



XML Prague 2024

Conference Proceedings

Prague University of Economics and Business
Prague, Czech Republic

June 6–8, 2024

XML Prague 2024 – Conference Proceedings

Copyright © 2024 Jiří Kosek

ISBN 978-80-907787-2-6 (pdf)

ISBN 978-80-907787-3-3 (ePub)

Table of Contents

General Information	v
Sponsors	vii
Preface	ix
Stormy First Draft – <i>Marta Bartnicka</i>	1
Navigating and Updating Trees of Maps and Arrays – <i>Michael Kay</i>	13
JSONPath: an IETF Proposed Standard, with comparisons to XPath – <i>Alan Painter</i>	37
Containerizing XML Build Tools to Facilitate CI/CD – <i>C. Edward Porter</i>	53
QTI and InDesign – <i>Mark Dunn</i>	63
XMQ/HTMQ - see XML and HTML in a new light – <i>Fredrik Öhrström</i>	85
<custom-element> DCE introduction – <i>Sasha Firsov</i>	103
Why Adding Some CSS Isn't Enough – <i>Anne Rudolf</i>	113
XML preserved from the past and into the future or? – <i>Karin Bredenberg</i>	123
Transparent Invisible XML – <i>Nico Verwer</i>	137
Round-tripping Invisible XML – <i>Steven Pemberton</i>	153
Towards RESTful XQuery 2.0 – <i>Adam Retter</i>	165
Tutorial Development XML Mashup with XProc – <i>Erik Siegel</i>	199
Modern Benchmarking of XQuery and XML Databases – <i>Alan Paxton and Adam Retter</i>	211
Simple Semantic Data Modeling in XML (SeMoX) – <i>Renzo Kottmann, Cedric Pauken, and Andreas Schmitz</i>	231
GEDCOM to RDF – <i>Robert Walpole</i>	251
Bridging XDM types in multiple native type systems – <i>O'Neil Delpratt and Matt Patterson</i>	263
natural-xml-diff: an XML Diffing Library – <i>Martijn Faassen</i>	281
It's Useful After All — VIN Numbers, DITA, and iXML – <i>Ari Nordström</i>	295

General Information

Date

June 6th, 7th and 8th, 2024

Location

Prague University of Economics and Business
W. Churchill Sq. 4, 130 67 Prague 3, Czech Republic

Organizing Committee

Petr Cimprich, *XML Prague, z.s.*
Vít Janota, *XML Prague, z.s.*
Káťa Prouzová, *XML Prague, z.s.*
Jirka Kosek, *xmlguru.cz & XML Prague, z.s.*
Martin Svárovský, *Memsources & XML Prague, z.s.*
Mohamed Zergaoui, *ShareXML.com & Innovimax*

Program Committee

Petr Cimprich, *Wunderman Thompson*
Jim Fuller, *MarkLogic*
Michael Kay, *Saxonica*
Jirka Kosek (chair), *Prague University of Economics and Business*
Ari Nordström, *Creative Words*
Uche Ogbuji, *Zepheira LLC*
Adam Retter, *Evolved Binary*
Andrew Sales, *Bloomsbury Publishing plc*
Felix Sasaki, *SAP SE*
John Snelson, *Progress Software*
Eric van der Vlist, *Dyomedeia*
Priscilla Walmsley, *Datypic*
Norman Tovey-Walsh, *Saxonica*
Mohamed Zergaoui, *Innovimax*

Produced By

XML Prague, z.s. (<http://xmlprague.cz/about>)
Faculty of Informatics and Statistics, VŠE (<http://fis.vse.cz>)

Sponsors

oXygen (<https://www.oxygenxml.com>)

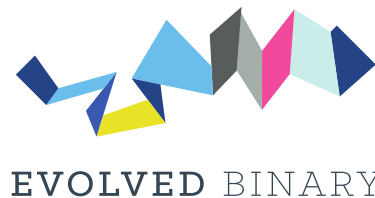
Antenna House (<https://www.antennahouse.com/>)

le-tex publishing services (<https://www.le-tex.de/en/>)

Saxonica (<https://www.saxonica.com/>)

Pantopix (<https://pantopix.com/>)

Evolved Binary (<https://evolvedbinary.com/>)



Preface

This publication contains papers presented during the XML Prague 2024 conference.

In its 17th year, XML Prague is a conference on XML for developers, markup geeks, information managers, and students. XML Prague focuses on markup and semantic on the Web, publishing and digital books, XML technologies for Big Data and recent advances in XML technologies. The conference provides an overview of successful technologies, with a focus on real world application versus theoretical exposition.

The conference takes place 6–8 June 2024 at the campus of Prague University of Economics and Business. XML Prague 2024 is jointly organized by the non-profit organization XML Prague, z.s. and by the Faculty of Informatics and Statistics, Prague University of Economics and Business.

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

The Thursday run in an un-conference style which provides space for various XML community meetings in parallel tracks. Friday and Saturday is devoted to a classical single-track format and papers from these days are published in the proceedings.

Since 2022 both Markup UK and XML Prague conferences are held in alternate years. We are looking forward to meeting you in May/June 2025 in London and in June 2026 in Prague.

We hope that you enjoy XML Prague 2024!

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

Stormy First Draft

Marta Bartnicka

Dolby

<marta.bartnicka@dolby.com>

Abstract

Stormy is a web tool developed to deliver generative AI for our internal users with two goals:

- *enhance baseline AI with company-specific technology information,*
- *protect the company intellectual property.*

This document covers the following key points:

- *The genesis of the StormyAI project: the two goals explained*
- *How we develop the tool: web UI, generative AI models under the hood, and how we “teach” genAI about our technologies*
- *How we capture use cases of information developers, support, and engineers: summary of learnings from alpha and beta phases*
- *Copyright and Licensing considerations for AI-generated content*

The presentation will include live demo of Stormy AI - in real time if I can get into company VPN, or from a back-up recoded demo. A significant part of the demo will be generating first draft of DITA XML documentation, working with XSLT and with custom XML.

1. The genesis of the StormyAI project

The project idea and name was inspired by the concept of Shitty First Draft¹ and named “Stormy” because “Project Shitty” might not be received well at the company. The main purpose of enabling generative AI for our internal users was to help them create first draft (whatever the specifier) for the writing of any kind that they need to perform within their job duties.

With that, we had to overcome two limitations of the publicly available genAI applications:

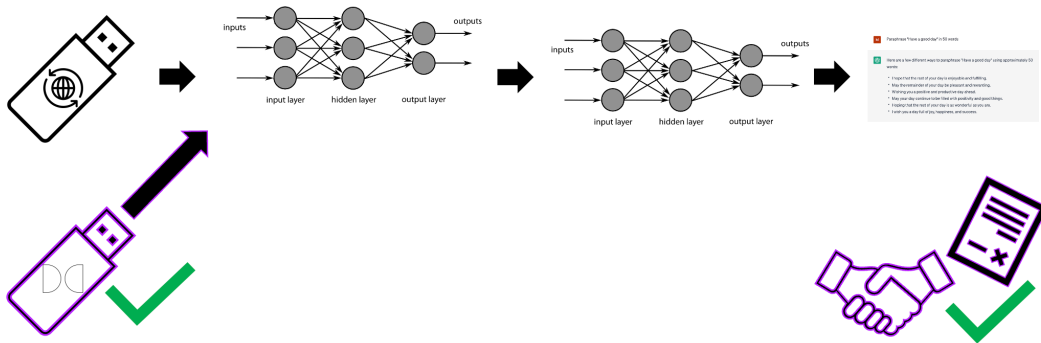
- Enhance baseline AI with company-specific technology information, because only a small fraction of our company’s documentation is publicly available and could be harvested by tools like ChatGPT into their training resources.

¹ <https://patrikedblad.com/productivity/the-shitty-first-draft/>

- Protect the company intellectual property, as the company's business is mainly about developing and licensing technologies, and thus, we wouldn't want to feed the public AI models with our content.

QUALITY SUMMIT 2023 SPEED TALK - MICHAL CHARYDCZAK - GPT VS. DOLBY ENGINEER

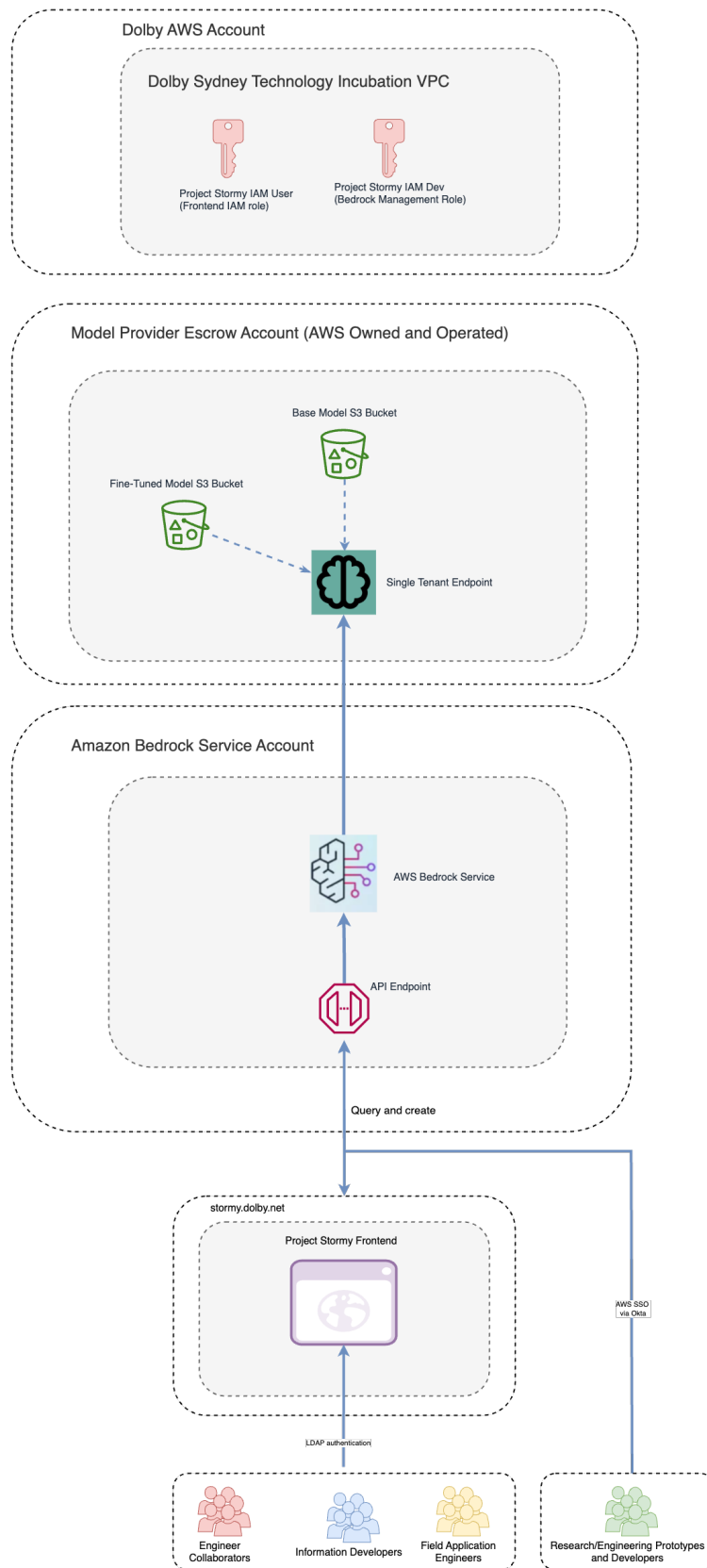
2024: What happened - StormyAI - useful and safe



2. How we develop StormyAI

2.1. ML infrastructure

The core infrastructure supporting StormyAI is AWS Bedrock (Amazon) that offers a choice of foundation models: Anthropic Claude, Meta Llama, Mixtral, and Stable Diffusion. Our infrastructure allows to explore multiple LLMs with the same web UI, user authentication, and company content delivered to the AIs.

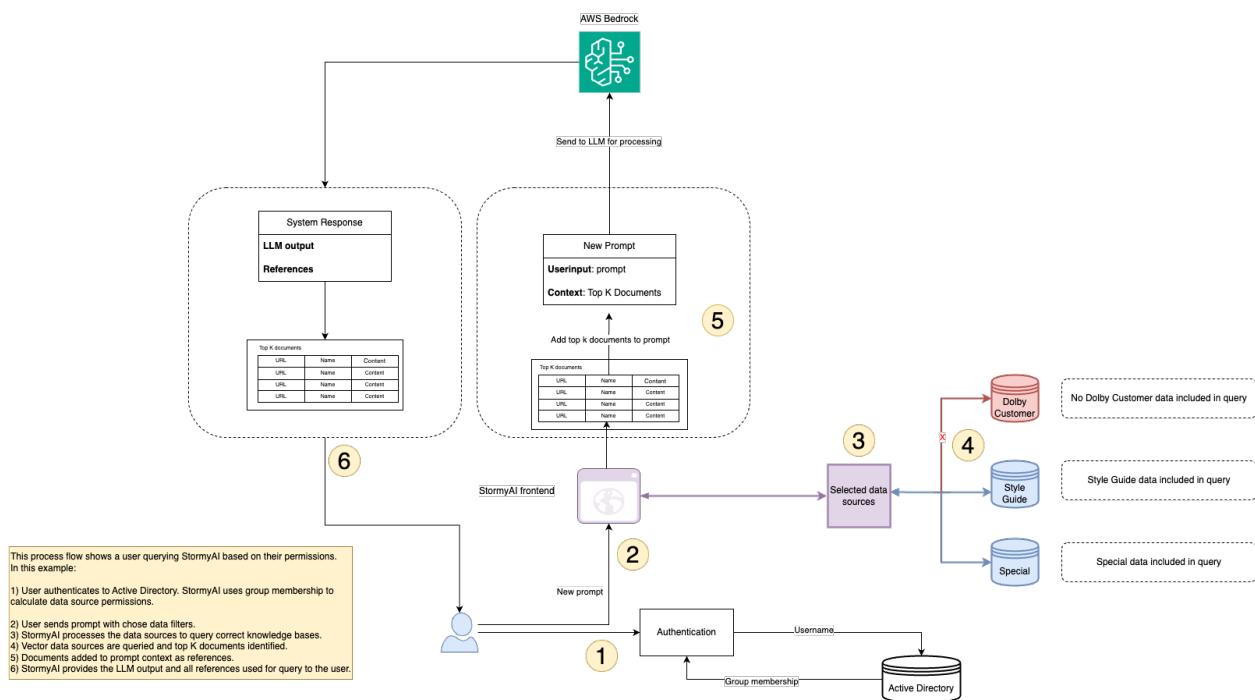


2.2. Information storage, retrieval, and security

AI systems are only as good as their data. To make Stormy useful, we need proprietary company content ingested into the system and made available to the LLMs. This includes gathering and filtering content from our public websites, from the portal that delivers proprietary content to licensing clients, and from the corporate style guides. We consider adding more content from the resources like company Confluence or GitLab repositories, however that will require more advanced filtering of either draft/obsolete, or classified/confidential content.

Stormy's web UI allows users to select one or more of the data sources available to them. Users can also select how they want the LLM to process the proprietary data:

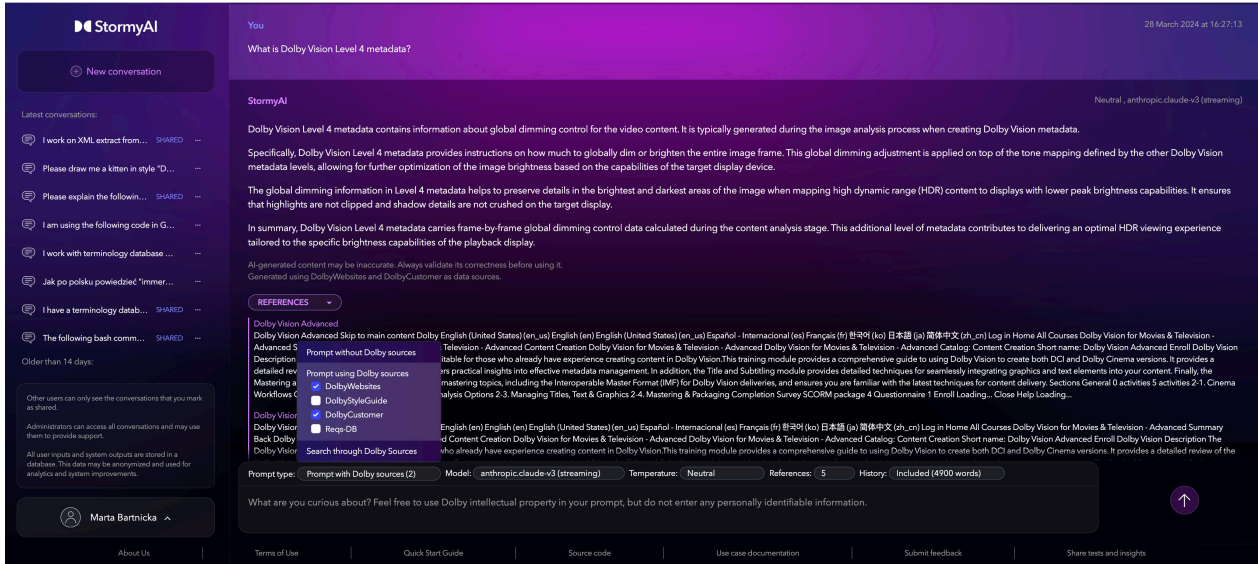
- Search the company content and return best results without generating anything
- Refine response generation via Retrieval-Augmented Generation², where the content that best matches a user prompt is added as additional context
- Generate response from a baseline AI model, without adding any company content.



Furthermore, we have built features to customise the contextual information available to the LLMs within individual user conversations, allowing users to upload files and to remove prompt/response pairs from a conversation.

² <https://aws.amazon.com/what-is/retrieval-augmented-generation/>

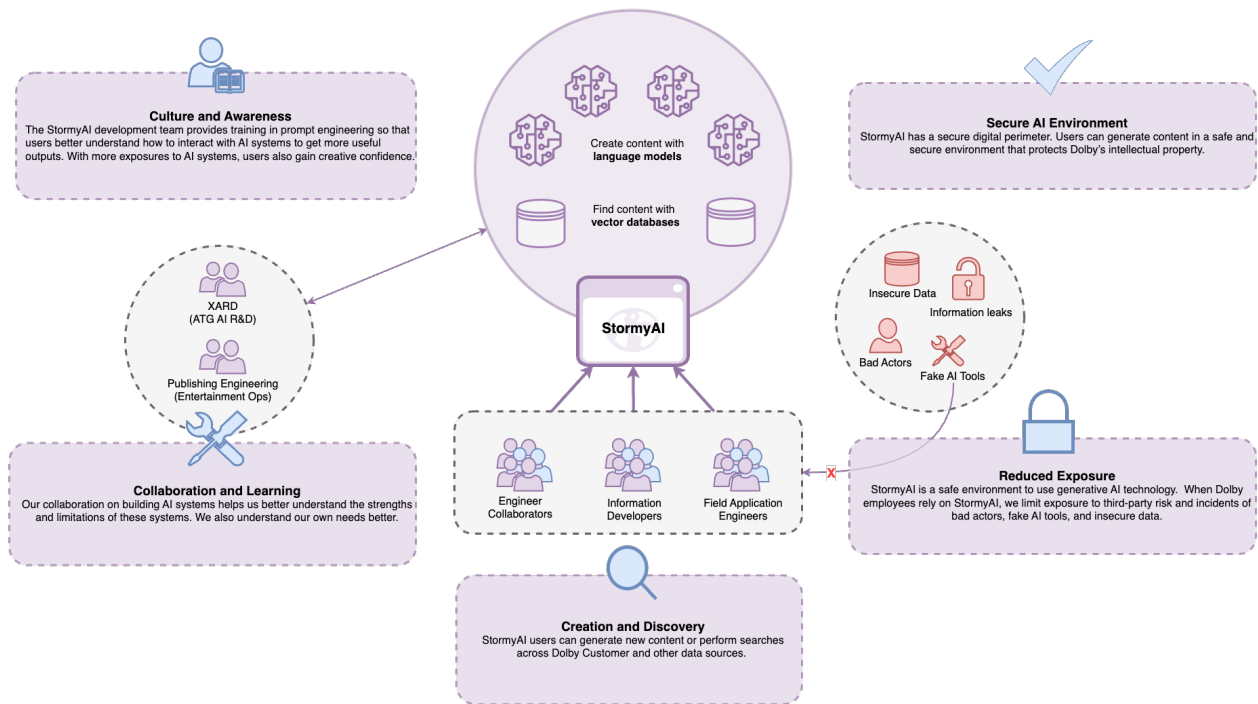
The platform leverages existing company authentication infrastructure to manage internal user permissions and the data sources they can use. We have built an application that enables users to interact with LLMs with a much greater level of control than commercially available tools.



2.3. Application layer and AI usability

For StormyAI, we needed to create an accessible and frictionless user experience, both from application and cultural perspectives. Our user base spans from information developers aka technical writers, through support specialists, to IT and other technology engineers. We hope that our frontend gradually turns the “minimum viable product” prototype into a “minimum lovable product” that encourages users to interact with the platform and to provide early feedback. Based on the expectations of our alpha users, we chose a conversational interface that allows users to refine their prompts and to switch LLMs mid-conversation.

As we iterated on the development of the platform, we provided prompt engineering training and built a community of nearly 400 users to learn and share knowledge about how to effectively use LLMs to create stormy first drafts. The tool allows to share a conversation or individual prompts, making it easier for users to exchange useful output and prompting examples.



3. How we capture Stormy use cases

3.1. Initial assumptions

When designing StormyAI, we considered the following use cases:

Information Developers	Support Specialists	Engineers
Create an initial draft of documentation	More effectively resolve customer issues	Quickly build documentation
Identify useful terminology	Identify how to improve products and training	Reuse the existing content
Detect inconsistencies in their content	Expand technical knowledge and expertise	Integrate genAI into tools

For all our internal users, we also expected the following enhancements:

- Writing in the company style
- Writing in English as a second language.

3.2. Learnings from alpha phase

Alpha user acceptance testing took about 3 months. Our user base grew from 20 to 200. The early adopters team, as planned, consists of Information Developers,

Support and Engineers. As our users explored generative AI, they have documented quite a few use cases that we'd not think about, for example:

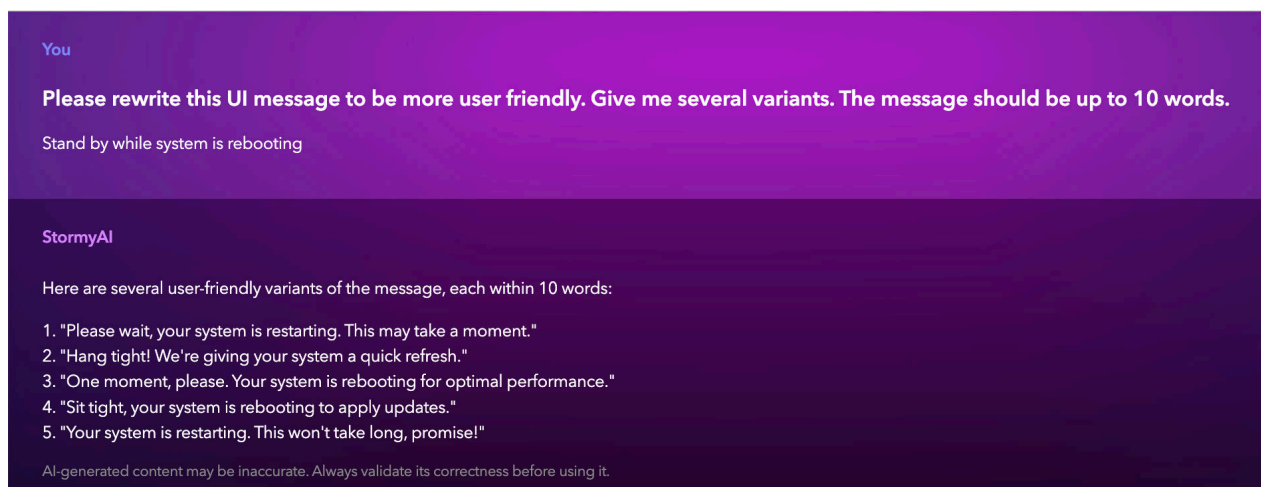
- Generate code for a Mermaid diagram
- Migrate legacy code for terminology exports from XSLT to Python
- Provide better versions of a draft UI message (UX writing)
- Convert draft notes from an engineer describing a new technology into a reference DITA XML topic
- Rewrite a text following certain style rules or provided patterns.

In the alpha phase, our Information Development users have quickly proven that StormyAI can generate valid DITA XML right out-of-the-box. DITA standard is public, well documented, and the Internet is full of good DITA examples - therefore, generic AI models can generate DITA that, syntax-wise, requires little or no editing.

The hard part is getting the right information into the DITA or any other type of content. We identified what we call “safe” and “unsafe” use cases for generative AI:

A safe use case is when a person prompting the AI knows exactly what the result should be, so they can fully check the correctness of the output - genAI is their “stupid digital assistant” that saves tedious, repetitive work. Examples of a safe use case can be:

- Editing your own content for linguistic correctness, because you know what's there factually
- Converting a paragraph to bullet point, because you also know and can check the accuracy and completeness of the result
- Generating or reworking code, because code either works and does what you want, or not
- Converting unstructured chapter of text into DITA XML.



The screenshot shows a chat interface with a purple header and a dark purple footer. The user's prompt is in white text on the purple background, and the AI's response is in white text on the dark purple background.

You

Please rewrite this UI message to be more user friendly. Give me several variants. The message should be up to 10 words.

Stand by while system is rebooting

StormyAI

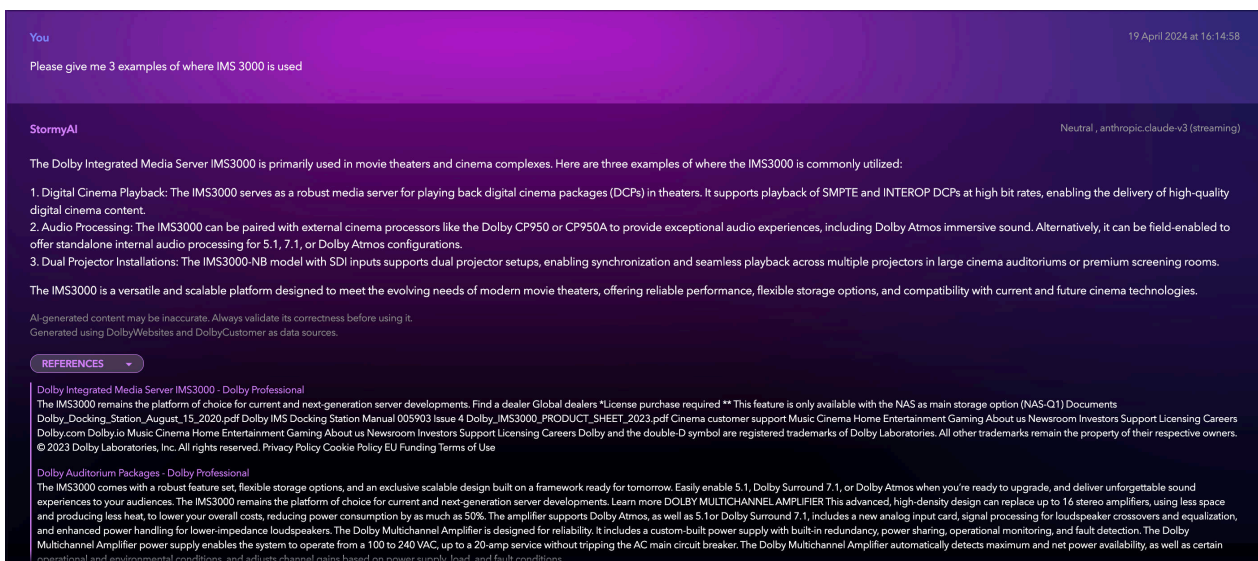
Here are several user-friendly variants of the message, each within 10 words:

1. "Please wait, your system is restarting. This may take a moment."
2. "Hang tight! We're giving your system a quick refresh."
3. "One moment, please. Your system is rebooting for optimal performance."
4. "Sit tight, your system is rebooting to apply updates."
5. "Your system is restarting. This won't take long, promise!"

AI-generated content may be inaccurate. Always validate its correctness before using it.

An unsafe use case is when a person prompts AI for information they don't have, or only have a part of it; in this case, they cannot immediately validate the output for factual correctness. In the unsafe use cases, we strongly recommend our users to validate the output in another reputable source of information or with an SME. Examples of an unsafe use case can be:

- Requesting an explanation of a technology that you're not familiar with, because the explanation can be hallucinated
- Asking for a few examples of a concept which you only know by its name, because some of the examples may not be valid (*as the last point in the image below*)
- Summarising a large document which you never read, because the summary may be incomplete or have added some related information that was not in the original document
- Converting draft notes into DITA XML documents that follow company standards of structuring content, because generative AI may interpret the standards incorrectly, even if provided an explanation or examples (moreover, some information from examples may bleed into the output).



Note that unsafe use cases can still be beneficial to Stormy AI users; the AI responses bring ideas to check and directions to explore that one would not think of without a significant research. The key to success is to evaluate.

3.3. Learnings from beta phase

- Using Dolby Sources improves technical quality and factual correctness of StormyAI outputs, but we still need to check them
- Even users with little programming experience successfully use StormyAI to generate scripts or rework legacy code (see the XSLT/XML use cases below)

- Safe use cases - text reworking - are in high demand from both native and non-native English users
- StormyAI was tested for summarising Teams meeting transcripts and is preferred over Microsoft Copilot
- StormyAI is explored by various user groups: technical writers, programmers, engineers, QA, legal, marketing, support
- Users request more Dolby Sources or capability to have persistent storages of their own context
- Only 3% users take time to rate StormyAI outputs

3.3.1. GenAI as help with XSLT

Generative AI is useful when we need to enhance complex XSLT code, for example:

- Analyse and rework legacy templates (e.g. consistent footnote handling in custom PDF processing)
- Analyse algorithms from legacy XSLT and rewrite the code in Python for further development (e.g. termbase extract parsing).

3.3.2. GenAI as help with other XML processing

Another area where GenAI is of great help is extracting data from XML or building XML from a specification or an example:

- Extracting terminology from terminology database export and generating Oxygen Terminology Checker file.

4. Copyright and Licensing considerations

Generally permitted use of copyright protected material and AI are not entirely clear yet. Issues around using copyright protected materials for AI purposes are still being worked out. Therefore, we advise our internal users to carefully check any AI-generated content for potential reuse of 3rd party copyright-protected resources that could have been absorbed into the training data for one of the generic AI models, with or without explicit consent of the creators and owners of those resources.

A lot of documentation and other content produced by our Information Developers or Engineers will, ultimately, become licensed assets of the company. Therefore, we advise our users to apply to AI-generated content the same checks and rules as they apply to human-generated output, Internet search results, or code from StackOverflow:

- Check for factual and linguistic correctness - consult SME, test code, copy edit documentation, etc.
- Check and attribute accordingly for the 3rd party license, copyright, or other rights (including open source)
- At some point, we may implement a mechanism for tagging AI-generated information³ for downstream use - to be determined.

Until further notice, our users must not directly use AI-generated content in the following cases:

- Building source code
StormyAI is not yet approved for source code. Dolby has not identified a legal framework for this use case. If generative AI is used to build code, and that code is actually from open source or code owned by others, then we might not be able to recognize revenue for that code or we might face challenges from others.
- Publicly sharing generative AI images in marketing material
Images generated from AI for marketing materials, such as presentations and documents, could be risky and possibly lead to reputational harm. There are cases where a third party copyright owner claims that the AI generated images infringe the third party's rights.

5. Conclusions

1. Generative AI is more helpful than we expected, and at the same time, more prone to factual errors (see the “unsafe use case” example above)
2. Dolby is obliged to deliver quality solutions that are technically correct; we are not yet ready to build a customer-facing genAI solution.
3. In the following months, we focus on improving StormyAI for our internal users (nearly 400 and counting!)
4. We need to explore the following areas:
 - More data sources, custom data sources
 - Textual output quality checks where possible
 - Legal clearance for AI code generation
 - Generating diagrams and images

³ <https://artificialintelligenceact.eu/>

6. Our next steps with StormyAI

6.1. Stormy Agents

Similar to “custom GPTs” by OpenAI, Stormy Agents will be a platform for Dolby employees to create a curated LLM experience for their specific use cases:

- Upload a set of documents to StormyAI to create a data source
- Specify custom prompt, context or examples
- Pre-select model or temperature.

Stormy Agents will enable more collaboration and sharing of StormyAI in custom workflows.

6.2. StormyAI Analytics

We start implementing automatic metrics to assess the quality of AI outputs. For example, we will compare prompts and responses to assess:

- Accuracy - does the response match prompt intent and context?
- Relevance - does the information address the prompt sufficiently?
- Appropriateness - does the tone/style fit the prompt context?
- Length of a conversation per topic - how many prompts resolve around same subject?

We also started gathering more user feedback to benchmark automatic metrics vs. user experience - without user feedback, the metrics can only tell when AI outputs are useless (for example, not related to prompts), but not whether they are useful and safe to use.

Navigating and Updating Trees of Maps and Arrays

Michael Kay

Saxonica

<mike@saxonica.com>

Abstract

This paper describes new features proposed for the 4.0 versions of XSLT, XPath, and XQuery, designed to make it easier and more efficient to query and update trees of maps and arrays, such as those typically derived by parsing JSON.

1. Terminology

Before we start, let's introduce some terminology.

- I'll use the term QT3 to refer to the family of specifications that includes XSLT 3.0, XPath 3.1, and XQuery 3.1 (all published in 2017); and I'll use QT4 to refer to the proposed 4.0 versions of these specifications which can be found at <https://qt4cg.org/>
- I'll use the term JTree to refer to an XDM data structure comprising a tree of maps and arrays; and I'll refer to the nodes in this tree as JNodes. This terminology isn't in the spec, but I find it convenient. The "J" stands for JSON, because these trees often originate from JSON data files, but it's important to remember that the XDM model for maps and arrays is much more liberal than JSON: for example, keys in a map can be any atomic type, not just strings.
- In XDM, the term "node" means a construct like an element, attribute, or text node in a tree that represents an XML document; and "tree" means a tree of such nodes. I won't be talking much about XML in this paper, so when I do, I will call these things "XML trees" and "XML nodes" to avoid any confusion.

2. What problem are we trying to solve?

One of the major new features in QT3 was support for maps and arrays as new data types. There were two main motivations for this. One was to extend the XDM data model with data structures that provided native representation of data found in JSON files: although we had no ambition to treat XML and JSON as equal citizens, we recognized that real-life applications had to be able to mix and match data from different sources. The second motivation was that users were

increasingly finding that representing everything (including transient working data) as XML could be cumbersome and inefficient.

Maps and arrays proved a very successful addition to the QT3 data model, but experience has shown that they are not yet as well supported by language features as XML node trees, and a significant part of the QT4 programme is aimed at filling the gaps.

At the same time, we're trying to come up with language features that exploit the intrinsic potential of maps and arrays to deliver performance benefits. In standards and specifications, the emphasis is on defining language features; performance objectives do not feature directly, but of course the design of language features is informed by the experience of implementors and users.

3. Previous work

Over the years I have written a number of applications that stretched the boundaries of what could be achieved with XSLT, and the challenges these posed are documented in a number of previous conference papers.

Back in 2007 [2] I experimented with using XSLT to write a query optimizer. I reckoned that implementing optimization rewrite rules is essentially a rule-based tree-to-tree transformation task, and in principle is therefore an ideal fit for XSLT. But my experiments suggested that doing tree rewrites in XSLT would be about 12 times slower than doing them in Java, which made the idea infeasible. But it set me thinking about why XSLT should be slower, and what could be done to fix it.

One potential way of avoiding the costs of transforming XML trees in such applications is to represent internal data using maps and arrays instead. So in 2016 [3] I explored the potential for using the QT3 facilities for transforming JSON. Although the XSLT 3.0 spec wasn't quite finished yet, development was on the home run and there wasn't really much scope for making further changes. That means the analysis in that paper of how to tackle JSON tree transformations still represents the state of the art; and it's clear that even simple transformations can be quite difficult. The conclusion of that paper was that there were significant limitations in the QT3 specifications for manipulating maps and arrays, and that indeed, the most practical way to implement several apparently-simple use cases was to convert the data to XML, transform it as XML, and then convert back to JSON.

At XML Prague in 2018 [4] I wrote about the problem of copying XML trees. In particular, I explored whether we could make tree copying faster by using a tree implementation that allowed sharing of subtrees. Node identities and parent pointers would be created on-the-fly during downwards navigation. The results were disappointing: while one particular operation (grafting a subtree) was dramatically faster, everything else (including all-important navigation operations)

slowed down a little, meaning there was no bottom-line benefit. But there's also an underlying problem: given a typical XSLT recursive-descent transformation, you can't tell when there's a particular subtree that isn't going to change, and that can therefore benefit from a fast copy operation. That's particularly true because even when you think a subtree is being copied unchanged, there's usually some hidden effect on the in-scope namespaces.

When I presented this paper, someone in the audience pointed out very politely that I had re-invented a technique well known in functional programming circles: zipper data structures, attributed to Gerard Huet [1].

At Markup UK in 2018 [5] I described the project that motivated some of this thinking: the task of writing an XSD schema validator using XSLT 3.0. This project (which was never finished to release quality) in fact made heavy use of maps and arrays as its internal data structures. (The use of maps here was nothing to do with JSON. It was primarily to enable a single pass over input XML trees to return multiple results: for example multiple ID values found in the tree, or multiple validation errors). The experience on this project led to the design of record types as found in the QT4 drafts, to enable stronger typing of maps used to hold heterogenous data; it also led to performance improvements in the way that maps are implemented in Saxon.

The following year, back in Prague [6], John Lumley and I described a project to implement an XSLT compiler in XSLT. The reason we used XSLT is that we needed a compiler that would run in the browser, and writing it in Javascript was too horrible to contemplate; in addition it seemed logical that since compilers are all about multiphase tree transformations, and that's what XSLT is supposed to be good at, it ought to be a natural fit. In any case, writing a compiler in its own language is always considered to be good for a software engineer's soul. But there were performance issues to tackle: issues already highlighted in the 2018 paper on efficient copy operations.

The paper contains a very detailed account of the techniques we used to get the performance up to a level where it was only three times slower than the existing Java compiler. Two of the significant factors were the need to represent complex values (such as data types) as strings so that they could be held in XML attributes, and the cost of copying XML trees, which turned out to be significantly caused by the complexity of getting the namespaces right. Unlike some of the previous projects described, this one did in fact complete, and resulted in the XSLT compiler that we use today in the SaxonJS project: though we did end up writing some critical parts (notably the XPath parser) in Javascript.

Most recently, my paper at Balisage in 2022 [8] covered similar ground to this paper, but the thinking has evolved since then so I feel that an update is overdue.

A theme that runs through a number of these papers is the dilemma of parent pointers. Should trees be implemented with parent pointers or not? Closely associated with this is the question of whether nodes should have a persistent, perma-

nent identity. As far as I'm aware, all tree models for XML have node identity and parent pointers, and all tree models for JSON don't. There's nothing intrinsic to XML or JSON that accounts for this difference, but it has become part of the culture. Without parent pointers and node identity, it's easy to share subtrees, which means that updates can be very efficient (you only need to copy the parts of the tree that actually change). With parent pointers, you end up copying the whole tree every time you want to make a small change. But without the ability to reference data higher up in the tree, some queries (especially on recursive data structures) become much harder to express. Perhaps the reason XML trees always have parent pointers is that XML is primarily designed for text processing, and textual data structures are intrinsically recursive.

It's worth pointing out that despite the limitations identified in this paper, maps and arrays in their QT3 form can be extremely useful. They play a key role, for example, in the Java-to-C# transpiler described in [7], where they are used in a way that doesn't encounter any of these problems.

4. The QT4 project

The project to create the 4.0 specifications was announced at XML Prague in 2020 [<https://archive.xmlprague.cz/2020/files/xmlprague-2020-proceedings.pdf>]; it took a while to build momentum but it is now in full swing, with well-attended Zoom meetings taking place weekly. To date 640 issues have been tabled, and several hundred of these have resulted in new features in the specifications. Thousands of test cases have been written, and both Saxon and BaseX have announced experimental implementations.

The project operates under the auspices of a W3C Community Group. As such, it receives some lightweight support and endorsement from the W3C organisation, but is largely free to do what it chooses. There is no requirement, for example, for specifications to go through milestones such as proposed and candidate recommendations, or for transitions in status to be approved by a formal vote among the wider W3C membership.

There is a great deal of new functionality in the QT4 specifications, much of it associated with support for maps and arrays, but in this paper I am going to concentrate on three aspects: recursive query, point update, and rule-based transformation.

The adjective deep here means that we are looking at maps and arrays not just as individual data structures, but at their use in combination to represent more complex data sets, typified by the representation of a complete JSON document. I refer to these as JTrees.

5. Overview

In the following sections I'm going to present three areas where the QT4 specifications offer new language features: deep query, point update, and rule-based transformation.

What I describe in the paper isn't necessarily identical with the current snapshot of the published specifications. In some cases there are proposals to change the current drafts that the community group is still working on. In some cases there is a broad consensus on the way forward, in other cases proposals have been put forward but not yet discussed, in other areas there have been discussions but alternatives are still on the table. So what I present here is a mix of what the current drafts actually say, and what I hope they will say in due course. It's very much a personal perspective. And with the best will in the world, the final specifications will have moved on from the current proposals. The process of agreeing language specifications may be slow, but it is fairly good and distinguishing what works and what doesn't, and at finding incremental improvements where they are needed.

6. Recursive Query

Let's look first at the question of recursive query.

The QT3 specifications offer the lookup operator `?` as a rough equivalent of the path operator `/` used for navigation in XML trees. It can be used both for selection by key within a map, and for selection by subscript within an array. Here is a simple example. Consider the example document used in the JSONPath specification, representing four books and a red bicycle:

```
{
  "store": {
    "book": [
      {
        "category": "reference",
        "author": "Nigel Rees",
        "title": "Sayings of the Century",
        "price": 8.95
      },
      {
        "category": "fiction",
        "author": "Evelyn Waugh",
        "title": "Sword of Honour",
        "price": 12.99
      },
      {
        "category": "fiction",
        "author": "Herman Melville",
```

```
    "title": "Moby Dick",
    "isbn": "0-553-21311-3",
    "price": 8.99
  },
  {
    "category": "fiction",
    "author": "J. R. R. Tolkien",
    "title": "The Lord of the Rings",
    "isbn": "0-395-19395-8",
    "price": 22.99
  }
],
"bicycle": {
  "color": "red",
  "price": 399
}
}
```

It is possible to find the average price of books by Tolkien with the query:

```
$data ?store ?book ?[ ?author = "J. R. R. Tolkien" ] ?price => avg()
```

Most of this is QT3 syntax, but there's one new QT4 construct here, namely the array filter expression `ARRAY?[PREDICATE]`. The takes an array, and filters it to retain the array members that match the predicate, in exactly the same way that XPath filter expressions have always been used to filter sequences.

To achieve the same effect in QT3, it is necessary to convert the array to a sequence by writing `book ? * [ ?author = "J. R. R. Tolkien" ]`. This works fine in this case where all the members of the array are single items (in this case, single maps), but is not useful in the general case where array members are arbitrary sequences: in such cases the QT3 solution is the higher-order `array:filter` function.

The limitations of the lookup operator start to become apparent when the data becomes more complex.

Firstly, QT3 offers no equivalent to the `//` operator used for searching the descendant axis of XML trees. The `//` operator is useful for a number of reasons:

- It's a handy shorthand: it avoids having to spell out long and complex paths in detail.
- It's useful where the same structure can occur at different places in the tree, for example when searching the above structure for a price, which might be the price of a book or of a bicycle. (The sample data here is a joke, but unfortunately that's often also true of data found in the real world.)
- It's invaluable when handling recursive data such as an organisation chart, where the same structures occur at different levels.

Secondly, this path-based syntax can't be used for join queries. That's equally true of XML-based path expressions using `/`, and the solution is the same: FLWOR expressions. Let's suppose that we want to know (somewhat surreally) how many books there are in the store that cost less than a bicycle. We can write:

```
let $bicycle-cost := ?store ?bicycle ?price
return count( ?store ?book ?[ ?price lt $bicycle-cost ] )
```

This still leaves another usability problem: all selections have to be downwards. There's no equivalent to the sibling, parent, or ancestor axes used when navigating XML. Now, it's arguable that these axes are most useful when searching text, and are less needed for JSON because no sane person would use JSON for representing text. But even with structured data, they can be very handy, especially when the data is recursive. A classic query is to find everyone who earns more than their manager: `//person[salary > ../salary]`. Similarly, the sibling axes are useful when data is ordered.

The `map:find()` function was a late addition to the QT3 specifications which attempted to provide an equivalent to the `//` operator, but it has proved almost entirely useless. The reason is that it only returns the values, it tells you nothing about the context in which they were found. You can find all the values of `first-name` in your entire dataset, but what can you do with the knowledge that the names that appear are John, Jane, and Mary? Finding all the values of `employee` (which are likely to be maps) is a bit more promising, but without the ability to navigate up the tree to discover context (like the department or location of each employee) the data is still of very limited value.

So let's look at how lookup paths have been improved in the QT4 specifications. There are a number of changes, some quite minor, some more significant. (In addition, I should emphasize, some of these are a solid part of the QT4 drafts, while others are proposals that are still being polished and refined.)

The most obvious change is that we've introduced the "deep lookup" variant, `A ?? B`. Like `//`, this recurses down through a JTree of maps and arrays to find its target. And like `//` (and like `map:find`), it returns a flattened sequence of items.

But there's a fundamental difference. When you return a flattened sequence of XML nodes, those nodes are located at a position within a structure, and you can get extra information about them: most commonly, you are selecting a sequence of elements, and from the elements you can get information about their attributes, their content, their siblings, their parents.

