been both consumed and produced using XSLT and XPath; for a while there even was a project to add "SVG Extensions to XPath" in order to process its geometry better.

This is interesting because it is not all that common for graphics people to care, or even know much about, XML. A large part of the reason for it to use XML in the first place was because it was intended to be embedded straight into XHTML, or of course any other XML language that could use some pretty shapes.

But with XHTML pining for the fjords, the question of what to do with SVG became rather acute. Browsers already largely supported it, so that there was little point in dropping it entirely. Besides, there was no proposed replacement and starting from scratch would have taken too long. The decision was therefore made to find a way to integrate it with HTML.

By and large, given that several of the important integration factors had already been figured out for XHTML integration and did not need to be updated, considering the size and complexity of SVG this turned out to be a lot easier than one may expect. However, the matter of syntax remained a problem. Keeping the XML syntax created

[2] http://saxonica.com/
[3] http://www.balisage.net/Proceedings/vol5/html/Toman01/BalisageVol5-Toman01.html

trouble at the HTML parsing level since one would have to switch parsers. What's more, embedding different syntaxes that are mostly the same but also subtly different inside one another is hardly the friendly thing to do to developers. After some intense debate, the decision was made to accept the transition of SVG to the HTML syntax, along with other niceties like automatic namespacing.

A number of participants in this discussion predicted trouble from this switch, notably due to cut and paste errors. In practice, now several years after the switch took place, very few complaints have been heard. On the contrary, the merged syntax has made it easier to better support SVG on the Web, and SVG is increasingly becoming mainstream thanks to that and the common usage of libraries that make use of this facility such as Raphaël. In fact it is such a success that the SVG WG is considering dropping SVG's own namespace entirely and automatically coercing it all into the HTML namespace, one way or another.

An interesting open question here is whether there are other languages that could be given the same treatment.

3.5. Distributed Extensibility on the Web

Much discussion has surrounded a property of "Distributed Extensibility" that XML is supposedly endowed with (by virtue of namespaces) when HTML is not. But is that concern justified?

The first thing to note is that saying that XML supports distributed extensibility is not a very useful statement to make. XML is just a syntax, and if you couldn't freely create languages with it it's not clear what it would be useful for. A more precise characterisation is that, given that namespaces can be used to mix vocabularies, XML *languages* support distributed extensibility. Anyone can grab an XML document that uses a given language, slap an extra namespace declaration, and start injecting attributes and elements from a different language without hurting the first.

The ability to extend a language independently of the agent that controls it is a very powerful and seductive one. However, in order for it to be genuinely useful, you need more than extensibility at the syntactical and semantic level. You also need the same extensibility to be at the very least harmless through most of the processing chain that is applied to the original language.

And that's where things start becoming less clear for distributed extensibility in the XML ecosystem. While building a processing tool chain that supports distributed extensibility in XML is technically possible, drove after drove of users have voted with their feet against it. In order to support DE, one would expect a schema language to accept by default namespaces that it is not aware of, yet the most popular choices in this area — XML Schema and RelaxNG — make this difficult. There is very little in the way of tooling support for manipulating trees that may compose multiple namespaces together while easily the ones that are not expected to be

present. And in practice, code is rarely written with that expectation in mind. Try throwing in additional namespaced elements into a DOM, SAX, XSLT, etc. processing pipeline and the chances are high that you will see it fail.

A number of best practices were initially worked on (in part in order to support versioning) that described useful rules for ignoring content from distributed extensibility, but there was never sufficient interest to finalise them. A technology like NVDL could prove helpful in a DE-enabled tool chain but it is hardly ever used — in fact the vast majority of XML users are likely never to have heard of it. RDDL (and namespace documents in general) which could make DE additionally useful by building discoverability into the system barely elicited sufficient interest to foster a few short-lived proposals.

It is therefore fair to say that while distributed extensibility was part of the initial XML vision, in practice it failed to see the light of day.

Meanwhile, work has progressed on a promising new HTML-based technology called Web Components[4]. Geared towards interactive documents, it makes it possible to decorate an existing document with any number of "shadow DOMs" that can hang off its elements, and can themselves contain further shadow DOMs recursively. The current work is still in its infancy, but it has roots in previous work called XBL which served a similar purpose in a different fashion.

The great value of shadow DOMs is that they make it possible to process independent yet composed document trees without any risk of seeing their processing chains step on one another's toes. It is too early to know if they could be transposed to batch operations easily or if that approach would even be workable in an XML context, but it's a space worth watching. It might just happen that the XML ecosystem could import concepts from Web Components in order to develop its own support for distributed extensibility.

3.6. Web Transformations

The Javascript community is presently seeing a small cottage industry in the production of templating languages. In fact, there are days on which one may get the impression that you can't be a proper Javascript developer if you haven't released your own templating language to GitHub.

String-interpolation templating languages are great when you need to generate HTML from a rather straightforward data structure, but they start to become cumbersome when your input increases in complexity, especially if it's a document itself.

For that, XSLT is currently king. But try as you might, getting your typical web hacker to even think about perhaps using it is nigh impossible. Back in 1998, a submission called STTS[5] was made to the W3C. It used a CSS based declarative

[4] http://dvcs.w3.org/hg/webcomponents/raw-file/tip/spec/shadow/index.html
[5] http://www.w3.org/TR/NOTE-STTS3

language to transform documents (primarily HTML back then), but never met with strong support. Re-evaluating it in today's context, it is interesting to note that it bears some similarities with the now popular HAML[6] templating language. That being said, its highly declarative nature is likely to make it only slightly more popular than XSLT with this crowd.

Would it be possible to implement a language or library built on the principles that make XSLT a great language but using an approach that would be familiar to Web developers? Reimplementing the full feature-set of an XSLT processor is not a minor undertaking, but a decent amount of it can be handled if we have JavaScript handy. Besides, since this is just a thought experiment at this point, we can probably stick to only porting XSLT's core processing model and see how far that gets us. That is a much smaller endeavour, in fact the XSLT processing model is so elegantly short that we can simply reproduce it here from the specification in its unabridged entirety:

> *A list of source nodes is processed to create a result tree fragment. The result tree is constructed by processing a list containing just the root node. A list of source nodes is processed by appending the result tree structure created by processing each of the members of the list in order. A node is processed by finding all the template rules with patterns that match the node, and choosing the best amongst them; the chosen rule's template is then instantiated with the node as the current node and with the list of source nodes as the current node list. A template typically contains instructions that select an additional list of source nodes for processing. The process of matching, instantiation and selection is continued recursively until no new source nodes are selected for processing.*
>
> *Implementations are free to process the source document in any way that produces the same result as if it were processed using this processing model.*
>
> *—XSLT 1.0, James Clark*

That doesn't solve all of our problems of course, but it's a start. Based on this we could imagine some form of JSLT code that might look like the following if we wanted to transform a simple DocBook document to HTML.

```
var sheet = require("jaspilite").sheet()
,   $     = require("jquery")
,   XL    = "http://www.w3.org/1999/xlink"
;
sheet.xmlns({
    "":   "http://docbook.org/ns/docbook"
,   "xl": XL
});
sheet.template(":root > article", function () {
```

[6] http://haml-lang.com/

```
    var title = $(this).find("info > title").text();
    return $("<html><head><title></title></head><body><h1></h1><content/></▶
body></html>")
              .find("title").text(title).end()
              .find("h1").text(title).end()
              .find("content").replaceWith($(this).apply());
});
sheet.template("info", function () {});
sheet.template("section", function () {
    return $(this).apply().wrapIn("<section></section>");
});
sheet.template("title", function () {
    var depth = $(this).parents("section").length;
    if (depth < 1) return;
    depth++;
    if (depth > 6) depth = 6;
    return $(this).apply().wrapIn(sheet.createElement("h" + depth));
});
sheet.template("para", function () {
    return $(this).apply().wrapIn("<p></p>");
});
sheet.template("emphasis", function () {
    return $(this).apply().wrapIn("<em></em>");
});
sheet.template("link[xl|href]", function () {
    var $link = $(this);
    return $link.apply().wrapIn("<a></a>").attr("href", ▶
$link[0].getAttributeNS(XL, "href"));
});
var $out = sheet.run("my-document.dkb");
console.log($out.html());
```

There are naturally some issues with the above code, but as a transformation sheet it is usable and ought to seem roughly familiar to JS developers in large part thanks to its reliance on jQuery. The first problem is namespaces: jQuery (and its underlying CSS engine, Sizzle) is not natively aware of namespaces. This can be worked around using a plugin, but it still shows when one tries to access a namespaced attribute. That being said, one thing that may not be readily obvious when used to XPath is that all of the CSS Selectors above use the DocBook namespace properly since unlike XPath CSS Selectors can have a default declaration applied to them.

Another issue is with injecting variables and finding a substitute for `apply-templates` in the output tree. As can be seen in the first template, the way in which that is done is by first creating the output structure, and then addressing some of its part to inject a value or replace a dummy element. That may seem cumbersome, but jQuery makes it mercifully short. What's more, as can be seen in

the other templates, all that's needed if one wishes to merely wrap the `apply-templates` in an element is to use the `wrapIn()` method which will simply do the right thing.

Finally, the most likely limitation with this approach has to do with just how limited CSS Selectors are. For a trivial example they'll shine, but try so much as to find a text node and you'll hit a wall. This could be fixed either by inventing new selectors (which is not necessarily difficult since jQuery is very extensible in this area) or by making it possible to use XPath in the same context. The latter might be a workable option as a more power alternative that one can reach for when needed.

Of course this is just a thought experiment and it is likely to have many more failings. There is no doubt that the crafty minds of XSLT experts will be able to find many an XSLT construct that could be difficult to reproduce here. That said, crazier ideas have become popular.

3.7. CSS Schema

The XML ecosystem is big on validation, so big in fact that drinking games involving reciting long lists of validation technologies had to be banned after too many xml-dev subscribers were found suffering of delirium tremens.

On the HTML side however, there is not that much to be found. There are, naturally, a number of HTML validators out there, but they tend not to be easily extensible for the sort of context-specific rules that one may wish to overlay on HTML. JSON Schema is being worked on, but it applies only to JSON documents.

Naturally, a lot of the XML validation toolkit can be applied to HTML documents with some relatively manageable amount of shoe-horning. But most of it at best uses moving parts that are unfamiliar to most Web developers, and at worst is used as the boogeyman with which young web hackers are kept in check.

In the spirit of reusing good ideas across the cultural gap, one schema language that lends itself well both to the common validation problems in an HTML context (which are mostly validation overlaid atop HTML itself) and to concepts that can be translated with some degree of ease is Schematron.

Taking the basic notions of rules and assertions, we could imagine (again, as a thought experiment) a CSS Schema language that could look like the following:

```
@rule head {
    title {
        assert:  "Page does not have a title.";
    }
    link[rel="stylesheet"][type="text/css"][href="std.css"] {
        assert:  "Page does not use the standard stylesheet.";
    }
    style {
        report:  "Page uses inline style rather than linking.";
```

```
    }
}
@rule body {
    :scope.std-body {
        assert:   "Page does not use the standard body class.";
    }
    :scope > div.std-top {
        assert:   "Page does not start with the required page top component.";
    }
}
```

The above is syntactically valid CSS that merely adds an `@rule` block and new properties called `assert` and `report` that all correspond to the same constructs in Schematron. Selectors are used in the same manner that XPath is. Of note is the `:scope` selector which can be used to select the current scope, matching the default context that XPath can use.

This can be used equally well offline or in a browser. One addition that could be interesting would be simply to specify regular style properties (e.g. `background: red;`) to the assertions so that when running in the browser the failure of an assertion would cause the style to be applied, making for nice visual feedback.

And since CSS can be applied interactively, this could notably be used in order to provide visual feedback inside browser-based document-editing systems.

4. Conclusion

Characterizing the relationship between the XML and HTML communities is largely a matter of perspective. At a high enough level, everyone is using technologies of one sort or another to manipulate trees in various ways. From this distance, there's almost no means to distinguish them. At the very closest level, there are persistent and irksome disagreements over the nature of error handling, mechanisms for distributed extensibility, even the virtue (or lack thereof) of extensible vocabularies. From this distance, the common ground is hard to see.

Surely, the reality is somewhere in between and there are opportunities to learn, share, harmonize, and diverge and join, for everyone.

XML5's Story

Anne van Kesteren
Opera
<annevankesteren@gmail.com>

1. Background

The short story is that XML is hard. There have been a lot of web developers that have tried to output XML to the browser and only a few have successfully succeeded in doing so. This is probably mostly because outputting XML is a vastly different exorcise from outputting HTML and given how similar they appear, this is not immediately obvious. With HTML it is okay to make a mistake, forgetting to escape an ampersand in "Black & White". With XML it is not. To output XML you need to build an internal XML structure for which an XML serializer will not yield any errors. And herein lies the problem, most HTML is simply generated by string concatenation. String concatenation is an easy concept to grasp for developers, working on a tree on the other hand and serializing said tree is at least an order of magnitude more difficult to grasp. In an article titled HOWTO Avoid Being Called a Bozo When Producing XML[3] Henri Sivonen lays out carefully all the steps that need to be taken to produce XML and not have it fall over. Although a lot of it is good advice for HTML too, none of it is required to publish some HTML on the web.

The trickier errors that have plagued developers trying to use XML for their site are errors at the encoding level, which are fatal in XML. Back in the blogging days there was a Trackback concept and quite often octets encoded in one way would end up on a page encoded in another way (e.g. windows-1252 encoded octets in a utf-8 encoded resource). For HTML documents this only gave some information loss (though you could usually still follow the URL), for XML documents an external Trackback could stop it from being an XML document (i.e. non-well-formed). This has happened to e.g. Sam Ruby and Matt Mullenweg though they typically fixed the problem fairly quickly.

Back in Evan Goer did a survey labeled the The XHTML 100[4] where he studied the sites of what he called the "Alpha Geeks". Out of the 119 sites he looked at, only one was completely fine. Mark Pilgrim had a similar experience looking at feeds (RSS, Atom) and went as far as stating that XML on the Web has failed[5].

Part of the problem here was that Internet Explorer did not support XHTML (so people ended up transmitting XHTML as HTML) and popular feed consumers were not actually using XML parsers. Popular browsers on phones back in the days were

[3] http://hsivonen.iki.fi/producing-xml/
[4] http://www.goer.org/Journal/2003/04/the_xhtml_100.html
[5] http://www.xml.com/pub/a/2004/07/21/dive.html

not much better, Simon Pieters revealed they performed poorly when it came to XML[6]. For contrast, as things stand now feed consumers are stricter, but feeds are less popular. Internet Explorer supports XHTML now, but the web developer community is mostly concerned with HTML (again), and the phone market has completely changed, aligning it much more with the general web.

2. Extensibility

Back in 2007 when I started looking at whether we could improve XML, HTML had non-draconian error handling and XML had the ability to include other languages. In particular, HTML did not support SVG and MathML when using HTML syntax. XBL 2.0 was also proposed around that time and planned to be a new XML-based vocabulary. SVG and MathML were already implemented to some extent in popular browsers and interest in XBL was there too. To begin using these on normal pages however web developers would have to switch to XML which is a pretty big step. My idea was that we could make that step drastically smaller by giving XML the same feature HTML has, non-draconian error handling.

Nowadays of course SVG and MathML can be used in HTML's HTML syntax. And XBL became the Component Model and will be HTML-based. These events make a potential XML5 less appealing, because the desired feature set (as seen from a web developer) is already met.

3. XML5

XML's extensibility story combined with it being a lot harder to use is what led to XML5. A version of XML fully backwards compatible with XML 1.0 documents that could also handle resources that were not technically XML documents, but still carried an XML MIME type. In other words, a version of XML that would not halt at well-formedness errors.

Whether XML5 is something still worth pursuing today is up for debate. The web developer community seems to have moved to HTML and feeds are becoming less popular. XML is still used as interchange format, but end users are no longer exposed to it. Well, almost, well-formedness violations are still problematic to some extent. Recently Andreas Bovens announced Opera would no longer show violations of XML well-formedness[7]. This is only for resources fetched through a browsing context and is mostly a user agent sniffing problem, but does show that people are still not quite competent enough to meet all the requirements. In Tim Bray's words, "bozos".

[6] http://simon.html5.org/articles/mobile-results
[7] http://my.opera.com/ODIN/blog/2011/09/28/no-more-xml-parsing-failed-errors

XML5 also came up as one of the ways HTML and XML could converge more. Per the HTML/XML Task Force Report[8]. (The W3C TAG has yet to publish it "officially".)

4. Principles

With those open questions, lets take a closer look at what was proposed back then as XML5. And in particular, how XML5 was designed. XML5 was inspired by the work done on HTML by Ian Hickson to define the HTML parser. The HTML parser can handle any given input and always produces a tree as a result. The resulting tree is not necessarily conforming to any standard, but if an octet went missing or a developer missed a mistake only revealing itself under certain circumstances the end user will be able to see the information provided by the resource. So the parser should continue in the face of erroneous input.

Already mentioned was fully backwards compatible with XML 1.0. Documents conforming to the XML 1.0 grammar have to be processed identically. Otherwise an XML 1.0 parser could not be replaced.

Even erroneous input should not affect the streaming nature of the parser. The HTML parser requires a tree to be kept in memory as some non-conforming content cannot be handled without it. XML5 should not have that problem.

Although there is some XML that comes with a schema, parsing of XML should not require it. This may lead to suboptimal handling of certain errors in a given vocabulary, but what is more important is that any given input will result in the same output across all XML parsers, whether they are schema-aware or not.

Which leads to the last principle, just as XML (if we ignore its optional feature for now) is fully deterministic now, XML5 should be fully deterministic too. Dealing with all potential input must be defined. Defining XML5 similar to how the HTML parser is defined today makes this very feasible.

5. The parser

Writing a conceptual XML5 parser has been done and there is an XML5 Playground[9] available online where you can test how it handles attribute values not delimited by quotation marks, for instance. The source code is available as well: xml5 project[10]. This implementation does not deal with octet decoding (just code points; characters if you will), but if we assume it does, the mostly theoretical XML5 parser would consist of three parts: input stream, tokenizer, and tree construction.

[8] http://www.w3.org/2010/html-xml/snapshot/
[9] http://quuz.org/xml5/play
[10] http://code.google.com/p/xml5/

In all these stages you have to deal with input that might be incorrect. The XML5 parser in question made certain decisions here, however there is no XML5 standard so the specifics are up for debate. They server to illustrate the concept.

E.g. in a shift_jis encoded document you might find a lead octet without a matching trailing octet. This would be caught in the input stream phase. Error handling from `text/plain` and `text/html` suggests the first octet will be turned into a U+FFFD code point and the trail octet will be looked at again. (The exact details of encoding error handling are somewhat ill-defined at this point.)

In the tokenizer stage when you tokenize a start tag you might encounter a code point that is neither whitespace nor a single/double quotation mark after an equal sign. The XML5 parser treats this as the first code point of the attribute's value. A < code point encountered when tokenizing the start tag's name, would simply become part of the name as it is neither whitespace nor a > code point.

The tree builder then deals with the cases where you encounter an end tag for a start tag that is not on the stack (ignored), and mistakes in the use of namespaces.

6. Stray thoughts

Once such a fault tolerant system is in place additions such as </> (closes currently open element) becomes easier as well. As with all new features uptake might take a while, but at least you know legacy clients (post XML 1.0 though) will not fall over.

Another thing to look at if XML is going to be touched is XML's optional feature (external parsed entities) and whether there is a way to phase that out. XML is also made non-deterministic by being open ended on encodings. And since encodings are generally not well defined, what constitutes well-formed XML and what not might be a bit of a challenge if you look at e.g. some octet combinations in the hz-gb-2312 encoding.

7. Closing

Thanks for taking the time to read it through. Looking forward to discuss the material in person!

XProc: Beyond application/xml

Vojtěch Toman
EMC Corporation
<vojtech.toman@emc.com>

Abstract

Although primarily an XML processing language, XProc is increasingly being used in environments that involve processing of non-XML data. However, the limited support – or lack thereof – for non-XML media types in XProc poses real issues, both for pipeline authors as well as performance-wise. This article looks at some of these issues and explores the possibilities of extending XProc to support processing of both XML and non-XML data.

Keywords: XProc, XML, XPath, MIME, data integration

1. Introduction

Unbelievable as it is, in May 2012 it will have been two years since the XProc specification [5] became a W3C Recommendation; surely enough time and backward perspective for the W3C XML Processing Working Group to start assessing the successes (and failures) of XProc in the real world – and to begin ruminating about new features and enhancements for the next version of the language, should there ever be one.

One of the interesting enhancement ideas[1] discussed within the working group is to provide better support for other media types than just XML media types. While supporting non-XML media types may seem outside the scope of what XProc is supposed do (after all, it is an XML pipeline language, and the XProc specification describes XProc as "a language for describing operations to be performed on XML documents," clearly indicating what the primary focus of XProc is), practical experience shows that besides processing XML data, real-life XProc pipelines often deal with non-XML data in one way or another. This data either comes from external sources (a JSON response returned by a web service or image data read from a file), or is produced by the pipeline itself (a PDF document or a ZIP archive representing an EPUB or an ODF document).

Unfortunately, support for non-XML media types is rather limited in XProc, as the language is based on the XML data model exclusively. The specification even states that: "Although some steps can read and write non-XML resources, what

[1] See the XProc V.Next wiki at http://www.w3.org/wiki/XprocVnext for more ideas that are being considered.

flows between steps through input ports and output ports are exclusively XML documents or sequences of XML documents." In practice, this means that in order for non-XML data to flow through the pipeline, the data must be wrapped in an XML element wrapper and typically also (in the case of non-text media types) base64-encoded. This is not only inefficient, but it also makes dealing with such data rather tedious for pipeline authors. For instance, a seemingly simple task such as processing a JSON message from a web service is almost impossible to do in standard XProc.

This article explores the implications of introducing non-XML media types to XProc, and proposes a possible scheme for implementing seamless support for both XML and non-XML media types into the language. Most of the ideas discussed in this article come from experiments with Calumet[2], EMC's XProc processor, which will likely include support for non-XML media types as an experimental, "at your own risk" feature.

2. Current level of support for non-XML media types in XProc

As already mentioned, the support for processing non-XML data is rather rudimentary in standard XProc. Obviously, this is mainly due to the specification-stipulated absolute that what flows in the pipeline can only be XML documents: XProc steps thus can only accept XML documents on their input ports and can only produce XML documents on their output ports. (By extension, this applies to complete pipelines as well since they are steps, too.) The options of how to deal with non-XML in XProc pipelines are therefore quite limited.

2.1. Using an external channel

Because non-XML data cannot flow through the pipeline, steps that process or produce such data often have to rely on an external channel (usually the file system) and refer to the data using URI references. This is a common technique in real-life XProc pipelines, one that has been used also by the XProc specification itself, as exemplified by the step `p:xsl-formatter` from the standard XProc step library:

```
<p:declare-step type="p:xsl-formatter">
  <p:input port="source"/>
  <p:input port="parameters" kind="parameter"/>
  <p:output port="result" primary="false"/>
  <p:option name="href" required="true"/>
  <p:option name="content-type"/>
</p:declare-step>
```

[2] http://developer.emc.com/xmltech/

The p:xsl-formatter step takes an XSL document, renders the content and stores the result to the URI specified by the href option. What appears on the result output port of p:xsl-formatter is a small XML document with the URI reference to the generated output.

The p:xsl-formatter step is an example of a step that takes XML input and produces non-XML data, but the same approach of using URI options is applicable also to steps that take non-XML input and produce XML, or that are non-XML at both ends. The common problem (especially with the latter), however, is how to ensure the right sequencing of such steps in the pipeline. Consider the following step, which processes data from the location specified via the source-href option and writes the result to the location specified in the result-href option:

```
<p:declare-step type="ex:process-binary-data"
                xmlns:ex="http://www.example.org/ns/xproc">
  <p:option name="source-href" required="true"/>
  <p:option name="result-href" required="true"/>
</p:declare-step>
```

The step has no input and output ports, so extra care needs to be taken when using it in the pipeline. Because the step has no ports that other steps in the pipeline can connect to, the order in which the steps in the pipeline will be executed may not be deterministic. In the pipeline fragment below, the execution order of the steps ex:generate-binary-data and ex:process-binary-data depends entirely on the XProc processor, because the steps are not (explicitly nor implicitly) connected: if we are lucky, then ex:generate-binary-data will be executed first, but there is nothing preventing the XProc processor from executing the steps in the opposite order, the consequences of which would be, most likely, fatal.

```
<p:pipeline>
  ...
  <ex:generate-binary-data href="output.data"/>
  <ex:process-binary-data source-href="output.data"
                          result-href="output2.data"/>
  ...
</p:pipeline>
```

The above situation can be worked around by using the more verbose p:with-option-based syntax for specifying the step options and using helper p:pipe bindings to enforce dependencies on other steps, or by wrapping problematic steps in compound steps with dummy input and output ports to which other steps can connect to. But all this would not be necessary if non-XML data could flow through the pipeline directly. If this were possible, the declaration of the ex:process-binary-data step might look as simple as this:

```
<p:declare-step type="ex:process-binary-data"
                xmlns:ex="http://www.example.org/ns/xproc">
```

```
  <p:input port="source" primary="true"/>
  <p:output port="result" primary="true"/>
</p:declare-step>
```

Using such a step would be almost as easy as using the standard `p:identity` step, and the pipeline authors would benefit greatly from the convenience of steps having primary input and output ports (automatic connections between subsequent steps etc.). Returning to the pipeline fragment from the previous example (and changing the declaration of `ex:generate-binary-data` in the same fashion as with `ex:process-binary-data`), the pipeline could then be rewritten as below:

```
<p:pipeline>
  ...
  <ex:generate-binary-data/>
  <ex:process-binary-data/>
  ...
</p:pipeline>
```

Such a pipeline would be safer (the execution order of the steps would be fixed because of the implicit connection between `ex:generate-binary-data` and `ex:process-binary-data`), easier to write, and probably also more efficient when run.

2.2. Base64-encoding

When pipeline authors want to (or are forced to) pass non-XML data from one step to another without using an external channel, a solution is to wrap the data in an XML wrapper element and base64-encode the data if it cannot be represented as a sequence of Unicode characters. This is exactly how XProc deals with non-XML data that the pipeline reads from an external location using the `p:data` binding or the `p:http-request` step.

The `p:http-request` step can be seen as a dynamic and more powerful version of the `p:data` binding. The `p:data` binding reads arbitrary resources from a fixed URI, performing the necessary wrapping and base64-encoding to represent the data as XML. The `p:http-request` supports using dynamically constructed URIs and additional HTTP-specific features such as authentication, headers etc., but the basic logic of how non-XML data is exposed to the pipeline is the same:

- If the media type is an XML media type or a text media type with a Unicode charset, the data is encoded as a sequence of Unicode characters (and wrapped in an XML element).
- If the media type is not an appropriate text type, or the media type is not recognized, the data is base64-encoded (and wrapped in an XML element).

The following example, taken from the XProc specification, uses `p:data` to read a CSV file:

```
<p:identity name="readcsv">
  <p:input port="source">
    <p:data href="stateabbr.csv"/>
  </p:input>
</p:identity>
```

If the processor is able to detect that the data is text, or if the data is annotated with a text media type information (for instance, when retrieving the resource over HTTP), the result of the `p:identity` step might look as follows:

```
<c:data xmlns:c="http://www.w3.org/2007/03/xproc-step"
        content-type="text/plain">
AL,Alabama
AK,Alaska
AZ,Arizona
...
</c:data>
```

If, however, the processor fails to detect that the data is text (or if the data is annotated with a binary content type), the result will be base64-encoded:

```
<c:data xmlns:c="http://www.w3.org/2007/03/xproc-step"
        content-type="application/octet-stream" encoding="base64">
QUwsQWxhYmFtYQpBSyxBbGFza2EKQVosQXJpem9uYQo
...
</c:data>
```

Notice the `content-type` and `encoding` attributes on the `c:data` wrapper element that can be used by subsequent steps in the pipeline to decode and process the data as needed.

Although admittedly sub-optimal, at least this ability to pass non-XML data as base64-encoded between steps seems like an acceptable compromise for many situations. Unfortunately, while it is relatively straightforward to produce base64-encoded content in XProc, there is only very little one can do with such content. In fact, there are only two options: sending it over HTTP using `p:http-request` or processing it using `p:unescape-markup` – however, the latter is only applicable if the base64-encoded data is XML or HTML. Apart from this, XProc does not offer much more. It is, for instance, not possible to base64-decode and store the data to an external location using the `p:store` step (which supports only XML documents), and every attempt to process base64-encoded data in a meaningful way leads almost inevitably to custom atomic steps or other XProc extensions.

3. Extending XProc to support non-XML media types

In order to allow non-XML data – in its raw form, not wrapped in an XML document nor encoded in any way – to flow through the pipeline, the requirement that what

flows between the steps in an XProc pipeline are exclusively XML documents needs to be relaxed: the steps must be able to consume and produce not only XML documents, but also non-XML data. However, this introduces an interesting challenge as XProc is built from the ground up on XML Infoset [1] and XPath data model [3][4]. XProc steps expect XML Infoset instances on the input ports and produce XML Infoset instances on the output ports. This leads to two options: either the processor needs to be able to provide some kind of a (synthetic) XML Infoset view on top of non-XML data, or the steps need to change so that they can operate not only on XML Infoset instances, but also on non-XML data where this makes sense (clearly the case of `p:identity`, `p:store` and a few others).

The addition of XPath, which XProc uses as the expression language, makes the situation even more interesting as it immediately begs the question of what does querying over non-XML data actually mean? Does it correspond to querying some kind of metadata gleaned from the original data? Or is it the ability to inspect the raw octet stream? The former would make sense for many binary formats: being able to query for the dimensions of an JPEG image would surely be a powerful feature. On the other hand, the value of being able to see the 5th, the 10th, or the 56961st byte of that same JPEG image is questionable in the XProc context. For many text and semi-binary formats, though, the ability to inspect the byte sequence might represent a very useful and practical way of extracting information from the data. So, both query models are valid and ideally both should be supported; it depends on the media type of the data and the actual use case which of the two makes more sense.

The extension scheme that this article proposes addresses the above by adhering to the following principles:

- Both XML and non-XML data can flow through the pipeline. Conceptually, and for compatibility with the current XProc specification, XML data flows as XML Infoset instances, and non-XML data as "raw" octet streams – in the remainder of this article, the union of XML and non-XML data is referred to as simply *data*.
- The data that flows through the pipeline is annotated with media type information.
- XProc steps can declare what media types they expect on their input ports and what media types they produce on their output ports. If data of an incompatible media type arrives on a port of a step, the XProc processor attempts to convert ("shim") the data to the appropriate media type.
- To allow for non-XML support in XPath, the XPath data model has been extended with a new type of node for representing binary data and with accessors for retrieving media type information associated with the nodes.

These principles are discussed in the following text in more detail.

Note
The proposed extensions to XProc change the behavior of the XProc processor and of some of the standard XProc constructs. Although the incompatibilities with the official XProc specification are relatively minor and fairly isolated, a processor that implements these extensions can no longer be considered a conformant XProc processor.

3.1. Media type annotations

For the purpose of distinguishing between data of different media types, the data that flows through the pipeline is annotated with media type information. The media type information can be provided either explicitly (for example by the pipeline user when passing the input data to the pipeline) or implicitly (for example when the XProc processor retrieves a resource over HTTP and processes the `Content-Type` header).

When no media type information is available (because it wasn't provided or because the XProc processor was not able to infer it when retrieving a resource), then – with the exception of the `p:data` binding and the `p:http-request` step, as will be discussed in Section 3.4 – the media type `application/xml` is assumed. This default reflects the most common scenario, which is processing of XML data.

In the situations when the defaulted or inferred media type is not correct, the pipeline author can enforce a specific media type in the pipeline, using the mechanisms described in Section 3.4.5.

3.2. Processing multiple media types via shimming

The steps in the pipeline (including the pipeline itself) can specify what media types they process and what media types they produce. This is done by specifying the media type on the input port and output port declarations (see Section 3.4.1 for exact details).

The media type on a port can be specified in two ways:

- A specific media type, such as `application/xml`, `text/plain`, or `application/octet-stream`.
- A wildcard (the "`*`" character) that matches any media type.

If no media type is declared on an input port or output port, then unless stated otherwise (see Section 3.4.3), the media type `application/xml` is assumed.

While evaluating a pipeline, the XProc processor performs the following algorithm when data appears on a port of a step:

- If the port media type is a wildcard or if the data media type is the same as the port media type, the data appears on the port with no modifications; otherwise

- if the XProc processor knows how to map (see the discussion below) from the data media type to the port media type, the data is converted to the port media type; otherwise
- the XProc processor performs one of the following fall-back actions:
  - If both the data and the port media types are XML media types, the data appears on the port with no modifications.
  - If the port media type is `application/xml`, the data is processed as if it was read via the standard XProc `p:data` binding with a `c:data` wrapper element.
  - If both the data and the port media types are text media types, the data appears on the port with no modifications.
  - Any other combination of the data and the port media types results in a dynamic error.

An important aspect of the above algorithm is that it applies not only to the input ports, but also to the output ports: before the data appears on an output port, it is converted to the appropriate media type. This leads to a number of interesting properties, especially in conjunction with compound steps – it is, for example, possible to create a `p:for-each` loop whose sub-pipeline produces data of all sorts of media types which are then "consolidated" into one media type as specified on the `p:for-each`'s output port.

The media type conversion applies only to the `p:input` and `p:output` elements. It does not take place when the XProc processor processes the `p:with-option`, `p:with-param`, and `p:variable` elements, nor the `p:xpath-context`, `p:iteration-source`, and `p:viewport-source` elements. It also does not apply when the XProc processor evaluates the test expressions of `p:choose`/`p:when` elements. In these cases, the XPath expressions use the original data as the context item.

The kinds of mappings between different media types the XProc processor supports is left implementation-defined. Admittedly, to ensure at least a minimum level of interoperability between different XProc processors, it would be best if there were a well-defined set of media type to media type mappings that the processors were required to support. Identifying such a set of mappings, however, is a research topic on its own, almost certainly requiring a broader discussion within the community, as for many media types there are no agreed-on "one size fits all" mappings that would satisfy all users or use cases. A typical example are the various XML/JSON mappings proposed in the recent past (a rather unfortunate situation as supporting JSON is clearly of prime interest in the XProc context).

3.3. Extensions to the XPath data model

Supporting XPath queries over non-XML data requires a number of extensions to the XPath data model. Note that for historical reasons, XProc allows using both

XPath 1.0 and XPath 2.0 - however, this and the following sections focus solely on XPath 2.0 and the XQuery and XPath data model (XDM) [2].

The XDM data model has been extended in two ways. First, in order to support media type annotations, the data model has been augmented with:

- A new property on the document node:
  `content-type`, possibly empty
- A new accessor defined on all kinds of XDM nodes:
  `dm:content-type($n as node()) as xs:string?`

For the document node, the `dm:content-type` accessor returns the value of the `content-type` property. For the other types of nodes (element, attribute, text, namespace, processing instruction, and comment), it returns the value of the `content-type` property of the owner document.

The second addition to the data model is the introduction of a new type of node – *binary data node* – to represent non-XML data. The binary data node has the following properties:

- `base-uri`, possibly empty
- `content-type`, possibly empty

For the binary data node, the `dm:base-uri` accessor returns the value of the `base-uri` property, and the `dm:content-type` accessor returns the value of the `content-type` property. The `dm:node-kind` accessor returns the value "`binary-data`". All other accessors defined on XDM nodes return the empty sequence for the binary node.

Note that it should be possible to expose the octet sequence of the binary data node by introducing a special property and an accessor (representing the octets for instance as a sequence of `xs:unsignedByte` or `xs:integer`). At the time of writing this article, however, this has not been implemented.

3.4. Extension to the XProc language

This section describes the extensions to the XProc language to support multiple media types. The extensions include XProc extension attributes, custom steps, and XPath extension functions. Some of these extensions modify the functionality of some XProc constructs to adapt them to the new processing environment.

All extension constructs are in the namespace `http://www.emc.com/documentum/xml/xproc-mime`; the conventional namespace prefix "`m:`" is used in the following text.

3.4.1. Media type annotations on `p:input` and `p:output`

Media type annotations can be added to input and output port declarations using the `m:content-type` extension attribute. The value of the `m:content-type` attribute is either an exact media type string (such as `application/xml`) or a wildcard, repres-

ented by the "*" character. If the m:content-type attribute is not specified on a port declaration, the media type application/xml is assumed.

The declaration below declares a step that accepts XML data on the source input port and that produces PDF output on the result output port:

```
<p:declare-step>
  <p:input port="source" m:content-type="application/xml"/>
  <p:output port="result" m:content-type="application/pdf"/>
  ...
</p:declare-step>
```

The following example declares a step that can process and produce data of any media type:

```
<p:declare-step>
  <p:input port="source" m:content-type="*"/>
  <p:output port="result" m:content-type="*"/>
  ...
</p:declare-step>
```

The m:media-type attribute cannot be used on parameter input ports; the media type of parameter input ports is always application/xml.

3.4.2. Modifications to p:data

For the purpose of better supporting non-XML media types, the p:data binding has been modified to support returning raw, not wrapped, data. This is possibly the biggest breaking change to the language, but one that provides great usability benefits in the multiple media types context: it makes it possible to process the results of p:data right away, without having to unwrap and decode first.

Where previously the p:data binding always encoded and wrapped the resource referred to via the href attribute (the wrapper being either a custom-specified element or the default c:data element), the modified p:data only encodes and wraps the resource when the pipeline author requests an explicit wrapper using the wrapper attribute. If no wrapper element is specified, p:data returns the resource "as is".

The semantics of the content-type attribute of p:data remains the same: if the resource comes with a media type annotation, that one must be used, otherwise the media type specified in the content-type attribute should be assumed. If no media type information can be associated with the resource, the media type application/octet-stream is assumed.

3.4.3. Modifications to built-in XProc steps

Adding media type annotations to step declarations and modifying the p:data binding is not enough: in order for non-XML data to truly flow through XProc

pipelines – including complex non-linear pipelines with loops, conditional logic, and recursion – a number of modifications to the standard built-in compound steps are necessary. The following text summarizes these modificatons.

- The `p:pipeline` shortcut supports input and output of any media type by default. It is equivalent to the following `p:declare-step`:

```
<p:declare-step>
  <p:input port="source" primary="true" sequence="false"
           m:content-type="*"/>
  <p:input port="parameters" primary="true" kind="parameter"/>
  <p:input port="result" primary="true" sequence="false"
           m:content-type="*"/>
</p:declare-step>
```

  This makes it possible to use `p:pipeline` to process both XML and non-XML data easily.
- The `p:for-each` step can be used to process data of any media type. The `current` implicit input port supports data of any media type. If the `p:for-each` step contains explicit `p:output` declarations, then, inside of the `p:for-each`, these output ports accept any media type regardless of the value of the `m:content-type` attribute. On the outside of `p:for-each`, however, the data appearing on the output port gets converted to the appropriate media type. By default, the implicit output port of `p:for-each` supports any media type.
- The `p:choose` step can process data of any media type. The `p:when` branches must declare the same numbers of output ports with the same names – however, these output ports may specify different media types. By default, the implicit output ports of `p:when` (and `p:choose`) support any media type.
- The `p:group` wrapper can be used to process data of any media type. By default, the implicit output port of `p:group` supports any media type.
- The `p:try` step can be used to process data of any media type. The `error` input port of `p:catch` (the XML representation of the dynamic error) accepts data of the media type `application/xml`. The output ports of the `p:group` and `p:catch` sub-pipelines of `p:try` must specify the same numbers of output ports with the same names, but they may declare different media types. By default, the implicit output port of `p:catch` supports any media type.

3.4.4. Modifications to the XProc standard step library

Most atomic steps from the standard XProc step library are too XML-specific (for instance `p:xinclude` or `p:validate-with-xml-schema`) to be easily applicable to non-XML data. Having said that, however, there is a small number of steps that can be – in some cases to great benefit – adapted for non-XML data processing. This requires both modifying the implementations of the steps and changing their declarations

in the standard step library by adding more relaxed media type annotations to their input and output ports. The list below summarizes the changes in more detail:

- `p:count` – can be used to process data of any media type. The output format of `p:count` (a `c:result` document with the count) remains unchanged from the specification.
- `p:http-request` – can produce output of any media type. Very much similar to the `p:data` binding, the `p:http-request` step now presents non-XML responses in their raw form, not base64-encoding nor wrapping them anymore. If the media type of the response data cannot be determined, the media type `application/octet-stream` is assumed.

  The "detailed" response mode of `p:http-request` remains unchanged from the specification, and so does handling of multipart responses.
- `p:identity` – can be used to process data of any media type.
- `p:sink` – can be used to process data of any media type.
- `p:split-sequence` – can be used to process data of any media type.
- `p:store` – can be used to store data of any media type. The XML serialization options are applied only for data that has an XML media types.
- `p:exec` – if the data that appears on the `source` input port is not XML, it is passed in its raw form to the command as its standard input.
- `p:xquery` – if the media type of the data that appears on the `query` input port is `application/xquery`, the data is passed to the query engine "as is".

3.4.5. Overriding media type information

On some occasions it may be necessary to be able to override the media type of the data: for example when the XProc processor fails to detect the media type (or detects it incorrectly), or when the pipeline author deliberately wants to use a different media type (for instance, to treat SVG data annotated as `image/svg+xml` as simply `application/xml`).

The override media type can be specified statically – on the XProc binding elements – or dynamically – using an extension step.

On the binding level, the override media type is specified using the `m:as-content-type` extension attribute. The fragment below shows an example of how to ensure that the XQuery data read from an external file is annotated as `application/xquery`:

```
...
<p:xquery>
  <p:input port="query">
    <p:data href="searchquery.xq"
            m:as-content-type="application/xquery"/>
```

```
    </p:input>
  </p:xquery>
  ...
```

Specifying the override media type on the binding level has the disadvantage that it is static; the override media type cannot be constructed dynamically. A dynamic override media type can be specified using the `m:as-content-type` extension step:

```
<p:declare-step type="m:as-content-type">
  <p:input port="source" sequence="true" m:content-type="*"/>
  <p:output port="result" sequence="true" m:content-type="*"/>
  <p:option name="content-type" required="true"/>
</p:declare-step>
```

The `m:as-content-type` step behaves as the standard `p:identity` step, except that it annotates the output data with the media type provided via the required `content-type` option.

Note that applying an override media type does not result in data conversion from the original media type to the override type; the override media type merely replaces the original data media type annotation. If the override media type is incompatible with the data media type (for example, an `application/xml` override for `application/pdf`), it is reasonable to expect that subsequent processing may fail.

3.4.6. XPath extension functions

To be able to query the media type of the data flowing through the pipeline, a new function has been added to the library of the XProc extension XPath functions: `m:content-type`.

The `m:content-type` function is declared as follows:

```
m:content-type() as xs:string?
m:content-type($arg as node()?) as xs:string?
```

The function returns the value of the `content-type` property for `$arg` as defined by the accessor function `dm:content-type()` for that kind of node (see Section 3.3). If `$arg` is not specified, the behavior is identical to calling the function with the context item (`.`) as argument.

4. Examples

This section presents a number of examples that illustrate how the proposed extensions can be used in real-life pipelines.

4.1. Media type-aware processing

The first example shows a simple pipeline that processes the input data (a sequence of XML documents and non-XML data) based on the media type information. The pipeline performs XInclude on XML documents while leaving the other data unmodified.

```
<p:declare-step version="1.0"
                xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:m="http://www.emc.com/documentum/xml/xproc-mime">
  <p:input port="source" sequence="true" m:content-type="*"/>
  <p:output port="result" sequence="true" m:content-type="*"/>

  <p:for-each>
    <p:choose>
      <p:when test="m:content-type()='application/xml'">
        <p:xinclude/>
      </p:when>
      <p:otherwise>
        <p:identity/>
      </p:otherwise>
    </p:choose>
  </p:for-each>

</p:declare-step>
```

4.2. Using compound steps for media type consolidation

An interesting application of adding the `m:content-type` attribute to output port declarations is to consolidate the media type of the results of a compound step. The example pipeline below takes an XHTML document and retrieves all images referred to using the `img` elements. Because the declaration of the `result` output port of the pipeline contains the `m:content-type` attribute with the value `image/jpeg`, all images will be converted to JPEG before they appear on the `result` output port of the pipeline. If the processor encounters an image with a media type that it does not know how to convert to `image/jpeg`, the pipeline will fail with a dynamic error.

```
<p:declare-step version="1.0"
                xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:c="http://www.w3.org/ns/xproc-step"
                xmlns:m="http://www.emc.com/documentum/xml/xproc-mime"
                xmlns:h="http://www.w3.org/1999/xhtml">
  <p:input port="source"/>
  <p:output port="result" sequence="true" m:content-type="image/jpeg"/>

  <p:for-each>
```

```
    <p:iteration-source select="//h:img"/>

    <p:add-attribute match="c:request" attribute-name="href">
      <p:input port="source">
        <p:inline>
          <c:request method="GET"/>
        </p:inline>
      </p:input>
      <p:with-option name="attribute-value" select="/h:img/@href"/>
    </p:add-attribute>

    <p:http-request/>
  </p:for-each>

</p:declare-step>
```

4.3. Processing JSON data

This example shows how to transparently process JSON data. Note that this example assumes that the XProc processor is able to map data from `application/json` to `application/xml`. For this particular example, a simple JSON-to-XML mapper has been implemented in Calumet using the open source JSON-lib[3] Java library. The table below shows the XML representation that the mapper produces for various input JSON strings:

JSON	**XML**

`{"prop": "value"}`	`<o>`
	`  <prop type="string">value</prop>`
	`</o>`

`{"prop1": [{"prop2": "value"}]}`	`<o>`
	`  <prop1 class="array">`
	`  <e class="object">`
	`  <prop2 type="string">value</prop2>`
	`  </e>`
	`  </prop1>`
	`</o>`

[3] http://json-lib.sourceforge.net/

JSON	XML

[
  {"prop1": "value",
  "prop2": 100,
  "prop3": false,
  "prop4": null}
]
	<a>
	  <e class="object">
	  <prop1 type="string">value</prop1>
	  <prop2 type="number">100</prop2>
	  <prop3 type="boolean">false</prop3>
	  <prop4 class="object" null="true"/>
	  </e>
	

The pipeline below retrieves the Twitter public timeline using the `p:http-request` step and then applies the `p:xquery` step to the JSON response data to extract user information. The `p:http-request` produces raw JSON data on its `result` output port. The JSON data is then passed to the `source` input port of the `p:xquery` step, but because the media type of the `source` port is `application/xml`, the processor first converts the data to XML. After that, the `p:xquery` step evaluates the XQuery and produces the results.

```
<p:declare-step version="1.0"
                xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:c="http://www.w3.org/ns/xproc-step"
                xmlns:m="http://www.emc.com/documentum/xml/xproc-mime">
  <p:output port="result"/>

  <p:http-request>
    <p:input port="source">
      <p:inline>
        <c:request method="GET" override-content-type="application/json"
            href="https://api.twitter.com/1/statuses/public_timeline.json"/>
      </p:inline>
    </p:input>
  </p:http-request>

  <p:xquery>
    <p:input port="query">
      <p:inline>
        <c:query><![CDATA[
          <users> {
            for $user in //e/user
            return
              <user>
                <id>{string($user/id)}</id>
                <screen_name>{string($user/name)}</screen_name>
                <name>{string($user/name)}</name>
              </user>
```

```
            } </users>
          ]]></c:query>
        </p:inline>
      </p:input>
      <p:input port="parameters"><p:empty/></p:input>
    </p:xquery>

  </p:declare-step>
```

4.4. Manipulating ZIP archives

The last example shows a slightly more complex pipeline that both consumes and produces binary data. The pipeline takes an ODF document (a ZIP archive) on the `source` input port and an image on the `image` input port, and produces a new ODF document from the original one by inserting the image into it. The pipeline also makes it possible to specify the image dimensions via the options `width` and `height`.

The pipeline assumes the existence of two extension steps – `ex:unzip` and `ex:zip` – that it uses for extracting information out of a ZIP archive and for creating new archives. The steps are analogous to the EXProc[4] `pxp:unzip` and `pxp:zip` extension steps, with the notable difference that they operate on ZIP streams directly as opposed to referring to external ZIP files via URI options.

`ex:unzip` The `ex:unzip` step takes a ZIP archive on the `source` input port and extracts its contents. The signature of the `ex:unzip` step looks as follows:

```
<p:declare-step type="ex:unzip"
                xmlns:ex="http://www.example.org/ns/xproc"
                xmlns:m="http://www.emc.com/documentum/xml/▶
xproc-mime">
  <p:input port="source" m:content-type="application/zip"/>
  <p:output port="result" sequence="true" m:content-type="*"/>
  <p:option name="file"/>
  <p:option name="content-type"/>
</p:declare-step>
```

By default, the step produces a sequence containing all files in the archive, but it is also possible to extract a specific file by specifying its path-name using the `file` option. The optional `content-type` option specifies the media type for the extracted files. In the absence of the `content-type` option, the extracted files have the media type `application/xml`. The base URIs of the extracted files will be the same as their archive path-names.

[4] http://exproc.org/proposed/steps/

`ex:zip` The `ex:zip` step reads the data that appears on the `source` input port and produces a new ZIP archive on the `result` output port.

```
<p:declare-step type="ex:zip"
                xmlns:ex="http://www.example.org/ns/xproc"
                xmlns:m="http://www.emc.com/documentum/xml/▶
xproc-mime">
  <p:input port="source" sequence="true" primary="true"
           m:content-type="*"/>
  <p:input port="manifest" m:content-type="application/xml"/>
  <p:output port="result" m:content-type="application/zip"/>
  <p:option name="compression-method" select="'deflated'"/>
  <p:option name="compression-level" select="'default'"/>
</p:declare-step>
```

By default, the path-names of the ZIP entries will be the same as the base URIs of the input data. This can be customized by providing a manifest document on the `manifest` input port. The manifest specifies the mappings from base URIs to ZIP path-names, and optionally also additional properties such as entry-specific compression settings or comment strings. The schema for the manifest document is the same as for the EXProc `pxp:zip` step; what the manifest might look like is best illustrated by an example:

```
<c:zip-manifest xmlns:c="http://www.w3.org/ns/xproc-step">
  <c:entry href="http://www.example.org/file.xml"
           name="file.xml" comment="An example file"/>
  <c:entry href="http://www.example.org/image.jpg"
           name="images/image.jpg" method="stored"/>
</c:zip-manifest>
```

The `ex:zip` step also supports options for specifying archive-level compression settings (`compression-method` and `compression-level`).

The example "insert image into ODF" pipeline consists of four main blocks:

1. Create the `ex:zip` manifest for inserting the image.
2. Unzip the input ODF document.
3. Create new versions of the extracted files `META-INF/manifest.xml` and `content.xml`.
4. Create a new ZIP archive that includes the image data as well as the new versions of the files `META-INF/manifest.xml` and `content.xml`.

The pipeline also shows an example of using the `m:as-content-type` attribute on the binding level. Because `ex:unzip` annotates the unzipped content as `application/octet-stream` by default, the pipeline uses `m:as-content-type` to make

sure that the files `META-INF/manifest.xml` and `content.xml` are treated as XML documents.

```
<p:pipeline name="main" version="1.0"
    xmlns:p="http://www.w3.org/ns/xproc"
    xmlns:c="http://www.w3.org/ns/xproc-step"
    xmlns:m="http://www.emc.com/documentum/xml/xproc-mime"
    xmlns:ex="http://www.example.org/ns/xproc"
    xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
    xmlns:manifest="urn:oasis:names:tc:opendocument:xmlns:manifest:1.0"
    xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
    xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
    xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <p:input port="image" m:content-type="*"/>
  <p:option name="width" select="'100px'"/>
  <p:option name="height" select="'100px'"/>

  <p:variable name="image-content-type" select="m:content-type()">
    <p:pipe step="main" port="image"/>
  </p:variable>
  <p:variable name="image-file-name"
              select="tokenize(base-uri(), '/')[last()]">
    <p:pipe step="main" port="image"/>
  </p:variable>

  <p:template name="create-zip-manifest">
    <p:input port="source">
      <p:pipe step="main" port="image"/>
    </p:input>
    <p:input port="template">
      <p:inline>
        <c:zip-manifest>
          <c:entry href="{base-uri()}" name="Pictures/{$file-name}"/>
        </c:zip-manifest>
      </p:inline>
    </p:input>
    <p:with-param name="file-name" select="$image-file-name"/>
  </p:template>

  <ex:unzip>
    <p:input port="source">
      <p:pipe step="main" port="source"/>
    </p:input>
  </ex:unzip>

  <p:for-each name="for">
```

```
<p:output port="result" m:content-type="*"/>
<p:variable name="path" select="base-uri()"/>

<p:choose>
  <p:when test="$path='META-INF/manifest.xml'">
    <p:template name="create-file-entry">
      <p:input port="template">
        <p:inline>
          <manifest:file-entry manifest:media-type="{$content-type}"
              manifest:full-path="Pictures/{$file-name}"/>
        </p:inline>
      </p:input>
      <p:with-param name="content-type" select="$image-content-type"/>
      <p:with-param name="file-name" select="$image-file-name"/>
    </p:template>

    <p:insert position="last-child" match="manifest:manifest">
      <p:input port="source">
        <p:pipe step="for" port="current"
                m:as-content-type="application/xml"/>
      </p:input>
      <p:input port="insertion">
        <p:pipe step="create-file-entry" port="result"/>
      </p:input>
    </p:insert>
  </p:when>

  <p:when test="$path='content.xml'">
    <p:template name="create-image">
      <p:input port="template">
        <p:inline>
          <text:p>
            <draw:frame text:anchor-type="paragraph"
                        svg:width="{$width}" svg:height="{$height}">
              <draw:image xlink:href="Pictures/{$file-name}"
                          xlink:type="simple" xlink:show="embed"
                          xlink:actuate="onLoad"/>
            </draw:frame>
          </text:p>
        </p:inline>
      </p:input>
      <p:with-param name="file-name" select="$image-file-name"/>
      <p:with-param name="width" select="$width"/>
      <p:with-param name="height" select="$height"/>
    </p:template>
```

```
          <p:insert position="last-child" match="office:body">
            <p:input port="source">
              <p:pipe step="for" port="current"
                      m:as-content-type="application/xml"/>
            </p:input>
            <p:input port="insertion">
              <p:pipe step="create-image" port="result"/>
            </p:input>
          </p:insert>
        </p:when>

        <p:otherwise>
          <p:identity/>
        </p:otherwise>
      </p:choose>
    </p:for-each>

    <ex:zip name="zip">
      <p:input port="source">
        <p:pipe step="for" port="result"/>
        <p:pipe step="main" port="image"/>
      </p:input>
      <p:input port="manifest">
        <p:pipe step="create-zip-manifest" port="result"/>
      </p:input>
    </ex:zip>

  </p:pipeline>
```

5. Conclusion

This article proposes a number of extensions to XProc to provide better support for non-XML media types. The approach taken is a pragmatic one: it involves extensions to the XProc language and to the processing model in order to allow non-XML data to flow through the pipeline and be processed by the XProc steps, but it also relies on the capabilities of the XProc processor that determines the kinds of conversions between different media types are supported (and what they look like). The viability of such an approach is to be seen. Abstracting from specific media type-to-media type conversion schemes might be viewed as too open and non-interoperable by some, while others might think that it provides the right level of flexibility in a world where no universally agreed-on or applicable set of conversions exist. Both viewpoints are rational, and the most practical solution will most probably lie somewhere in-between.

Ideally, the ideas presented in this article will serve as a starting point for further discussions in the community and within the XML Processing Mode Working Group. Practical experiments with an existing XProc implementation show that the proposed extensions are implementable and can be used in practice with interesting results.

Bibliography

[1] Cowan, John – Tobin, Richard: XML Information Set (Second Edition). W3C Recommendation, 4 February 2004. http://www.w3.org/TR/xml-infoset/

[2] Berglund, Anders – Fernández, Mary – Malhotra, Ashok – Marsh, Jonathan – Nagy, Marton – Walsh, Norman: XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition). W3C Recommendation, 14 December 2010. http://www.w3.org/TR/xpath-datamodel/

[3] Clark, James – DeRose, Steve: XML Path Language (XPath) Version 1.0. W3C Recommendation, 16 November 1999. http://www.w3.org/TR/xpath/

[4] Berglund, Anders – Boag, Scott – Chamberlin, Don – Fernández, Mary F. – Kay, Michael – Robie, Jonathan, Siméon, Jérôme: XML Path Language (XPath) 2.0 (Second Edition). W3C Recommendation, 14 December 2010. http://www.w3.org/TR/xpath20/

[5] Walsh, Norman – Milowski, Alex – Thompson, Henry S.: XProc: An XML Pipeline Language. W3C Recommendation, 11 May 2010. http://www.w3.org/TR/xproc/

Understanding NVDL The Anatomy of an Open Source XProc/XSLT implementation of NVDL

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

Abstract

NVDL stands for Namespace-based Validation and Dispatching Language. It is an ISO standard, like Relax NG and Schematron. NVDL allows to validate documents containing markup from different vocabularies without the need to change the schema of each vocabulary to know about the others, to use different schema languages to validate different parts of the document and to perform multiple validations.

In this presentation I will show an XSLT implementation of the NVDL dispatching plus an XProc orchestration of dipatching and validation tasks. This will allow to quickly understand how NVDL works, especially for people with an XSLT background.

I developed the XSLT implementation of NVDL dispatching initially as part of oNVDL (the open source implementation of NVDL now contributed to Jing) to help me understand how NVDL works. Now, having XProc available it is easy to put all the processing together, add also the validation steps and create a complete NVDL implementation based on XProc and XSLT.

Keywords: NVDL, XML, validation, XSLT, XProc, XML Schema, Relax NG, Schematron, Current advances in XML

1. Introduction

The NVDL processing has two logical parts, first we have a dispatching part where the document is processed to extract document fragments called validation candidates and then these document fragments are validated with the specified schema. To understand NVDL you need to understand how these validation candidates are obtained, especially because they can overlap.

2. NVDL Dispatching

2.1. Splitting the Document into Sections

The NVDL dispatching is the most important thing to understand. First, NVDL acts on sections. Sections are adjiacent fragments from the same namespace. We have element sections and attribute sections. Element sections can be further split if the NVDL script defines a trigger that specifies the elements that can split sections, the new sections start with these elements. So, the first processing step is to split the document into element and attribute sections. This takes into account the document content and the triggers defined in the NVDL script.

In our implementation we have this processing done by the getSections.xsl stylesheet. This takes the NVDL script as a parameter and determines the defined triggers from that. The result is a document with the sections marked with special elements and attributes from the `http://www.oxygenxml.com/nvdl` namespace.

For example, let's consider the following document that contains XHTML plus XForms

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-model href="xhtml-xforms.nvdl" type="application/xml"
  schematypens="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xforms="http://www.w3.org/2002/xforms">
  <head>
    <title>Sample</title>
    <xforms:model id="myForm">
      <xforms:submission id="submit" method="post"
        action="http://www.example.com/xforms/request"/>
      <xforms:instance id="my" xmlns="">
        <myData>
          <input>Initial input</input>
        </myData>
      </xforms:instance>
    </xforms:model>
  </head>
  <body>
    <h1>XForms sample</h1>
    <p>Input</p>
    <p>
      <xforms:input ref="/myData/input">
        <xforms:label>Input Form Control</xforms:label>
      </xforms:input>
    </p>
    <p>Submit:</p>
    <p>
```

```
      <xforms:submit submission="my">
        <xforms:label>Submit Me</xforms:label>
      </xforms:submit>
    </p>
  </body>
</html>
```

The sections are marked with elements and attributes from the `http://www.oxygenxml.com/nvdl` namespace that uses the prefix `n`. The NVDL script specifies head and body as trigger elements, that is they split a XHTML section when they appear, resulting new sections that start with these elements. The result of getting the section infromation for the above sample file is

```
<n:section xmlns:n="http://www.oxygenxml.com/nvdl"
  ns="http://www.w3.org/1999/xhtml">
  <html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:xforms="http://www.w3.org/2002/xforms">
    <n:section xmlns="" ns="http://www.w3.org/1999/xhtml">
      <head xmlns="http://www.w3.org/1999/xhtml">
        <title>Sample</title>
        <n:section xmlns="" ns="http://www.w3.org/2002/xforms">
          <xforms:model xmlns="http://www.w3.org/1999/xhtml"
            id="myForm" n:attSection1="id" n:attSection1ns="">
            <xforms:submission id="submit" method="post"
              action="http://www.example.com/xforms/request"
              n:attSection1="id method action" n:attSection1ns=""/>
            <xforms:instance xmlns="" id="my"
              n:attSection1="id" n:attSection1ns="">
              <n:section ns="">
                <myData>
                  <input>Initial input</input>
                </myData>
              </n:section>
            </xforms:instance>
          </xforms:model>
        </n:section>
      </head>
    </n:section>
    <n:section xmlns="" ns="http://www.w3.org/1999/xhtml">
      <body xmlns="http://www.w3.org/1999/xhtml">
        <h1>XForms sample</h1>
        <p>Input</p>
        <p>
          <n:section xmlns="" ns="http://www.w3.org/2002/xforms">
            <xforms:input xmlns="http://www.w3.org/1999/xhtml"
              ref="/myData/input" n:attSection1="ref" n:attSection1ns="">
              <xforms:label>Input Form Control</xforms:label>
```

```
            </xforms:input>
          </n:section>
        </p>
        <p>Submit:</p>
        <p>
          <n:section xmlns="" ns="http://www.w3.org/2002/xforms">
            <xforms:submit xmlns="http://www.w3.org/1999/xhtml"
              submission="my" n:attSection1="submission" n:attSection1ns="">
              <xforms:label>Submit Me</xforms:label>
            </xforms:submit>
          </n:section>
        </p>
      </body>
    </n:section>
  </html>
</n:section>
```

Note
Notice the sections that start with head and body elements from the XHTML namespace, they are generated because these elements are defined as trigger elements in the NVDL script, otherwise these elements will be included in the XHTML section that starts with the html element.

The `n:section` element marks the start of a section and the ns attribute specifies the namespace of the elements from that section. It can contain elements from the section namespace and other sections. The other section can appear anywhere, when the namepsace of an element is different or when a trigger element is encountered.

The attrinute sections are also marked. We use attributes in this case, again in the `http://www.oxygenxml.com/nvdl` namespace with the following form `n:attSectionX` and `n:attSectionXns` (where *X* stands for a number, from 1 to the number of attribute sections), as can be seen in the previous example. The `n:attSectionXns` attribute identifies the namespace of the attribute section while the `n:attSectionX` attribute contains the names of all the attributes forming this attribute section separated by space.

2.2. Converting the NVDL script to XSLT

The most important thing in order to understand NVDL is that its processing is similar with the XSLT processing, but instead of working on elements and attributes it works on element and attribute sections. Thus an NVDL script can be converted to an XSLT stylesheet that can process the document with sections and produces the dispatching output. Here it is the NVDL script for processing XHTML and XForms:

```
<?xml version="1.0" encoding="UTF-8"?>
<rules xmlns="http://purl.oclc.org/dsdl/nvdl/ns/structure/1.0"
  xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
  startMode="html">
  <trigger ns="http://www.w3.org/1999/xhtml" nameList="head body"/>
  <!-- Validations of XHTML and XForms -->
  <mode name="html">
    <namespace ns="http://www.w3.org/1999/xhtml">
      <validate schema="http://www.w3.org/1999/xhtml/xhtml.rng"
        useMode="allXHTML"/>
      <validate schema="XFormsHTMLWrapper.xsd" useMode="allXForms"/>
    </namespace>
  </mode>
  <!-- Attaches all XHTML sections, ignores everything else -->
  <mode name="allXHTML">
    <namespace ns="http://www.w3.org/1999/xhtml">
      <attach/>
    </namespace>
    <anyNamespace>
      <unwrap/>
    </anyNamespace>
  </mode>
  <!-- Attaches all XForm sections, ignores everything else -->
  <mode name="allXForms">
    <namespace ns="http://www.w3.org/2002/xforms">
      <attach/>
    </namespace>
    <anyNamespace>
      <unwrap/>
    </anyNamespace>
  </mode>
</rules>
```

This generates as the dispatching result two validation candidates, one contains all the XHTML content and the other puts all the XForms inside the `html` element. The corresponding XSLT stylesheet for this NVDL script is

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:n="http://www.oxygenxml.com/nvdl" version="2.0"
  exclude-result-prefixes="n">

  <xsl:template match="/">
    <xsl:variable name="content">
      <n:dispatch>
        <xsl:apply-templates mode="mode_html"/>
      </n:dispatch>
    </xsl:variable>
```

```
  <xsl:apply-templates mode="flatten" select="$content"/>
</xsl:template>

<!--Templates for mode html-->
<xsl:template mode="mode_html"
  match="n:section[@ns='http://www.w3.org/1999/xhtml']">
  <n:validate schema="http://www.w3.org/1999/xhtml/xhtml.rng"
    useMode="allXHTML">
    <xsl:apply-templates mode="mode_allXHTML"/>
  </n:validate>
  <n:validate schema="XFormsHTMLWrapper.xsd" useMode="allXForms">
    <xsl:apply-templates mode="mode_allXForms"/>
  </n:validate>
</xsl:template>
<xsl:template match="n:section" mode="mode_html">
  <n:reject>
    <xsl:apply-templates mode="mode_html"/>
  </n:reject>
</xsl:template>

<!--Templates for mode allXHTML-->
<xsl:template mode="mode_allXHTML"
  match="n:section[@ns='http://www.w3.org/1999/xhtml']">
  <xsl:apply-templates mode="mode_allXHTML"/>
</xsl:template>
<xsl:template match="n:section" mode="mode_allXHTML">
  <xsl:variable name="thisSection" select="generate-id(.)"/>
  <xsl:apply-templates
    select="//n:section[generate-id(ancestor::n:section[1])=$thisSection]"
    mode="mode_allXHTML"/>
</xsl:template>

<!--Templates for mode allXForms-->
<xsl:template mode="mode_allXForms"
  match="n:section[@ns='http://www.w3.org/2002/xforms']">
  <xsl:apply-templates mode="mode_allXForms"/>
</xsl:template>
<xsl:template match="n:section" mode="mode_allXForms">
  <xsl:variable name="thisSection" select="generate-id(.)"/>
  <xsl:apply-templates
    select="//n:section[generate-id(ancestor::n:section[1])=$thisSection]"
    mode="mode_allXForms"/>
</xsl:template>

<xsl:template match="*[namespace-uri()!='http://www.oxygenxml.com/nvdl']"
  mode="#all" priority="-100">
```

```
    <xsl:copy>
      <xsl:apply-templates
        select="@*[namespace-uri()!='http://www.oxygenxml.com/nvdl']"
        mode="#current"/>
      <xsl:apply-templates select="@n:*" mode="#current"/>
      <xsl:apply-templates mode="#current"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="@*[namespace-uri()!='http://www.oxygenxml.com/nvdl']"
    mode="#all" priority="-100">
    <xsl:copy/>
  </xsl:template>
  <xsl:template match="@n:*" mode="#all" priority="-100"/>

  <xsl:template match="n:dispatch" mode="flatten">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates select=".//n:*" mode="flatten1"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="n:*" mode="flatten1">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:apply-templates
        select="text()|*[namespace-uri()!='http://www.oxygenxml.com/nvdl']"
        mode="flatten2"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="*[namespace-uri()!='http://www.oxygenxml.com/nvdl']"
    mode="flatten2">
    <xsl:copy>
      <xsl:copy-of
        select="@*[namespace-uri()!='http://www.oxygenxml.com/nvdl']"/>
      <xsl:apply-templates
        select="text()|*[namespace-uri()!='http://www.oxygenxml.com/nvdl']"
        mode="flatten2"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

As you can see, the `namespace` and `anyNamespace` rules are transformed to XSLT templates matching the corresponding sections and the actions contain along with their execution also an apply-templates that triggers a top-down procesing of the document. The NVDL modes are exactly the XSLT modes these templates are defined in and selecting a different `useMode` for an action results in applying templates in

the XSLT mode corresponding to the XSLT mode. Here it is how they correspond for the html mode:

```
    <mode name="html">
      <namespace ns="http://www.w3.org/1999/xhtml">
```

-->

```
    <xsl:template mode="mode_html"
      match="n:section[@ns='http://www.w3.org/1999/xhtml']">
```

then we have

```
        <validate schema="http://www.w3.org/1999/xhtml/xhtml.rng"
          useMode="allXHTML"/>
```

-->

```
      <n:validate schema="http://www.w3.org/1999/xhtml/xhtml.rng"
        useMode="allXHTML">
        <xsl:apply-templates mode="mode_allXHTML"/>
      </n:validate>
```

and

```
        <validate schema="XFormsHTMLWrapper.xsd" useMode="allXForms"/>
```

-->

```
      <n:validate schema="XFormsHTMLWrapper.xsd" useMode="allXForms">
        <xsl:apply-templates mode="mode_allXForms"/>
      </n:validate>
```

and finally

```
      </namespace>
    </mode>
```

-->

```
    </xsl:template>
```

The additional part in the XSLT code

```
    <xsl:template match="n:section" mode="mode_html">
      <n:reject>
        <xsl:apply-templates mode="mode_html"/>
      </n:reject>
    </xsl:template>
```

is the result of something similar with the XSLT buil-in rules, the NVDL built-in rule that says that if an anyNamespace rule is not specified then an any namespace reject action is implied:

```
  <anyNamespace><reject/></anyNamespace>
```

The XSLT for the generated script is obtained by applying an XSLT stylesheet on the NVDL script. This is similar with the Schematron skeleton implementation. This is a little more complex, reaching about 300 lines of XSLT code. Its source can be found in the oNVDL project.

2.3. Getting the dispatch output

The result of applying this styleheet on the XML document with marked up sections is the dispatching result, which for our sample looks like this:

```
<n:dispatch xmlns:n="http://www.oxygenxml.com/nvdl">
  <n:validate schema="http://www.w3.org/1999/xhtml/xhtml.rng"
    useMode="allXHTML">
    <html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xforms="http://www.w3.org/2002/xforms">
      <head>
        <title>Sample</title>

      </head>
      <body>
        <h1>XForms sample</h1>
        <p>Input</p>
        <p>

        </p>
        <p>Submit:</p>
        <p>

        </p>
      </body>
    </html>
  </n:validate>
  <n:validate schema="XFormsHTMLWrapper.xsd" useMode="allXForms">
    <html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xforms="http://www.w3.org/2002/xforms">
      <xforms:model id="myForm">
        <xforms:submission id="submit" method="post"
          action="http://www.example.com/xforms/request"/>
        <xforms:instance xmlns="" id="my">

        </xforms:instance>
      </xforms:model>
      <xforms:input ref="/myData/input">
        <xforms:label>Input Form Control</xforms:label>
      </xforms:input><xforms:submit submission="my">
        <xforms:label>Submit Me</xforms:label>
```

```
      </xforms:submit>
    </html>
  </n:validate>
</n:dispatch>
```

3. Orchestrating Validation with XProc

Now, all these transformations can be orchestrated with XProc. Even more, we can run also the validate actions as specified in the dispatch output also with XProc and thus getting the full NVDL implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc" name="main"
  xmlns:c="http://www.w3.org/ns/xproc-step"
  xmlns:n="http://www.oxygenxml.com/nvdl" version="1.0">

  <p:input port="source" primary="true"/>
  <p:input port="nvdl"/>

  <p:output port="result" primary="true" sequence="true"/>
  <p:output port="sections">
    <p:pipe port="result" step="split"/>
  </p:output>
  <p:output port="compiled">
    <p:pipe port="result" step="compile"/>
  </p:output>
  <p:output port="dispatch">
    <p:pipe port="result" step="dispatch"/>
  </p:output>

  <!-- Extract the NVDL script filename -->
  <p:xslt name="extractNVDLFilename">
    <p:input port="parameters"><p:empty/></p:input>
    <p:input port="source">
      <p:pipe port="nvdl" step="main"/>
    </p:input>
    <p:input port="stylesheet">
      <p:inline>
        <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          version="2.0">
          <xsl:template match="/">
            <result><xsl:value-of select="document-uri(.)"/></result>
          </xsl:template>
        </xsl:stylesheet>
      </p:inline>
    </p:input>
```

```
</p:xslt>

<p:xslt name="split">
  <p:input port="source">
    <p:pipe port="source" step="main"/>
  </p:input>
  <p:input port="stylesheet">
    <p:document href="getSections.xsl"/>
  </p:input>
  <p:with-param name="nvdl" select="/result/text()">
    <p:pipe port="result" step="extractNVDLFilename"/>
  </p:with-param>
</p:xslt>

<p:xslt name="compile">
  <p:input port="source">
    <p:pipe port="nvdl" step="main"/>
  </p:input>
  <p:input port="stylesheet">
    <p:document href="nvdl2xslt.xsl"/>
  </p:input>
  <p:input port="parameters"><p:empty/></p:input>
</p:xslt>

<p:xslt name="dispatch">
  <p:input port="source">
    <p:pipe port="result" step="split"/>
  </p:input>
  <p:input port="stylesheet">
    <p:pipe port="result" step="compile"/>
  </p:input>
  <p:input port="parameters"><p:empty/></p:input>
</p:xslt>

<p:filter name="candidates" select="//*[self::n:validate or self::n:reject]"/>

<p:for-each>
  <p:identity name="candidate"/>
  <p:choose>
    <p:when test="ends-with(/*/name(), 'reject')">
      <p:error code="Reject">
        <p:input port="source">
          <p:inline>
            <message>Content rejected by a "reject" action.</message>
          </p:inline>
        </p:input>
```

```
  </p:error>
</p:when>
<p:when test="ends-with(/*/@schema, '.rng')">
  <p:load name="loadedSchema">
    <p:with-option name="href"
      select="resolve-uri( /*/@schema, document-uri(/))"/>
  </p:load>
  <p:validate-with-relax-ng>
    <p:input port="source" select="/*/*[1]">
      <p:pipe port="result" step="candidate"/>
    </p:input>
    <p:input port="schema">
      <p:pipe port="result" step="loadedSchema"/>
    </p:input>
  </p:validate-with-relax-ng>
</p:when>
<p:when test="ends-with(/*/@schema, '.xsd')">
  <p:load name="loadedSchema">
    <p:with-option name="href"
      select="resolve-uri( /*/@schema, document-uri(/))"/>
  </p:load>
  <p:validate-with-xml-schema>
    <p:input port="source" select="/*/*[1]">
      <p:pipe port="result" step="candidate"></p:pipe>
    </p:input>
    <p:input port="schema">
      <p:pipe port="result" step="loadedSchema"/>
    </p:input>
  </p:validate-with-xml-schema>
</p:when>
<p:when test="ends-with(/*/@schema, '.sch')">
  <p:load name="loadedSchema">
    <p:with-option name="href"
      select="resolve-uri( /*/@schema, document-uri(/))"/>
  </p:load>
  <p:validate-with-schematron>
    <p:input port="source" select="/*/*[1]">
      <p:pipe port="result" step="candidate"></p:pipe>
    </p:input>
    <p:input port="schema">
      <p:pipe port="result" step="loadedSchema"/>
    </p:input>
    <p:input port="parameters"><p:empty/></p:input>
  </p:validate-with-schematron>
</p:when>
<p:otherwise>
```

```
          <p:error code="SchemaNotSupported">
            <p:input port="source">
              <p:inline>
                <message>Unsupported schema type!</message>
              </p:inline>
            </p:input>
          </p:error>
        </p:otherwise>
      </p:choose>
      <p:wrap match="/" wrapper="document"
        wrapper-namespace="http://www.oxygenxml.com/nvdl" wrapper-prefix="n"/>
      <p:add-attribute match="/*" attribute-name="status" attribute-value="valid"/>
    </p:for-each>
  </p:declare-step>
```

The schema detection is implemented right now based on extension but the XProc script can be modified to lookup the namespace of the root element of the schema and thus automatically use the corresponding validate step.

4. Conclusions and Further Work

The implementation of NVDL in XProc and XSLT makes NVDL more accessible as it can be used whereever there is an XProc engine. It also provides an insight into NVDL processing showing how it works in terms of XSLT, which many people already know, thus allowing them to understand quickly how NVDL works.

The implementation needs more tests and maybe also optimizations. Currently if the validation fails then the processing ends, it will be nice to change the XProc script to just annotate the dispatching result with the validation outcome for each validation candidate.

5. References

The NVDL standard is freely available from http://standards.iso.org/ittf/PubliclyAvailableStandards/c038615_ISO_IEC_19757-4_2006(E).zip

The oNVDL project is hosted on Sourceforge and its current status is described on the oXygen XML website http://www.oxygenxml.com/onvdl.html

JSONiq

XQuery for JSON, JSON for XQuery

Jonathan Robie
<jonathan.robie@emc.com>

Matthias Brantner
<matthias.brantner@28msec.com>

Daniela Florescu
<dana.florescu@oracle.com>

Ghislain Fourny
<ghislain.fourny@inf.ethz.ch>

Till Westmann
<till.westmann@28msec.com>

Abstract

XML and JSON have become the dominant formats for exchanging data on the Internet, and applications frequently need to send and receive data in many different JSON-based or XML-based formats. For XML data, a query language like XQuery can be used to query data, create or update data, transform it from one format to another, or route data. Adding JSON support to XQuery allows it to perform these tasks for both XML and JSON, combining data from multiple sources as needed. In addition, JSON support gives XQuery a light-weight, simple, and useful data structure that can often simplify queries.

JSONiq is a query language for JSON, based on XQuery. It is designed to allow an existing XQuery processor to be rewritten to support JSON with moderate effort. One profile of JSONiq removes everything directly related to XML, adding JSON constructors and navigation. Another profile of JSONiq includes the full XQuery language, with added JSON support, allowing queries to consume or produce JSON, XML, or HTML.

1. Introduction

XQuery is the standard query language for XML, and has been implemented in databases, streaming processors, data integration platforms, application integration platforms, XML message routing software, web browser plugins, and other environments. Both XML and JSON are both text formats that represent hierarchical data, each implies a data model, and both formats require a query language that can

easily query hierarchies to create hierarchies. JSON is now being used in many of the same environments as XML, and a variety of JSON query languages are emerging, including MongoDB's BSON, IBM's jaql, and CouchDB and Sqlite's UnQL. JSONiq is a query language based on XQuery, which makes it easy for an XQuery processor to support JSON in the same environments that currently use XQuery.

Many JSON programmers shy away from XQuery because they do not want the complexity of XML. For these programmers, we have developed a simpler profile called XQ--[1]. Because of the modular, compositional design of XQuery, it is easy to remove the XML-specific portions of the language, such as XML constructors and XML path expressions, and add the much simpler constructors and navigation needed for JSON. The resulting language is simpler and easier to optimize, and well suited to JSON views in middleware. But it includes sophisticated query capabilities, including grouping and windowing, that are very useful in a JSON environment. Unless stated otherwise, the queries in this paper use the XQ-- profile.

Another profile of JSONiq includes the full XQuery language, with added JSON support, allowing queries to consume or produce JSON, XML, or HTML. This profile is called XQ++[2]. Every valid XQ-- query has the same syntax and semantics in XQ++, and every valid XQuery has the same syntax and semantics in XQ++. XML continues to be widely used for data interchange on the Internet, and many applications need to process both JSON and XML. XML has significant advantages for document data, is well supported by standards, is part of a rich ecosystem of tools, libraries, and language extensions, and is required by many existing data interchange standards. Particularly in applications where data resembles human documents, including many healthcare, financial, government, intelligence, legal, and publishing applications, XML continues to have advantages over JSON — particularly when document data needs to be queried. Adding the JSONiq extensions to XQuery allows queries to process or produce JSON, XML, or HTML, combining and transforming data from any of these formats as needed. And adding JSON objects and arrays to XQuery also provides these useful data structures to programs that process only XML.

The W3C XSL Working Group has been working on support for maps, with proposals for importing and exporting JSON. At the time JSONiq was designed, we felt the XSL work was not yet sufficient for JSON processing, and the XQuery Working Group was not working on maps or JSON support. The XML Query Working Group is now working with the XSL Working Group to determine requirements for maps and for querying and processing JSON. Both the XSL Working Group proposal for maps and JSONiq are contributing to this work.

[1] http://jsoniq.org/grammars/xq--/ui.xhtml
[2] http://jsoniq.org/grammars/xq++/ui.xhtml

2. JSONiq in a Nutshell

JSONiq consists of the following extensions to XQuery:

- Support for JSON's datatypes, adding nulls, and mapping other JSON types to equivalent XML Schema types.
- Extensions to the XPath and XQuery Data Model (XDM) to support JSON objects, arrays, and object pairs.
- Navigation for JSON Objects and JSON Arrays.
- Constructors for JSON Objects, Pairs, and JSON Arrays, using the same syntax as JSON.
- Support for XQuery expressions within JSONiq Constructors, and for JSONiq constructors within XQuery expressions.
- Type matching expressions to allow the type of JSONiq datatypes to be specified in function parameters, return types, and other expressions that specify XQuery types.

JSONiq adds constructors for creating JSON objects and arrays, and member accessors for navigating them. JSON constructors use the same syntax as JSON objects and arrays. For instance, the following query creates a JSON object for a social media site:

```
{
  "name" : "Sarah",
  "age" : 13,
  "gender" : "female",
  "friends" : [ "Jim", "Mary", "Jennifer"]
}
```

JSONiq member accessors navigate JSON objects using the names of name/value pairs or the position of array items.