With `??`, all you get back are values. If you do `??name`, you get back a set of names. There's no context; no way of finding out any other information beyond the actual strings. If the values you get back are themselves maps, you can do a little bit better, you can drill down into the content of those maps. Remember, a map (unlike an XML element) doesn't even have a name. You might find it via a name (consider `store` and `bicycle` in the example above), but the name isn't part

of the map, it's not available as part of the information returned by the `??` operator.

The fact that maps and arrays (unlike XML elements) have no name, is rather fundamental. The expression `//event/*` selects elements that have names, and the names often serve to distinguish one event from another. The names are a property of the element. In JSON, names perform a subtly different role: they don't identify what kind of value you are dealing with, they identify its role in relation to a parent object, and an expression such as `??event?*` selects values without identifying what kind of object they are.

In JSON structures the best way to identify classes of value is often not by name, but by structure. If we want to process locations, we probably can't search for objects named `location`, we have to search for objects that have `longitude` and `latitude` properties. For that reason we've added type-based selectors to lookup expressions, so you can do `??type(record(longitude, latitude))` to find all the maps (objects) having `longitude` and `latitude` properties.

The fact that the lookup operators flatten their results brings additional problems. Suppose you have data like this:

```
{
  "readings": [
    { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ... }
    { "Week":2, "Mo": (4, 8), "Tu": (4, 5), "We": ... }
  ]
}
```

Now the result of `??Mo` is `(12, 16, 18, 4, 8)`, and the result of `??Tu` is `(4, 5)` which is pretty meaningless - all the internal structure has been lost. We just have a set of numbers, and no idea where they came from.

So the next change we have made is to introduce modifiers, which enable you to return something other than the flattened result. For example, the result of `??entry::Mo` is a sequence of key-value pairs:

```
{"key": "Mo", "value": (12, 16, 18)},
{"key": "Mo", "value": (4, 8)}
```

This retains a lot more structure - it becomes possible to ask questions like *"On which Monday was the average reading at its highest?"*.

The syntax allows four modifiers: `entry::` returns the key-value pair as in the above example, `key::` returns the key (when selecting into an array, this is the numeric index), `value::` returns the value part, with each value wrapped as an array so that different values are kept separate, and `content::` returns the flattened sequence. For compatibility with QT3, `?x` is short for `?content::x`.

The biggest change, however, is that the results returned by the `?` and `??` operators are now labeled with their provenance. This means that you can now do queries like `??entry::*[?value?salary > ?parent()?salary]`.

How does that work?

The maps and arrays in a JTree don't have parent pointers, and we're not changing that. Parent pointers prevent a subtree being shared by two different trees, which is the major reason why copying subtrees in the XML model is so expensive. Instead, though, when we search for data by downwards navigation, we can remember how we reached the values that we found, and we can make this information available. This is essentially the idea behind zipper structures [1].

The idea of a zipper structure is that as you navigate into a structure, you keep a trail of where you have visited, so that you can back out. This effectively turns a one-way list into a two-way list, or a tree without parent pointers into one that allows upwards navigation; and it does so without making the structure mutable, or losing the properties that allow functional modifications (modifications that create a new list or tree without destroying the old one).

The solution to this in QT4 is taken straight from the zipper model. The result of a deep search (which can now be done using the deep lookup operator: `$data??person`) is a set of labeled values. The expression returns the required JNodes, but these carry an extra label identifying where they were found. The label is not part of the persistent data, it is additional information added by the search process, so the same value found by two different routes can carry different labels. We can immediately see how this allows common subtrees to be shared without the need for physical copying.

What information is available in the labels? We can describe this at two levels: an internal level that describes the data that is retained, and an external level that describes how this is exposed to user applications. At the internal level, there are two degrees of granularity associated with a downwards selection: selecting an entry with a given key (which in the case of an array means the array member at a given index), and selecting a specific item within the value part of the entry.

So, when selecting an entry (key-value pair) the information retained in the label consists of two things: the containing JNode (a map or array), and the key/index by which the entry was selected within that JNode. Of course the containing JNode will also have a label identifying its own provenance, so one can follow a chain of labels all the way to the root JNode of the query.

At the next level of granularity, when we select an individual item within a value, there is an additional piece of information: the integer position of the item within the value.

Looking back at the example where the query `??Mo` returned the sequence (12, 16, 18, 4, 8), we can now see that these items carry hidden labels as follows:

- 12: container: { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}, key: "Mo", position: 1

- 16: container: { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}, key: "Mo", position: 2
- 18: container: { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}, key: "Mo", position: 3
- 4: container: { "Week":2, "Mo": (4, 8), "Tu": (), "We": ...}, key: "Mo", position: 1
- 8: container: { "Week":2, "Mo": (4, 8), "Tu": (), "We": ...}, key: "Mo", position: 2

There are two ways this data is currently exposed (this may change):

- Firstly, the function `selection-path` can be applied to any item. If the item has a label indicating how it was reached, the function returns a sequence of records, one for each step in the selection (in innermost-to-outermost order). The first record in the result, for the value 12 in the example above, is the record `{"key": "Mo", "position": 1, "container": { "Week":1, "Mo": (12, 16, 18), "Tu": (), "We": ...}}` The second and subsequent records represent the selection path of the container, defined recursively. So if the `selection-path` function is called with the labeled value 12 as the argument, then (for example), `selection-path(12)?key` will return the sequence `("Mo", 1, "readings")`; reversing this sequence gives the sequence of keys needed to select the item from the root.
- Secondly, when the modifier `entry::` is used, the returned entries include functions that make the same information available, but in more digestible form. The proposed functions include `ancestors()`, which returns all the nested JNodes, innermost first, `parent()` which returns the same as `ancestors()[1]`, `root()` which returns `ancestors()[last()]`, and `ancestor-keys()` which returns the sequence of key values used to select the ancestors at each level.

7. Point Update

In my 2016 paper I presented two use cases for transformation of JSON trees, and I have continued to use these to test the adequacy of proposed new language features.

The first use case is what I call a "point update": *increase the price of all products tagged with the keyword "ice" by 10%*. I call it a "point update" because we identify the places in the tree structure that need to change, and then say how they should change. That requirement is very easily stated in English, and it's not very difficult to devise XQuery or XSLT syntax to express the requirement, for example in XQuery

```
update $root {  
  replace ??product[?tag = "ice"]?price with . * 1.1  
}
```

or in XSLT

```
<xsl:update root="$root"  
  replace="??product[?tag = 'ice']?price"  
  with=". * 1.1"/>
```

In both cases the idea is that the instruction returns a new version of the tree rooted at `$root` (leaving the original intact) which differs from the original in that the selected prices have changed.

This is sometimes called *in-situ update*, but that's misleading, because existing data is not changed.

This is similar in concept to the kind of updating expressions available in the XQuery Update Facility (XQUF). However, it has far less complexity. XQUF essentially has two modes of operation:

- Pending Update Lists, where the updates are deferred until the query has finished execution, and the query itself has no opportunity to read the updated data;
- Copy-Modify operations, where the updates are performed on a copy of the data, leaving the original unchanged. But with XML (because of XML node identity and parent pointers) copying an XML tree typically takes time and memory proportional to the size of the tree.

The current proposed specification for this feature describes the semantics in terms of a recursive-descent rule based transformation, rather as if it were implemented in XSLT. The only complication is how to tell when you are processing a value selected by the expression `??product[?tag = 'ice']?price`. One way to do this would be to restrict the syntax of this expression to a pattern-like syntax so you can test each value in the tree when you get to it. But if you turn to the semantics of patterns in XSLT, the definition depends on being able to navigate upwards in the tree. Instead, the approach we have adopted is that you can use any selection expression you like, and the values you select are tagged with a temporary label, so that you can identify them during the recursive descent traversal, and process them accordingly.

The QT3 data model says:

This version of the XPath Data Model does not specify whether or not maps have any identity other than their values. No operations currently defined on maps depend on the notion of map identity. Other specifications, for example, the XQuery Update Facility, may make identity of maps explicit.

That was a committee compromise. The WG knew that maintenance of identity is expensive in many ways: not only does it create the need to make physical copies of

data that is otherwise unchanged during a transformation, it also complicates the semantics of the language because functions are no longer purely functional (if a function creates new node each time it is called, then optimizations such as loop lifting become invalid). But the WG at the time couldn't see how update operations could be defined without some notion of object identity.

The solution adopted in QT4 is to have a notion of identity, but one that is transient and exists only while an updating operation is in progress. Conceptually, the update starts by making a copy of the input in which all JNodes have identity; it then modifies this copy in-situ, and then strips off the identifiers.

The great thing is, however, that this complexity exists only in the formal semantics of the operation. As far as users are concerned, it's fairly intuitive what's meant by

```
update $root {  
  replace ??product[?tag = "ice"]?price with . * 1.1  
}
```

Performing this operation in QT3, whether in XQuery or XSLT, is surprisingly difficult, as I showed in my 2016 paper. I ended up converting the data to an XML tree, updating the XML tree, and then converting back to maps and arrays.

The implementation of this update expression in Saxon is very different from the formal semantics¹. Rather than doing a top-down traversal of the whole tree structure (which would take time proportional to the size of the tree), Saxon dives straight in to the selected nodes and makes the changes locally. Internally, you have to find the affected nodes in the tree, make local copies of those nodes with modified properties, and then you have to work your way back up the tree making copies of the affected parent nodes so they point to the modified children rather than to the originals. This needs to take into account that when creating a modified parent node, you might need to incorporate multiple modified child nodes. Finally, when you've worked your way back up to creating a modified copy of the original root node, you can return that as the result of the expression.

So for this use case, we don't need to expose the zipper model to the user; but it needs to be there in the background; the route to each modified node in the tree needs to be retained so that the ancestor nodes can be reconstructed.

The second use case in my 2016 paper involved a hierarchic inversion. Starting with a JSON dataset representing courses and the students attending each course, the requirement is to invert this to create a dataset organised first by student, listing the courses taken by each student. This is more of a tree transformation task than a point update task, and I tackle it in the next section.

¹Saxon has had an implementation of something very similar for some years: see <https://www.saxonica.com/documentation12/index.html#!extensions/instructions/deep-update>.

8. Rule-based Transformation

For earlier thoughts on this subject, see the section "Template-based Transformation" in my Balisage 2022 paper: [8].

Transformation by applying template rules during the course of a recursive tree walk is the characteristic processing model of the XSLT language, and there is no intrinsic reason why it shouldn't work just as well with JTrees as with XML trees. The main challenge is that it's tricky to define match patterns when the things we are matching don't have distinguishing names.

There are three parts to the problem: how to break arrays and maps down into components that can conveniently be matched by template rules; how to define the corresponding match patterns; and how to construct new arrays and maps within the body of the corresponding template rules.

There are proposals for how to do this in the current XSLT 4.0 draft specification, and further ideas in my 2022 Balisage paper, but the more recent work on deep query suggests enhancements to these features that are not yet fully worked out. So what follows is my current proposal for a revision to the draft specification.

A good way to approach this is to start by thinking about what the standard default template rules should do. The desired effect is that we can define a mode of processing (say with `<xsl:mode on-no-match="traversal"/>`) which has the effect that if there are no user-written template rules in the mode, the effect is an identity transformation. But we want to design it so that it becomes easy to inject user-written template rules to customize the processing of particular constructs.

So let's start by defining a set of built-in rules for this new processing mode.

8.1. Built-In Rules

For arrays, we want the default rule to process all the members of the array. The question is, how should the array members be represented? Remember that an array member is in general a sequence rather than a single item. Although QT4 has generalised the concept of the context item so that it is now a context value (which can be any sequence), for the time being, XSLT templates are still applied to individual items, so we need to package up each array member as a single item.

In my Balisage 2022 paper I proposed representing an array member as a "parcel", which was essentially a sequence wrapped up as a single item. That raises the question, should parcels be a new kind of item in the data model, or should we reuse some existing kind of item (candidates being an array, a zero-arity function, or a single-entry map). The problem with introducing a new kind of item is that extensions to the type system are expensive and disruptive. The problem with reusing existing kinds of item is that it's harder to define match patterns that match them nicely.

My current proposal is to represent array members as records (that is maps) with (at least) the following fields:

- `array-member`: a boolean, always true. This field exists purely for convenience in writing match patterns: a template rule can use `match="record(array-member, *)"` with a high level of confidence that the rule won't accidentally match something else.
- `member`: the actual value of the array member, an arbitrary sequence.
- `index`: the 1-based index position of the array member within the array (just in case different processing is applied to different members based on their position).

The template rule for an array member is required to return a new array member; this must be represented by a record containing a `member` field, but the other fields are optional and ignored.

So the default template rule for arrays breaks up the arrays into its members, applies templates to each of them individually, and then reassembles the result:

```
<xsl:template match="array(*)">
  <xsl:array use="?member">
    <xsl:apply-templates select="
      for member $m at $index in .
      return {'array-member':true(), 'member':$m, 'index':$index}"
      mode="#current"/>
  </xsl:array>
</xsl:template>
```

A few observations on this code:

- `match="array(*)"` is one of a new class of match patterns that match items by type. In XSLT 3.0 this was written, clumsily, as `match=". [. instance of array(*) ]`.
- `xsl:array` is a new instruction to construct an array. The contained sequence constructor delivers one item for each member of the new array. If a `use` attribute is present, then it is evaluated once for each of these items, to convert the supplied item to the required array member. In this case we expect the applied template rule to return a value of type `record(member)`, and the `use` expression extracts the contents of the `member` field to form the array member.
- `for member` is a new variant of the `for` expression (or XQuery FLWOR expression) that binds the variable `$m` to each member of the array in turn, as well as binding `$index` to its 1-based index position.
- The expression `{'x':1, 'y':2}` is a map constructor, equivalent to the XPath 3.1 expression `map{'x':1, 'y':2}`. The `map` keyword is no longer needed. (It was required in XPath 3.1 because some members of the design team wanted

to reserve "bare brace" syntax with no leading keyword for a different purpose.)

The default processing rule for members of an array is to apply templates to each item in the value of the member individually: recall that in the general case, an array member includes zero or more items. The processing returns a new array member that replaces the original:

```
<xsl:template match="record(array-member as xs:boolean,
                           member as item()*,
                           index as xs:integer,
                           *)">
  <xsl:apply-templates select="
    for $item at $pos in ?member
    return {'array-member-item': true(),
            'member': $m,
            'index': $index,
            'item': $item,
            'position': $pos}"
    mode="#current"/>
</xsl:template>
```

Observations:

- It's probably unlikely that many users would want to override processing at this level; but it provides the option for completeness. A template rule at this level has access to the entire array member and its index position within the array, and also to an individual item within the array member and its position within the array member.
- A template rule at this level is expected to return one or more items, which will substitute for the original item in the new array member.

The default processing for individual items is then to apply templates to the item:

```
<xsl:template match="record(array-member-item as xs:boolean,
                           member as item()*,
                           index as xs:integer,
                           item as item(),
                           position as xs:integer,
                           *)">
  <xsl:apply-templates select="?item" mode="#current"/>
</xsl:template>
```

And the default processing for items is to move the item to the output unchanged:

```
<xsl:template match="item()">
  <xsl:sequence select="."/>
</xsl:template>
```

Note that this just creates a reference to the existing item, it doesn't require making a deep copy of the item. This makes a big difference if an array or map contains XML nodes. The transformed JTree will contain the original XML nodes, rather than copies of these nodes. If copying or transformation of the XML nodes is required, this can be achieved by overriding the template rule for the individual XML nodes.

Maps are handled in a similar way to arrays. The top-level default template rule for maps splits it into its constituent entries (key-value pairs), marking each one with a `map:entry` field for ease of matching:

```
<xsl:template match="map(*)">
  <xsl:map on-duplicates="op(',')">
    <xsl:apply-templates select="
      for entry {$key, $value} in .
      return {'map-entry':true(), 'key': $key, 'value':$m}"
      mode="#current"/>
  </xsl:array>
</xsl:template>
```

Observations:

- `match="map(*)"` is another new type-based match pattern, matching any instance of the specified type.
- The `xsl:map` instruction exists in XSLT 3.0, but the `on-duplicates` attribute is new: it controls what happens when the sequence constructor delivers more than one map entry with the same key. The value is a function that in general takes two values having the same key and returns a single value; the expression `op(',')` returns an arity-2 function equivalent to the dyadic `,` (comma) operator which forms the sequence-concatenation of two values.
- The `for entry {$key, $value}` construct is a proposed extension to the `for` or `FLWOR` expression syntax for iterating over the entries in a map; it has been mooted as proposed XPath 4.0 syntax, but is not yet in the draft specification.

The default template rule for processing map entries looks like this:

```
<xsl:template match="record(map-entry as xs:boolean,
                           key as xs:anyAtomicType,
                           value as item()*,
                           *)">
  <xsl:map-entry key="?key">
    <xsl:apply-templates select="
      for $item at $pos in ?value
      return {'map-entry-item': true(),
        'key': ?key
        'value': ?value,
        'item': $item,
        'position': $pos}"
```

```
        mode="#current"/>
    </xsl:map-entry>
</xsl:template>
```

And again, the default processing at this level is to apply templates to the individual items directly:

```
<xsl:template match="record(map-entry-item as xs:boolean,
                           key as xs:anyAtomicType,
                           value as item()*,
                           item as item(),
                           position as xs:integer,
                           *)">
    <xsl:apply-templates select="?item" mode="#current"/>
</xsl:template>
```

Once again, it is unlikely that user applications would want to override the processing at this level, but the option is there for completeness.

Not stated in the details above is that each of these built-in templates passes any template parameters through unchanged.

In addition, the code for the built-in templates fails to show explicitly that each item passed to any of these template rules is labeled with provenance information indicating its position within the JTree being navigated. The existence of this label means that the function `selection-path` can be applied to the item to obtain information about the route used to select the item. The result of the function is a sequence of records, from innermost to outermost order, each containing some or all of the following fields:

- `container`: the map or array containing the value.
- `key`: the key (for an entry in a map) or index (for a member in an array) of the value within its container.
- `position`: the index position of an individual item within an array member or map value, when that value is a sequence of items.

Note that while built-in template rules maintain this information and pass it on to the template rules they call, user-written template rules may or may not do so, depending on how they are written. If they use the lookup operators `?` and `??` to make downwards selections, then the provenance of the values they select is maintained.

8.2. Use Cases

The most common use case is to override processing at the level of an individual entry in a map. Template rules at this level are expected to return zero or more map entries, in the format delivered by the `xsl:map-entry` instruction. Here are some examples of template rules that do this:

```
<xsl:template match="record(map-entry, *) [?key='note']"/>

<xsl:template match="record(map-entry, *) [?key='mark']">
  <xsl:map:entry key="'mark'" select="upper-case(?value)"/>
</xsl:template>

<xsl:template match="record(map-entry, *) [?key='price']">
  <xsl:next-match/>
  <xsl:map:entry key="'currency'" select="'USD'"/>
</xsl:template>
```

The first of these template rules (with `[?key='note']`) returns no output, so the matching map entries are deleted.

The second (with `[?key='mark']`) returns a map entry having the same key, but with the value converted to upper case.

The third (with `[?key='price']`) performs the built-in processing for map entries (by invoking `xsl:next-match`) and then adds a new map entry, setting currency to "USD".

The transformation presented earlier (increase the price of all products tagged with the keyword "ice" by 10%) can be achieved in a number of ways. One way is to match selected product entries and then to modify the price directly using `map:put`:

```
<xsl:template match="record(map-entry, *)
  [?key='product']
  [?value?tag = 'ice']">
  <xsl:map:entry key="'product'">
    <xsl:sequence select="map:put(?value, 'price', ?value?price *
1.1)"/>
  </xsl:map:entry>
</xsl:template>
```

8.3. Grouping

While some transformations benefit from the rule-based recursive-descent paradigm, others involve wholesale reconstruction of a new document. Grouping and hierarchic inversion tasks often fall into this category.

Consider this use case from my 2016 paper, involving hierarchic inversion. The aim is to start with a JSON file containing a list of faculties and courses and the students enrolled on each course, and to invert this to produce a file containing a list of students, with the courses that each one is taking. Specifically, this is the input:

```
[{
  "faculty": "humanities",
  "courses": [
```

```
{
  "course": "English",
  "students": [
    {
      "first": "Mary",
      "last": "Smith",
      "email": "mary_smith@gmail.com"
    },
    {
      "first": "Ann",
      "last": "Jones",
      "email": "ann_jones@gmail.com"
    }
  ]
},
{
  "course": "History",
  "students": [
    {
      "first": "Ann",
      "last": "Jones",
      "email": "ann_jones@gmail.com"
    },
    {
      "first": "John",
      "last": "Taylor",
      "email": "john_taylor@gmail.com"
    }
  ]
}
],
{
  "faculty": "science",
  "courses": [
    {
      "course": "Physics",
      "students": [
        {
          "first": "Anil",
          "last": "Singh",
          "email": "anil_singh@gmail.com"
        },
        {
          "first": "Amisha",
          "last": "Patel",

```

```
        "email": "amisha_patel@gmail.com"
      }
    ]
  },
  {
    "course": "Chemistry",
    "students": [
      {
        "first": "John",
        "last": "Taylor",
        "email": "john_taylor@gmail.com"
      },
      {
        "first": "Anil",
        "last": "Singh",
        "email": "anil_singh@gmail.com"
      }
    ]
  }
]
}}
```

and this is the desired output:

```
[
  {
    "email": "ann_jones@gmail.com",
    "courses": [
      "English",
      "History"
    ]
  },
  {
    "email": "amisha_patel@gmail.com",
    "courses": ["Physics"]
  },
  {
    "email": "anil_singh@gmail.com",
    "courses": [
      "Physics",
      "Chemistry"
    ]
  },
  {
    "email": "mary_smith@gmail.com",
    "courses": ["English"]
  },
]
```

```
{
  "email": "john_taylor@gmail.com",
  "courses": [
    "History",
    "Chemistry"
  ]
}
```

In XSLT terms, this is clearly a grouping query. We want to select the students from the input file, group them by email address, and for each email address, output the courses attended by that student.

Here's the solution. The critical dependency is the call on `ancestors` which enables us to trace back to the JNodes visited during the deep lookup operation `json-doc('courses.json')??email`.

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="4.0">

  <xsl:template name="xsl:initial-template">
    <xsl:array>
      <xsl:for-each-group select="json-doc('courses.json')??
entry::email" group-by="?value">
        <xsl:sort select="current-grouping-key()" />
        <xsl:map>
          <xsl:map-entry key="'email'"
            select="current-grouping-key()" />
          <xsl:map-entry key="'courses'"
            <xsl:array select="current-group() ? ancestors() ?
course"/>
          </xsl:map-entry>
        </xsl:map>
      </xsl:for-each-group>
    </xsl:array>
  </xsl:template>

</xsl:transform>
```

Notes on this solution:

- `entry::email` selects a sequence of map entries or key-value pairs in the form `record(key, value)`; the record also includes an `ancestors` field which is a function providing access to the containing maps and arrays.
The grouping key `?value` is the actual email address.
- The expression `current-group() ? ancestors() ? course` doesn't currently work. The LHS of a dynamic function call must be a single function, but `current-group() ? ancestors` is a sequence of functions. I've raised an issue

to fix that. To make it work without this fix, it needs to be written as
(current-group() ! ?ancestors()) ? course

We can also solve this use case using XQuery, taking advantage of the `group by` clause in a `FLWOR` expression. The following should work:

```
array {  
  for $email in json-doc('courses.json')??entry::email  
  order by $email?value  
  group by $email?value  
  return {  
    'email': $email?value,  
    'courses': array { for $e in $email  
                        return $e?ancestors()?course }  
  }  
}
```

In the interests of full disclosure, I am using a development version of Saxon that implements a slightly different version of the syntax from the snapshot presented in this paper (we are dealing here with rapidly evolving specifications and proposals, and the implementation is often either ahead of the spec or a little behind it). With this version of Saxon, the XSLT code that actually works is:

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"  
version="4.0">  
  
  <xsl:template name="xsl:initial-template">  
    <xsl:array>  
      <xsl:for-each-group select="pin(json-doc('courses.json'))??  
email" group-by=".">  
        <xsl:sort select="current-grouping-key()"/>  
        <xsl:map>  
          <xsl:map-entry key="'email'"  
                        select="current-grouping-key()"/>  
          <xsl:map-entry key="'courses'">  
            <xsl:variable name="labels" select="current-group() !=  
> label()"/>  
            <xsl:variable name="ancestors" select="($labels ! ?  
ancestors())[. instance of record(course, students)]"/>  
            <xsl:array select="$ancestors?course"/>  
          </xsl:map-entry>  
        </xsl:map>  
      </xsl:for-each-group>  
    </xsl:array>  
  </xsl:template>  
  
</xsl:transform>
```


And the working XQuery code is:

```
array {  
  for $email in pin(json-doc('courses.json'))??email  
  order by $email  
  group by $email  
  return {  
    'email': $email,  
    'courses': array { for $e in $email  
                        return label($e)?ancestors()[. instance of  
map(*)]?course }  
  }  
}
```

9. Conclusions

A major theme running through the proposals for XSLT 4.0 and XQuery 4.0 is improved support for manipulation of maps and arrays, both to make the languages more suitable for processing JSON, and also to improve the usability and performance of maps and arrays when used for internal working data within an XML transformation.

My 2016 paper at XML Prague demonstrated the shortcomings of the 3.0/3.1 specifications for achieving some simple use cases in this regard; with that version of the language, the best solution for JSON processing has often been to convert the JSON to XML, transform the XML, and then convert back to JSON.

This paper describes how these challenges have been addressed in the proposals for XSLT 4.0 and XQuery 4.0. It describes new language features in three main areas:

- Recursive Query
- Point Update
- Rule-based Transformation

and revisits the 2016 use cases to show how the new features work together to solve the problem.

References

- [1] Gerard Huet. *The Zipper*. Journal of Functional Programming. 7 (5): 549–554 doi:10.1017/s0956796897002864. S2CID 31179878.
- [2] Michael Kay. *Writing an XSLT Optimizer in XSLT*. Extreme Markup Languages, Montreal, 2007. Available at <http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html>.

- [3] Michael Kay. *Transforming JSON using XSLT 3.0*. XML Prague 2016. Available at <https://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf> and at <https://www.saxonica.com/papers/xmlprague-2016mhk.pdf>.
- [4] Michael Kay. *XML Tree Models for Efficient Copy Operations*. XML Prague 2018. Available at <https://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf> and at <https://www.saxonica.com/papers/xmlprague-2018mhk.pdf>.
- [5] Michael Kay. *An XSD 1.1 Schema Validator Written in XSLT 3.0*. Markup UK 2018. Available at <https://markupuk.org/2018/Markup-UK-2018-proceedings.pdf> and at <https://www.saxonica.com/papers/markupuk-2018mhk.pdf>.
- [6] Michael Kay and John Lumley. *An XSLT compiler written in XSLT: can it perform?*. XML Prague 2019. Available at <https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf> and at <https://www.saxonica.com/papers/xmlprague-2019mhk.pdf>.
- [7] Michael Kay. *<transpile from="Java" to="C#" via="XML" with="XSLT"/>*. Markup UK 2021. Available at <https://markupuk.org/2018/Markup-UK-2021-proceedings.pdf> and at <https://www.saxonica.com/papers/markupuk-2021mhk.pdf>.
- [8] Michael Kay. *XSLT Extensions for JSON Processing*. Balisage 2022. Available at <https://www.balisage.net/Proceedings/vol27/html/Kay01/BalisageVol27-Kay01.html>.

JSONPath: an IETF Proposed Standard, with comparisons to XPath

Alan Painter
HSBC Continental Europe
<alan.painter@hsbc.fr>

Abstract

The Internet Engineering Task Force (IETF) have recently promoted to the status of "Proposed Standard" the RFC9535 entitled "JSONPath: Query Expressions for JSON". [1] This new proposed standard cites in Section 1.2 that JSONPath is "inspired by XML's XPath". As the XML community has used different versions of XPath for querying XML for many years, extended more recently to querying JSON via XPath 3.1, a comparison of the relatively new JSONPath proposed standard and XPath can be of some utility.

Keywords: XML, JSON, Query, Standards

1. A newly proposed internet standard for an established query language

Whereas JSONPath as a standard is quite recent (February 2024), many "JSON-Path" implementations have been available since at least 2007 when JSONPath was originally proposed by Stefan Gössner.[4]

A number of commercial and open-source products use JSONPath or similar queries as part of their frameworks.

- REST-assured, a Java/Kotlin testing framework for REST APIs [5]
- SmartBear™, a testing and monitoring framework for REST and SOAP APIs[6]
- Kubernetes™, the popular container orchestration engine [7]
- Oracle™ products use JSONPath in a number of offerings, including the Process Modelling Framework (PMF), part of the Oracle Financial Services Analytical Applications Infrastructure (OFS AAI) [8]
- IBM™'s Netezza Performance Server (NPS) data warehouse and analytics platform [9]
- many other products

Because the JSONPath proposed standard is quite new, many of the existing offerings that predate RFC9535 will not be entirely compatible with the new proposed standard. There is an inventory of nearly fifty (50) JSONPath implementations in over twenty (20) different programming languages in Christoph Burgmer's github repository **json-path-comparison**. [2] This comparison catalogs the results of more than two hundred (200) different JSONPath queries for each implementation, giving the "consensus" result as well as the differing results.

One observation of the different query results is that there are many divergences in results from implementation to implementation. In this respect, RFC9535 is addressing these divergences by providing a clear standard. Section 1.2 of RFC9535 mentions that this is one of the motivations for proposing the standard.

2. Key events in the history of XML, JSON, XPath and JSONPath

As a starting point to this comparison of JSONPath and XPath, it can be enlightening to see an cherry-picked list of some the major events in their respective definitions.

Table 1. A list of selected key events in the XPath / JSONPath standards history

Date	Org	Document	Author(s)	Status	References
1998-02	W3C	Extensible Markup Language (XML) 1.0	Bray, Paoli, Sperberg-McQueen	REC	[10]
1999-11	W3C	XML Path Language (XPath) Version 1.0	Clark, DeRose	REC	[11]
2006-07	IETF	Media Type for JSON	D. Crockford	INF	[12]
2006-08	W3C	Extensible Markup Language (XML) 1.1	Bray, Paoli, Sperberg-McQueen, et al	REC	[13]
2007-02	Blog	JSONPath - XPath for JSON	Gössner		[4]
2010-12	W3C	XML Path Language (XPath) 2.0	Berglund, Kay, Robie, et al	REC	[14]
2010-12	W3C	XQuery 1.0 and XPath 2.0 Data Model (XDM)	Berglund, Walsh, et al	REC	[15]
2013-10	ECMA	JSON Data Interchange Format		REC	[16]
2014-03	IETF	JSON Data Interchange Format	Bray		[17]
2017-03	W3C	XML Path Language (XPath) 3.1	Robie, et al	REC	[18]
2017-03	W3C	XQuery and XPath Data Model 3.1	Walsh, Snelson, et al	REC	[19]
2017-12	IETF	JSON Data Interchange Format	Bray	STD	[20]
2024-02	IETF	JSONPath: Query Expressions for JSON	Gössner, Normington, Bor-mann	REC	[1]

Status: **REC** = Recommendation, **INF** = Informational, **PSTD** = Proposed Standard, **STD** = Internet Standard

One point of interest is that both the W3C XML recommendations (XML1.0 and XML1.1) and the IETF JSON recommendations (RFC7159 and RFC8259) list Tim Bray as being either amongst the editors (XML) or the sole editor (JSON).

Furthermore, Tim Bray is one of the co-chairs of the IETF Working Group that was created for producing the JSONPath standard.[3]

3. The JSONPath standard basics

3.1. Review of the JSON structure

As defined in IETF STD90 / RFC8259 [20], JSON has a minimalist set of values:

Table 2. Table of JSON Values (RFC8259)

Name	Examples	Category	Notes
Object	{ "a" : 1.00, "b" : "hello", "c" : true, "d" : null }	Structured	Unordered
Array	[0, 1, true, null, [{ "a" : "A" }], []]	Structured	Ordered
Number	42, 3.14159, 1.45e-9	Primitive	No distinct float/integer types
String	"hello", "", "tr\u00e9s bien"	Primitive	Unicode
Literal Boolean	true, false	Primitive	
Literal null	null	Primitive	

3.2. Additional data types added in RFC9535 for JSONPath evaluation

RFC9535 describes a few additional Value types that are required to process JSON via JSONPath:

Table 3. Types defined for JSONPath processing (IETF RFC9535)

Type	Instances	Notes
Nothing		When no JSON Value is selected, distinct from JSON literal <i>null</i>
Nodelist	5 {} null "hello" true [1, 2, 3]	A list of JSON Nodes (really Values)
ValueType	JSON Values or Nothing	Function extension type
LogicalType	LogicalTrue or LogicalFalse	Function extension type, distinct from literal true/false
NodesType	Nodelists	Another function extension type

The requirement in JSONPath to define a **odelist** type seems to be quite similar to the XPath1.0 [11] requirement to define the **node-set** type. In both XPath and JSONPath, the execution of the query yields an intermediate result that cannot be expressed in XML or JSON directly, hence a new type definition is made. Whereas the XPath1.0 **node-set** is described as "an unordered of nodes without duplicates", the JSONPath **odelist** is both ordered and may contain duplicates as well as primitive values.

JSONPath describes a **Nothing** type that is distinct from the JSON *null* literal. This allows for distinguishing between a *null* literal from a JSON Value and selecting no JSON values at all, hence **Nothing**.

3.3. The Overall Structure of a JSONPath Query

JSONPath describes three major elements of an overall JSONPath evaluation:

Table 4. Elements of a JSONPath evaluation

Name	Type	Description
Query Expression	String	JSONPath expression to be evaluated against a JSON Value
Query Argument	JSON Value	The query expression will be evaluated against this value
Output Values	Nodelist	The list of JSON Values from the Query Argument selected by the JSONPath Expression
Output Locations	Part of Nodelist	RFC9535 defines a JSON <i>node</i> from a <i>odelist</i> as a "pair of a value along with its location", where the "location" is a JSONPath expression that designates the position of that Value in the Query Argument. More discussion in the following sections.

RFC9535 Section 2.3.1.3 provides these expression examples and results against the following JSON Value used as the Query Argument:

JSON Query Argument:

```
{
  "o" : { "j j" : { "k.k" : 3 } },
  "" : { "@ " : 2 }
}
```

Table 5. Some examples of JSONPath evaluations

Query	Result Values	Result Locations
<code>\$.o['j j']</code>	<code>{ "k.k" : 3 }</code>	<code>\$['o']['j j']</code>
<code>\$.o['j j']['k.k']</code>	3	<code>\$['o']['j j']['k.k']</code>
<code>\$.o["j j"]["k.k"]</code>	3	<code>\$['o']['j j']['k.k']</code>
<code>\$["'"]["@"]</code>	2	<code>\$['\']['@']</code>

3.4. JSONPath Identifiers, Segments and Selectors

A JSONPath expression can be described as a series of Identifiers, Segments and Selectors

3.4.1. JSONPath Identifiers

There are two identifiers in JSONPath

Table 6. JSONPath Identifiers

Symbol	Usage
\$	the root identifier, corresponding to the entire <i>query argument</i>
@	the current node (in the context of a filter selector expression)

A JSONPath expression *must* start with a **root identifier** (\$), not even allowing a space before this initial, obligatory identifier. In effect, this requires that the initial, singular *odelist* evaluated by the JSONPath query is the *Query Argument*.

The **current node identifier** (@) is used within a *Filter Selector* expression as described below.

3.4.2. JSONPath Segments

For those familiar with XPath, a JSONPath *segment*, in its canonical form, presents itself much like an *XPath predicate*, using square braces ('[' and ']') to delineate the *segment*.

Table 7. JSONPath Segments (Canonical Form)

Form	Type	Usage
<code>[(selectors)]</code>	Child Segment	its selectors choose 0 or more children of the <i>odelist</i> from any preceding segments

Form	Type	Usage
<code>.. [(selectors)]</code>	Descendant Segment	its selectors choose 0 or more descendants of the <i>odelist</i> from any preceding segments

The semantics of the JSONPath *segment* is that it processes the *odelist* that is the result of any preceding JSONPath *segments*, with the initial *segment* taking the *singular oodelist* consisting of the *root identifier* (\$) as the obligatory initial *odelist*.

The essential structure of the JSONPath query with *canonical segments* is then a *root identifier* (\$) followed by a (possibly zero-length) series of *segments*.

`$ [ (selectors) ] [ (selectors) ] ...`

The output of a *segment* (i.e. the *odelist* presented to either the following *segment* or as the output of the JSONPath Query, in the case of the final *segment*) contains either **children** or **descendants** of the JSON values of the incoming *odelist*. This means that, for a series of **N** *child segments* in a query, the output of the JSONPath Query will contain only values that are at depth of **N** in the Query Argument. For a series of **M** *segments* which include at least one *descendant segment*, the output of the JSONPath Query will contain only JSON values at a depth of **M** or greater in the Query Argument. (RFC9535 Section 2.5)

Whereas the XPath *predicate* has a similar form to the *canonical JSONPath segment*, the semantics are different. Where XPath *predicate* will "filter" the input *node-set* (XPath 1.0) or *sequence* (XPath 2.0 and later), producing a subset of the input values, the JSONPath segment will produce *children* or *descendants* of the input JSON *node-list*.

3.4.2.1. Multiple selectors in a segment

A JSONPath segment contains at least one JSONPath selector and can contain a comma-separated (', ') list of selectors. Each JSON value in the input *odelist* is presented to the *selectors* in the *segment*, in order, for processing. A value from the input *odelist* appears on the output *odelist* as many times as it is selected by the different *selectors* and in the order of its appearance in the list of *selectors*.

3.4.2.2. Descendant Segment Traversal Order

In the case of a *descendant segment*, the child values of the input *odelist* and their *descendants* are presented to the *selectors* breadth-first. Arrays elements are visited in order. The values of object members are visited in any order because JSON objects are unordered.

3.4.2.3. JSONPath Segment Shorthand

Whereas the canonical form for JSONPath *segments* is expressed with square braces ('[' and ']'), there is a shorthand form which, after a fashion, combines the

segment with the *selector*. These shorthand forms can be converted to canonical form without changing the semantics of the *JSONPath Query*. This paper presents the shorthand forms below, after presenting the *selectors*.

3.4.3. JSONPath Selectors

The JSONPath *segment* contains a list of at least one JSONPath *selector*. JSONPath includes **five** different types of *selector*. A selector can choose values from a JSON structured value (i.e. object or array) but chooses *Nothing* from a *primitive* JSON value.

Table 8. JSONPath Selectors

Examples	Type	Selects from
['key'] or .key	Name Selector	a member value from an <i>object</i>
[*] or .*	Wildcard Selector	chooses all elements of an array (in order) and all member values from an object (no order)
[2] or [-1]	Index Selector	selects a single element from a array at the 0-based index. A negative index value is added to the size of the array to obtain the 0-based index value. (e.g. -1 chooses the last element in the array)
[1:5:2] or [::-1]	Array Slice Selector	chooses N elements of an array according to the slice definition (start:end:step)
[?@.price < 10] or [?@.author]	Filter Selector	chooses N elements of an <i>array</i> or M member values of an <i>object</i> based upon the results of the filter expression

The Name, Wildcard and Index *selectors* are fairly evident without additional explanation. The Array Slice *selector* is common to **python** and **javascript** programming languages.

3.4.3.1. Filter Selectors

The Filter *selector* is similar to the use of an XPath expression within a predicate in that it chooses values based upon the result of the expression. It is within the Filter *selector* that we can use the following two constructs of JSONPath:

- the current node identifier, '@'

- function extensions (currently five of them, including *length()*, *count()*, *search()*, *match()* and *value()*)

3.4.3.2. Types of expressions in Filter selectors

There are two types of expressions inside of Filter *selector*

Table 9. Types of Filter Selectors

Form	Type	Usage
<code>\$[?@.a=='b', ? @.b=='x']</code>	Comparison	choose a node based upon the boolean outcome of a binary comparison operation (one of '==', '!=', '<', '>', '<=', '>=')
<code>\$[?@.a @.b] or \$[?! @.a]</code>	Existence	choose a node based upon the existence (or not) of one or more nodes from the JSONPath expression

3.4.3.3. A few Notes on Filter Selector Expressions

In the following notes, some example JSONPath Queries are given that will act on this Query Argument modified from RFC9535 (Section 2.3.5.3).

```
{
  "obj"   : { "x" : "y" },
  "arr"   : [2, 3],
  "bool"  : false,
  "null"  : null
}
```

- A comparison of '==' between two expressions that both resolve to Nothing (no JSON value) yields **true**.
Example: `$.absent1 == $.absent2` resolves to **true**
- Array and object comparison is possible.
Example: `$.obj == $.obj` evaluates to **true**:
Example: `$.arr == $.arr` evaluate to **true**:
Example: `$.obj == $.arr` evaluates to **false**
- Compare the boolean literal value in the Query Argument to `true` in order to evaluate its value
Example: `$.bool == true` evaluates to **false**
- Compare a value to JSON literal `null`
Example: `$.null == null` evaluates to **true**

3.4.3.4. Segment and Selector Shorthands

There are some shorthands available in JSONPath that have identical semantics to their canonical equivalents.

Table 10. JSONPath Segment / Selector Shorthands

Shorthand	Canonical	Description
.name	['name']	Child segment with a Name selector
..name	..['name']	Descendant segment with a Name selector
.*	[*]	Child segment with a Wildcard selector
..*	..[*]	Descendant segment with a Wildcard selector

3.4.3.5. Function Extensions

The Filter selector expressions can use function extensions. These functions can be used as part of either comparison expressions or existence expressions. There are five functions listed in RFC9535 with the provision of adding additional functions. No user-defined functions can be created via JSONPath itself.

Table 11. JSONPath Filter Selector Function Extensions

Name	Return type	Description
length()	unsigned integer or Nothing	returns the number of unicode chars in a String argument, the number of elements in an Array argument, the number of members in an Object argument, Nothing for any other argument.
count()	unsigned integer	returns the count of the number of nodes in the nodelist returned by its query expression (mostly the same as the XPath equivalent)
match()	LogicalType	returns whether or not the string argument matches a given regular expression (mostly the same as it's XPath equivalent)
search()	LogicalType	returns whether or not the string argument contains a given substring (mostly the same as it's XPath equivalent)

Name	Return type	Description
value()	a single JSON nodeNothing	returns the first JSON value in the nodelist, if any (in XPath, for expression \$x, would be (\$x)[1])

The RFC9535 `match()` function is based upon a subset of the "XML Schema Definition Language (XSD)" specification for the regular expression syntax and semantics. [23] [24]

3.5. JSONPath Compliance Testing Suite

A work separate from RFC9535, the "JSONPath Compliance Test Suite" [21], provides a series of input documents, JSONPath queries and validated results. The compliance tests are compiled into a single JSON document, with 578 different tests as of this writing. Some examples are:

```
{
  "name": "basic, name shorthand, underscore",
  "selector": "$._",
  "document": {
    "_": "A",
    "_foo": "B"
  },
  "result": [
    "A"
  ]
},
{
  "name": "basic, wildcard shorthand, object data",
  "selector": "$.*",
  "document": {
    "a": "A",
    "b": "B"
  },
  "results": [
    [
      "A",
      "B"
    ],
    [
      "B",
      "A"
    ]
  ]
},
}
```

```
{
  "name": "basic, name shorthand, symbol",
  "selector": "$.&",
  "invalid_selector": true
},
```

In these three tests, excerpts from the test suite, there are, for all three tests, the *Query Argument* (**document**) and the *Query* (**selector**).

In the first test, the *Output Values* are presented (**result**) as a Array of JSON values. The query result, from the standard, is indeed not an Array, but a *odelist* of JSON values, but as *odelist* cannot be expressed directly in JSON, the tests are encapsulating the *odelist* results, which are ordered, as an Array.

The second test has a non-deterministic order and hence a different key for the result values, (**results**). This indicates that there is a series of possible orderings of the results and all different combinations are valid. The top level of the Array in **results** contains the different possible *odelists* which, themselves, are presented in the second level of the array.

The use of the member name **result** or **results** is exclusive.

In the case of an error in the query, neither **result** nor **results** is provided; rather, an **invalid_selector** member name, which always contains the value `true` in the tests. The second test in the above list is an example. There is no error code indicating the reason for the error.

3.6. JSONPath *odelist* locations

RFC9535 defines a *node* in a resulting *odelist* is pair of

- a JSON value from the *Query Argument*
- a *location* which is a constrained JSONPath expression which indicates the position of the result within the *Query Argument*.

The *location* is expressed as a *Normalized Path*, which simply means that it is a JSONPath expression in canonical format consisting of *child segments* with square brackets), each *segment* with a single *Name* or *Index* selector.

RFC9535 itself gives illustrations of the examples results using locations. Here is an excerpt from section 2.3.1.4

2.3.1.3. Examples

JSON:

```
{
  "o": {"j j": {"k.k": 3}},
  "'': {"@": 2}
}
```

Queries:

The examples in Table 5 show the name selector in use by child segments.

Query	Result	Result Paths	Comment
<code>\$.o['j j']</code>	<code>{ "k.k": 3 }</code>	<code>\$['o']['j j']</code>	Named value in a nested object

The compliance test suite itself does not provide the result locations.

It's not clear that locations are a strict requirement of output in a conforming implementation. There is, however, an example of an implementation, the "Jsonpath Online Evaluator" [22], that does provide the output paths that correspond to the locations. That implementation predates by at least two years the publication of RFC9535.

4. JSONPath Expressions Compared to XPath 3.1 Expressions

4.1. Similar mechanisms in XPath3.1 and JSONPath for selecting JSON Values

We can compare some of the different expressions available for JSON Values.

Table 12. XPath / JSONPath Expressions on JSON Values

JSONPath	XPath 3.1	Comments
<code>\$</code>	<code>.</code> or <code>current()</code> or <code>\$root</code>	Root JSON Value, either as current context item or in a variable. This can be any value in XPath: here suggesting the current context item or a variable.
<code>@</code>	<code>.</code>	The JSONPath identifier corresponds to the current context item within an XPath predicate
<code>.key</code> or <code>['key']</code>	<code>?.key</code> or <code>.('key')</code> or <code>map:get (\$m, 'key')</code>	JSONPath Child Segment/ Name Selector Shortcut

JSONPath	XPath 3.1	Comments
[5]	<code>?.?5</code> or <code>.(5)</code> or <code>array:get(\$a, 5)</code>	JSONPath Child Operator with Arrays
[-1]	<code>array:get(\$a, array:size(\$a))</code>	Last element in the array
..[3]	<i>In XPath 3.1, this will require some functions, especially passing a higher-order function to a recursive descent function</i>	Third element of all descendants that are arrays
<code>\$store.book[-1]</code>	<code>?.?store?book?*[last()]</code>	In XPath, <code>last()</code> for a sequence is equivalent to <code>[-1]</code> for a JSON Array
<code>\$.store.book[?@.isbn]</code>	<code>?.?store?book?*[.?.isbn]</code>	All books with an ISBN number in the map
<code>\$.store.book[0, 2]</code>	<code>?.?store?book?*[position() = (1, 3)]</code>	Books 1 and 3

4.2. Some things that can be done in XPath 3.1 but not in JSONPath

JSONPath is intentionally limited to extractions of values from the *Query Argument*. Because of this, at least, there are many things that can be done in XPath 3.1 that are not available in JSONPath.

- cannot obtain the value of an arbitrary expression in the result .. only values from with the *Query Argument* are available
- cannot extract a list of member names (keys) from JSON Objects .. only member values or entire objects can be extracted (Section 1.1.1)
- cannot construct an arbitrary JSON object or array to use as comparison node
- JSONPath defines no variables that can be used from within its expressions
- XPath has an extensive library for date manipulation, string manipulation, URI resolution, etc
- XPath allows for user-defined functions, for-declensions, arbitrary map/array instantiation, higher-order function evaluation, etc

5. Conclusion

JSONPath and XPath have some interlinked history, common origins and even common participants. XPath is a much older standard, by 25 years for XPath1.0 compared to the RFC9535. JSONPath has benefited from both XPath1.0 and XSD regular expression definitions (for the `match()` function).

XPath can be used as a general purpose expression language. It excels at constructing and transforming structured data, including the maps and arrays that make up JSON. In most probability, the advent of RFC9535 will not present an opportunity for conversion to JSONPath for current users of XPath 3.1.

JSONPath has shown very successful adoption prior to RFC9535, with around 50 different implementations in over 20 host languages. It is used in a large array of systems, both open-source and commercial, as a means of expressing how actions are to be performed based upon data that is presented as JSON. Whereas the majority of these systems predate RFC9535 and are not strictly compliant, there may be a difficult choice to be made on the part of the implementors and vendors: maintain backwards compatibility or change to be compliant with RFC9535.

Bibliography

- [1] S. Gössner, G. Normington, and C Bormann. *JSONPath: Query Expressions for JSON*. 2024. Internet Engineering Task Force (IETF). <https://datatracker.ietf.org/doc/rfc9535/>.
- [2] Christoph Burgmer. *json-path-comparison*. <https://cburgmer.github.io/json-path-comparison>.
- [3] *IETF JSON Path Working Group (closed)*. <https://datatracker.ietf.org/wg/jsonpath/about/>.
- [4] *JSONPath - XPath for JSON*. Stefan Gössner. 2007-02-21. <https://goessner.net/articles/JsonPath>.
- [5] *REST-assured (github)*. Johan Haleby. <https://github.com/rest-assured/rest-assured/wiki/Usage#json-using-jsonpath>.
- [6] *SmartBear™ ReadAPI: a low-code API testing platform*. <https://support.smartbear.com/readyapi/docs/testing/jsonpath-reference.html>.
- [7] *Kubernetes™ Production Grade Container Orchestration*. <https://kubernetes.io/docs/reference/kubectl/jsonpath/>.
- [8] *Oracle™ Financial Services Analytical Applications Infrastructure Process Modelling Framework (PMF) Orchestration Guide*. https://docs.oracle.com/cd/E60058_01/PDF/8.0.6.x/8.0.6.2.0/OFSAAI_PMF_Orchestration_Guide_8.0.6.2.0.pdf.
- [9] *IBM™ Netezza Performance Server (NPS) - an advanced data warehouse and analytics platform*. <https://www.ibm.com/docs/en/netezza?topic=ddt-jsonpath>.
- [10] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. 1998-02-10. W3C. <https://www.w3.org/TR/1998/REC-xml-19980210>.

- [11] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0*. 1999-11-16. World Wide Web Consortium (W3C). <https://www.w3.org/TR/xpath-10/>.
- [12] Douglas Crockford. *Media Type for JSON*. 2006-07. Internet Engineering Task Force (IETF). <https://datatracker.ietf.org/doc/rfc4627/>.
- [13] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1*. 2006-09-26. World Wide Web Consortium (W3C). <https://www.w3.org/TR/xml11/>.
- [14] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. *XML Path Language (XPath) 2.0*. 2010-12-14. World Wide Web Consortium (W3C). <https://www.w3.org/TR/xpath20/>.
- [15] Anders Berglund, Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. *XQuery 1.0 and XPath 2.0 Data Model (XDM)*. 2010-12-14. World Wide Web Consortium (W3C). <https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/>.
- [16] *The JSON Data Interchange Format*. 2013-10. European Computer Manufacturers Association (ECMA) International. https://ecma-international.org/wp-content/uploads/ECMA-404_1st_edition_october_2013.pdf.
- [17] Douglas Crockford. Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. 2014-03. Internet Engineering Task Force (IETF). <https://www.rfc-editor.org/rfc/rfc7159.txt>.
- [18] Jonathan Robie, Michael Dyck, and Josh Spiegel. *XML Path Language (XPath) 3.1*. 2017-03-21. World Wide Web Consortium (W3C). <https://www.w3.org/TR/xpath-31/>.
- [19] Norman Walsh, John Snelson, and Andrew Coleman. *XQuery and XPath Data Model 3.1*. 2017-03-21. World Wide Web Consortium (W3C). <https://www.w3.org/TR/xpath-datamodel-31/>.
- [20] Douglas Crockford. Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. 2017-12. Internet Engineering Task Force (IETF). <https://www.rfc-editor.org/rfc/rfc8259.txt>.
- [21] *JSONPath Compliance Test Suite*. Glyn Normington. <https://github.com/jsonpath-standard/jsonpath-compliance-test-suite>.
- [22] *JSONPath Online Evaluator*. Kazuki Hamasaki. <https://github.com/ashphy/jsonpath-online-evaluator>. <https://jsonpath.com/>.

- [23] Tim Bray. *I-Regexp: An Interoperable Regular Expression Format*. 2023-10. Internet Engineering Task Force (IETF). <https://www.rfc-editor.org/rfc/rfc9485.txt>.
- [24] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes Second Edition*. 2004-10-28. World Wide Web Consortium (W3C). <https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/>.

Containerizing XML Build Tools to Facilitate CI/CD

C. Edward Porter
SAS Institute
<edward.porter@sas.com>

1. Introduction

Modern Git-based source management systems are popular stores for XML content. Each of the most-widely used Git remote repository hosts, e.g., GitHub, GitLab, and Bitbucket, offer continuous integration/continuous delivery (CI/CD) solutions to automate everything from validation and testing to builds and artifact delivery. CI/CD facilitates event-driven processes such that well-designed CI/CD pipelines can offer developers the freedom to push and merge changes quickly, seamlessly, and with confidence that the automated pipeline will build tested and deliverable artifacts.

One approach to managing the environments used in CI/CD pipelines is containerizing build tools. Creating Docker images of build tools ensures that the build environment and tools used in any instance are consistent and guaranteed to function as expected. The subject of this article is an exploration of the concepts and use of containerized build tools with regards to standard XML build tooling to facilitate XML authoring in a modern software-development "docs-as-code" manner.

2. Concepts

When prompted, any experienced developer could likely immediately offer a horror story of spending hours configuring their development environment only to find that when pushed to their remote repository their work fails to build in their build pipeline. Since time immemorial, incongruities and incompatibilities between development and build environments have slowed development and produced hard-to-diagnose errors. These headaches can be exacerbated in the new CI/CD paradigm, where pull requests will be rejected until builds finally compile cleanly, possibly grinding the pipeline to a halt. Just as developers experience these issues, so does the modern technical writer. Schemas and Schematron validation can provide guardrails, but the ways a writer may break a doc build are still myriad.

In the development world, Docker "Dev containers" are becoming increasingly common. Collaborative teams develop a shared Docker image containing all of the required development tooling and build tools preconfigured. Team

members can then use tools, such as Visual Studio Code, to develop within a container ????. Similarly, in the past several years, it has become increasingly popular in the CI/CD world to reduce reliance on a carefully configured build stack on build machines (e.g., a Jenkins server with several distinct Python installations, a local Antenna House instance, Saxon, Java, etc.) and instead to utilize Docker images containing the entire build stack, such that build steps for a given project are easily portable between different CI/CD servers and services.

An approach similar to the "dev" container/build container for software development can prove powerful in standardizing the XML stack to ease documentation development and builds. Rather than containerizing a development environment, one can containerize XSL stylesheets and the transformation controller, such that a writer execute transforms locally in the same exact manner as the documentation will eventually be transformed with in the CI/CD pipeline. In the following section, we explore a process utilizing containerized XML build tooling at SAS for processing and delivering content generated from embedded Doxygen-based software documentation.

3. In Practice

At SAS, containerized tooling permeates the technology stack across the organization. In the Documentation DevOps group, two examples stand out. The simpler use case facilitates publication of Doxygen-based documentation embedded in software source and will be the subject of this paper. On a larger scale, the forthcoming migration to GitHub for source management is driving the adoption of containerized build environments for the majority of the company's hundreds of XML docsets, utilizing lessons learned from our Doxygen work.

3.1. Doxygen to XIS pipeline

SAS software and programming documentation is primarily authored in XML or LaTeX. Over the last twenty years, SAS has developed a robust custom content management system (CMS), dubbed XIS (XML Information System), and publishing pipeline for delivering content to the web as HTML, ePub, and PDF as well as embedded documentation in our software. The system is mature and delivers, high-volume, high-quality content to end-users. While the majority of SAS documentation is either authored by technical writers in XML or by advanced analytics software developers in LaTeX, by tradition or acquisition, we have several outlier instances of documentation maintained by development groups outside of either ecosystem. Historically, if these edge cases intend to take advantage of the standard publishing tools and our documentation portal ???, we have developed lift-and-shift, one-way transformation tools to convert their documentation to SAS XIS DTDs, leaving developers to maintain and author new content using XIS in XML format.

Recently, however, the Documentation DevOps team started a unique project with developers in the Risk Solutions group who must continue to deliver their documentation bundled with their software to customers for end-user customization purposes by necessity while also delivering standardized content to SAS Help Center. For this project, we developed a containerized tool utilized both by developers locally to test and preview converted content and also by the CI/CD builds. In this section, we explore their tooling, and how we facilitate continued authorship in their existing manner by transforming their output to XML projects conforming to our tooling needs.

3.1.1. Doxygen

SAS Risk documentation is authored using Doxygen. Doxygen is a popular documentation tool for generating code documentation directly from source code. It combines LaTeX and LaTeX-like tagging with markdown support to allow developers to document source code in line and generate PDF and HTML output. It supports multiple source code formats, and in the case of this project, the Doxygen content documents macros and programs coded in the SAS programming language. Through the use of common LaTeX-like structural tags (e.g., `\brief`, `\details`, and `\param`, developers document the function, input/output parameters, and expected output of SAS macros.

3.1.2. Source Project Structure and Pipeline

SAS Risk software packages with Doxygen authored documentation are all structured similar with a similar pipeline for deliverables. Doxygen source documentation content resides in two places: 1) embedded within source `.sas` files (SAS program files) and in 2) stand-alone `.txt` files within a `../source/doc/` folder. The (1) embedded documentation contains documentation specific to the SAS program in which it is embedded, generally a brief and detailed description of the program or macro, input/output parameters, and any other pertinent details. The (2) standalone `.txt` files are more general documentation files from FAQs to how to interact with a given package, details regarding implementation, etc.

In addition to the source Doxygen content, these repositories also contain the complete compiled documentation, all the HTML files, images, and other accompanying metadata. This compilation is automated. Whenever a developer pushes a commit and merges it into the development branch, a series of continuous integration steps occur, including the "docgen" step that compiles the documentation, which is then available to be committed back into the repository.

3.1.3. Doxygen to XIS

Given the robust CI/CD pipeline that Risk developers have deployed, the Documentation DevOps team aimed to identify a way by which the existing pipeline

could be leveraged to somehow transform the Doxygen documentation into something deliverable to Help Center. We briefly explored modifying the Doxygen HTML transformation to produce content Help Center could render, but the format of its output differed drastically from Help Center expectations and the level of effort for modifying the Doxygen transform appeared high. Fortunately, it turns out that Doxygen can generate XML conforming to its own schema ???, so a plan soon coalesced around generating Doxygen XML and developing a transformation from Doxygen to XIS XML. With the content converted to XIS XML, the task of delivering to Help Center becomes trivial.

3.1.4. Designing the Doxygen to XIS Pipeline

Utilizing existing tooling and processes was a guiding principle when developing the Doxygen to XIS pipeline. Thus, we decided to represent these Doxygen projects as XIS projects, such that once the XIS XML is generated, it can simply be checked into its associated XIS project, committed and submitted, and the XIS pipeline will build and deliver rendered content to Help Center automatically. This design decision also offered us a phased approach to automating the conversion. As soon as the Doxygen to XIS tooling was containerized and available to the Risk developers, they could run that tooling locally, generate XIS XML, validate it, and manually commit it to their XIS projects. Then, as the transform and tooling matured, they could automate on their end, integrating it with the "doc-gen" step of their GitLab CI/CD pipeline to generate the Doxygen XML, convert to XIS, and finally deliver to the XIS project.

3.1.5. Containerizing and Automating the Doxygen to XIS Transform

There were several requirements for a Doxygen to XIS conversion tool. First, there should be little to no manual intervention required for the XIS XML output from the tool, such that it emerges well-formed and valid against the XIS DTD. It should be ready to be checked in to the XIS project associated with the SAS product without further manipulation. Any changes required to facilitate that should be achievable in the Doxygen source markup. Thus the second requirement: the conversion tool should be executable by developers themselves in their development environment to aid in revising the Doxygen source tagging prior to commit. In addition, it should be capable of being added to their existing CI/CD process with ease. Finally, as with all SAS tooling and infrastructure, security should be a top priority. For these reasons, we pursued containerization as a pathway to development and deployment.

3.1.5.1. The Doxygen Transform image

The Doxygen to XIS transformation is comprised of a Java-based Saxon Initializer to register an extension function that generates valid eid attribute values (an XIS

adaptation of UUIDs) and the doxygen.xml XSL stylesheet. A visualization of the Docker image appears below. The Java classes, minimal dependencies, and the XSL are all compiled into a "fat" jar file, such that the resulting transformation can be run via CLI with no classpath dependencies. Though SAS has a site license for Saxon, by using Home Edition we avoid the need to package the license file. We avoided requiring Professional Edition features by packaging the UUID generating class as an extension. Otherwise, the XSL transformation templates are fairly straightforward.

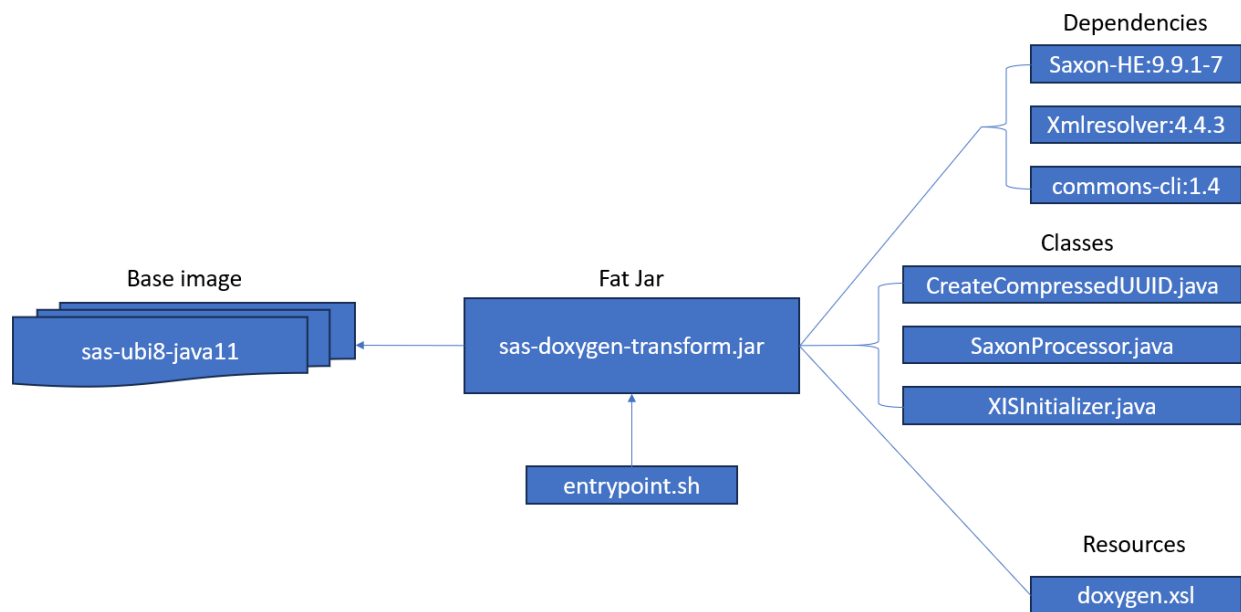


Figure 1. Doxygen Transform Docker Image Components

Security is a major concern at SAS; thus, developers are strongly encouraged to utilize universal base images (UBI) whenever possible. The Doxygen transform image uses the sas-ubi8-java11 base image. By utilizing that base image, we off-load the responsibility of making security updates to the Java runtime environment to the maintainers of the base image. Thus, we are only responsible for the few dependencies required by our transformation controller and our reliance on Java 11 itself.

Notably, this Docker image is not a service. It is not intended to be run continuously. Instead, it essentially functions as a containerized CLI executable. The last file in the image, `entrypoint.sh`, makes this possible. The `ENTRYPOINT` for the Docker image is set to the path for `entrypoint.sh` shell script. It provides the end user or the CI tooling a way to configure the container when it is instantiated with the volumes for input and output and to pass in parameters as necessary to execute a transformation. It also only accepts a single file at a time for transformation. The intention is that when multiple files need to be transformed, the CI code will manage that process. This use of an Docker image follows an implementation

model very similar to the one described in GitHub documentation for creating a "Docker container action" ???.

3.1.5.2. Doxygen Transform Image in Use

Generally, if developers have sufficiently simple documentation, they can simply write their documentation in Doxygen markup, commit and push their code, and expect that the transformation from Doxygen XML to XIS will go smoothly. However, for more troublesome content, by containerizing the Doxygen transformation, they can run the transforms locally to produce XIS XML locally, open it in XIS and Oxygen to validate and visualize the results in Author view, modify their source, and work iteratively to re-tag as necessary to produce satisfactory output. The Doxygen transformation image relies on the volume mount functionality provided by Docker to mount input and output directories and several parameters to configure the transformation. By mounting entire directories, the transformation can pick up any included content in the source directory, and when writing output, it can produce not only the converted XIS XML file, but also write an equations to separate, included files, that can be transformed from the source LaTeX markup to MathML XIS requires. A transformation command might resemble something like the following:

```
docker run -v [inputDir]:/input \
-v [outputDir]:/output \
-i --rm -e INPUT=[input-file-name.xml] \
-e OUTPUT=[output-file-name.xml] \
-e ALIAS=[projectAlias] \
registry.unx.sas.com/xis/doxygen-transform:0.0.17
```

3.1.5.3. Doxygen Transform Image in Continuous Integration

The source code repositories for the SAS projects containing Doxygen documentation are managed in GitLab. GitLab offers continuous integration (CI) functionality fairly similar to GitHub. CI functionality is YAML file based in which "pipelines" are divided into stages comprising of one or more "jobs" ???. These projects all utilize a robust collection of shared pipelines that contain stages like smoke tests, regression tests, linting, etc.

The end goal for integrating the Doxygen to XIS transformation into the Risk CI/CD workflow is visualized in the figure below. The process starts with a developer making changes to one or more source files containing Doxygen documentation. When they push to their development branch, numerous CI processes initiate. One of them is the Doxygen transformation. The changeset is iterated over, generating XIS XML files for each file containing Doxygen content. These outputs are copied to an associated XIS project CVS dropzone. An ad-hoc documentation build preview is kicked off, generating a PDF for review. If the devel-

oper is content with the new build, they then manually commit all changed XIS XML files to the CVS repository, and the XIS CI/CD build process will launch resulting in a new preproduction build for editorial review and eventual delivery on release day.

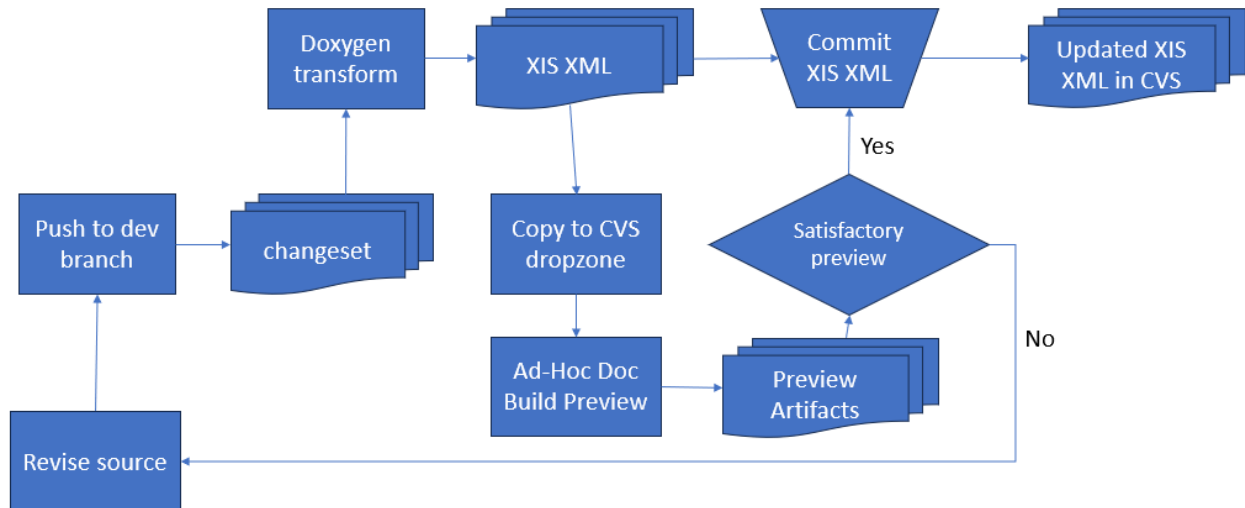


Figure 2. Continuous Integration Workflow

4. Challenges and Lessons Learned

4.1. Content Challenges

The challenges faced by this project were largely not tooling or pipeline related. The single most challenging aspect has been the content itself and the banal, perennial difficulty in converting from loosely structured content to a rigorously designed DTD. In all, these Doxygen-based documentation projects represent thousands of pages of content. While the larger, conceptual documentation is fairly uniformly structured section and topic-based content, the content embedded in SAS programs and macros varies widely. The Doxygen markdown to XML transform also relies on whitespace and carriage returns, so seemingly minor variations between formatting in different files produces different output XML. Further, the XIS DTD rigorously enforces a rule regarding a limited number of introductory elements preceding subordinate topics in nested topics, whereas Doxygen has no such constraint. Thus, without reauthoring and homogenizing the input, much of the XIS XML output proved to be invalid against the DTD. Significant time has been spent and continues to be spent triaging validation errors determining whether they necessitate input tagging changes or changes to the XSL.

4.2. Containerization, Secrets, and Security

Containerizing an XSL transformation simplifies many issues but can also introduce its own. Secret and key management is one such issue. For instance, while not the only XSL processor on the market, Saxon is surely the dominant one. While Saxon Home Edition (HE) is free to use, advanced Saxon features require a professional or enterprise (PE/EE) license. At SAS, we own a site license, but of course, one needs access to the license file to run the processor. For the purposes of this project, thus far, we have not required any advanced features and are relying instead on packaging HE with the Doxygen transform. If and when we identify a need to use PE/EE features, we will have to identify a way by which to supply the license file. This would not be problematic in the CI/CD pipeline, as one can store secrets in GitLab and GitHub, but short of including the license file within the image itself, providing this license file to individual users would be more challenging.

Managing the user context within a container is another security issue presented by containerization. It is good practice to avoid running processes as root when executing code with input from external sources within a Docker container. Thus, by SAS policy, one of the last steps in a dockerfile is to set the user to a non-root account. Since this tool requires that an input directory and output directory be configured as volume mounts within the Docker container, the `entrypoint.sh` and instructions for running the container were such that originally, the user running the Docker container was expected to mount folders within the container to a home folder for the user configured to execute the transformation. This worked well in the CI/CD pipeline and on Linux machines, but proved unworkable for users on Windows machines. While Windows machines could mount directories in the user's home directories as expected, there were permissions issues whenever the transform attempted to create or write to subdirectories within mounted folders, even within the home directory. These issues proved insurmountable, and in an effort to be cross platform compatible, we eventually set aside the requirement to execute the transform as a non-root user. When we can identify a fix or can abandon support for executing the transform from a Windows machine, we will revisit this decision.

4.3. Decoupling and Developing Generalized Resusable Images

Thus far, the XSL and XSL transformation controller are tightly coupled within our Docker image. The entire project is based on a Gradle build that builds the XSL transform controller with its accompanying dependencies and packages them along with the XSL and other resources in a "fat" jar for Docker to pick up when building the image. For this project, this model is reasonable, but if similar projects arise with different XSL requirements, it could prove useful to develop an internal SAS XSL transform controller image excluding XSL, but including one

or more transform extension classes to facilitate the needs of all of our XSL style-sheets. Then, this new XSL transform base image could be leveraged by docker-files for myriad transformations.

Ideally, perhaps, there would be an Docker images in the official Docker registry for Saxon CLI. There are a few unmaintained images, but official ones with uniform entry points and ways by which to provide extensions and mounts could prove very useful for others developing XSL/XQuery pipelines with CI/CD tooling. The Antenna House Docker image for its CLI would be a good model for this project ???.

5. Conclusion

The raison d'etre of XSL is transformation. Methods for accomplishing and automating these transformations have varied and proliferated throughout its existence, from shell scripts, to Maven, Gradle, XProc and more. With the push for modular, reconfigurable tooling, at SAS, we are exploring yet another paradigm, containerized build tooling in CI/CD pipelines. By utilizing containerized build tooling, we produce platform-agnostic tools that can easily be run both in the development environment and also run with identical results in the CI/CD pipeline configured in the source management system of choice for that project.

The Doxygen to XIS CI/CD pipeline explored in this paper is a simple example of the power of containerized build tooling. As SAS migrates its documentation repositories from CVS to a modern git-based VCS in GitHub, we are applying the lessons learned in this Doxygen project as we look to exploit more containerized tooling, like the Antenna House Formatter Docker Image ??? and containerized Gradle build environments, to migrate from complex Jenkins builds with specially configured build environments to GitHub workflows run on generic on-prem build machines. Such is the power of containerizing build tooling for the XML stack. From a simple workflow with a few build steps to complex pipelines with containerized tooling and versioned declarative build scripts, XML transformations begin to resemble software builds, making them immediately more familiar to software developers, more maintainable, and more easily migrated to future tooling. In this article we merely scratched the surface of the flexibility and power of containerized tooling, but with luck we have provided a solid introduction to its utility.

Bibliography

- [1] *Antenna House Formatter Docker Image*. <https://www.antennahouse.com/formatter-docker-image>. Antenna House. 2024.

- [2] *Creating a Docker container action*. <https://docs.github.com/en/actions/creating-actions/creating-a-docker-container-action>. GitHub. 2024.
- [3] *compound.xsd*. <https://github.com/doxygen/doxygen/blob/master/templates/xml/compound.xsd>. Doxygen. 2024.
- [4] *doxygen*. <https://doxygen.nl/>. doxygen. 2024.
- [5] *"Developing inside a container"*. <https://code.visualstudio.com/docs/devcontainers/containers>. Microsoft Corporation. 2024.
- [6] *"Getting started with GitLab CI/CD"*. <https://docs.gitlab.com/ee/ci/>. GitLab. 2024.
- [7] *"SAS Help Center"*. <https://documentation.sas.com>. SAS Institute. 2024.

QTI and InDesign

Transformations to produce student worksheets

Mark Dunn

Oxford University Press

<mark.dunn@oup.com>

Abstract

This paper describes a project to automate a process for generating student worksheets in print and digital formats from a single source. The source XML format is QTI 2.1. The requirement was to produce from this source an Adobe InDesign document from which we can export a print PDF suitable for publication. We describe some basic concepts of InDesign and QTI, and outline the proposed new process, the XSLT transformation design, and some of the particular challenges that were encountered.

Keywords: QTI, IDML, InDesign, XSLT, XProc

1. Introduction

Oxford University Press produces assessments for students learning English. These consist of short activities where the student must look at an image, listen to an audio file, or read a text, and then choose the correct response from multiple options, fill in a gap to complete a sentence, etc.

Currently each assessment is created twice, once for print and once for digital.

The initial version of the content is made in Microsoft Word and imported into InDesign.

There is a lengthy process of review and correction, which also takes place in InDesign. Designers and editors work together to produce a layout ready to print.

The same content is recreated in a digital platform to produce an online, fully interactive version of the assessment.

If further corrections are needed, they are also done twice, once in InDesign and again on the digital platform.

This project proposes a novel way of approaching a single-source workflow for print and interactive English Language Teaching materials. Few large publishers have solved this challenge fully for complex educational content.

The use of QTI as the XML model for assessment content means that producing the digital output is straightforward. Several learning platforms natively use QTI for their assessments. This paper focuses on producing the print output, which is much more challenging.

2. Project requirements

The requirements stem from the desire to reduce duplication of effort by creating and updating content in one place for both print and digital formats.

We want to automate the import of XML (in the standard QTI format for assessment content) into InDesign and the application of styles to the text.

The “Import XML” feature of InDesign partly automates this, e.g. XML elements can be mapped to InDesign styles in the template.

But this is not sufficient for the complex assessment content we are dealing with. For example, XML attributes cannot be mapped to styles, so features of QTI that are distinguished by `@class` attribute values must be styled manually after import.

A more fully automated process still has to fit around work done by hand in InDesign to finesse the layout. The print pages are densely populated and designed to appeal to students. The layout is far from the standard layout you might find across series of academic titles or journals.

An InDesign document generated by automated transformation from QTI may end up with numerous styling overrides assigned to its components by a designer, e.g. to resize or move an image, to arrange part of the text in multiple columns instead of a single column, etc. This is all part of the process to make the content attractive to the student and to fit neatly on to a page.

If the text content also needs to be updated or corrected, we want to make this in the source QTI, so that it only has to be done once for both print and digital outputs.

Third parties such as government ministries that control curricula have the ability to request changes late in the production process. By the time the QTI XML has been updated, the InDesign layout may have had styling overrides applied to it. When the XML is updated and re-imported, we want to preserve any styling overrides that have been made, and not reset the content to a default layout.

3. InDesign basics

Adobe InDesign is an application used for typesetting books, magazines, leaflets, worksheets, etc.

A common workflow is to import content from Word. Styling in Word can be mapped automatically to styling in InDesign.

Once imported, text and images etc are laid out in “Spreads”. Further styling can be applied.

Once the layout is approved, a PDF suitable for printing can be exported.

4. InDesign concepts

I will explain a few terms that will be used in this article.

4.1. IDML

IDML is a human-readable packaging format for InDesign.

An IDML file is a zip package of XML files describing design, layout, text content, etc.

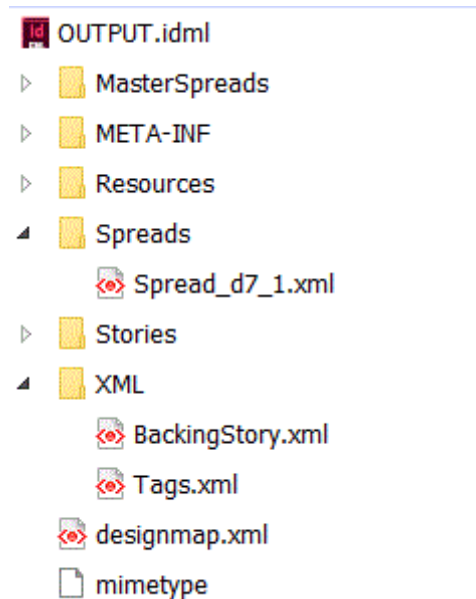


Figure 1. IDML package contents

4.2. Spread

A Spread consists of one or two Pages, and objects of various kinds (Text Frame, Image, etc) with properties defining their position, size, orientation etc on the Page.

A Page may inherit properties from a Master Spread

Each Spread and Master Spread is an XML file in the IDML document.

The Spreads are what appear in the print publication.

The Master Spread does not appear, but if a Page on a Spread is linked to a Master Spread, the content of the Master Spread will appear on that Page in the print publication. This feature enables a page footer to be created once, as an object in the Master Spread, and appear on each Page that links to that Master Spread.

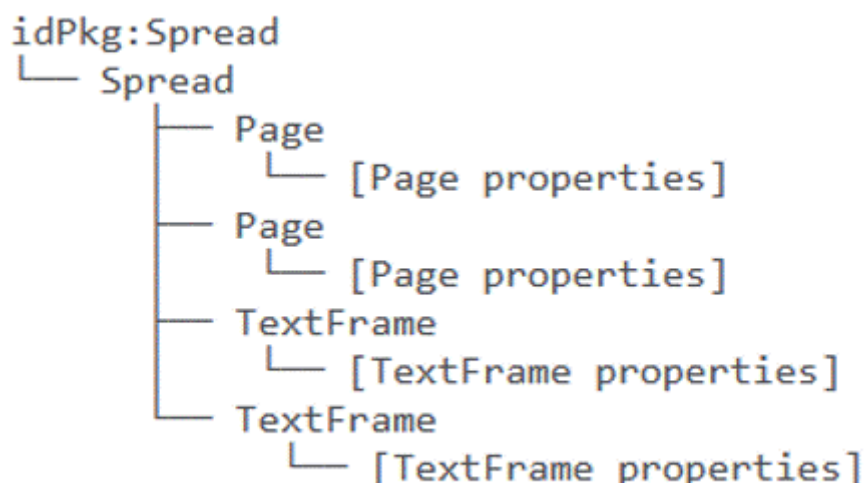


Figure 2. IDML Spread outline XML structure

4.3. Text Frame

A Text Frame is an object on a Spread.

It displays text that is physically held in a linked Parent Story.

Text Frames can be chained together if the Parent Story is too long for one Text Frame. The Story then overflows into the next linked Text Frame.

For example, the text of an entire book can be held in a single Story that is displayed across Text Frames chained together across the Pages of the book.

Linked Text Frames can also be used to produce effects such as two-column layout. (The columns are linked Text Frames.)

We also make use of “anchored” Text Frames, which are objects within a Story that appear inline with the content of the Story.

4.4. Story

A Story contains a chunk of text that is to be displayed on a Page (or Pages).

Each Story is an XML file in the IDML document.

InDesign has an “Import XML” feature. Importing XML using this feature creates a “Backing Story”, which is a separate XML file in the IDML document.

A Backing Story is only visible in the structured view in InDesign, until elements are dragged into Spreads.

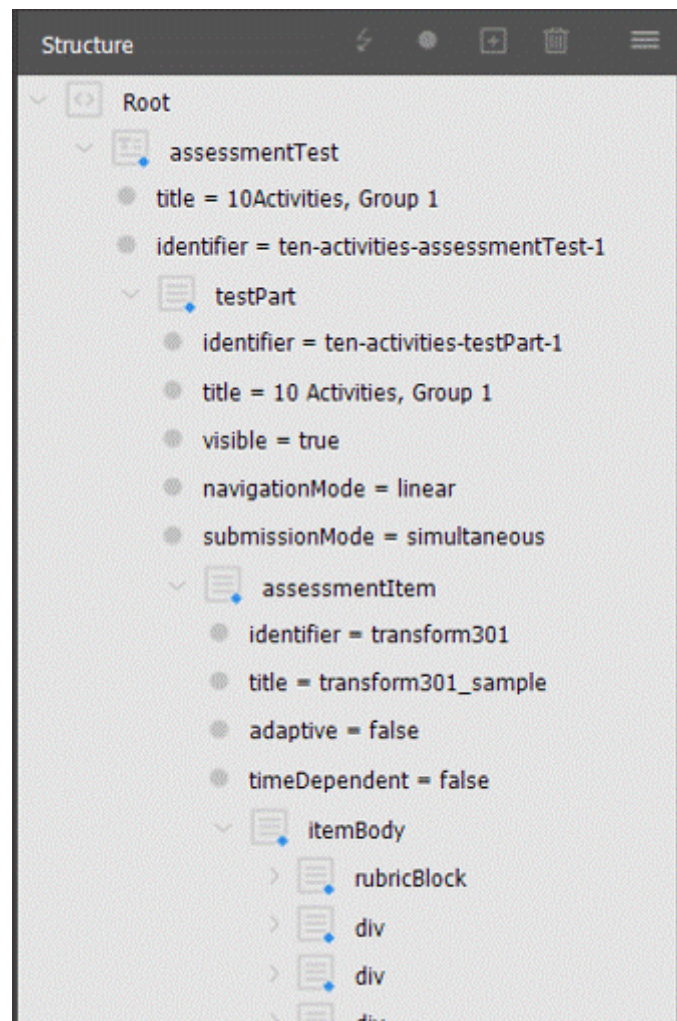


Figure 3. Structure panel in InDesign

Drag an element from the structured view to a Spread to create a Text Frame with the content of that node.

(This also snips the XML and creates a new Story as Parent Story for the Text Frame.)

Paragraph and Character styles can be applied to the text.



Figure 4. IDML Story fragment (outline view)

5. QTI basics

QTI¹ is an industry standard XML model for assessment content.

The model contains a mixture of presentational elements (XHTML) and functional elements.

Types of interaction include text entry, multiple choice, and ordering.

One or more interactions of the same type may be grouped together as an “assessment item” (or “activity”).

QTI also contains structures for holding the correct responses and for scoring the student's own responses against them and providing feedback.

5.1. QTI example: multiple choice interaction

This example shows:

- a variable to hold the student's response, with an indication of the correct response
- a variable to hold the student's score
- a variable to hold the feedback to be given to the student, depending on their response

¹ <https://www.imsglobal.org/activity/qtiapip>

- an instruction to the student
- a reference to an external file containing the text they must read
- a question
- three options for the answer
- feedback for each option
- a reference to an external file containing the logic for processing the response and assigning values to the variables

```
<?xml version="1.0" encoding="UTF-8"?>
<assessmentItem xmlns="http://www.imsglobal.org/xsd/imsqti_v2p1"
  xmlns:xi="http://www.w3.org/2001/XInclude"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.imsglobal.org/xsd/imsqti_v2p1 http://
www.imsglobal.org/xsd/imsqti_v2p1.xsd"
  identifier="mcq-example" title="Multiple choice example" adaptive="false"
timeDependent="false">
  <responseDeclaration identifier="RESPONSE" cardinality="single"
baseType="identifier">
    <correctResponse><value>Choice_C</value></correctResponse>
  </responseDeclaration>
  <outcomeDeclaration identifier="SCORE" cardinality="single" baseType="float">
    <defaultValue><value>0</value></defaultValue>
  </outcomeDeclaration>
  <outcomeDeclaration identifier="FEEDBACK" cardinality="single"
baseType="identifier"/>
  <itemBody>
    <rubricBlock view="candidate"><p>Read the text and choose the correct answer.</
p></rubricBlock>
    <div id="mcq-activity" class="activity-item">
      <xi:include href="rt-mcq-example.xml"/>
      <p>What does the writer like about Blackpool?</p>
      <choiceInteraction responseIdentifier="RESPONSE1" maxChoices="1">
        <simpleChoice identifier="Choice_A">The tower</simpleChoice>
        <simpleChoice identifier="Choice_B">The North Pier</simpleChoice>
        <simpleChoice identifier="Choice_C">The Pleasure Beach</simpleChoice>
      </choiceInteraction>
      <feedbackBlock outcomeIdentifier="FEEDBACK" identifier="Feedback_A">
        <p><span class="emphasis">The tower</span> is not correct because the writer
says
          "I am afraid of heights and didn't climb the tower".</p>
      </feedbackBlock>
      <feedbackBlock outcomeIdentifier="FEEDBACK" identifier="Feedback_B">
        <p><span class="emphasis">The North Pier</span> is not correct because
```

```
        it is closed for renovation.</p>
    </feedbackBlock>
    <feedbackBlock outcomeIdentifier="FEEDBACK" identifier="Feedback_C">
        <p><span class="emphasis">The Pleasure Beach</span> is correct because the
writer
        says that "I went on the roller coaster at the Pleasure Beach three
times".</p>
    </feedbackBlock>
</div>
</itemBody>
<responseProcessing template="match_correct.xml"/>
</assessmentItem>
```

Figure 5. QTI multiple choice activity

5.2. QTI example: response processing

This example shows the file for processing the student's response to the multiple-choice activity shown above.

It contains structures for assigning the student a score of 1 or 0 and specific feedback, depending on their response.

```
<?xml version="1.0" encoding="UTF-8"?>
<responseProcessing xmlns="http://www.imsglobal.org/xsd/imsqti_v2p1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.imsglobal.org/xsd/imsqti_v2p1 http://
www.imsglobal.org/xsd/qti/qtiv2p1/imsqti_v2p1.xsd">
    <responseCondition>
        <responseIf>
            <match>
                <variable identifier="RESPONSE"/>
                <correct identifier="RESPONSE"/>
            </match>
            <setOutcomeValue identifier="SCORE">
                <baseValue baseType="float">1</baseValue>
            </setOutcomeValue>
        </responseIf>
        <responseElse>
            <setOutcomeValue identifier="SCORE">
                <baseValue baseType="float">0</baseValue>
            </setOutcomeValue>
        </responseElse>
    </responseCondition>
</responseCondition>
    <responseIf>
```

```
<match>
  <variable identifier="RESPONSE"/>
  <baseValue baseType="identifier">Choice_A</baseValue>
</match>
<setOutcomeValue identifier="FEEDBACK">
  <baseValue baseType="float">Feedback_A</baseValue>
</setOutcomeValue>
</responseIf>
</responseCondition>
<responseCondition>
  <responseIf>
    <match>
      <variable identifier="RESPONSE"/>
      <baseValue baseType="identifier">Choice_B</baseValue>
    </match>
    <setOutcomeValue identifier="FEEDBACK">
      <baseValue baseType="float">Feedback_B</baseValue>
    </setOutcomeValue>
  </responseIf>
</responseCondition>
<responseCondition>
  <responseIf>
    <match>
      <variable identifier="RESPONSE"/>
      <baseValue baseType="identifier">Choice_C</baseValue>
    </match>
    <setOutcomeValue identifier="FEEDBACK">
      <baseValue baseType="float">Feedback_C</baseValue>
    </setOutcomeValue>
  </responseIf>
</responseCondition>
</responseProcessing>
```

Figure 6. QTI response processing template

5.3. QTI example: text entry interaction

This example shows the `<textEntryInteraction>` element that indicates where a student will provide their response.

In print this will be rendered as a write-on line. In a digital format it will be a text box.

```
<?xml version="1.0" encoding="UTF-8"?>
<assessmentItem xmlns="http://www.imsglobal.org/xsd/imsqti_v2p1"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

xsi:schemaLocation="http://www.imsglobal.org/xsd/imsqti_v2p1
http://www.imsglobal.org/xsd/qti/qtiv2p1/imsqti_v2p1.xsd"
identifier="gapfill-example" title="Text entry sample"
adaptive="false" timeDependent="false">

<responseDeclaration identifier="RESPONSE" cardinality="single" baseType="string">
  <correctResponse>
    <value>your</value>
  </correctResponse>
</responseDeclaration>

<outcomeDeclaration identifier="SCORE" cardinality="single" baseType="float">
  <defaultValue>
    <value>0</value>
  </defaultValue>
</outcomeDeclaration>

<itemBody>
  <rubricBlock view="candidate">
    <p>Complete the sentence with the correct form of the word in brackets.</p>
  </rubricBlock>
  <div id="gapfill301-activity-item-1" class="activity-item">
    <div id="gapfill301-artwork-1" class="artwork-supplementary">
      
    </div>
    <p>Where is <textEntryInteraction id="gapfill301-textEntryInteraction-1"
      responseIdentifier="RESPONSE"/> hat, Nigel?
      <span class="prompt">you</span></p>
    </div>
  </itemBody>

  <responseProcessing template="../rptemplates/match_correct_no_feedback.xml"/>
</assessmentItem>

```

Figure 7. QTI text entry activity

5.4. QTI package

A QTI “package” contains individual files for activities, tests (groupings of activities), media assets, response processing, and a manifest.

The manifest contains metadata about each resource in the package.

6. Outline of the new process

The starting point is an IDML “template” containing Master Spreads, styles, and placeholder Text Frames.

We then want to import the content of a QTI package using the following steps:

- unzip the IDML package
- transform the QTI XML into IDML Stories with appropriately styled content
- update other files in the IDML package
- zip up the new IDML package

I wrote an XProc pipeline (executed by MorganaXProc-IIIse²) to run this process.

7. XSLT transformation design

The source document for the main transformation is the QTI manifest.

The path to the folder containing the unzipped IDML template is passed as a parameter.

From the manifest, we identify the test and load it as a document. (The “test” is an XML file containing references to the individual activities, which are separate XML files.)

We will be creating new Spreads for the activities, based on the Master Spread. The Master Spread in the template contains Text Frames for the test heading and content, positioned and sized appropriately, and containing placeholder content. The new Spreads will contain clones of these Text Frames that point to new Stories containing the actual test header and content.

The stylesheet creates a global variable which provides details of the new Spreads, Text Frames, and Stories, mapped to the QTI nodes that provide the Story content for each Text Frame.

From the IDML folder, we identify the Backing Story.