- If $o is an Object, then $o("n") returns the Pair named "n", or the empty sequence if no such pair exists.
- If $a is an Array, then $a($posn) returns the member at position $posn, or the empty sequence if no such member exists.
- If $p is an Pair, it delegates member accessors to its value.

For instance, if the above object is bound to the variable `$sarah`, then `$sarah("age")` returns `13`[3]. Member accessors can be chained to navigate down through objects and arrays. For instance, `$sarah("friends")(1)` returns `Jim`, the first friend in the array. In deeply nested objects, member accessor chains function like path expres-

[3]JSONiq member accessors use the same syntax as XQuery dynamic function invocation. If `$x` is bound to a function, then `$x("foo")` is a function call; if `$x` is bound to an object, then `$x("foo")` is a member accessor.

sions; for instance, the member accessor chain `$entry("app$control")("yt$state")("name")` navigates a Youtube feed.

JSONiq allows expressions in JSON constructors in the same way that XQuery allows expressions in XML constructors. The following JSONiq query creates a new user named "Jennifer", one year older than Sarah, with a friend list based on Sarah's. Jennifer does not appear on her own friends list, but Sarah does:

```
let $sarah := collection("users")[.("name") = "Sarah"]
return {
    "name" : "Jennifer",
    "age"  : $sarah("age") + 1,
    "friends" : [ values($sarah("friends")) except "Jennifer", "Sarah" ]
}
```

The result of the above query is:

```
{
    "name" : "Jennifer",
    "age"  : 14,
    "friends" : [ "Jim", "Mary", "Sarah"]
}
```

JSONiq also adds a few functions, including updating functions. But most of the power of JSONiq comes from the existing XQuery language, which is well designed for transformations on hierarchical structures, well understood, and widely implemented.

3. Grouping Queries for JSON

JSONiq allows the same functionality for JSON that XQuery provides for XML, except for functionality that depends directly on the properties of XML. This includes joins, grouping, and windowing. This section demonstrates this using a grouping example based on a similar example in the XQuery 3.0 Use Cases.

Suppose collection("sales") is an unordered sequence that contains the following objects:

```
{ "product" : "broiler", "store number" : 1, "quantity" : 20  },
{ "product" : "toaster", "store number" : 2, "quantity" : 100 },
{ "product" : "toaster", "store number" : 2, "quantity" : 50 },
{ "product" : "toaster", "store number" : 3, "quantity" : 50 },
{ "product" : "blender", "store number" : 3, "quantity" : 100 },
{ "product" : "blender", "store number" : 3, "quantity" : 150 },
{ "product" : "socks", "store number" : 1, "quantity" : 500 },
{ "product" : "socks", "store number" : 2, "quantity" : 10 },
{ "product" : "shirt", "store number" : 3, "quantity" : 10 }
```

We want to group sales by product, across stores.

Query:

```
{
  for $sales in collection("sales")
  let $pname := $sales("product")
  group by $pname
  return $pname : sum(for $s in $sales return $s("quantity"))
}
```

Result:

```
{
  "blender" : 250,
  "broiler" : 20,
  "shirt" : 10,
  "socks" : 510,
  "toaster" : 200
}
```

Now let's do a more complex grouping query, showing sales by category within each state. We need further data to describe the categories of products and the location of stores.

collection("products") contains the following data:

```
{ "name" : "broiler", "category" : "kitchen", "price" : 100, "cost" : 70 },
{ "name" : "toaster", "category" : "kitchen", "price" : 30, "cost" : 10 },
{ "name" : "blender", "category" : "kitchen", "price" : 50, "cost" : 25 },
{ "name" : "socks", "category" : "clothes", "price" : 5, "cost" : 2 },
{ "name" : "shirt", "category" : "clothes", "price" : 10, "cost" : 3 }
```

collection("stores") contains the following data:

```
{ "store number" : 1, "state" : CA },
{ "store number" : 2, "state" : CA },
{ "store number" : 3, "state" : MA },
{ "store number" : 4, "state" : MA }
```

The following query groups by state, then by category, then lists individual products and the sales associated with each.

Query:

```
{
  for $store in collection("stores")
  let $state := $store("state")
  group by $state
  return
     $state : {
       for $product in collection("products")
       let $category := $product("category")
       group by $category
```

```
        return
          $category : {
             for $sales in collection("sales")
             where $sales("store number") = $store("store number")
               and $sales("product") = $product("name")
             let $pname := $sales("product")
             group by $pname
             return $pname : sum( for $s in $sales return $s("quantity") )
          }
       }
}
```

Result:

```
{
  "CA" : {
      "clothes" : {
         "socks" :  510
      },
      "kitchen" : {
         "broiler" : 20,
         "toaster" : 150
      }
  },
  "MA" : {
      "clothes" : {
         "shirt" : 10
       },
      "kitchen" : {
         "blender" : 250,
         "toaster" : 50
      }
  }
}
```

4. JSON Views in Middleware

XQuery is used in middleware systems to provide XML views of data sources. Because JSON is simpler than XQuery, JSON-based views are an attractive alternative to XML-based views in applications that use large scale relational, object, or semi-structured data. JSONiq provides a powerful query language for systems that provide such views.

This example assumes a middleware system that presents relational tables as JSON arrays. The following two tables are used as sample data.

Table 1. Users

userid	firstname	lastname
W0342	Walter	Denisovich
M0535	Mick	Goulish

The JSON representation this particular implementation provides for the above table looks like this:

```
[
  { "userid" : "W0342", "firstname" : "Walter", "lastname" : "Denisovich" },
  { "userid" : "M0535", "firstname" : "Mick", "lastname" : "Goulish" }
]
```

Table 2. Holdings

userid	ticker	shares
W0342	DIS	153212312
M0535	DIS	10
M0535	AIG	23412

The JSON representation this particular implementation provides for the above table looks like this:

```
[
  { "userid" : "W0342", "ticker" : "DIS", "shares" : 153212312 },
  { "userid" : "M0535", "ticker" : "DIS", "shares" : 10 },
  { "userid" : "M0535", "ticker" : "AIG", "shares" : 23412 }
]
```

The following query uses the fictitious vendor's `vendor:table()` function to retrieve the values from a table, and creates an Object for each user, with a list of the user's holdings in the value of that Object.

```
[
  for $u in vendor:table("Users")
  order by $u("userid")
  return {
    "userid" : $u("userid"),
    "first" :  $u("firstname"),
    "last" :   $u("lastname"),
    "holdings" : [
         for $h in vendor:table("Holdings")
         where $h("userid") = $u("userid")
         order by $h("ticker")
         return {
```

```
            "ticker" : $u("ticker"),
            "share" : $u("shares")
          }
      ]
    }
]
```

5. JSON with XML and HTML

When JSONiq is used together with the full XQuery language, it can be used to convert data from one format to another, whether the formats use JSON, XML, or HTML. It can also be used to combine data from multiple formats, transform it, and create a result in any desired format.

For instance, suppose the following JSON data needs to be converted to HTML:

```
{
  "col labels" : ["singular", "plural"],
  "row labels" : ["1p", "2p", "3p"],
  "data" :
     [
        ["spinne", "spinnen"],
        ["spinnst", "spinnt"],
        ["spinnt", "spinnen"]
     ]
}
```

The following query uses the XQ++ profile of JSONiq. It creates an HTML table from this JSON Object, using the column headings and row labels specified:

```
<table>
  <tr> (: Column headings :)
     {
        <th> </th>,
        for $th in values(json("table.json")("col labels"))
        return <th>{ $th }</th>
     }
  </tr>
  {  (: Data for each row :)
     for $r at $i in values(json("table.json")("data"))
     return
        <tr>
         {
           <th>{ json("table.json")("row labels")($i) }</th>,
           for $c in values($r)
           return <td>{ $c }</td>
         }
        </tr>
```

```
    }
  </table>
```

The result of the query is the following HTML table:

```
<table>
   <tr>
     <th> </th>
     <th>singular</th>
     <th>plural</th>
   </tr>
   <tr>
     <th>1p</th>
     <td>spinne</td>
     <td>spinnen</td>
   </tr>
   <tr>
     <th>2p</th>
     <td>spinnst</td>
     <td>spinnt</td>
   </tr>
   <tr>
     <th>3p</th>
     <td>spinnt</td>
     <td>spinnen</td>
   </tr>
</table>
```

6. Conclusion

JSON and Cloud-based computing have brought new challenges to the Internet. XML is no longer the universal data interchange format once envisioned, but it is still widespread and XML-based services need to be used together with JSON-based services and various other data sources.

XQuery is a powerful and mature query language that has been implemented in many environments. By adding a small number of constructors and member accessors for JSON objects and arrays, JSONiq makes it possible for queries to consume, combine, or produce data in any JSON, XML, or HTML format, converting among them as needed. Any existing XQuery processor can add JSON support with moderate effort. Data integration was always an important application for XQuery, and adding support for JSON allows XQuery to play well with both of the dominant data interchange formats on the Internet. Where XML-based views of data such as relational databases are currently used, JSON-based views provide a simpler alternative.

In some implementations and environments, both JSON and XML will be supported. In other environments, only JSON support will be provided for the sake of simpler, more easily optimizable implementations.

Bibliography

[1] *JSONiq: Language Specification*[4]. Jonathan Robie. Matthias Brantner. Daniela Florescu. Ghislain Fourny. Till Westmann.

[2] *JSONiq: Use Cases*[5]. Jonathan Robie. Matthias Brantner. Daniela Florescu. Ghislain Fourny. Till Westmann.

[3] *XQuery 3.0: An XML Query Language. W3C Working Draft 13 December 2011.*[6]. World Wide Web Consortium. 13 December 2011.

[4] *XQuery and XPath Data Model 3.0. W3C Working Draft 13 December 2011.*[7]. World Wide Web Consortium. 13 December 2011.

[4] http://jsoniq.com/docs/spec/en-US/html/index.html
[5] http://jsoniq.com/docs/spec/en-US/html/index.html
[6] http://www.w3.org/TR/xquery-30/
[7] http://www.w3.org/TR/xpath-datamodel-30/

Corona: Managing and Querying XML and JSON via REST

Jason Hunter
MarkLogic Corporation

Abstract

What if you tried to provide the core value of MarkLogic Server to people who didn't want to learn XQuery? The very idea might be blasphemous in XML crowds, but is it possible? How far could you get?

This paper explores that issue and shares our experience in designing and developing the open source "Corona" project. Corona provides a set of REST endpoints to store, retrieve, query, and analyze documents held inside MarkLogic. Corona exposes the vast majority of MarkLogic's most popular XML-aware indexes and features, but in a way that lets them code in whatever language they'd like.

Note

The Corona project is in pre-release and details are subject to change. Source code is available at `https://github.com/marklogic/Corona`.

1. MarkLogic Architecture

First, a quick refresher. MarkLogic Server is a database purpose-built for Big Data. It's document-centric, transactional, search-centric, structure-aware, schema-agnostic, high performance, and clusters on commodity hardware. MarkLogic specializes in a new type of indexing that enables ad hoc queries against documents with widely varying schemas, delivering subsecond answers.

Developers write applications in MarkLogic using XQuery and XSLT. Sometimes they invoke the XQuery from Java, such as when integrating with a pre-existing Java stack, and sometimes they write the entire application in XQuery and XSLT and use MarkLogic directly as the application server. Usually XQuery acts as a scripting language and XSLT as a styling language.

Unfortunately XQuery, despite being a W3C standard, is not widely known. It's easy to learn and highly productive, but it's still a new language. People often want to solve problems with the minimal disruption possible, and learning a new language can feel like a disruption.

The challenge then is to expose the core MarkLogic functionality — the main things people do — as a set of services callable from other languages, letting people

be successful without knowing XQuery, or ideally even having special knowledge about MarkLogic's internals. We've been exploring to what extent this is possible with an open source project named Corona, and we've been very happy with the results.

2. Corona

Corona is built as a set of web endpoints exposing the core MarkLogic application services. It's an architecture often described as REST, even if it's not strictly RESTful in all aspects. By using REST principles it's easy to access Corona functionality from any environment, work with load balancers when in a clustered environment, and integrate with caching proxies to offload work.

During development thus far we've focused mostly on the REST layer; we plan to add standard language endpoints above this layer in the future, to make it possible for people in languages such as Java to make service calls without thinking about the HTTP underpinnings.

Note
If you'd like to review the REST API documentation as the functionality is described below, it's available at https://github.com/marklogic/Corona/wiki. This paper describes functionality rather than exact access methods.

3. Corona User Roles

Corona assumes three job roles for individuals:

1. The Corona Developer. This person does their day to day programming against the Corona endpoints. They're a pro with Java, .NET, Ruby, or some other language, and the Corona documentation is the only exposure they have to MarkLogic.
2. The Corona Admin. This person controls Corona's administrative settings. For example, they adjust current query settings, any stored transformations which may be called, and index settings. They do this via Corona endpoints separate from those available to the regular Corona Developer. They often dictate the document schema(s) for an application. They are familiar with MarkLogic behaviors, but they do not access MarkLogic's administrative port.
3. The MarkLogic Admin. This person installs MarkLogic, and uses MarkLogic's administrative port to manage forests, system uptime, and get Corona installed and started. They're the classic IT database administrator, often not a programmer, and don't need to be familiar with the applications being deployed.

We envision a typical project will have several Corona Developers, one or two Corona admins, and a shared MarkLogic Admin who assists with other projects as well. On a smaller project a single person could play all roles.

4. Storing Documents

Corona stores XML, JSON, text and binary documents. Simple REST calls are used to place a document, retrieve it, delete it, or replace it. MarkLogic natively supports XML, text, and binary. For JSON documents Corona is mapping them to XML documents, in a format designed for efficient queries.

Each document can have certain metadata associated with it:

Name	A unique name
Permissions	Security rules for what roles can view and modify the document; users and roles are managed by the MarkLogic Admin
Properties	Key-value metadata for the document
Collections	Named grouping for documents, as an alternative to implicit grouping by the directory path in the name
A Quality	An integer representing the intrinsic relevance of a document in a search
Extracted text and other metadata	For binaries, such as when inserting a JPEG the EXIF data will be extracted as metadata

5. Document Retrieval

Sometimes when retrieving a document you want the full document back and sometimes just a piece. To specify a piece you provide an extra parameter on the retrieval call. For XML documents this parameter is a simplified XPath expression (simplified for security purposes, to disallow a Corona Developer from executing arbitrary code). For JSON documents it's a JSON path (a custom notation invented here that looks like JavaScript object traversal, the closest standard yet available). This cuts down wire transmission overhead, especially for larger documents.

It's also possible to wholly transform the document as part of its retrieval. To do this you can specify a parameter on the retrieval call indicating the name of a transformer that should process the document. (You can transform a document as part of the insert call as well.)

Transformers are XQuery or XSLT scripts. They're specified by name, not as code. This is for security. The library of available transformers has to have been established earlier by a Corona Admin, using a separate and secured transformers

management REST endpoint. This prohibits regular developers from invoking arbitrary server-side code, an important feature since we assume they lack MarkLogic familiarity.

6. Search Queries

Corona includes extremely robust support for queries. Queries can be specified similar to a traditional database with value, range, and geospatial constraints; or like a search engine with free-text relevance-based language-aware constraints. (Having both in one transactional system is one of the advantages of MarkLogic.)

Query results can be sorted by search relevance or (soon) by a scalar such as a date or price. Result items can be paged (to view say 10 results at a time), snippeted (to show a blurb containing the matching terms), and highlighted (to bold the matching words).

A search result can include a simple description of the matching documents, or include the documents within the result as well, for efficiency by avoiding repeated web calls. When fetching the documents as part of a search, the same XPath/JSONPath subsetting options and transformation features are available.

Corona includes three ways to issue search queries:

6.1. Key/Value Query Service

This is a simple endpoint, for executing a quick retrieval based on a key (JSON key, XML element, etc) that's equal to a certain value.

6.2. String Query Service

This is a user-friendly way to specify a query as a specially marked-up string similar to those used by Google. This is something a Corona Developer could pass directly from the user interface text box to the Corona back-end for execution. It accepts queries using a "string query syntax". For example:

```
winter NEAR storm (title:Lebowski OR title:Country OR title:Fargo) AND
(cast:Buscemi OR cast:Jones) -director:Ethan
```

6.3. Structured Query Service

This is a programmer-friendly way to specify a query as a set of hierarchical query constraints expressed using an XML or JSON encoding. It accepts queries using the "structured query syntax" which is fully expressive and arbitrarily complex. For example:

```
{"and": [
  {
```

```
      "element": "author",
      "equals": "Noam Chomsky"  // Can be a boolean, number, string or array
    },
    {
      "range": "price",
      "from": 10.00,
      "to": 14.99
    },
    {
      "geo": "location", // The name of the geo index
      "region": { "polygon": [
        {"point": {"latitude": 1, "longitude": -1}},
        {"point": {"latitude": 1, "longitude": 1}},
        {"point": {"latitude": -1, "longitude": 1}},
        {"point": {"latitude": -1, "longitude": -1}},
        {"point": {"latitude": 1, "longitude": -1}}
      ]}
    }
  ]}
```

7. Search Configuration Management

It's often necessary for a Corona Admin to configure some aspects of the Corona environment to facilitate effective queries. Corona lets these admins manage several aspects of the environment.

7.1. Places

A Place gives an assigned name to a set of locations in a document, either JSON keys or XML nodes. For example, RSS has a variety of formats. A single place called "title" could be created that aliases "rdf:title" (RSS 0.9), "title" (RSS 0.91 thru 2.0, no namespace), and "atom:title" (Atom 1.0) into one.

Queries can use Places to indicate where a query constraint should apply. In string queries the Place name automatically becomes a field prefix, available to the user. A user can type title:"all the king's men" and Corona will understand that the phrase has to appear in one of the locations specified by the Place "title".

There's also a special place, the place without a name, which controls the behavior of searches that aren't field constrained. This is very important because the majority of users won't type fielded constraints.

When defining a Place you can assign relevance weights to each specified location. This helps maintain high-quality relevance-sorted results.

7.2. Ranges

A Ranges gives an assigned name to a location in a document, either a JSON key or XML node, that should be treated as a scalar value. For example, a "birthday" element might be assigned the type of date.

Each range creates an index in the background that enables:

1. Fast range queries on that scalar (i.e. limiting to dates between X and Y)
2. (Soon) Optimized sorting of results by that scalar (i.e. sort by date)
3. Fast extraction of the scalar's values (i.e. show birthday occurrences by month).

Range values can be assigned into named "buckets". Each bucket represents a subset of possible values for the scalar. For example, timestamps can be bucketed into days, dates can be bucketed into months, or prices can be bucketed into "Cheap" and "Expensive".

7.3. Named Queries

If a particular query is to be reused frequently, either alone or in combination with other query constraints, then for convenience and performance it can be registered as a named query. The query can then be embedded in another query just by specifying its name. Internally a named query gets optimized so repeated calls will be fast.

An upcoming feature will enable MarkLogic's powerful "reverse query" functionality against named queries. Reverse queries enable you to take a document and quickly determine which of a large set of pre-existing queries would match it. It's the reverse of a usual query. It's commonly used for alerting (tell me when…) or automated classification (route based on…). Technically it involves indexing the queries by making a decision tree out of them and running the document through the tree. With Corona a named query can have associated with it a URL, and that URL will be called whenever a new document arrives matching the named query.

7.4. Facets

A facet is data of a certain type often displayed next to a query that tells you analytics about the query results, such as the top senders for emails matching your query.

To execute a facet query you specify a Range name (or names) as well as an optional query, and Corona returns all the distinct values (or distinct bucket names) for documents matching the query, as well as the frequency count for each.

It's a fairly simple idea but it's tremendously powerful and enables accurate analytics against documents without pre-defining your dimensions. This technique is how MarkMail.org produces the facetes on the left hand side of each search result.

7.5. Namespaces

XML Namespaces are centrally managed in Corona. This allows all references to namespaced elements and attributes in Corona to simply use the namespace prefix and rely on the central management system to dictate the associated URI.

8. Transactions

Transactions are a core feature of MarkLogic and Corona exposes them so that multiple REST requests can be grouped together into a singular transaction, which at the end can be atomically committed or rolled back. There's one endpoint to start a transaction, one to commit a transaction, and one to rollback a transaction. Most requests accept an optional token to indicate in what transaction the request should be placed into. You can have any number of transactions in process at a given time.

9. Environment Variables

Environment variables allow a Corona administrator to adjust aspects of the Corona runtime. Much like UNIX environment variables, these are name-value pairs that control execution behavior. An important pair of variables dictates what, if any, transformers should execute during all insertions and retrievals. This allows a Corona Admin to, for example, enforce an XML Schema for all incoming documents or invisibly adjust the document format (and reverse the adjustment on output) to optimize some query.

10. What's Next?

That was a long list of features. What's next?

The biggest remaining item, besides the higher-level language bindings discussed earlier, is extensibility. Corona should allow Corona Admins to add new REST endpoints in a supported manner. If there's something that needs to get done, which Corona doesn't expose, a new endpoint should make it possible. Often times we expect custom endpoints to be used to achieve a performance gain by moving some business logic execution closer to the data. It might also be used to expose MarkLogic features that Corona isn't directly supporting: entity enrichment, thesaurus expansion, spell check, SVM classifier, document fragmentation, schema validation, highly structured properties, and custom snippeting. The list of what you can do in custom code is long — just about limitless — and custom endpoints should make sure Corona never paints a user into a corner without an escape hatch.

11. Discussion

Does Corona satisfy the goal of providing the core value of MarkLogic Server to people who didn't want to learn XQuery? We think yes.

It provides to a Corona Developer a highly scalable document store which should appear as a simple web service with enough power to handle more than a billion documents. It makes it easy for that developer to run advanced queries (value, range, text, geospatial) against the documents, using either a string or a structured query syntax. It exposes fast aggregates in the form of facets. It even includes multi-statement transactions.

It isolates to the Corona Admin the special knowledge about MarkLogic and lets them use XQuery and XSLT as needed, to support document transformations as well as adding new endpoints.

It also isolates MarkLogic administration expertise to the MarkLogic Admin who should be able to support multiple applications without particular knowledge about any of them.

At the end of the day, Corona should make it easier for everybody on a project to get the maximum value out of MarkLogic while minimizing how much expertise they need to develop. We think that's terrific.

Treating JSON as a subset of XML

Using XForms to read and submit JSON

Steven Pemberton
CWI, Amsterdam
<steven.pemberton@cwi.nl>

Abstract

XForms 1.0 was an XML technology originally designed as a replacement for HTML Forms. In addressing certain shortcomings of XForms 1.0, the next version, XForms 1.1 became far more than a forms language, but a declarative application language where application production time could be reduced by an order of magnitude compared with traditional procedural programming.

Although XForms treats its data internally as if it is XML, using XPath both to address data and to calculate new values, it is not the intention that external data necessarily be only in XML.

An obvious data format widely in use on the web is JSON. There are several mappings defined in both directions between XML and JSON, but largely because JSON can only represent a subset of what XML can represent, many of the mappings are cumbersome, and make data-references both JSON-specific, and difficult to write.

Ideally, an XForm processing JSON data shouldn't have to know which data format has been used, so that JSON data can be selected with natural XPath selectors. Furthermore, XForms doesn't need the full generality of translating any XML to JSON, since the only need is to read and write data to and from existing JSON sources. In other words, it only needs to process existing JSON. This simplifies the mapping, and makes the selectors needed with minor exceptions opaque to the data format.

This paper presents the mapping proposed for XForms 2.0, the special cases that have had to be dealt with, and discusses generalisation to other formats, such as VCARD.

Keywords: XForms, JSON, XML

1. Introduction

XForms is an XML technology originally designed as a replacement for HTML Forms [7]. It was designed by doing an analysis of HTML features, and a requirements analysis derived from usage of HTML Forms and other electronic forms systems [XForms requirements]. The resultant design used a MVC-based approach

to the data, used intent-based controls, and as well as the standard URL and POST forms of data submission, added XML as a first-class data format, both for initialising data from external sources, as for submission.

What this concretely means is that the data is physically separated from the controls in the form. The data is placed in the head of the document, and the controls bind to the data.

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <model xmlns="http://www.w3.org/2002/xforms">
       <instance>
          <data xmlns=""><year>2012</year>...</data>
       </instance>
    </model>
  </head>
<body>
```

Controls in the body refer to values in the data instance(s) using XPath expressions [15]:

```
<input ref="year">...
<input ref="event[1]/title/@language">...
```

The controls can be initialised by putting values in the data:

```
<data xmlns=""><year>2001</year>...</data>
```

or the data can also be initialised from external sources:

```
<instance src="http://www.example.org/events"/>
```

Relationships between, and restrictions on, values can be specified in the model, allowing dependent values to be calculated automatically and data checking to be performed on the client rather than on the server.

```
<bind nodeset="year" constraint=". &gt; 1752"/>
<bind nodeset="state" required="../country = 'USA'"/>
<bind nodeset="age" calculate="../thisyear - ../birthdate/year"/>
```

Values can be exposed in the document itself, using an output control:

```
The result for the year <output ref="year"/> is ...
```

Controls are intent-based, by expressing what the control should do, rather than how it should look. So a control like this:

```
<select1 ref="colour">
   <label>Colour:</label>
   <item><label>red</label><value>#ff0000</value></item>
   <item><label>green</label><value>#00ff00</value></item>
   <item><label>blue</label><value>#0000ff</value></item>
 </select1>
```

can be represented in different ways depending purely on styling: as radio buttons, a drop-down menu, a select box, or anything else that the designer can think of. It also makes it easier to make forms device independent and accessible, since there is no explicit binding to presentation.

Further details of XForms can be found in the Quick Reference [11], Tutorial [XForms Tut], and Specification. [XForms 1.1]

2. Experience

Initial experience showed XForms to be far more powerful and flexible than the HTML Forms it was replacing, but that it had too slavishly followed the HTML design in some aspects, particularly in the use of fixed strings rather than (potentially) calculated values for such things as the submission URI. As a consequence this restricted what was possible with the language.

As a consequence, XForms 1.1 [9] addressed these shortcomings, and the resultant language turned out to be far more than a forms language, but a declarative application language. Since XForms has input, output, and a processing engine, XForms is Turing-complete, and much more than just forms is now possible with the language.

Experience with some large projects has shown that application production time can be reduced by an order of magnitude compared with traditional procedural programming [16], with one large project reporting a reduction from 5 years with 30 programmers using traditional programming, to 1 year with 10 programmers using XForms.

3. Data Opacity

Although XForms treats its data internally as if it is XML, using XPath both to address data as to calculate new values, it is not the intention that external data be only in XML (clearly, considering the other formats produced by XForms, such as URL-encoding). Just as a photo editor in general doesn't care in what format the image is kept externally in order to be able to edit the image, neither does XForms require the external data to be in XML form. However, since the *internal* form of the data that XForms deals with is XML (since the data is accessed using XPath), there has to be a mapping between the external form and the internal one.

An obvious data format widely in use on the web is JSON [4]. There are several mappings defined in both directions between XML and JSON, for instance Badgerfish [1], and JXON [5], but largely because JSON can only represent a subset of what XML can represent, many of the mappings are cumbersome, and make data-references both JSON-specific, and difficult to write.

For instance, just to take one example, here of the mapping from JXON, the following XML:

```
<BOOKS>
  <BOOK id="1">
    <TITLE>My Favorite Book</TITLE>
    <PRICE>1.23</PRICE>
  </BOOK>
  <BOOK id="1a">
    <TITLE>XML for Dummies</TITLE>
    <PRICE>5.25</PRICE>
  </BOOK>
  <BOOK id="3">
    <TITLE>JSON for Dummies</TITLE>
    <PRICE>200.95</PRICE>
  </BOOK>
</BOOKS>
```

would be transformed [5] into:

```
{
 "childNodes": [
  {
   "childNodes": [
    {
     "childNodes": ["My Favorite Book"],
     "tagName": "TITLE"
    },
    {
     "childNodes": [1.23],
     "tagName": "PRICE"
    }
   ],
   "id": 1,
   "tagName": "BOOK"
  },
  {
   "childNodes": [
    {
     "childNodes": ["XML for Dummies"],
     "tagName": "TITLE"
    },
```

```
      {
       "childNodes": [5.25],
       "tagName": "PRICE"
      }
     ],
     "id": "1a",
     "tagName": "BOOK"
    },
    {
     "childNodes": [
      {
       "childNodes": ["JSON for Dummies"],
       "tagName": "TITLE"
      },
      {
       "childNodes": [200.95],
       "tagName": "PRICE"
      }
     ],
     "id": 3,
     "tagName": "BOOK"
    }
   ],
   "tagName": "BOOKS"
  }
```

It is left as an exercise to the reader to derive the equivalent JSON selector for the XPath `BOOKS/BOOK[1]/@title`.

4. JSON in XForms

During the design phase of a suitable mapping for JSON for the coming XForms 2.0 standard [10], we went through several iterations before coming to a key realisation: since the aim is only to address existing JSON stores, it is not necessary to be able to convert every possible XML representation into an equivalent JSON representation, only the reverse. This reduces the task considerably, since it means several features of XML do not have to be addressed, such as namespaces, attributes, and mixed content.

Some of the requirements for a mapping from JSON to XML for XForms included:

- All possible JSON values be representable
- Round-trippable, so that you can both read from and submit to a JSON store.
- As natural-looking selectors as possible.

Ideally, an XForm processing JSON data shouldn't have to know which data format has been used; so that, for instance, data such as

```
{"company":"example.com", "locations":[{"city": "Amsterdam"},{"city": "London"}]}
```

with the right mapping could be selected with XPath selectors like

```
locations/city[1]
```

In this way data could be loaded using content negotiation [HTTPG], and will work whether the data comes in as XML or JSON.

The basic mapping designed is rather simple [13]. Since JSON has no attributes, all content can be represented in elements, and attributes are therefore free to be used to help with the mapping.

Since a JSON value can have several values at the top level, a root element is used `<json>`. JSON names become XML elements:

```
{"name": "XForms"}
```

becomes

```
<json><name>XForms</name></json>
```

Strings are the default datatype. In order to allow the processor to distinguish between {"size": 30} and {size: "30"} when serialising, other types are marked:

```
"age": 21
```

becomes

```
<age type="integer">21</age>
```

and

```
"registered": true
```

becomes:

```
<registered type="boolean">true</registered>
```

Nested values are obvious:

```
"name": {"given": "Isaac", "family": "Newton"}
```

becomes

```
<name><given>Isaac</given><family>Newton</family></name>
```

Arrays are marked specially:

```
"colour": ["red", "green", "blue"]
```

becomes

```
<colour starts="array">red</colour><colour>green</colour><colour>blue</colour>
```

This allows selectors like `colour[3]` to work, but also allows to distinguish things like single element arrays:

```
{city: ["Amsterdam"]}
```

from

```
{city: "Amsterdam"}
```

and empty arrays:

```
{"set": []}
```

from

```
{"set": ""}
```

5. Example

To take an example from the JSON site:

```
{"bindings": [
        {"ircEvent": "PRIVMSG", "method": "newURI", "regex": "^http://.*"},
        {"ircEvent": "PRIVMSG", "method": "deleteURI", "regex": "^delete.*"},
        {"ircEvent": "PRIVMSG", "method": "randomURI", "regex": "^random.*"}
    ]
}
```

would become

```
<json>
   <bindings starts="array">
      <ircEvent>PRIVMSG</ircEvent><method>newURI</method><regex>^http://.*</▶
regex>
   </bindings>
   <bindings>
     <ircEvent>PRIVMSG</ircEvent><method>deleteURI</method><regex>^delete.*</▶
regex>
   </bindings>
   <bindings>
     <ircEvent>PRIVMSG</ircEvent><method>randomURI</method><regex>^random.*</▶
regex>
   </bindings>
</json>
```

and a JSON selector like

```
bindings[0].method
```

would become in XPath

```
bindings[0]/method
```

6. Special Cases

There are a small number of special cases that have to be accounted for:

- JSON allows the empty name "", which XML does not allow.
- JSON names may contain characters that are not allowed as name characters in XML.
- JSON strings may contain any Unicode character; XML disallows most characters below #x20 [14].

The first two are easy to deal with: any character that is not possible in XML is replaced with an underscore, and an attribute `name` is added to the element giving the correct name. The empty name is replaced with a single underscore, and an empty `name` attribute is used.

For example:

```
"$": "$"
```

would be transcribed:

```
<_ name="$">$</_>
```

The third is harder to deal with, with an example being:

```
{"backspace": "\b"}
```

The backspace character is completely disallowed in XML (even hex encoded), leaving the only option to leave those illegal characters encoded in JSON notation.

7. Implementation

Implementation of the mapping is relatively trivial: at the point where an implementation normally receives a document of type `application/xml` (or similar), either during initial instance initialisation from an external resource, or as the return value of a submission, if the media type of the resource is `application/json`, the resource can be parsed, and transformed to an equivalent XML instance, as described above. The media type can be recorded as an attribute of the root element, so that it can be reused if the instance is to be resubmitted as JSON.

8. Extension to other formats

Clearly this method can be extended to other datatypes such as VCARD [6] and iCalendar [iCal]. For instance an iCalendar value such as

```
BEGIN:VCALENDAR
METHOD:PUBLISH
PRODID:-//Example/ExampleCalendarClient//EN
VERSION:2.0
BEGIN:VEVENT
ORGANIZER:mailto:a@example.com
DTSTART:19970701T200000Z
```

```
DTSTAMP:19970611T190000Z
SUMMARY:ST. PAUL SAINTS -VS- DULUTH-SUPERIOR DUKES
UID:0981234-1234234-23@example.com
END:VEVENT
END:VCALENDAR
```

can be transformed to

```
<VCALENDAR>
  <METHOD>PUBLISH</METHOD>
  <PRODID>-//Example/ExampleCalendarClient//EN</PRODID>
  <VERSION>2.0</VERSION>
  <VEVENT>
    <ORGANIZER>mailto:a@example.com</ORGANIZER>
    <DTSTART>19970701T200000Z</DTSTART>
    <DTSTAMP>19970611T190000Z</DTSTAMP>
    <SUMMARY>ST. PAUL SAINTS -VS- DULUTH-SUPERIOR DUKES</SUMMARY>
    <UID>0981234-1234234-23@example.com</UID>
  </VEVENT>
</VCALENDAR>
```

9. Conclusions

Due to the lack of a need to represent arbitrary XML in JSON, dealing with external JSON values in XForms becomes easy, and natural, in most cases not even exposing the fact that the external data type is not XML in the XForm. The approach can be extended to other types, and thanks to the generality of XML, mostly without restriction.

Bibliography

[1] David Sklar, What is Badgerfish?, http://www.sklar.com/badgerfish/

[2] R. Fielding, *et al.*, Hypertext Transfer Protocol -- HTTP/1.1, ISOC 1999 http://www.w3.org/Protocols/rfc2616/rfc2616-sec12.htmlhttp://www.w3.org/Protocols/rfc2616/rfc2616-sec12.html

[3] B. Desruisseaux, Ed., Internet Calendaring and Scheduling Core Object Specification (iCalendar), IETF 2009, http://tools.ietf.org/html/rfc5545http://tools.ietf.org/html/rfc5545

[4] Introducing JSON http://www.json.org/http://www.json.org/

[5] David Lee, JXON: an Architecture for Schema and Annotation Driven JSON/XML Bidirectional Transformations, Proc Balisage, 2011. http://www.balisage.net/Proceedings/vol7/html/Lee01/BalisageVol7-Lee01.htmlhttp://www.balisage.net/Proceedings/vol7/html/Lee01/BalisageVol7-Lee01.html

[6] S. Perreault, vCard Format Specification, IETF, 2011 http://tools.ietf.org/html/rfc6350http://tools.ietf.org/html/rfc6350

[7] Micah Dubinko *et al.* (eds.), XForms 1.0, W3C 2003. http://www.w3.org/TR/2003/REC-xforms-20031014/http://www.w3.org/TR/2003/REC-xforms-20031014/

[8] Micah Dubinko *et al.* (eds.), XForms Requirements, W3C 2001, http://www.w3.org/TR/xhtml-forms-reqhttp://www.w3.org/TR/xhtml-forms-req

[9] John Boyer (ed.), XForms 1.1, W3C 2009, http://www.w3.org/TR/2009/REC-xforms-20091020/http://www.w3.org/TR/2009/REC-xforms-20091020/

[10] John M. Boyer, *et al.* (eds.), XForms 2.0, W3C 2012, http://www.w3.org/MarkUp/Forms/wiki/XForms_2.0http://www.w3.org/MarkUp/Forms/wiki/XForms_2.0

[11] Steven Pemberton, XForms 1.1 Quick Reference, W3C 2010, http://www.w3.org/MarkUp/Forms/2010/xforms11-qr.htmlhttp://www.w3.org/MarkUp/Forms/2010/xforms11-qr.html

[12] Steven Pemberton, XForms for HTML Authors, W3C 2010,-authors/ http://www.w3.org/MarkUp/Forms/2010/xforms11-for-html-authors/http://www.w3.org/MarkUp/Forms/2010/xforms11-for-html-authors/

[13] Steven Pemberton et al., JSON-based instances and submissions, W3C 2011, http://www.w3.org/MarkUp/Forms/wiki/Jsonhttp://www.w3.org/MarkUp/Forms/wiki/Json

[14] Tim Bray, *et al.* (eds.), Extensible Markup Language (XML) 1.0 (Fifth Edition), http://www.w3.org/TR/REC-xml/#charsetshttp://www.w3.org/TR/REC-xml/#charsets

[15] James Clark, *et al.* (eds.), XML Path Language (XPath), W3C 1999, http://www.w3.org/TR/xpath/http://www.w3.org/TR/xpath/

[16] Steven Pemberton, XRX - Restful XForms, CWI 2011, http://www.cwi.nl/~steven/Talks/2011/07-05-steven-xrx/#appshttp://www.cwi.nl/~steven/Talks/2011/07-05-steven-xrx/#apps

RESTful XQuery

Standardised XQuery 3.0 Annotations for REST

Adam Retter
Adam Retter Consulting
<adam@adamretter.org.uk>

Abstract

Whilst XQuery was originally envisaged and designed as a query language for XML, it has been adopted by many as a language for application development This, in turn, has encouraged additional and diverse extensions, many of which could not easily have been foreseen.

This paper examines how XQuery has been used for Web Application development, current implementation approaches for executing XQuery in a Web context, and subsequently presents a proposal for a standard approach to RESTful XQuery through the use of XQuery 3.0 Annotations.

Keywords: Query 3.0, Annotations, REST, HTTP, Standard

1. Introduction

1.1. Background

XML Query Language (XQuery) was originally born from several competing query languages for XML[1]. All of these languages had in common the noble yet limited goal of querying XML. They focused on XML as a read-only store for data. In addition, whilst several of these predecessors recognised the Web as a critical factor, like their successor XQuery, none of them attempted to implement constructs in the language that supported use as a (Web) server-side processing language.

With the adoption and use of XQuery, because of its functional nature and module system which permit the organisation of code units, people attempted to write complex processing applications in XQuery. As the limits of what was achievable in XQuery were tested, real world scenarios emerged which called for additional XQuery facilities, resulting in extension standards: XPath and XQuery Update[2] and XQuery Full-Text[3].

Triggered by XQuery users developing increasingly complex applications in XQuery, and the understanding that XQuery could easily produce XHTML, an XQuery processor operating on an XML Database was for the first time in 2003 coupled with a Web Server and REST interface in the eXist Native XML Database project[4][5].

With the advent of being able to use XQuery as a server-side processing language, developers were soon building complete data driven Web Applications entirely in XQuery.

Today most XQuery vendor's products operating on collections of XML documents, provide some mechanism for invoking this processing from the Web by URI[6–10].

The W3C XQuery Working Group has itself recognised the value in XQuery as a general purpose processing language through a new extension standard which enhances XQuery for this purpose: XQuery Scripting Extensions[11].

1.2. Problem Statement

The value in using XQuery as a server side processing language is well recognised by both vendors, users and the XQuery Working Group. However, to date there has been no effort to standardise how XQuery may be invoked in a Web context. Presently, each vendor has their own non-standard approach to wiring Web requests and XQuery scripts together; which in-turn causes developers to create non-portable platform dependent XQuery when coding for the Web.

Sadly non-portable XQuery code that relies on vendor extensions or mechanisms, limits and fragments the XQuery community; it is much harder to share useable code and promote an environment of learning from peers and building on existing work, when code that one would hope should run on any XQuery processor simply cannot.

Efforts such as the EXPath[12] and EXQuery[13] projects have attempted to promote portable XQuery code by creating community standardised versions of existing vendor extensions. The EXPath project has attempted to standardise a HTTP Client[14] request module for XQuery. However, there is no such vendor independent standard for invoking server-side XQuery on the Web.

1.3. Contributions

The W3C XQuery 3.0 language specification[15] (currently a Last Call Working Draft) introduces several new features to XQuery. This paper proposes a new vendor agnostic standard for invoking XQuery from the Web based on the new feature of Annotations present in XQuery 3.0.

1.4. Outline

This paper first attempts a brief description of the fundamentals of XQuery and REST and why it is desirable to combine these in Section 2. Section 3 reviews and critiques several current approaches. Based on this knowledge, a standard for a vendor agnostic approach is proposed in Section 4. Section 5 describes a concrete

technical implementation of the proposed standard and Section 6 discusses the conclusions of this work and possible future work.

2. Fundamentals

2.1. XQuery

XML Query Language (XQuery) is a W3C Recommendation[1] for writing queries against the XPath and XQuery Data Model (XDM)[16], which is to say, the logical structure of XML documents. Now in its Second Edition of Version 1.0, the W3C XQuery Working Group is currently finalising the new upcoming version, numbered 3.0[15]. XQuery is a Turing-complete[17] functional programming language which is centred around FLWOR (For Let Where Order-by Return) statements which utilise path expressions to address the XDM. XQuery code may be grouped into functions and modules, of which there is always a main module where processing begins.

Whilst XQuery is most usually used for querying XML documents, there are several serialization options for the results of an XQuery: XML, XHTML, HTML and Text; The specification of the serialization mechanism has been formalised in XQuery 3.0. The capability to produce XHTML, HTML from querying XML documents whilst remaining in the same data type model, removes the impedance mismatch between the data query language and application programming language that is present in other environments[18]; XQuery therefore lends itself well to rapid (X)HTML generation. The ability to also serialize results as Text allows for dynamic generation of CSS, JavaScript and JSON amongst others, indeed some vendors already provide a JSON serialization option for their XQuery implementations.

Example 1. XQuery 3.0: Querying XML and generating some simple XHTML

```
xquery version "3.0";

declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
declare option output:method "xhtml";
declare option output:version "1.1";
declare option output:doctype-system "http://www.w3.org/TR/xhtml11/DTD/▶
xhtml11.dtd";
declare option output:doctype-public "-//W3C//DTD XHTML 1.1//EN";

<html xmlns="http://www.w3.org/1999/xhtml">
    <head>
        <title>Word of the Day</title>
    </head>
```

```
    <body>
        <h1>Hello</h1>
        <p>Todays special word is:
       <span>{doc("mydoc.xml")//word[xs:date(@date) eq current-date()]/text()}</▶
span>
        </p>
    </body>
</html>
```

2.1.1. XQuery 3.0 Annotations

Whilst XQuery 3.0 introduces many new features, an understanding of Annotations in XQuery 3.0 is fundamental to the contribution of this paper. Annotations declare properties of functions or variables, zero or more annotations may be added to a function or variable declaration. Annotations start with the '%' character and consist of an expanded qualified name and an optional value, the value being a sequence of literals.

Example 2. XQuery 3.0 Annotations

```
xquery version "3.0";

declare namespace java = "http://java";

declare
%java:method("java.lang.Math.sin")
function local:calculate-sin($a as xs:double) as xs:double external

<sin>{
    local:calculate-sin(1.4)
}</sin>
```

Apart from the `%private` and `%public` annotations, no other annotations are defined by the XQuery 3.0 specification. However, the specification states, “Implementations MAY define further annotations, whose behaviour is implementation-defined”, and it is this property which this paper exploits to define a standard set of RESTful Annotations for XQuery 3.0.

2.2. REST

Representational State Transfer (REST) is an architectural style developed by Dr. Roy Fielding for his doctoral thesis. REST describes the architectural design principles of the evolved Web and remains abstract from the implementation: “REST is defined by four interface constraints: identification of resources; manipulation of resources

through representations; self-descriptive messages; and, hypermedia as the engine of application state"[19]. Applications that adhere to REST are described as RESTful.

The Web utilises Hyper-Text Transfer Protocol (HTTP) as its transport and Uniform Resource Identifier (URI) as its addressing mechanism, and can be described as RESTful.

However, Web Sites built with HTML (and possibly JavaScript) typically only use a subset of the full HTTP capabilities; commonly just HTTP verbs GET and POST, with a blanket HTTP Accept header which allow for the retrieval of resources and transmission of simple form data and encoded files. In contrast, RESTful Web Services implemented over HTTP, may use the full range of HTTP verbs and HTTP content negotiation of resources to retrieve or store representations against rich descriptive URI namespaces. Such RESTful applications benefit over the incumbents (SOAP, RPC, etc.), in that additional vocabularies and technology are not required; they, like the Web, are submittable to the same caching, transformation and intermediate security mechanisms due to the common layered architecture.

Arguably, RESTful HTTP Web Services, through their use of descriptive verbs (e.g. GET, PUT, POST, DELETE, OPTIONS etc.) and transfer of representations, are perhaps more applicable to Document Management Systems, such as XML Databases, than they are to the hypermedia systems like the Web which are largely still read-only.

It is the RESTful HTTP Web Service style of REST that shall be considered from hereon in this paper.

2.3. XQuery and REST together

XQuery processors are often integrated with large systems that maintain many XML documents, such as XML Databases. Considering the following as Resources in REST terminology, XML documents, or representations of, such as the output of an XQuery; RESTful HTTP Web Services have many desirable and symbiotic properties for addressing and describing manipulations of said XML resources, whilst XQuery affords the implementation power to realise such implementation.

3. Review of Current Approaches

Current approaches to invoking XQuery in a RESTful manner using HTTP are examined in this section.

There are many vendors of XQuery processors[21], most of the Document Repository or XML Database vendors with such processors provide HTTP API's to the document store with the facility to invoke XQuery scripts remotely against the document store. The notable exception to this is Servlex, an implementation of EXPath Webapp Module. Rather, it is concerned with mapping URI's to the invocation of any Servlex Servlet (i.e. an XPath function, XSLT named template, XQuery main

module, XSLT stylesheet, XProc pipeline or XProc step)[22] without concern for document stores.

Providing a complete review of all current approaches would be too resource intensive and lengthy for this paper; as such, four vendors' products have been chosen for examination. These have been chosen to represent products which may contrast, yet are widely used in the industry, and/or provide justification and inspiration for developing a standard approach based on XQuery 3.0 Annotations.

3.1. eXist-db

eXist-db was chosen because 1) it is arguably the most popular and widely used Open Source Native XML Database, 2) it appears to have been the first such product to offer the facility to invoke XQuery via HTTP and 3) it has a history of transparently mapping XQuery functions directly to HTTP APIs, in the form of its XQuery SOAP Server[23].

eXist-db provides two mechanisms for invoking XQuery in a RESTful manner over HTTP.

3.1.1. REST Server

The first mechanism relies on a REST Server which is embedded into eXist-db. This REST Server allows any resource stored into the database to be addressed by a URI, and the database content to be manipulated by HTTP POST, PUT and DELETE, but it does not support content negotiation.

For example if one wanted to retrieve the XML document on Hamlet from the Classics, Shakespeare collection, a HTTP GET on *http://localhost:8080/exist/rest/db/classics/shakespeare/hamlet.xml* may retrieve the desired document.

eXist-db provides three proprietary XQuery function modules for accessing the HTTP context of the REST Server, namely, *Request Module*, *Response Module* and *Session Module*. XQuery invocation by REST Server is possible by:

1. Sending an XQuery to the REST Server for execution against a database collection or document URI context.
   a. For HTTP GET a parameter may be appended to the URL query-string which contains an XQuery to execute. For example to retrieve speeches given in Hamlet - *http://localhost:8080/exist/rest/db/classics/shakespeare/hamlet.xml?_query=//speech* or for example, to retrieve speeches given in any Shakespeare script - *http://localhost:8080/exist/rest/db/classics/shakespeare/?_query=//speech*
   b. For more complex XQuerys, the XQuery may be wrapped in a CDATA section of a simple XML document and sent to the server URI by HTTP POST.

2. An XQuery main module, and supporting modules may be pre-stored into the database. This approach is akin to stored procedures in a relational database, however the REST server makes the main module invokable by URI. For example performing a HTTP GET or POST against *http://localhost:8080/exist/rest/db/some-script.xqy* would invoke the some-script.xqy XQuery main module.

The REST Server is a powerful mechanism that has been used to build entire and complex enterprise web applications in pure XQuery, however URI's serviced by the REST Server implicitly mirror the database collection hierarchy in eXist-db which is not always desirable. Hyperlinking to child resources and collections are implicit in the response (when requesting a database or collection), thus supporting the REST promise of hypermedia for application state. However, when requesting resources themselves, there is no mechanism for hyperlinking to related resources, a disadvantage when compared to xDB's XML REST Framework in Section 3.3.3. In addition parameters can only be passed to XQuery modules via HTTP through URL query-string parameters or POST'ed form fields, which leads to complex URLs. This makes building applications with simple, logical and descriptive URI schemes a challenge.

3.1.2. XQuery URL Rewriting

XQuery URL Rewriting filters all HTTP requests to eXist-db. If a HTTP request URI matches a prefix defined in a configuration file, then that URI is mapped to a collection in the database. Typically that collection may then contain a Controller written as an XQuery main module and named '*controller.xql*'. If a Controller is found, then processing of the HTTP request is handed to the XQuery, and the XQuery has absolute access to the HTTP request and response and may take any action it wishes, returning any desirable HTTP response and status code, or handing the request off to another XQuery or Servlet.

Example 3. eXist-db, URL Rewriting Configuration snippet

```
<forward pattern="/webdav/" servlet="milton"/>
<forward pattern="/atom/" servlet="AtomServlet"/>
<root pattern="/solutions" path="xmldb:exist:///db/apps/local/solutions/"/>
<root server-name="www.example.com" pattern="/*" path="xmldb:exist:///db/com/▶
example/www/"/>
```

Example 4. eXist-db, URL Rewriting Controller snippet (controller.xql)

```
(: homepage :)
if($exist:path eq "/" or $exist:path eq "/home.xml") then
    template:process-template($rel-path, $exist:path, $DEFAULT-TEMPLATE, ▶
($menus, fn:doc(fn:concat($rel-path, "/home.xml"))))
```

```
(: login page :)
else if($exist:path eq "/login") then
    if(security:login(request:get-parameter("username", "unknown"), ▶
request:get-parameter("password", "unknown")))then
        local:redirect("entry/browse")
    else
        local:redirect("./?login=failed")

(: user sign-up page :)
else if($exist:path eq "/register") then
    if(request:get-method() eq "GET")then
       template:process-template($rel-path, $exist:path, $DEFAULT-TEMPLATE, ▶
($menus, fn:doc(fn:concat($rel-path, "/registration.xml"))))
    else if(request:get-method() eq "POST")then
        let $request-data := request:get-data()/user return
            if(security:register-user($request-data))then
                local:redirect("entry/browse")
            else
            (
                (: could not register the user - xform will show error :)
                response:set-status-code(400),
              <message>Unable to register the user '{$request-data/username}'</▶
message>
            )
    else
        local:ignore()
else
    local:ignore()
```

XQuery URL Rewriting is much more flexible than the REST Server as it decouples the logical application URI space from the logical database URI space, it also allows you complete control over the lifecycle of HTTP Requests made to the database. However, with this flexibility comes complexity. For real-world enterprise applications the Controller can end up becoming several thousand lines of XQuery code, which is really just encoding if/else statements to match URI patterns and/or HTTP actions. The order of statements in the Controller becomes a concern, and this non-declarative approach becomes very hard to maintain and debug as the code size grows.

3.2. MarkLogic

MarkLogic was chosen because 1) it is almost certainly the most successful commercial Native XML Database Server, 2) whilst a newer creation than eXist-db, it ultimately provides similar REST capabilities but through a different approach, and in addition has layered some new frameworks atop these, 3) MarkLogic caters for the

enterprise market whereas Open Source projects like eXist-db are better known in smaller, educational or public institutions.

MarkLogic provides three mechanisms for invoking XQuery in a RESTful manner over HTTP.

3.2.1. HTTP App Server

MarkLogic's HTTP App Server varies somewhat from that of eXist-db's REST Server. MarkLogic differentiates between a Modules database and a Content database.

Resources stored into the Content database (i.e. XML documents) are not directly accessible by URI from the HTTP App Server. They may only be accessed via an XQuery Module. Resources stored into the Modules database (i.e. XQuery Modules, or Binary files such as JPEG, CSS etc) are all accessible by URI. An advantage over eXist-db is that resources stored into the database may be assigned an arbitrary logical URI, rather than the URI being representative of a logical collection or folder hierarchy.

XQuery invocation by HTTP App Server is possible by pre-storing XQuery modules into MarkLogic's Modules database. Like eXist-db, this is akin to stored procedures in a relational database. Likewise, the HTTP App Server makes the main module invokable by URI. For example, performing a HTTP GET or POST against *http://localhost:8060/some-script.xqy* would invoke the some-script.xqy XQuery main module.

Unlike in eXist-db's REST Server, in MarkLogic's HTTP App Server, resources in the Modules or Content databases cannot be manipulated by HTTP POST, PUT or DELETE. Similarly there is no support for content negotiation by default. Like eXist-db, passing parameters to XQuery modules via HTTP must be achieved through URL query-string parameters or POST'ed form fields. Similarly, MarkLogic also provides a proprietary XQuery function module for accessing the context of the HTTP interaction, namely the functions, *xdmp:get-request-**, *xdmp:set-response-**, *xdmp:get-session-** and *xdmp:set-session-**.

When comparing the HTTP App Server against eXist-db's REST Server, as the default out-of-the-box experience, it is less RESTful in its approach, as it does not support URI addressing of document resources or manipulation by HTTP methods. However, it does not purport to being an advanced REST server, and more complex RESTful requirements may be addressed by additional mechanisms discussed below.

3.2.2. URL Rewriting

URL Rewriting may be setup independently for each HTTP App Server. When enabled, all HTTP requests to a specific HTTP App Server are filtered by executing the configured XQuery main module for each HTTP Request that is received. This

XQuery module is typically called '*url_rewrite.xqy*' and is responsible for rewriting URL's. The approach is much simpler than that of eXist-db described in Section 3.1.2, and the purpose of the script is to solely return a single String value which is the rewritten URI path.

Example 5. MarkLogic, URL Rewriting snippet (url_rewrite.xqy)

```
let $url := xdmp:get-request-url() return

    (: homepage :)
    if(fn:matches($url, "^/$") or fn:matches($url, "^/home.xml$")) then
        "/home.xqy"

    (: login page :)
    else if(fn:matches($url, "^/login$")) then
        "/login.xqy"

    (: user sign-up page :)
    else if(fn:matches($url, "^/register$")) then
        if(xdmp:get-request-method() eq "GET")then
            "/registration-form.xqy"
        else if(xdmp:get-request-method() eq "POST")then
            "/sign-up.xqy"
        else
            "/nowehere.html"
    else
        "/nowehere.html"
```

URL Rewriting adds greater flexibility to MarkLogic's HTTP App Server by decoupling the logical URI space from the actual modules database URI space. It is much simpler than the approach taken by eXist-db, with the disadvantage in functionality leading to the advantage of there being less non-declarative XQuery URL Rewriting code to maintain. However, it results in the same problem, which is that large applications will call for unwieldy and complex rewriting rules, creating a spaghetti of if/else statements, which will ultimately obscure the very URI's that are of such importance.

3.2.3. XQuery Libraries

A number of additional XQuery libraries such as the MarkLogic REST Library or Corona[24] are available for MarkLogic App Server. Each of these libraries may be installed as a URL Rewriter for a particular HTTP App Server, and in addition provide an XQuery module whose functions may be invoked from your own XQuery code to simplify handling of incoming HTTP REST requests.

These libraries substantially improve upon the standard URL Rewriting, by allowing you to have declarative XML files that define the URL mappings, and can make working with RESTful requests much simpler. However, they still have the disadvantages of, requiring additional XQuery glue code, and moving the declarative URL mappings away from the end-points of execution, which could lead to difficulty when identifying which URI rules apply to which code sites.

Example 6. MarkLogic, REST Library, declarative XML URL rewriting snippet

```
<options>
    <request uri="^/(.+)/act(\d+)$" endpoint="/endpoint.xqy">
        <uri-param name="play">$1.xml</uri-param>
        <uri-param name="act" as="integer">$2</uri-param>
    </request>
    <request uri="^/(.+)/?$" endpoint="/endpoint.xqy">
        <uri-param name="play">$1.xml</uri-param>
    </request>
</options>
```

3.3. EMC xDB

EMC xDB was chosen because 1) it has a different technical heritage, as it was originally designed as a product to be embedded within Java Applications, 2) it purports to be the most widely used XML Native Database (due to its embedded nature), 3) whilst it can operate as a standalone server, it is typical to have to construct your own REST API layer.

xDB provides a single mechanism for invoking XQuery in a RESTful manner over HTTP. In addition, there are also two proscribed self-build mechanisms from EMC publications which will also be briefly reviewed.

3.3.1. xDB REST API

With the release of version 10.1 of xDB, a Web Client was provided for the standalone server, which allows administration and management of the database from a Web Browser. This Web Client is underpinned by the xDB REST API[25]. Whilst the xDB REST API is mentioned briefly in the documentation, and there is limited documentation accompanying an xDB installation, it is understood that this API is not intended for production consumption. However, there are some aspects of the xDB REST API which make it interesting to consider here.

This xDB REST API is most similar to eXist-db's REST Server in that it allows any resource stored into the database to be addressed by a URI, and the database content to be manipulated by HTTP PUT and DELETE. An advantage of the xDB REST API when compared to the others, is that it does support some content negotiation (i.e. XML or JSON) for retrieving lists of what is present in the database, and

also metadata about the resources themselves. For example if one wanted to retrieve the XML document on Hamlet from the Classics database, a HTTP GET on *http://localhost:1280/federation/classics/shakespeare/hamlet.xml* may retrieve the desired document. However, for our focus on XQuery, a major disadvantage is that there are no XQuery function modules for accessing the HTTP context in xDB. It is also impossible to invoke the execution of XQuery main modules stored into the database, as calling these modules by URI simply results in their textual code content. XQuery invocation by xDB REST API is possible by:

1. Sending an XQuery to the REST API for execution against a database, library (a.k.a. collection) or document URI context.
   a. a) For HTTP GET a parameter may be appended to the URL query-string which contains an XQuery to execute. For example to retrieve speeches given in Hamlet – *http://localhost:1280/federation/classics/shakespeare/hamlet.xml/_xquery?query=//speech* or for example, to retrieve speeches given in any Shakespeare script – *http://localhost:1280/federation/classics/shakespeare/_xquery?query=//speech*.
   b. b) For more complex XQuerys, the XQuery may be wrapped in a CDATA section of a simple XML document and sent to the server URI by HTTP POST.

The REST API is a capable mechanism and, whilst not supporting the execution of pre-stored XQuery main modules, it does have in its favour support for content negotiation, and hyperlinking to child resources and libraries are implicit in the response (when requesting a database or library), thus supporting the REST promise of hypermedia for application state. However, when requesting resources themselves, there is no mechanism for hyperlinking to related resources, a disadvantage when compared to EMC's xDB XML REST Framework described in Section 3.3.3. URI's serviced by the REST API, implicitly mirror the database hierarchy in xDB, which is not always desirable. In addition parameters can only be passed to the XQuery module via XQuery external variables, which may be bound in the XML document, when using the HTTP POST approach. This coupled with the lack of XQuery functions for accessing the HTTP context, makes building applications in pure XQuery impossible, and building applications with simple, logical and descriptive URI schemes a difficult challenge.

3.3.2. Implementing a RESTful API (JAX-RS)

Published by EMC is an approach designed by Martin Probst[10] which couples together xDB standalone server with JAX-RS (Java API for RESTful Web Services) for the purposes of database management and XQuery execution. The article describes implementing an example REST API for use with xDB, and allows anyone to build upon this. In fact the API described in the article is almost certainly the exact API discussed in Section 3.3.1. However, it is not the similarity that we are

interested in, but rather the idea that JAX-RS, which is a REST API framework, is itself used to produce a domain specific REST API with relevance for XQuery. The ability to use JAX-RS in this way is conceptually no different to using the URL Rewriting support in eXist-db or MarkLogic for writing a domain specific REST API.

The reason for looking at JAX-RS here is that, like URL rewriting in eXist-db and MarkLogic, or WebApp Descriptors in Servlex, JAX-RS provides a declarative mechanism for defining the URL rewriting/mapping rules. However, JAX-RS has a major advantage over the other approaches, which enables the declarative URL rewriting rules to live alongside the code sites of execution. This is achieved through annotations. These annotations should allow easy determination of code and URI relationships due to their proximity to the executable code sites, whilst the declarative nature should make reasoning about the mappings simple.

Example 7. xDB, JAX-RS API Snippet

```
@GET
@Path("_query")
public Response doXQuery(@QueryParam("xquery") String query, @Context ▶
HttpServletRequest context) {

    //get the xquery sent to us
    if (query == null)
      throw new ▶
WebApplicationException(Response.status(Status.BAD_REQUEST).entity(
          "xquery parameter is required").build());

    //execute the query
    String results = executeXQuery(query, context.getParameterMap());

    //return the results in the http response
    String contentType = xquery.getOptions().get(new QName(XHIVE_NS, ▶
"content-type"));
    return Response.ok(results, contentType).build();
}
```

The JAX-RS Java annotations `@GET` and `@Path("_query")` imply that, should the HTTP application server encounter a HTTP GET for the path '*_query*', then the code in the function '*doXQuery(...)*' should be executed. In JAX-RS paths are always defined relative to the server's HTTP URI context. The `@QueryParam("xquery")` annotation injects the value of the HTTP URL query-string parameter named '*xquery*' into the String function parameter called '*query*'.

3.3.3. XML REST Framework

The XML REST Framework[20], an approach advocated by Cornelia Davis at EMC, advances the JAX-RS approach described in Section 3.3.2, to produce a framework for building domain specific REST API's for xDB. Resultant APIs do not permit the direct execution of XQuery by HTTP, rather the framework indirectly executes developer defined XQuery code for each REST API function invoked. Each REST endpoint is coded as a POJO (Plain Old Java Object) in Java with JAX-RS annotations, and it is this which mediates the execution of the XQuery and the marshalling and de-marshalling of the XQuery external variables and output over HTTP for the client.

Arguably, the most interesting aspect of this framework and its approach is that it recognises that whilst REST APIs are good at delivering representations or resources, they often lack hyperlinking in the returned resource content to enable further resource URI's to be autonomously determined.

The initial technical implementation detail of the XML REST Framework[26], describes the situation thusly, "(as is sadly common in many RESTful services today) the consumer has no choice but to leverage the knowledge of these URI templates". In further implementation[27], the XML REST Framework is extended to allow XSLT to be applied to content before it is returned as the result of a REST response to a request; XSLT can be used to modify XML content responses and insert hyperlinks to other related resource representations. Whilst XSLT is used in this instance, this could easily be substituted for XQuery. The XML REST Framework has since been advanced, so that the REST end-points in Java now call an XProc pipeline rather than the XQuery directly. The advantage of this is that multiple processes can be placed inside the XProc pipeline, including XQuery. The disadvantage is the XQuery is moved further away from the HTTP context and this could make understanding the code used to fulfil the request for a specific URI more convoluted.

The XML REST Framework for xDB stands alone from all other reviewed vendor approaches to REST in that it attempts to address the hyper-linking tenant of the REST architecture. The approach to embedding hyperlinks with XSLT is however only applicable when hypermedia instances such as XML or XHTML are used for application state.

3.4. Servlex

Servlex was chosen because 1) it is an implementation of a public common standard (the EXPath Webapp Module) for wiring HTTP Requests to XML processors, 2) unlike others, it does not require a Document Repository or XML Database, and 3) it provides a purely declarative approach to URI mapping, instead placing constraints on the interface with the underlying XML processing code. Servlex is the reference implementation of the EXPath Webapp Module.

Servlex provides a single mechanism for wiring HTTP Requests to XML processors, this mechanism is the declarative vocabulary of the Webapp descriptor, an XML file named '*expath-web.xml*'. This descriptor must be placed inside a larger EXPath Package[28]. Whilst Servlex supports mapping URI's to XPath, XQuery, XSLT and XProc code, inline with the focus of this paper, we will only consider herein Servlex's ability to interoperate with XQuery.

Example 8. Servlex, Webapp Descriptor snippet (expath-web.xml)

```
<webapp xmlns="http://expath.org/ns/webapp/descriptor"
    xmlns:app="http://example.org/ns/my-website"
    name="http://example.org/my-website"
    abbrev="myweb"
    version="1.3.0">
    <title>My example website</title>

    <resource pattern="/style/.+\.css" media-type="text/css"/>
    <resource pattern="/images/.+\.png" media-type="image/png"/>

    <servlet>
        <xquery function="app:product-page"/>
        <url pattern="/product/(.+)">
            <match group="1" name="id"/>
        </url>
    </servlet>
</webapp>
```

The example descriptor would map the URL starting with '*/product/*' to the function '*app:product-page*' in the XQuery module of the namespace '*http://example.org/ns/my-website*', which would need to be pre-defined in the EXPath Package descriptor '*expath-pkg.xml*'. Servlex places constraints on its interface with the XQuery processor[29], so an example module showing a possible function is illustrated below.

Example 9. Servlex, XQuery Module (products.xqy)

```
module namespace app = "http://example.org/ns/my-website";

declare namespace web="http://expath.org/ns/webapp";

declare function app:product-page($request as element(web:request), $bodies ▶
as item()*) as item()* {
    (
        <web:response status="200" message="Ok"/>
        ,
        <debug>Chosen Product ID was:
            <id>{/web:path/web:match[@name eq 'id']/text()}</id>
```

```
            </debug>
        )
    };
```

The declarative nature of the URL mapping in Servlex is an advantage over the eXist-db and MarkLogic mechanisms, as it allows the URIs to remain outside of the code and easily visible and maintainable; however, it is a disadvantage when compared with the JAX-RS approach recommended for use with xDB in Section 3.3.2, as the declarative URI patterns are moved away from the code execution sites. Another disadvantage of the Servlex approach is that it enforces an interface which must be present in XQuery function signatures which are URI mapped by Servlex. This can lead to the creation of adapter functions and glue code to act as a facade for Servlex to interface with the desired XQuery module functions to be invoked. Apart from URI mapping, for identification of resources, Servlex leaves all other REST requirements to the implementer of the XQuery processing code, which allows great flexibility at the expense of the time required for implementation of a Servlex Servlet.

3.5. Summary

Whilst all of the reviewed vendors' products differ in their approaches and enabling facilities for developers to build RESTful applications in XQuery, each offers at least one mechanism for XQuery to be executed by HTTP.

Table 1. Summary of compliance with RESTful interface constraints

		Identification of Resources	**Manipulation, Representations**	**Self-Descriptive Messages**	**Hyper-media for Application State**
eXist-db	REST Server Section 3.1.1	Direct	GET/POST/PUT/DELETE XQuery representations	Yes	Browsing XQuery developer option
	XQuery URL Rewriting Section 3.1.2	Direct and Indirect	GET/POST/PUT/DELETE XQuery representations Content negotiation	Yes	XQuery developer option

		Identification of Resources	**Manipulation, Representations**	**Self-Descriptive Messages**	**Hyper-media for Application State**
MarkLogic	HTTP App Server Section 3.2.1	Indirect - Only XQuerys	GET XQuery representations	Yes	XQuery developer option
	URL Rewriting Section 3.2.2	Indirect	GET XQuery representations	Yes	XQuery developer option
	XQuery Libraries Section 3.2.3	Indirect	GET/POST/PUT/DELETE XQuery representations Content negotiation	Yes	XQuery developer option
EMC xDB	xDB REST API Section 3.3.1	Direct	GET/POST/PUT/DELETE XQuery representations Content negotiation	Yes	Browsing XQuery developer option
	JAX-RS Section 3.3.2	Indirect	GET/POST/PUT/DELETE Java programmed representations Content negotiation	Yes	Java developer option
	XML REST Framework Section 3.3.3	Indirect	GET/POST/PUT/DELETE Indirect XQuery representations Content negotiation	Yes	Yes Injected hyperlinking via. XSLT
EXPath	Servlex Section 3.4	Indirect Only XQuery/XSLT/XProc	GET/POST/PUT/DELETE Indirect XQuery/XSLT/XProc representations Content negotiation	Yes	XQuery/XSLT/XProc developer option

eXist-db's REST Server and EMC's xDB REST API provide the most RESTful out-of-the-box experience without the need to write additional code because they permit manipulation of the database (including XQuery main modules) by the use of HTTP verbs. In addition, when browsing resources hyper-media state with hyper-linking is automatically delivered. These features meet the RESTful architectural requirements of: Identification of Resources, Manipulation of Manifestations, Self-Descriptive Messages and some limited support for Hyper-Media for Application State. eXist-db's REST Server has the slight advantage for building applications, because 1) XQuery can be pre-loaded into the database through the HTTP verbs PUT or

POST and then invoked with HTTP GET, and 2) XQuery function modules are provided to support the HTTP context.

However, if we compare all options reviewed available for building XQuery RESTful Web Applications the conclusion is different. We will dismiss EMC xDB because without also writing Java alongside XQuery, its REST API is too limited. There is no access to the HTTP context from XQuery, and there is no support for URI Rewriting to decouple the application URI space from the physical database layout.

Both the URL Rewriting capabilities of eXist-db and MarkLogic are impressive, however by default they both use an XQuery main module to do the URL rewriting, which as previously discussed can lead to maintainability issues. MarkLogic probably has the advantage that there are several XQuery Libraries that extend their URL rewriting mechanism to both allow URL mappings to be declaratively defined, whilst also simplifying many common HTTP/REST functions required by developers. Servlex, like MarkLogic's XQuery Libraries, provides a declarative approach to URL rewriting, however unlike MarkLogic and eXist-db the mechanisms defined for accessing the HTTP context from the processors is still embryonic and not widely tested. There is a problem with the declarative URL rewriting approach in MarkLogic and Servlex in that the URL rewrite rules are moved away from the code sites, which in complex applications can make the mapping between functions and URI's difficult to maintain. Conceptually this is solved in JAX-RS as used in xDB as the URI's are still declarative yet precede the code site.

Section 4 proposes a set of Annotations for XQuery 3.0 which enable the best features found in the reviewed products above, but that maintain both the advantages of declarative URI Rewriting and keeping URI rules close to the code site.

4. Standardised XQuery 3.0 Annotations for REST

Herein we present a set of XQuery 3.0 Annotations to enable the construction of RESTful Web Services in XQuery. The goals of our approach are:

1. *Interoperability*. It is envisaged that this paper will provide the basis for a public XQuery community standard, which if adopted by vendors, would permit portable XQuery Web Services. To enable this, an implementation agnostic description is provided.
2. *Simplicity* for XQuery developers. Developers should not have to maintain external or complex code for wiring RESTful services to XQuery functions.
3. *Technical improvement*. Having reviewed existing approaches in Section 3, we build upon the best aspects of each vendor's approach.

4.1. Approach

For mapping RESTful requests to executable code, the declarative approach discussed in Section 3, is felt to be the most advantageous, particularly when the declaration is an annotation on the code site (Section 3.3.2). Therefore, our approach is heavily influenced by that of JAX-RS[30]. However, we simplify and deviate from JAX-RS due to the language structure differences between Java and XQuery. Where JAX-RS describes Resource Classes and Resources Methods for Java, in XQuery we simply use the term Resource Function; for mapping HTTP calls to XQuery invocation, our unit of granularity is the XQuery function. Through the use of annotations on functions in XQuery, we declaratively mark-up the HTTP capabilities of a function.

To minimise refactoring by developers when adding annotations to existing code, two measures must be respected by implementations:

1. Implementations must support annotated functions which have additional function parameters which are not annotation mapped, providing the cardinality type of those un-mapped parameters accepts an empty sequence.
2. Implementations must not enforce the order of function parameters. Whether mapped by annotations or not is unimportant, as annotations explicitly name the parameters to which they are mapped.

For the purposes of this paper, HTTP Multipart Requests and Responses are considered out of scope. However, some attention has been paid to not preventing support for these in future, and this is briefly discussed in Section 6.2

4.2. Resource Functions

A Resource Function is an XQuery function which has been marked up with RESTful web service annotations. These annotations indicate to a processor that when presented with a RESTful web service request, that matches the constraints indicated by the annotations, the function should be invoked and the result returned as the result of the service request.

Whilst the concept of dynamically and transparently mapping web service calls to XQuery function invocation has previously been proved[23], this is the first time XQuery annotations have been used to provide a standardised and developer controllable approach.

4.3. Resource Function Constraints

Constraints restrict the service requests that a Resource Function may process.

4.3.1. URI Path and Templates

A '*Path Annotation*' provides for URI templates and allows the URI of a RESTful web service to be mapped to a Resource Function. A Resource Function must contain a single path annotation. Additional annotations may also be used to constrain the Resource Function.

The path annotation is named '`%rest:path`' and takes a single mandatory literal string, which describes the URI path for this service. The URI path is relative to a base URI defined by the implementation.

The URI path itself may contain zero or more URI templates which denote path segments that map to named function parameters. A URI template, has the syntax '*{$fn-param-name}*', where '*fn-param-name*', is the name of a parameter to the annotated function, whose value should be taken from the path. Parameters addressed by URI templates, must meet the following constraints:

1. Cardinality that allows for an atomic value, otherwise an error should be raised by the implementation.
2. Type that inherits from xs:anyAtomicType, otherwise, an error should be raised by the implementation. In addition, conversion from the URI segment string to the required type should be performed at run-time, and an error raised if conversion is impossible.

Example 10. XQuery Path Annotation

```
declare
    %rest:path("/stock/widget/{$id}")
function local:widget($id as xs:int) {

    (: get the widget :)
    fn:collection("/db/widgets")/widget[@id eq $id]
};
```

For example, a HTTP GET on the following URI, would cause the Widget with the '*id*' of '*1981*' to be retrieved: *http://www.widget-factory.com/stock/widget/{$id}*.

When many URI paths are defined, conflicts may occur. It is implementation defined how these should be resolved. However, most specific URI paths must always be evaluated before less specific URI paths, to ensure that lesser paths do not unintentionally consume requests.

4.3.2. HTTP Methods

Resource Functions may be constrained to zero or more HTTP methods by means of a method annotation. Unless otherwise constrained by a method annotation, the path annotation of a Resource Function applies to all HTTP methods.

Annotations are defined for all HTTP 1.1 methods except TRACE and CONNECT. All methods may return resources except for HEAD, which must only return a *rest:response* element.

Example 11. XQuery Method Annotation

```
declare
    %rest:GET
    %rest:POST
    %rest:path("/widget/{$id}")
function local:widget($id as xs:int) {

    (: get the widget :)
    fn:collection("/db/widgets")/widget[@id eq $id]
};
```

Method annotations POST and PUT may take an optional string literal which map the HTTP request body to a named function parameter. The same syntax as that used for URI templates is applied, for example `%rest:POST("{$request-body")`, would inject the request body into the function through the function parameter named '*request-body*'. The function parameter for the request body must meet the following constraints:

1. Cardinality that allows for one or more of the typed item(s).
2. Typing that is compatible with the request body. The type of the request body is determined by the HTTP Content Type header and may be constrained by means of the `%rest:consumes` annotation (see Section 4.3.3). The interpretation of the request body is similar to that of the EXPath HTTP Client[14]:
   a. If the media-type is a text media-type, the function parameter type will be xs:string.
   b. If the media-type is an XML media-type, the request body is parsed as XML and the function parameter type will be document-node().
   c. If the media-type is a HTML media-type, the content is tidied-up and parsed as XML. The parameter type will be document-node(). The tidying process is implementation defined as no known standard exists.
   d. Otherwise, a binary media type is assumed, and the function parameter type will be xs:base64Binary.

4.3.3. Media-Type Capabilities

Support for content negotiation is indirectly provided by two annotations:

1. `%rest:consumes`, which constrains a Resource Function, by only accepting requests for which one of the defined Internet media-types matches the HTTP Content-Type header of the request.
2. `%rest:produces`, which constrains a Resource Function, by only accepting requests for which the mime-type matches the HTTP Accept Header.

Both annotations take a single mandatory String Literal which contains an Internet media-type.

Example 12. XQuery Consumes and Produces Annotations

```
declare
    %rest:PUT("{$body}")
    %rest:path("/widget")
    %rest:consumes("application/xml", "application/atom+xml")
    %rest:produces("application/xml")
function local:widget($body as document-node(element(widget)) {

    (: store the widget :)
    let $db-uri := xmldb:store("/db/widgets", (), $body),

    (: return a hyper-link :)
    $rest-uri := rest:get-absolute-uri("wiget", $body/widget/@id) return
       <a xmlns="http://www.w3.org/1999/xhtml" href="{$rest-uri}">{$rest-uri}</▶
a>
};
```

4.4. Resource Function Parameters

Parameters to Resource Functions are extracted from the RESTful Web Service request and passed in as additional function parameters. Unlike constraints, parameters are always optional. Resource Function Parameters use the same URI template syntax as described in Section 4.3.1 to map the parameter onto a function parameter. They may also provide a default value should the parameter not be present in the request. Resource Function Parameters always place the following constraints on the function parameters that they map to:

1. Cardinality that allows for: zero or many atomic values in the case of Query, Form or Header parameters, or zero or one atomic values in the case of Cookie parameters, otherwise an error should be raised by the implementation.
2. Type that inherits from xs:anyAtomicType, otherwise, an error should be raised by the implementation. In addition, conversion from the parameter string to the required type should be performed at run-time, and an error raised if conversion is impossible.

4.4.1. Query String Parameters

The annotation *rest:query-param* is provided for accessing parameters in the Query String of the URL used for the RESTful Web Service request.

Example 13. Query String Parameter Annotation with a default value

```
declare
    %rest:GET
    %rest:path("/widget/{$id}")
    %rest:query-param("client", "{$client}", "unknown")
function local:widget($id as xs:int, $client as xs:string*) {

    (: get the widget :)
    fn:collection("/db/widgets")/widget[@id eq $id][@client = $client]
};
```

4.4.2. Form Field Parameters

The annotation *rest:form-param* is provided for accessing parameters from a HTML Form submitted in the request body of the RESTful Web Service request with the Internet media-type '*application/x-www-form-urlencoded*'.

Example 14. Form Field Parameter Annotation

```
declare
    %rest:GET
    %rest:path("/widget/{$id}")
    %rest:form-param("client", "{$client}")
function local:widget($id as xs:int, $client as xs:string*) {

    (: get the widget :)
    fn:collection("/db/widgets")/widget[@id eq $id][@client = $client]
};
```

4.4.3. HTTP Header Parameters

The annotation *rest:header-param* is provided for accessing HTTP Request headers from the RESTful Web Service request. If a single Header field value contains comma separated values, an implementation must extract each value from the comma separated list into an item in the sequence provided to the function parameter.

Example 15. HTTP Header Parameter Annotation

```
declare
    %rest:GET
    %rest:path("/widget/{$id}")
```

```
    %rest:header-param("X-Client-Type", "{$client-type}")
function local:widget($id as xs:int, $client-type as xs:string*) {

    (: get the widget :)
   fn:collection("/db/widgets")/widget[@id eq $id][@clientType = $client-type]
};
```

4.4.4. Cookie Parameters

The annotation *rest:cookie-param* is provided for accessing HTTP Cookies from the RESTful Web Service request.

Example 16. Cookie Parameter Annotation

```
declare
    %rest:GET
    %rest:path("/widget/{$id}")
    %rest:cookie-param("locale", "{$locale}")
function local:widget($id as xs:int, $locale as xs:string?) {

    (: get the widget :)
    fn:collection("/db/widgets")/widget[@id eq $id][@locale = $locale]
};
```

4.5. Resource Function Serialization

Herein we detail how the results of a Resource Function may be serialized back to an HTTP response for the RESTful web service response.

A Resource Function may return one of three response types:

1. A Resource, i.e. just content.
2. HTTP headers, for example acknowledging the request or providing a status code or additional information.
3. Both HTTP Headers and a Resource.

When returning Resources from Resource Functions, we need to consider how the Resource should be serialized for use in the RESTful web service response. Rather than define a serialization mechanism, XQuery 3.0 in §2.24 of the specification already specifies how serialization of an XDM[31] (i.e. the result of an XQuery) may be controlled by use of Output Declarations. However, Output Declarations are only applicable to XQuery Main Modules, so when re-purposing Output Declarations for serialization the following rules should be applied:

1. If the function is from within a Main Module, and an output declaration exists, then we use this as the default serialization settings for each Resource Function in that module.

2. Output Declarations may be re-written as annotations on any Resource Function e.g. `%output:method("xml")`. These annotation output declarations override any defaults from (1). This is also applicable if our function comes from a Library Module, of which the XQuery specification forbids Output Declarations.
3. If no Output Declaration, annotated or otherwise, is provided, then the default is to serialize as XML, UTF-8 encoding, with indenting.

Each of the three possible result types of a Resource Function needs to be handled in a different manner by an implementation, and as such we provide appropriate function signature restrictions, and detail how annotations interact with these:

1. For a function that returns just a resource, either:
   a. If the result type is omitted from the function, it is assumed to be *document-type(element())* or just *element()*, and XML Serialization should be applied to the result of the function. The annotation `%output:method`, if present, must be set to '*xml*'.
   b. If the result type is present, it must be a type which is compatible with the chosen serialization method, defined by either the XQuery 3.0 output declaration or overridden by the `%output:method` annotation on the Resource Function. The default serialization method is XML. If the result type is not compatible with the serialization, an implementation must throw an error.
2. For a function that returns just HTTP headers:
   The result type of the function must be defined as *document-type(element(rest:response))*. Any other annotations that effect the serialization of the result are ignored.
3. For a function that returns both HTTP headers and a resource:
   The result type of the function must be defined as *item()+*. The first item in the result sequence is the HTTP headers i.e. *document-type(element(rest:response)*, the second item in the result sequence is the resource itself. The rules of both (1) and (2) must be applied to the result sequence.

4.5.1. REST Response Format

As described above, a REST Response document may be returned from a function either with or without a Resource. The purpose of this document is to control the REST (in this case HTTP) response sent back to the client of the RESTful web service.

Example 17. REST Response Format

```
<rest:response>
    (http:response?)
</rest:response>

<http:response status?="integer" reason?="string">
```

```
    (http:header*)
</http:response>

<http:header name="string" value="string"/>
```

Should the status be omitted for the response, or a REST Response document not returned from a Resource Function, then the status defaults to 200 OK. It is expected that implementations will make use of sane defaults for HTTP headers as part of their HTTP responses, however any default headers must be overridable by those values set in the REST Response document.

4.6. REST Function Module

RESTful annotations are designed to be hosted by an implementation that provides a Web Server end-point. As the URI Paths of Resource Functions are defined relative to an implementation defined base URI (see Section 4.3.1), it is impossible without additional support to return hyper-links to the client for Resources. A simple module which provides just two XQuery extension functions is proposed to enable the resolution of absolute URIs, such as the use-case shown in Example 12.