We transform the Backing Story XML, importing and applying templates to QTI XML from the test and activities.

This process tries to go with the grain of the manual process by importing XML into the Backing Story and extracting nodes into new Stories and Text Frames.

The core part of the transformation is relatively straightforward. An element in the source QTI XML is transformed to a `<XMLElement>` element with a `@MarkupTag` attribute containing the name of the original element. An attribute in the source XML becomes a `<XMLAttribute>` element with `@Name` and `@Value` attributes containing the original attribute's name and value. A text node is retained, wrapped in a `<Content>` element.

It is necessary for us to include a representation of the source XML structure, as it is crucial to supporting the process of updating an InDesign document when source XML is updated, e.g. with corrections³

² <https://www.xml-project.com/morganaxproc-iiiise.html>

There are several more complex features of the transformation that are needed to produce the desired output. These are described next.

7.1. Application of styles

When matching a node that has a paragraph or character style, we apply the style by creating a `<ParagraphStyleRange>` or `<CharacterStyleRange>` element.

The mapping of QTI structures to template styles is hard-coded into the XSLT and accessed via functions `getParagraphStyle()` and `getCharacterStyle()`.

This XML fragment shows a fragment of a Story representing a write-on line populated with an example answer.

```
<XMLElement
  Self="d8e143_textEntryInteraction_default"
  MarkupTag="XMLTag/textEntryInteraction">
  <ParagraphStyleRangeAppliedParagraphStyle="ParagraphStyle/answer-example">
    <CharacterStyleRange
      AppliedCharacterStyle="CharacterStyle/$ID/[No character style]"
      <Content>your</Content>
      <Br/>
    </CharacterStyleRange>
  </ParagraphStyleRange>
  <XMLAttribute
    Self="d8e143_XMLAttribute_responseIdentifier"
    Name="responseIdentifier"
    Value="RESPONSE2"/>
</XMLElement>
```

Figure 8. IDML Story fragment for element with paragraph style

The IDML hierarchy of `<ParagraphStyleRange>` (for applying block-level styles) containing `<CharacterStyleRange>` (for inline styles) containing `<Content>` (for text nodes) is straightforward, but the rules for how `<XMLElement>` elements (representing the source XML structure) are interleaved are complex and undocumented. InDesign is highly sensitive to errors in this structure.

A second XSLT transformation on the generated Stories was necessary in order to impose these rules.

7.2. Creation of new Stories

We have already prepared a global variable identifying the QTI nodes that provide the content for Text Frames in the output Spreads.

³ #corrections-workflow

When matching one of these nodes in the QTI XML, we create a new Story whose identifier is already provided by the global variable.

By using `<xsl:result-document>` we move out of the Backing Story and create a new file for the Story. We continue the processing of the source QTI XML from that node but now within the new Story. This mirrors the manual process in InDesign of dragging a node from the structured view, which creates a new Story.

In some places within the new Story we use the same logic to create additional new Stories. For example, an activity may include a “write-on line” (gap for the student to enter their answer). To render this we create an “anchored” Text Frame within the Story. This TextFrame links to a new Parent Story that we create to hold the content. This may be empty if the line is blank, or contain the correct response if the markup indicates that it is a worked example.

0 Where is your hat, Nigel? (you)

Figure 9. Worked example of a text entry activity

7.3. Creation of new Spreads

We then make the new Spreads, containing:

- Page(s) linked to Master Spread
- Text Frames linked to the new Stories, cloned from corresponding placeholder Text Frames in the Master Spread

(Stories are not visible until we assign them to Text Frames in Spreads. The Story identifier associated with each Text Frame is provided in the global variable we created earlier.)

Because a Page in a new Spread is linked to the Master Spread (in order to inherit its properties from there), Text Frames on the Master Spread will be visible.

Some of these we want to keep, e.g. a generic page footer.

But we don't want to see the placeholder Text Frames in the final document, because we have cloned them into the new Spreads and linked them to Stories containing the test content. So we remove these placeholders from the Master Spread.

7.4. Update the Design Map

The Design Map is a manifest of the resources in the IDML package. Because the contents of the package have changed (new Spreads etc) we need to update this.

It was simplest to do this as a separate XSLT transform within the XProc pipeline.

8. Bear traps

IDML appears to have no official current public documentation, but I did find an old spec⁴ (in a public GitHub repository).

But this did not help with a peculiar problem I had.

The process creates a placeholder for an image if the image file is not yet available. This placeholder is a Text Frame whose content is the “art brief”, i.e. a description of what the image will contain.

A Text Frame gets content by linking it to a parent Story. The `@ParentStory` attribute of the `TextFrame` element matches the `@Self` attribute of the Story whose content we want to appear in the Text Frame.

The Text Frame in my sample was linked correctly to a Story containing the art brief, but the content was not visible in the output.

This fragment appears in the main Story for the activity. The source QTI is “snipped” at this point: the Story stops at the Text Frame, which points (with the `@ParentStory` attribute) to a new Story containing the content beyond this point.

```
<XMLElement Self="resource-fragment-group2-1_img_default"
  MarkupTag="XMLTag/img"
  XMLContent="resource-fragment-group2-1">
  <TextFrame Self="d11e141_img_TextFrame"
    ParentStory="d11e141"
    ContentType="TextType"
    ItemTransform="1 0 0 1 0 0"
    AppliedObjectStyle="ObjectStyle/WoL-V">
    <Properties>
      <!-- various Text Frame properties -->
    </Properties>
  </TextFrame>
</XMLElement>
```

Figure 10. Text Frame linked to Story “d11e141”

The new Story contains the art brief, pulled from the QTI manifest metadata about the image resource.

```
<idPkg:StoryDOMVersion="18.5"
  xmlns:idPkg="http://ns.adobe.com/AdobeInDesign/idml/1.0/packaging">
  <Story Self="d11e141">
```

⁴ <https://raw.githubusercontent.com/jorisros/IDMLlib/master/docs/idml-specification.pdf>

```

<XMLElement
  Self="resource-fragment-group2-19_img_default"
  MarkupTag="XMLTag/img"
  XMLContent="resource-fragment-group2-19">
  <ParagraphStyleRange AppliedParagraphStyle="ParagraphStyle/artwork-ref">
    <CharacterStyleRange AppliedCharacterStyle="CharacterStyle/$ID/[No character
style]"/>
  </ParagraphStyleRange>
  <XMLElement
    Self="d6e449_string_default"
    MarkupTag="XMLTag/string">
    <ParagraphStyleRange AppliedParagraphStyle="ParagraphStyle/artwork-ref">
      <CharacterStyleRange AppliedCharacterStyle="CharacterStyle/$ID/[No
character style]">
        <Content>Photo of Blackpool Tower</Content>
        <Br/>
      </CharacterStyleRange>
    </ParagraphStyleRange>
  </XMLElement>
  <!--
    various XMLAttribute elements
    representing attributes of the source <img> element
  -->
</XMLElement>
</Story>
</idPkg:Story>

```

Figure 11. Story “d11e141”

After lengthy experimentation I discovered that InDesign is sensitive to ordering of items in the Design Map. This file is part of the IDML package and contains references to each of the Stories in the package. If one Story contains a Text Frame that points to another, the Design Map must reference the Story that contains the Text Frame *before* the Story that the Text Frame points to.

My code was ordering the references in alphabetical order of file name, which happened to be wrong for the desired output.

9. Workflow for content corrections

A key requirement of the new process is that corrections and updates to the content are made in one place for both print and digital formats.

Currently this is an iterative process between Editors and Designers and takes place in InDesign itself.

InDesign does allow you to create and update XML tagging using the structured view. So it is possible to make changes in InDesign and export updated QTI XML from it, which we could then pass on to the digital platform.

But InDesign is not an XML editing tool, and does not contain a complete representation of the source QTI. Our judgment was that making content corrections in InDesign would not be a viable solution.

Our proposed workflow is to update the QTI XML and re-import it into InDesign.

But the layout may be updated in InDesign following the initial import, e.g. to position placeholder images.

The initial InDesign output uses Text Frames from the default Master Spread for its content layout, and hard-coded default sizes for the image placeholders.

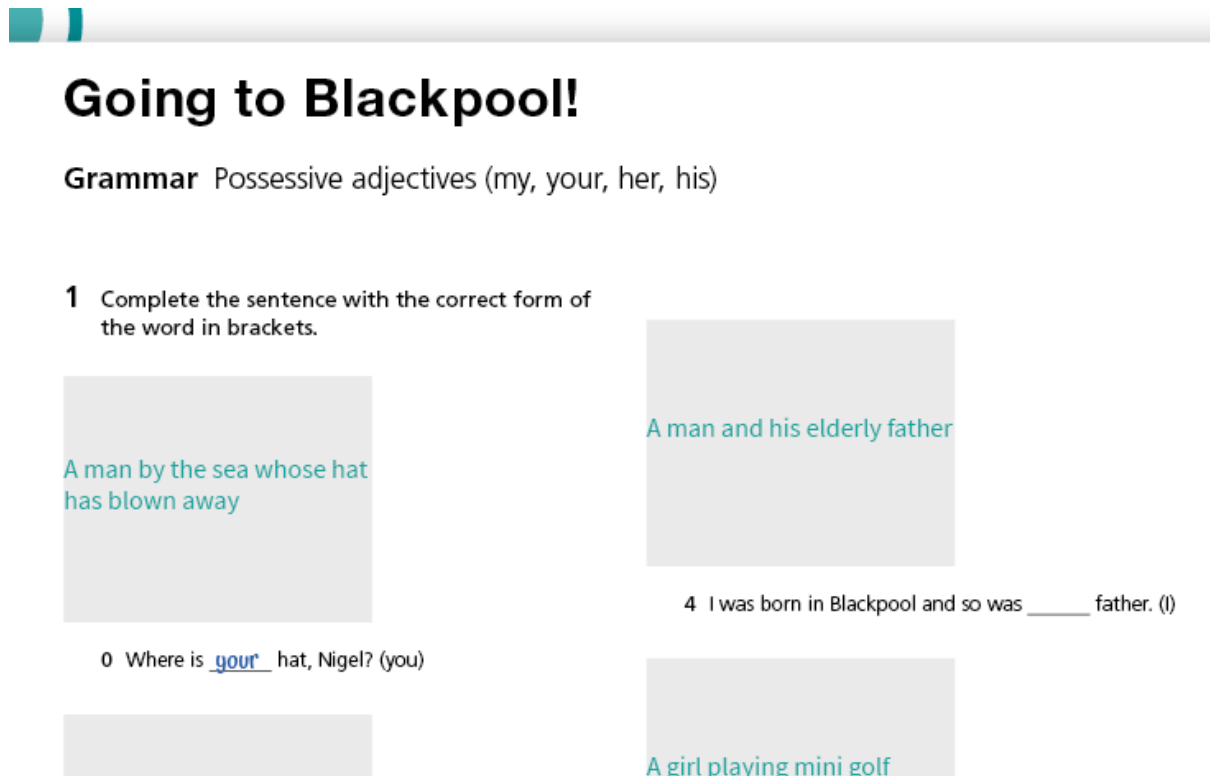


Figure 12. Initial InDesign output for a text entry activity

The designer takes this output and makes adjustments to produce the desired appearance for this activity:

- The instruction now fills the full width of the page instead of being restricted to the first column
- The image placeholders have been enlarged and aligned with the text of the activity items

Resizing the image placeholders has the effect of pushing item 3 into the second column and creating a symmetrical layout for the activity.

Going to Blackpool!

Grammar Possessive adjectives (my, your, her, his)

1 Complete the sentence with the correct form of the word in brackets.

A man by the sea whose hat has blown away

0 Where is your hat, Nigel? (you)

An empty theatre viewed from the stage

3 The theatre is old but _____ seats are comfortable. (it)

Figure 13. InDesign output after layout changes

When we re-import the QTI XML we don't want to reset the layout and force the designer to re-apply the layout changes. We need to preserve those changes.

To “refresh” the InDesign document with updated source XML, we use the updated InDesign output as the IDML source instead of the template. The XSLT identifies that Stories in the new IDML source are already associated with QTI nodes from the source XML. We can do this because of the `<XMLElement>` and `<XMLAttribute>` tagging within the Stories.

Instead of creating new Spreads, we copy the existing spreads, their Text Frames, and styling properties applied to them. The XSLT recreates the Stories from the source XML using the updated QTI XML.

Some styling overrides in the updated InDesign output are encoded in Story XML, e.g. `ParagraphStyleRange/@LeftIndent`. Although we are recreating the Story XML, the XSLT is able to copy these overrides from the source Story by identifying an association between the styling override and the node in the source QTI.

Getting the `<XMLElement>` elements interleaved correctly with the IDML styling elements was crucial to getting this process working.

The end result is that the text content of the IDML document is completely refreshed from the source QTI XML but the adjusted styles and layout from the initial IDML output are preserved.

Going to Skegness!

Grammar Possessive adjectives (my, your, her, his)

1 Complete the sentence with the correct form of the word in brackets.

A woman by the sea whose hat has blown away

0 Where is your hat Frida? (you)

An empty theatre viewed from the stage

3 The theatre is old but _____ seats are comfortable. (it)

Figure 14. InDesign output refreshed from updated QTI, with layout preserved

10. Importing images

When an image has been supplied, we can update the QTI XML to include a reference to the image file, including its dimensions.

```

```

Figure 15. QTI (XHTML) markup for image reference

The transformation replaces the placeholder text frame with a Rectangle of the same shape and position, with the Image placed within it.

The initial output was not promising.



Figure 16. First attempt at importing image

To understand and fix the problem I had to analyse the IDML markup for the Rectangle.

The properties of the `<Rectangle>` element (replacing the placeholder `<TextFrame>` element) illustrate the layout changes made by the designer to the original IDML output.

(Some IDML markup has been omitted for clarity.)

```
<TextFrame ItemTransform="1 0 0 1 0 0">
  <Properties>
    <PathGeometry>
      <GeometryPathType PathOpen="false">
        <PathPointArray>
          <PathPointType Anchor="0 -112"/>
          <PathPointType Anchor="0 0"/>
          <PathPointType Anchor="140 0"/>
          <PathPointType Anchor="140 -112"/>
        </PathPointArray>
      </GeometryPathType>
    </PathGeometry>
  </Properties>
</TextFrame>
```

Figure 17. IDML markup for original placeholder TextFrame

The four `<PathPointType>` elements give the X and Y coordinates of the corners of the TextFrame, relative to its anchor point, the top-left corner.

After the Text Frame has been moved and re-sized by the designer, the properties have changed and are inherited by the replacement Rectangle containing the imported Image.

```
<Rectangle ItemTransform="1 0 0 1 10.106443481445353 -49.211789047799016">
  <Properties>
    <PathGeometry>
      <GeometryPathType PathOpen="false">
        <PathPointArray>
          <PathPointType Anchor="-10.106443481445353 -112"/>
          <PathPointType Anchor="-10.106443481445353 49.21178904779901"/>
          <PathPointType Anchor="223.75182423518075 49.21178904779901"/>
          <PathPointType Anchor="223.75182423518075 -112"/>
        </PathPointArray>
      </GeometryPathType>
    </PathGeometry>
  </Properties>
  <Image ItemTransform="1 0 0 1 0 0">
    <Link LinkResourceURI="file:/path/to/nigel.jpg"/>
  </Image>
</Rectangle>
```

Figure 18. IDML markup for Rectangle containing Image

The four `<PathPointType>` elements give the X and Y coordinates of the corners of the Rectangle, relative to its original anchor point. The 5th and 6th components of the Rectangle's `@ItemTransform` attribute describe the X and Y movement of the Rectangle away from its original anchor point.

The image was placed at its full size and positioned with its top-left corner at the original anchor point of the frame.

To rectify this, we needed to update the `@ItemTransform` attribute of the image, to place it correctly within the containing Rectangle.

The requirements were to scale the image's width to the width of the Rectangle and to centre it vertically within the Rectangle. The XSLT makes a calculation based on the image's dimensions, the dimensions of the Rectangle, and also the height of the original placeholder Text Frame.

The 1st and 4th components of the `@ItemTransform` attribute are the scaling factor, the 5th and 6th are the X and Y movement to position the scaled image correctly within the Rectangle.

```
<Image ItemTransform="0.17989097516663546 0 0 0.17989097516663546
  -10.106443481445353 -117.1121551430023">
  <Link LinkResourceURI="file:/path/to/nigel.jpg"/>
</Image>
```

Figure 19. Updated IDML markup for Image

The scaled image is slightly taller than the containing Rectangle, resulting in the top and bottom edges being cropped.



Figure 20. Image positioned correctly

11. Next steps

Much progress has been made, but there is still a lot to do.

There are more types of QTI activity not yet handled by the transformation.

For the content corrections process, the transformation preserves a number of types of InDesign styling override, but there are more to do.

The process for making corrections to the QTI XML is also not yet confirmed. If we are not making content corrections in InDesign, we still want a process that allows us to see the changes in InDesign as quickly as possible.

We could create a transformation from the Word manuscript to QTI, which would allow corrections to be made in Word (a familiar tool for Editors) and transformed to QTI and then IDML in a single process.

As an alternative, I have created an authoring framework for QTI in Oxygen XML Editor⁵ as a proof of concept.

With the XProc pipeline set up as an External Tool in Oxygen, this would allow an Editor to update XML in a WYSIWYG environment and immediately see the results in InDesign.

Embedded Schematron rules (with QuickFix⁶) can allow us to ensure the Editor produces valid XML output.

Ultimately we want to produce books as well as worksheets. This is expected to involve blending QTI with the standard BITS data model for books.

For now this is a successful demonstration of an innovative method of producing worksheets for students. It automates many aspects of the process but leaves scope for a designer to customize the layout for individual assessments.

⁵ <https://www.oxygenxml.com/>

⁶ <http://schematron-quickfix.github.io/sqf/spec/SQFSpec.html>

Bibliography

- [1] *1EdTech Question & Test Interoperability (QTI) Specification*. <http://www.imsglobal.org/question/index.html>
- [2] *Adobe InDesign User Guide*. <https://helpx.adobe.com/uk/indesign/user-guide.html>
- [3] *IDML File Format Specification (Version 8.0)*. <https://raw.githubusercontent.com/jorisros/IDMLlib/master/docs/idml-specification.pdf>
- [4] *MorganaXProc-IIIse*. <https://www.xml-project.com/morganaxproc-iiiise.html>
- [5] *Schematron Quick Fixes Specification*. <http://schematron-quickfix.github.io/sqf/spec/SQFSpec.html>

XMQ/HTMQ - see XML and HTML in a new light

Fredrik Öhrström

Viklauverk AB

<fredrik.ohrstrom@viklauverk.com>

Abstract

XMQ is an alternative format for XML/HTML which is easier for humans to read and write when used for data-oriented use cases, such as configuration files, documents with layout, and generic data storage. For these use cases the standard format for XML/HTML is not always ideal. For example there is unnecessary verbosity in using markup tags both for opening and closing tags and the whitespace rules makes pretty printing hard and sometimes impossible.

XMQ solves these problems by using braces for the hierarchical structure and it simplifies whitespace handling by requiring all content whitespace to be quoted. Tags with simple content can be presented as key=value pairs where safe values need not be quoted at all. Quoted content has its incidental indentation removed which means that it can always be pretty printed at any indentation. This makes XMQ easier to read and write for humans and maps well to other key value based languages like JSON while maintaining full XML compatibility. XMQ can always be printed in a single line compact form without newlines, which is useful when writing to logfiles, and often requires less bytes than the corresponding compact XML/HTML.

Keywords: XML, HTML, XMQ, HTMQ, JSON

1. Introduction

When XML was first proposed as a subset of SGML in 1996, the focus was on simplicity of parsing (formal and concise, with a minimum of optional features) and that “terseness is of minimal importance”[1] The anti-terseness goal was probably a reaction against the configurable concrete syntax of SGML. This configurability and the ability to drop closing tags added a significant amount of complexity to the SGML parser. In fact you could redefine start and end tags (and several other language settings) to suit your environment and personal preferences.[2] Of course terseness was desirable in the 1980s and 1970s since we edited SGML files using text editors with a screen size of 80x24 or 80x25 characters[9] and terseness was even more important for the precursor GML from 1969.

When markup was first introduced, as the name implies, it was used to perform “semantic markup of textual documents in a manner that permits the separation of the underlying content from the formatting instructions”.[6] The distinct opening and closing tag was therefore well suited to this task. Likewise, the whitespace that was already present in the text before the tags were added, had their own interpretation and the added tags had to live with whatever whitespace convention the underlying text used.

However, over the years XML/HTML has been used more and more as data-oriented languages, ie the tags are not just markup, they are the content and they are often in the majority. With this in mind, perhaps it is time to revisit the anti-terseness goal.

The distinction between data-oriented XML and document-oriented XML has been discussed before and DuCharme writes that they are not rigid categories.[3] However many programmers are used to the JSON format for data storage and prefer to re-implement the XSD/XSLT functionality as JSON schemas [7] rather than using XML/XSD. The primary driver for JSON adaption is of course the integration with Javascript, it is after all its own object notation. However JSON is often used even in other data storage situations (like configuration files) where Javascript is not strictly involved. You can see this in tools like quicktype which is used to create JSON mappings for all sorts of programming languages.[8]

Kay lists[5] three reasons for the popularity of JSON versus XML: 1) simpler grammar 2) better match to data models 3) better support by web applications. The grammar is indeed simpler, but for 2, in my opinion it only matches a single data model (Javascript) which lacks object types. We have already mentioned 3 (integration with Javascript) as the primary driver for JSON usage.

To this list, I would like to add and emphasize that the popularity of JSON comes from it being data-oriented (for data storage), not document-oriented (markup). The difference is obvious when you compare JSON and XML in your text editor. When used for data storage, the closing tags in XML are verbose and visually obscures the content. Kay writes that XML is verbose[4] but not more so than JSON.[5] If we compare XML with JSON by counting the bytes, then "car_speed":123 is shorter than <car_speed>123</car_speed>. Interestingly XML itself uses key="value" pairs for the attributes inside the elements. This is probably because the attributes were seen as data storage, not markup. Perhaps Kay is comparing JSON to XML attribute data storage, eg <car speed="123"/> which is similar in size and not more verbose.

Why do we not use only element attributes for data storage? We could, and sometimes we do. For example SVG files store most of the data in attributes. But that is not how the use of XML has evolved over time instead often a key=value is stored as <key>value</key> not as <data key="value"/>. In pom.xml files we have <groupId>junit</groupId>, in rss we have <link>http://www.example.com/blog/post/1</link>, in WIPO standard ST.26 we have <INSDQualifier_value>other

RNA</INSDQualifier> and in this very docbook we have <abbrev>1996XML</abbrev>.

But the big difference between JSON and XML when used for data storage is how whitespace is handled. JSON is much simpler compared to XML in regards to whitespace since all strings must be quoted. The whitespace complexity in XML again comes from its markup origins. The markup tags were inserted into existing text already with whitespace and from this we have all the complicated whitespace rules that might even prevent us from pretty printing.

FtanML [4] was another alternative format for XML which claimed to solve the whitespace problem "There's no ambiguity about where whitespace is and is not significant. It's only significant if it appears in a string, or in rich text." FtanML also gave an interesting solution for permitting pretty printing to any indentation by starting each indented line with a backslash. However FtanML did not support namespaces whereas XMQ does. FtanML reused the < > symbols which in my opinion made it difficult for a reader to quickly see if it was FtanML or XML. If FtanML had fully implemented their tool, including automatic pretty printing (or reformatting as they called it) then they would have achieved almost the same goals as XMQ.

The Q in XMQ does not have a meaning, it is merely an available triplet of characters similar to XML that has not been reserved for file formats before.

2. An alternative format

Let us now declare the goals for an alternative data-oriented XML/HTML format:

- Store as key=value pairs when possible. This reduces verbosity and implies avoiding closing tags.
- Content whitespace must always be quoted and visible. This enables simple and safe pretty printing.

There are of course many possible formats that fulfill these goals. I will now demonstrate the XMQ format using examples and explanations. We begin with a simple data storage for a shipping order:

```
shiporder {
  id   = 889923
  type = container
  shipto(sailing = '')
  {
    address = 'The Vasa Museum
               Galärvarvsvägen 14
               115 21 Stockholm
               Sweden'
    // Remember to verify coord.
    coord = '''59°19'41.0"N 18°05'29.0"E'''
  }
  rules
}
```

```
<shiporder>
  <id>889923</id>
  <type>container</type>
  <shipto sailing="">
    <address>The Vasa Museum
Galärvarvsvägen 14
115 21 Stockholm
Sweden</address>
    <!-- Remember to verify coord. -->
    <coord>59°19'41.0"N 18°05'29.0"E</coord>
  </shipto>
  <rules/>
</shiporder>
```

The hierarchical style with braces should look familiar (introduced for the B programming language in 1969[10]) but note:

- XMQ files are always UTF8 encoded and the only permitted separating whitespace between tokens are 32 (space), 10 (newline) and 13 (carriage return). This whitespace is never content whitespace. Like XML a single 10, or a single 13 or a pair 13/10 is treated as a single 10.
- Safe values after = can be stored as plain text (see 889923 container), no quoting needed!
- Unsafe values (after =) with newlines, whitespace or () { } ' " or leading = & // /* must be quoted. A quote starts with one or more single quotes (apostrophes) and ends with the same number of quotes.
- Two single quotes always mean the empty string (see sailing).
- In multiline quotes, the incidental indentation is removed (see address).
- Quotes containing single quotes (apostrophes) are quoted using n+1 single quotes (see coord). Note that two single quotes are reserved for the empty string. You will therefore see a single quote ' or three quotes ''' or more quotes.
- Single line comments use // and multi line comments use /* */.
- Comments containing comments are commented using n+1 slashes (eg /// /* ///).

This means that you can quote any block of text (except for invisible spaces near newlines) with enough single quotes and you can comment any block of text with enough slashes.

The automatic incidental indentation removal allows any text block to be pretty printed into any indentation. The quoted content that remains after incidental indentation has been removed is marked with red below:

```
shiporder {
  id   = 889923
  type = container
  shipto(sailing = '')
  {
    address = 'The Vasa Museum
               Galärvarsvägen 14
               115 21 Stockholm
               Sweden'
    // Remember to verify coord.
    coord = '''59°19'41.0"N 18°05'29.0"E'''
  }
  rules
}
```

The ideas for incidental indentation removal and n+1 quotes originated in the expert group for Java text blocks and was a collaborative effort led by Jim Laskey and Brian Goetz. The seed to the idea to separate desired whitespace from incidental(accidental) was planted by Kevin Bourrillion and the idea for n+1 quotes came from John Rose.[11] The Java text blocks eventually did not use the n+1

quotes and the exact rules for how the incidental indentation is removed also differs from the XMQ, but the concept is the same.

The incidental indentation removal solves a major whitespace problem in XML/HTML. The normal whitespace rule for XML and HTML will collapse the newline and the indentation beginning the next line. This can be used for pretty printing of XML/HTML, ie to indent the subsequent lines in the XML `<address>...</address>` to the same indentation as "The Vasa Museum". Alas this is not always possible, for example if the xml node has `xml:space="preserve"` set or if it is an HTML `<pre></pre>` node, or if the CSS for HTML node has been set to `white-space:pre`.

A pretty printer could be designed to take all of this into account, but it is hard. Especially since the pretty printing will depend on the external dependency to a CSS, that might not yet be available or is ever changing. I consider it a bug in XML/HTML that an IDE cannot safely pretty print the source code for a web page or an XSLT transform without risking to introduce/removing significant whitespace.

The XMQ approach is to make the content whitespace explicit. This means that you can see which whitespace will be part of the XML/HTML (the content whitespace) and which whitespace is merely separating whitespace between the XMQ tokens.

The content whitespace (that you now clearly see) will then be subject to the XML/HTML interpretation rules that are controlled by `xml:space="preserve"` or CSS `white-space:pre` etc. The content whitespace will then be interpreted as significant whitespace or insignificant. These interpretations can change, but you will still be able to pretty print the XMQ, since pretty printing only introduces separating whitespace, not content whitespace.

3. There is always a compact printing

If we remove as much separating whitespace as possible and change the single line comment to `/* */` and escape the newlines, then we can print any XMQ in a compact form on a single line. The ability to print a complete JSON document on a single line has been quite useful in log files (JSONL format), where each log entry can be added as a single appended line. With XMQ we can do the same (XMQL format). The linebreaks below are of course only artifacts of the limited width of the page.

```
shiporder{id=889923 type=container shipto(sailing=''){address=('The Vasa Museum'&#10;'Galärva  
rvsvägen 14'&#10;'115 21 Stockholm'&#10;'Sweden')}/*Remember to verify coord.*/coord='''59°1  
9'41.0"N 18°05'29.0"E'''}rules}
```

You can see character entities like `
` for newlines and compound values like `address=('...'
'...')` which normally is a multiline quote but where escaped newlines are intermingled with quotes to create the compact form. The entities

must be outside of the quotes, since if they were left inside the quotes they would no longer be entities, ie the quotes behave like CDATA blocks.

Even multiline comments can be printed as compact XMQ since `/*` means a newline. This is not possible with XML/HTML since there is no standardized way to escape newlines inside html/xml comments.

```
type {
  name = number
  /* After the type we define all the
    necessary --regex-- patterns to
    detect numbers. */
  pattern = [0-9]+
  pattern = [0-9]+.[0-9]
}
```

```
type{name=number /*After the type we define all the*/necessary --regex-- patterns to*/detect
t numbers.*/pattern=[0-9]+ pattern=[0-9]+.[0-9]}
```

Note that, for historical reasons, XML/HTML does not permit two or more consecutive dashes inside a comment. This is quite a showstopper if you just want to comment out some large part of your document. As you can see two dashes are permitted in XMQ-comments and the XMQ tool works around this problem when converting to XML/HTML by adding a very specific char (U+2410 Symbol for Data Link Escape) in such a way there are no two consecutive dashes in the XML. When loading from such xml, the DLE symbol is instead removed to restore the two dashes.

```
<?xml version="1.0" encoding="utf8"?>
<type><name>number</name><!--After the type we define all the
necessary -DLE-regex-DLE- patterns to
detect numbers.--><pattern>[0-9]+</pattern><pattern>[0-9]+.[0-9]</pattern></type>
```

4. Tool support

The XMQ language is already implemented in a tool that can be downloaded as prebuilt executables for different platforms, or tested on-line, from <https://libxmjq.org> or built from source <https://github.com/libxmjq/xmq> The tool can convert between XMQ/HTMQ/XML/HTML/JSON and also apply XSLT/XSLQ transforms along with some other useful functions similar to existing tools like `xmlstarlet`.^[12]

```
# Render XML/HTML/JSON as colored and pretty printed XMQ in your
terminal.
xmq pom.xml
cat index.html | xmq
xmq < data.json
```



```
# Use the built in pager to scroll up and down.
xm q pom.xml page
xm q pom.xml pa
cat rss.xml | xm q pa
xm q pa < index.html
xm q data.json pa

# View the pretty printed and colored XMQ in your default browser.
xm q pom.xml browse
xm q pom.xml br
xm q br < index.html

# View a json file as xm q in a pager or browser
xm q request.json pager
curl -s 'https://dummyjson.com/todos?limit=10' | xm q br

# Convert to xml/html/json
xm q data.xmq to-xml > data.xml
xm q page.htmq to-html > page.html
xm q data.xmq to-json > data.json

# View a large index.html but delete script and style tags.
xm q index.html delete //script delete //style pager

# The same but view using your default browser.
xm q index.html delete //script delete //style br

# Apply an xslq transform to some json to generate a html page.
xm q todos.json transform todos.xslq to-html > list.html

# Apply an xslq transform to generate plain text.
xm q todos.json transform todosframed.xslq to-text > list.txt

# Render content for html/tex typesetting.
xm q input.xmq render-tex > view_input.tex ; xelatex input_as_tex.tex
xm q input.xmq render-html > view_input.html
```

The tool is written in C and licensed under the MIT license. It uses libxml2 and libxslt since those libraries are available on many platforms already.

5. Rendering HTML as XMQ/HTMQ

We can now visualize HTMQ and HTML. Note that the HTML is manually pretty printed to simplify the comparison. HTMQ is just XMQ but with a structure that permits simple conversion to HTML. An element like `br` in XMQ will convert to

 in HTML (a self closing element) but convert to
 in XML. Also XMQ always permits attributes without values br(a) which translates into <br a=""/> for XML but into <br a> for HTML. We could say that XMQ is a slight superset of XML and HTML.

<pre> !DOCTYPE = html html { body { h1 = Welcome! 'Rest here weary traveller and s' a(href = https://a.b.c) = lee 'p until morning. Say ' &nabla; } } </pre>	<pre> <!DOCTYPE html> <html> <body> <h1>Welcome!</h1> Rest here weary traveller and sleep until morning. Say &nabla; </body> </html> </pre>
--	---

- Text that does not immediately follow an equal sign = is called a standalone quote (see 'Rest here ...' and 'p until ...') and must always be quoted. If you do not quote them, they will be interpreted as elements (see html body a h1).
- XMQ pretty printing is straightforward whereas the HTML line breaks are weird to prevent spaces inside the word sleep.
- XMQ entities like ∇ (∇) must be outside of the quotes.
- In the xmq, it is obvious that there is exactly a single space between Say and the nabla.

If you convert from HTMQ to HTML using the tool: `xmq welcome_traveller.htmq to-html > generated.html` Then you will see that the tool does not pretty print the HTML since it wants to preserve the XMQ content whitespace exactly as it was written. If you convert the generated HTML back using: `xmq --trim=none generated.html to-xmq > back.htmq` then it should recreate an identical HTMQ file.

By default when loading XML and HTML, then xmq tool will use a heuristic to remove some whitespace but keep some. This heuristic works most of the time, but might remove some leading/ending content whitespace. This heuristic can improve over time. XMQ can preserve all whitespace loaded from an XML/HTML file. If you load the manually pretty printed HTML file using this command: `xmq --trim=none welcome_traveller.html to-htmz --escape-non-7bit` then you will see the HTML indentation as quoted content whitespace which is pretty printed as XMQ! This is usually not useful, but it illustrates the amount of whitespace present in an HTML file that might or might not be significant.

```

!DOCTYPE = html
html {
  &#10;
  body {
    &#10;
    h1 = Welcome!
    &#10;
    a(href = https://a.b.c) = lee
    'p
      until morning.
      Say'
    &#10;
    &nabla;
    &#10;
  }
  &#10;
}

```

6. Rendering JSON as XMQ

We can use the XMQ tool to convert shiporder to JSON: `xmq shiporder.xmq to-json | jq` . You can see that the XML element name is folded as the key `"_":"shiporder"` and attributes are folded as children prefixed with underscores.

```

{
  "_": "shiporder",
  "id": 889923,
  "type": "container",
  "shipto": {
    "_sailing": "",
    "address": "The Vasa Museum\nGalärvarvsvägen 14\n115 21 Stockholm\nSweden",
    "///": "Remember to verify coord.Remember to verify coord.",
    "coord": "59°19'41.0\"N 18°05'29.0\"E"
  },
  "rules": {}
}

```

This is a different mapping between XML and JSON, compared to the standardised XSLT 3.0 mapping. This mapping instead uses the XMQ format which creates key=value pairs when the content of an element `<key>` consists only of text and entity nodes.

For normal JSON this mapping generates a much more readable XMQ than the XSLT 3.0 mapping. You can download JSON from an REST api and view as XMQ in your default browser: `curl -s 'https://dummyjson.com/todos?skip=4&limit=2' | xmq br` or you can view any downloaded json file: `xmq data.json`

An interesting observation is that the JSON objects lacks type information. This is visible as the underlines for element names. The XML element names

serves both as a key and as a type. When there is only a single element, we can say that the element name serves as a key. When there are multiple elements, then we can say that element name serves as the object type.

<pre>- { todos(A) { - { id = 5 todo = 'Solve a cube' completed = false userId = 31 } - { id = 6 todo = 'Bake pastries' completed = false userId = 39 } } total = 150 skip = 4 limit = 2 }</pre>	<pre>{ "todos": [{ "id": 5, "todo": "Solve a cube", "completed": false, "userId": 31 }, { "id": 6, "todo": "Bake pastries", "completed": false, "userId": 39 }], "total": 150, "skip": 4, "limit": 2 }</pre>
--	--

If you consider this, then JSON is not always a better match for data models (see reason 2 in the introduction), since the type is not available. In this particular example it would be obvious to replace the underlines with a `Todo` element name, which you can do in XMQ/XML but not in JSON.

There are more details to the XMQ/JSON conversion which are left out from this article. The conversion is designed so that we can work with and view JSON using XMQ, not work with and view XML/HTML as JSON. It is possible to convert any XML/HTML document to JSON but it will not be necessarily be convenient to work with.

7. XSLT transforms

The clear distinction between content whitespace and separating whitespace in XMQ is very useful when working with XSLT. In the example below, we can generate whitespace exact output and till pretty print the XSLQ transform.

```
xmql todos.json transform format.xslq to-text > output.txt
```

```

xsl:stylesheet(version = 1.0
               xml:space = preserve
               xmlns:xsl = http://www.w3.org/1999/XSL/Transform
               xmlns:fo = http://www.w3.org/1999/XSL/Format)
{
  xsl:output(method = text)
  xsl:template(match = _/todos)
  {
    '
    &#10;
    xsl:for-each(select = _)
    {
      '
      xsl:value-of(select = ''substring(concat(todo,'
      '
      &#10;
      '
      '
    }
    '
    &#10;
  }
  xsl:template(match = total)
  xsl:template(match = skip)
  xsl:template(match = limit)
}

```

The generated output will be the framed text:

Solve a cube
 Bake pastries

8. DTD and XSD

The inline DTD is the value assigned to the !DOCTYPE. There are no changes the the DTD language.

```

!DOCTYPE = 'goo [
  <!ENTITY copy "&#169;">
  <!ENTITY alfa "ALFA&copy;">
  <!ENTITY banana "<x>BaNaNa</x>">
]'
goo = ( 'XMQ<'
  &alfa;
  '>minions say:'
  &banana; )

```

```

<!DOCTYPE goo [<!ENTITY copy "&#169;">
  <!ENTITY alfa "ALFA&copy;">
  <!ENTITY banana "<x>BaNaNa</x>"> ]>
<goo>XMQ<&alfa;&gt;minions say:&banana;</goo>

```

Since XSD:s are normal XML they are rendered as XMQ in the same way as other XML.

9. Corner cases

In XMQ, a key=value is an equivalent syntactic sugar for: key{'value'} Note that for attributes, you can only write: key=value or key='value' since braces cannot be

used in attributes. In the example below all values are: 123 If the xmq tool detects that all children of an element are either text or entities, then it will present the element as a key value pair.

```
content {
  value = 123
  value = '123'
  value { '123' }

  element(value = 123
          value = '123')
}
```

XMQ is designed with the assumption that we rarely need significant leading/ending whitespace/quotes. But when this is needed, it can be expressed in different ways:

```
// Leading and ending two spaces
spaces = '  alfa  '
spaces = ( &#32;&#32; 'alfa' &#32;&#32; )
spaces { &#32;&#32; 'alfa' &#32;&#32; }

// Leading and ending single quotes
apos = '''
      'alfa'
      '''
apos = ( &#39; 'alfa' &#39; )
apos { &#39; 'alfa' &#39; }

// Leading and ending newlines
newlines = ( &#10; 'alfa' &#10; )
newlines { &#10; 'alfa' &#10; }
```

You can see the magenta colored parentheses () after the equal = sign. This is a compound value which can only consist of quotes and entities. Compound values are mandatory for the attribute values that need multiple quotes/entities since braces {} cannot be used inside an attribute value.

```
content(newlines = ( &#10; 'alfa' &#10; ))
```

10. Comparisons of file sizes

The tool has a built in function for summarizing statistics of documents. We can use this to compare the size of XML (with no indentation and as few newlines as possible) with compact XMQ. The first example is a WIPO ST.26 XML file recording RNA sequences. The compact XMQ uses 45% less bytes to store it. The meta size is the sum of the tag names, the attribute names and the attribute content. The text size is the sum of the text content. We can see as we predicted in the beginning of this article, that the ST.26 is used for data storage and not markup

since the meta data uses significantly more bytes than the text content. Note that if the opposite is true (text > meta), then this does not necessarily mean that the file contains markup, it can still be data storage with a lot of data.

You use the following command to generate compact XMQ from the original XML: `xmq st26.xml to-xmq --compact > st26.xmq`

You use this command to get the statistics: `xmq st26.xml statistics`

Table 1. Sizes

Source	XML size	XMQ size	Reduction %	Meta	Text
WIPO ST.26	85721	47337	45%	33878	8846
MAME software list	5353	4364	18%	2353	259
JLINE3 pom.xml	13700	8906	35%	3632	4080
Homepage www.dn.se	547156	491377	10%	256275	16450
Article below www.dn.se	376961	359244	5%	121688	8101

To get valid meta and text sizes for HTML pages you have to use this command: `xmq page.html delete //style delete //script statistics` This is because modern dynamic HTML pages store style sheets `<style>` and javascript code `<script>` in the page, and these are not content, but count as text, unless we delete them.

The pom.xml is clearly used for data storage, still the text size is larger than the meta size. This can of course happen when enough data is stored. On the other hand, the HTML pages have meta data that is 15 times larger than the actual text content. They ought to be considered as data storage for document layout, not markup of text.

The pom.xml uses `<key>value</key>` for data storage and the HTML mostly uses attributes `key="value"` for data storage. (The exceptions being the script and style elements that we deleted.) With XMQ they are both normalized into `key=value` pairs.

Note that it is interesting that the compact XMQ is often smaller than the compact XML, but this is not a goal of the XMQ language. The purpose of XMQ is help humans to read and write data-oriented content. This is why the ability to safely and easily pretty print is important. The compact form is useful because it

is the opposite of pretty printing and this symmetry helped improve the quality of the XMQ language during the design process.

11. Conclusion

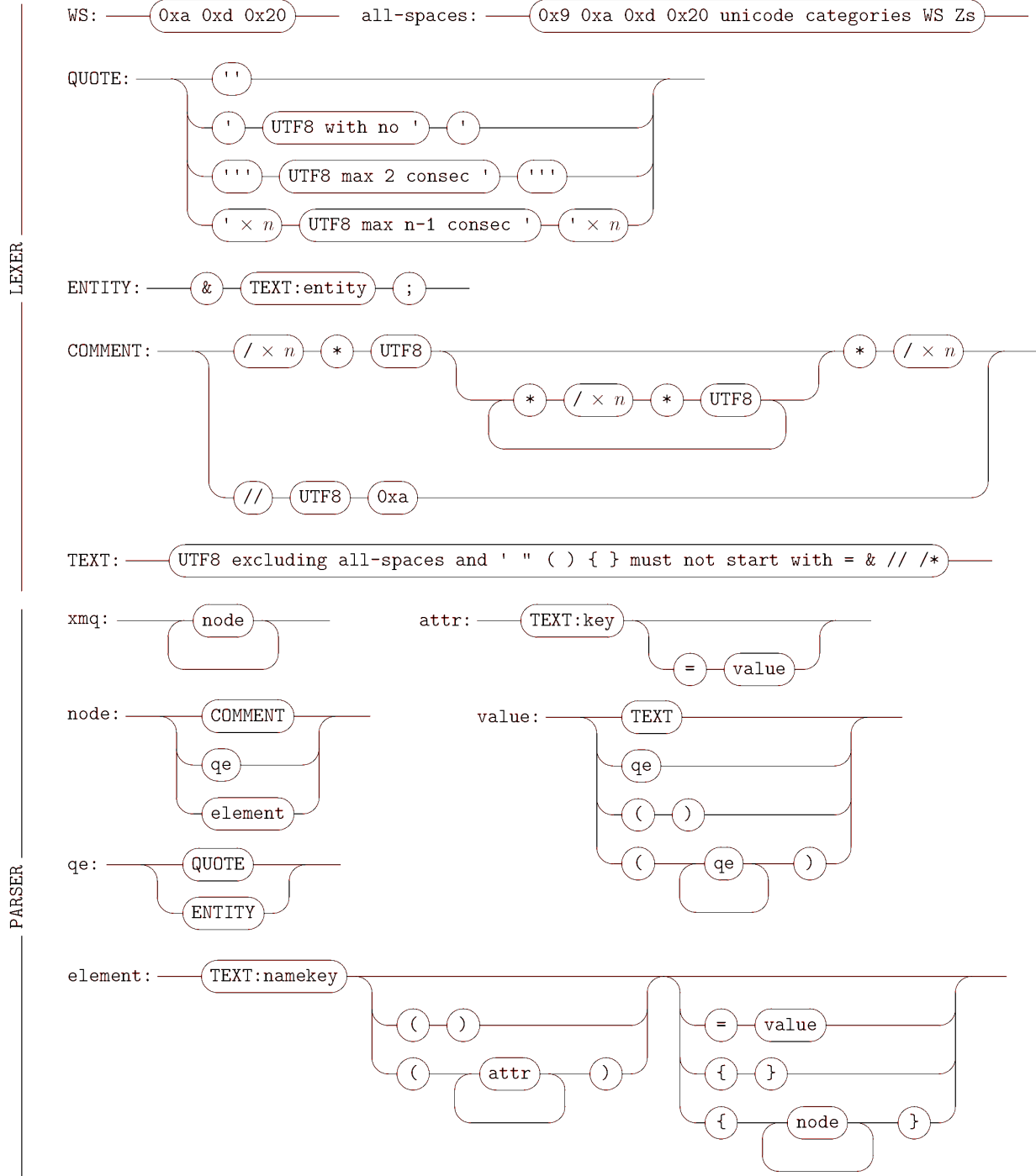
With XMQ the assumed dichotomy between markup languages (like XML) and key-value data store languages (like JSON) has been removed. We can now use XML for document-oriented situations and XMQ for data-oriented situations. They are interchangeable and all the years of effort going into XSLT/XSD and other tools can still be used with XMQ.

XMQ can always be pretty printed and it is easier to read and edit since we can see the content whitespace clearly.

The XMQ format has been in development since 2019 and many different syntaxes has been evaluated by the author. The current syntax is stable and no changes are planned.

A. Grammar & rules

Input must be valid UTF8 (0x9 | 0xa | 0xd | [20-d7ff] | [e000-ffff] | [10000-10ffff])
CRLF pairs (0xd 0xa) and standalone CR (0xd) are treated as LF (0xa) when parsing.



RULES	
	TEXT:namekey TEXT:key and TEXT:entity
r1.	Must start with a letter or underscore.
r2.	Cannot start with the letters xml (or XML, or Xml, etc).
r3.	Can contain letters, digits, hyphens, underscores and periods.
r4.	Can contain a single colon separating the TEXT into two parts, each following r1,r2,r3. Two permitted exceptions to TEXT:namekey rule r1 and r3.
r5.	A single !DOCTYPE before the first element and ?pi elements. If quoted content contains at least one newline then:
r6.	Leading (and ending) WS will be trimmed to newlines only, leaving one out.
r7.	All spaces before a newline are removed.
r8.	Incidental indentation (some spaces after a newline) is removed.
r9.	The indent to be removed is the minimum source code indentation for text within the block where empty lines are ignored.
r10.	The first line is prefixed with spaces if the following lines have lesser source code indentation. TEXT:namekey is (only for syntax highlighting) either a name or a key.
r11.	it is a key if '=' follows immediately (ie no attributes), otherwise it is a name. A quote with only spaces and a single newline is equivalent to the empty string. The element age=123 is shorthand for age{'123'}
EXAMPLES	
car { // An example structure. regnr = 'ABC 123' color = red img = /www/y.png tag = <car> }	<car> <!-- An example structure. --> <regnr>ABC 123</regnr> <color>red</color> /www/y.png <tag><car></tag> </car>
div(id = 32) { h1 = Welcome! 'Rest here weary traveller:' a(href = https://a.b.c) { img(url = /img/i.png) 'Click here!' } }	<div id="32"> <h1>Welcome!</h1> Rest here weary traveller: ↪Click here! </div>
	Html cannot be pretty printed with newlines here, whereas xmq can be pretty printed without introducing whitespace.
Explicit spaces: abc = ' ' abc {' ' }	
Spaces surrounding newline: abc = (' ' #10; ' ') abc {' ' #10; ' ' }	
Value with leading/ending quotes: x = (#39; 'quoted quote' #39;) or: ''' 'quoted quote' '''	
or: x { #39; 'quoted quote' #39; }	
A single newline: abc = #10; abc { #10; }	
or: abc = ' ' '	

Bibliography

- [1] Tim Bray, C.M. Sperberg-McQueen *W3C Working Draft*, 14 November 1996.
<https://www.w3.org/TR/WD-xml-961114.html>

- [2] Wayne Wohler *SGML Declarations*. <https://xml.coverpages.org/wlw11.html>
- [3] Bob DuCharme *Documents vs. Data, Schemas vs. Schemas*. <https://www.snee.com/xml/xml2004paper.html>
- [4] Michael Kay *The FtanML Markup Language*. <https://doi.org/10.4242/BalisageVol10.Kay01>
- [5] Michael Kay *Transforming JSON using XSLT*. <https://www.saxonica.com/papers/xmlprague-2016mhk.pdf>
- [6] Library of Congress USA *Standard Generalized Markup Language (SGML)*. ISO 8879:1986 <https://www.loc.gov/preservation/digital/formats/fdd/fdd000465.shtml>
- [7] Sourcemeta *A curated list of awesome JSON Schema resources, tutorials, tools, and more*. <https://github.com/sourcemeta/awesome-jsonschema>
- [8] Glide *Quicktype generates strongly-typed models and serializers from JSON, JSON Schema, TypeScript, and GraphQL queries, making it a breeze to work with JSON type-safely in many programming languages*. <https://app.quicktype.io/>
- [9] Ken Shirriff *IBM, sonic delay lines, and the history of the 80×24 display*. <http://www.righto.com/2019/11/ibm-sonic-delay-lines-and-history-of.html>
- [10] Ken Thompson *B (programming language)*. [https://en.wikipedia.org/wiki/B_\(programming_language\)](https://en.wikipedia.org/wiki/B_(programming_language))
- [11] Brian Goetz *Author's email conversation discussing the origins of Java Text Blocks*.
- [12] Mikhail Grushinskiy *XMLStarlet tool*. <https://xmlstar.sourceforge.net/>

<custom-element> DCE introduction

Sasha Firsov

<sasha@syngرافact.ai>

1. Introduction

Two years ago, the Web Components W3C Working Group asked for volunteers for Declarative Custom Element, the web components defined declaratively. I took the challenge and there is a result as the all-mighty XSLT fusion with HTML5 as the base for a next HTML standard proposal.

Web components have introduced the ability to extend the HTML markup with custom tags. Currently it is done by "custom element" browser API available only via JavaScript.

The need for defining the custom elements declaratively led to a series of Declarative Custom Element (DCE) implementations as JS libraries. These were presented to Web Components Community Group as POC for DCE as W3C proposal.

<custom-element> is the only DCE implementation which chose the pure declarative web application as its principle. It allows the creation of fully functional web applications without JS by using exclusively declarative markup. XSLT 1.0 syntax is the only fully functional native in the browser templating stack. It is used as the template markup by **<custom-element>** along with XPath and XML DOM as a data layer.

For now, it is implemented as open-source JS library published on GitHub, NPM, and CDN.

It is under consideration of being W3C WCCG proposal for DCE.

Implementation of **<custom-element>** as the browser plugin, mobile and IoT SDK, and secure browser is the planned evolution towards native platforms.

What **<custom-element>** can bring to the web as an applications platform?

2. Problem: Composed Web Applications challenges

After web 2.0 given by Dynamic HTML we have been transitioned to the next level of evolution. IMO the web 3.0 is characterized by syndication of many vendors applications and services on the host web application pages. It would be a surprise to find any web application which does not include at least few 3rd party apps. The site analytics, 3rd party authentication, likes, feedback just to name a few of most popular ones. The reason is pretty simple: the business specialization which allows to provide a better and cost-effective service.

As of now the most of 3rd party services are integrated with main web applications by JavaScript which does not have any safeguards around content and user interaction.

With the 3rd party apps integration as a general pattern, the web 3.0 apps become vulnerable and unreliable due to:

- Larger attack surface. The assemblage of applications contributes to a broadened attack surface, with the vulnerabilities of one becoming a risk to all.
- Security as a Common Denominator: security levels are only as strong as the least secure constituent, often defined by sub-application vulnerabilities.
- No Isolation Between Apps: current models lack effective isolation, with JavaScript allowing extensive, uncontrolled access across applications.

What if I need to provide the nice UX for login flow and still keep the other services like feedback, social media, analytics on the same page without compromising the user input?

3. Solution: <custom-element> application runtime

Without JavaScript there is no context sharing, hence the insulation between the 3rd party apps naturally comes from <custom-element> concept:

- Nothing is shared. Instead, strict ownership matches the DCE encapsulation. Context owner can control the child DCE, but none is leaked upwards.
- The business logic as transformation rule set resides in XSLT. Which results only in generated content.
- CSS styles are scoped to DCE and do not impact outer content.

4. Value proposition: Reimagining Development with DCE

4.1. Low-Code Principles

HTML-centric development facilitated by DCE eliminates the reliance on specialized JS developers, ushering in a new era of no-code principles driven by AI.

4.2. Reduced Development Cost

By shifting to HTML declarative methods, we foresee a transformation in development costs, championing efficiency and accessibility.

4.3. Integration Ease

Many companies lack the budget, skills, or knowledge to integrate 3rd-party apps securely. As the AI angle enables faster development overall and eliminates the need for coding, the DCE lowers the barrier of creating highly secured apps.

5. DCE overview

Extending the W3C standards as Web Components, DCE would provide a micro-frontend patterns for embedding 3rd party components into the main web application and insulation mechanism.

The key differentiator of `<custom-element>` among other DCE concepts is a declarative syntax as for application UI as for the business logic.

```
<custom-element>
  <form>
    <label>
      <input type="text" value="Type time update" slice="txt" slice-update="keyup"/>

      <span> Character count:
        <b> { string-length(//slice/txt) } </b>
      </span>
    </label>
  </form>
</custom-element>
```

The web components and **XSLT templating** co-exist on same stack without conflicting with each other. The data layer exposes as attributes as the payload passed to the instance (where tags actually used).

It also provides the data which can be changed in runtime by the named **slice** as result of data extraction in the moment of HTML events. DCE data is always belonging to own instance and is not exposed not to the container not to children unless passed though template which guarantee a safe embedding even 3rd party DCE components.

The **rendering cycle** is happen during initialization and every time when data is changed. The rendering is optimized to update the only changed DOM which allows to keep the HTML state as in HTML elements/web components as in browser page - like a selection.

5.1. Basic coding principles

5.1.1. Declaration

is done by `<custom-element>` with XSLT template inside.

```
<custom-element tag="greet-element" hidden="">
  <slot> Hello </slot> World!
</custom-element>
<greet-element></greet-element>
<greet-element>👋</greet-element>
```

Hello World!

👋 World!

The **tag** attribute defines the custom tag to be used later for rendering the content.

5.1.2. Preventing the side effects of declaration.

In order to avoid the showing of template content, use the **hidden** attribute as in sample above. It would prevent displaying of the content but not the DOM like styles and injection into page content. To avoid the DCE declaration interfering with the page, wrap the body into `<template>`.

```
<custom-element tag="dce-1">
  <template><!-- template needed to avoid styles leaking into global HTML -->
    <style>
      button{ border: 0.2rem dotted blue; }
    </style>
    <button><slot>Blue borders</slot></button>
  </template>
</custom-element>
<button>Default border</button>
<dce-1>1st</dce-1>
<dce-1>2nd</dce-1>
```

Default border

1st

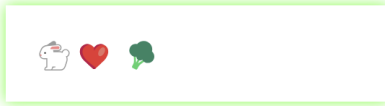
2nd

The wrapping into **template** has longer syntax but is safer than **hidden** attribute in current browsers.

5.1.3. Templating features: slots

`<slot>` from HTML standard allows to substitute the template parts marked by named slot element


```
<custom-element tag="dce-1-slot" hidden="">
  🐾❤️<slot name="slot1"> 🍄 </slot>
</custom-element>
<dce-1-slot></dce-1-slot>
<dce-1-slot><i slot="slot1"> 🥕 </i></dce-1-slot>
```



5.1.4. Templating with XSLT

{title} is the synonym for <xsl:value-of select="title">. The XPath selector gives as DCE data access as the full XPath power of XSLT 1.0.

```
<h3> {title} </h3>                                <!-- title is an attribute in instance
                                                    mapped into /*/attributes/title -->
<if test="//smile">                                <!-- data-smile DCE instance attribute,
                                                    mapped into /*/dataset/smile
                                                    used in condition -->
                                                    <!-- data-smile DCE instance attribute, used as HTML
                                                    content -->
    <div>Smile as: {//smile}</div> <!-- /datadom/dataset/smile -->
</if>
```

This syntax is working as for HTML text between tags as for attributes:

```

```

5.1.5. Data layer

Has a DOM structure associated with DCE instance and available through XPath in the template.

```
▼ datadom:
  ▼ attributes:
    data-smile: " 🐾 "
    pokemon-id: "1"
    title: "bulbasaur"
  ▼ dataset:
    smile: " 🐾 "
  ► payload: {span xmlns="http://www.w3.org/1999/xhtml" slot="": '
    slice/:
```

/datadom is a root /attributes have a list of attributes where value matches the attribute' one. It is also a default XPath node. /dataset holds data-xxx attributes / payload is a DOM content of DCE instance on the moment of initialization /slice is the map of component **slice** name to its value. The "change" event would trigger its update.

5.1.6. Slice

Provides the mechanism of live data feed from the element with **slice**="xxx" attribute to the **/slice/xxx** from its value. It is propagated during initialization and later updated by either change event or by event name defined by slice-update attribute: `<input type="text" value="Type time update" slice="txt" slice-update="keyup">`

The custom elements in order to be DCE-compatible, have to expose its data over **value** property and emit proper event(s).

5.1.7. Browser API access

Is done by custom elements via **value** property and accessed via **slice**.

Currently **URL/location**, **localStorage**, and **fetch** API implemented as part of <custom-element> package. See the demo pages ¹ for details.

5.2. Modular development

<custom-element> can have the template hosted inside or referenced from the external URI by **src** attribute. For relative in the HTML DOM, use the hash URL to element with ID.

```
<template id="template1">
  <slot> Hello </slot> World!
</template>

<custom-element tag="dce-internal" src="#template1"></custom-element>
<!-- no need for loading fallback as the template exists -->

<dce-internal>👋</dce-internal>
<dce-internal></dce-internal>
```

👋 World!

Hello World!

URL also can be a generic link from which the DOM is going to be read. It means the valid content is HTML, SVG, MathML, or generic XML to be used as template. In case the URL follows hash #ID, the DOM subtree with this ID is used as template. That makes HTML file a library, i.e. container for multiple templates. Example #6².

¹ <https://unpkg.com/@epa-wg/custom-element/index.html>

² <https://unpkg.com/@epa-wg/custom-element@0.0.18/demo/external-template.html>

```
<custom-element src="html-template.html#wave">
  <template><i>loading HTML from templates file ...</i></template>
</custom-element>
<custom-element src="html-template.html#dwc-logo">
  <template><i>loading SVG from templates file ...</i></template>
</custom-element>
<custom-element src="html-template.html#sophomores-dream">
  <template><i>loading MathML from HTML file ...</i></template>
</custom-element>
```



$$\int_0^1 x^x dx = \sum_{n=1}^{\infty} (-1)^{n+1} n^{-n}$$

From external HTML file: <b id="wave">
<svg id="dwc-logo" xmlns=http://www.w3.org/2000/svg³ ...
<math id="sophomores-dream">...

5.3. Styling

By default <custom-element> inherits container CSS. Which means the page developer can tune up DCE internals. But usually should not be necessary as the styles defined within DCE would be applied to instances and do not leak to the page level even if they written with very basic rules. It is safe to write css rules on “top” level which would be applied for this DCE.

```
<template id="template-5">
  <style>
    color:green;
    i{ color: blue;}
  </style>
  <p>
    green
    <i>blue</i>
  </p>
</template>
<custom-element src="#template-5"></custom-element>
```

green blue

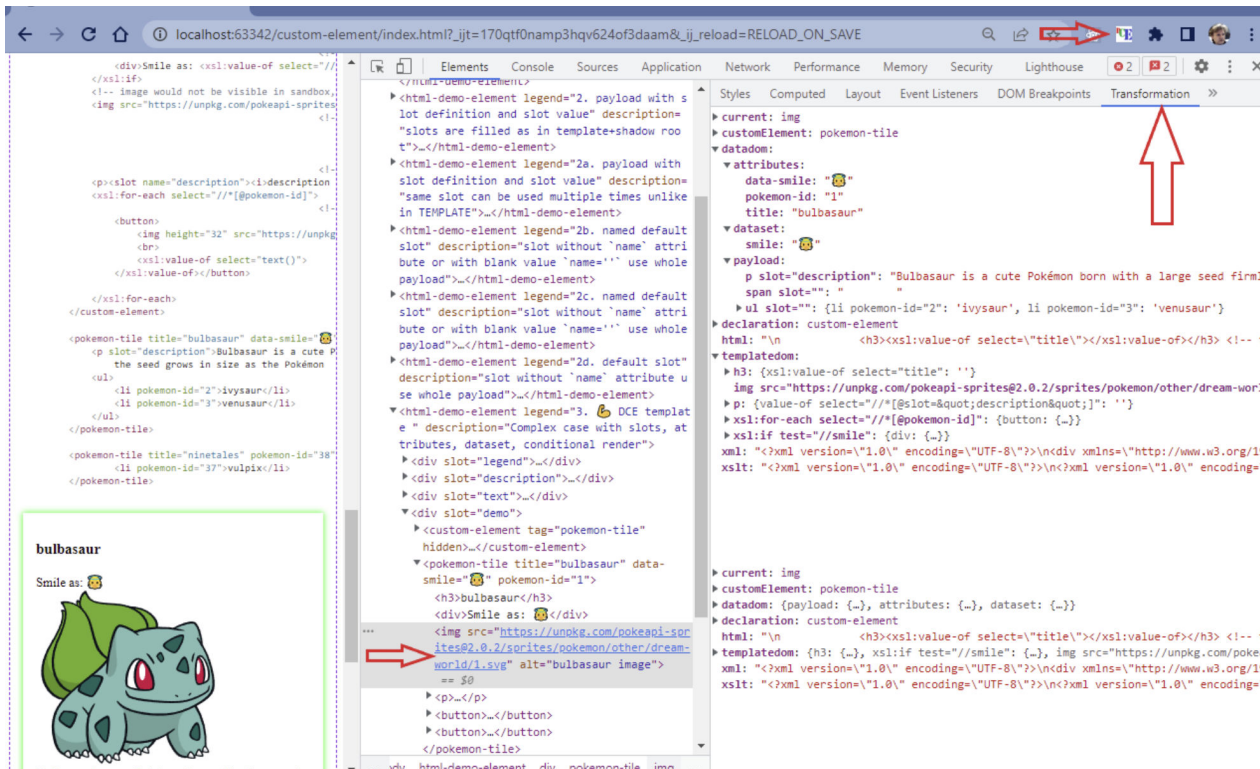
More CSS samples ⁴

³ <http://www.w3.org/2000/svg>

5.4. Development support

Instructions in README show on how to add IDE support for custom tags created by <custom-elements>. It would make the consumer of DCE life easier.

During development of DCE itself, there is a need to see what data is available and how they would be injected into template. The Chrome/Edge devtool plugin⁵ adds the “transformation” panel to DOM explorer next to CSS.



It displays as data set as template in tree-view and as XML string to copy for debugger in your favorite XSLT IDE. Visual Studio Pro allows to debug XSLT step-by-step with call stack, breakpoints, and variables inspection.

In-browser step-by-step debugger is not ready yet. But we can work on it. Are there volunteers in the audience?

6. Target Audience

Who would benefit from <custom-element> adoption?

- High Security Audiences

High-stakes environments like government, finance, and the military demand robust attack resiliency, making them prime candidates for <custom-element> and Syngرافact.AI' platform.

⁴ <https://unpkg.com/@epa-wg/custom-element@0.0.18/demo/scoped-css.html>

⁵ <https://chromewebstore.google.com/detail/epa-wgcustom-element/hiofgpmmkdemdbdogipagmbbb-mefefhbl>

- Web Apps on the Rise

Platforms with dense 3rd party integrations, such as CMS, social networks, and project management suites, stand to benefit profoundly from Syngrafact.AI` solutions.

- CMS/Publishing systems

Google Sites, Adobe Experience Manager, Vercel, Wordpress, Drupal...

- Application portals

LifeRay, IBM WebSphere, Drupal...

- Social network sites/apps

With IoT devices, games, ads, etc. integration the number of 3rd party apps would burst on another level.

- Project management suites

Jira, Confluence, GitHub,...

7. Conclusion

With <custom-element> the browser acquired a platform for declarative applications creation and web application owners the ability of safe inclusion of 3rd party code. With the security barrier replaced by secure platform, the empowering web applications by external features would become a business decision instead of security safeguards effort. Which should result in increase of web applications fusion on the level of magnitude.

Would your business benefit? Please share on

<https://fosstodon.org/tags/declarativeCustomElement> ⁶

Or comment your wishes on GitHub by creating an issue or discussion thread.

Anyone is welcome to join the Open-Source project:

<https://github.com/EPA-WG/custom-element> ⁷

References

- [1] *Declarative Syntax for Custom Elements*. WICG webcomponents proposal strawman. 2018 <https://github.com/WICG/webcomponents/blob/gh-pages/proposals/Declarative-Custom-Elements-Strawman.md> ⁸
- [2] Web Components CG. *WCCG in Declarative custom elements*. 2022 <https://github.com/w3c/webcomponents-cg/issues/32> ⁹

⁶ <https://fosstodon.org/tags/declarativeCustomElement>

⁷ <https://github.com/EPA-WG/custom-element>

⁸ <https://github.com/WICG/webcomponents/blob/gh-pages/proposals/Declarative-Custom-Elements-Strawman.md>

- [3] Sasha Firsov.*custom-element* Polyfill and POC for DCE proposal. 2024 <https://github.com/EPA-WG/custom-element> ¹⁰

⁹ <https://github.com/w3c/webcomponents-cg/issues/32>

¹⁰ <https://github.com/EPA-WG/custom-element>

Why Adding Some CSS Isn't Enough

Anne Rudolf

DOCUFY GmbH

<anne.rudolf@docufy.de>

Abstract

CSS is often understood as an easy styling language. But when it comes to large projects, structure is mandatory. Printed user manuals use various XML constructs, which are styled differently per customer, document type or even within the same document. But the more layout requirements there are for the same semantic construct, the more complex styling becomes.

To manage the amount of CSS for print and web, it seems obvious to consult concepts from web CSS like OOCSS, BEM or ITCSS. Although they focus on web interfaces, they point out the use of visual patterns. With them styling is detached from content and structure, and it becomes scalable.

This talk provides some ideas on how to structure CSS projects based on safety notices as an example.

1. Introduction

This is going to be the usual problem-solution-story about layout designing with CSS, how structure ideas from web CSS can be adapted for print and finally how XSL and XPath might be part of a solution. SCSS and XSL are used for the layout configuration tool *DOCUFY Layouter*. Among other things *DOCUFY* develops software for technical documentation. To publish user manuals and the like, customer XML is transformed into the in-house exchange format DYXML. This is the input format for the *DOCUFY Layouter*. In this context it is sufficient to describe DYXML as a restricted XHTML5, split up into topic files that are referenced by a main file called tree. As the content structure is HTML and XSL-FO is no longer developed, CSS was the obvious choice. But XSL was also needed from the start to generate one linearized XML file for PDF and to add directories, chapter numbers, page area content and such. At first *DOCUFY Layouter* was only able to output layouts for PDF, but later two HTML formats followed. Thanks to CSS this was easy.

2. Safety Notices as an Example

To illustrate the problem, safety and warning notices give a good example, because they became highly configurable. For technical writers they are an impor-

tant part of user manuals and therefore also very sensible. In DFXML a safety notice consists of a signal word panel, which has a signal word and a safety alert symbol. If there is a specific hazard, it might have a symbol panel. And naturally, there is the message itself and its caption. As the images are part of a certain safety standard, they are provided by the exchange format.

```
<div data-role="warning" data-class="safety-iso_7010">
  <div data-role="signalword-panel">
    
    <p data-role="signalword">WARNING</p>
  </div>
  <div data-role="symbol-panel">
    
  </div>
  <div data-role="message-panel">
    <p data-role="caption">Risk of electrical shock</p>
    <ul/>
  </div>
</div>
```

Figure 1. XML structure for safety notice

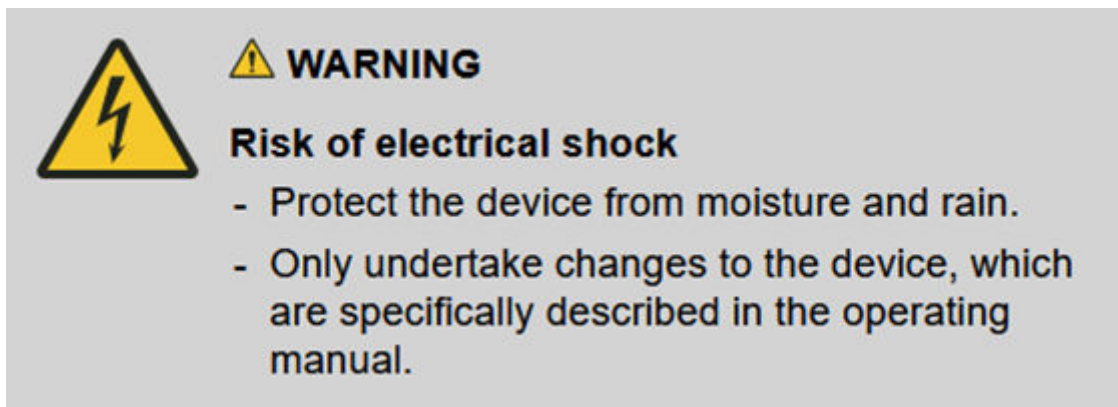


Figure 2. Styled output of PI-FAN safety notice (ISO 7010) [5]

The first design of safety notices that *DOCIFY Layouter* supported corresponded to the design of the earlier XSL-FO transformation. It is configurable only to a certain degree and does not use any exchange format for publishing. Nowadays the simplest way to style this danger notice would be the grid layout.

```
[data-role = warning] {
  display: grid;
  grid-template-areas: "symbol signalword" "symbol message";
```



```
    grid-template-columns: 4em 1fr;
    gap: 0.75em;
    padding: 0.75em;
    background-color: lightgrey;
}
[data-role = signalword-panel] {
    grid-area: signalword;
}
[data-role = symbol-panel] {
    grid-area: symbol;
}
[data-role = symbol-panel] img {
    max-width: 100%;
}
[data-role = message-panel] {
    grid-area: message;
}
[data-role = safety-alert-symbol] {
    max-height: 1em;
}
[data-role = signalword] {
    display: inline-block;
    font-weight: bold;
}
```

Figure 3. Grid layout for safety notice

At the start of a project, CSS selectors consist of one class and sometimes an element. Selectors have a specificity of 1-0 or 1-1. But there are other risk levels, and the symbol panel is optional. So far CSS selectors, with the help of SCSS, are still organized. SCSS is a precompile language that provides logical structures, mixins and selector nesting. SCSS files are split by DYXML structures and organized in common or output specific folders.

However, the number of requirements increases over time. The ANSI safety standard was implemented. For a multi-column layout, a more compact design variant was needed. Configurations for colors, borders and fonts were added. The safety standards do not define a concrete visualization of safety and warning notices. But because they are a sensible part of the user manual, customers tend to have many requirements for them. This led to a layout configuration to handle arrangement variants of safety notices, which were implemented with SCSS, and some additional wrapper elements generated by XSL. For the safety chapter that describes general hazards at the start of a manual, a reduced design was necessary for the ISO standard. For the ANSI standard safety notices within safety chapters and within instructions must not be noticeable, meaning that they have

another arrangement variant. Without the panel of the signal word an exclamation mark must be displayed, which might be language specific.

In addition to the left-to-right layout the right-to-left layout was implemented for languages like Arabic. Many CSS declarations might not have a direction, but some might. Because there are also HTML layouts, the compact and embedded arrangement variants are needed for the responsive layout on mobile devices. Due to cross-media there might also be technical differences between the consumers of CSS. For *DOCIFY Layouter* these are the browsers and the *Antenna House Formatter*. The Formatter only supported grid layout recently. Therefore, the arrangement variants for PDF had to be implemented as tables and floats.

With these requirements it becomes harder to manage selectors. SCSS can generate all necessary rules, but it cannot solve the problem that specificity is no longer predictable. With every added requirement, a specificity conflict might occur, resulting in even higher specificities if possible to implement with CSS. For the signal word panel alone, there might be more than 10 selectors. Because specificity is implicit for CSS selectors, it is barely scalable. There is no granular precedence as known in XSL.

From the perspective of the CSS cascade, there are no practical alternatives to specificity and order. The `!important` flag, inline styles or shadow context provide no sustainable solutions or are simply not possible in the environment. Browsers started to support CSS cascade layers 2 years ago, see [caniuse.com](https://caniuse.com/layer)¹. There seems to be no PDF generator that supports this feature yet. It might alleviate the specificity problem, if you work with stylesheets from various sources [9]. But that is not the use case here.

3. Structure Ideas from Web CSS

CSS originates from the web environment. Despite their differences, web and print layouts are both user-driven. But since web CSS developed simultaneously with the web itself, it already faced the question of organizing large projects. As a first overview I will summarize some developments that proved to be interesting.

Unsurprisingly, managing large CSS projects started with good practices also known from other languages: Selectors should be as general as possible, be consistent, split up files, comment the code.

As the web evolved there seemed to be the necessity for more structure. Nicole Sullivan introduced an object-oriented concept in 2008 and described those modules as re-usable visual patterns [8]. Structure should be separated from the skin [8]. However, she focused more on the concept itself than on its concrete realization.

¹ <https://caniuse.com/?search=%40layer>

Based on her ideas the naming methodology BEM (Block-Element-Modifier) by Yandex offered a more concrete idea to keep specificity as low as possible [6]. It adds a class for every visual pattern, which it calls blocks and its descendants.

```
.block-name, .block-name__element-name { }  
.block-name.block-name--modifier { }  
.block-name__element-name.block-name__element-name--modifier { }
```

Figure 4. BEM naming methodology

From 2011 onwards, architectural concepts were created that sort CSS rules into types or levels. One of the first was SMACSS (Scalable and Modular Architecture for CSS) [7]. It proposes the rule types ‘base’, ‘layout’, ‘module’, ‘state’ and ‘theme’. The base rules are element selectors, the layout rules describe the general layout of the web page. Modules such as sidebars or search bars are arranged within the layouts and might have states and different themes.

The SMACSS structure was strongly inspired by websites. It would take effort to adapt e.g. the ‘state’ rules for print. SCSS partly supersedes the ‘theme’ rules and separating ‘theme’ and ‘state’ from the ‘module’ might make it harder to maintain.

Parallel to architecture ideas, there are other developments that abandon the specificity of selectors [6]. One example is Atomic CSS, with which a class is added to an element for every CSS declaration, meaning per property and explicit value. A framework is essential for this approach. Because users can individually set any border width etc., it is impossible to implement this kind of solution.

ITCSS (Inverted Triangle CSS) by Harry Roberts turns 10 this year and because its levels are very abstract, I found it the most promising structure to use for cross-media. Its levels are based solely on CSS, perhaps because CSS nowadays does not only style websites but also web applications. The abstract approach makes it easy to use the concept also for print. Moreover, levels of ITCSS express a certain specificity range, they include the BEM methodology and there exists a version that considers SCSS [4] as it became more prominent in the CSS community.

ITCSS proposes seven levels. The specificity increases per level, which means that the last levels have the highest specificity and contain the most explicit rules.

1. ‘settings’: global variables and configurations
2. ‘tools’: functions and mixins
3. ‘generics’: general CSS selectors and @-rules
4. ‘elements’: bare HTML element selectors
5. ‘objects’: (if needed) general layout and structure classes
6. ‘components’: re-usable visual parts/patterns (e.g. search bar, directory entry)

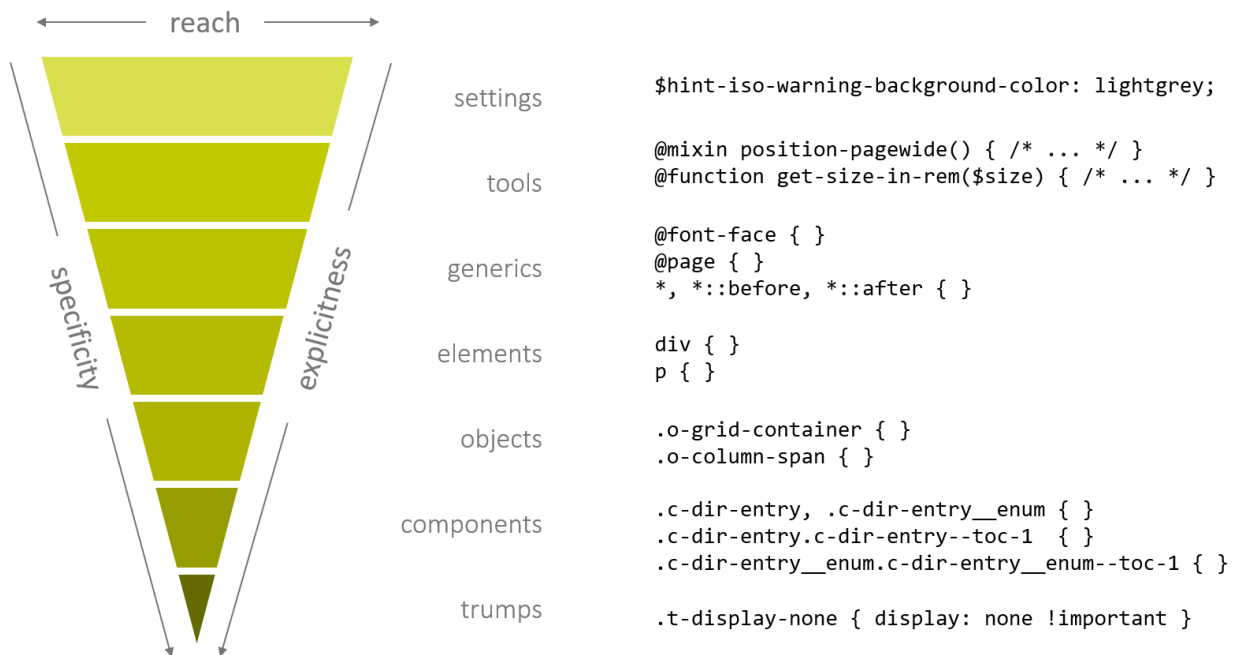


Figure 5. ITCSS levels with (S)CSS examples

7. 'utilities' / 'trumps': style rules to overwrite preceding levels for specific contexts

According to BEM, components in ITCSS may have descendants that depend on the structure and layout of the component. They or their descendants may also have varying themes that are represented by an added class value. To differentiate between the levels, class values may have prefixes.

However, all web CSS structures assume that HTML is provided by the layout and then content is added. The layout may even decide which content is displayed. With user manuals the content is first and the layout adapts to it.

4. Content-Driven Visual Patterns

To use ITCSS for the cross-media stylesheets of *DOCIFY Layouter*, the DYXML elements must be mapped to visual patterns. With this separation styling becomes scalable. Safety notices as a semantic construct in DYXML must then be mapped to their looks, in this case the arrangement variants. Each of these visual patterns might have visual variants per risk level or reduced styling. The mapping can take XML context, layout configurations or output characteristics into account. Other semantic constructs might have only one visual pattern. Tables may be styled by element selectors alone. But as soon as there are different stylings for tables, they too have variants.

As XSL is already in use, this leads to a simple XSL template that extends the XML element by the class values for components, objects or utilities. But it takes

effort to map every semantic construct that extends the styling of the pure XML element.

In this way, the strengths of XPath and CSS are utilized. XPath selects the elements. Unlike CSS selectors, it can select more granular and take the following-axes into account. Meanwhile, CSS selectors stay simple and predictable even with multiple requirements. So, CSS rules can provide specificity and styling.

```
<xsl:template match="dyxml:div[@data-role and contains-token(@data-
class, 'safety-iso_7010')]">
  <xsl:variable name="component" as="xs:string" select="'hint-iso-' ||
$dl-param-hint-iso-variant"/>
  <xsl:variable name="has-symbol" as="xs:string?" select="
    if (descendant::dyxml:div[@data-role='symbol-panel'])
    then ($component || '--has-symbol') else ()"/>
  <xsl:variable name="level" as="xs:string" select="$component || '--'
|| @data-role"/>
  <xsl:variable name="reduced" as="xs:string?" select="
    if (
      ancestor::dyxml:section[contains-token(@data-class, 'safety-
chapter')]
      and $dl-param-hint-safety-chapter-reduced
    ) then ($component || '--reduced')
    else ()"/>
  <xsl:variable name="mobile" as="xs:string?" select="
    if ($dl-output-responsive) then ($component || '--mobile') else ()"/>
  <xsl:copy>
    <xsl:attribute name="class" select="
      string-join(($component, $has-symbol, $level, $reduced), ' ')/>
    <xsl:apply-templates select="@* | node()"/>
  </xsl:copy>
</xsl:template>
```

Figure 6. Linking XML elements to visual patterns

The modularization of visual patterns might make a few selectors obsolete that simply overwrote others. As the selector for the right-to-left layout is compiled by the SCSS across the entire stylesheet, this attribute is not queried in the XPath. As it is a general setup, the specificity of the CSS selectors is still predictable.

Retrospectively and also per element, you can see four states of a CSS layout that are connected to the number of requirements. *DOCIFY Layouter* started with approx. 100 layout configurations and one output. Currently there are more than 1000 configurations, some more complex than others, and three outputs and it is still growing.

First safety notices could be selected easily via CSS selectors. With increasing requirements, they could still be selected, but selectors became more specific.

Important to notice is that visual patterns were there from the start, even if there is no organized mapping. But because each is equivalent to a single XML construct it was less obvious.

In the next stages adding attributes or elements per XSL is the only way to style the element. With increasing requirements even that becomes hard to maintain because the number of visual patterns increases, as shown with the example of safety notices.

Other semantic constructs with less requirements are in one of the preceding stages. Tables can have variants in the stylesheets of *DOCIFY Layouter*, but they do not need more attributes. Inline elements like a term displayed in a user interface are less complex and can still be easily selected with CSS.

For smaller projects, a pure CSS solution is possible to keep organized. SCSS aids CSS cascade layers perhaps in combination with ITCSS could provide some structure. But as soon as selectors tend to become more specific, XPath offers more possibilities to select elements and helps to keep CSS simple.

5. Conclusion

With increasing requirements selectors reflect XML semantics. With higher specificities, CSS projects become less scalable. There is the need for a better and more modular structure. XPath, if possible in the setup, can handle complex selections of element subsets far better than CSS selectors and may therefore be used to add attributes for visual patterns to the XML. One remaining challenge here is the comprehensible labelling of visual patterns.

For large or growing projects, it is not enough to simply add CSS. Structure is mandatory. However, it is equally important not to think in semantic categories, but in visual ones and to consciously differentiate between the two.

6. Bibliography

- [4] Ekene, Peter: *How To Solve Large-Scale CSS Bottlenecks with ITCSS and BEM*, 19 December 2019. <https://www.digitalocean.com/community/tutorials/how-to-solve-large-scale-css-bottlenecks-with-itcss-and-bem>
- [5] Gruenert, R. - Ziegler, Wolfgang: *PI-Fan*, 2015. www.pi-fan.de
- [6] Pfeiffer, Andrei: *The evolution of scalable CSS*, 24 January 2022. <https://andreipfeiffer.dev/blog/2022/scalable-css-evolution>
- [7] Snook, Jonathan: *Scalable and Modular Architecture for CSS*, 2012. <https://smacss.com/files/smacss-en.zip>
- [8] Sullivan, Nicole: *Object Oriented CSS*, 2009. <https://fronteers.nl/congres/2009/sessions/object-oriented-css>

- [9] Suzanne, Miriam: *A Complete Guide to CSS Cascade Layers*, 21 February 2022.
<https://css-tricks.com/css-cascade-layers/>

XML preserved from the past and into the future or?

Karin Bredenberg
Kommunalförbundet Sydarkivera
<karin.bredenberg@gmail.com>

The story about standards used now and for the future

Some have been saying XML is dead ¹ (we know that is not true), and the use we see for XML most often involves publications, but so much more relies on XML and XML being a part of the future. You might not know that the final resting place for a lot of the information (which will be called data in the rest of the text) created by agencies, municipalities and others is being saved for the future with the aid of digital preservation by archives both national and others no matter if it's a physical artefact or digital data. The digital data is exported in different ways out of the system on which it was created to facilitate the saving and reuse of data without requiring the originating system. The export involves creating some type of itinerary and transforming the data into formats suitable for moving the data to the archive or whoever is the receiver. In almost all cases, the itinerary is in the XML format following a standard, providing an XML schema for describing a transfer and all of the components required besides the data itself. The data will come in many different forms such as database dumps, images in various file formats, PDFs and XML documents.

A thank you to Jaime Kaminski for aiding me with my English language.

1. When preserving data is not really your game

So, how can you end up thinking about a final resting place for the data when your title and education are not about saving data for the future? Imagine striving for a bachelor's degree in computer science focused on programming and needing to write a paper to achieve that professional title. The paper is about transferring data in the cultural sector using a standard with an XML schema as its core to an archive. The standard most often used is METS ² (Metadata Encoding & Transmission Standard). METS is a standard for describing data and its

¹<https://web.archive.org/web/20220610095940/https://developpaper.com/is-xml-dead/> is a starting point for getting more information regarding the statement

²<https://www.loc.gov/standards/mets/>

administrative components to be transferred to another system or a digital archive. The administrative components (often called provenance) range from the creator's description to information about how the data was created. For example, what software was used or if it was digitised with a scanner, and rights like copyright concerning the data are applicable, as well as much more, depending on what is known. What does all this have to do with XML is most certainly your question now, well you see the standards being both standards from well-established standardisation organs and de-facto standards created from the need of them that are being used in making sure the past is available in the future is based upon XML. This is because XML is both human and machine-readable. Doing this type of evaluation of the standards in this area demands the skillset of an engineer to be able to discover all the facets and possibilities that can be used. Furthermore, the facets and possibilities are increasing and include the requirements of the EU Digital Single Market ³, now in its next phase EU Digital strategy where reuse and the 'once only' principle uses all types of data.

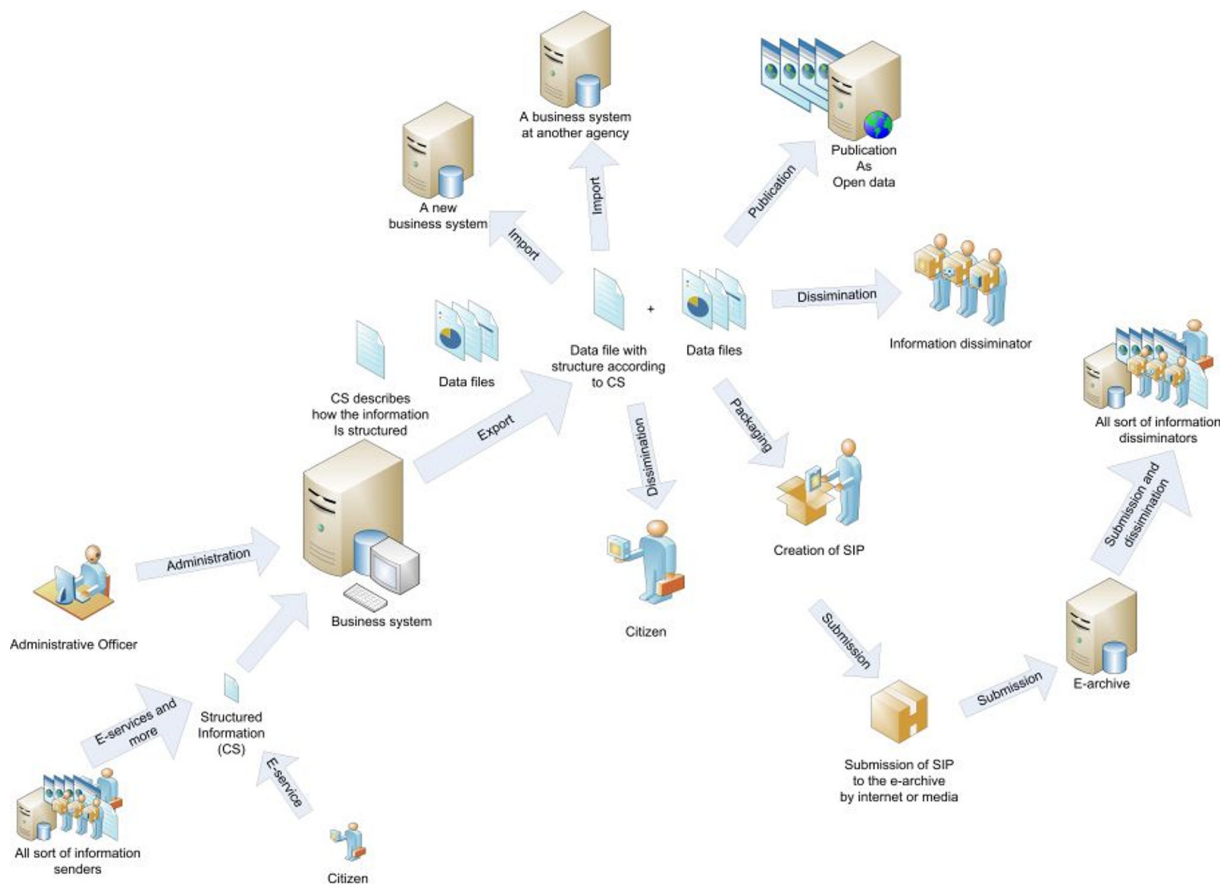


Figure 1. Illustration of common use and reuse of data by Karin Bredenberg

³<https://eufordigital.eu/discover-eu/eu-digital-strategy/>

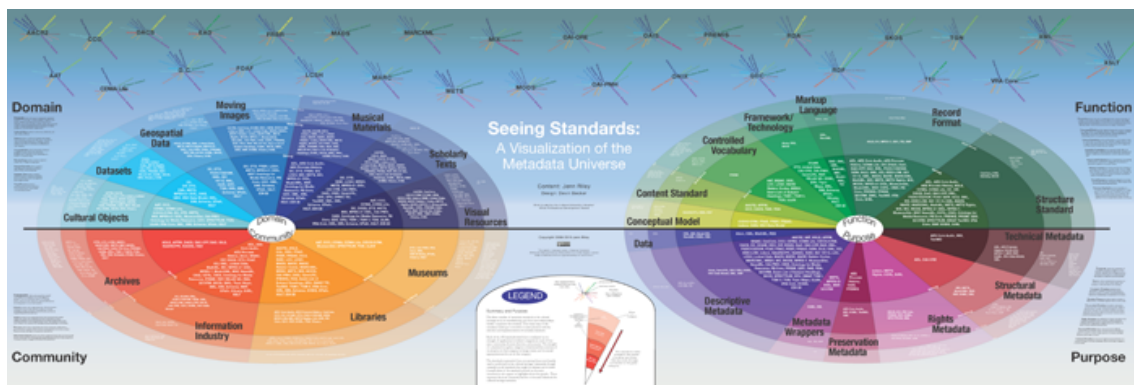


Figure 2. The visualisation of the Metadata Universe by Jenn Riley. (The recommendation is to look on-line)

There are a lot of standards within this area of transferring and storing data, and to emphasise what is being used is, in most cases, XML with the support of one or more XML schema. We need rules to be sure the data is in the expected form and structure. A setup with XML schemas that, if well designed, makes it possible for both humans and machines to understand and the data being saved easily. And yes, there are numerous uses and creations of XML schemas that make it impossible to understand the data they are supposed to house. So, when thinking about the future in the past, there are numerous standards based upon XML available. They still thrive, and behold, we are just now admitting the demise of xlink and creating our own attributes and elements to create links to files and other resources. On the other hand, there is also a big need to describe how to use these standards to facilitate a common understanding. This is, in most cases, made in specifications. You see, having a standard doesn't necessarily mean knowing what you should do is easy. Haven't we all been there and tried to understand what an ISO standard is saying we should do?