```
rest:get-base-uri() as xs:anyURI
```

Returns the base URI of the web context in which the invoking Resource Function is executing. The result of this function should be stable across invocations within an implementation.

```
rest:get-absolute-uri($path-segments as xs:anyAtomicType+) as xs:anyURI
```

Returns an absolute URI by concatenating the base URI as returned by `rest:get-base-uri()` with each path segment in the parameter *$path-segments*, separating each by a '/' character. The result of this function should be stable across invocations within an implementation.

5. Proof of Concept

As a proof of concept, the approach described in Section 5 has been implemented in the eXist-db Open Source Native XML Database project by the author of this paper. eXist-db is written in the Java programming language, with an XQuery parser written in ANTLR. Both Java and ANTLR were used to implement the proof of concept.

5.1. Implementation

There were several steps to the implementation:

1. Adding support for XQuery 3.0 Annotations to eXist-db's XQuery parser. eXist-db supports XQuery 1.0 with a high-level of compliance, but recently has started a staged adoption of XQuery 3.0 support; annotations were previously missing.
2. Adding support for Multiple-Triggers per-database Collection. It was decided to develop the majority of the RESTful Annotations implementation outside of the core product code. eXist-db provides for database triggers, to enable developers to execute arbitrary code upon various events. Previously, eXist-db only supported a single trigger per database Collection. Rather than modify the XQuery parser further, a database Trigger was used to support this implementation.
3. Implementation of support for the RESTful, the RESTful web services that they declare and the supporting XQuery functions extension module.

The implementation is made up of two distinct processes:

1. When a user stores an XQuery Module into the database, a Trigger is fired. This trigger initiates several steps in order:
   a. Compilation of the XQuery Module into an AST (Abstract Syntax Tree)
   b. Additional static analysis upon any User Defined Functions that have RESTful annotations. These are known as Resource Functions.
   c. Compilation of a Regular Expression for the Path Annotation of each Resource Function. This expression enables both path matching and extraction of value for URI templates.
   d. Creation of a RESTful Service Object for each Resource Function, which contains the Path Regular Expression, any additional constraint annotations, any optional parameter annotations, a reference back to the database location of the XQuery Module, and the name and arity of the Resource Function (User Defined Function).
   e. Registration of the RESTful Service Object with the RESTful Service Registry for each HTTP Method that it supports.
2. When a HTTP Web Request is made to the system, the receiving Servlet initiates the several steps in order:
   a. Querying the RESTful Service Registry for any Resource Functions which can service the incoming Request.
   b. The Service Registry examines the RESTful Service Objects registered with it to determine if they are applicable to the incoming request. If so, the RESTful Service Object is returned.
   c. A new instance of the XQuery Module referenced by the RESTful Service Object is compiled, and function parameters (as defined by the RESTful annotations) are extracted from the request and mapped into a function call to the underlying User Defined Function.

d. The User Defined Function is executed. The result is serialized back to the HTTP client based on the rules defined in $4.3.

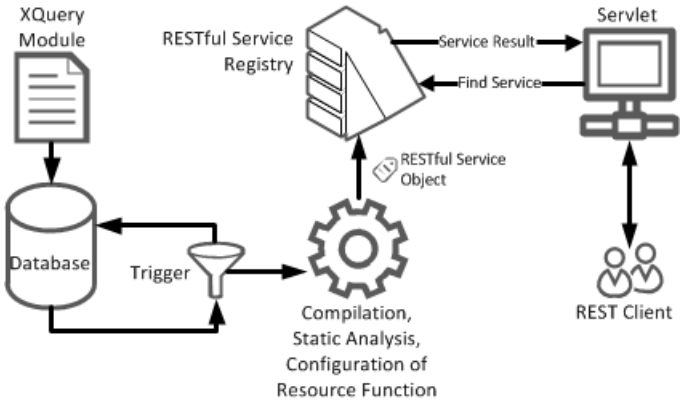

Figure 1. Implemented Architecture for RESTful XQuery Annotations

5.2. Evaluation

The modifications to eXist-db to support the implementation of RESTful annotations took three working days. Implementation of the RESTful annotations and web server itself took another three days. This figure does not include the Resource Function Parameters defined in Section 4.4, which will be implemented in the near future.

The implementation certainly feels cleaner than that of the current REST Server and URL Rewrite in eXist-db, and in use is much simpler as a potentially complex controller.xql does not need to be authored or maintained. The new implementation permits for complete declarative decoupling of the URI space from the Resources themselves, without having to declare the URI space externally of the code sites themselves.

In addition the RESTful XQuery Annotations approach, due to its JAX-RS heritage, should feel very familiar to those with an existing knowledge of Java programming, hopefully easing any transition.

A port of the XML Summer School's[32] 'See What I Think' project[33] to the new RESTful Annotations was undertaken. The project is a learning tool originally written in XQuery atop eXist-db's REST Server and URL Rewriting framework. The port was undertaken to understand how the use of Annotations compares to the previous approach. The resultant port uses slightly more lines of XQuery code than the original project, however it is subjectively easier to understand the wiring of URIs to XQuery functions as the URIs are declarative. The port did not attempt to re-structure the original code layout which was written around eXist-db's URL Rewriting Framework. Arguably if the code was further restructured instead around

the concept of Resource Functions, then the lines of code could be reduced and the code-base may become more modular.

5.3. Further Work

Certainly the majority of the code written for the eXist-db implementation could trivially be made useable for other implementations in Java, by removing any direct eXist-db dependencies and replacing them with implementation agnostic interfaces. However, other implementations would need to support something akin to Triggers or consider modifying their XQuery 3.0 parser with rules for XQuery RESTful Annotations.

In addition, as the majority of XQuery implementations appear to be written in either Java or C++, should the majority of the code be made implementation independent, it would be interesting to perform a port to C++.

6. Conclusion

In this work we tackled the problem of using XQuery as a Server Side processing language for the Web whilst maintaining vendor independence and portability of XQuery code. The presented approach, driven by the review of current vendors' products, was to propose an implementation agnostic set of RESTful Annotations for XQuery 3.0. The goals of the approach developed in this paper were Interoperability, Simplicity and Technical Improvement:

- *Interoperability* has been addressed by proposing a set of Annotations which do not require any particular product or programming language for implementation. Whilst these Annotations permit the development of RESTful services, they do not constrain implementors choice of platform or technology.
- *Simplicity* has been addressed by developing an approach that is not disruptive. XQuery developers can continue to write XQuery in the same method that they always have. Should they require to provide RESTful Web Services, they can simply add the RESTful annotations to their functions, enabling them as Resource Functions.
- *Technical Improvement* has been achieved by understanding the strengths and limitations of the approaches taken in current vendors' products and building upon these. The strength of using URI Rewriting, in a declarative mechanism from XQuery frameworks has been enhanced by removing the framework aspect and ensuring that the declarations of intention appear alongside the code target function.

Additionally, we justify how our approach meets the four interface constraints of REST:

1. *Identification of Resources*

URIs are used to identify resources (or representations) and are mapped (or partially mapped) to XQuery functions for delivery. This is achieved through the use of Path Annotations and URI Templates as described in Section 4.3.1. XQuery Functions themselves enable further mechanisms for the addressing of XML documents or other resources through fn:doc() and fn:collection(), and/or XML nodes through XPath.

2. *Manipulation of Resources Through Representations*

   Resource Functions (RESTful Annotated XQuery Functions) permit the generation of representations of resources through XQuery processing, and serialization of the the representation through repurposing the XDM Serialization rules as described in Section 4.5. In addition content negotiation may be enabled by the mechanisms described in Section 4.3.3 to enable a server to deliver different serializations (e.g. XML, XHTML, Text, JSON etc) of a resource depending on the constraints of a request.

   Manipulation of resources in typical RESTful HTTP service requests is by the use of both URI addressing (1) and HTTP methods. Many XQuery implementations have vendor extensions for manipulating document stores and the XQuery Update specification provides for document modifications. The mechanism for mapping HTTP methods to XQuery functions, which may utilise vendor extensions and/or XQuery Update is described in Section 4.3.2.

3. *Self-Descriptive Messages*

   The use of HTTP for RESTful services with support for all relevant HTTP methods provides for the constraint of Self-Descriptive messages. If the message content itself is XML or (X)HTML the message content in itself is also hopefully self-descriptive. Indeed, our approach attempts to be lightweight and not constrain developers and, as such, they have complete control over the message content.

4. *Hypermedia as the Engine of Application State*

   The result of a Resource Function should permit a developer to provide as much hyper-linking of the application state as they desire. Whilst attempts for specific systems have described mechanisms such as XSLT for embedding hyperlinks in resultant content[20], the approach developed in this paper is rather to provide the generic constructs to enable this, without prescribing an approach to developers.

6.1. Limitations

Currently the approach developed by this paper does not provide support for either HTTP Matrix Parameters or HTTP Multipart requests or responses.

In addition, there is currently no mechanism for the default handling of URI's that do not meet a declared URI Path, for example, customised HTTP 404 Page Not

Found responses. However, this could perhaps be considered out of scope for such a project.

6.2. Future Work

Future work would include technical support for both HTTP Matrix Parameters and HTTP Multipart requests and responses. Certainly it is envisaged that HTTP Multipart responses from Resource Functions could be achieved by returning a sequence of functions as the result, where each function is responsible for generating both the REST Response Headers and content for each part of a multipart response.

We believe that our declarative approach based on Annotations makes maintainability of RESTful web applications easier. However, in large applications there could still be some difficulty involved in developers understanding the bindings between Resource Functions and Web Services. As such, additional functions could be added to REST Function module (see Section 4.6), to enable the lookup of functions based on Request parameters.

Whilst we have called our approach '*standardised*', it is surely more independent that standardised. Rather we hope that it will form the basis of a publicly available and agreed standard. It is, of course, recognised that further work would be required to create a thorough technical standard based on this work, and communities such as EXQuery or the W3C Community Groups might be an appropriate vehicle for such work.

The Annotations that we have developed, once standardised could also be applied to other XML processing languages such as XSLT and XProc. Whilst XSLT and XProc do not directly have the concept of Annotations, they certainly support mechanisms that would yield a similar implementation and result.

Bibliography

[1] XQuery 1.0: An XML Query Language (Second Edition)http://www.w3.org/TR/xquery/[Accessed: 19-Nov-2011].

[2] XQuery Update Facility 1.0http://www.w3.org/TR/xquery-update-10/[Accessed: 19-Nov-2011].

[3] XQuery and XPath Full Text 1.0http://www.w3.org/TR/xpath-full-text-10/[Accessed: 19-Nov-2011].

[4] SourceForge.net Repository - [exist] Contents of /trunk/eXist-1.0/src/org/exist/http/HttpServerConnection.javahttp://exist.svn.sourceforge.net/viewvc/exist/trunk/eXist-1.0/src/org/exist/http/HttpServerConnection.java?limit_changes=0&revision=133&view=markup&pathrev=133[Accessed: 20-Nov-2011].

[5] SourceForge.net Repository - [exist] Contents of /trunk/eXist-1.0/src/org/exist/xpath/functions/request/RequestParameter.javahttp://exist.svn.sourceforge.net/viewvc/exist/trunk/eXist-1.0/src/org/exist/xpath/functions/request/RequestParameter.java?revision=158&view=markup&pathrev=158[Accessed: 20-Nov-2011].

[6] eXist-db Developer's Guide - Calling Stored XQueries eXist-db Developer's Guidehttp://www.exist-db.org/devguide_rest.html#d1915e781[Accessed: 14-Jan-2012].

[7] 28msec RESTful Conventions 28msechttp://www.28msec.com/documentation/sausalitobasics-restfulservices[Accessed: 14-Jan-2012].

[8] BaseX REST - BaseX Documentation BaseX Wikihttp://docs.basex.org/wiki/REST[Accessed: 14-Jan-2012].

[9] MarkLogic, Application Programming in XQuery and XSLT - HTTP App Server Functions MarkLogic Server Documentation.http://docs.marklogic.com/5.0doc/docapp.xqy#display.xqy?fname=http://pubs/5.0doc/xml/xquery/programming.xml%2353595[Accessed: 14-Jan-2012].

[10] Martin Probst EMC EMC Community Network - ECN: Implementing a RESTful API for xDB using Jersey/JAX-RS EMC Developer Network.https://community.emc.com/docs/DOC-4276[Accessed: 14-Jan-2012].

[11] XQuery Scripting Extension 1.0http://www.w3.org/TR/xquery-sx-10/[Accessed: 19-Nov-2011].

[12] EXPath - Standards for Portable XPath Extensionshttp://www.expath.org/[Accessed: 19-Nov-2011].

[13] EXQuery - Standards for Portable XQuery Applicationshttp://www.exquery.org/[Accessed: 19-Nov-2011].

[14] EXPath - HTTP Clienthttp://www.expath.org/modules/http-client/[Accessed: 14-Jan-2012].

[15] XQuery 3.0: An XML Query Languagehttp://www.w3.org/TR/xquery-30/[Accessed: 19-Nov-2011].

[16] XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)http://www.w3.org/TR/xpath-datamodel/[Accessed: 14-Jan-2012].

[17] S Kepser A Proof of the Turing-completeness of XSLT and XQuery May 2002.

[18] M Kaufmann D Kossmann Developing an Enterprise Web Application in XQuery Web Engineering, vol. 5648/2009, pp. 465–468, 2009.

[19] Roy Fielding Architectural Styles and the Design of Network-based Software Architectures University of California, Irvine, 2000.

[20] Cornelia Davis Programming Application Logic for RESTful Services Using XML Technologies in Proceedings of Balisage: The Markup Conference 2011, Montreal, Canada, 2011, vol. 7.

[21] Liam Quin XML Query Implementations 15-Jan-2012http://www.w3.org/XML/Query/#implementations[Accessed: 15-Jan-2012].

[22] Florent Georges EXPath - Servlet fgeorges.org Wiki, 15-Jan-2012http://fgeorges.org/wiki/EXPath#Servlet_definition[Accessed: 15-Jan-2012].

[23] Adam Retter SOAPServer - SourceForge.net Repository - [exist] Revision 3626http://exist.svn.sourceforge.net/viewvc/exist?view=revision&revision=3626[Accessed: 15-Jan-2012].

[24] Corona - Installation Corona - GitHub, 15-Jan-2012https://github.com/marklogic/Corona/wiki/Installation[Accessed: 15-Jan-2012].

[25] EMC xDB 10.1.0 manual - Web clienthttp://developer.emc.com/docs/documentum/xdb/manual/index.html#doc:topic/web_client.html[Accessed: 18-Jan-2012].

[26] Cornelia Davis EMC EMC Community Network - ECN: Building Domain Specific RESTful Services over xDB with the EMC XML REST Framework (Version 1)https://community.emc.com/docs/DOC-6434[Accessed: 18-Jan-2012].

[27] Cornelia Davis EMC EMC Community Network - ECN: Adding Hyperlink Insertion and XML Transformations to the XMLREST Frameworkhttps://community.emc.com/docs/DOC-6485[Accessed: 18-Jan-2012].

[28] Florent Georges Packaging System EXPath, 11-Nov-2010http://expath.org/spec/pkg[Accessed: 15-Jan-2012].

[29] Florent Georges CXAN: a case-study for Servlex an XML web framework in XML Prague 2011 Conference Proceedings, Lesser Town Campus Prague, Czech Republic, 2011, vol. 2011-519.

[30] Mark Hadley Paul Sandoz JAX-RS: Java API for RESTful Web Services Version 1.0 Sun Microsystems Inc., 08-Sep-2008.

[31] XSLT and XQuery Serialization 3.0http://www.w3.org/TR/xslt-xquery-serialization-30/[Accessed: 22-Jan-2012].

[32] XML Summer Schoolhttp://xmlsummerschool.com/[Accessed: 23-Jan-2012].

[33] See What I Think | Free software downloads at SourceForge.nethttp://sourceforge.net/projects/seewhatithink/[Accessed: 23-Jan-2012].

Compiling XQuery code into Javascript instructions using XSLT

Exploiting XQuery grammar

Alain Couthures
agenceXML
<alain.couthures@agencexml.com>

Abstract

There are different approaches for having XQuery in the browser. Developing a plugin is hazardous because of the number of versions of browsers. Performance is to be considered when porting an existing Java implementation with an engine such as GWT. Compiling XQuery code into Javascript instructions is another possibility and XML technics can facilitate this. XQueryX is an XML notation for XQuery code and converting into it is a first step to consider. Then, the resulting tree can be transformed with XSLT back to Javascript. XSLT 1.0 is powerful enough to write such a compiler, from text to tree and back to text (with the nodeset function), for both XQuery and XQuery Update Facility. This has been demonstrated by XSLTForms with its own XPath 1.0 engine: an XSLT 1.0 stylesheet to transform (with errors detection) XPath 1.0 expressions into Javascript objects and a collection of Javascript classes to effectively evaluate the XPath 1.0 expressions. YAPP is another interesting tool for generating a XSLT 1.0 parser from the grammar of a language expressed in BNF notation (XQuery Grammar is currently described in EBNF notation). After parsing, for more performance, instead of having the resulting Javascript instructions being the image of the tree, it is possible to generate the minimal number of effective instructions after optimization ([n] detection, maximal use of DOM API, ...), as compilers should always do. Finally, there are functions to be written so the generated instructions are limited to pertinent loops and calls.

Keywords: xml, xquery, xslt, javascript, ebnf, grammar, compiler, xforms, xsltforms

1. Introduction

XQuery at browser-side is interesting for developers already writing XQuery code at server-side: just one programming language. But XQuery has not been designed to handle interactivity within the browser (events, controls),

XForms is good for interactions. XForms is currently specified with its own actions in XML notation to avoid Javascript use. XForms 2.0 Specifications will encourage the use of XPath 2.0. A transformation function is also at study based on XSLT or XQuery. Defining actions associated to controls appears to because more and more complex (loops performed with attributes, variables,...) and XQuery Update Facility might be more appropriate.

So, it is time to consider how to implement XPath/XQuery in XSLTForms (an XForms implementation based on XSLT 1.0 to generate HTML+Javascript) with an industrial approach instead of a specific XPath 1.0 engine (with extra functions and dependencies detection).

Browsers cannot execute XQuery instructions natively. As for any other programming language, an implementer can choose between writing a plugin or coping with Javascript engines.

Plugins require to be installed and they have to be different for each kind of browser: they are not easy to use widely.

On the contrary, Javascript engines have a good compatibility and they are becoming faster and faster. There are generic tools to convert from high level programming languages into Javascript instructions so it is possible to get an XQuery engine written in Javascript from the same engine written in Java, for example. Performance optimization is difficult and mixing Javascript instructions of XSLTForms with such generated instructions might be cumbersome.

Actually in most situation, an XQuery engine is not required: there is no need to parse each XQuery instruction to interpret it at run time because, even if an eval() function can be very powerful, most XQuery instructions are static. XSLTForms is already implementing an XPath 1.0 compiler into Javascript instructions.

Compiling XQuery code is interesting for performance. Such a compiler should not be written in Javascript itself because it is too much iterative to be fast. XSLT 1.0 engines are faster and freely available almost everywhere: browsers, Windows, Linux, ASP.Net, PHP, Java EE,... Again, XSLTForms is already using XSLT 1.0 for its XPath 1.0 compiler.

2. Parsing XQuery with XSLT 1.0

2.1. Existing specifications and tools

2.1.1. XQuery Grammar

XQuery is described in EBNF notation. Example:

```
  FLWORExpr ::= (ForClause | LetClause)+ WhereClause? OrderByClause? "return" ▶
ExprSingle
  ForClause ::= "for" "$" VarName TypeDeclaration? PositionalVar? "in" ▶
```

```
ExprSingle ("," "$" VarName TypeDeclaration? PositionalVar? "in" ExprSingle)*
  PositionalVar ::= "at" "$" VarName
  LetClause ::= "let" "$" VarName TypeDeclaration? ":=" ExprSingle ("," "$" ▶
VarName TypeDeclaration? ":=" ExprSingle)*
  WhereClause ::= "where" ExprSingle
  OrderByClause ::= (("order" "by") | ("stable" "order" "by")) OrderSpecList
  OrderSpecList ::= OrderSpec ("," OrderSpec)*
  OrderSpec ::= ExprSingle OrderModifier
  OrderModifier ::= ("ascending" | "descending")? ("empty" ("greatest" | ▶
"least"))? ("collation" URILiteral)?
```

XQuery is also described in a non-normative XML notation (http://www.w3.org/2011/08/qt-applets/xgrammar_src.zip) to be used with JavaCC. This XML grammar notation allows to have multiple languages (XPath 1.0/2.0/3.0, XQuery 1.0/3.0, XQuery Update Facility 1.0, XSLT 2.0 Match Patterns,...) and corresponding entry points.

Contrary to EBNF notation, the XML grammar notation is evolving according to the W3C Query Working Group requirements for both JavaCC use and EBNF productions in specifications. Main elements are: grammar, language, start, production, ref, choice, sequence, optional, oneOrMore, zeroOrMore, string, level, binary, postfix, token, char, charClass, charCode, charRange, state, transition, tref. There are extra informations with them expressed by attributes (@lookahead, @process-value, @unfold, @break, @if, @not-if, @show, @node-type, @condition, @prefix-seq-type, @prod-user-action, @token-user-action, @nt-user-action-start, @nt-user-action-end, @whitespace-spec, @subtract-reg-expr, @needs-exposition-parens, @inline, @force-quote, @is-xml, @is-macro, @next-state, @action).

```
  <g:production name="FLWORExpr10" exposition-name="FLWORExpr" if="xcore ▶
xquery10">
    <g:oneOrMore if="xquery10" name="FLWORClauseList">
      <g:choice name="ForOrLet">
        <g:ref name="ForClause"/>
        <g:ref name="LetClause"/>
      </g:choice>
    </g:oneOrMore>
    <g:choice if="xcore" name="ForOrLetCore">
      <g:ref name="ForClause"/>
      <g:ref name="LetClause"/>
    </g:choice>
    <g:optional if="xquery10" name="OptionalWhere">
      <g:ref name="WhereClause"/>
    </g:optional>
    <g:optional name="OptionalOrderBy" if="xquery10">
      <g:ref name="OrderByClause"/>
    </g:optional>
    <g:string>return</g:string>
```

```
  <g:ref name="ExprSingle"/>
</g:production>
```

2.1.2. YAPP

YAPP (http://www.o-xml.org/yapp/) is a BNF engine written in XSLT 1.0 and generating a dedicated parser as another XSLT1.0 stylesheet . A specific notation is defined for describing the grammar: grammar, terminal, ignore, end, delimited, equals, construct, option, part.

```
<terminal name="axisName">
  <equals>ancestor::</equals>
  <equals>ancestor-or-self::</equals>
  <equals>attribute::</equals>
  <equals>child::</equals>
  <equals>descendant::</equals>
  <equals>descendant-or-self::</equals>
  <equals>following::</equals>
  <equals>following-sibling::</equals>
  <equals>namespace::</equals>
  <equals>parent::</equals>
  <equals>preceding::</equals>
  <equals>preceding-sibling::</equals>
  <equals>self::</equals>
</terminal>
<construct name="LocationPath">
  <option>
    <part name="RelativeLocationPath"/>
  </option>
  <option>
    <part name="AbsoluteLocationPath"/>
  </option>
</construct>
<construct name="AbsoluteLocationPath">
  <option>
    <part name="slash"/>
    <part name="RelativeLocationPath"/>
  </option>
  <option>
    <part name="slashslash"/>
    <part name="RelativeLocationPath"/>
  </option>
  <option>
    <part name="slash"/>
  </option>
</construct>
```

XSLT templates for tokenizing are separately generated. Custom XSLT templates can be associated with the grammar so a fully operational XSLT 1.0 stylesheet is generated from the grammar.

YAPP comes with a BNF grammar file with templates for dynamically recreate the grammar in the specific notation so YAPP can also be used to directly generate an XSLT 1.0 parser from a BNF text file.

```
<construct name="Grammar">
  <option>
    <part name="Rules"/>
    <part name="end"/>
  </option>
</construct>

<construct name="Rules">
  <option>
    <part name="Rule"/>
    <part name="Rules"/>
  </option>
  <option>
    <!-- empty -->
  </option>
</construct>

<construct name="Rule">
  <option>
    <part name="name"/>
    <part name="def"/>
    <part name="Definition"/>
    <part name="Definitions"/>
    <part name="semicolon"/>
  </option>
</construct>
```

With extra templates, YAPP is capable to push further for a better output structure.

For example, with a subset of XPath 1.0 grammar:

```
../queen
```

becomes

```
<Expr>
  <LocationPath>
    <RelativeLocationPath>
      <Step>
        <AbbreviatedStep>
          <dotdot>..</dotdot>
```

```
        </AbbreviatedStep>
      </Step>
      <slash>/</slash>
      <Step>
        <AxisSpecifier/>
        <NodeTest>
          <NameTest>
            <ncname>queen</ncname>
          </NameTest>
        </NodeTest>
      </Step>
    </RelativeLocationPath>
  </LocationPath>
  <end/>
</Expr>
```

YAPP has been written for Xalan-Java. Portability is not compromized but xalan:nodeset() function is heavily used.

No error management mechanism is incorporated in YAPP.

2.1.3. Jaxen (package org.jaxen.expr) and AJAXForms

AJAXForms (the ancestor XForms implementation for XSLTForms) has a Java part to generate Javascript instructions from XPath 1.0 expressions. It is based on Jaxen Expr package: when building an XPath expression, Java objects are created according to the expression structure and error management is performed at the same time.

```
try {
    this.xpath = new DOMXPath(this.expression);
} catch (JaxenException e) {
   throw new XPathException(e.getLocalizedMessage(), this.expression);
}
```

AJAXForms profits from this compiled form of expression to generate Javascript code to create similar Javascript objects.

```
private StringBuffer build(Expr expr, boolean instance) {
    StringBuffer jsExpr = new StringBuffer();

    if (expr instanceof BinaryExpr) {
        BinaryExpr bi = (BinaryExpr) expr;

        if (expr instanceof UnionExpr) {
            jsExpr.append("new UnionExpr(");
            jsExpr.append(build(bi.getLHS()));
            jsExpr.append(",");
            jsExpr.append(build(bi.getRHS()));
            jsExpr.append(')');
```

```
            } else {
                jsExpr.append("new BinaryExpr(");
                jsExpr.append(build(bi.getLHS()));
                jsExpr.append(", '");

                if (expr instanceof AdditiveExpr) {
                    jsExpr.append(((AdditiveExpr) expr).getOperator());
                } else if (expr instanceof EqualityExpr) {
                    jsExpr.append(((EqualityExpr) expr).getOperator());
                } else if (expr instanceof LogicalExpr) {
                    jsExpr.append(((LogicalExpr) expr).getOperator());
                } else if (expr instanceof MultiplicativeExpr) {
                    jsExpr.append(((MultiplicativeExpr) expr).getOperator());
                } else if (expr instanceof RelationalExpr) {
                    jsExpr.append(((RelationalExpr) expr).getOperator());
                }

                jsExpr.append("',");
                jsExpr.append(build(bi.getRHS()));
                jsExpr.append(')');
            }
  ...
```

The resulting string will create all the corresponding Javascript objects.

```
  . &gt; /range/from
```

becomes:

```
  new XPath(". &gt; /range/from",
            new BinaryExpr(new LocationExpr(false, new StepExpr('self',new ▶
NodeTestAny())),
                           '&gt;',
                           new LocationExpr(true,  new StepExpr('child',new ▶
NodeTestName('','range')),
                                                   new StepExpr('child',new ▶
NodeTestName('','from')))),
            '','');
```

2.1.4. XSLTForms 1.0

XSLTForms 1.0 has its own XPath 1.0 parser written in XSLT 1.0. It has not been generated but written by hand so unusual features might not be correctly supported. New features are difficult to add.

```
    <xsl:variable name="precedence" xmlns:xsl="http://www.w3.org/1999/XSL/▶
Transform">./▶
.;0.|.;1.div.mod.*.;2.+.-.;3.&lt;.&gt;.&lt;=.&gt;=.;4.=.!=.;5.and.;6.or.;7.,.;8.</▶
xsl:variable>
```

```
    <xsl:template name="xp2js" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
      <xsl:param name="xp"/>
      <xsl:param name="args"/>
      <xsl:param name="ops"/>
      <xsl:variable name="c" select="substring(normalize-space($xp),1,1)"/>
      <xsl:variable name="d" select="substring-after($xp,$c)"/>
      <xsl:variable name="r">
        <xsl:choose>
          <xsl:when test="contains('./@*',$c)">
            <xsl:variable name="t"><xsl:call-template ▶
name="getLocationPath"><xsl:with-param name="s" select="concat($c,$d)"/></▶
xsl:call-template></xsl:variable>
            <xsl:value-of select="substring-before($t,'.')"/>
            <xsl:text>.new XsltForms_locationExpr(</xsl:text>
            <xsl:choose>
            <xsl:when test="$c = '/' and not(starts-with($ops,'3.0./'))">true</▶
xsl:when>
              <xsl:otherwise>false</xsl:otherwise>
            </xsl:choose>
            <xsl:value-of select="substring-after($t,'.')"/><xsl:text>)</▶
xsl:text>
          </xsl:when>
          <xsl:when test="$c = &quot;'&quot;">
            <xsl:variable name="t">'<xsl:value-of ▶
select="substring-before($d,&quot;'&quot;)"/>'</xsl:variable>
            <xsl:value-of select="concat(string-length($t),'.new ▶
XsltForms_cteExpr(',$t,')')"/>
          </xsl:when>
```

XSLTForms XPath parser does not depend on node-set() XSLT 1.0 extension: formatted strings are heavily used for exchanging complex structures between named templates and only text is generated.

For example, detecting whether node sorting is required or not is performed on the resulting string. Errors when parsing are integrated in the resulting string and extracted after processing.

```
  <xsl:otherwise>~~~~Unexpected char at '<xsl:value-of select="concat($c,$d)"/▶
>'~#~#</xsl:otherwise>
```

2.1.5. XQueryX

XQueryX is not an human-friendly XML notation. XQueryX structure is optimized for processing so it is slightly different from XQuery Grammar structure (different element names, extra elements,...).

```
  <xsd:complexType name="flworExpr">
    <xsd:complexContent>
```

```
    <xsd:extension base="expr">
      <xsd:sequence>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element ref="forClause"/>
          <xsd:element ref="letClause"/>
        </xsd:choice>
        <xsd:element name="whereClause" minOccurs="0"/>
        <xsd:element name="orderByClause" minOccurs="0"/>
        <xsd:element name="returnClause"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:element name="flworExpr" type="flworExpr" substitutionGroup="expr"/>

<xsd:element name="forClause">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="forClauseItem" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="forClauseItem">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="typedVariableBinding"/>
      <xsd:element ref="positionalVariableBinding" minOccurs="0"/>
      <xsd:element name="forExpr" type="exprWrapper"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="letClause">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="letClauseItem" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="letClauseItem">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="typedVariableBinding"/>
```

```
      <xsd:element name="letExpr" type="exprWrapper"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="whereClause" type="exprWrapper"/>

<xsd:element name="returnClause" type="exprWrapper"/>
```

XQueryX comes with an XSLT 1.0 stylesheet to generate XQuery equivalent but not the reverse one...

```
<xsl:template match="xqx:flworExpr">
  <xsl:value-of select="$NEWLINE"/>
  <xsl:value-of select="$LPAREN"/>
  <xsl:apply-templates select="*"/>
  <xsl:value-of select="$RPAREN"/>
</xsl:template>

<xsl:template match="xqx:forClause">
  <xsl:text> for </xsl:text>
  <xsl:call-template name="commaSeparatedList"/>
  <xsl:value-of select="$NEWLINE"/>
</xsl:template>

<xsl:template match="xqx:forClauseItem">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="xqx:forExpr">
  <xsl:value-of select="$NEWLINE"/>
  <xsl:text>    in </xsl:text>
      <xsl:apply-templates/>
</xsl:template>

<xsl:template match="xqx:letClause">
  <xsl:text> let </xsl:text>
  <xsl:call-template name="commaSeparatedList"/>
  <xsl:value-of select="$NEWLINE"/>
</xsl:template>

<xsl:template match="xqx:letClauseItem">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="xqx:letExpr">
  <xsl:value-of select="$ASSIGN"/>
```

```
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="xqx:returnClause">
  <xsl:text> return </xsl:text>
  <xsl:apply-templates select="*"/>
  <xsl:value-of select="$NEWLINE"/>
</xsl:template>

<xsl:template match="xqx:whereClause">
  <xsl:text> where </xsl:text>
  <xsl:apply-templates select="*"/>
  <xsl:value-of select="$NEWLINE"/>
</xsl:template>
```

2.2. Proposed architecture

2.2.1. XQuery Grammar Parser generating intermediate XML document

The XML notation of the XQuery grammar is used to generate the corresponding XSLT templates with an XSLT stylesheet inspired from YAPP: templates for tokenizing and templates for productions.

This new XSLT stylesheet is more complex than the one for YAPP for EBNF explicit support but without automat mechanism.

Multiple languages support requires to add tests within generated templates.

Instead of using intermediate XML fragments, formatted strings avoid repetitive node-set() calls.

The intermediate XML document notation directly reflects what was recognized according to the grammar.

So,

```
../queen
```

becomes:

```
<PathExpr>
  <RelativePathExpr>
    <StepExpr>
      <AxisOrFilterStep>
        <AxisStep>
          <ForwardOrReverseStep>
            <ReverseStep>
              <ReverseAxisOrAbbrev>
                <AbbrevReverseStep/>
              </ReverseAxisOrAbbrev>
            </ReverseStep>
```

```
          </ForwardOrReverseStep>
          <PredicateList/>
        </AxisStep>
      </AxisOrFilterStep>
    </StepExpr>
    <RelativePathExprTail>
      <Slash/>
      <StepExpr>
        <AxisOrFilterStep>
          <AxisStep>
            <ForwardOrReverseStep>
              <ForwardStep>
                <ForwardAxisOrAbbrev>
                  <AbbrevForwardStep>
                    <NodeTest>
                      <KindOrNameTest>
                        <NameTest>
                          <QNameOrWildcard>
                            <_QName_or_EQName>
                              <QName>
                                <QNameChoiceList>
                                  <FunctionQName>
                                    <FunctionQNameChoiceList>
                                      <QNameToken>
                                        <LocalPart>queen</LocalPart>
                                      </QNameToken>
                                    </FunctionQNameChoiceList>
                                  </FunctionQName>
                                </QNameChoiceList>
                              </QName>
                            </_QName_or_EQName>
                          </QNameOrWildcard>
                        </NameTest>
                      </KindOrNameTest>
                    </NodeTest>
                  </AbbrevForwardStep>
                </ForwardAxisOrAbbrev>
              </ForwardStep>
            </ForwardOrReverseStep>
          </AxisStep>
        </AxisOrFilterStep>
      </StepExpr>
    </RelativePathExprTail>
  </RelativePathExpr>
</PathExpr>
```

Errors are embedded in resulting formatted text so they can be extracted.

2.2.2. XQueryX generation

A single node-set() call is required at the end to allow to process the resulting XQueryX for target language generation.

There is a way to validate if the generated XQueryX document is correct: apply the XQueryX to XQuery XSLT stylesheet, run the two XQuery codes and check that results are identical. This allowed also to beautify XQuery code.

Finally,

```
../queen
```

becomes:

```
<pathExpr>
  <stepExpr>
    <xpathAxis>parent</xpathAxis>
    <anyKindTest/>
  </stepExpr>
  <stepExpr>
    <xpathAxis>child</xpathAxis>
    <nameTest>queen</nameTest>
  </stepExpr>
</pathExpr>
```

3. Generating Javascript instruction

3.1. The object approach

3.1.1. Object creation

From XQueryX structure, XSLT templates create a similar tree of objects creations. A non-leaf node becomes an object, its children being parameters for its constructor method.