2. Standards, de facto standards and specifications

To fully understand the complexity of preserving data for the future, a look at Jenn Riley's Visualisation of the Metadata Universe ⁴ is needed. This visualisation has been around for a while, but it is still the best overview of standards like the ones from some sort of a standardisation institute like ISO or de-facto standards like METS and PREMIS, something which has become seen as a standard and used by many even without it being created and maintained by a standardisation institute, primarily used in the cultural sector, archives, libraries and museums for storing and transferring data. The cultural sector is usually responsible for caring for the past, from all types of analogue artefacts like documents and books to digital artefacts in many different file formats.

⁴<http://jennriley.com/metadatamap/>

What is lacking in the image is some (many) of the standards that utilise different types of linked open data standards like RDF, which adds to the number of standards being used. (I assure you; the cultural sector uses RDF nowadays, some more than others.) In the image, numerous standards are shown, which are displayed in the context of their function and where they are used. Almost all standards on the metadata map created by Jenn Riley have an XML format available described with a DTD or an XML schema. Both the ISO standard RelaxNG and W3C XML schema formats are used for the XML schemas. The choice depends solely on the skills of the schema's creator. At the same time, it's also common to ensure that all types of schemas are available, so transformations from RelaxNG to XML schema and vice versa are often used to facilitate both available versions. Yes, some have not left the DTDs. DTDs are still around since old software is still in use, and they are based upon using a DTD, and the funding for upgrading all parts to current methods and techniques is not available, so if it still works, it works. The fact is that many standards used in digital preservation have one or more interpretations in XML with an XML schema as the rule base. There are also written specifications explaining how to use the standard since most of them contain many elements and attributes, and all are unnecessary to make something future proof. Even so, it is important to understand that specifications are created to ensure that all understand and use the standards in the same way, despite all the possible ways you can use a standard, so we reach interoperability and reuse advantages. Several existing standards are currently undergoing revisions to ensure they are sustainable and fit for purpose.

In many cases, these revisions will not be backwards compatible, meaning that software and methods will need to be updated, and new specifications will be required to explain the revised standard. We also see new standards developed where linked open data is one part of the interpret, the other is still an XML interpretation. However, in the XML versions, there are accommodations for reusing values from ontologies and ensuring that it is possible to create relations between described objects and other objects described elsewhere.

3. Parts needed for storing data for the future

When we talk about digital preservation and the objects stored for the future, several different parts are needed. The digital object can be an image, a document, or a database, let's do it this way you name the digital object, it most certainly needs to be preserved for the future and be authentic and possible to show for a future being. So, we have the digital object itself, which is described together with technical metadata so the file/files can be catered for in the long run, we need to know the format, the size, and checksums, for example. We need to know who created the object to record its provenance and if that gives us some rules for using it. We also need to know how it came to be and its relation to other digital or physical

objects created by the same creator or other creators if that information is known. There is no end for what needs to be known; new factors, digital objects and authorisations are coming into play daily. Look at the digital signatures where they need to be possible to validate for some time and then known to have existed for the object in the long term; no one will be able to validate a digital signature in 50 years. And how should we integrate all these AIs and automation? What information must we preserve to know that AI created or influenced the information? Will that information be needed? How to ensure we don't create the same thing again since no one has made this before....

In short, data with a digital origin is a bit trickier to cater for compared to a physical document consisting of ink written on parchment. We still need to ensure that the goal of preservation is reached. It is well described in the preservation pyramid created by Pricilla Caplan for the PREservation Metadata Implementation Strategies Editorial Board ⁵ (PREMIS EC) to show the means and goals of the digital preservation for a digital object. PREMIS EC is the maintainer for the only standard (de-facto standard) that is available for preservation metadata, PREservation Metadata Implementation Strategies ⁶. PREMIS offers a data dictionary, and it is being used worldwide, so you might even be using PREMIS without knowing it. It provides the option to understand what needs to be recorded regarding the preserved objects, the rights that may need to be taken into account, the agents that do something with the object, and the events that happen to the object. All these parts are needed to fulfil the preservation goals.



Figure 3. The preservation pyramid illustration created by Pricilla Caplan used in PREMIS.

⁵<https://www.loc.gov/standards/premis/premis-editorial-committee.html>

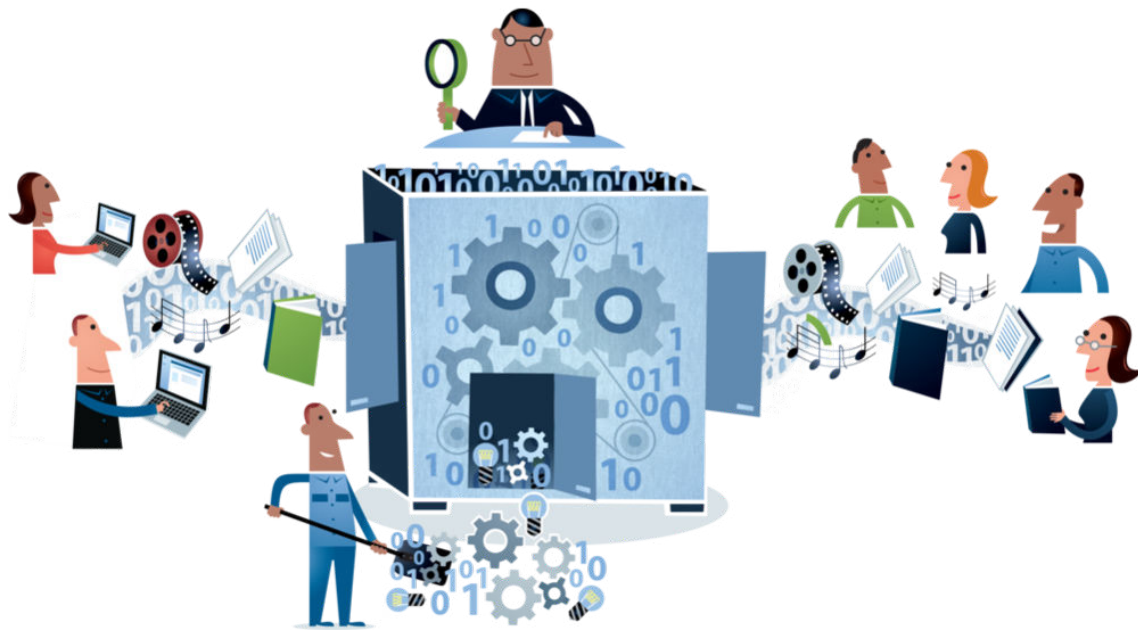
⁶<https://www.loc.gov/standards/premis/>

4. Why bother?

So, what does all this mean, and why does this even matter to all attending XML Prague? Well, you see, your help is needed to ensure XML is not only something for creating nice print documents or focusing on creating new XML languages since we all want something simpler. There is a need to ensure that XML and all the required parts like XSLT, Schematron and XML schema are, if not updated, but maintained and at least available for the never-ending future. Achieving that goal means ensuring that all the data created today can be stored and reused in the future with well-known standards. We need this sustainability and future thinking since the nicely printed PDF, which is a common document format, will not be standing on its own and will not be self-descriptive about its origin in the future without a lot of accompanying metadata, which gives the creator and the circumstances that it became to be created and known. You all need to be our champions in the longevity of XML!

As a champion, you will need to be familiar with several different parts, which will make this tricky to handle in some cases. The number one thing to remember is that when it comes to preserving something for the future is, who is doing the work? Usually, the one doing it is a cultural sector worker and, most commonly, an archivist. This position requires you, in most cases, to get a university degree in the humanities, where you then add classes to become an archivist, learning how the agencies have been working and how to order things following a specific national scheme which is specific to each country. This education (yes, there are a few exceptions) does not take into account the change that has happened with data; it has moved from analogue to digital, which means the archivists are a little bit out of their ordinary knowledge zone and are not prone to understand that when we are talking about digital preservation, it is not a one-person job, it's a collaboration between many different skill sets. This means that people with minimal technical skills are trying to solve problems that are a piece of cake for another profession. Think about writing transformations in XSLT. Can everyone really write an export into some form that takes a full data set from a database describing all the actions done in a company (aka. records management), and then an XSLT transformation which turns it into something fit for importing to another system or preserve it for the future in XML following a specific XML schema? The question is, should they? The answer is no; you need the correct knowledge and expertise to do that. Sure, you can learn parts of it, but to make sure that the result is possible to preserve for the future, everything needed needs to be part of exports and transformations and follow the standards, de-facto standards and specifications available. At the same time, there is also the opposite: that's XML, that's programming, that's IT! This means that some of the cultural workers just leave the table, and there we are, will anything be preserved

for the future? That's the first part of the ones doing the work. The other part of this work is the other professions, and they also need to step up their game.



Digitalbevaring.dk

Figure 4. We need to work together! An image from https://commons.wikimedia.org/wiki/File:Knowledge3_DigitalPreservation.png

We all need to start working together and ensure the data created will last forever. Looking deeper at the second part of workers needed to preserve the data for the future is the technically skilled professions. A question: how many times does a programmer validate the export result in XML before setting it into production? The answer is that in many cases, this never happens until the data ends up at the place where it is supposed to be stored forever, and there you have the expertise in understanding the data and the XML, all the tests are run, and voila it's not well-formed or valid. Many things need correction, which could have been avoided if a validation was made before sending it out as "The export". Where does this leave XML as being a part of saving data for the future? Well, the demands are very often that a system needs to create an export in the form of XML following a demanded XML schema since, so far, it is what makes it easier to understand the data without its originating system in a neutral way. What often is lacking is the knowledge of how to ensure the XML is valid and not just well-formed. Even not well-formed XML might be the export result... We in the XML community need to continue to inform about XML and how it needs to be well-formed and validated, how to use Schematron to create validations and not only use what XML schema provides regarding the use of the correct attributes

elements, we also need to and in the validation of external value lists, element and attribute dependencies but also of the content to make sure the data is understandable for the future user of the data. All this so that when there are expectations on the software creators to make sure when there are demands on an export suitable for long-term preservation made by the software users, it is implemented and is well-formed and validates.

5. About specifications for saving the data

As written earlier, understanding the standards that are being used in the area of digital preservation specifications is critical. Specifications that are aimed at supporting many, so all use the same interpretation of the standards. This is still not enough; there is a need for numerous guidance documents ranging from writing XML to using a specification. And all this since you cannot count on the person implementing standards, de-facto standards with the aid of the specification and its validation to have the appropriate background knowledge and all the correct skills. We are lucky! Within the European world of digital preservation, there are specifications and guidance documents to aid with all these parts. More work is needed and will be done; the start is there!

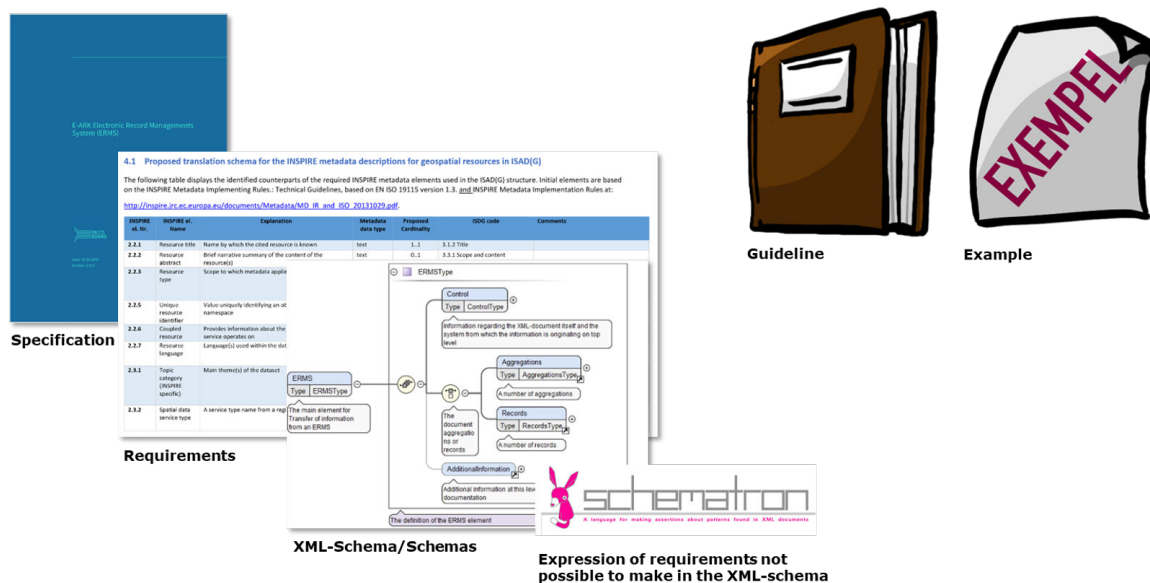


Figure 5. All the parts that constitutes a specification following the eArchiving specifications. Image by Karin Bredenberg

The story of eArchiving specifications originates with work that started at the National Archives of Sweden and was enhanced in the E-ARK project ⁷. The project delivered seven draft specifications, which were all built upon creating

⁷<https://eark-project.com/>

profiles of how to use standards and de-facto standards. These can be split into four specifications describing profiles of the standard METS and then: one for electronic records management, one for geodata, and finally, one for using the SIARD⁸ standards. The project ended, but no one wanted the work to be forgotten and unused, as is so often the case. So, to make sure these specifications did not stop evolving, the project created the “Digital Information LifeCycle Interoperability Standards Board” (DILCIS Board⁹). During the project E-ARK4ALL¹⁰ (which was responsible for setting up and maintaining the eArchiving Building Block), the Board took over the specifications. It brought them to a stabilised state, published them and then developed more specifications. Work is carried out together with the experts of specific content, so the experts write the specifications. This means that if you are an expert on a type of data see that you think others would benefit from using in a common way, you can contribute by either creating a new specification or if you have a commonly used schema or specification to send in a suggestion of it getting an endorsement. It is also possible for all that have an interest in the specifications no matter if you are a data producer or responsible for developing an export or software following the specifications to give comments and suggestions using issues in GitHub¹¹ for the specification you want to comment on since all the specifications are maintained there. If GitHub is not your preferred way of giving comments and suggestions it is also possible to send an e-mail¹² to the DILCIS Board, which will be registered in GitHub so all comments and suggestions are thoroughly documented. The Board is currently set up with ten members who handles the specifications. The Board is simultaneously the principal producer of and maintainer¹³ of specifications in the current eArchiving Initiative¹⁴ through the E-ARK Foundation¹⁵ and the eArchiving Common Services Platform (E-ARK CSP).

There are two types of specifications, which will be described more in the following sections:

- Description of an information package
- Description of the content

⁸<https://dilcis.eu/content-types/siard>

⁹<https://dilcis.eu/>

¹⁰<https://e-ark4all.eu/>

¹¹<https://github.com/DILCISBoard>

¹²info@dilcis.eu

¹³As maintainers the DILCIS Board is responsible for keeping the specifications up to date but not just updating as wanted instead it has been created procedures for the creation and maintenance so it is following a schema for revisions, <https://dilcis.eu/guidelines/procedures>

¹⁴<https://digital-strategy.ec.europa.eu/en/activities/earchiving>

¹⁵<https://www.e-ark-foundation.eu/>

6. What the specifications for an information package describes

The final resting place for the data is the e-archive, which is in most cases built upon the ISO standard, Reference Model for an Open Archival Information System (OAIS Reference Model)¹⁶ and its use of information packages, the Submission Information Package (SIP), the Archival Information Package (AIP) and the Dissemination Information Package (DIP). More parts are described in the reference model, but everyone focuses on the information packages.

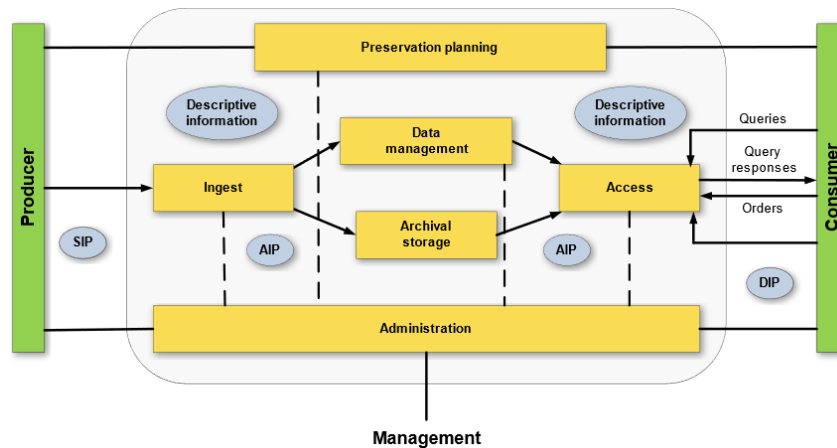


Figure 6. Image of OAIS reference model. Image courtesy of the DILCIS Board.

The different packages in the OAIS reference model are described, and their inventory or manifest is stated with METS when you are using the eArchiving specifications. The manifest in the form of the METS document describes all the parts of the packages. This way, both machines and humans can understand the package. METS itself is a rather open standard with only one mandatory element when you are using version one (as I wrote earlier, many standards are going through revisions), which is a structural description of the package. The standards also demand the creation of a profile for the exact use case describing the use of the standard and its elements and attributes. Others can then reuse the profile with the same use case or as a starting point for adding requirements.

7. Besides a package we need content specifications

Well, we now have a package described in XML. We need some content such as images, PDFs or data structured in XML that gives us the data recorded in the package. This means we might have one or more XML documents to describe the information and its structure and another XML document to describe the whole package with all the content but not always the structure of the information itself. We can even handle a relational database in this way, extracting it in XML format

¹⁶<https://www.iso.org/standard/57284.html>

following the standard SIARD and then packaging it in a SIP for transfer to the archive. Specifications of content are growing steadily. There are lots of different kinds of information that need to be described. Luckily, there are many standards for describing information in XML, so there is no need to reinvent the wheel; you just need to write specifications to explain how you should use them. If there is no specification available but there is a standards-based format, you can, after agreeing upon this solution with the receiver of the information, just use the information package specification for creating the manifest and then transfer the information.

8. The result

With an information package that has both information about the content and the data itself, it is now possible to store it in an e-archive or with a more often used term in a digital preservation system for the future. It might be as file/files as they are or in a compressed format like a TAR file. There will be a need to ensure that the data is not altered so its authenticity is not changed, formats don't become obsolete and so on. But we leave this for the digital preservation system and utilisation of the de-facto standard PREMIS.

9. Hurdles to pass

What does the use of all this XML mean? Well as noted earlier, it creates demands on a number of roles involved in the data's both creation and export possibility from the systems used as much as on the one receiving the data and taking care of it for the future. And don't forget the creators of the standards and XML-schema who need to be able to make sure what they create is suitable.

The first hurdle is as simple as creating the software and planning the creation of the export and the resulting XML document, but not understanding how to connect it to its XML schema, thus not, in some cases, even creating well-formed and invalid XML.

The second hurdle is the belief that validation is something that happens on its own without any or some intervention.

The third hurdle is the difficulties that occur when the average user does not know how to validate the XML in combination with Schematron because of poor or no knowledge of either XML or Schematron and not having access to somebody with the requisite knowledge. It has become more common that validation of an XML schema is aided with a specialised Schematron that adds conditional validation and other validation rules like externally hosted value lists. We must not forget that Schematron adds its own complexities when combined with a non-relaxNG schema. This complexity can be overwhelming when you don't know XML or only have a basic understanding of it. Many guidelines and tips are necessary to validate the data.

Some extra hurdles can be added, such as the belief that attributes are for machines and elements are for people. We also need to fall back to the beginning of this text; XML is dead, so why are we using that? And add in, why don't you create an API so it all becomes automatic? Well, an API where the form and structure of the information in the originating system are always unknown is not easy to create. Topping it with, 'but JSON is so much easier to use, why don't you provide this in JSON'?

10. Where does all this leave us?

I get back to the need for champions, champions for XML to prove it is not dead and that we all need to do digital preservation together no matter what our profession is! We need to be the knights in shining armour that guides all users to sustainable use of the standards, de-facto standards, and specifications, and we need to ensure that data created today will be around in the future and in a form that can be used and reused.

11. Some final remarks on how to prepare for and thus avoid the hurdles

Preservation of our data requires us to be champions in many ways besides making sure that XML will be around forever. I will here give some further bullet points about what is required and some benefits, so we don't run into any hurdles and thus lose our data.

Remember that all no matter how you are connected to the data are welcome to give comments on both standards, de-facto standards and specifications when there are call for comments open or by creating issues in the way the maintainers want them.

When you are a producer of data:

- Ensure that your system can export the data from the system in a structured way using one or more XML schemas, preferably adhering to a commonly agreed-upon specification or standard.
- Make sure that the system creator, whether in-house or a system you buy, sets up a road map for following specifications and standards, including when a new version will be implemented and how the cost will be covered.
- When the requirements from specifications and/or standards are common for the systems an export from one system can be imported into another system more easily, thus facilitating easier system changes.
- Exporting in a common way, following specifications and/or standards, ensures that the data are ready for digital preservation in the long term.

When you are a producer of XML from a system that then can be transferred:

- Many software store data in a database, which means code needs to be created to extract data from it and place it in an XML document.
- When writing an export/import adhering to specifications and/or standards make sure to go through the requirements and see which ones are mandatory or optional, and ensure that the mandatory ones are implemented.
- Make sure to check that the XML being created is both well-formed and valid.
- Remember that an API that works for all possible database setups is not possible. The creators of the specifications and standards will only have information about their specific part, the specification, and/or the standards and not about the originating systems.
- The road map for implementation or update will guide the business decisions needed to be made.
- Make sure that an implementation and/or update follows the version of the specification and/or standard you say you are adhering to. Mixing versions without checking the result will not make you adhere.
- When the requirements from specifications and/or standards are common for the systems, an export from one system can be imported into another system more easily, thus giving the benefit of making the export/import of data easier.
- Exporting in a common way, following specifications and/or standards, ensures that the data is ready for digital preservation in the long term.

When you are a user of XML transferred from a system:

- The benefit of getting data that follows a specification and/or standard means that the data's requirements are known and explained, making the data easy to reuse or handle both now and in the future.

When you provide tools or applications with XML capabilities:

- If you implement specifications and/or standards, go through the requirements, determining which are mandatory or optional, and making sure that the mandatory ones are implemented.
- Make sure that you check that XML that is created is both well-formed and valid.
- If it is possible to add warnings that the XML has not been checked for well-formedness or validity when an export is made, please do so and make sure the checks can be performed.
- When tools and applications come ready with specifications and/or standards implemented for data export/import, the benefit will be that the people responsible for the tools/applications have full control of the implementation.

- Using specifications and/or standards ensures that the data follows a common understanding and at the same time it is ready for long-term digital preservation.

Transparent Invisible XML

Nico Verwer

<nverwer@rakensi.com>

Abstract

Invisible XML (ixml) is a language for specifying grammars that can be used to parse plain text and turn it into structured XML. This works when the input is just plain text with an implicit structure, but not when the input is already structured XML, to which we want to add more structure.

In this paper, we present a modification to the Markup Blitz ixml parser, which makes it possible to augment existing XML markup with XML elements from the parser output. In BaseX, this modification is available as a slightly altered version of the proposed `fn:invisible-xml` XQuery 4.0 function. When applying the parsing function to an XML fragment, the parser only sees the text content of that fragment, hence the name “transparent” invisible XML.

We also propose a modification to ixml itself, allowing existing XML elements to be recognized as non-terminals. This makes it possible to use “pre-parsed” input that is generated by another ixml parser, or by another method like named entity recognition. An ixml grammar can refer to these “pre-parsed” elements, and make them part of larger recognized structures.

1. Introduction

The motivation for writing this paper, and developing the software described in it, was the “Link eXtractor” (LX) project. [1] This project analyzes Dutch and European legal texts, such as laws, treaties, and other regulatory texts, as well as case law, court rulings, judgments and advisories. It detects references to other legal texts, resolves the targets of these references, and inserts links in the source document. The references that the LX detects consist of plain text without any links. Therefore, the LX uses parsing expression grammars and generates parsers to recognize the references. The parsers, XSL transformations and other components are tied together as pipelines in Apache Cocoon, under Java 8.

Although the LX application has been running successfully for almost 10 years, it is in need of a complete overhaul. Many of its software components have become obsolete, and we would like to use more modern technologies, such as XQuery, XSLT 3, Invisible XML, and a modern Java version. This required the development of the technologies described in this paper, mainly Transparent Invisible XML.

Invisible XML, or ixml is a language for specifying grammars that can be used to parse plain text and turn it into structured XML. [2] It was designed by Steven Pemberton, who gave several presentations and tutorials on ixml. [3] [4] Version 1.0 of the ixml specification was published in June 2022. There are several implementations of ixml, and it has an active community.

An XPath function

```
fn:invisible-xml($grammar as item()?, $options as map(*)?)
  as function(xs:string) as item()
```

has been proposed as part of the XPath and XQuery Functions and Operators 4.0 draft. [5] This function takes a grammar and options, and returns a parser in the form of a function. As the signature indicates, the returned parser function takes plain text (`xs:string`) and returns either an XML representation of the parsed input string, or an error.

Sometimes, the input that we want to parse is not just plain text, but it is text within an XML document. Parsing enriches this document by adding additional elements around the recognized text. There may be markup (XML elements) inside the recognized text, which must be preserved during parsing. The parser that is generated by `fn:invisible-xml` only handles plain text, so what can we do with the additional markup?

It is easy to remove the XML markup, but not so easy to put it back after parsing. Also, the XML elements in the parser's output must be added to the existing XML elements. This cannot be done arbitrarily; The result must be well-formed (and possibly valid) XML. Solving both problems makes the XML elements in the input transparent to the parser, hence the name “transparent invisible XML”, or `tixml`.

As an example, consider this fragment from a report of the Parliamentary Assembly of the Council of Europe (<https://pace.coe.int/en/files/30065/xml>):

```
<Para>
...
immunity under Article 40.<Emphasis Role="italics">a</Emphasis>
of the Statute of the Council of Europe and
the General Agreement on Privileges and Immunities of the Council of
Europe
and its Additional Protocol
...
</Para>
```

A parser for references to legal documents should recognize “Article 40.a of the Statute of the Council of Europe” and make it possible to put a link to this article into the document.

A first step could be to make an ixml grammar with a rule for `treaty`, which recognizes all the Council of Europe's treaties. Then a rule like the following would parse the reference.

```
reference : ( part-type , s , part-id , s , of , s )? , treaty .
```

The `s` and `of` nonterminals are whitespace and connecting words, and will not appear in the output.

However, there is an `<Emphasis>` element around the “a”. If we were to use the serialized XML document as input to the parser, we would have to allow for XML elements like `<Emphasis>`. There may be many such elements that possibly occur at many places, depending on the schema of the document. Incorporating these (serialized) elements in the grammar is a lot of work, and obfuscates the grammar to the point that its original intention is lost. Furthermore, serializing and re-parsing becomes very complicated.

Another option is to throw away the `<Emphasis>` and other elements, and parse the text content of the document. The `<Emphasis>` and other elements must be put back into the document after parsing, since we want to enrich the document with links, not change its original structure. If we can do this, the `<Emphasis>` and other elements element transparent to the parser, which is the purpose of `tixml`.

For the document fragment given above, the `tixml` parser output would include

```
<reference>
  <part-type>Article</part-type>
  <part-id>40.<Emphasis Role="italics">a</Emphasis></part-id>
  of the <treaty>Statute of the Council of Europe</treaty>
</reference>
```

The existing markup is still present, and the elements resulting from parsing have been added.

Accompanying this paper is a modified version of an existing ixml parser generator, Markup Blitz, that provides a way to transparently parse XML instead of just plain text. [7] This results in a slightly modified version of the proposed `fn:invisible-xml` XQuery 4.0 function.

Instead of ignoring XML elements in the input text when parsing, one might want to use them, as part of the grammar rules. Some XML elements in the input to the parser may be the result of “pre-parsing”, and should be taken into account by the ixml parser. An example is named entity recognition (NER), which recognizes and marks fixed text fragments from a (often very large) list.

In the last section of this paper, we propose an extension to ixml, which makes it possible to refer to XML elements in the input document. These XML elements will be treated as non-terminals, without any actual parsing of their text content.

They can be part of a rule defining a larger non-terminal, so the parser behaves like it has recognized these “pre-parsed” non-terminals.

2. Separating markup from content

Consider the XML fragment

```
<r>AB<s><m>CDEF</m><n>GHIJKL</n></s>M<t>N<p>OPQR</p>STU<q>VWX</q>YZ</t></r>
```

This represents a hierarchical structure that may be drawn like a tree:

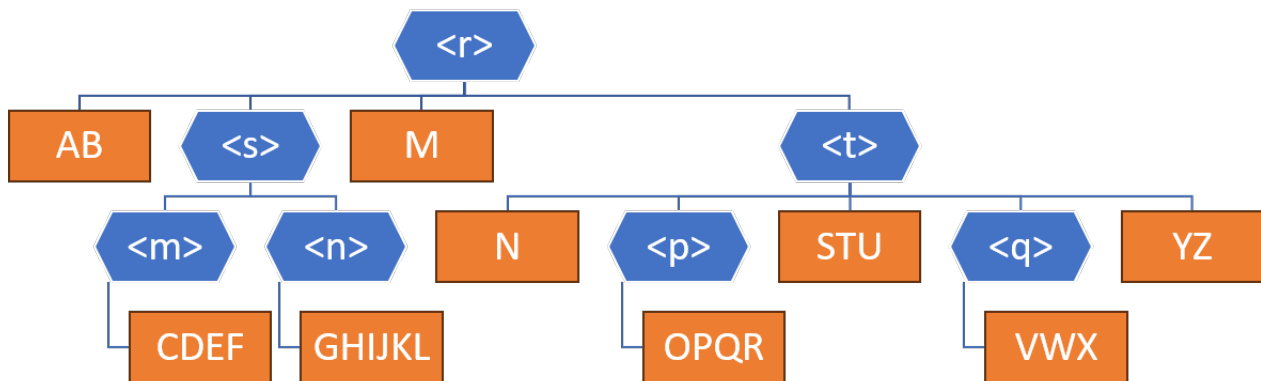


Figure 1. XML DOM tree

This tree is a way to visualize the DOM, which is one of the forms in which XML documents are represented in XML processing software. Another form is SAX, where the nodes in the tree are streamed in depth-first left-to-right order.

Neither DOM, nor SAX is convenient for a parser that takes plain text as its input, like the parser produced by `fn:invisible-xml`, which expects a `xs:string` as its input. We could take the text content of the XML fragment using the `string()` function, and give that to the parser. Of course this will throw away the original structure, which should be preserved, as stated in the introduction.

As an example, consider the previous XML fragment, and an `ixml` grammar that has a rule

```
k : "NOPQR" .
```

An `ixml` parser will not recognize `k` when given the serialized XML as input, because of the `<p>` element around "OPQR". When given only the text in the input (`//r/string()`), the parser will recognize `k`, but there is no simple way to get the markup within the `<r>` element back.

We could find a way to feed the `text()` nodes into the parser, and hope to restore the markup based on the position of the `text()` nodes (using parent and sibling relations) in the XML document. However, the `ixml` parser does require one `xs:string` as its input, and concatenating the characters inside `text()` nodes

into one string loses the structure. Registering additional information about the start and end of each `text()` node leads to the solution that is described next.

In order to make the structure of an XML document “transparent” to the `ixml` parser, a representation of the document that separates the markup from the text content is needed. For the implementation of `tixml` we use SMAX (Separated Markup API for XML). [8] [9] SMAX was designed in order to be able to process the text content of an XML document in a linear way, without intervening markup. SMAX retains the structure and markup of the document and provides methods to modify the structure by inserting elements in a way that preserves well-formedness.

A SMAX document keeps the structure (XML elements) separate from the text of an XML document, by putting the text in a single string or character stream, and storing the start and end character position of every element. A visualization similar to the DOM tree illustrates this idea.

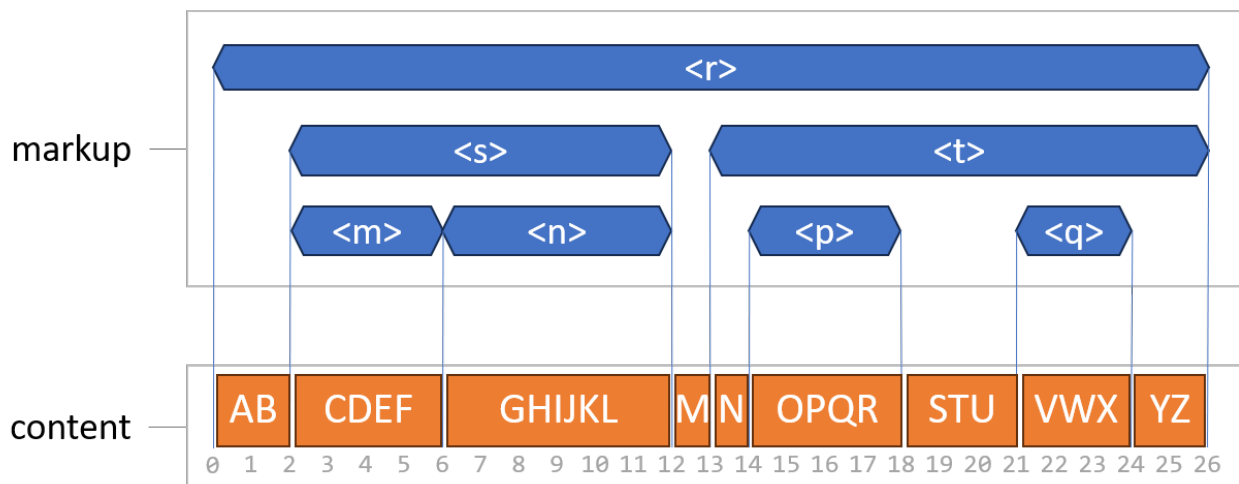


Figure 2. XML representation in SMAX

The thin blue lines represent the start and end character positions that are kept with each element.

3. Combining SMAX and `ixml`

The content string from the SMAX representation of an XML fragment can be given to an `ixml` parser. The parser then produces new markup in the form of XML elements, and SMAX merges these generated elements with the original markup. Thus, the original XML markup becomes transparent to the parser, and we have `tixml`.

The process of parsing and inserting new elements is illustrated in the following diagram. The `ixml` parser is used to recognize “NOPQR” as the non-terminal “k”, and to generate the `<k>` element. When the `<k>` element is inserted, the text

"N" and the element `<p>` will become its children. The insertion is done with the SMAX function `insertMarkup`, and SMAX uses one of the available balancing strategies to make sure that the document remains well-formed.

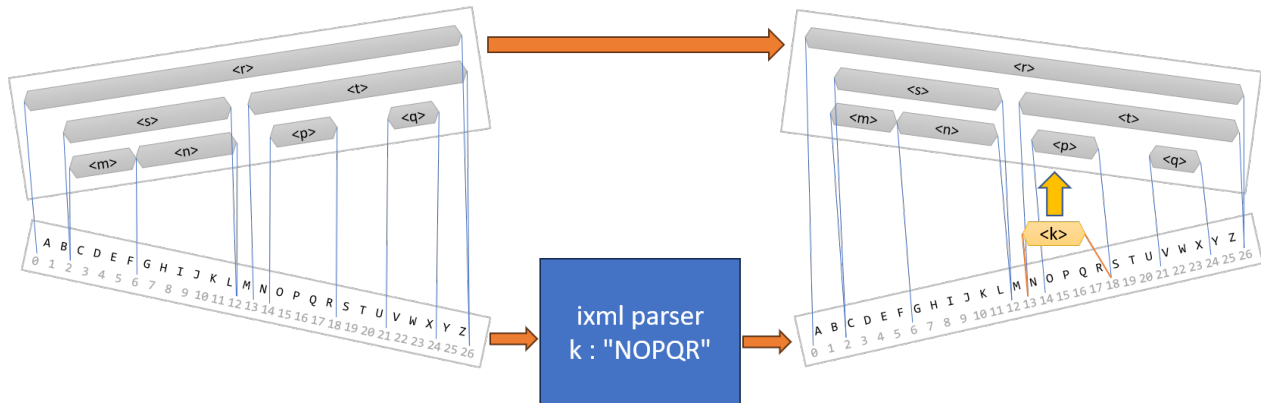


Figure 3. Using SMAX to insert elements from the ixml parser

After parsing, the XML fragment would look like

```
<r>AB<s><m>CDEF</m><n>GHIJKL</n></s>M<t><k>N<p>OPQR</p></k>STU<q>VWX</q>YZ</t></r>
```

Notice how the end tag `</k>` comes after `</p>`.

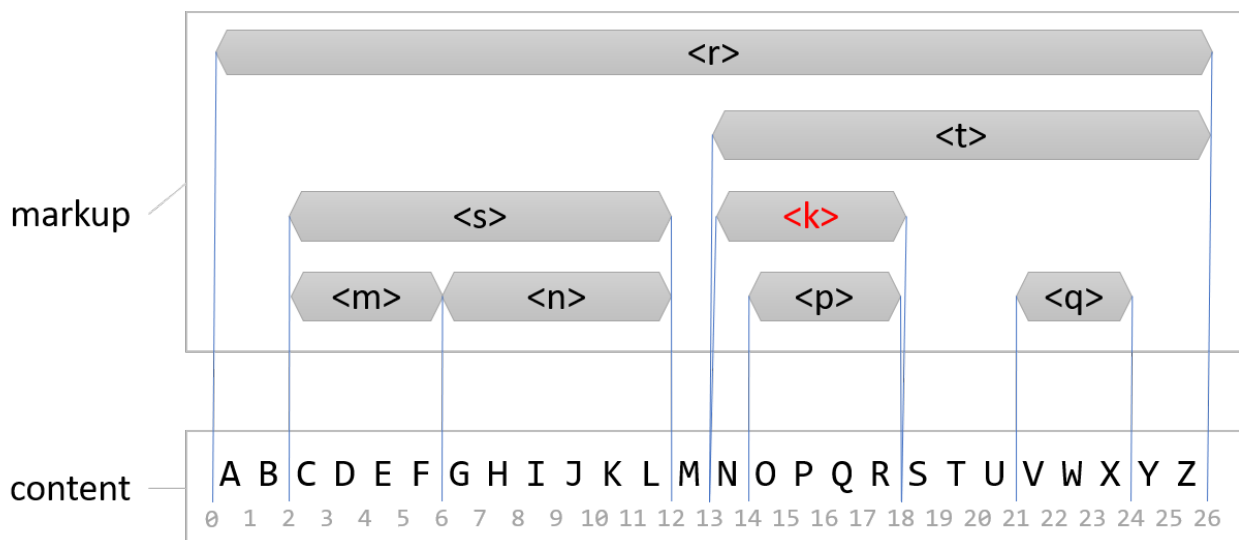


Figure 4. SMAX document with insertion

The Markup Blitz parser has been modified to implement `tixml` as a new XQuery function:

```
fn:transparent-invisible-xml($grammar as item()?, $options as map(*))
  as function(element()) as item()
```

Internally, this function uses SMAX and ixml. At the time of writing, the tixml parser accepts an `element()` as input, but we plan to extend this so that `xs:string` is also accepted. The new function is made available in BaseX. [6] An eXist version may be available at the time this paper is published, or shortly thereafter. The code (still under construction) is available on GitHub. [10]

The tixml software has also added a way to tell the parser to look for text *fragments* that the parser recognizes, instead of requiring the whole input text to match. This comes in the form of some new options, as shown in the examples below. These options will change in the near future, in order to align them with newer versions of Markup Blitz, which Gunther Rademacher has kindly provided to allow the new fragment-matching feature.

As an example, consider again the document fragment used in the introduction. In our modified version of BaseX, this can be parsed with the following XQuery program.

```
let $input :=
<Para>
  ... immunity under Article 40.<Emphasis Role="italics">a</Emphasis>
  of the Statute of the Council of Europe and
  the General Agreement on Privileges and Immunities of the Council of
  Europe
  and its Additional Protocol ...
</Para>

let $grammar := `[
  reference : ( part-type , s , part-id , s , of , s )? , treaty .
  part-type: ["Aa"] , "rticle" .
  part-id: ["1"-"9"] , ["0"-"9"]* , ( "."? , ["a"-"z"] )? .
  of : ",",
      ; "of" , (s , "the")?
  .
  -s : [Zs; #9; #a; #d]+ .
  treaty: "Statute of the Council of Europe" .
]`

let $parser-options := map{ 'multiple-matches': true(), 'skip-unmatched-
words': true(), 'longest-match': true() }

let $parser := transparent-invisible-xml($grammar, $parser-options)

return $parser($input)
```

The output is:

```
<Para>
  ... immunity under <reference><part-type>Article</part-type> <part-
```

```
id>40.<Emphasis Role="italics">a</Emphasis></part-id>
  <of>of the</of> <treaty>Statute of the Council of Europe</treaty></
reference> and
  the General Agreement on Privileges and Immunities of the Council of
Europe
  and its Additional Protocol ...
</Para>
```

We can add all treaties of the Council of Europe, changing the grammar rule for treaty:

```
treaty : "Second Additional Protocol to the Convention on Cybercrime on
enhanced co-operation and disclosure of electronic evidence"
      ; "Protocol amending the Convention for the Protection of
Individuals with regard to Automatic Processing of Personal Data"
      ; "Protocol amending the Additional Protocol to the Convention on
the Transfer of Sentenced Persons"
...
      ; "Convention for the Protection of Human Rights and Fundamental
Freedoms"
      ; "General Agreement on Privileges and Immunities of the Council of
Europe"
      ; "Statute of the Council of Europe"
.
```

The whole list contains 226 items, and is not fully shown.

When running the modified XQuery, the output is:

```
<Para>
  ... immunity under <reference><part-type>Article</part-type> <part-
id>40.<Emphasis Role="italics">a</Emphasis></part-id>
  <of>of the</of> <treaty>Statute of the Council of Europe</treaty></
reference> and
  the <reference><treaty>General Agreement on Privileges and Immunities
of the Council of Europe</treaty></reference>
  and its Additional Protocol ...
</Para>
```

We see that another treaty has also been recognized.

The new `fn:transparent-invisible-xml` function has made the markup in the document transparent to the parser, but retains it in the result, inserting new markup as instructed by the parser. No fundamental changes to `ixml` have been made, only some new options were introduced to allow the recognition of text fragments instead of parsing the whole text content of the input document.

4. Named entity recognition

In the previous example, a rule for 226 treaties was added to the grammar, which makes the grammar quite long and tedious to read. However, this is small in comparison with a grammar for the more than 276000 titles in Dutch and European legislation and case law. This is indeed a problem, as we will see shortly.

Another problem is, that parsing a text fragment as a `treaty` does not indicate *which* treaty this is. When a reference is detected, we would like to add a link to the target document, which requires some identification of the target, usually a standardized identifier.

To solve both problems, we could make a huge grammar with one non-terminal and an accompanying rule for each legal document. For COE treaties, this might look like

```
treaty-CETS-224 : "Second Additional Protocol to the Convention on
Cybercrime on enhanced co-operation and disclosure of electronic
evidence" .
treaty-CETS-223 : "Protocol amending the Convention for the Protection
of Individuals with regard to Automatic Processing of Personal Data" .
...
treaty-ETS-002 : "General Agreement on Privileges and Immunities of
the Council of Europe" .
treaty-ETS-001 : "Statute of the Council of Europe" .

treaty : treaty-ETS-001 ; treaty-ETS-002 ;
...
        ; treaty-CETS-223 ; treaty-CETS-224 .
```

Parsing with this expanded grammar yields the following output.

```
<Para>
... immunity under <reference><part-type>Article</part-type> <part-
id>40.<Emphasis Role="italics">a</Emphasis></part-id>
<of>of the</of> <treaty><treaty-ETS-001>Statute of the Council of
Europe</treaty-ETS-001></treaty></reference> and
the <reference><treaty><treaty-ETS-002>General Agreement on Privileges
and Immunities of the Council of Europe</treaty-ETS-002></treaty></
reference>
and its Additional Protocol ...
</Para>
```

The next step in the processing pipeline would be to split the type (`treaty`) from the identifier (ETS-001 etcetera), and put it as an attribute in the `<treaty>` element.

Unfortunately, for hundreds of thousands of legal documents, the compiled grammar will become too large to fit in main memory. Even with 10000 entries,

compiling the grammar resulted in an “Out of Main Memory” error, with 8GB heap space.

In this case, a `ixml` parser seems to be the wrong tool for the job. Fortunately, we can use named entity recognition (NER) to recognize large numbers of fixed text strings.

One of the functions provided by [10] is

```
named-entity-recognition($grammar as item(), $options as map(*)?)
  as function(node()) as node()
```

This function is very similar to the `fn:transparent-invisible-xml` function, because it also performs recognition of text fragments, which are defined by a grammar. However, in this case the grammar is very different. It is a text file with lines containing an identifier, followed by `<-`, followed by a tab-separated list of text strings that correspond to the identifier.

For the COE treaties, a named entity recognizer is defined by the following XQuery fragment.

```
let $ner-grammar := ``[
  CETS-224 <- Second Additional Protocol to the Convention on Cybercrime
on enhanced co-operation and disclosure of electronic evidence
  CETS-223 <- Protocol amending the Convention for the Protection of
Individuals with regard to Automatic Processing of Personal Data
  ...
  ETS-002 <- General Agreement on Privileges and Immunities of the
Council of Europe
  ETS-001 <- Statute of the Council of Europe
]``

let $ner-options := map
{ 'case-insensitive-min-length': 4, 'fuzzy-min-length': 4
, 'word-chars': '', 'no-word-before': '-'
}

let $ner := smax:named-entity-recognition($ner-grammar, <treaty id=""/>,
$ner-options)

return $ner($input)
```

Named entity recognition is not the main topic of this paper, so we will not discuss every detail. In this case, every named entity has only one text string. The second parameter in `smax:named-entity-recognition($ner-grammar, <ner entity-id=""/>, $ner-options)` is a template for marking recognized named entities.