```
<pathExpr>
  <stepExpr>
    <xpathAxis>parent</xpathAxis>
    <anyKindTest/>
  </stepExpr>
  <stepExpr>
    <xpathAxis>child</xpathAxis>
    <nameTest>queen</nameTest>
  </stepExpr>
</pathExpr>
```

becomes:

```
    new pathExpr(new stepExpr(XPATHAXIS_parent), new stepExpr(XPATHAXIS_child, ▶
  new nameTest("queen"))
```

3.1.2. Object evaluation

Each generated object has an evaluate() method with the standard parameters: the reference of the current node and a namespace resolver.

XSLTForms 1.0 is using the same approach:

```
  XsltForms_binaryExpr.prototype.evaluate = function(ctx) {
    var v1 = this.expr1.evaluate(ctx);
    var v2 = this.expr2.evaluate(ctx);
    var n1;
    var n2;
    n1 = XsltForms_globals.numberValue(v1);
    n2 = XsltForms_globals.numberValue(v2);
    if (isNaN(n1) || isNaN(n2)) {
      n1 = XsltForms_globals.stringValue(v1);
      n2 = XsltForms_globals.stringValue(v2);
    }
    var res = 0;
    switch (this.op) {
      case 'or'  : res = XsltForms_globals.booleanValue(v1) || ▶
  XsltForms_globals.booleanValue(v2); break;
      case 'and' : res = XsltForms_globals.booleanValue(v1) && ▶
  XsltForms_globals.booleanValue(v2); break;
      case '+'   : res = n1 + n2; break;
      case '-'   : res = n1 - n2; break;
      case '*'   : res = n1 * n2; break;
      case 'mod' : res = n1 % n2; break;
      case 'div' : res = n1 / n2; break;
      case '='   : res = n1 === n2; break;
      case '!='  : res = n1 !== n2; break;
      case '<'   : res = n1 < n2; break;
      case '<='  : res = n1 <= n2; break;
      case '>'   : res = n1 > n2; break;
      case '>='  : res = n1 >= n2; break;
    }
    return typeof res === "number" ? Math.round(res*1000000)/1000000 : res;
  }
```

For XForms, dependencies have to be identified: a binding evaluation has to be performed again only if the value of an intermediate node has been modified.

3.2. Run-time data model and functions set

3.2.1. Data model

Javascript objects are to be used for every data. Sequences data can easily be modelized with a Javascript array, except for ranges for which bounds have to memorized.

Because Javascript data types are not as rich as XPath atomic types, objects are to be used. Defining classes might be not good for performance, just JSON structures for type and value.

3.2.2. Functions

XPath operators and functions equivalents have to be rewritten in Javascript as in XSLTForms 1.0.

3.3. Effective instructions

3.3.1. Loops and temporary variables

Steps in a location are transformed into successive loops while predicates are filtering expressions evaluated for each item. This means that variables have to be generated for processing.

3.3.2. Possible optimizations

There are many possibilities when generating instructions to optimize them.

For example:

- Attributes can be read directly with DOM API
- Positional predicates detection allows to limit loops
- Dependencies are not to be detected when this is not required

4. Conclusion

Even if compiling XQuery code into Javascript instructions with XSLT this way is not highly optimized, this is a robust and evolutive solution which can be used at server-side or at client-side.

It is also possible to optimize the generation of the Javascript instructions without modifying the XQuery parser.

Implementing an XQuery/XSLT hybrid

Parsing and compiling Carrot

Evan Lenz
MarkLogic Corporation
<evan.lenz@marklogic.com>

Abstract

The idea for Carrot, "an appetizing hybrid of XQuery and XSLT," was first presented at Balisage 2011. Since then, progress has been made on implementing Carrot by way of compilation to XSLT. This paper describes current progress on the Carrot parser and compiler, detailing certain fundamental aspects of the current architecture.

Keywords: XML, XSLT, XQuery, Carrot

I first presented the idea for "Carrot: An appetizing hybrid of XQuery and XSLT"[1] at Balisage 2011[2]. At that time, it was still only in the conception stage. Although most of the chief ideas for the language were fleshed out, no implementation was yet available and my initial attempts at generating a Carrot parser had failed. Now, several months later, I have an XQuery-based Carrot parser and the compiler is moving right along. This paper describes my current strategy for implementing Carrot, and highlights things learned along the way.

1. Introduction

XQuery and XSLT are closely related languages. They share a common data model and large syntactic subset in XPath 2.0. Depending on a programmer's background, he or she tends to gravitate toward (or be repelled from) one language or the other. However, despite the large overlap, each language provides unique value. XSLT has template rules, whereas XQuery is concise and composable. Carrot aims to bridge the gap between the two, bringing into a single hybrid language the best features of both languages, particularly:

- the power and flexibility of XSLT's template rules and modularization features, and
- the conciseness and composability of XQuery expressions.

[1] http://balisage.net/Proceedings/vol7/html/Lenz01/BalisageVol7-Lenz01.html
[2] http://balisage.net/2011/Program.html

Carrot has a minimalistic design, both in terms of its syntax and its feature set. It relies heavily on the syntax of XQuery and the semantics of both XSLT and XQuery, rarely departing from either. While Carrot can be thought of as an alternative syntax for XSLT, it is more properly described as an XQuery/XSLT hybrid, because it affords the full power of XQuery expressions. In that sense, it can also be thought of as a host language for XQuery expressions.

For purposes of this introduction, the basic idea of Carrot is that template rules can be defined and invoked using a function-like syntax, distinguished from a normal function by use of the caret symbol (^). Like XSLT, a rule is defined using a pattern (what looks like the "function argument"), a mode name (what looks like the "function name"), and a numeric priority value (implicit or explicit). Unlike XSLT, rules can be invoked (i.e. templates applied) inline—within an expression. As with XQuery, all expressions are fully composable with each other. In fact, 90% of the Carrot grammar is identical to the XQuery grammar. XQuery expressions are extended to support ruleset invocations, text node literals, and shallow copy constructors. In this way, Carrot combines the best of both languages.

Although Carrot has been fully introduced elsewhere[3], a tutorial-like introduction is presented for the reader's convenience in Appendix A. If you are new to Carrot or want to be reminded of what it's about, you should read that introduction first.

2. Implementation approaches

There are several approaches that could be taken to implementing Carrot:

1. native implementation
2. compilation to XSLT
3. compilation to XQuery

One piece of feedback received at Balisage was that compilation to XQuery or XSLT would be particularly valuable, in that it would allow users to continue development outside of Carrot, or use any XQuery or XSLT implementation that does not directly support Carrot. For that reason, the current implementation I'm working on consists of an XQuery-based parser and an XSLT-based compiler that generates XSLT 2.0 code (#2 above). This will enable users to run Carrot programs by calling a function library.

The plan is to provide two implementations: one for Saxon and one for MarkLogic. (They will differ only in how the processor's extensions call XSLT from XQuery and vice versa.) Since both of these products support both XQuery and XSLT, they are able to support Carrot.

[3] http://balisage.net/Proceedings/vol7/html/Lenz01/BalisageVol7-Lenz01.html

3. Parsing Carrot

The Carrot parser was implemented using these steps:

1. Start with the XQuery 1.0 EBNF grammar[4]
2. Hand-modify the grammar, turning it into a grammar for Carrot[5]
3. Run the grammar through the REx parser generator[6] (online utility), using these options:
   - -xquery -tree

3.1. Step 1: Start with the XQuery grammar

The first step is simple. I started with the EBNF grammar for XQuery 1.0 provided here: http://www.bottlecaps.de/rex/XQueryV10.ebnf.

3.2. Step 2: Define Carrot by manually modifying the XQuery grammar

Next, I disabled large portions of the XQuery grammar that had to do with top-level declarations. Some of these will be reintroduced (starting with `NamespaceDecl`) with or without modification, as Carrot evolves.

The rest of this section will show the Carrot-specific grammar rules, current at the time of this writing. The XQuery production rules on which these depend have been left unchanged (except when some slight refactoring was beneficial in providing extra hints to the compiler). The latest version of the Carrot grammar can be found here: https://github.com/evanlenz/Carrot/blob/master/parser/Carrot.ebnf.

3.2.1. Carrot definitions

The following production rule indicates the overall structure of a Carrot module:

```
/* Top-level Carrot module */
Carrot   ::= CarrotModule EOF
CarrotModule
         ::= (NamespaceDecl Separator)*
             /* other top-level declarations will go here (import, etc.) */
             ((VarDecl | FunctionDecl | RuleDecl) Separator)*
```

The `NamespaceDecl` production rule in Carrot has been left unchanged; namespace declarations use the same syntax as XQuery. After any namespace declarations (and other declarations for import, etc., which have not yet been but will soon be defined for Carrot), the Carrot module consists of zero or more:

[4] http://www.bottlecaps.de/rex/XQueryV10.ebnf
[5] https://github.com/evanlenz/Carrot/blob/master/parser/Carrot.ebnf
[6] http://www.bottlecaps.de/rex/

- variable definitions (`VarDecl`),
- function definitions (`FunctionDecl`), and
- rule definitions (`RuleDecl`).

(Unlike XQuery, a Carrot module consists entirely of definitions. There is no query body in Carrot, and Carrot makes no distinction between main modules and query modules.)

The following example Carrot module includes a namespace declaration followed by one of each kind of definition:

```
declare namespace my="http://example.com";
$foo     := "a string";   (: VarDecl :)
my:foo() := $foo;         (: FunctionDecl :)
^(/)     := my:foo();     (: RuleDecl :)
```

Each definition is terminated using the colon character:

```
Separator ::= ';'
```

3.2.1.1. Variable definitions

Variable and function definitions in Carrot are functionally equivalent to the same constructs in XQuery, with a slightly simpler syntax. In the case of variable definitions, the "declare variable" verbiage has been removed:

```
VarDecl ::= '$' QName TypeDeclaration? ':=' Expr
```

Also, for consistency with the other two types of definitions (`FunctionDecl` and `RuleDecl`), the right-hand side of a variable definition is allowed to have a full `Expr`, not just `ExprSingle`. In practice, this means that Carrot, unlike XQuery, doesn't require you to put parentheses around a sequence expression that's bound to a variable. For example, `$foo := 1,2,3;` is a valid variable definition in Carrot, whereas in XQuery, when binding the value to a top-level variable, you would need to write the expression using parentheses: `(1,2,3)`.

3.2.1.2. Function definitions

Similarly, in the case of function definitions, "declare function" has been removed. Also, for consistency with the other two types of definitions (`VarDecl` and `RuleDecl`), the binding operator (`:=`) is used rather than curly braces (`{}`):

```
FunctionDecl ::= FunctionName '(' ParamList? ')' TypeDeclaration? ':=' Expr
```

3.2.1.3. Rule definitions

Rule definitions have no analogue in XQuery. They correspond directly to template rules in XSLT, but in Carrot they use a function-like syntax:

```
  RuleDecl ::= '^' (ModeName ('|' ModeName)* )? '(' Pattern (';' RuleParamList)? ▶
')' Priority? ':=' Expr
  ModeName ::= (QName | '#current' | '#default')
  Priority ::= (IntegerLiteral | DecimalLiteral)
  RuleParamList
           ::= ParamWithDefault (',' ParamWithDefault)*
  ParamWithDefault
           ::= Tunnel? Param (":=" ExprSingle)?
  Tunnel   ::= 'tunnel'
```

The production rule for `Pattern` was copied unchanged from the XSLT 2.0 specification.

3.2.2. Carrot expressions

Since Carrot expressions are just XQuery expressions with some extensions, most of the production rules for XQuery expressions were left unchanged. At this time, Carrot includes only three extensions to XQuery expression syntax:

- ruleset invocations (`^mode(nodes)`),
- shallow copy constructors (`copy{...}`), and
- text node literals (`` `my text node` ``).

The identity transform in Carrot makes use of the first two of Carrot's extended expression syntax constructs (a ruleset invocation and a copy constructor):

```
^(@*|node()) := copy{ ^(@*|node()) };
```

And here's an example Carrot rule that uses a text node literal:

```
^title(name) := `Mr. `;
```

3.2.2.1. Ruleset invocations

In the case of ruleset invocations (equivalent to `xsl:apply-templates` in XSLT), XQuery is extended by adding `RulesetCall` to `PrimaryExpr`:

```
PrimaryExpr ::= Literal
              | VarRef
              | ParenthesizedExpr
              | ContextItemExpr
              | FunctionCall
              | OrderedExpr
              | UnorderedExpr
```

```
                | Constructor
                | RulesetCall /* Carrot-specific */
```

Here's the production rule for `RulesetCall`:

```
RulesetCall ::= '^' ModeName? '(' Expr? (';' RulesetCallParamList)? ')'
RulesetCallParamList ::= InitializedParam (',' InitializedParam)*
InitializedParam ::= Tunnel? Param ':=' ExprSingle
```

3.2.2.2. Shallow copy constructors

Shallow copy constructors were added by extending `ComputedConstructor` in the XQuery grammar:

```
ComputedConstructor ::= CompDocConstructor
                      | CompElemConstructor
                      | CompAttrConstructor
                      | CompTextConstructor
                      | CompCommentConstructor
                      | CompPIConstructor
                      | CompCopyConstructor /* Carrot-specific */

CompCopyConstructor ::= 'copy' '{' Expr '}'
```

3.2.2.3. Text node literals

Text node literals were added by extending `Literal` in the XQuery grammar:

```
Literal  ::= NumericLiteral
           | StringLiteral
           | TextNodeLiteral /* Carrot-specific */
```

The production rule for `TextNodeLiteral` is very similar to the XQuery production rule for `StringLiteral` (shown unmodified below for comparison purposes):

```
  TextNodeLiteral ::= '`' ( PredefinedEntityRef | CharRef | EscapeTick | [^`&] ▶
)* '`' /* ws: explicit */
  StringLiteral  ::= '"' ( PredefinedEntityRef | CharRef | EscapeQuot | [^"&] ▶
)* '"'
                   | "'" ( PredefinedEntityRef | CharRef | EscapeApos | [^'&] ▶
)* "'" /* ws: explicit */
```

3.2.3. Conclusion

And with that, we have all the production rules that are unique to Carrot. The rest of the grammar comes straight from XQuery 1.0 (or XSLT 2.0, in the case of `Pattern`).

3.3. Step 3: Generate the parser

The final step is generating the parser using the REx parser generator[7]. Running this online utility with the -xquery and -tree options results in an XQuery-based parser which outputs an XML parse tree for the given Carrot module.

> **Note**
> For full interoperability with MarkLogic Server, it was necessary to add a default function namespace declaration to the auto-generated XQuery parser: `declare default function namespace "http://www.w3.org/2005/xpath-functions";`

Writing the compiler thus becomes "a small matter of XML transformations". Since the XML consists solely of element markup added around the original Carrot text, the string-value of the output is the same as the original Carrot code. This is a very useful property since it means that translation from XPath in the Carrot to XPath in the resulting XSLT is trivial: just get the string-value.

For example, the following Carrot expression is also a valid XPath 2.0 expression:

```
for $x in /names/name return lower-case(.)
```

Thus, compilation to XSLT 2.0 in this case is trivial. We just get the string-value of the XML parse tree for the expression, as shown in Example 1 (with extra line breaks added).

Example 1. XML parse tree for a simple FLWOR expression

```
<Expr><ExprSingle><FLWORExpr><ForClause><TOKEN>for</TOKEN><ForBinding> <TOKEN>$</▶
TOKEN>
<VarName><QName><FunctionName><QName>x</QName></FunctionName></QName></VarName> ▶
<TOKEN>
in</TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr>
<MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr>
<CastExpr><UnaryExpr><ValueExpr><PathExpr> <TOKEN>/</▶
TOKEN><RelativePathExpr><StepExpr>
<AxisStep><ForwardStep><AbbrevForwardStep><NodeTest><NameTest><QName><FunctionName><QName>
names</QName></FunctionName></QName></NameTest></NodeTest></AbbrevForwardStep></▶
ForwardStep>
<PredicateList/></AxisStep></StepExpr><TOKEN>/</▶
TOKEN><StepExpr><AxisStep><ForwardStep>
<AbbrevForwardStep><NodeTest><NameTest><QName><FunctionName><QName>name</QName></▶
FunctionName>
</QName></NameTest></NodeTest></AbbrevForwardStep></ForwardStep><PredicateList/▶
></AxisStep>
```

[7] http://www.bottlecaps.de/rex/

```
</StepExpr></RelativePathExpr></PathExpr></ValueExpr></UnaryExpr></CastExpr></▶
CastableExpr>
</TreatExpr></InstanceofExpr></IntersectExceptExpr></UnionExpr></▶
MultiplicativeExpr>
</AdditiveExpr></RangeExpr></ComparisonExpr></AndExpr></OrExpr></ExprSingle></▶
ForBinding>
</ForClause> <TOKEN>return</▶
TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr>
<AdditiveExpr><MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr>
<CastableExpr><CastExpr><UnaryExpr><ValueExpr><PathExpr><RelativePathExpr><StepExpr>
<FilterExpr><PrimaryExpr><FunctionCall><FunctionName> <QName>lower-case</QName></▶
FunctionName>
<TOKEN>(</▶
TOKEN><ExprSingle><OrExpr><AndExpr><ComparisonExpr><RangeExpr><AdditiveExpr>
<MultiplicativeExpr><UnionExpr><IntersectExceptExpr><InstanceofExpr><TreatExpr><CastableExpr>
<CastExpr><UnaryExpr><ValueExpr><PathExpr><RelativePathExpr><StepExpr><FilterExpr>
<PrimaryExpr><ContextItemExpr><TOKEN>.</TOKEN></ContextItemExpr></▶
PrimaryExpr><PredicateList/>
</FilterExpr></StepExpr></RelativePathExpr></PathExpr></ValueExpr></UnaryExpr></▶
CastExpr>
</CastableExpr></TreatExpr></InstanceofExpr></IntersectExceptExpr></UnionExpr>
</MultiplicativeExpr></AdditiveExpr></RangeExpr></ComparisonExpr></AndExpr></▶
OrExpr>
</ExprSingle><TOKEN>)</TOKEN></FunctionCall></PrimaryExpr><PredicateList/></▶
FilterExpr>
</StepExpr></RelativePathExpr></PathExpr></ValueExpr></UnaryExpr></CastExpr></▶
CastableExpr>
</TreatExpr></InstanceofExpr></IntersectExceptExpr></UnionExpr></▶
MultiplicativeExpr>
</AdditiveExpr></RangeExpr></ComparisonExpr></AndExpr></OrExpr></ExprSingle></▶
FLWORExpr>
</ExprSingle></Expr>
```

which yields the original:

```
for $x in /names/name return lower-case(.)
```

So for a large portion of Carrot, compilation to XPath 2.0 is trivial. In fact, we can just use XSLT's built-in template rule for elements and text nodes (process children and copy text, respectively). A single call to `<xsl:apply-templates/>` takes care of this large portion:

```
<xsl:apply-templates mode="xpath" select="Expr"/>
```

For an implementation that compiles to XQuery, getting the string-value takes care of an even larger portion of Carrot (the portion that is identical to XQuery 1.0).

Of course, not everything in Carrot can be trivially translated to XSLT or XQuery. But this is where things start to get interesting.

4. Compiling Carrot

It may help to broadly identify the nature of the translation task, depending on what the compiler's target language is (XQuery or XSLT). The following tables show the obvious corollaries for each major construct in Carrot:

Table 1. Obvious corollaries in XSLT

In Carrot:	Translated to:
Rule definition	`<xsl:template>`
Variable definition	`<xsl:variable>`
Function definition	`<xsl:function>`
XPath expression	XPath expression
non-XPath expression	???

Table 2. Obvious corollaries in XQuery

In Carrot:	Translated to:
Rule definition	???
Variable definition	`declare variable...`
Function definition	`declare function...`
XQuery expression	XQuery expression
non-XQuery extension	???

Both tasks have their murky areas that will require extra thought (signified by question marks in the above tables). While I hope that there will be a Carrot compiler that targets XQuery, I chose to start with XSLT, because Carrot is designed to behave very much like XSLT. In particular, rules in Carrot are equivalent to template rules in XSLT, whereas XQuery has no corresponding built-in construct. Also, even though imports and includes haven't yet been added to the Carrot grammar as of this writing, the intention is that Carrot modules will relate to each other the same way that XSLT modules relate to each other, e.g., using import precedence. Simulating XSLT's import precedence is one task that I won't have to worry about, whereas it will be extra work for the person (perhaps me in the future) who is writing a Carrot compiler that outputs XQuery.

Since Carrot is a hybrid of both XSLT and XQuery, the task of compiling it is closely related to some other projects involved in translating from one language to the other:

- Translating XQuery to XSLT: David Carlisle's xq2xml project[8]
- Translating XSLT to XQuery: "Compiling XSLT 2.0 into XQuery 1.0," by Fokoue, Rose, Siméon, and Villard[9]

As I've found some of the insights in David Carlisle's xq2xml project to be helpful for compiling Carrot to XSLT, I'm sure that insights in the latter project would be similarly helpful in compiling Carrot to XQuery.

Having decided to target XSLT, the next step is to implement the transformation from the XML parse tree representation to an XSLT 2.0 stylesheet that implements the behavior of the Carrot module. Naturally, I chose to use XSLT for this task. (As a matter of fact, once I've got a fairly solid baseline compiler working, I plan to port the compiler itself to Carrot.) Generating the top-level definitions themselves is pretty straightforward. For example, a function definition maps directly to an `<xsl:function>` element:

```
<xsl:template match="FunctionDecl">
  <out:function name="{FunctionName/QName}">
    <xsl:apply-templates select="TypeDeclaration"/>
    <xsl:apply-templates select="Expr"/>
  <out:function>
<xsl:template>
```

Things start to get tricky as soon as expressions are involved. In XQuery and Carrot, there is only ever one syntactic context for the construction of XPath data model values (sequences of nodes, strings, numbers, etc.), namely the *expression*. Anything you can create or reference is created or referenced by an expression, in XQuery and Carrot. However, that's not true in XSLT, which has two syntactic contexts for constructing sequences:

- expressions, and
- sequence constructors.

Expressions are limited to XPath 2.0 syntax and appear in attribute values, whereas sequence constructors consist of zero or more:

- literal result elements,
- text nodes,
- XSLT instructions, or
- extension elements

[8] http://monet.nag.co.uk/xq2xml/
[9] http://www2005.org/cdrom/docs/p682.pdf

Most importantly, not everything in XQuery (or Carrot) can be expressed in XPath. If it could, then our task would be much simpler. For example, a global variable definition would be trivial to implement. We could just copy the expression as is into an `<xsl:variable>`'s "select" attribute:

```
<xsl:template match="VarDecl">
  <out:variable name="{QName}" select="{Expr}"/>
    <xsl:apply-templates select="TypeDeclaration"/>
  <out:variable>
<xsl:template>
```

And in fact that will work just fine for some variable definitions. For example, using the above rule:

```
$foo := 'my string';
```

would translate to:

```
<xsl:variable name="foo" select="'my string'"/>
```

But things break as soon as the right-hand side has something other than an XPath expression:

```
$foo := <my-element>hello</my-element>
```

Now we need a sequence constructor, *not* an expression, in the XSLT result. So we'll have to do something more sophisticated: recognize the non-XPath expression and act accordingly.

Similarly, if we're already in the context of a sequence constructor, e.g., inside `<xsl:function>` and we come across an expression, we'll need to switch back to an expression context, such as the "select" attribute of `<xsl:sequence>`. For example, consider the following function definition in Carrot:

```
my:title() := <TITLE>{ /doc/title }</TITLE>;
```

In the compiled-to-XSLT version of this code, the `<TITLE>` element constructor translates directly to a literal result element in XSLT (i.e. a sequence constructor), but for its contents, we'll need to switch back to an expression context, using `<xsl:sequence>` (or more idiomatically, `<xsl:copy-of>`):

```
<xsl:function name="my:title">
  <TITLE>
    <xsl:sequence select="/doc/title"/>
  </TITLE>
</xsl:function>
```

But what if the expression wasn't just a simple XPath expression? What if it had a predicate containing an element constructor? That's perfectly legal in Carrot and XQuery, but not XPath (and thus not in XSLT):

```
/doc/title[deep-equal(., <title><em>A special title</em></title>)]
```

Now we find ourselves in a situation of having to switch *back* to using a sequence constructor context. This can be done, of course, but we'll need to defer it to a variable or function definition:

```
/doc/title[deep-equal(., my:special-title())]
```

What this illustrates is that Carrot and XQuery are fully syntactically composable. Values can only be represented by expressions, and where an expression can appear, *any* expression can appear. We might say that XSLT is *semantically* composable: any value you can create in XQuery, you can also create in XSLT. But since values can (and must) be represented in two different ways in XSLT, the language is not syntactically composable.

The challenge is then in identifying when to switch back and forth between these two contexts: expressions and sequence constructors, and having done that, translate from the various XQuery/Carrot expressions into equivalent XSLT constructs. But first, since the XML parse tree is pretty noisy (as shown in Example 1), try cleaning it up, removing whatever noise I can. Thus the compiler, as currently implemented, breaks the task into three steps, achieving the following separation of concerns:

1. Simplify the parse tree, removing noise.
2. Annotate the expressions (as XPath or sequence constructor).
3. Generate the XSLT.

The rest of this section is about the details of each step. To see the full source code for the latest Carrot compiler, see https://github.com/evanlenz/Carrot/tree/master/compiler.

Warning
The Carrot compiler is a work in progress, not yet ready for prime time.

4.1. Step 1: Simplify the parse tree

Consider the following simple Carrot module:

```
^(/) := for $x in /names/name return lower-case(.);
```

The FLWOR expression shown here is the same as the one shown in Example 1. After running simplify.xsl[10] against the full XML parse tree for this module, we get a much-simplified result (with indentation added):

```
<Carrot>
  <CarrotModule>
    <RuleDecl>
```

[10] https://github.com/evanlenz/Carrot/blob/master/compiler/simplify.xsl

```
<TOKEN>^</TOKEN>
<TOKEN>(</TOKEN>
<Pattern>
  <PathPattern>
    <TOKEN>/</TOKEN>
  </PathPattern>
</Pattern>
<TOKEN>)</TOKEN>
<TOKEN>:=</TOKEN>
<Expr>
  <FLWORExpr>
    <ForClause> <TOKEN>for</TOKEN>
      <ForBinding> <TOKEN>$</TOKEN>
        <VarName>
          <QName>
            <FunctionName>
              <QName>x</QName>
            </FunctionName>
          </QName>
        </VarName> <TOKEN>in</TOKEN>
        <ExprSingle>
          <PathExpr> <TOKEN>/</TOKEN>
            <AxisStep>
              <ForwardStep>
                <AbbrevForwardStep>
                  <NodeTest>
                    <NameTest>
                      <QName>
                        <FunctionName>
                          <QName>names</QName>
                        </FunctionName>
                      </QName>
                    </NameTest>
                  </NodeTest>
                </AbbrevForwardStep>
              </ForwardStep>
            </AxisStep>
            <TOKEN>/</TOKEN>
            <AxisStep>
              <ForwardStep>
                <AbbrevForwardStep>
                  <NodeTest>
                    <NameTest>
                      <QName>
                        <FunctionName>
                          <QName>name</QName>
```

```
                                </FunctionName>
                              </QName>
                            </NameTest>
                          </NodeTest>
                        </AbbrevForwardStep>
                      </ForwardStep>
                    </AxisStep>
                  </PathExpr>
                </ExprSingle>
              </ForBinding>
            </ForClause> <TOKEN>return</TOKEN>
            <ExprSingle>
              <FunctionCall>
                <FunctionName> <QName>lower-case</QName>
                </FunctionName>
                <TOKEN>(</TOKEN>
                <ExprSingle>
                  <ContextItemExpr>
                    <TOKEN>.</TOKEN>
                  </ContextItemExpr>
                </ExprSingle>
                <TOKEN>)</TOKEN>
              </FunctionCall>
            </ExprSingle>
          </FLWORExpr>
        </Expr>
      </RuleDecl>
      <Separator>
        <TOKEN>;</TOKEN>
      </Separator>
    </CarrotModule>
  </Carrot>
```

Only the elements that are actually descriptive (and useful) remain. The stripped-out elements are artifacts of how the entire XQuery grammar is defined. Most importantly, the simplification enables the compiler to confidently know that, if it comes across a given element, then that element is descriptive of the contained expression. For example, if `<AdditiveExpr>` appears in the source, then it means that there is actually an additive expression present, e.g. `2 + 2`.

The following template rule in simplify.xsl shows how the simplification is achieved:

```
<!-- If only one child, then it's a useless container -->
<!-- If the compiler comes across one of these elements, it
     means the element contains more than one child, which
     means it actually is what it says it is, e.g. "$x or $y"
```

```
      would be annotated as an <OrExpr> but not "'hello'" (even
      though it technically is an OrExpr the way grammar is defined.)
-->
<xsl:template mode="simplify"
              match="*[count(*) eq 1]
                      [local-name() = ( 'OrExpr'
                                      , 'AndExpr'
                                      , 'ComparisonExpr'
                                      , 'RangeExpr'
                                      , 'AdditiveExpr'
                                      , 'MultiplicativeExpr'
                                      , 'UnionExpr'
                                      , 'IntersectExceptExpr'
                                      , 'InstanceofExpr'
                                      , 'TreatExpr'
                                      , 'CastableExpr'
                                      , 'CastExpr'
                                      , 'UnaryExpr'
                                      , 'PathExpr'
                                      )]">
  <xsl:apply-templates mode="#current"/>
</xsl:template>
```

4.2. Step 2: Annotate the expressions

Once the XML parse tree is simplified, we can start to categorize expressions as being XPath-friendly or needing a sequence constructor. These are not mutually exclusive, which is to say that an expression can start off being annotated as a sequence constructor but have a sub-expression annotated as XPath, which in turn can have a sub-expression annotated as a sequence constructor, etc.

To demonstrate this, consider the following example:

```
$names := for $x in //names return <firstname>{$x}</firstname>;
```

The FLWOR expression itself doesn't use any non-XPath constructs, except in the `return` clause (a sub-expression). The annotate.xsl[11] stylesheet thus annotates the outer-level expression as XPath, using the `<c:XPATH>` tag:

```
<Carrot xmlns:c="http://evanlenz.net/carrot">
  <CarrotModule>
    <VarDecl>
      <TOKEN>$</TOKEN>
      <QName>
```

[11] https://github.com/evanlenz/Carrot/blob/master/compiler/annotate.xsl

```
  <FunctionName>
    <QName>names</QName>
  </FunctionName>
</QName> <TOKEN>:=</TOKEN>
<c:XPATH>
  <Expr>
    <FLWORExpr>
      <ForClause> <TOKEN>for</TOKEN>
        <ForBinding> <TOKEN>$</TOKEN>
          <VarName>
            <QName>
              <FunctionName>
                <QName>x</QName>
              </FunctionName>
            </QName>
          </VarName> <TOKEN>in</TOKEN>
          <ExprSingle>
            <PathExpr> <TOKEN>//</TOKEN>
              <AxisStep>
                <ForwardStep>
                  <AbbrevForwardStep>
                    <NodeTest>
                      <NameTest>
                        <QName>
                          <FunctionName>
                            <QName>names</QName>
                          </FunctionName>
                        </QName>
                      </NameTest>
                    </NodeTest>
                  </AbbrevForwardStep>
                </ForwardStep>
              </AxisStep>
            </PathExpr>
          </ExprSingle>
        </ForBinding>
      </ForClause> <TOKEN>return</TOKEN>
      <ExprSingle>
        <c:SEQUENCE_CONSTRUCTOR needs-helper="yes">
          <Constructor>
            <DirectConstructor>
              <DirElemConstructor> <TOKEN><</TOKEN>
                <QName>
                  <FunctionName>
                    <QName>firstname</QName>
                  </FunctionName>
```

```
                    </QName>
                    <TOKEN>></TOKEN>
                    <DirElemContent>
                      <CommonContent>
                        <c:XPATH>
                          <EnclosedExpr>
                            <TOKEN>{</TOKEN>
                            <Expr>
                              <VarRef>
                                <TOKEN>$</TOKEN>
                                <VarName>
                                  <QName>
                                    <FunctionName>
                                      <QName>x</QName>
                                    </FunctionName>
                                  </QName>
                                </VarName>
                              </VarRef>
                            </Expr>
                            <TOKEN>}</TOKEN>
                          </EnclosedExpr>
                        </c:XPATH>
                      </CommonContent>
                    </DirElemContent>
                    <TOKEN></</TOKEN>
                    <QName>
                      <FunctionName>
                        <QName>firstname</QName>
                      </FunctionName>
                    </QName>
                    <TOKEN>></TOKEN>
                  </DirElemConstructor>
                </DirectConstructor>
              </Constructor>
            </c:SEQUENCE_CONSTRUCTOR>
          </ExprSingle>
        </FLWORExpr>
      </Expr>
    </c:XPATH>
  </VarDecl>
  <Separator>
    <TOKEN>;</TOKEN>
  </Separator>
 </CarrotModule>
</Carrot>
```

All is well in XPath mode until the annotator comes across the `Constructor` instance in the parse tree. It recognizes that as a non-XPath sub-expression and thus annotates the result with `<c:SEQUENCE_CONSTRUCTOR>`. The "needs-helper" attribute is a signal to the compiler that it will need to create an auxiliary function to implement the given sequence constructor.

It then switches back to XPath mode once again when encountering the `EnclosedExpr` instance.

Depending on the context in the stylesheet, there will sometimes be a bias toward XPath mode and sometimes a bias toward sequence constructor mode. For example, at the top level of a function definition, there's no choice but to start with a sequence constructor (`<xsl:function` doesn't have a "select" attribute). But when we're generating a global `<xsl:variable>` element, we don't want to assume a bias toward sequence constructors if we want to generate idiomatic code. For example, we'd rather generate this:

```
<xsl:variable name="foo" select="3"/>
```

than the equivalent version using a sequence constructor:

```
<xsl:variable name="foo" as="item()*">
  <xsl:sequence select="3"/>
</xsl:variable>
```

The advantage of separating out the expression categorization (step 2) from XSLT generation (step 3) is that the two can be tweaked independently. For example, if we wanted to continue improving the compiler's output in generating more idiomatic XSLT with regard to expression categorization, we can add more scenarios to annotate.xsl without affecting the subsequent XSLT generation stage.

How do we decide whether an expression needs to use a sequence constructor? It depends on whether or not it's a valid XPath expression. Here's an excerpt from annotate.xsl, which shows a list of all the non-XPath XQuery expressions:

```
<xsl:template mode="requires-sequence-constructor" match="*"/>
<xsl:template mode="requires-sequence-constructor" match=" TypeswitchExpr
                                                         | ValidateExpr
                                                         | ExtensionExpr
                                                         | OrderedExpr
                                                         | UnorderedExpr
                                                         | Constructor
                                                         | ▶
QuantifiedExpr[TypeDeclaration]
                                                         | RulesetCall
                                                         | TextNodeLiteral
                                                       | FLWORExpr[ LetClause
                                                                  | ▶
WhereClause
                                                                  | ▶
```

```
OrderByClause
                                                                    | ▶
ForClause/ForBinding/TypeDeclaration
                                                                    | ▶
ForClause/ForBinding/PositionalVar
                                                                    ]">
    <xsl:sequence select="true()"/>
  </xsl:template>
```

I came up with this list by clicking through the hyperlinked grammars in the XPath 2.0 and XQuery 1.0 specifications, respectively (opened in adjacent browser tabs). Wherever the XQuery grammar deviated from the XPath 2.0 grammar, I noted the exception. I could rely on this list because of the fact that "Any expression that is syntactically valid and executes successfully in both XPath 2.0 and XQuery 1.0 will return the same result in both languages." [1] Even though the `FLWORExpr` production is not listed in the XPath 2.0 grammar, `ForExpr`, a subset of `FLWORExpr`, *is* allowed. The above pattern delineates the subset of `FLWORExpr` that's unique to XQuery (and thus requires a sequence constructor using `<xsl:for-each>`).

4.3. Step 3: Generate the XSLT

Once the parse tree has been simplified and annotated, it's time to convert it to XSLT. Let's walk through an example of how this is done. The following template rule converts a Carrot rule definition to an XSLT template rule in the output:

```
<xsl:template match="RuleDecl">
  <out:template match="{c:xpath(Pattern)}">
    <xsl:apply-templates select="ModeName[1],
                                 Priority,
                                 RuleParamList/ParamWithDefault"/>
    <xsl:apply-templates select="Expr"/>
  </out:template>
</xsl:template>
```

For the `Pattern`, we cannot safely output it unchanged into the resulting "match" attribute. That's because the pattern might have a predicate containing a non-XPath sub-expression (such as an element constructor, full FLWOR expression, etc.). For that reason, we need to process it further, being sure to handle any nested expressions that are categorized as sequence constructors. We defer this processing to the `c:xpath()` function. We do the exact same thing elsewhere when processing expressions that need to be output as XPath in the result. For example, here's the rule for processing Carrot ruleset invocations:

```
<xsl:template match="RulesetCall">
```

```
  <out:apply-templates select="{c:xpath(Expr)}">
    <xsl:apply-templates select="ModeName,
                                 RulesetCallParamList/InitializedParam"/>
  </out:apply-templates>
</xsl:template>
```

Now let's look at the `c:xpath()` function itself:

```
<!-- Convert a parsed Carrot expression to XPath -->
<xsl:function name="c:xpath" as="xs:string">
  <xsl:param name="carrot-expr"/>
  <xsl:variable name="xpath" as="xs:string">
    <xsl:value-of>
      <xsl:apply-templates mode="xpath" select="$carrot-expr"/>
    </xsl:value-of>
  </xsl:variable>
  <xsl:sequence select="$xpath"/>
</xsl:function>
```

Applying templates in the "xpath" mode makes use of XSLT's built-in template rule for elements and text nodes (process children, and copy text, respectively). So, the default, is to copy the expression text through unchanged—which works great for XPath expressions. It's only when it encounters a sequence constructor annotation that it needs to do something different, namely call out to a helper function:

```
<!-- Call a helper function from within XPath -->
<xsl:template mode="xpath" match="c:SEQUENCE_CONSTRUCTOR">
  <xsl:value-of select="c:helper-name(.)"/>
  <xsl:text>(</xsl:text>
  <!-- Pass in all the necessary context -->
  <xsl:apply-templates mode="helper-arg" select="c:helper-params(.)"/>
  <xsl:text>)</xsl:text>
</xsl:template>
```

Elsewhere, we generate the helper functions at the top level of the stylesheet:

```
<!-- Delegate non-XPath expressions to stylesheet helper functions -->
<xsl:template mode="helper" match="c:SEQUENCE_CONSTRUCTOR[@needs-helper]">
  <out:function name="{c:helper-name(.)}">
    <xsl:apply-templates mode="helper-param" select="c:helper-params(.)"/>
    <xsl:apply-templates select="."/>
  </out:function>
  <!-- Check for sub-expressions that also require help -->
  <xsl:apply-templates mode="#current"/>
</xsl:template>
```

And thus we see another great reason to categorize expressions ahead of time. Since we need to generate both the helper functions and the calls to them, separately, in different places in the result, the annotations save us from having to detect the need for them more than once. That work has already been done.

The rest of the task in generating XSLT is translating from various Carrot/XQuery constructs to XSLT. The following table shows some of the mappings from XQuery-specific constructs:

Table 3. Some mappings from XQuery to XSLT

In Carrot:	**Translated to:**
for	`<xsl:for-each>` with `<xsl:variable>`
let	`<xsl:variable>`
where	`<xsl:if>` or pulled into a predicate when feasible
order by	`<xsl:sort>`, but only under certain circumstances; see [2]
typeswitch	`<xsl:choose>`

5. Conclusion

This paper has described the current implementation of the Carrot parser and the implementation progress of the Carrot compiler. If you're interested in participating on this project in any way, join the [9]

A. Introduction to Carrot

A.1. Background and influences

Carrot is not the first XSLT-inspired project to provide a shorter syntax than XSLT itself. Syntax shorthands have included Paul Tchistopolskii's XSLScript[1], Sam Wilmott's RXSLT[2], and another project called XSLTXT[3]. Although none of these projects provided direct inspiration for Carrot, they all address one of the same desires that Carrot addresses: being able to program in XSLT more concisely. However, unlike these projects, Carrot addresses more than XSLT's verbosity. It also addresses XSLT's limited composability. For example, in XSLT you can't include

[1] http://markmail.org/message/niumiluelzho6bmt
[2] http://www.wilmott.ca/rxslt/rxslt.html
[3] http://savannah.nongnu.org/projects/xsltxt

an element constructor in a path expression (like you can in XQuery and Carrot) or apply templates inside a path expression (which you can uniquely do in Carrot).

A more direct inspiration was James Clark's proposal for Unifying XSLT and XQuery element construction[4]. Written during the early days of the W3C activity on XQuery, that proposal suggested that XQuery and XSLT language constructs could be used interchangeably if XQuery used an XML-based syntax (via a simple document element wrapper). As we now know, things didn't turn out that way. Carrot takes essentially the opposite approach. Rather than make XQuery use an XML-based syntax like XSLT's, make XSLT (Carrot, actually) use a non-XML-based syntax like XQuery's.

Carrot is also inspired by Haskell's syntax, which defines functions using pattern-matching and an equation-like syntax.

A.2. Introduction by example

Carrot is best understood by example. Here's an example of XSLT's syntax for a template rule (henceforth "rule"):

```
<xsl:template match="para">
  <p>
    <xsl:apply-templates/>
  <p>
</xsl:template>
```

In Carrot, you'd write the above rule like this:

```
^(para) := <p>{^()}</p>;
```

There are a few things to note about the above. To define a rule in Carrot, you use the same operator that XQuery uses for binding variables (:=). Everything on the right-hand side up to the semi-colon is an expression in Carrot. An expression in Carrot is simply an XQuery expression, plus some extensions. In this case, the expression is using the extended syntax for invoking rules:

```
^()
```

which is short for:

```
^(node())
```

just as:

```
<xsl:apply-templates/>
```

is short for:

```
<xsl:apply-templates select="node()"/>
```

[4] http://www.jclark.com/xml/construct.html

All rules belong to a *ruleset* (equivalent to a "mode" in XSLT). The above examples use the unnamed ruleset (there's just one of these). Here's an example that belongs to a ruleset named "toc":

```
^toc(section) := <li>{ ^toc() }</li>;
```

The above is short for:

```
<xsl:template match="section" mode="toc">
  <li>
    <xsl:apply-templates mode="toc"/>
  </li>
</xsl:template>
```

Here's the identity transform in Carrot:

```
^(@*|node()) := copy{ ^(@*|node()) };
```

This recursively copies the input to the output, one node at a time.

Here's a Carrot script that creates an HTML document with dynamic content for its title and body, converting <para> elements in the input to <p> elements in the output:

```
^(/) :=
 <html>
   <head>
     { /doc/title }
   </head>
   <body>
     { ^(/doc/para) }
   </body>
 </html>;

^(para) := <p>{ ^() }</p>;
```

As a comparison, here's what you'd have to write if you were using regular XSLT:

```
<xsl:transform version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <html>
      <head>
        <xsl:copy-of select="/doc/title"/>
      </head>
      <body>
        <xsl:apply-templates select="/doc/para"/>
      </body>
    </html>
  </xsl:template>
```

```
  <xsl:template match="para">
    <p>
      <xsl:apply-templates/>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

Just as in XSLT, rules in Carrot can be associated with more than one mode. In XSLT, this template rule belongs to two modes:

```
<xsl:template mode="foo bar" match="bang"/>
```

Here's the equivalent rule in Carrot, belonging to two rulesets:

```
^foo|bar(bang) := ();
```

A.3. Carrot definitions

A Carrot module consists of a set of unordered *definitions*. Unlike XQuery, there is no distinction between main modules and library modules. Likewise, a Carrot module has no "body." Instead, there are only definitions. Carrot is more like XSLT in this regard. Also unlike XQuery, Carrot modules need not be associated with a namespace.