The result of running this XQuery program is


```
<Para>
  ... immunity under Article 40.<Emphasis Role="italics">a</Emphasis>
  of the <treaty id="ETS-001">Statute of the Council of Europe</treaty>
  and
  the <treaty id="ETS-002">General Agreement on Privileges and
  Immunities of the Council of Europe</treaty>
  and its Additional Protocol ...
</Para>
```

5. Combining NER and tixml

Now we can recognize large numbers of named entities, but we immediately run into another problem. Our ixml grammar has a rule

```
reference : ( part-type , s , part-id , s , of , s )? , treaty .
```

What should the grammar rule for `treaty` be? Treaties are marked by `<treaty>` elements, but these will be transparent to the ixml parser.

In the previous version of the Link eXtractor software, this problem was solved by serializing the XML input, and including grammar rules for `<treaty>` elements, so that these would be recognized. This was a very inelegant solution. Serialization and subsequent re-parsing of the XML introduced unnecessary overhead. The grammars had to include rules for XML parsing, which is distracting from what the grammars should express.

A slightly more elegant solution is to put extra markers in the text, signifying the transparent presence of a `<treaty>` element. Typically, these markers are short text strings that should not occur anywhere in the original text. Short markers have a high chance of running into Wethern's Law of Suspended Judgment, but for this example we will use `[[[ and ]]]`. The grammar rule for `treaty` now becomes

```
-treaty: "[[[" , ~[]+ , "]]]" .
```

Note that this requires the 'shortest-match' option to be true in `transparent-invisible-xml($ixml-grammar, $ixml-options)`, otherwise the `~[]+` in `treaty` will match from the first `[[[ until the last ]]]`.

Now we can put everything together. A simple XSLT transforms the output of the NER into

```
<Para>
  ... immunity under Article 40.<Emphasis Role="italics">a</Emphasis>
  of the <treaty id="ETS-001">[[[Statute of the Council of Europe]]]</
treaty> and
  the <treaty id="ETS-002">[[[General Agreement on Privileges and
  Immunities of the Council of Europe]]]</treaty>
  and its Additional Protocol ...
</Para>
```

This can be parsed with a tixml parser using the grammar with the modified rule for `treaty`. After parsing, the `[[[ ]]]` markers can be removed by another XSLT. If we encapsulate these XSLTs in two locally defined functions called `$mark-treaties` and `$unmark-treaties`, the process of recognizing references can be coded as

```
return $input => $ner() => $mark-treaties() => $parser() => $unmark-  
treaties()
```

Inserting and removing special markers works, but it introduces accidental complexity. In the following section, we propose a more elegant solution.

6. Pre-parsed non-terminals

Instead of making XML elements in the input document transparent, one might actually want to make them part of the grammar. This is the case when an application uses different techniques to recognize implicit structure in the text of a document. An example is named entity recognition (NER), as discussed in the previous section.

The result of NER is a document with `<treaty>` elements:

```
<Para>  
  ... immunity under Article 40.<Emphasis Role="italics">a</Emphasis>  
  of the <treaty id="ETS-001">Statute of the Council of Europe</treaty>  
  and  
  the <treaty id="ETS-002">General Agreement on Privileges and  
  Immunities of the Council of Europe</treaty>  
  and its Additional Protocol ...  
</Para>
```

The tixml parser ignores the `<Emphasis>` and `<treaty>` elements by making them transparent. In order to use the `<treaty>` elements as treaties in the grammar, they were made opaque by surrounding their text contents with special markers in the previous section. It would be nice if we could skip the mark and unmark steps, and use the `<treaty>` elements in the grammar itself.

In order to make this possible, we propose an extension to (t)ixml that recognizes specific XML elements as “pre-parsed” non-terminals. Normally, the tixml parser inserts a `<treaty>` element into the document when it has matched a text fragment that matches the rule for `treaty`. With pre-parsed non-terminals, the parser will handle the text within existing `<treaty>` elements as if this text has already matched the rule for `treaty`. The pre-parsed non-terminal `treaty` corresponds to the existing XML element `<treaty>`. This works regardless of the content (text and markup) inside the `<treaty>` element, which the parser will ignore.

Extending the parser in this way requires a new notation for pre-parsed non-terminals. An obvious choice is to use angle brackets around the name of the non-terminal / XML element. As an example, the rule for `reference` becomes

```
reference : ( part-type , s , part-id , s , of , s )? , <treaty> .
```

When the parser is about to match a `<treaty>`, it should check if there is a `<treaty>` element beginning at the current position in the input. If there is, the pre-parsed non-terminal `<treaty>` is recognized, and the input position is forwarded to the position where the `<treaty>` element ends.

The output of the tixml parser then includes

```
... immunity under
<reference>
  <part-type>Article</part-type>
  <part-id>40.<Emphasis Role="italics">a</Emphasis></part-id>
  of the <treaty id="ETS-001">Statute of the Council of Europe</treaty>
</reference>
and the
<reference>
  <treaty id="ETS-002">General Agreement on Privileges and Immunities of
the Council of Europe</treaty>
</reference>
and its Additional Protocol ...
```

When the tixml parser generator has been modified to recognize pre-parsed non-terminals, the pipeline for recognizing references becomes:

```
return $input => $ner() => $parser()
```

Thus, we have removed the accidental complexity that was caused by marking and unmarking.

The implementation of pre-parsed non-terminals using SMAX and ixml is quite complicated, because it requires modifications in the core of the parser code. Schematically, this modification is illustrated in the following diagram.

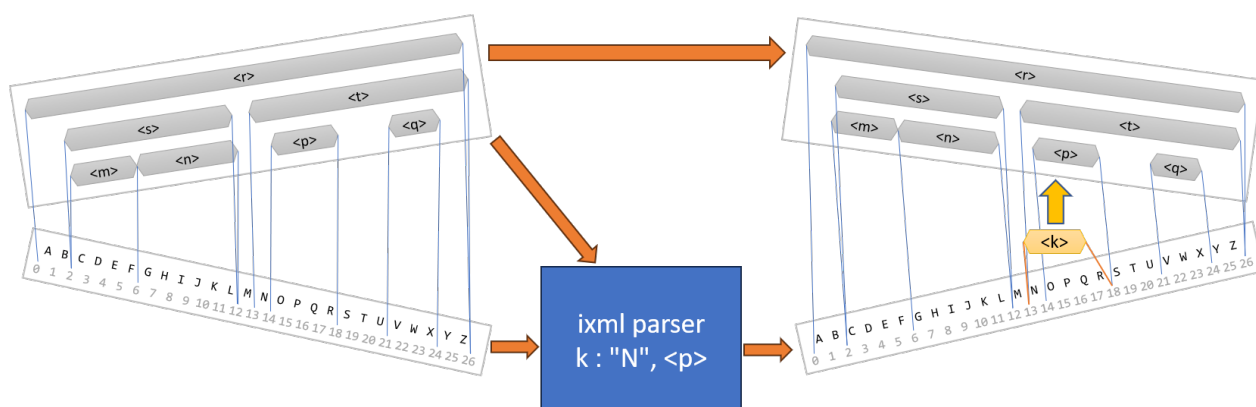


Figure 5. Pre-parsed non-terminals in ixml + SMAX

Both the SMAX markup and text content are given to the parser. The parser uses the markup for recognizing pre-parsed non-terminals.

7. Conclusion

The original Link eXtractor (LX) software consists of several NER, parsing, and XSLT transformations, some of which can be enabled or disabled by parameters and configuration. Each parsing transformation is surrounded by serializing and parsing. Using text markers would be an improvement, but pre-parsed non-terminals will allow us to greatly simplify the whole process. This is one of the aims of the next stage of the LX project.

We believe that the techniques presented in this paper will also be useful for other use cases. Enriching both plain text and XML documents by adding markup is useful in different contexts than the LX.

Transparent ixml as described in the first few sections has been implemented, and has been published on GitHub. [10] Work on implementing pre-parsed non-terminals has not commenced yet at the time of writing. When we have a working version, the software will become available in the same repository. Before considering using the software, note that tixml and pre-parsed non-terminals are not part of the ixml standard. Maybe they will be at a later stage, probably in a way that differs from the upcoming implementation.

References

- [1] M. van Opijnen, N. Verwer and J. Meijer: *Beyond the Experiment: The Extendable Legal Link Extractor*. 2015 https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2626521
- [2] Steven Pemberton, et. al.: *Invisible XML*. Invisible XML <https://invisiblexml.org/>
- [3] Steven Pemberton: *Invisible XML (ixml) Tutorial*. 2021 <https://homepages.cwi.nl/~steven/ixml/tutorial/>
- [4] Steven Pemberton: *Advanced ixml Hands On*. 2022 <https://homepages.cwi.nl/~steven/Talks/2022/11-07-ixml-advanced/>
- [5] Michael Kay (editor): *XPath and XQuery Functions and Operators 4.0*. W3C Editor's Draft 21 May 2024 <https://qt4cg.org/specifications/xpath-functions-40/Overview.html#ixml-functions>
- [6] Christian Grün et al.: *BaseX, The XML Framework, Lightweight and High-Performance Data Processing*. BaseX, The XML Framework, Lightweight and High-Performance Data Processing <https://basex.org/>
- [7] Gunther Rademacher: *Markup Blitz*. Markup Blitz <https://github.com/GuntherRademacher/markup-blitz>

- [8] Nico Verwer: *Plain text processing in structured documents*. 2020 <https://declarative.amsterdam/article?doi=da.2020.verwer.plain-text-processing>
- [9] Nico Verwer: *SMAX (Separated Markup API for XML)*. SMAX (Separated Markup API for XML) <https://github.com/nverwer/SMAX>
- [10] Nico Verwer: *BaseX-SMAX-functions*. BaseX-SMAX-functions <https://github.com/structured-data-friends/BaseX-SMAX-functions>

Round-tripping Invisible XML

Steven Pemberton

CWI, Amsterdam

<steven.pemberton@cwi.nl>

Abstract

Invisible XML takes textual documents where the structure is implicit and produces documents with the structure made explicit. This paper addresses the question of the extent to which it is possible to recreate the original textual document from its structured version, how it can be done, and what if any the ramifications are for ixml.

Keywords: ixml, structured data, parsing, XML, round-tripping, grammar transformation, serialisation

1. Contents

- Introduction
- Round-tripping
- The Problem Space
- Grammar Transformation
- Dealing with Attributes
- Strict and Permissive Grammars
- Syntactic Equivalences
- Loss of Information
- Ambiguity
- Inserted Layout
- Future Work
 - Similarities between Serialisation and Transformation
 - Possible Additions to ixml
- Conclusion
- References

2. Introduction

Invisible XML [1] is a language and process that takes linear textual input, recognises the implicit structure in the input, and converts it to an equivalent structured XML output. It does this by parsing the input using a grammar describing the format of the input document, and serialising the resultant parse-tree as XML, using extra information in the grammar to drive the serialisation.

If that were all it did, then round-tripping the XML back to text would be trivial: it would simply be a case of concatenating the text nodes of the XML, and you'd be done.

However, there are issues with regards to ixml serialisation:

- input characters may be deleted from the parse-tree on serialisation;
- extra characters that weren't in the input may be inserted;
- some parse-tree nodes may be serialised as attributes rather than elements, causing a reordering of the input text, since attributes appear before element content.

As hinted at in earlier papers on ixml [2], [3], round-tripping could be achieved by having a special-purpose general parser which attempts to recreate a parse-tree that could have produced the serialisation, and then concatenating the resulting text nodes.

This paper takes a different approach: by transforming the input grammar into a grammar that represents all possible serialisations of the input grammar, it can use the same parser as used by ixml, with some small additions, to parse the serialisation back into a parse-tree that would have produced that serialisation.

This raises a number of technical issues similar to the normal ixml process, in particular what to do with ambiguity, where a serialisation could have been produced by more than one input.

3. Round-tripping

The term round-tripping is normally considered to be the process of recreating the original document that produced the output you have [4]. For several reasons discussed below, except in limited cases, it is not possible to recreate a character-perfect original document from the ixml output. This is partly under the grammar author's control, which is to say, it is possible to write an ixml grammar that can be perfectly round-tripped. On the other hand, in the other direction round-tripping is always possible: it is always possible to construct from the ixml *output* an input that would create exactly the same output again.

So for the purposes of this paper, our definition of round-tripping is not "create an identical input to the one that created this output", but:

Create an input that would produce the identical output.

As an example, an ixml grammar that reads a program in a programming language, deleting comments and spurious whitespace, on round-tripping would of course no longer contain the comments and extra whitespace: they were deleted and didn't appear in the structured version. However the round-tripping would still produce effectively the same program.

4. The Problem Space

Let's take the simplest case. Here is a simple grammar for dates:

```
date: day, "/", month, "/", year.  
day: d, d.  
month: d, d.  
year: d, d, d, d.  
-d: ["0"-"9"].
```

Given as input

```
30/06/2024
```

this will produce as output:

```
<date>  
  <day>30</day>/  
  <month>06</month>/  
  <year>2024</year>  
</date>
```

As you can see, all characters in the input are preserved in the output, in order. The only difference is that tags have been placed around the characters to indicate the structure.

So to round-trip this, all that is necessary is to concatenate the text nodes of the XML, to give the input that created the output:

```
30/06/2024
```

This is what is meant above by "it is possible to write an ixml grammar that can be perfectly round-tripped": if the grammar deletes no characters, and only uses elements, round-tripping is trivial.

The fact that some tags have been suppressed in the output (namely the `<d>` elements, due to the `-` before the rule for `d`) has no effect, since we are only interested in characters.

However, ixml has facilities to control the serialisation format. For instance, the slash characters `/` in the input are there only as punctuation to separate the different parts, but play no role in the output, and so can be deleted from the serialisation:

```
date: day, -"/", month, -"/", year.
```

giving

```
<date>  
  <day>30</day>  
  <month>06</month>  
  <year>2024</year>  
</date>
```

Now we can no longer just concatenate the text nodes to get back to the input.

Similarly, characters can be inserted into the serialisation. Suppose the input date format only had two digits for the year, but the output serialisation required four. You can write this:

```
date: day, -"/", month, -"/", year.  
day: d, d.  
month: d, d.  
year: +"20", d, d.  
-d: ["0"-"9"].
```

which for the input 30/06/24 would give:

```
<date>  
  <day>30</day>  
  <month>06</month>  
  <year>2024</year>  
</date>
```

Finally, elements can be serialised as attributes:

```
date: day, -"/", month, -"/", year.  
day: d, d.  
month: d, d.  
@year: +"20", d, d.  
-d: ["0"-"9"].
```

which gives

```
<date year='2024'>  
  <day>30</day>  
  <month>06</month>  
</date>
```

This reorders the input, since attributes are serialised before elements.

These issues need to be addressed in order to enable round-tripping of content in the general case.

5. Grammar Transformation

As already pointed out, earlier papers have suggested using a special-purpose parser to deal with round-tripping, but this paper proposes a different approach: transforming the input ixml grammar into a different ixml grammar that recognises the output format, and then using the same parser that ixml already uses.

As an example, for the first grammar above

```
date: day, "/", month, "/", year.  
day: d, d.  
month: d, d.
```

```
year: d, d, d, d.  
-d: ["0"-"9"].
```

a grammar can be generated that recognises the serialisation:

```
date: -"<date>", day, "/", month, "/", year, -"</date>".  
-day: -"<day>", d, d, -"</day>".  
-month: -"<month>", d, d, -"</month>".  
-year: -"<year>", d, d, d, d, -"</year>".  
-d: ["0"-"9"].
```

Using this grammar and the regular ixml parser to parse the output XML, gives:

```
<date>30/06/2024</date>
```

(We have glossed over the issue of extra inserted whitespace for pretty printing, which we will discuss later; we will also detail later reasons why the root element remains).

This leads immediately to the second example. where everything is the same except the rule:

```
date: day, -"/", month, -"/", year.
```

with a transformed equivalent rule:

```
date: -"<date>", day, +"/", month, +"/", year, -"</date>".
```

which similarly generates the same output

```
<date>30/06/2024</date>
```

And finally (for now), the version with added output characters

```
date: day, -"/", month, -"/", year.  
day: d, d.  
month: d, d.  
year: +"20", d, d.  
-d: ["0"-"9"].
```

with transformation:

```
date: -"<date>", day, +"/", month, +"/", year, -"</date>".  
-day: -"<day>", d, d, -"</day>".  
-month: -"<month>", d, d, -"</month>".  
-year: -"<year>", -"20", d, d, -"</year>".  
-d: ["0"-"9"].
```

giving as output:

```
<date>30/06/24</date>
```

So from these three examples we can see some patterns for the transformations emerging:

- non-hidden rules except for the root become hidden rules in the transformation, and start and end with its tags, both marked as deleted;
- hidden rules remain hidden, and no tags are recognised;
- in alternatives:
 - nonterminals remain the same;
 - regular terminals stay the same;
 - deleted terminals become inserted terminals;
 - inserted terminals become deleted terminals.

6. Dealing with Attributes

The main problem left arises with attributes, which turn up at a different position in the serialisation. For instance the grammar

```
date: day, -"/", month, -"/", year.  
day: d, d.  
month: d, d.  
@year: +"20", d, d.  
-d: ["0"-"9"].
```

which generates the output

```
<date year='2024'>  
  <day>30</day>  
  <month>06</month>  
</date>
```

which can be *recognised* with

```
date: -"<date", year, -">", day, +"/", month, +"/", -"</date>".  
-day: -"<day>", d, d, -"</day>".  
-month: -"<month>", d, d, -"</month>".  
-year: -" year='", -"20", d, d, -"'.  
-d: ["0"-"9"].
```

but would produce on output

```
<date>2430/06/</date>
```

In other words, while it can recognise the attribute, it turns up in the wrong place for the round-tripping, and so a mechanism is needed for putting it back in the right place.

In normal ixml processing, when serialising to XML, the ixml grammar identifies the place in the input where the eventual attribute is, and implicitly identifies the place where the attribute must be serialised on output, namely the nearest ancestor element that isn't hidden. This is done with a two-pass algorithm: when serialising an element, children of the element, and of hidden children elements

are traversed first to find rules marked as attributes, which are then serialised before the element children.

To reverse this process, we must recognise the attribute at its implicit position in the input, and *explicitly* identify in the grammar the place where it needs to be serialised; this will always be later in the output than where the attribute was found. We can do this by defining two extra *marks*: one to indicate that the attribute as parsed should not be serialised at that position, and another to indicate the position where it should be serialised.

This is comparable to how terminals are inserted and deleted on serialisation: `- "abc"` means *"parse the string but serialise nothing"*, while `+ "abc"` means *"parse nothing, but serialise the string"*. Unfortunately, `-` has already been assigned a different role for nonterminals on serialisation, but `+` is free to use with the same meaning.

So using the mark `*` to mean *"parse the input, but serialise nothing"*, and `+` to mean *"parse nothing but serialise the node of this name from earlier in the tree marked with a `*`"*, we can specify:

```
date: -"<date", *year, -">", day, +"/", month, +"/", +year, -"</date>".
-day: -"<day>", d, d, -"</day>".
-month: -"<month>", d, d, -"</month>".
-year: -" year='", -"20", d, d -"'".
-d: ["0"-"9"].
```

This now correctly produces

```
<date>30/06/24</date>
```

Note that these are not changes to the ixml language itself, but only internal additions: the transformations are done on internal representations of the grammar, and not on external representations (but see the section on future work).

7. Strict and Permissive Grammars

There are two (non-normative) terms used to describe two major applications of ixml grammars: *strict* and *permissive*.

If it is certain that the input being processed with ixml is correct, the grammar can be laxer in what it accepts. The date grammar is a good example of this. If dates are only being recognised, and not checked for correctness, then `"d, d"` is a perfectly good pattern for recognising the day number. Even though this would also recognise 32 or 99, since such dates never occur, you don't have to worry about them. Such grammars are referred to as *permissive*.

On the other hand if the input might not be correct, then the grammar needs to be stricter. With dates again, you must then only accept single digits in the range 1 to 9, and double digits up to 31. Another example is the grammar for ixml itself, which clearly has to be strict.

For round-tripping we can assume the input is correct, and so the round-tripping grammar can be permissive. For instance, if we take a grammar for a programming language, that contains the rule

```
function-call: name, -"(", parameters, -")".
```

If parameters can be empty (for instance `now()`), then the output for that could be either

```
<function-call><name>now</name><parameters></parameters></function-call>
```

or

```
<function-call><name>now</name><parameters/></function-call>
```

and we need to generate grammar alternatives to recognise both forms.

But if parameters cannot be empty, then we wouldn't need to generate an alternative for the second case, since it will never occur. However, exactly since it will never occur, we don't have to worry about it being in the grammar anyway: since it will never occur, that alternative will never be matched.

Similarly for attributes. If we have a hypothetical grammar that contains something like:

```
element: a, b, c, body.  
@a: acontent.  
@b: bcontent.  
@c: ccontent.  
acontent: ...  
bcontent: ...  
ccontent: ...  
body: ...
```

if we were being strict, we might want to generate alternatives that recognise that a, b, and c could appear in any order:

```
-element: -"<element", (abc; acb; bac; bca; cab; cba), -">", body, -"</  
element>".  
abc: a, b, c.  
acb: a, c, b.
```

etc.

However, we can actually produce a permissive rule like:

```
-element: -"<element", (a; b; c)*, -">", body, -"</element>".  
-a: " a=''", -acontent, "'".  
-b: " b=''", -bcontent, "'".  
-c: " c=''", -ccontent, "'".
```

since this specifies that the element has three attributes, without requiring them to be in any particular order. While this grammar also permits `<element a="xxx"`

a="yyy"> , this is not a problem, since it will never actually occur in the XML we are required to process.

8. Syntactic Equivalences

ixml adds a number of extensions to its grammar-description language, to make life easier for the author, and to make grammars more readable, extensions such as ? for options, *, **, +, and ++ for repetition, and (and) for grouping.

However, while these extensions add expressiveness to the language, they don't add any recognition power: as pointed out in the ixml specification, they can all be straightforwardly transformed into grammars without the extensions, while recognising the same language, and in fact all implementations do this transformation as a form of code generation when processing the language.

As a consequence, the round-tripping process doesn't have to take the syntactic extensions into account; it can either assume the initial transformations have already been done, or do them itself; either way it only has to transform simplified grammar rules that contain none of the extensions.

9. Loss of Information

As pointed out earlier, some information in the source document can be lost when transforming to XML. If a programming language is being recognised, it might be decided not to include nodes for comments in the resulting XML. So on round-tripping, those comments will not reappear in the document. But this is not a problem, since our aim is just to produce a text document that would result in the same XML serialisation.

However, there are some constructs where a decision has to be made, principally because of *inclusions* and *exclusions*.

An inclusion is a ixml construct that allows any character from a set of characters to be matched. We have seen them already in the dates example: ["0"-"9"] matches any single digit. The transformation of such an inclusion remains the same: if the input has a single digit, the output will have the same single digit, and so we recognise it in the same way. The problem comes with deletions. For strings - "/" becomes + "/" on transformation: a deleted slash on input becomes an inserted slash on round-tripping. However, with a deleted inclusion, such as - ["0"-"9"], all we know is that a digit was in the input and deleted, but we don't know which.

There are two options here, both with the same effect. Either transform - ["0"-"9"] by taking a character from the allowable set (for instance the first), and transform to +"0", or update the serialisation process to allow the construct + ["0"-"9"], though with the same semantic of outputting the first character from the set.

Either of these still match the requirement that the resulting round-trip would produce the identical serialisation.

The case for exclusions is slightly harder. A construct such as `~["0"-"9"]` deletes any single character that *isn't* a digit. In this case a character in the *complement* of the set has to be chosen.

10. Ambiguity

In regular ixml processing it can occur that an input document matches the grammar in more than one way. The ixml processor is required to only serialise one of the possible matches, but must also report that the result is ambiguous: that this serialisation is only one of the possible interpretations of the input.

In a similar way, round-tripping can also be ambiguous, in this case meaning "*this is only one of the possible strings that can produce the same serialisation*". In fact, many ixml grammars that are not ambiguous will produce ambiguous round-trips. As a simple example, if symbols on the input may be separated by one or more spaces, that are then deleted on serialisation:

```
input: sym**spaces.  
spaces: -" "+.
```

then on round-tripping any number of spaces would be acceptable. This can best be solved by producing the minimum acceptable on round-tripping: one for "+", and zero for "*".

In a similar way:

```
-optional-a: -"a"?.
```

is ambiguous, because nothing is produced in the output, so we don't know if an "a" was present or not in the input. In this case, either option is acceptable, though the shorter case is probably preferable.

Ambiguity is marked in ixml by adding an attribute to the root element. This is why it is necessary to retain the root element on output of the round-trip: the output is (flat) XML anyway, and we need somewhere to report ambiguity.

11. Inserted Layout

Some implementations may produce a "pretty printed" serialisation of the XML, with added newlines and indentation. While this could be added to the grammar productions in the transformed grammar, it adds a danger of extra ambiguity, since elements may already have whitespace in them as part of the serialisation. For that reason it is better to round-trip XML without added whitespace.

12. Future Work

12.1. Similarities between Serialisation and Transformation

It is worth remarking that the code to produce the transformed grammars is very similar to the code used for serialisation. This should be unsurprising, since both processes walk the parse-tree, and while one outputs the serialisation, the other outputs a grammar that recognises that serialisation. Consequently the two processes are almost isomorphic.

This raises the question: what happens if you take the transformed grammar, and *transform it again*, what does it produce? Well, it round-trips the round-trip: it produces a serialisation that would have produced the round-tripped text. In other words, it serialises the input as XML.

This already works in basic cases, for instance, round-tripping twice the original example in this paper, you get

```
&lt;date>&lt;day>30&lt;/day>&lt;month>06&lt;/month>&lt;year>2024&lt;/year>&lt;/date>
```

(The `<`s are because it is outputting text that represents XML, but not XML itself. For readability, here is that output with the `<`s expanded:)

```
<date><day>30</day><month>06</month><year>2024</year></date>
```

The corollary of this is that the serialisation part of an `ixml` processor could be greatly simplified, only having to output characters, and not having to worry about elements and attributes. Serialisation could be then done by twice transforming the input grammar, and using that instead.

Transformations are currently only partially isomorphic, since there is no reverse operation for the `*` mark, and the proposal mentioned above of allowing the mark `+` on inclusions and exclusions would need to be adopted to prevent loss of information in the grammar on transformation.

A second corollary however is that the `ixml` processor is now unbound from XML, and could be used to produce other serialisations in a fairly straightforward way.

12.2. Possible Additions to `ixml`

Although the additions to `ixml` to enable round-tripping were added to the internal form of `ixml`, and not the external form, if these facilities were to be made available to users as well, some marks would be needed to be added to the language. These are:

- Parse and don't serialise, notated with `*` in this paper;
- Parse nothing, and serialise a node from elsewhere, notated with `+` in this paper, but probably needing to be more general than presented here;

- Flatten a node: undo the recognised structuring by serialising the node as if it were the content of an attribute.

13. Conclusion

Contrary to expectations, it is possible to round-trip the ixml process, as long as you properly define what is understood by round-tripping, and slightly adapt ixml serialisation. This technique is very simple, and not only that, actually allows the ixml processor to be simplified, and to some extent generalised. An added advantage is that, with some work that still has to be done, the process is entirely reversible.

References

- [1] Steven Pemberton. *Invisible XML Specification*. invisiblexml.org. 2022. <https://invisiblexml.org/1.0/> .
- [2] Steven Pemberton. *Invisible XML*. Proceedings of Balisage: The Markup Conference 2013 vol. 10. 2013. 10.4242/BalisageVol10.Pemberton01. <http://www.balisage.net/Proceedings/vol10/html/Pemberton01/BalisageVol10-Pemberton01.html>..
- [3] Steven Pemberton. *A Pilot Implementation of ixml*. Proc. XML Prague 2022. 2022. 41-50. 978-80-907787-0-2 (pdf). <https://archive.xmlprague.cz/2022/files/xmlprague-2022-proceedings.pdf#page=51> .
- [4] Wikipedia editors. *Round-trip format conversion*. Wikipedia. 2024. https://en.wikipedia.org/wiki/Round-trip_format_conversion .

Towards RESTful XQuery 2.0

Adam Retter

Evolved Binary

<adam@evolvedbinary.com>

Abstract

In 2012, RESTful XQuery: Standardised XQuery 3.0 Annotations for REST, proposed a set of standardised annotations and associated machinery for XQuery 3.0 that could be implemented by an XQuery implementation running within a Web context to service REST calls by invoking XQuery User Defined Functions. RESTful XQuery, became colloquially known as RESTXQ 1.0 and was rapidly implemented within a number of XQuery products.

Through real-world use of RESTXQ 1.0, it became apparent through feedback from implementers and users that RESTXQ 1.0 was lacking some ideally desired capabilities; in some products these perceived holes have been plugged by non-standard vendor specific extensions. Additionally, JAX-RS which inspired RESTXQ, and the state-of-the-art for Web communication protocols, have both advanced since 2012.

Herein we identify and review relevant literature and works since 2012. We use this knowledge, along with feedback from implementers and users of RESTXQ, to propose an update to RESTXQ 1.0, namely RESTXQ 2.0.

Keywords: XQuery, REST, RESTXQ, XML Database, JSON, Web, HTTP

1. Introduction

In the Java programming language ecosystem, JAX-RS 1.0 [40] which had been in development since 2006 was finalised and published in 2008. The JAX-RS standard described, amongst other things, how developers could add simple annotations to their Java functions so that those functions, when executed within a suitable web server context, could process HTTP requests and generate responses. JAX-RS promoted REST [11] principles, but it did not strictly enforce them upon the developer. JAX-RS was well received, and its simple approach was, at the time, considered a paradigm shift from the previous need to write Java Servlets and/or Enterprise Java Beans.

In parallel, the W3C XQuery 3.0 standard [48] was published in 2014; it had been under development since shortly after the W3C XQuery 1.0 standard [46] was published in 2007. XQuery 3.0 added a key new language feature in the form of Annotations.

Before the final publication of the XQuery 3.0 standard, the RESTful XQuery [39] paper and the accompanying RESTXQ 1.0 standard [9] were published in 2012. RESTXQ utilised the newly proposed annotations in XQuery 3.0, to adapt concepts from JAX-RS 1.1 [41] that were usable in the Java programming language, to be similarly usable in the XQuery programming language.

RESTXQ 1.0 can be considered a framework for developing Web Applications in XQuery. It emphasises simplicity and ease of use, and it offers a “*convention over configuration*” approach. Whilst RESTXQ JAX-RS prescribes a RESTful approach and provides the primitives for building such RESTful API’s, the choice of how closely to adhere to REST principles remains at the discretion of the developer. It is in fact possible for the developer to also build HTTP 1.1 API’s with RESTXQ that are not in fact RESTful at all!

In our Section 2, we put forward our theory and review the current state of the art in respect to RESTXQ 1.0. Our contributions to this area of research are in section 3, where we propose enhancements and additions to the RESTXQ 1.0 standard that we envisage could lead to a new RESTXQ 2.0 standard. Finally in section 4, we reflect on our work, draw our conclusions and discuss future opportunities for further endeavour.

2. Theory and Review of Literature

Alongside the RESTful XQuery paper, a RESTXQ 1.0 standard [9] was also published in 2012, since then it has received only a few minor non-functional corrections.

In the intervening period, work has progressed in the fields of XQuery standardisation, web and REST software application frameworks, and web communication protocols (on which REST often depends). Additionally, feedback on the experiences of implementers and users of RESTXQ 1.0 has been collated.

We pose the question: *Is RESTXQ 1.0 still fit for purpose today?*

To answer this question we evaluate the state-of-the art within the domains of significance to RESTXQ.

2.1. Review of JAX-RS

Herein we review only the developments of JAX-RS that we believe are relevant or adaptable to RESTXQ, we omit any developments that are solely concerned with the use of JAX-RS within a Java environment.

After the initial version of JAX-RS in 2008, the subsequent JAX-RS 2.0 [32] update is probably the largest of all revisions to date, and introduces several new features:

1. The addition of a ‘*qs*’ quality parameter to any media type within a Produces annotation. When multiple media types with the same ‘*q*’ quality factor are

acceptable to the client, this allows a Resource Function to indicate a media type preference.

2. Filters that allow modification of the properties of the incoming request and/or outgoing response. For example, a Filter could change an incoming POST request into a PUT request; this occurs before an attempt is made to match it to a suitable Resource Function.
3. Entity Interceptors can modify the body of the incoming request and/or outgoing response. For example, an Entity Interceptor might transparently implement GZip decompression of the request, and compression of the response.
4. Validation Constraint Annotations via support for Bean Validation [38]. Custom validation annotations cause a validation function to be called to validate the annotated parameter/property. A validation failure implicitly results in an HTTP error response with the status code 400 Bad Request.
5. Asynchronous Processing of requests by a Resource Function. The Resource Function accepts a callback function as a parameter, spawns a new thread, and returns control to the caller. Once the new thread completes, it invokes the callback function to yield the result. If a Resource Function has to either, utilise significant compute time, or wait on other resources, this can be a useful tool to avoid blocking the caller (i.e. the server).
6. Optional support for JSON. When implemented, handling of request and response body entities with the media-type `application/json` for Resource Functions of JSON and JSON-P must be supported. We can likely discard JSON-P as, due to security issues, it has largely been superseded by cross-origin resource sharing.

JAX-RS 2.1 [33] introduces SSE (Server-sent Events) [53]. A Resource Function with a `Produces` annotation of the media type `text/event-stream` can be used to send events to one or more clients. Server-sent Events are still initiated like any other HTTP request from the client to the server, however unlike a normal request that has a single response, the client holds open the connection, and the server may dispatch many events (responses) over time to the client. As the Resource Function in JAX-RS may need to be able to generate many responses before completing, this is implemented with a callback, and has similarities to the Asynchronous Processing feature added previously in JAX-RS 2.0. SSE can be used as a more performant alternative to the older HTTP Long Polling technique [56].

JAX-RS 3.0 [3] adds a single additional constraint annotation for the HTTP method PATCH [18], which allows for Resource Functions to process partial updates to resources.

JAX-RS 3.1 [4] adds special handling for consuming a request body with the `multipart/form-data` media type. A Resource Function may simply have a

parameter that receives a list or array of parts, or if the number of parts is known statically ahead of time by the developer, they may add the correct number of parameters each annotated as a Form Parameter.

JAX-RS 4.0 [5] adds nothing of relevance for RESTXQ.

2.2. Review of XQuery Standardisation

The latest standard for XQuery, version 3.1 [49], adds support for JSON (JavaScript Object Notation). JSON emerged in 2001 [2] as a mechanism for serialisation and deserialisation of objects in the JavaScript programming language. It was standardised first in 2006 [17] and then again in 2017 [6]. JSON is currently described by the ECMA (European Computer Manufacturers Association) as: “a lightweight, text-based, language-independent syntax for defining data interchange formats”. JSON has seen rapid and significant adoption, this has likely been driven by the meteoric rise of JavaScript as the predominant language for building interactive client-side web pages. The addition of JSON in XQuery 3.1 allows XQuery to increase its applicability for working with data, and to remain relevant when working with the web.

Simultaneously to the publication of XQuery 3.1 in 2017, three complimentary standards were also published:

1. XQuery and XPath Data Model 3.1 [52],
2. XPath and XQuery Functions and Operators 3.1 [50], and
3. XSLT and XQuery Serialization 3.1 [51].

These complimentary standards further strengthened the position of XQuery as a language for processing JSON. New data types were defined to correspond to JSON Map and Array types, new standard library Functions were defined for consuming JSON input, and a new serialisation method for producing JSON output was also defined.

2.3. Review of REST Frameworks

Several HTTP and/or REST frameworks were reviewed in the original RESTful XQuery paper as part of its literature review. We briefly examine how each of those have developed since 2012:

1. The eXist-db REST Server [42] received only one small update since 2012 that is relevant to our research: support for the HTTP PATCH method was added in 2021.
2. The eXist-db XQuery URL Rewriting framework [43] has received no significant updates since 2012, and therefore we need not consider it further. In 2021, Roaster, a higher-level library [27] to make working with URL Rewriting easier was presented. This library does not promote or enforce REST principles,

instead it routes incoming HTTP requests to XQuery functions by parsing a provided OpenAPI 3.0 specification document. To establish a link between an operation defined in an OpenAPI specification document, and an XQuery function, the developer adds the name of the target XQuery function to the `operationId` parameter of the OpenAPI operation; unfortunately, at this time prefixed names are used which can lead to ambiguity in function routing.

3. Marklogic HTTP App Server added support for JSON in 2015 [36], and also added support for GraphQL [24] in 2024 [35]. GraphQL provides a query language API for reading, writing, and subscribing to data, and this is all done over a single HTTP endpoint. This is in direct contrast to REST where multiple endpoints for each resource are typically defined; this does not however prevent this single endpoint being defined as a REST endpoint.
4. The MarkLogic REST Library [37] for use with Marklogic URL Rewriting has received no significant updates since 2012. The Corona library [30] also for use with MarkLogic URL Rewriting has received no updates since 2012 and is now marked as discontinued. As an alternative to URL Rewriting implemented in XQuery, MarkLogic has added the facility for a static declarative set of URL Rewriting rules called Declarative XML Rewriter [34]. This new facility allows rules to be expressed in XML; the stated advantage of this is improved performance due to not having to dynamically evaluate code.
5. The EMC xDB product no longer seems to exist in its own right. It appears to have been integrated into a larger opaque product called Documentum, which was subsequently acquired by OpenText [55]. Therefore we can discard consideration of the xDB REST API, xDB JAX-RS API, and xDB XML REST Framework.
6. Neither the EXPath Web Applications standard [7], nor EXPath Servletx [8] (the only known implementation of the standard), have received any significant updates since 2012. As projects, they have received no updates since 2015 and appear to have been abandoned and therefore we need not consider them further.

Not reviewed in the original RESTful XQuery paper were the frameworks provided by the BaseX Native XML Database. In 2012, BaseX already provided a REST Server [14] implementation that had been developed in 2011 as part of BaseX version 7.0 to replace its previous JAX-RX [12] approach. BaseX's REST Server in terms of API and functionality is very similar to that of eXist-db's REST Server, and therefore we need not consider it separately any further here.

In the Java programming language ecosystem, two interesting projects have appeared since 2012 that are in the relatively new class of microservice frameworks. These are not JAX-RS implementations, but independently make use of function annotations to enable the developer to build RESTful applications:

1. Micronaut [29] released in 2018, and
2. Inverno [26] released in 2021.

The REST facilities offered by these frameworks and the annotations that are provided for that purpose are unremarkable when compared to JAX-RS, in most part due to their similarity; the strengths of these new frameworks lie elsewhere. As such we need not consider them further in this context.

Conspicuous by omission from the original RESTful XQuery paper is the Spring Framework and its Spring REST module. In 2012 Spring REST did not offer a function annotation based approach and so was not relevant for consideration. Whilst it has since added this facility, it offers no significant advantage or differentiator over JAX-RS, and therefore we likewise need not consider it further in this context.

Apart from the BaseX extensions for RESTXQ 1.0 which are covered in Section 2.6.2, there has been little, if any, innovation with respect to REST frameworks in the XQuery programming language ecosystem since 2012.

2.4. Review of Web Communication Protocols

The original RESTful XQuery paper and RESTXQ 1.0 standard published in 2012 were designed to work atop the HTTP 1.1 [23] communications protocol.

Since 2012, the standard for the HTTP/2 communications protocol [21] was published in 2015, and the standard for the HTTP/3 [22] communications protocol in 2022.

HTTP/2 and HTTP/3 use the same HTTP semantics (request methods, status codes, and message fields) as HTTP 1.1. The former focuses on improving performance over TCP, whilst the latter focuses on improving performance by switching from using TCP to UDP. As the HTTP semantics have remained the same in the newer HTTP versions, RESTXQ should be compatible with both HTTP/2 and HTTP/3, and therefore we need not concern ourselves further on this topic.

The WebSocket Protocol [19] published in 2011 was not discussed in the original RESTful XQuery paper. The reasoning for that is unknown, but is likely due to the fact that the WebSocket protocol in of itself does not easily align with REST principles.

The WebSocket Protocol begins from a client sending an HTTP request that asks the server to upgrade it to the WebSocket protocol. Once upgraded and established, the client or server is free to send messages at any time to the other until the connection is closed. WebSockets allow full bi-directional asynchronous communication between the client and server. Like SSE, WebSockets have an asynchronous nature, but the implementation requirements are more demanding due to their bi-directional nature. However, there appears to be little that would prohibit the initial setup request from the client following a RESTful approach.

The Web Socket Protocol which describes sending messages between client and server or vice-versa, does not enforce any constraints on the format of those messages. Therefore in theory, although likely an impedance mismatch, one could potentially implement REST like operations and representations atop a Web-Socket by dispatching those as messages. WebSockets have been used as a more performant alternative to the older HTTP Long Polling technique [56].

Interestingly BaseX provides a set of XQuery functions annotations [15], unrelated to RESTXQ, that allow for working with WebSockets from XQuery running in a web context.

2.5. Review of REST Alternatives

RPC (Remote Procedure Call) is a mechanism for a (client) process to send a request that causes invocation of a procedure (or function) in a separate (server) remote process, and then receive back any result generated by the procedure.

XML-RPC [44] which operates over HTTP, describes the request for the procedure invocation and the response containing the result of its invocation in XML. One stated goal for using XML over HTTP for RPC during the design of the XML-RPC standard was that of human readability of the protocol. XML-RPC later informed the design of the SOAP (Simple Object Access Protocol) standard.

JSON-RPC [25] is very similar to XML-RPC, but as the name implies, uses JSON instead of XML for the format of its request and response. Additionally, it is also transport agnostic, unlike XML-RPC which is restricted to HTTP.

There are also many competing binary RPC protocols, including the modern: gRPC [13] which aims for performance and portability and uses a bespoke TCP protocol, and Cap'n Proto [45] that aims for high performance, but is transport agnostic.

Compared to REST which uses an endpoint URI for each resource, RPC transported over HTTP typically uses only a single endpoint URI with the request body instead describing the procedure to invoke. When comparing XML-RPC or XML-JSON (over HTTP) with RESTXQ, the difference between them becomes less pronounced as then both the RPC and REST approaches allow the remote execution of a function via an XML or JSON request/response; the difference is conceptual, with the developer taking either a resource or function centric design approach.

In theory, although likely an impedance mismatch with much overhead, one could potentially implement a REST like approach atop RPC by wrapping REST parameters and entities within parameters to RPC function invocations; a protocol within a protocol.

Messaging protocols and systems are a wide ranging domain. We define Messaging as a mechanism for loosely coupled and distributed communication. Broadly this usually falls into one of two models:

1. Point to Point, or
2. Publish and Subscribe.

Both models usually involve some form of broker middleware that provides queuing (point to point) or topics (publish and subscribe). The brokers act as the intermediaries allowing systems to indirectly communicate by exchanging messages.

There are many competing messaging protocols, including: the AMQP (Advanced Message Queuing Protocol) standard [1] which is an implementation agnostic wire protocol, and the MQTT OASIS standard [31] which whilst also an implementation agnostic wire protocol, has been designed for low-overhead and strict message delivery guarantees, thus suiting it to use in embedded devices (e.g. IoT (Internet of Things)). Such messaging protocols, as well as a wire format, typically define an envelope format for a message (e.g. framing and headers), whilst leaving the format of the content of the message entirely up to the user.

Unlike REST which uses a Request-Response pattern and is typically a synchronous operation, messaging is always unidirectional and therefore asynchronous. By correlating two opposing unidirectional messages with a common identifier it is possible to create a Request-Response like pattern known as Request-Reply [16], unlike the synchronous Request-Response, Request-Reply remains asynchronous. In theory, one could potentially implement an (asynchronous) REST like approach atop a messaging protocol, queues could be used instead of URI, and REST parameters could be represented as message headers, whilst REST entities could be transmitted in the message body itself.

GraphQL (Graph Query Language) [24], was developed as an alternative to REST with one of its stated goals being that of reducing the number of API calls that need to be made to obtain the required information. Unlike REST where multiple endpoints each provide a representation of a specific type of resource, GraphQL instead opts for a single end-point which may be queried once to return data of multiple types. This does not however prevent the single GraphQL endpoint potentially being defined as a REST endpoint. One may think of a GraphQL API as being similar to a database query API but designed for use across the Web. Both GraphQL and REST typically use HTTP as their transport mechanism, and both can use XML or JSON as their data transfer format. GraphQL is designed from a different conceptual standpoint than that of REST, it requires the developer to identify their data types and model their domain in terms of the queries they wish to perform across those data types, whereas REST of course simply requires the developer to model in terms of the resources (the types of “thing”) that they have in their domain. It is not clear at this time whether a REST like approach could be developed as a layer above a GraphQL API, however it seems unlikely that this would offer much benefit as fundamentally GraphQL and REST each seem suited to solving different problems.

2.6. Review of RESTXQ 1.0 Feedback

Alongside our review of the latest developments in the relevant domains, it has been 12 years since the RESTful XQuery paper and associated RESTXQ 1.0 standard were delivered. Over that time period there has been feedback from users and implementers that we must also consider when determining if RESTXQ 1.0 is still fit for purpose.

2.6.1. EXQuery Project Feedback on RESTXQ 1.0

The RESTXQ 1.0 standard was originally released under the stewardship of the EXQuery Project. The EXQuery is a community project that promotes XQuery, and offers an environment to standardise and publish any extensions to XQuery.

We briefly examine the topics of feedback about the RESTXQ 1.0 standard received by the EXQuery project:

1. More flexible and advanced URI templates for use within Path Annotations. One such suggestion that was received, was that it would be helpful to be able to use Regular Expressions¹ within URI templates²³
2. Support for extracting parameters (also known as Matrix Parameters in JAX-RS) from path components within Path Annotations.
3. A new `%rest:PATCH` annotation to support the HTTP Patch method, and a new `%rest:method` annotation⁴ be useful when working with extensions to HTTP such as WebDAV.
4. The value of Header Parameters should not be tokenized on a comma⁵. It is not clear that there was a well defined use-case for this rule in RESTXQ 1.0, and it has since been observed to cause problems with header values that require commas in them, such as the If-Modified-Since header.
5. A facility for injecting in, and/or programmatically obtaining, a model of the entire HTTP Request⁶. This could be made available as an XDM document node and/or Map.
6. Functions to programmatically interrogate and access parts of the HTTP Request⁷.
7. Support for JSON via parsing request bodies to Map or Array types, and serialisation of Map or Array types via a `%output:method("json")` annotation.