There are three kinds of definitions in Carrot:

- global variables,
- functions, and
- rules.

A.3.1. Global variables

A global variable definition is very similar to a variable declaration in XQuery, except that you don't need the "declare variable" verbiage. Whereas in XQuery you would write:

```
declare variable $foo := "a string value";
```

In Carrot you would instead write:

```
$foo := "a string value";
```

A.3.2. Functions

A function definition is just like a function declaration in XQuery except that you don't need the "declare function" verbiage and, instead of curly braces, you use the

same binding operator (:=) as a variable definition. For example, whereas in XQuery, you would declare functions like this:

```
declare function my:foo() { "return value" };
declare function my:bar($str as xs:string) as xs:string { upper-case($str) };
```

In Carrot, you would instead write:

```
my:foo() := "return value";
my:bar($str as xs:string) as xs:string := upper-case($str);
```

Why not just use the regular XQuery syntax? Two reasons: conciseness (lower signal-to-noise ratio) and consistency (with the other two types of definitions).

A.3.3. Rules

The third type of definition is a rule. This corresponds to a template rule in XSLT. For example, this rule matches any element node (*):

```
^foo(*) := "return value";
```

Unlike a function definition, the "argument" of a rule definition ("*" in the above case) is *not* an (optional) formal parameter list; instead it is a required pattern (as XSLT defines a pattern). Thus, it's illegal to have an empty set of parentheses in a rule definition:

```
^foo() := "return value"; (: NOT LEGAL :)
```

Note the asymmetry with ruleset invocations, where it *is* legal to call `^foo()`, which is short for `^foo(node())`.

Of course, rules can also have parameters (just as template rules can have parameters in XSLT). The syntax for declaring these is very similar to an XQuery function parameter list, except that it comes after the pattern and is separated from the pattern by a semicolon:

```
^foo(* ; $str as xs:string) := concat($str, .);
```

Carrot also supports tunnel parameters, as in XSLT. To indicate a tunnel parameter, you add the keyword "tunnel" before the parameter:

```
^foo(* ; tunnel $str as xs:string) := concat($str, .);
```

Unlike XQuery functions, parameters in a rule are identified by name, not position. Thus the syntax for passing them looks very similar to how they are declared, and the order of parameters is insignificant. The following expression applies the "foo" ruleset to the context node, passing the tunnel parameter $str with the value "Hello":

```
^foo(. ; tunnel $str := "Hello")
```

What about conflict resolution among multiple matching rules? Carrot behaves the same as XSLT: rules with higher import precedence win, followed by rules with

higher priority. Default priority is based on the syntax of the pattern, just as in XSLT. You can also specify the priority explicitly (right before the binding operator :=), as in the first rule of this example, which explicitly sets the priority to 1:

```
^author-listing( author[1]      ) 1 :=          ^();
^author-listing( author         )   := ", "    , ^();
^author-listing( author[last()] )   := " and " , ^();
```

A.4. Carrot expressions

The right-hand side of a Carrot definition, whether it be a variable, function, or rule, is a Carrot expression. The context for the expression evaluation is the same as it is for sequence constructors within a template rule in XSLT. For example, the context node is the node matched by the rule's pattern.

A Carrot expression is an XQuery expression with some extensions:

- ruleset invocations — `^mode(nodes)`
- shallow `copy{…}` constructors
- text node literals — `` `my text node` ``

Let's look at each of these extensions in turn and the rationale behind each one.

A.4.1. Ruleset invocations

Ruleset invocations (i.e., "apply-templates" in XSLT) are largely Carrot's *raison d'etre*. They are not possible in XQuery; thus, the extension is required. Not only that, but XSLT can't invoke rules (apply templates) in an expression either. In Carrot, all definitions are bound to an expression, so the only way to "do" anything is to write an expression. (Unlike XSLT, Carrot does not make a distinction between "instructions" and "expressions"; everything is an expression.)

A.4.2. Shallow copy constructors

Shallow copy constructors are possible in XSLT but not XQuery. The difference between a copy constructor and using an XQuery element constructor is that, in the latter case, the namespace context comes from the query rather than the source document. XQuery allows you to perform deep element copies from the source document, but not shallow copies. Without this ability, modified identity transforms are impractical in XQuery. The semantics of Carrot's copy constructor are essentially the same as XSLT's <xsl:copy> instruction. For example, when the context node is not an element node, it behaves the same as if a deep copy were being performed.

Note
XSLT 2.1/3.0 promises to add a "select" attribute to <xsl:copy> to make it convenient to perform a shallow copy of a node other than the context node. This is largely unnecessary in Carrot, since copy constructors can be easily composed within an expression, making it convenient to write, for example, `foo/copy{…}`.

A.4.3. Text node literals

Carrot also adds text node literals, using the back-tick (`) for the delimiter. This extension may at first seem to be of minimal value, since XQuery already allows you to construct text nodes using `text{…}`, and strings using quotes (or apostrophes). However, in practice, text node literals will often be the preferred syntax, as the following examples should make clear. Consider the following template rules in XSLT:

```
<xsl:template mode="file-name" match="doc">doc</xsl:template>
<xsl:template mode="file-ext" match="doc">.xml</xsl:template>

<xsl:template match="/doc">
  <result>
    <xsl:apply-templates mode="file-name" select="."/>
    <xsl:apply-templates mode="file-ext" select="."/>
  </result>
</xsl:template>
```

In Carrot, you might naturally rewrite the above as follows:

```
^file-name(doc) := "doc";
^file-ext (doc) := ".xml";
^(/doc)         := <result>{ ^file-name(.), ^file-ext(.) }</result>
```

The problem is that this will produce an undesired result:

```
<result>doc .xml</result>
```

The extra space results because of the way in which sequences of atomic values are combined to make a text node in XQuery. Contiguous sequences of text nodes, on the other hand, are merged together without any intervening spaces, so you could fix things by using explicit text node constructors:

```
^file-name(doc) := text{"doc"};
^file-ext (doc) := text{".xml"};
```

The problem here is that it may be an edge case with a large syntactic cost if you want to cover your bases (six extra characters for every text node). If in 90% of cases, using a string will result in the exact same behavior as if you had used a text node, you will be strongly tempted as a user to use quotes instead of `text{…}` everywhere.

However, you will get bugs in the remaining 10% of your code because of the way sequences of strings are concatenated to make a text node in XQuery.

Whereas it's more verbose in XQuery to construct a text node (using `text{…}`) than it is to return a string (using quotes), it's more verbose in XSLT to return a string (using <xsl:sequence>) than it is to return a text node (using a literal text node in the stylesheet). Text node literals in Carrot address this imbalance by making it equally convenient to create text nodes and strings. Thus, we naturally rewrite our Carrot definitions to get the desired result, without having to think about whether this is an edge case or not:

```
^file-name(doc) := `doc`;
^file-ext (doc) := `.xml`;
```

The existence of text node literals makes it easy to follow a simple rule: use text node literals when you are constructing part of a result document; use string literals when you know you want to return a string.

A.4.4. Expression semantics

Expressions in Carrot, unless otherwise noted here, are assumed to have the same semantics as in XQuery. Carrot operates on exactly the same data model as XQuery 1.0 and XPath 2.0.

One exception is that namespace attribute declarations on element constructors in Carrot do not affect the default element namespace for XPath expressions. Carrot is more like XSLT in this regard, in that it makes a distinction between the default namespace for input documents and the default namespace for output documents ("xpath-default-namespace" in XSLT), thereby correcting what is arguably a design bug in XQuery.

A.4.5. What about xsl:for-each, xsl:for-each-group, etc.?

Given that XQuery expressions do not include everything that it's possible to do in an XSLT template rule, that begs the question: What do all the XSLT instructions get mapped to in Carrot? In many cases, Carrot simply does not have an analogue. In some cases, that's because XQuery already provides a different way to achieve the same use case. For example, <xsl:for-each> does not have a direct analogue in Carrot. For iteration over a sequence, you can use "for" expressions, or even just "/" when applicable. The following Carrot (and XQuery) expression constructs a new <bar> element for each <foo> element, rendering <xsl:for-each> unnecessary for this case: `foo/<bar/>`. Similarly, Carrot does not support <xsl:sort>. For sorting sequences in Carrot, you would instead use "order by", as in XQuery. Local variables are defined using "let" expressions. Etc.

The biggest area not currently addressed by Carrot—and which remains an open question—is how to perform grouping. There are a few answers to this question, not all mutually exclusive:

1. Extend Carrot to support grouping.
2. Import an XSLT 2.0 stylesheet when you need grouping.
3. Wait for grouping to be added to XQuery 3.0 expressions and use those.

At this stage, the operative answers to this question are #2 and #3.

Designing support for multiple output documents (corresponding to <xsl:result-document> in XSLT) and how it interacts with document{} node constructors is on my TODO list. (If you have ideas, I'd be happy to hear them.)

Bibliography

[1] Boag, Scott – Chamberlin, Don – Fernández, Mary F. - Florescu, Daniela – Robie, Jonathan – Siméon, Jérôme: XQuery 1.0: An XML Query Language (Second Edition). W3C Recommendation, 14 December 2010. http://www.w3.org/TR/xquery/#id-introduction

[2] Carlisle, David: XQ2XML: Transformations on XQueries. Slide #19 of XML Prague 2006 presentation. http://www.xmlprague.cz/2006/slides06/carlisle/dpc-prague2006-19.html

[3] Carlisle, David: XQ2XML project site. http://monet.nag.co.uk/xq2xml/

[4] Fokoue, Achille – Rose, Kristoffer – Siméon, Jérome – Villard, Lionel: Compiling XSLT 2.0 into XQuery 1.0. Paper at WWW2005. http://www2005.org/cdrom/docs/p682.pdf

[5] Rademacher, Gunther: REx Parser Generator. online utility http://www.bottlecaps.de/rex/

[6] Lenz, Evan: Carrot: An appetizing hybrid of XQuery and XSLT. Paper. Balisage 2011. http://balisage.net/Proceedings/vol7/html/Lenz01/BalisageVol7-Lenz01.html

[7] Lenz, Evan: Carrot: An appetizing hybrid of XQuery and XSLT. Slide presentation. Balisage 2011. http://www.slideshare.net/evanlenz/carrot-an-appetizing-hybrid-of-xquery-and-xslt

[8] Carrot: An appetizing hybrid of XQuery and XSLT. Project site. https://github.com/evanlenz/Carrot

[9] Carrot. Google Group site. http://groups.google.com/group/carrot-xml

[10] Tchistopolskii, Paul: XSLScript. Email announcement on xml-dev, 13 October 2000. http://markmail.org/message/niumiluelzho6bmt

[11] Wilmott, Sam: RXSLT. project site. http://www.wilmott.ca/rxslt/rxslt.html

[12] XSLTXT. project site. http://savannah.nongnu.org/projects/xsltxt

[13] Clark, James: Unifying XSLT and XQuery element construction. 27 May, 2001. http://www.jclark.com/xml/construct.html

Transform.xq

A Transformation Library for XQuery 3.0

John Snelson
MarkLogic Corporation
<john.snelson@marklogic.com>

Abstract

It has long been held that one of the use cases for which XSLT [3] excels over XQuery [1] is document transformation. With its central rule based template engine and extensibility features, specifying and customising a transformation are straightforward. Although XQuery users have achieved transformation capabilities using recursive functions and type switch expressions, these have been found lacking in the expressiveness of the pattern syntax, and the extensibility of the resulting transformation code.

XQuery 3.0 [2] provides many powerful new features which extend the boundaries of what can be accomplished in an XQuery program. This paper introduces Transform.xq, an XQuery 3.0 module which implements rule based transformations using higher order functions. It further extends this library with automatic construction of template "modes" using reflection extensions to examine the XQuery 3.0 annotations on functions designated for use as template rules.

1. Introduction

The power of XQuery 3.0 means that many things that used to fall firmly into the realm of language extensions are now possible in the language itself. The principle language mechanisms that provide this extensibility are higher order function support and function annotations.

The Transform.xq library uses these XQuery 3.0 features and others to address the shortcomings of XQuery when describing transformations. A look at its design will not only help XQuery programmers to take advantage of the power of declarative transformations, but will also equip them to create similarly powerful XQuery libraries - providing functionality that was previously only possible by extending the XQuery language itself.

2. Head First into XQuery 3.0

Since XQuery 3.0 is still very new to most people, this paper won't assume a good understanding of its features. Instead, it will adopt a "head's first" tutorial approach, and explain the features from XQuery 3.0 as they become relevant to the topic at hand.

2.1. Creating a Mode

```
let $mode := tfm:mode((
  tfm:rule("section/title", function($mode, $node) {
    <h2>{ $mode($node/node()) }</h2>
  }),
  tfm:rule("article/title", function($mode, $node) {
    <h1>{ $mode($node/node()) }</h1>
  }),
  tfm:rule("section", function($mode, $node) {
    <div>{ $mode($node/node()) }</div>
  }),
  tfm:rule("article", function($mode, $node) {
    <html>
      <head><title>{ $node/title/string() }</title></head>
      <body>{ $mode($node/node()) }</body>
    </html>
  })
))
return $mode(<article>...</article>)
```

The `tfm:mode()` function is a quintessential higher order function, both accepting a sequence of rules as functions, and returning a function itself. The function returned by `tfm:mode()` can be called with a node as argument in order to execute the transformation specified by the rules on that node.

The new type of expression passed into the `tfm:rule()` function calls is an inline function. It defines the parameter names and body of an anonymous function. That function is then returned as a value, and passed into `tfm:rule()` as an argument.

You can see that the function returned by `tfm:mode()` is assigned to the `$mode` variable, and that function is then called in the return clause, using the new but familiar trailing parentheses notation.

These two new expressions form the bulk of the higher order functions feature in XQuery 3.0. In addition you can reference a named function as an expression using its name followed by a hash (or pound if you're from the US) sign and the arity (the number of arguments the function accepts) like this: `tfm:mode#1`.

2.2. The Main Transform.xq Functions

```
declare function tfm:mode(
  $rules as (function(xs:string) as function(*)?)*
) as function(node()*) as item()* { ... };
```

Looking at the function signature for `tfm:mode()`, you can see some new SequenceType syntax that can be used to check the type of functions. The SequenceType `function(*)` matches any function, whist the more specific function types specify the parameter types and return type of the function.

The sequence of rules passed into `tfm:mode()` as an argument are actually functions from `xs:string` to `function(*)?`. The `tfm:rule()` function returns a special wrapper function that encapsulates the information about a rule, and returns that information when passed the correct string as an argument.

```
declare function tfm:rule(
  $pattern as xs:string,
  $action as function(
      function(node()*) as item()*,
      node()
    ) as item()*
) as function(xs:string) as function(*)?
{
  tfm:predicate-rule(tfm:pattern($pattern), $action)
};
```

The `tfm:rule()` function takes the pattern to match as a string, and the action to perform as a function. The action function itself takes the mode function, and the matched node as arguments. The pattern string is passed to `tfm:pattern()`, which compiles the pattern into a predicate function.

```
declare function tfm:predicate-rule(
  $predicate as function(node()) as xs:boolean,
  $action as function(
      function(node()*) as item()*,
      node()
    ) as item()*
) as function(xs:string) as function(*)?
{
  function($k as xs:string) as function(*)?
  {
    switch($k)
      case "predicate" return $predicate
      case "action" return $action
      default return ()
```

```
    }
};
```

Patterns are a useful special case for the more general predicate function. More complex matching behaviour that cannot be specified using the pattern syntax can be acheived using `tfm:predicate-rule()` and a custom predicate function. A rule is considered to match if the predicate function returns true - if it returns false or raises an error, then the rule does not match.

The `tfm:predicate-rule()` function returns an inline function that wraps the predicate and action. This will be returned by the new rule function when the strings "predicate" and "action" are passed in as arguments. This demonstrates the closure of the inline function - the variables `$predicate` and `$action` from the surrounding scope are available from the body of the inline function, and the inline function carries (or closes over) their values with it, even though the variables may not be in scope where the function is evaluated. This facility forms the basis for creating many new data structures with XQuery 3.0.

You can also see an example of another XQuery 3.0 expression, the switch expression. This behaves like typeswitch but chooses a branch depending on the value of the operand rather than its type.

3. Pattern Matching

The `tfm:pattern()` function takes a string containing a subset of XSLT 2.0 pattern syntax, and compiles it into a single predicate function. This subset currently excludes predicates and matching using an element or attribute's type, but there is opportunity to add these features in the future using calls to implementation specific eval functions. A grammar for the pattern syntax currently supported is included in Appendix A.

3.1. Parsing Patterns

To parse the pattern strings, Transform.xq uses a REx [5] generated parser. REx is an excellent parser generator by Gunther Rademacher, which has the option of targeting XQuery. The generated parser is then executed to create an XML parse tree - effectively the source string marked up with XML elements spanning every grammar production and token that was matched. This turns out to be a very effective method of parsing using XQuery - rather than executing specific XQuery code associated with each production (as is often done with parser generators), the compile stage essentially becomes a transformation of the parse tree document produced by the parser.

3.2. Compiling Patterns

```
declare %private function pat:compile-nametest($prev, $qn, $resolver)
{
  typeswitch($qn)
    case element(NCNameColonStar) return
      let $prefix := substring-before($qn,"*")
      let $ns :=
        namespace-uri-from-QName($resolver($prefix || "fake"))
      return function($n) {
        namespace-uri($n) eq $ns and
          (empty($prev) or $prev($n/..))
      }
    case element(StarColonNCName) return
      let $localname := substring-after($qn,"*:")
      return function($n) {
        local-name($n) eq $localname and
          (empty($prev) or $prev($n/..))
      }
    case element(Star) return
      function($n) {
        empty($prev) or $prev($n/..)
      }
    case element(QName) return
      let $name := $resolver($qn)
      return function($n) {
        node-name($n) eq $name and
          (empty($prev) or $prev($n/..))
      }
    default return error(xs:QName("tfm:BADNAMETEST"),
      "Invalid name test: " || $qn)
};
```

At the heart of the transformation to compile the pattern, the `pat:compile-nametest()` function creates functions which check the element or attribute has the correct name. The predicate function for the previous step in the path is provided in the `$prev` argument. The parse tree node in `$qn` contains the nametest, and the function uses the typeswitch style of XQuery transformation [1], to select the correct inline function for it.

The inline functions follow a common pattern: After checking for the correct name, they then call the previous step's predicate function with the parent node if the function exists. This chains the special purpose name testing functions together

[1] Hopefully the last time I have to write this style of transformation in XQuery - alas it's not possible to bootstrap Transform.xq with itself in this case!

into a single function, using the closure mechanism. Suitably clever XQuery implementations will be able to partially specialize the functions produced in this way such that they produce optimized function bodies equivalent to a hand written function.

4. Modes

```
declare function tfm:mode(
  $rules as (function(xs:string) as function(*)?)*
) as function(node()*) as item()*
{
  let $map := fold-left(tfm:add-rule#2, map:create(), $rules)
  return tfm:run-mode($map,?)
};
```

The `tfm:mode()` function returns a single function which applies the mode. This is created using partial application to specialize the private `tfm:run-mode()` function with the processed rules. Partial application specifies some arguments for a function, and uses the question mark ("?") syntax to represent missing arguments. It results in a function with reduced arity, which accepts the missing arguments when called.

4.1. Rule Matching Optimization

The mode is executed by searching the rules for one whose predicate matches the node being transformed. Done naively this would result in a linear search through the rules for the one which applies. A common optimization in XSLT engines is to store template rules by the type of node which they match. This allows the set of potentially matching rules to be narrowed quickly.

In Template.xq, this optimization is performed by looking at the type of the argument accepted by the predicate function. The pattern parser is careful to return a predicate function with the most specific argument type it can, so that the rule is optimized well. Custom predicate functions can also take advantage of this optimization by declaring their argument to be one of the primary node types (element(), attribute(), etc.).

```
declare %private function tfm:add-rule($map,$rule)
{
  let $predicate := $rule("predicate")
  return
    if(not($predicate instance of function(*))) then
      error(xs:QName("tfm:BADPREDICATE"),
        "The predicate should be a function")
    else if(function-arity($predicate) ne 1) then
```

```
        error(xs:QName("tfm:BADPREDICATE"),
          "The predicate should have arity 1")
      else
        let $map := if($predicate instance of
          function(element()) as xs:boolean)
            then tfm:add($map,"element",$rule) else $map
        let $map := if($predicate instance of
          function(attribute()) as xs:boolean)
            then tfm:add($map,"attribute",$rule) else $map
        let $map := if($predicate instance of
          function(document-node()) as xs:boolean)
            then tfm:add($map,"document",$rule) else $map
        let $map := if($predicate instance of
          function(comment()) as xs:boolean)
            then tfm:add($map,"comment",$rule) else $map
        let $map := if($predicate instance of
          function(text()) as xs:boolean)
            then tfm:add($map,"text",$rule) else $map
        let $map := if($predicate instance of
          function(processing-instruction()) as xs:boolean)
            then tfm:add($map,"pi",$rule) else $map
        return $map
};
```

When a function `$f` is matched against a function type, the return type of `$f` must be a subtype of the expected return type. However, the argument types of `$f` must be supertypes of their expected types. This is because a function is logically allowed to be more permissive in what it accepts compared to what it is expected to accept.

This means that if `$f` has the signature `function(node()) as xs:boolean` it will match the function types `function(element()) as xs:boolean`, `function(attribute()) as xs:boolean`, and `function(node()) as xs:boolean` (to name a few). This fact is used in the `tfm:add-rule()` function to find all predicates that will match each of the permissable node types.

Having determined the types of nodes a predicate function will match, rules are stored in a map indexed by node type, in order to quickly narrow the set of applicable rules to test.

4.2. For the Lack of Maps

The previous optimization relies on the ability to update and retrieve a set of rules by the type of node that they match. This requires some kind of map facility in the language.

Sadly the XSLT 3.0 maps feature arrived too late to make it into XQuery 3.0. What's a standards conformant library supposed to do? Implement maps using higher order functions, or course! That's exactly what the RBTree.xq library [6] does.

```
declare function map:create(
) as function() as item()*
{ ... };

declare function map:put(
  $map as function() as item()*,
  $key as item(),
  $value as item()*
) as function() as item()+
{ ... };

declare function map:get(
  $map as function() as item()*,
  $key as item()
) as item()*
{ ... };
```

Using a similar closure technique to that used by `tfm:predicate-rule()` to wrap rules, RBTree.xq provides an efficient implementation of an immutable red-black tree [7], and uses that to implement a general purpose map data structure in pure XQuery 3.0.

4.3. Extending Modes

One of the powerful features of XSLT is the way that modes can be extended without altering the original stylesheet. Since there is an extensible map data structure at the heart of a mode function, it is also possible to extend Transform.xq mode functions.

```
declare %private variable $tfm:magic as element() := <magic/>;

declare function tfm:extend-mode(
  $mode as function(node()*) as item()*,
  $rules as (function(xs:string) as function(*)?)*
) as function(node()*) as item()*
{
  let $map := $mode($tfm:magic)
  let $map := fold-left(tfm:add-rule#2, $map, $rules)
  return tfm:run-mode($map,?)
};
```

A special private global variable is defined, `$tfm:magic`, which can be passed to a mode function in order to return the map of rules that it contains. This takes advantage of the fact that constructed elements each have a unique identity by matching the mode function's input nodes against `$tfm:magic` using the `is` operator. It is then a simple matter of using the original mode's map as the base case for constructing the extended mode's map.

5. Using Annotations to Create Modes

Wouldn't it be nice to make the definition of Transform.xq modes more like XSLT? It is much simpler to be able to declare template rules at the top level, without having to go through the process of constructing a sequence of rules, and then constructing a mode function from them. XQuery 3.0 function annotations open up an opportunity to do just that.

```
declare %tfm:rule("html","article/title",10)
function local:html1($mode, $node)
{
  <h1>{ $mode($node/node()) }</h1>
};

declare %tfm:rule("html","title")
function local:html2($mode, $node)
{
  <h2>{ $mode($node/node()) }</h2>
};

declare %tfm:rule("html","section")
function local:html3($mode, $node)
{
  <div>{ $mode($node/node()) }<div>
};

declare %tfm:rule("html","article")
function local:html4($mode, $node)
{
  <html>
    <head><title>{ $node/title/string() }</title></head>
    <body>{ $mode($node/node()) }</body>
  </html>
};

let $mode := tfm:named-mode("html")
return $mode(<article>...</article>)
```

Function annotations start with a "%" followed by a QName, and then optionally by a sequence of string or numeric literals contained in parentheses. XQuery 3.0 defines the `%private` annotation itself, but allows implementations and users to add their own annotations to functions as well.

Transform.xq uses the `%tfm:rule` annotation to declare a mode name, pattern, and optional numeric priority on an action function defined in a query prolog. The `tfm:named-mode()` function can then be used to return the mode function, given the mode name as argument. The priority field is use to order the rules, with a higher priority rule being tried for a match before a lower priority rule.

5.1. On Reflection

Unfortunately the XQuery 3.0 specification doesn't contains any reflection capabilities to allow named function discovery and inspection of annotations. However, given modest extension functions, we can easily implement `tfm:named-mode()`.

```
declare function tfm:named-mode(
  $name as xs:string
) as function(node()) as item()*
{
  tfm:mode(
    for $f in xdmp:functions()
    where xdmp:annotation($f, xs:QName("tfm:mode")) = $name
    let $predicate := xdmp:annotation($f, xs:QName("tfm:pattern"))
    return tfm:rule($predicate, $f)
  )
};
```

The MarkLogic specific experimental function `xdmp:functions()` can be used to return every named (top-level) function that is known about anywhere in the currently executing query, whilst `xdmp:annotation()` will return an annotation value from it's function argument, given a name as an `xs:QName`. These two simple functions are sufficient to implement the `tfm:named-mode()` funtionality, as well as a great many other interesting uses for function annotations.

5.2. Platform Independence

The presence of such implementation specific functions in an XQuery module would mean that the module could not be executed on a different XQuery implementation when using XQuery 1.0. Historically, this has been a major headache to anyone hoping to implement a cross-platform library in XQuery. However, new XQuery 3.0 functionality gives us a way to write such libraries and still use platform specific functions when available.

```
declare %private function tfm:functions(
) as function() as function(*)*?
{
  (
    function-lookup(
      fn:QName("http://marklogic.com/xdmp","xdmp:functions"),0)
  )[1]
};
```

The `fn:function-lookup()` function takes the name of a function and it's arity as arguments and returns the function if it is in-scope, or the empty sequence if it doesn't exist. This provides both a way to check if the function exists, and a way to call that function or recover gracefully. Transform.xq uses this technique to adapt to work with difference platform specific reflection APIs, or to return a useful error message otherwise.

6. Future Enhancements

There's plenty of scope for expanding this library in the future. I'd like to implement predicates containing numeric literals, simple path expressions, and comparisons. I don't have the appetite to implement a full XPath 2.0 interpreter in XQuery itself, but using the `fn:function-lookup()` trick I could implement any predicate expression using implementation specific eval functions. Another possibility that struck me as interesting was the possibility of implementing CSS selectors as an alternative pattern syntax.

7. Conclusion

XQuery 3.0 includes many powerful features that will vastly increase the scope of what can be achieved using the language. Transform.xq is a simple to use yet powerful library that significantly extends the capabilities of XQuery. However the techniques the library uses have the potential to be used for the creation of any number of new and as-yet unimagined libraries.

Transform.xq is available under the Apache License v2 from GitHub[2].

A. EBNF for the Pattern Syntax

This appendix uses the same notation as the XQuery specification [2]. Non-terminals not explicitly defined by this paper are references to non-terminals in the XQuery grammar.

[2] http://github.com/jpcs/transform.xq

```
Pattern ::= PathPattern ('|' PathPattern)*
PathPattern ::= ('/' | '//')? RelativePathPattern
RelativePathPattern ::= PatternStep (('/' | '//') PatternStep)*
PatternStep ::= (ChildAxis | AttributeAxis)? NodeTest
ChildAxis ::= 'child' '::'
AttributeAxis ::= 'attribute' '::' | '@'

NodeTest ::= KindTest
           | NameTest
NameTest ::= QName
           | NCNameColonStar
           | StarColonNCName
           | Star
KindTest ::= DocumentTest
           | ElementTest
           | AttributeTest
           | PITest
           | CommentTest
           | TextTest
           | AnyKindTest

DocumentTest ::= 'document-node' '(' ')'
AttributeTest ::= 'attribute' '(' ( QName | Star )? ')'
ElementTest ::= 'element' '(' ( QName | Star )? ')'
AnyKindTest ::= 'node' '(' ')'
TextTest ::= 'text' '(' ')'
CommentTest ::= 'comment' '(' ')'
PITest ::= 'processing-instruction' '('
  ( NCName | StringLiteral )? ')'
```

Bibliography

[1] *XQuery 1.0: An XML Query Language (Second Edition)*[1].

[2] *XQuery 3.0: An XML Query Language*[2].

[3] *XSL Transformations (XSLT) Version 2.0*[3].

[4] *Carrot: An appetizing hybrid of XQuery and XSLT.*[4]. Evan Lenz.

[5] *REx Parser Generator.*[5]. Gunther Rademacher.

[1] http://www.w3.org/TR/2010/REC-xquery-20101214/
[2] http://www.w3.org/TR/2011/WD-xquery-30-20110614/
[3] http://www.w3.org/TR/2007/REC-xslt20-20070123/
[4] http://www.balisage.net/Proceedings/vol7/html/Lenz01/BalisageVol7-Lenz01.html
[5] http://www.bottlecaps.de/rex/

[6] *RBTree.xq: A red/black tree implemented using XQuery 3.0 higher order functions.*[6]. John Snelson.

[7] *Red-Black Trees in a Functional Setting*[7]. Chris Okasaki. Journal of Functional Programming, 9(4):471-477, July 1999.

[6] https://github.com/jpcs/rbtree.xq
[7] http://www.eecs.usma.edu/webs/people/okasaki/pubs.html#jfp99

Building Bridges from Java to XQuery

A new approach to invoke XQuery functions from Java as if they were Java methods.

Charles Foster
<charles@xqj.net>

Abstract

The Java ecosystem surrounding relational databases has enjoyed years of improvements such as a standard means to invoke stored SQL code from JDBC and time-saving object relational mapping (ORM) frameworks such as Hibernate.

For Java programmers working with XQuery [1] and XML databases, there is the XQuery API for Java (XQJ); however it lacks the means to invoke stored XQuery code and currently provides no answer to the very successful ORM paradigm.

This paper introduces a novel, RPC style approach to calling XQuery from Java. Through the use of Java reflection, Java programmers are now able to invoke XQuery functions as if they were regular Java methods, with ease.

As for parameters and return types of both Java methods and XQuery functions, Java data types are mapped to XDM [2] data types according to the rules described in the XQJ specification. Additionally, POJOs [3] are mapped to XML which may save programmers a great deal of time, just as ORM technologies have done for SQL.

This paper proposes extensions to the XQJ APIs as well as discussing some initial implementations, namely MarkLogic, eXist and Sedna XQJ APIs.

Keywords: XQuery, Java, XQJ

1. Introduction

The XQuery API for Java provides a standard way to submit XQuery expressions to an XQuery processor and the means to handle their consequential result sequences.

The XQJ API is to XML databases and XQuery processors as the JDBC API is to relational databases and SQL, but lacks a standard way of invoking stored XQuery code, e.g. invokable XQuery modules or XQuery module functions.

Since JDBC 2.0, Java acquired a standard means to invoke stored procedures within relational databases. Stored procedures were stored directly in the database and thus can reduce compilation overhead, network traffic and enabled programmers to embed business logic as an API in the database.

XQuery being a dual query and functional programming language is perfectly placed to offer exactly the same service.

This paper proposes extensions to the XQJ interfaces which would allow programmers to call stored XQuery module functions as if they were regular native Java methods with great ease, introducing an RPC/Service like model to Java's relationship with XQuery processors. Example implementations of these XQJ extensions are available for MarkLogic, eXist and Sedna and can be found at the xqj.net website[1].

1.1.1. Summary of Contributions

- Introduces the notion of calling XQuery functions from Java as if they were native regular Java methods.
- Introduces an RPC/Service-like model to Java's relationship with XQuery processors.
- Introduces binding Java interfaces to XQuery library modules, enabling easy switching of XQuery implementations.

2. Related Work

2.1. Invoking XQuery Main Modules from Java with XCC

The MarkLogic XCC Driver[2] allows programmers to invoke an XQuery main module[3] which may be stored within a MarkLogic Server through its `ModuleInvoke` interface. Program behavior can be altered by binding Java values to XQuery external variables defined in the main module, the result of this invocation then produces an XQuery result sequence which must then be handled by the programmer.

While achieving the goal of invoking stored XQuery code, the approach may not be intuitive to the seasoned Java developer who is taking their first tentative footsteps in the exotic new world of XQuery, especially when they can see that XQuery functions are available which appear to bare some vague similarity, at least in concept, to Java methods. Many are left wondering if they could just invoke the XQuery functions directly, but ultimately accept the status quo.

2.2. Invoking XQuery Functions from Java with Saxon

Saxon APIs for instance, do have the ability to call XQuery functions directly from Java. (see Appendix A).

[1] xqj.net http://www.xqj.net/
[2] MarkLogic XCC http://developer.marklogic.com/products/xcc/5.0
[3] Main Module Definition http://www.w3.org/TR/xquery/#dt-main-module

Once obtaining an `XQueryExpression` (essentially an XQuery main module), the programmer can then get a handle on an XQuery function via obtaining a `UserFunction`.

From there, aside from the necessity to create a `Controller` instance, the XQuery function can be invoked by using the `UserFunction`'s `call` method, but this is not entirely straight-forward.

If the XQuery function accepts parameters, regular Java values must first be converted into a descendant of `ValueRepresentation` which is responsible for describing XDM data.

The result of calling the XQuery function from Java will also return a `ValueRepresentation` which the programmer must then handle and convert back into a regular Java value or values.

This definitely achieves the goal of invoking XQuery functions from Java. However, this approach requires quite a degree of boilerplate code. Conversion between XDM and Java data types adds complexity and thus is not entirely programmer friendly.

2.3. Redstone XML RPC Library and Java Proxies

Greger Olsson's XML-RPC implementation [4] has a novel approach to invoking XML-RPC web services from Java. Like both XQJ and JAXB specifications, it has default mapping rules for converting between Java data types and XML-RPC data types. Most importantly, through the use of Java proxies [5], an XML-RPC web service can be defined as a regular Java interface. So Java programmers can invoke XML-RPC web services as if they were Java methods.

3. Invoking XQuery Functions from Java

3.1. Defining the approach

XML in Java is a second class citizen, neither XML or any of the XDM data types are native data types as they are in XQuery.

So far, Java APIs for parsing, processing and transforming XML have been a little inelegant, so approaching the problem with an unconventional approach may not be such a bad idea.

Conceptually, XQuery functions bare some resemblance to Java methods.

This paper discusses using a regular Java interface as a facade for an arbitrary XQuery library module[4], where by each Java interface method relates directly to an XQuery library module's function.

[4] Library Module Definition http://www.w3.org/TR/xquery/#dt-library-module

Java method signatures are mapped to XQuery function signatures by adhering to the Java / XDM data type mapping rules defined in the XQuery API for Java Specification [6].

Invoking a Java interface method would in turn invoke the according XQuery function, marshalling Java parameter values into XDM values as necessary. The XQuery result would then be unmarshalled back into a Java value that is compatible with the Java method's return type.

An implementation of this approach would likely need to use Java reflection, specifically dynamic proxies [5]. The MarkLogic, eXist and Sedna XQJ implementations found at xqj.net all use Java reflection to achieve this. However, any XQJ implementation could also implement these interfaces.

3.2. A Simple Example

3.2.1. XQuery Library Module

Consider the following very simple XQuery library module, which contains three very simple functions.

```
module namespace eg = "http://www.example.com";

declare function eg:contains-any-of
  ($arg as xs:string?,
  $searchStrings as xs:string*) as xs:boolean
{
  some $searchString in $searchStrings
  satisfies fn:contains($arg,$searchString)
};

declare function eg:word-count
  ($arg as xs:string?) as xs:integer
{
  fn:count(fn:tokenize($arg, '\W+')[. != ''])
};

declare function eg:multiply
  ($a as xs:float, $b as xs:float) as xs:float
{
  $a * $b
};
```

3.2.2. Java Interface Definition

Now consider the following Java interface as a facade for the XQuery library module.

```
public interface Example
{
  public boolean containsAnyOf(String arg, String... searchStrings);

  public int wordCount(String arg);

  public float multiply(float a, float b);
}
```

This raises some points to address.

Firstly, Java method names are not identical to XQuery function names and with good reason. Aside from the fact that Java methods can not contain hyphen characters, there is a mismatch between Java method naming conventions [5] and commonly accepted XQuery function naming conventions[6]. As such, XQuery function names will be identical to Java method names, with one caveat.

When translating XQuery function names to Java method names, hyphens are dropped and instead cause the following letter to be capitalized. For example, the XQuery function name `hours-from-duration` would translate to the Java method name `hoursFromDuration`. The default name translation behavior, can also be overridden (see Overriding default behavior with Annotations).

Secondly, the XQuery function `eg:word-count` returns a value with an XDM type `xs:integer`, which leaves the possibility that the returned value may fall outside the primitive `int` range. However, this is allowed for simplicity's sake. It is the implementation's responsibility to convert the returned XDM value into an instance of the method's expected return type. If the implementation is unable to perform the mapping; an exception is thrown. In this instance, `int` could be replaced with numerous other return types, such as `BigInteger`, `long` or `XQItem`.

Thirdly, the `containsAnyOf` Java method signature includes a tailing varargs [7] parameter; this maps quite well to the `eg:contains-any-of` XQuery function which declares its last parameter with an XDM sequence type of `xs:string*`. The Java method has been declared this way out of convenience, but the method could also have been declared in any one of the following ways.

```
public boolean containsAnyOf(String arg, String[] searchStrings);
public boolean containsAnyOf(String arg, List<String> searchStrings);
public boolean containsAnyOf(String arg, XQItem[] searchStrings);
```

[5] Java Language Specification (Second Edition). 6.8.3 Method Names http://java.sun.com/docs/books/jls/second_edition/html/names.doc.html#34563

[6] XQuery Style Conventions. 5.2 Function declarations. http://xqdoc.org/xquery-style.pdf

[7] http://download.oracle.com/javase/1,5.0/docs/guide/language/varargs.html

3.2.3. Client Code

By using an XQJ implementation, which also implemented the interface defined in Appendix B, the following code would then be possible.

```
Example example =
  XQConnection2.createXQModuleProxy(
    "http://www.example.com",
    "/modules/example.xqy",
    Example.class
  );

boolean contains =
  example.containsAnyOf("abc", "bc", "xy");

int totalWords =
  example.wordCount("The quick brown fox jumps over the lazy dog");

float product =
  example.multiply(3f, 4f);
```

Some XML databases differentiate between stored XQuery modules by module namespaces and have no concept of module URIs[8]. By making the library module namespace URI and library module URI arbitrary, the XQuery implementation can be switched with ease, even at runtime, if necessary.

3.3. Overriding default behavior with Annotations

3.3.1. Java Method to XQuery Function Name Translation

Overriding default name translation behavior is achieved by using the `XQFunctionName` annotation which is declared in Appendix C.

Sometimes, default name translation behavior may cause unsuitable results. For instance, a programmer wishing to use the method `insertXMLContent` probably doesn't want its ultimate XQuery function endpoint to be called `insert-x-m-l-content`. Instead they would probably prefer `insert-xml-content`.

Consider the following example, which shows the means to override the default name translation behavior.

```
public interface MyXMLStore
{
  @XQFunctionName("insert-xml-content")
```

[8] Sedna Programmer's Guide, 2.5.5 Managing Modules http://www.sedna.org/progguide/ProgGuidesu8.html#x14-570002.5.5.

```
  public void insertXMLContent(Source xmlSource);
}
```

3.3.2. Java to XDM Data Type Mapping

There are rules to convert Java primitives and objects to XDM data types defined in section 14.2 of the XQJ specification [6]. Methods accepting regular Java values as parameters will map to XDM types for XQuery functions according to these rules.

Overriding default value mapping rules is achieved by using the `XQCastAs` annotation which is declared in Appendix D.

Consider the following XQuery function, which accepts a single parameter that has an XDM type of `document-node(element())`.

```
declare function insert-content(
  $uri as xs:string,
  $value as document-node(element()))
{
  xdmp:document-insert($uri, $value)
};
```

A Java interface which declares the method signature `public void insertContent(String uri, Source value)` would suffice. However, the user may wish to use a `String` as the content's value, e.g. `public void insertContent(String uri, String value)`, but by default the implementation may try to map `value` to an `xs:string` instance.

Consider the following example, which shows the means to override the default Java value to XDM mapping behavior.

```
public interface MyXMLStore
{
  public void insertContent(
    String uri,
    @XQCastAs("document-node(element())") String value
  );
}
```

3.4. Plain Old Java Objects Mapping

POJO [3] to XML mapping is not new concept, XStream[9] is a notable example of a Java package which does this very successfully.

As well as defining default mapping rules for regular Java and XDM data types which are included in Appendix E for reference; the XQJ specification [6] implies

[9] XStream http://xstream.codehaus.org/

that when mapping Java data types not covered in the specification (e.g. user-defined POJOs), behavior should be implementation-defined.

When invoking an XQuery function via a Java proxy where the XQuery function accepts parameters of XDM types such as `document-node(element())` or `element()`, regular POJO instances can be used instead of common readable XML sources, such as JAXP `Source` or StAX `XMLStreamReader` instances.

The XQJ implementation is expected to marshal the POJO instances to an XML structure for the XQuery function, where all the POJO's descendent member fields are also mapped to XML elements or attributes. Furthermore, the member fields are serialized according to the default mapping rules defined by the XQJ Specification (14.2) [6].

A Java interface method may also define its return type as a user-defined POJO, in this case the XQuery function must return a `document-node(element())` or a `element()` which when unmarshalled is compatible with the POJO definition. The XQJ implementation is responsible for unmarshalling the XML document to a new POJO instance. If the XQJ implementation is unable to unmarshal the returning XDM item into the defined POJO, an exception is thrown.

This approach allows Java to deal solely with regular Java types and POJOs while allowing XQuery to solely deal with atomic XDM data types and XML.

The approach of binding POJOs to XQuery function parameters and their return types is complementary to and does not exclude using regular Java representations of an XML readable source such as `Source`, `XMLStreamReader` or even `InputStream`. A `JAXBContext` may also be used, but mapping will then be controlled externally.

Just as object to relational mapping tools like Hibernate[10] has saved programmers time with relational databases, this technique used in conjunction with XQJ may save programmers' time with XML databases and XQuery. This approach doesn't intend to act as a Hibernate or ORM replacement, instead simply offers a practical solution for handling POJOs with XQuery and XML databases. Furthermore, the ORM programming paradigm does have its imperfections, largely due to an object-relational impedance mismatch[11]. The act of shredding tree like Java objects into separate relational tables and back again lacks a degree of elegance whereas the approach of serializing tree like Java objects into XML documents, which can then be stored as conceptual units of information in an XML database, is perhaps more intuitive and elegant than the ORM paradigm could ever hope to achieve.

3.4.1. A Simple POJO Use Case

Consider the following data-centric XML document

[10] Hibernate http://www.hibernate.org/

[11] Object-relational impedance mismatch http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch

```
<person>
  <first-name>John</first-name>
  <last-name>Smith</last-name>
  <date-of-birth>1970-04-12</date-of-birth>
  <phone-numbers>
    <number>123-789<number>
    <number>789-123<number>
  </phone-numbers>
</person>
```

Because the XML document's data model is data-centric, it can easily be mapped to Java objects.

Consider the following Java code which describes the data model of a person XML document.

```
class Person
{
  String firstName;
  String lastName;
  XMLGregorianCalendar dateOfBirth;
  PhoneNumbers phoneNumbers;
}

class PhoneNumbers
{
  public PhoneNumbers(String[] number) {
    this.number = number;
  }

  String[] number;
}
```

Instances of these POJO classes can now be used as Java method parameters, where the XQJ implementation is responsible for marshalling the POJO instance into XML for the XQuery function. Also, a Java interface method can return an instance of these POJO classes where the XQJ implementation is responsible for unmarshalling XML documents into POJO instances.

Now consider the following Java interface code, which acts as a facade for an arbitrary XQuery library module chosen at runtime.

```
public interface PersonStore
{
  public Person getPerson(String uri);

  public void insertPerson(String uri, Person person);
}
```

Note that the Java interface allows a POJO to be used as both a method parameter as well as a return type.

Consider the following XQuery library module implementation, which is compatible with the defined Java interface.

```
module namespace ps = "http://www.person-store.com";

declare function ps:get-person(
  $uri as xs:string) as element(person)?
{
  fn:doc($uri)/element()
};

declare function ps:insert-person(
  $uri as xs:string,
  $person as element(person))
{
  xdmp:document-insert($uri, $person)
};
```

Now that POJO definitions are created, along with a Java interface facade and an XQuery library module, consider the following Java code.

```
PersonStore personStore = XQConnection2.createXQModuleProxy(
  "http://www.person-store.com",
  "/modules/person-store.xqy",
  PersonStore.class);

Person john = new Person();

john.firstName = "John";
john.lastName = "Smith";
john.dateOfBirth = DatatypeFactoryImpl.newXMLGregorianCalendar("1970-04-12");
john.phoneNumbers = new PhoneNumbers(
  new String[] { "123-789", "789-123" }
);

personStore.insertPerson("/john-smith.xml", john);
john = null; // John is gone!

john = personStore.getPerson("/john-smith.xml"); // Welcome back John!
```

The above example creates a Person POJO instance and persists it into the database only to then to lose the local reference. A POJO instance that would have been equal to the lost instance[12] is then retrieved from the database.

[12] Users must supply the `equals(Object)` and `hashCode()` methods on their POJOs.

The XQJ implementation is not expected to keep track of node or object identity, while trivial for an IntraVM XQuery engine, such functionality in a client/server based implementation would be very difficult to achieve.

3.4.2. POJO Mapping Complications

This approach lends itself well to data-centric XML structures, but is incompatible with document-centric XML structures which contain mixed content.

While developing a proof of concept implementation for MarkLogic, eXist and Sedna XQJ APIs, XStream was used. XStream's default mapping of Java data types to XDM data types is incompatible with the mapping defined by the XQJ specification [6]. As such, custom XStream `Converter` code had to be written in order for the POJO/XML mapping of regular Java data types to be consistent with the mapping rules outlined in the XQJ specification.

4. Code Generation

Java interfaces as facades for XQuery library modules can be hand-crafted depending on how the programmer wishes data types to be mapped at runtime.

However, some XQuery library modules may have a sizeable amount of complex functions which would make hand-crafting Java interfaces not only irksome, but prone to human error.

Conversely, creating an XQuery library module implementation based on a complex Java interface may also prove tiresome and be susceptible to human error.

The `xquery2java` command line program performs static analysis of a user-defined XQuery library module, identifying function signatures and generating a compatible Java interface, which can then be used as a Java facade for the XQuery module.

The `java2xquery` command line program performs the inverse operation of `xquery2java`, by reading a Java interface source or compiled class file, then generating an XQuery library module stub where the XQuery functions do nothing, but have strictly typed signatures which are then ready to be fleshed out with actual working code.

5. Conclusion

This paper has shown the possibility of a simple, convenient and practical new approach to invoking XQuery code from the Java environment. It also proposes an answer to the successful ORM programming paradigm which relational databases have enjoyed for years and is well overdue in the XML/XQuery world.

The XQJ interface extensions proposed in this paper currently work within the MarkLogic, eXist and Sedna XQJ implementations, but there is no reason why other

vendors could not also follow suit. If the proposals outlined in this paper become successful, there may be an argument for adding them to the next official release of the XQJ specification.

6. Acknowledgements

Thanks go to Jim Fuller, Adam Retter and Dr. Stephen Foster who greatly helped me by giving advice and reviewing this paper. Thanks also go to Miguel De Melo for providing motivation and inspiration.

A. Saxon code example for invoking XQuery functions

```
Configuration config = new Configuration();
StaticQueryContext sqc = config.newStaticQueryContext();

XQueryExpression exp1 = sqc.compileQuery(
  "declare namespace f='f.ns';" +
  "declare function f:t1($p as xs:integer) { $p * $p };" +
  "declare function f:t2($p as xs:integer) { $p + $p };" +
  "1"
);

QueryModule qm = exp1.getStaticContext();

UserFunction fn1 = qm.getUserDefinedFunction("f.ns", "t1", 1);
UserFunction fn2 = qm.getUserDefinedFunction("f.ns", "t2", 1);

Controller controller = exp1.newController();

IntegerValue[] arglist = new IntegerValue[1];
for (int x=1; x<1000000; x++) {
  arglist[0] = IntegerValue.makeIntegerValue(BigInteger.valueOf(x));
  ValueRepresentation v1 = fn1.call(arglist, controller);
  ValueRepresentation v2 = fn2.call(arglist, controller);
  System.err.println("Returned product " + v1 + "; sum =" + v2);
}
```

B. Subset of the XQConnection2 Java interface

This is a subset of the XQJ2 (XQJ Squared)[1] interface, which is an experimental API built on top of the XQJ interfaces. This extension's approach is similar to the StAX2 (StAX Squared) Interfaces[2].

```
package com.xqj2;

import javax.xml.xquery.*;

public interface XQConnection2 extends XQConnection
{
  public <T> T createModuleProxy(
    String namespaceUri,
    String moduleUri,
    Class<T> clazz) throws XQException;

  public <T> T createModuleProxy(
    String namespaceUri,
    String moduleUri,
    Class<T> clazz,
    XQStaticContext properties) throws XQException;

  /** Other XQJ2 Methods **/
}
```

C. The XQFunctionName Annotation

```
package com.xqj2.proxy;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(value = RetentionPolicy.RUNTIME)
public @interface XQFunctionName
{
  public String value();
}
```

[1] XQJ2 API interfaces https://github.com/cfoster/xqj2
[2] "StAX2" API. Tatu Saloranta. http://docs.codehaus.org/display/WSTX/StAX2

D. The XQCastAs Annotation

```
package com.xqj2.proxy;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(value = RetentionPolicy.RUNTIME)
public @interface XQCastAs
{
  public String value();
}
```

E. XQJ's Java Data Type to XDM Data Type mapping table

Table E.1. Java Data Type Default XQuery Data Type(s)

Java Data Type	XDM Data Type
boolean	xs:boolean
byte	xs:byte
byte[]	xs:hexBinary
double	xs:double
float	xs:float
int	xs:int
long	xs:long
short	xs:short
java.lang.Boolean	xs:boolean
java.lang.Byte	xs:byte
java.lang.Float	xs:float
java.lang.Double	xs:double
java.lang.Integer	xs:int
java.long.Long	xs:long
java.lang.Short	xs:short
java.lang.String	xs:string
java.math.BigDecimal	xs:decimal
java.math.BigInteger	xs:integer

Java Data Type	XDM Data Type
javax.xml.datatype.Duration	depending on Duration Object state, one of the following; xs:dayTimeDuration, xs:yearMonthDuration, xs:duration, xs:date, xs:dateTime, xs:gDay, xs:gMonth, xs:gMonthDay, xs:gYear, xs:gYearMonth, xs:time
javax.xml.namespace.QName	xs:QName
org.w3c.dom.Document	document-node(element(*, xs:untyped))
org.w3c.dom.DocumentFragment	document-node(element(*, xs:untyped))
org.w3c.dom.Element	element(*, xs:untyped)
org.w3c.dom.Attr	attribute(*, xs:untypedAtomic)
org.w3c.dom.Comment	comment()
org.w3c.dom.ProcessingInstruction	processing-instruction()
org.w3c.dom.Text	text()

Bibliography

[1] XQuery 1.0: An XML Query Language (Second Edition) W3C Recommendation. [1]

[2] XQuery 1.0 and XPath 2.0 Data Model (XDM) W3C Recommendation. [2]

[3] Plain Old Java Object [3]

[4] Redstone XML-RPC Library - Greger Olsson. Redstone. [4]

[5] Engineering Java™ Proxy Objects using Reflection - Karen Renaud and Huw Evans. University of Glasgow. [5]

[6] XQuery API for Java™ (XQJ) 1.0 Specification - Spec Lead: Jim Melton, Oracle. Editor: Marc Van Cappellen, DataDirect Technologies. March, 2009

[1] http://www.w3.org/TR/2010/REC-xquery-20101214/
[2] http://www.w3.org/TR/xpath-datamodel/
[3] http://en.wikipedia.org/wiki/Plain_Old_Java_Object
[4] http://xmlrpc.sourceforge.net/
[5] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.111.2217&rep=rep1&type=pdf

A Wiki-based System for Schema and Data Evolution

Lorenzo Bossi
Dept. of Biology, Informatics and Communication (BICOM) – Insubria University
<lorenzo.bossi@uninsubria.it>

Alberto Trombetta
Dept. of Biology, Informatics and Communication (BICOM) – Insubria University
<alberto.trombetta@uninsubria.it>

Abstract

The community of users of a large data-driven web site may directly contribute to its management by feeding corrections and new additions, thus keeping "fresh" the information provided by the site. However, several issues may arise due to the fact that users may modify data in a more or less controlled way. Starting from a real-world scenario, we point out such issues and we present a simple and efficient framework. The proposed solution has been implemented in a XML-based prototype framework, that have been tested with large, real-world datasets.

1. Introduction

Large electronic commerce sites allow users to actively participate in collecting feedback information about items and products available on such sites. Typically, users may add comments and rank or tag items according to their preferences. Typically, the information provided by e-commerce sites' users is used for providing tailored recommendations to them and as the functionalities offered by to the users for adding and manipulating information are very limited: users typically can only add comments to the items and rate them, according to some ranking criteria. Following the recent trend of community-based information management [DRC06, Doa07, BCLM11] – according to which the information may be directly manipulated by end users – we propose to augment the "expressive power" of tools users employ in organizing and managing the semistructured (XML-based) information they provide about items, in order to allow an expressive and useful structuring of the information itself. As such, users may add and structure information about items (or even add new items) of their interest. We adopt an approach inspired by wikis, in which users may structure the data they provide in complex ways and manage it in a collaborative way.

Our application scenario (see Section 1.1) deals with a vast number of XML documents containing information about items as shown on a large e-commerce site, and since such information is not completely unstructured, our framework assumes that each document may (possibly in a loose way) adhere to one of a relatively small number of schemas (thereon called *templates*).

Henceforth, users may create and modify both documents and their corresponding templates. Furthermore users may interact by modifying documents and templates created by other users as well, thus adding and modifying information in a collaborative way (of course, such interactions have to be regulated by proper access policies and trust/reputation mechanisms stating which users may modify which data. In the present work we do not deal with such issues). Having the users such possibilities entails that our system has to take into account the interplay occurring among a modified template and the corresponding (unmodified) documents. More precisely the system, enforcing the constraints about the document structure, supports users in finding ill-formed documents by allowing them to improve their content.

While supporting a community-based approach for updating documents and templates offers the advantage in keeping information up to date (provided by users' feedbacks), such approach poses several non-trivial questions about the correct management of such information. In particular, (i) when and how updates on templates are reflected on the corresponding documents? (ii) How to manage the rollbacks of unwanted (possibly malicious) updates *without* erasing subsequent licit ones? And how such rollbacks on templates affect the corresponding documents?

This may be non-trivial tasks, as we will argue in the following. As for point (i) above, the problems we incur in dealing with a community-inspired data management approach come from the existence of integrity constraints that are expressed in a template and that the corresponding documents have to satisfy; whereas for point (ii), the difficulties arise in guaranteeing the maximum possible number of updates without losing validity of documents with respect of their templates.

In fact classical wikis handle structured data through the use of infoboxes [14], but they don't support any kind of constraint enforcement which can guarantee the uniformity of pages which use them.

The framework we present in this work is based on XML Schema [TMBM04, MB04] for the definition of the basic structure of templates, on Schematron [8] for the definition of more complex integrity constraints and on a XQuery Update-like language for the query language. As it is well known, updating XML documents in a consistent way is a far from trivial task [4] and our work can be summarized as follows.

Our Contributions. In this paper, we focus on the following main topics:

i. describe the prototype for a wiki focused on store and manage semistructured data,

ii. define ad-hoc methods for automatic documents evolution upon template change,
iii. propose a new kind of revision control system that is centered on how to minimize loss of new data and maximize data coherence in case of template rollback,
iv. define a simple, ad-hoc, XML-based data model to store data and an XQuery like update language for it.

We remark that we do not address relevant issues concerning access control policies to documents and templates (and the corresponding enforcement). We are aware of the paramount relevant of such aspect and we plan to work on them in the near future.

1.1. Our motivating scenario

As a real-world example of a large repository of relatively small and relatively uniform documents, we consider the price comparison service http://www.shoppydoo.it. Such site holds the large majority of the market share in Italy with more than 2 million users and it has a very significant presence in other european countries (e.g. Spain, France, Germany, Netherlands) and non-european as well (e.g. Brazil). More details on data volumes are detailed in Section 5.

The site stores information about more than 1 million items, grouped in roughly one hundred categories. Items are described in pages containing their technical details. Such information is displayed in concise and tabular form for letting users quickly find and compare items.

For example, the information about technical details about digital camera must specify brand and model, as well as camera resolution and memory support. Other less relevant – but still useful information – may comprise the presence of features like an image stabilizer or a face detector, etc.

The users of such site form an online community that may create, update and share information about items by interacting with the site itself. At the present time, the site does not allow its users to actively participate in the management of the displayed information. Our long-term goal is to provide community-based information capabilities to a large, e-commerce-based web site.

Thus, for example, in the case that the user Alice notices that the page describing her preferred camera reports incorrect information about its resolution, she can correct it. Further, Bob is a more active user and notes that almost every digital camera is able to connect to a PC and thus explicitly specifying such information is useless. As such, he decides to modify the digital camera template in order to remove such information from every corresponding document.

In what follows we present methods and techniques that allow users to directly manage such updates and to control the interplay between documents and templates. We will present in a more detailed way the actions performed by such users, as we unfold our motivating scenario in the following sections.

2. Related Works

To the best of our knowledge, the most mature work closest to ours is present in [2] in which the authors describe a prototype for a wiki for structured data. The main difference between our project consist that they manage only one (usually fairly big) XML document. It turns out that the considered schemas are simpler than ours: for example, they do not allow to specify the type of the data but only the tree structure. Rather, the author focus on query language issues. They are developing a powerful query language which let to select only a fragment of the XML based on various constraints which can involve also annotation on nodes. Finally they support only two types of schema updates: insertion that happen automatically when inserting a data which require a schema extension and deletion which delete also the corresponding data subtree.

Other works deal only with the schema evolution. We think that the most important for our context are [9] in which the authors propose a conceptual model for XML Schema evolution. They use a graphical environment to define schemas and schemas update. Then some normalizations are performed on updates to minimize their. After schema update, they recheck document validity and perform a document update. But there's no way to update node values on documents. Another interesting work is [7], where the authors define a set of update primitives for XML Schema. They study which evolution primitives are known not to compromise documents validity. Then, they use a labeling process to keep track of the document portions whose validity might have been compromised so they can revalidate only subtree to speedup the process. Their approach to documents evolution consist on the detection of the minimal modifications required to make the documents valid for the evolved schema. But only document structure can evolve, not document data.

3. Documents and Templates

We customarily represent a document as an XML tree. For example, a document containing information about a given digital camera may be structured in the following – rather conventional – way: the camera's model is stored in an alphanumeric string, its megapixel capacity is an integer number, the supported memory is a single value chosen from a set of alphanumeric strings. Further, we add a *section* element which is used to group related elements under a section name. Each element name is unique in its section.

Templates are defined in a way similar to what proposed by Examplotron [13]. We have chosen such approach for its ease of use (see Section 3.2). Thus, we define a template as an instance of a empty document, where each element has an additional boolean attribute specifying whether it is mandatory or not in the documents.

More formally, a document is composed by *data* nodes and *elements* nodes where: a *data node* contains only a string value; an *element node* is a tuple associated with a

name, a type and a set of children nodes; Elements may have simple or complex types, where complex types are in {*enum*, *values*, *section*} and simple types are in {*int*, *float*, *str*, *bool*}. The type of an element node defines restrictions on the set of children nodes. That is,

i. simple type elements and *enum* elements children set must contain only a data node;
ii. *values* elements children set must be a non empty set of distinct data nodes;
iii. *section* element children sets can be only a non empty list of children elements with distinct names (in this way, elements can be found without ambiguity).

As such, the document is an unordered tree of section and value elements, where the root is a section element.

A template is composed by *template elements* which are similar to documents' one. In addition, they have an extra boolean *mandatory* attribute. Every template has to satisfy the following constraints:

i. template elements of simple type must have an empty set of children;
ii. template elements with type *values* or *enum* must have a non empty set of distinct data nodes;
iii. template elements with type section must have a non empty set of children elements with distinct names.

As for documents, a template is an unordered tree where root element is a template section element.

A document is *valid*, of course we assume the well-formedness, if all data nodes contain values that:

i. an integer, if the type of parent node is *int*;
ii. a float, if the type of parent node is *float*;
iii. a boolean, if the type of parent node is *bool*;
iv. a non empty string, if the type of parent node is *str*.

A document is *valid with respect to a template* if and only if:

i. it is valid (see above);
ii. all document's elements are also present in the template with the same name and the same type;
iii. all template's elements with mandatory set has a corresponding element in the document;
iv. the children of *enum*, *values* and *section* elements of the document are also children of the corresponding template's elements.

3.1. Our scenario, continued

As one may suspect, documents and templates are stored in XML files. Document elements are serialized in XML nodes where the tag name defines the node type and the name is stored in an attribute. For example an element *Model* of type *string* is serialized as `<str name="Model">Z80</str>`.

Templates enforce the information type of complex element, in such a way that the *Wiewfinder* can be one of *optical* or *LCD*, while *Extra feature* must be one or more between a list of valid values. Templates XMLs are very similar to documents' one, the main difference is that simple elements are empty and that every element has an attribute *mandatory* which contain a boolean value. For example the element *Model* in a template is `<str name="Model" mandatory="true"/>`

3.2. Validation of documents

We use XML Schema [MB04, TMBM04] to validate documents and templates as serialized XML documents. This first step validates the overall structure of XML documents. Regarding documents, XML Schema is used to check that they are structured in sections containing the named values and the correct types of simple elements' values. Regarding templates, XML Schema is used to check sections, the uniqueness of name into their sections and the presence of a mandatory boolean attribute for each element.

As said before, users specify templates as an empty document. As such, in order to validate a document with respect to some template, templates themselves have to be rewritten in some suitable XML schema language.

We use Schematron [8] since its assertion rule validation style makes error reporting clearer and the usage of XPath constraints allows the definition of constraints over unordered sets of context-dependent elements. Furthermore, Schematron checks the presence of mandatory elements, the absence of illegal elements and the correctness of *values* and *enum* children elements. The conversion from a template to its corresponding Schematron schema is performed by an XSLT stylesheet [3].

As already pointed out, the main advantage in writing templates in the above presented XML format is its compact and easily readable syntax, that can be promptly deployed by the community users.

3.3. Evolution of templates and documents

Returning to our motivating scenario, since Bob reputes very important to know if a digital camera is able to record videos, he adds a new boolean mandatory field called *video recording* in the camera template. We note that this kind of update invalidates all the documents associated to the corresponding template. We patch this

problem adding a special page showing all invalid documents to let active users of the community perform updates on such documents, to restore their validity again.

3.3.1. Interacting with templates and documents

Community users can read, create and modify documents and templates. After a document is updated, the following steps are performed: (i) check whether the document is valid according to the corresponding XML Schema, if this is not the case, reject the update; (ii) otherwise get the associated template and translate it into a Schematron document; (iii) validate the document with the corresponding Schematron to check if it complies with the template and return the validation results.

A template update can (i) leave all associated documents valid (for example, the insertion of a new value in an enumeration); (ii) invalidate all associated documents (for example, adding a mandatory element in a mandatory section); (iii) require a necessary update and consequent re-checking of all associated documents (for example, deleting an element); or, finally, (iv) leave the documents in an unpredictable state, regarding their validity (for example, an optional value becomes mandatory). In this case, the only way to discern the documents' validity is to re-check them all. Since cases (iii) and (iv) are similar, because (iv) is like (iii) with an empty update, we treat them in the same way.

The update language we propose allows community users to create and update elements' types, their name, mandatory fields, add, modify and delete elements.

Since the community users may perform updates on templates and documents defined by the previously defined data model, we do not need the full expressive power of XQuery Update Facility [4] and, thus, our update language is basically a simplified version of XQuery Update. First, we define the element selector as a string that allows to find an element in unambiguous way. It is formed by the element name preceded by all sections name separated by the slash sign (e.g. `/Digital camera/Brand`). We also define a data selector as the selector of its container followed by a slash sign followed by the data text value wrapped by brackets (e.g. `/Digital camera/Supported memory/[CompactFlash]`). Those selectors can be easly translated into XPath [1] expressions. The latest example in XPath is written `/*/*[@name='Digital camera']/*[@name='Supported memory']/*[text()='CompactFlash']`.

3.3.2. Evolution of documents

In this section we describe our document update language and how each command can be translated in an XQuery Update statement.

Every command that we had defined take a selector as parameter. Whether it is an element selector or a data selector is first transformed into an XPath selector

while the command is recognized and translated into a valid XQuery Update statement.

A user may delete either a data or element node with the command `delete node <selector>`. In this case we need only to translate the selector to have a valid update.

Add new data into a document let a user to fill values list. This operation is performed by the command `insert data(text) into <elementSelector>` which is converted into `insert node <value>text</value> into <xpathSelector>`.

Since the most of elements node need a data child to be valid (all simple and *enum* types) we provide the command `insert node(type, name, value) into <elementSelector>` to insert both of them. When translated into XQuery Update, such command becomes `insert <type name="name">value</type> into <xpathSelector>`. Differently, *section* and *values* elements have may child nodes, so we need an overload for this command that doesn't require a value.

The last command we describe is for replacing the value of a data node. Its syntax is `replace value of <selector> with <newval>` and is translated into XQuery Update in `replace <xpathSelector>/text() with 'newval'`

3.3.3. Evolution of templates

Template evolution is more difficult because for every defined command we need not only to update – of course – the template but also to decide whether the documents associated have to be modified as well and, in the affirmative case, perform such updates. It is important to note that our node selectors (and the translated XPath equivalents) are valid both on documents and templates.

To add a new field on template specify we should specify the name, the type and if this information is to be mandatory. This operation is performed with the instruction `insert node(type, elName, isMandatory) into <elementSelector>`, which we translate it into XQuery Update with `insert <type name='elName' mandatory='isMandatory'/> into <xpathSelector>`. If the inserted node is not mandatory, after this update documents are still valid. If the inserted node is mandatory and all ancestors sections are mandatory too, after this update all associated documents are no more valid. Otherwise all associated document must be re-checked to define if they are still valid.

Another useful command is similar to the last one but lets user to specify a default value for the new elements. The template update is equal to the last one, but in this case we have also the following document update `insert <type name='elName'>defaultVal</type> into <xpathSelector>`.

New data nodes can be added to an enum or values element with `insert data(val) into <elementSelector>`. This update preserves validity of documents and can be written in XQuery Update as `insert <value>val</value> into <xpathSelector>`.

The command `change type <elementSelector> with newType` is useful to change the type of an element. The template and document XQuery Update is `rename node <xpathSelector> as newType`. Is important to note that, depending on the update and on the old data, document validity could be preserved after this update.

In a similar way, to change the name of an element we use `changename <elementSelector> with newName` and we apply to the template and the corresponding documents the instruction `replace <elementSelector>/@name with newName`.

The command `changemandatory <selector> with (true|false)` is used to change the mandatory value of an element. It is translated in `replace <elementSelector>/@mandatory with (true|false)`. After this operation, if the updated element is not mandatory all documents are still valid. If it is mandatory, as well as all the section ancestors, all documents is marked as invalid. Otherwise, all documents have to be revalidated.

The simplest operation is the deletion of a node, that is performed in the same way both on templates and documents by the XQuery `delete node <xpathSelector>`.

The last operation is `replace value of <selector> with newVal` that uses XQuery functions to compute `newVal`. This operation is used to perform an update to a field on all documents associated to the current template.

Is important to note that particular updates sequence can leave the system in an inconsistent state. For example we have a template with a not mandatory element, then some documents will contain this element while some other will not. All documents and templates are valid. This is the initial state. A user decide that the element must be mandatory and update the template. As described previous, after the template update, all documents are revalidate. So we have some valid documents (those have the mandatory elements) and some invalid ones (all the other). In this state another user see the new template and change back the same element to be optional. An unskilled or heedless user can perform this operation as a normal update, instead using the correct feature of the versioning system. So the template is updated however no operation on document is performed because this kind of update don't compromise their validity. But all documents that was marked as invalid is now valid again and in this final state we have all templates and documents valid but with some document marked as invalid. Therefore we need also a process scheduled for re-checking at regular intervals the validity of documents to achieve an eventual consistency of data.

3.4. Revision control support

Revision control is a very relevant feature for wikis. It is useful for monitoring a page evolution and to deal with modifications performed by malicious users.

Implementing a revision control system for our wiki-based repository is not trivial, since documents are represented as XML trees with complex constraints occurring among their elements, as specified by their corresponding templates.

In order to illustrate the problems in building a revision control system fulfilling the above mentioned conditions, we consider the scenario in which a user wishes to undo a template update, but – in the meanwhile, after the template update – the documents associated to such template have been modified, so they are no more valid with the old template.

In this scenario we can operate in two different and completely opposite way: (1) we can revert all documents to their valid versions with the old template or (2) we can leave all documents to the last revision and revert only the template. Both solutions have pros and cons: The former maintains consistency between documents and templates but it can potentially loose many useful document updates. The latter does not loose document updates, but it can potentially leave all documents in an invalid state.

There is no single best solution to the problem of how to retain document updates while satisfying template consistency too, since it may depend very heavily from the context. For example, if one wishes to undo revert a malicious template update (possibly the outcome of a deliberate act of vandalism), it is pointless to save the corresponding document updates because documents contain information corrupted by the vandalism act, as well. In this case, Solution (1) appears to be the best fit. On the other side, inconsistencies between a template and the corresponding documents arising from a minor change in the template structure (e.g., an element may become optional) seem to be viably managed by Solution (2).

Given the extreme sensitiveness to the context of the chosen solution, our aim is to define helpful techniques and tools that support the user in adopting the best possible solution that fits its needs.

In particular, we define a hierarchy of possible solutions formed by the two scenarios introduced above, plus other two intermediate levels that offer a sort of "interpolation" between such two extremes. In more detail, when a user has to "revert" a template to a previous version, we can choose to:

1. revert all the corresponding documents to their previous versions, in agreement with the reverted template;
2. revert only the structure (as dictated by the reverted template) of the documents but not the content;
3. revert only the document involved by the template update;
4. leave the document unmodified.

The following examples show a real-world scenario the four options just introduced.

As a deliberate act of vandalism, consider the following: an user deletes a large part of a template and renames the elements of the remaining part with unrelated content. A document update performed with the new template, can only try to limit the damages, but can not improve the document quality. In this case the best

solution is ignore the document updates and revert documents as they were before the template update.

As for the second option, consider a car radio template, describing its features. In the *Car Radio* template, a user rename the *Audio* section in *Speaker system*. The semantic of the section does not change. So every document update is legit. But all the other templates which describe consumer electronics stuffs contain an *Audio* section. So, to uniform the templates, is better revert this update undoing the rename but without loose any update to the documents (Level 2).

Consider a *Notebook* template which contains a *Memory* and a *HD* sections. Both of them contains two integer *Total size* and *Max size*. An unskilled user, who does not know the difference between RAM memory and Hard Disk, can rename the *Memory* section in *Disk sizes* section hoping to make the template better. This update changes the semantic of the section. So every document update which involve this section is compromised. When someone reverts the template update, the best solution should revert all documents updates which involved this section, but should keep all the others (Level 3).

Suppose that someone add a new data *Autofocus* under the multi-values *Extra features* in a camera template. This is a useful information, but it is redundant since there is an optional boolean field *Auto focus* in the section *Lens system* yet. In this case document updates happen after the template update contains useful information but in the wrong place. The better solution consist to revert the template but not the documents so no information is loose (Level 4). In this way we can encourage the community to correct the documents moving the information in the right place.

4. Our framework

Our framework, as seen in Figure 1, is structured in four components: the XML repository, the query/update processor, the document and template validity checker and the user interface.

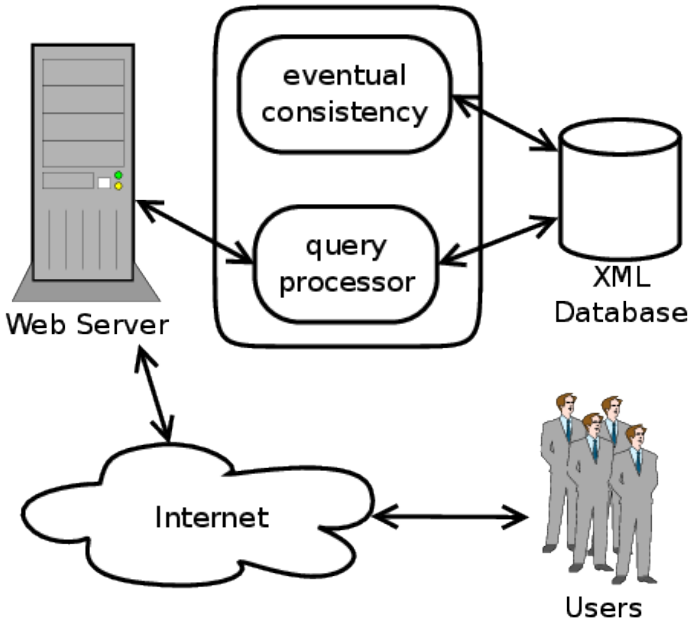

Figure 1. Framework schema

For the XML repository we need a system that can efficiently handle many XML files and that supports XQuery Update. We choose eXist [12] because, as of now, is the most standards-compliant and extensible among free native XML databases. This component provides the persistence of data, the XQuery Update engine and the XML validation.

The query processor component takes as input the user queries and translates them into XQuery Update. It communicates with the database by sending updates, asking for validations and performs commit or rollback depending on validation results. Algorithms described in Section 3 are implemented in this component.

The user interface provide a set of features which let the user to intuitively interact with the prototype without the requirement of a formal training. Since the purpose of the prototype is to build a collaborative system, the user interface is a web application. It lets the user to (i) easily find documents and templates based on different criteria, (ii) read a rendered version of templates and documents and (iii) edit templates and documents in an interactive way.

5. Experimental results

Since the relatively small size of documents and templates and since updates and validations are performed on a native XML database, we can safely assume that a single update operation is performed in a relatively short, constant time roughly equal to 10 ms.

Thus, the performance issues of our framework depends on the number of documents that have to be checked, depending on the kind of update operation that has been issued.

All tests have been performed on a PC with an Intel® Core™ 2 Duo P8700 CPU and 4 GB of RAM running Windows Vista™. The framework is implemented in C# 3.0 and deploys eXist 1.4 [12] as the XML native database.

As already mentioned in the introduction, the dataset used in the experiments come from the ShoppyDoo online price comparison service which is visited by more than two million of unique user per month and compare prices of four million offers from fifteen hundred merchants. Every document describes the technical details of a single product and there is one template for product category. In total, our dataset contains 9605 documents and 72 templates, totaling about 65 MB of XML files. The less populated category (Digital photo frames) contains only 3 documents, while the most populated one (LCD, LED and plasma TVs) contains 798 documents. We want remark that the current dump of Wikispecies [15] is approximately 370 MB of XML data. It's interesting for us show that our test dataset is big about the 18% of a small Wikimedia Foundation project.

As explained in Section 3.3.1, a template update can fall into three different categories, depending upon its impact on the associated documents: it can leave them all valid, it can invalidate all of them or it can make necessary to update and recheck all of them. As such, we performed editing templates tests with different quantities of documents and with the three different kinds of update. The results are shown in Figure 2.

The first type of update is very fast and it is not affected by how many documents are associated to the modified template. Since the algorithm needs only to update one (typically) small XML document (the template itself) it executes in about 100 milliseconds.

We point out the in our current prototype, all documents have a metadata field that records their validity with respect to their templates. In this way, the second type of update performs in a time linear in the number of documents. Since updating the validity to given value is very fast, all our tests ended within 150 milliseconds.

The last type of update is the worst because, after the template update, the prototype has to perform an update to all the documents and they all have to be revalidated. The time is linear to the number of documents, and since validation process is slower than the update execution, the worst case takes about two minutes to execute.

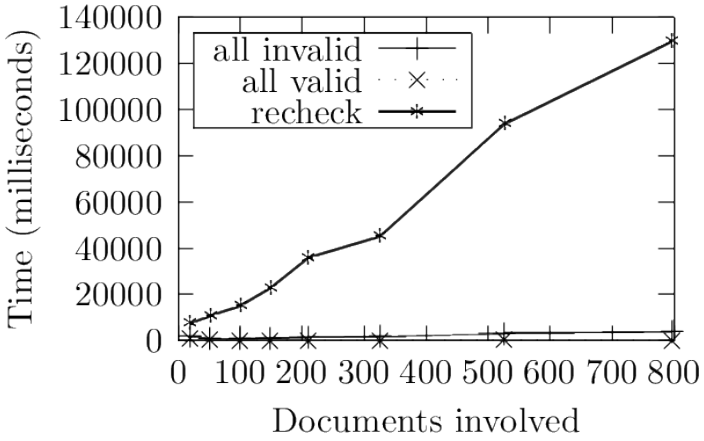

Figure 2. Template update performance test

6. Conclusions

Although our framework is still under development, the first experimental results show that updates of templates associated with small – yet significative – sets of documents execute in reasonable time; still, there is ample space for optimizing the proposed procedures. The optimization process may involve as well a phase of logical redesign of large XML schemas into smaller, more manageable ones. Along with optimization issues, we are dealing with the realization of a suitable web application that allow users to easilyinteract with the repository.

Finally, the other major topic deserving further investigation is – of course – how to regulate the access of users to data, allowing them to modify items created by other users. To this end, we are investigating the integration of classical access control techniques with incentive-based mechanisms borrowed from reputation systems, in order to elicit a collaborative behaviour from users but keeping an eye – at the same time – on what they can do.

Bibliography

[1] Anders Berglund, Scott Boag, Donald D. Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML path language (XPath) 2.0 (second edition)*. W3C recommendation. W3C. December 2010.

[2] Peter Buneman, James Cheney, Sam Lindley, and Heiko Müller. *The database wiki project: A general-purpose platform for data curation and collaboration*. 15–20. *SIGMOD Record*. 40. 3. 2011.

[3] James Clark. *XSL transformations (XSLT) version 1.0*. W3C recommendation. W3C. November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116.

[4] Don Chamberlin, Jonathan Robie, Daniela Florescu, Jim Melton, Jérôme Siméon, and Michael Dyck. *XQuery update facility 1.0*. Candidate recommendation. W3C. June 2009. http://www.w3.org/TR/2009/CR-xquery-update-10-20090609/.

[5] AnHai Doan. *Data quality challenges in community systems*. QDB. 2007.

[6] AnHai Doan, Raghu Ramakrishnan, Fei Chen, Pedro DeRose, Yoonkyong Lee, Robert McCann, Mayssam Sayyadian, and Warren Shen. *Community information management*. 64–72. *IEEE Data Eng. Bull.*. 29. 1. 2006.

[7] Giovanna Guerrini, Marco Mesiti, and Daniele Rossi. *Impact of xml schema evolution on valid documents.*. WIDM. pages 39–44. ACM. 2005.

[8] R. Jelliffe. *Schematron*. Web page. October 2000. http://www.ascc.net/xml/resource/schematron/.

[9] Meike Klettke. *Conceptual xml schema evolution - the codex approach for design and redesign.*. http://dblp.uni-trier.de/db/conf/btw/btw2007w.htmlKlettke07. BTW Workshops. pages 53–63. Verlagshaus Mainz, Aachen. 2007.

[10] Ashok Malhotra and Paul V. Biron. *XML schema part 2: Datatypes second edition*. W3C recommendation. W3C. oct 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[11] Henry S. Thompson, Murray Maloney, David Beech, and Noah Mendelsohn. *XML schema part 1: Structures second edition*. W3C recommendation. W3C. oct 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[12] *eXist-db open source native xml database*. Web page. http://exist.sourceforge.net/.

[13] Eric van der Vlist. *Examplotron*. Technical report. 2003. http://examplotron.org/.

[14] *Help:infobox — wikipedia, the free encyclopedia*. 2011. Web page. http://en.wikipedia.org/w/index.php?title=Help:Infobox&oldid=455091951. Online; accessed 18-November-2011.

[15] *Wikispecies main page*. 2011. Web page. http://species.wikimedia.org. Online; accessed 28-November-2011.

Jiří Kosek (ed.)

XML Prague 2012
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 2012

ISBN 978-80-260-1572-7