¹<https://github.com/exquery/exquery/issues/38>

²<https://github.com/exquery/exquery/issues/3>

³<https://github.com/exquery/exquery/issues/8>

⁴<https://github.com/exquery/exquery/issues/22>

⁵<https://github.com/exquery/exquery/issues/32>

⁶<https://github.com/exquery/exquery/issues/20>

⁷<https://github.com/exquery/exquery/issues/11>

Potentially also auto-translation by serialisation from XML document-node or element to JSON, and Map/Array to XML.

8. Support for more easily working with HTTP multipart request and response bodies⁸.
9. In-built validation of parameters extracted by RESTXQ from REST requests⁹. For example, whilst query parameters are always optional, as they are classed as Resource Function Parameters, it might be useful to be able to validate if they are present (and the format of their value) or not.
10. New `%rest:error` and `%rest:error-param` annotations¹⁰ that may be added to functions to indicate that they should be called to handle XQuery errors and provide a suitable response to the client.
11. Automatic generation of WADL (Web Application Description Language) for RESTXQ functions¹¹.
12. Annotations to explicitly describe security constraints that must be met for a Resource Function to be executed.
13. The ability to define a catch-all Resource Function that is executed if no other Resource Functions match the incoming request.

2.6.2. BaseX Extensions to RESTXQ 1.0

A proof-of-concept implementation for RESTXQ was created in 2012 for the eXist-db Native XML Database. That implementation was designed to closely follow the RESTXQ 1.0 standard, and as such it has seen no innovation since then.

Conversely, the BaseX Native XML Database rapidly adopted and implemented RESTXQ 1.0 shortly after the paper and standard were published, but has continued to extend its implementation based on feedback from its users. As such BaseX offers a number of interesting vendor specific extensions to RESTXQ 1.0. We briefly examine the extensions to RESTXQ provided in BaseX:

1. An extension to URI templates that are used in a `%rest:Path` annotation so that they may support regular expressions¹².
2. An extension to the `%rest:POST` and `%rest:PUT` annotations to support the conversion of Multipart request body entities¹³ to items (within a sequence) that are injected into the named function parameter. This also supports the uploading of files from HTML forms.

⁸<https://github.com/exquery/exquery/issues/3>

⁹<https://github.com/exquery/exquery/issues/29>

¹⁰<https://github.com/exquery/exquery/issues/16>

¹¹<https://github.com/exquery/exquery/issues/15>

¹²<https://docs.basex.org/wiki/RESTXQ#Paths>

¹³https://docs.basex.org/wiki/RESTXQ#Multipart_Types

3. An extension to the `%rest:POST` and `%rest:PUT` to support converting request bodies that contain CSV, JSON, or HTML into appropriate XDM types¹⁴ (Maps or Document Nodes) before injecting them into the named function parameter. This includes a set of `%input:*` annotations that offer some configuration of the conversion process.
4. A new `%rest:method`¹⁵ annotation to support additional HTTP Methods that are not defined in the HTTP 1.1 standard.
5. An extension to the `%rest:produces` annotation to support server-side quality factors that can influence the selection process when matching the request to a Resource Function.
6. Consideration of Quality Factors¹⁶ from the HTTP request's Accept Header when matching the request to a Resource Function.
7. A new `rest:wadl()`¹⁷ function that can generate a WADL (Web Application Description Language) XML document of all registered Resource Functions.
8. A new `%rest:error`¹⁸ and associated `%rest:error-param` annotations that may be added to a function that is responsible for handling errors.
9. A new `%rest:single`¹⁹ annotation that ensures only a single instance of a function is running concurrently for a specific user session.
10. Automated generation of a response to an HTTP OPTIONS request if there is no matching Resource Function defined.
11. Automated mapping of HTTP HEAD requests to an equivalent Resource Function for HTTP GET (if available); note that the response body from the GET Resource Function will be discarded. This mapping only occurs when there is no matching Resource Function for the HTTP HEAD request.
12. A new `rest:forward`²⁰ element that may be returned in the `rest:response` element of a Resource Function. This instructs the RESTXQ implementation to redirect the request to a different Resource Function.
13. A new `output:serialization-parameters`²¹ element that may be returned in the `rest:response` element of a Resource Function. This instructs the RESTXQ implementation to override any serialisation settings with the specified settings.

¹⁴https://docs.basex.org/wiki/RESTXQ#Content_Types

¹⁵https://docs.basex.org/wiki/RESTXQ#Custom_Methods

¹⁶https://docs.basex.org/wiki/RESTXQ#Quality_Factors

¹⁷https://docs.basex.org/wiki/RESTXQ_Module#rest:wadl

¹⁸https://docs.basex.org/wiki/RESTXQ#Catch_XQuery_Errors

¹⁹https://docs.basex.org/wiki/RESTXQ#Query_Execution

²⁰<https://docs.basex.org/wiki/RESTXQ#Forwards>

²¹https://docs.basex.org/wiki/RESTXQ#Response_Element

3. Proposal for RESTXQ 2.0

Our review has shown that whilst new protocols and facilities have been developed, REST as a protocol has not been rendered obsolete by a newer development. Likewise, the feedback on RESTXQ 1.0 demonstrates that it is being utilised and that users and implementers are requesting additional features.

In response to our earlier posed question, “Is RESTXQ 1.0 still fit for purpose today?”, we believe that in general the principles and design of RESTXQ have stood the test of time and remain sound, but that it would benefit from some refinement and the addition of new features. Therefore we herein set out a proposal for enhancing RESTXQ 1.0, with the presupposition that these may inform a RESTXQ 2.0 standard.

3.1. Backwards Compatibility

We should likely consider how a newer RESTXQ version 2.0 standard may, or may not, be compatible with the previous version of RESTXQ 1.0, and what that means in practice for users of RESTXQ.

Our literature review highlights that RESTXQ 1.0 has achieved some success through adoption, therefore it is our position that RESTXQ 2.0 should be an iteration that builds upon RESTXQ 1.0.

If we were to only add new features to RESTXQ 2.0, without changing any of the existing features inherited from RESTXQ 1.0, we should be able to ensure forwards compatibility between 1.0 and 2.0, i.e. code written for RESTXQ 1.0 should run fine on a RESTXQ 2.0 implementation. However, if a user were to write code for RESTXQ 2.0 and try to run it on a REST 1.0 implementation, the outcome is unknowable. The RESTXQ 1.0 standard offers no facility for describing that the code has been written against it for that version. Furthermore, there may be limitations in RESTXQ 1.0 that are desirable to further tighten or relax in order to deliver a good experience for the developer in RESTXQ 2.0.

There is no perfect solution to this, as RESTXQ 1.0 is already widely deployed in the wild.

To try and improve the situation by enabling it to be detected eagerly in future we propose that a new Option Declaration should be defined for use in the XQuery Prolog: `declare option rest:version "1.0";` in addition the following rules should be added to the RESTXQ 2.0 standard for this feature:

1. If the `rest:version` option is not explicitly defined, then an implementation **MUST** default to the value `1.0`.
2. The value for the option **SHOULD** be: `1.0` or `2.0`.
3. Vendors **MAY** add implementation defined additions to the options value by appending a string starting with the `+` character, e.g. `1.0+BaseX`.

4. If a newer or older RESTXQ feature is used that is not defined in the indicated version, and this is detectable by the implementation, preferably statically or dynamically, then the implementation **MUST** throw an appropriate error.
5. A `rest:version` option only applies to the Resource Functions within the XQuery Library Module in which it is defined; it **MUST NOT** propagate to imported Library Modules.

3.2. Support for the HTTP Patch Method

Informed by review of JAX-RS 3.0, eXist-db, and the EXQuery Project, we propose adding an additional HTTP Method Annotation for the HTTP Patch Method: `%rest:PATCH`. This annotation has the same requirements as the existing `%rest:POST` and `%rest:PUT` annotations in RESTXQ 1.0, i.e.: “it may take an optional string literal which maps the HTTP request body to a named function parameter”.

Our proposal for the HTTP Patch Method is not backwards compatible with RESTXQ 1.0, but is forwards compatible with any later version.

3.3. Support for any HTTP Method

Informed by review of the EXQuery Project, and BaseX, we propose adding an additional HTTP Method Annotation as a catch-all for any HTTP Method for which there is not an existing corresponding annotation: `%rest:method`. In addition the following rules should be added to the RESTXQ 2.0 standard for this feature:

1. It **MUST** take a string literal as its first parameter that identifies the name of the HTTP Method.
2. It **MUST** take a string literal as its second parameter which maps the HTTP request body to a named function parameter; this is similar to `%rest:POST`, `%rest:PUT`, or `%rest:PATCH`.

Our proposal for improved URI Templates is not backwards compatible with RESTXQ 1.0, but is forwards compatible with any later version.

3.4. Improved URI Templates in Path Annotations

RESTXQ 1.0 Path Annotations have the concept of URI Templates, whereby one URI Template extracts and maps the value of one path segment from the URI path to a Resource Function Parameter.

The URI Templates defined in RESTXQ 1.0 should not be confused with the IETF URI Template standard [20]. Whilst URI Templates in RESTXQ 1.0 appear very similar to Level 1 of the IETF URI Template standard, they can be differentiated by the fact that RESTXQ 1.0 URI Templates must start with a `$` character.

Informed by review of the EXQuery Project, and BaseX, we believe that there is a clear demand for enhanced URI Templates in RESTXQ so that they afford the user greater expressivity when describing a path segment that they wish to match.

We have considered the adoption of IETF URI Template in RESTXQ 2.0, however we believe it is a poor for two reasons:

1. Whilst RESTXQ 1.0 URI Templates are used to extract values from URI path segments, the IETF URI Template was instead designed to inject values into URI, and therefore operates in the opposite direction. This impedance mismatch is even discussed at the end of the IETF URI Template standard:

Some URI Templates can be used in reverse for the purpose of variable matching: comparing the template to a fully formed URI in order to extract the variable parts from that URI and assign them to the named variables. Variable matching only works well if the template expressions are delimited by the beginning or end of the URI or by characters that cannot be part of the expansion, such as reserved characters surrounding a simple string expression. In general, regular expression languages are better suited for variable matching.

2. IETF URI Template is focused on the entirety of the URI after the scheme and host (including query strings, fragment identifiers, etc), whilst RESTXQ 1.0 URI Template are used within Path Annotations and so necessarily focus on just segments within the URI path.

We propose that instead of adopting IETF URI Templates, we instead adopt an approach using Regular Expressions; similar to that already in use in BaseX's RESTXQ extensions.

Previously in RESTXQ 1.0 the rules for URI templates²² within Path Annotations are defined as:

The path string MAY contain zero or more URI templates which denote path segments that MUST map to named function parameters. Parameters addressed by templates in the URI path must meet the following constraints:

1. *The cardinality MUST allow for an atomic value, otherwise an error should be raised by the implementation, i.e., it must not be of type empty-sequence().*
2. *The type MUST inherit from xs:anyAtomicType, otherwise, an error should be raised by the implementation.*

Conversion from the URI segment string to the required type is performed at runtime, and an error MUST be raised if conversion is impossible."

To support regular expressions and more flexible path matching, we propose revising this to:

²²[https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html# path-annotation](https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#path-annotation)

The path string MAY contain zero or more URI Templates which capture zero or more path segments. Each of these URI Templates MUST map to a named function parameter of the Resource Function. Regular Expressions may also be used within a URI Template. Matching a URI against a URI Template requires one of three strategies:

1. Simple Matching when no regular expression is used, or
2. Basic Regular Expression Matching when a regular expression without capturing groups is used, or
3. Capturing Regular Expression Matching when a regular expression with one or more capture groups is used.

Additionally explicit typing information may be optionally added to a named function parameter within a URI Template. Any function parameter addressed by a URI Template within the URI path must meet the following constraints:

1. Where a URI Template requires a Simple Match strategy:
 - a. The cardinality MUST allow for an atomic value, otherwise an error should be raised by the implementation, i.e., it must not be of type empty-sequence().
 - b. The type MUST inherit from `xs:anyAtomicType`, otherwise, an error should be raised by the implementation.
 - c. If the URI template indicates an explicit typing, then the type MUST be either `xs:anyAtomicType` or the type indicated in the URI template.
2. Where a URI Template requires a Basic Regular Expression Matching strategy:
 - a. The cardinality MUST allow for zero or one, or zero or more atomic value, otherwise an error should be raised by the implementation, i.e., it must not be of type empty-sequence(), and subtypes of `xs:anyAtomicType?`, or `xs:anyAtomicType*` are allowed. The type MUST inherit from `xs:anyAtomicType`, otherwise, an error should be raised by the implementation.
 - b. The type MUST inherit from `xs:anyAtomicType`, otherwise, an error should be raised by the implementation.
 - c. If the URI template indicates an explicit typing, then the type MUST be either `xs:anyAtomicType` or the type indicated in the URI template.
3. Where a URI Template requires a Capturing Regular Expression Matching strategy:
 - a. The cardinality MUST allow for an array value, otherwise an error should be raised by the implementation, i.e., it must not be of type empty-sequence().

- b. *The type MUST inherit from `array(xs:anyAtomicType)`, otherwise, an error should be raised by the implementation.*
 - c. *The cardinality of the inner array type MUST be compatible with the widest bound of all capturing groups used. For example, an optional capturing group would require the array type of the parameter to have an inner cardinality of zero-or-one (?). Whilst, for example, if an optional capturing group and an optional repeating capturing group were used, this would require the array type of the parameter to have an inner cardinality of zero-or-more (*).*
 - d. *If the URI template indicates an explicit typing, then the inner array type MUST be either `xs:anyAtomicType` or the type indicated in the URI template.*
 - e. *If the URI template indicates more than one explicit typing of different types, then the inner array type MUST be `xs:anyAtomicType`.*
4. *Conversion from the URI path substring matching the URI Template to the required type is performed at run-time, and the conversion must adhere to the rules defined in section N. An error MUST be raised if conversion is impossible.*

Example 1. Valid RESTXQ 1.0 URI Template Examples

```
declare
  %rest:path("/stock/widget/{$wid}")
function local:widget($wid)

declare
  %rest:path("/stock/widget/{$wid}/part/{$pid}")
function local:widget($wid, $pid)

declare
  %rest:path("/stock/widget/{$wid}/part/{$pid}")
function local:widget($wid as xs:int, $pid as xs:int)

declare
  %rest:path("/stock/widget/{$wid}/part/{$pid}")
function local:widget($wid as xs:int?, $pid as xs:int?)

declare
  %rest:path("/stock/widget/{$wid}/part/{$pid}")
function local:widget($wid as xs:int?, $pid as xs:int?)

declare
  %rest:path("/stock/widget/{$wid}/part/{$pid}")
function local:widget($wid as xs:int*, $pid as xs:int*)
```

```
declare
  %rest:path("/stock/widget/{$wid}/part/{$pid}")
function local:widget($wid as xs:int+, $pid as xs:int+)
```

Example 2. Proposed RESTXQ 2.0 URI Template Examples

```
(: Support for explicit conversion types to
: indicate the type for a parameter
: )
declare
  %rest:path("/stock/widget/{$wid(xs:int)}")
function local:widget($wid)

(: Support for a Basic Regular Expression Matching :)
declare
  %rest:path("/stock/widget/{$wid=[0-9]+}")
function local:widget($wid)

(: Support for Basic Regular Expression Matching,
: and an explicit conversion type
: )
declare
  %rest:path("/stock/widget/{$wid(xs:int)=[0-9]+}")
function local:widget($wid)

(: Support for a Basic Regular Expression Matching
: that captures more than one path segment.
: )
declare
  %rest:path("/stock/widget/{$path=.+}")
function local:widget($path)

(: Support for multiple Basic Regular Expression Matching
: within a single path segment.
: Note that multiple URI templates not utilising regular
: expressions within a single path segment are invalid
: and MUST raise an error.
: )
declare
  %rest:path("/stock/widget/{$type=[A-Z]{3}}{$code=[0-9]+}")
function local:widget($type, $code)

(: Support for multiple Basic Regular Expression Matching
: within a single path segment with explicit conversion
: types.
: Note that multiple URI templates not utilising regular
```

```
: expressions within a single path segment are invalid
: and MUST raise an error.
:)
declare
    %rest:path("/stock/widget/{$type(xs:string)=[A-Z]{3}}
{$code(xs:int)=[0-9]+}")
function local:widget($type, $code)

(: Support for Capturing Regular Expression Matching :)
declare
    %rest:path("/stock/widget/{$wid=([A-Z]{3})([0-9]+}")
function local:widget($wid)
declare
    %rest:path("/stock/widget/{$wid=([A-Z]{3})([0-9]+}")
function local:widget($gid as array(xs:anyAtomicType))

declare
    %rest:path("/stock/gadget/{$gid=(?:([A-Z]{3})-)+([0-9]+}")
function local:gadget($gid as array(xs:anyAtomicType+))

declare
    %rest:path("/stock/gadget/{$gid=(?:([A-Z]{3})-)+([0-9]+)?")
function local:gadget($gid as array(xs:anyAtomicType*))

(: Support for Capturing Regular Expression Matching
: with explicit conversion types.
:)
declare
    %rest:path("/stock/gadget/{$gid(xs:string, xs:int)=(?:([A-Z]{3})-)+
+([0-9]+)?")
function local:gadget($gid as array(xs:anyAtomicType*))
(: Support for Capturing Regular Expression Matching
: where the entire match is also appended to the array
: parameter as an `xs:string`; caused by the `==`
: instead of the usual `=`.
: Note that using `==` is only allowed for Capturing
: Regular Expression Matching, using it for Basic
: Regular Expression Matching is invalid and MUST
: raise an error.
:)
declare
    %rest:path("/stock/gadget/{$uid==( [0-9]+ )-( [0-9]+ )"}
function local:gadget($gid as array(xs:anyAtomicType))
```

Our proposal for improved URI Templates is not backwards compatible with RESTXQ 1.0, but is forwards compatible with any later version.

3.5. Improved HTTP Header Parsing

In RESTXQ 1.0, if the value of an HTTP Header parameter contained one or more comma characters then the value would be tokenized on the comma character. This served to work well for some headers, but caused issues with headers such as `If-Modified-Since` where the value must be formatted as a date as defined by either: RFC 1123, RFC 850, or ANSI-C's `asctime()` function. As the date format of RFC 1123 and RFC850 both include a comma character, this made it impossible to inject the header value easily via the `%rest:header-param` annotation.

Informed by review of the EXQuery Project, we propose to add additional options to control the parsing of HTTP Header Parameters.

In RESTXQ 1.0, the arguments for the `%rest:header-param` are:

1. The first argument is the name of the HTTP Header to parse.
2. The second argument is the name for the Resource Function parameter to inject the header value into.
3. The third (and any subsequent) arguments are the default values to inject into the named Resource Function parameter if the HTTP Header is not present in the HTTP request.

We propose modifying the format of the second argument to allow control over the parsing of the HTTP Header value. At present the second argument must be of the format: `{ $\$$ fn-param-name}`. We propose instead to allow additional arguments after the Resource Function Parameter name like: `$\$$ fn-name-name(tokenize='pattern')`. In addition, the following rules should be added to the RESTXQ 2.0 standard for this feature:

1. If a `tokenize` option is provided, and the value of the option is not the literal string `false`, an implementation **MUST** tokenize the value from the header on the provided pattern by following the rules of the `fn:tokenize` function before injecting it into the named resource function parameter.
2. If a `tokenize` option is provided, and the value of the option is the literal string `false`, tokenization **MUST** not be performed on the value from the header before injecting it into the named resource function parameter.
3. If a `tokenize` option is not provided, and a header 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.

Our proposal for improved HTTP Header Parsing is not backwards compatible with RESTXQ 1.0, but is forwards compatible with any later version.

3.6. Support for Server Side Quality Factors

Informed by review of JAX-RS 2.2, and BaseX, we propose modifying the media type argument(s) used within `%rest:produces` annotations to allow for an additional server-side quality factor argument.

In RESTXQ 1.0, if an HTTP request from a client indicates that it accepts a response adhering to more than one media-type with the same quality factor, and if there is one or more resource functions that match more than one of those media-types, it is currently ambiguous as to which media-type should be returned to the client. We propose adding a `qs` argument that takes a quality factor that is used to resolve the ambiguity in such situations. The `qs` argument is the server-side equivalent to the `q` argument that may be used in the HTTP Accept header within an HTTP request from the client.

Example 3. Example of Server Side Quality Factors

```
declare function
  %rest:GET
  %rest:path("/README")
  %rest:produces("text/plain;qs=0.5")
  %output:method("text")
local:text-readme()

declare function
  %rest:GET
  %rest:path("/README")
  %rest:produces("text/markdown;qs=1.0")
  %output:method("text")
local:text-markdown()
```

Our proposal for Server Side Quality Factors is not backwards compatible with RESTXQ 1.0, but is forwards compatible with any later version.

3.7. Inclusion of Quality Factors in Matching Resource Functions

The HTTP Mechanics section of RESTXQ 1.0 describes the mechanism for selection of a Resource Function based on media-types²³ which is fed by the `%rest:consumes` and `%rest:produces` annotations. This section includes no explicit consideration of quality factors. We propose that this section should be expanded to include an explicit and detailed explanation of how client-side `q`

²³<https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#media-type-preference>

quality factors, and server-side qs” (see Section 3.6) should be considered when matching HTTP requests to Resource Functions.

3.8. Support for JSON

XQuery 3.0 did not have any in-built support for JSON, and as this was the target for RESTXQ 1.0, it likewise had no in-built support for JSON.

XQuery 3.1 and its associated standards added a number of key features for working with JSON. Whilst it is possible to use RESTXQ 1.0 in XQuery 3.1, the user is left to manually parse and serialise JSON requests and responses.

Informed by review of JAX-RS 2.0, XQuery 3.1, MarkLogic’s HTTP App Server, the EXQuery Project, and BaseX, we propose adding new features and mechanisms to support more easily working with JSON in RESTXQ. Just as the changes to XQuery 3.1 to support JSON were pervasive (data model, operators, parsing and serialisation), the necessary changes required to support this in RESTXQ would also be extensive.

Our proposals for JSON support are not backwards compatible with RESTXQ 1.0, but are forwards compatible with any later version.

3.8.1. Support for JSON Request Bodies

We propose that the REST method annotations that may extract a request body and inject it into a Resource Function parameter (`%rest:POST`, `%rest:PUT`, `%rest:PATCH`, and `%rest:method`) need to be updated to also be able to process a request body containing JSON.

Previously in RESTXQ 1.0 the rules for resource functions that were to receive a request body²⁴ are defined as:

The function parameter for the request body must meet the following constraints:

1. *The cardinality MUST allow for one or more of the typed item(s).*
2. *The type MUST be 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. The interpretation of the request body is similar to that of the EXPath HTTP Client:*
 - a. *If the Media Type indicated by the Content-Type header matches `text/*` (excluding `text/xml`), the function parameter type will be `xs:string`.*
 - b. *If the Media Type indicated by the Content-Type header is of an XML type, the request body is parsed as XML and the function parameter type will be `document-node()`. At least the Media Types `application/xml`*

²⁴<https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#method-annotation>

and text/xml MUST be interpreted as XML types, although implementations MAY support additional XML types.

- c. Otherwise, a binary media type is assumed, and the function parameter type will be `xs:base64Binary`.*
- d. An implementation MAY provide support for other input type mappings such as HTML or JSON."*

To support JSON, we propose inserting an additional rule between 2.2 and 2.3 and revising rule 4:

2.3 If the Media Type indicated by the Content-Type header is of a JSON type, the request body is parsed (following the rules of the `fn:parse-json` function) as JSON and the function parameter type will be either `map(*)` or `array(*)`. At least the Media Type `application/json` MUST be interpreted as a JSON type, although implementations MAY support additional JSON Media Types.

2.4 Otherwise, a binary media type is assumed, and the function parameter type will be `xs:base64Binary`.

2.5 An implementation MAY provide support for other input type mappings such as HTML or CSV.

We further propose to allow additional arguments to the resource function parameter name that allow the user to configure options for parsing of the request body, for JSON these should follow the options set out in the `fn:parse-json` function (except for the fallback function), for example:

Example 4. JSON Parsing of HTTP Request Body in Method Annotation

```
%rest:PUT("${request-body(liberal=true,duplicates=reject,escape=true)}")
```

3.8.2. Support for JSON Response Bodies

We propose that the version of the W3C XSLT and XQuery Serialization used by RESTXQ²⁵ be updated from 3.0 (as used in RESTXQ 1.0) to 3.1; this also would allow support for serialisation of Maps and Arrays to JSON.

3.9. Support for Multipart

Informed by review of JAX-RS 3.1, the EXQuery Project, and BaseX, we propose adding support for the parsing and serialisation of HTTP Multipart requests and responses.

Our proposals for Multipart support are not backwards compatible with RESTXQ 1.0, but are forwards compatible with any later version.

²⁵<https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#serialization>

3.9.1. Support for Multipart Form Data Requests

We propose that the Form Parameter Annotation (`%rest:form-param`) in RESTXQ 1.0 be extended to support Multipart Form Data [47] as identified by the media type `multipart/form-data`. In RESTXQ 1.0, the rules²⁶ for converting the value of the form field to a Resource Function parameter are:

1. *The cardinality MUST allow for zero or many atomic values in the case of Query²⁷, Form²⁸, or Header²⁹ parameters, or zero or one atomic value in the case of Cookie³⁰ parameters, otherwise an error MUST be raised by the implementation.*
2. *The type MUST inherit from `xs:anyAtomicType`, otherwise, an error MUST be raised by the implementation.*

We propose revising this to:

1. The cardinality MUST allow for zero or many atomic values in the case of Query³¹, Form³², or Header³³ parameters, or zero or one atomic value in the case of Cookie³⁴ parameters, otherwise an error MUST be raised by the implementation.
2. The type MUST inherit from `xs:anyAtomicType` or be an `map(*)` type, otherwise, an error MUST be raised by the implementation.
3. When an `xs:anyAtomicType` (or subtype) is indicated:
 - a. When the request Media Type is `application/x-www-form-urlencoded` the parameter is injected with the converted value of the form field.
 - b. When the request Media Type is `application/form-data` the parameter is injected with the body of the multipart part (i.e. a converted value for textual form fields, or an `xs:base64Binary` for file fields).

²⁶<https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#resource-function-parameters>

²⁷ <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#query-param-annotation>

²⁸ <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#form-param-annotation>

²⁹ <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#header-param-annotation>

³⁰ <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#cookie-param-annotation>

³¹ <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#query-param-annotation>

³² <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#form-param-annotation>

³³ <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#header-param-annotation>

³⁴ <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#cookie-param-annotation>

4. When a `map(*)` type is indicated:

- a. When the request Media Type is `application/x-www-form-urlencoded` the parameter is injected with a map with a single body property containing the converted value of the form field.
- b. When the request Media Type is `application/form-data` the parameter is injected with a map of the format:

```
{
  parameters: [
    {"name": "filename", "value": "filename if file"},
    {"name": "filename*", "value": "non-ascii filename if file"}
  ],
  headers: [
    {name: "my-header-name", value: "my-header-value"}
  ],
  "body": "either converted value or xs:base64binary"
}
```

3.9.2. Support for multipart/* Requests

We propose that the REST POST Method annotation should be updated so that it may handle parsing of a multipart request body and inject it into a Resource Function parameter.

Previously in RESTXQ 1.0 the rules for processing the request body for the POST Method annotation³⁵ are defined as:

The function parameter for the request body must meet the following constraints:

1. *The cardinality MUST allow for one or more of the typed item(s).*
2. *The type MUST be 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. The interpretation of the request body is similar to that of the EXPath HTTP Client:*
 - a. *If the Media Type indicated by the Content-Type header matches `text/*` (excluding `text/xml`), the function parameter type will be `xs:string`.*
 - b. *If the Media Type indicated by the Content-Type header is of an XML type, the request body is parsed as XML and the function parameter type will be `document-node()`. At least the Media Types `application/xml` and `text/xml` MUST be interpreted as XML types, although implementations MAY support additional XML types.*

³⁵<https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#method-annotation>

- c. Otherwise, a binary media type is assumed, and the function parameter type will be `xs:base64Binary`.
- d. An implementation MAY provide support for other input type mappings such as HTML or JSON.

To support Multipart requests, we propose inserting an additional rule between 2.2 and 2.3:

2.3 If the Media Type indicated by the Content-Type header is of a Multipart type, the request body is parsed and the parts extracted. The body of each part MUST be parsed as described previously. Note that this may be a recursive process as a multipart part may itself have a multipart body. The function parameter type will be `map(*)`. At least the Media Types `multipart/mixed`, `multipart/related`, and `multipart/alternative` MUST be interpreted as Multipart types, although implementations MAY support additional Multipart Media Types. The map has the format:

```
{
  parameters: [
    {"name": "filename", "value": "filename if file"},
    {"name": "filename*", "value": "non-ascii filename if file"}
  ],
  headers: [
    {name: "my-header-name", value: "my-header-value"}
  ],
  "body": "either converted value or xs:base64binary"
}
```

3.9.3. Support for Multipart Responses

We propose adding a new serialisation type for Multipart that will work with the existing serialisation mechanism of RESTXQ. The new serialisation type could be called `rest:multipart`, and can be used like: `%output:method("rest:multipart")`.

This new serialisation would take as input a sequence of Maps (multipart parts) in the format of:

```
{
  "rest:response" {
    "output:serialization-parameters": {
      "output:method": "xml"
    }
  },
  parameters: [
    {"name": "filename", "value": "filename if file"},
    {"name": "filename*", "value": "non-ascii filename if file"},
  ]
}
```

```
],
headers: [
  {name: "my-header-name", value: "my-header-value"}
],
"body": "either converted value or xs:base64binary"
}
```

Each body within the map can be serialised according to the existing rules, or its serialisation method overridden with an optional `output:method` parameter.

3.10. Support for Serialization within the Response

Informed by review of BaseX, we propose adding an additional `output:serialization-parameters` element to the `rest:response` element; this element is defined in the W3C XSLT and XQuery Serialization 3.1 standard. This element would allow the developer to dynamically determine the serialisation parameters of a Resource Function during processing of the function. It would complement, but override, any corresponding `%output` prefixed function annotations.

3.11. Support for Handling XQuery Errors

Informed by review of BaseX, we propose adding an additional Error Handling Annotation: `%rest:error`. The annotation may take one or more name tests for XQuery error codes. These name tests have the same syntax as that defined for Try/Catch Expressions in the W3C XQuery 3.1 standard.

When executing a Resource Function, if an XQuery error is raised, and if a function with a matching `%rest:error` annotation is present, this function **MUST** be executed. This function **MUST** return any of the types that any other Resource Function may return. If two functions are annotated with the same name test, an implementation **MUST** raise an error. If an error occurs during the processing of a function annotated with the `%rest:error` annotation, processing **SHOULD** cease and the implementation **MUST NOT** call any further functions annotated with `%rest:error`.

In addition, we also propose adding an additional Error Parameter Annotation: `%rest:error-param`. This annotation **MUST** take two string literal parameters. The first parameter is the qualified name of an error parameter, for example: `err:code`, `err:description`, `err:value`, `err:module`, `err:line-number`, `err:column-number`, `err:additional`, or any other implementation defined name. The second parameter is the name of the XQuery function parameter into which the value must be injected, the format of this name follows that of the other RESTXQ parameter annotations.

3.12. Non Sequitur

There are a number of features or facilities that we have discovered during our literature review that we consider to be out of scope or inappropriate for RESTXQ:

1. Filters and Entity Interceptors as described in JAX-RS allow a great deal of flexibility, but it remains unclear how these would be implementable in RESTXQ.
2. Validation Constraint Annotations as described in JAX-RS and requested in feedback to the EXQuery Project and based on the Java Bean Validation standard would be a welcome addition. These are implemented in Java as a set of annotations that may be applied to function annotations. Unfortunately, XQuery does not yet permit annotations to be added to function parameters.
3. The ability to define a catch-all Resource Function that is executed if no other Resource Functions match the incoming request. This was requested in feedback to the EXQuery Project however we believe that such a feature would be poorly aligned with the declarative approach of RESTXQ.
4. A new `%rest:single` annotation that ensures only a single instance of a function is running concurrently for a specific user session. This has been implemented in BaseX. However, we believe that this should not be included in RESTXQ and should remain a vendor extension, otherwise it would force all implementers to track session state; REST works very hard to avoid server-side state.
5. Automated generation of a response to an HTTP OPTIONS request if there is no matching Resource Function defined. This has been implemented in BaseX. However, we believe that this should not be included in RESTXQ as it produces a response that the developer had not indicated that they intend to produce.
6. Automated mapping of HTTP HEAD requests to an equivalent Resource Function for HTTP GET (if available); note that the response body from the GET Resource Function will be discarded. This mapping only occurs when there is no matching Resource Function for the HTTP HEAD request. This has been implemented in BaseX. However, we believe that this should not be included in RESTXQ as it produces a response that the developer had not indicated that they intend to produce.
7. A new `rest:forward` element that may be returned in the `rest:response` element of a Resource Function. This instructs the implementation to perform a server-side redirect of the request to a different Resource Function. This has been implemented in BaseX. However, we believe that this should not be included in RESTXQ as it makes the handling of requests more opaque, and

equivalent functionality can already be obtained through an XQuery function call to the desired code instead.

8. A new `rest:wadl()` function that can generate a WADL (Web Application Description Language) XML document of all registered Resource Functions. This has been implemented in BaseX. However, we believe that this should not be included in RESTXQ and should remain a vendor extension, otherwise it would force all implementers to provide support for WADL.
9. We believe that GraphQL, RPC, and Messaging protocols are best left as alternatives to REST, and therefore should not be integrated into RESTXQ.

4. Conclusion

In 2012 when the RESTful XQuery paper was published, REST was already a well defined and established architectural approach to designing Web APIs. It is therefore perhaps not unsurprising that since then the improvements to REST frameworks and implementations that we have identified and discussed have been few and of an incremental rather than foundational nature.

REST remains relevant today, and it is the de-facto standard for building Web API. We have argued that RESTXQ 1.0 whilst still relevant, could benefit from some refinement and additional features. Furthermore, we have identified 11 such enhancements that could be added to RESTXQ 1.0 to create a RESTXQ 2.0 standard. Finally, we have discussed a number of features and vendor extensions that we believe at this time should remain out-of-scope for RESTXQ.

4.1. Future Work

1. It may be desirable to add further support to URI Template used within Path Annotations to assist the user in extracting URI Matrix Parameters [54] from URI paths such as: `/ user/ ;id=3,4,5;type=p` or `/ user/ ;id=3;id=4;id=5;type=p`.
2. It may be desirable to also allow a Map representation of the REST Response document³⁶ (`rest:response`).
3. It may be desirable to support parsing of additional types of request bodies such as CSV or HTML for Method Annotations. Ideally a generic mechanism should be identified for this that allows any mediatype to be supported through a user defined parsing function.
4. Establishing if there is a need to programmatically access all or parts of the HTTP Request. The EXQuery Project has prototyped this in their HTTP

³⁶<https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html#response-format>

Request Module [10]. It appears however at an impedance mismatch with the declarative nature of RESTXQ. We can only speculate that perhaps it was designed in lieu of alternative facilities that could be available in a newer version of RESTXQ as set out in this paper.

5. It remains unclear as to how Asynchronous Processing, as described in JAX-RS 2.0 and then built upon in JAX-RS 2.1 for SSE, could be implemented in RESTXQ. Investigation into adapting different processing models in XQuery [28] for use in RESTXQ needs to be undertaken.
6. Consideration should be given to whether WebSockets could be implemented using a similar approach to that of the Asynchronous Processing used by SEE in JAX-RS 2.1. At which point, whether Websockets belong in a future RESTXQ standard, or whether they should be defined separately needs to be discussed.
7. Consideration should be given to establishing a separate set of Security Annotations to describe security constraints that must be met for a function to be executed. Such an approach has already been demonstrated by BaseX. RESTXQ may need some further work to have it describe how it interacts with such annotations, for example, should authentication and authorisation failures be mapped directly to HTTP 401 Unauthorized and HTTP 403 Forbidden response codes.
8. The OpenAPI standard lends itself well to the production of code generation tools. Several such tools have already been developed for a variety of other programming languages. Development of a tool for generating a skeleton XQuery Library Module from an OpenAPI standard could be an interesting project.
9. Development and publication of a clear and complete standard for RESTXQ 2.0

References

- [1] *AMQP v1*. 2011. AMQP Working Group. <http://www.amqp.org/sites/amqp.org/files/amqp.pdf>.
- [2] *The JSON Saga*. 2001. Douglass Crockford. <http://crockford.com/pp/jsonsaga.ppt>.
- [3] *Jakarta RESTful Web Services, version 3.0*. 2020. Eclipse Foundation. <https://jakarta.ee/specifications/restful-ws/3.0/jakarta-restful-ws-spec-3.0.pdf>.
- [4] *Jakarta RESTful Web Services, version 3.1*. 2022. Eclipse Foundation. <https://jakarta.ee/specifications/restful-ws/3.1/jakarta-restful-ws-spec-3.1.pdf>.

- [5] *Jakarta RESTful Web Services, version 4.0*. 2024. Eclipse Foundation. <https://jakarta.ee/specifications/restful-ws/4.0/jakarta-restful-ws-spec-4.0.pdf> .
- [6] *The JSON Data Interchange Syntax*. 2017. European Computer Manufacturers Association. https://ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf .
- [7] *Web Applications*. 2013. EXPath Community Group. <https://expath.org/spec/webapp> .
- [8] *Servlex*. 2015. EXPath Community Group. <https://servlex.net/doc> .
- [9] *RESTXQ 1.0*. 2012. 2012. <https://exquery.github.io/exquery/exquery-restxq-specification/restxq-1.0-specification.html> .
- [10] *HTTP Request Module 1.0*. 2013. The EXQuery Project. <https://exquery.github.io/expath-specs-playground/request-module-1.0-specification.html> .
- [11] *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Roy Fielding. https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf .
- [12] *JAX-RX:Unified REST Access to XML Resources*. 2010. Sebastian Graf, Lukas Lewandowski, and Christian Grün. <https://kops.uni-konstanz.de/server/api/core/bitstreams/f1ca6230-9540-4cc4-b5de-5af0fabcf11d/content> .
- [13] *gRPC - A high performance, open source universal RPC framework*. 2024. gRPC Authors. <https://grpc.io/> .
- [14] *BaseX - REST*. 2011. , . Christian Grün. <https://docs.basex.org/wiki/REST> .
- [15] *BaseX - WebSockets*. 2018. Christian Grün. <https://docs.basex.org/wiki/WebSockets> .
- [16] *Request-Reply - Enterprise Integration Patterns*. 2003. Addison Wesley. Gregor Hoppe and Bobby Woolf. <https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html> .
- [17] *The application/json Media Type for JavaScript Object Notation (JSON)*. 2006. Internet Engineering Task Force. <https://www.rfc-editor.org/info/rfc4627> .
- [18] *PATCH Method for HTTP*. 2010. Internet Engineering Task Force. <https://www.rfc-editor.org/info/rfc5789> .
- [19] *The WebSocket Protocol*. 2011. Internet Engineering Task Force. <https://www.rfc-editor.org/info/rfc6455> .
- [20] *URI Template*. 2012. Internet Engineering Task Force. <https://datatracker.ietf.org/doc/html/rfc6570> .

- [21] *Hypertext Transfer Protocol Version 2 (HTTP/2)*. 2015. Internet Engineering Task Force. <https://www.rfc-editor.org/info/rfc7540> .
- [22] *HTTP/3*. 2022. Internet Engineering Task Force. <https://www.rfc-editor.org/info/rfc9114> .
- [23] *Hypertext Transfer Protocol - HTTP/1.1*. 1997. Internet Engineering Task Force. <https://www.rfc-editor.org/info/rfc2068> .
- [24] *Information technology - Database languages - GQL, ISO/IEC 39075:2024*. 2024. International Standards Organization. <https://www.iso.org/standard/76120.html> .
- [25] *JSON-RPC 2.0 Specification*. 2013. JSON-RPC Working Group. <https://www.jsonrpc.org/specification> .
- [26] *The Inverno Framework - Using a module*. 2012. Jeremy Kuhn. <https://inverno.io/docs/release/reference/html/index.html#using-a-module> .
- [27] *Roaster - declarative routing for eXist-db*. 2021. Juri Leino. <https://declarative.amsterdam/presentations/presentations/da.2021.leino.declarative-routing-for-existdb> .
- [28] *Task Abstraction for XPath Derived Languages*. 2019. Debbie Lockett and Adam Retter. <https://www.adamretter.org.uk/papers/task-abstraction-for-xpdl-february-2019.pdf> .
- [29] *Micronaut Framework - Creating a Server Application*. 2024. Micronaut Foundation. <https://docs.micronaut.io/4.4.8/guide/#creatingServer> .
- [30] *Corona - A REST interface to MarkLogic*. 2012. 2012. <https://github.com/marklogic-community/Corona> .
- [31] *MQTT Version 5.0*. 2019. OASIS. <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html> .
- [32] *JAX-RS: Java API for RESTful Web Services, version 2.0*. 2013. Oracle Corporation. https://download.oracle.com/otn-pub/jcp/jaxrs-2_0-fr-eval-spec/jsr339-jaxrs-2.0-final-spec.pdf .
- [33] *JAX-RS: Java API for RESTful Web Services, version 2.1*. 2017. Oracle Corporation. https://download.oracle.com/otn-pub/jcp/jaxrs-2_1-final-eval-spec/jaxrs-2_1-final-spec.pdf .
- [34] *Creating a Declarative XML Rewriter to Support REST Web Services*. 2024. Progress Software Corporation. <https://docs.marklogic.com/guide/app-dev/XMLrewriter> .

- [35] *MarkLogic 11 Release Notes - New Features - GraphQL*. 2024. Progress Software Corporation. <https://docs.marklogic.com/11.0/guide/release-notes/en/new-features-in-marklogic-11/graphql.html> .
- [36] *MarkLogic Knowledge Base - JSON in versions 6, 7, and 8*. 2015. Progress Software Corporation. <https://help.marklogic.com/Knowledgebase/Article/View/218/0> .
- [37] *Creating an Interpretive XQuery Rewriter to Support REST Web Services*. 2024. Progress Software Corporation. https://docs.marklogic.com/guide/app-dev/rest#id_45642 .
- [38] *JSR 303: Bean Validation*. 2009. Red Hat. https://download.oracle.com/otn-pub/jcp/bean_validation-1.0-fr-oth-JSpec/bean_validation-1_0-final-spec.pdf .
- [39] *RESTful XQuery*. 2012. Adam Retter. https://www.adamretter.org.uk/papers/restful-xquery_january-2012.pdf .
- [40] *JAX-RS: Java API for RESTful Web Services, Version 1.0*. 2008. Sun Microsystems. <https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>³⁷ .
- [41] *JAX-RS: Java API for RESTful Web Services, Version 1*. 2009. Sun Microsystems. https://download.oracle.com/otn-pub/jcp/jaxrs-1.1-mrel-eval-oth-JSpec/jax_rs-1_1-mrel-spec.pdf .
- [42] *REST-Style Web API*. 2019. The eXist-db Authors. https://www.exist-db.org/exist/apps/doc/devguide_rest .
- [43] *URL Rewriting*. 2019. The eXist-db Authors. <https://www.exist-db.org/exist/apps/doc/urlrewrite> .
- [44] *XML-RPC Specification*. 2003. UserLand Software. <https://xmlrpc.com/spec.md> .
- [45] *Cap'n Proto - Introduction*. 2013. Kenton Varda. <https://capnproto.org/> .
- [46] *XQuery 1.0: An XML Query Language*. 2007. World Wide Web Consortium. <https://www.w3.org/TR/xquery/> .
- [47] *HTML 5 - Multipart form data*. 2014. W3C. <https://www.w3.org/TR/2014/REC-html5-20141028/forms.html#multipart-form-data> .
- [48] *XQuery 3.0: An XML Query Language*. 2014. World Wide Web Consortium. <https://www.w3.org/TR/xquery-30/> .
- [49] *XQuery 3.1: An XML Query Language*. 2017. World Wide Web Consortium. <https://www.w3.org/TR/xquery-31/> .

³⁷ <https://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf>

- [50] *XPath and XQuery Functions and Operators 3.1*. 2017. World Wide Web Consortium. <https://www.w3.org/TR/xpath-functions-31/> .
- [51] *XSLT and XQuery Serialization 3.1*. 2017. World Wide Web Consortium. <https://www.w3.org/TR/xslt-xquery-serialization-31/> .
- [52] *XQuery and XPath Data Model 3.1*. 2017. World Wide Web Consortium. <https://www.w3.org/TR/xpath-datamodel-31/> .
- [53] *HTML, 9.2 Server-sent Events, living standard - Last Updated 24 May 2024*. 2024. World Wide Web Consortium. <https://html.spec.whatwg.org/multipage/server-sent-events.html> .
- [54] *Matrix URIs - Ideas about Web Architecture*. 1996. World Wide Web Consortium. <https://www.w3.org/DesignIssues/MatrixURIs.html> .
- [55] *Documentum*. 2024. Wikipedia. <https://en.wikipedia.org/wiki/Documentum> .
- [56] *Push technology - Long polling*. 2024. Wikipedia. https://en.wikipedia.org/wiki/Push_technology#Long_polling .

Tutorial Development XML Mashup with XProc

Erik Siegel
<erik@xatapult.nl>

Abstract

When you create a tutorial, training or course, you have to maintain a lot of information: slides, exercises, instructions, etc. Numbering and referencing must be correct. There is usually duplication: documents are used in multiple exercises and sometimes also shown on slides.

It's rather a chore to keep everything consistent during development and maintenance. This presentation describes an attempt to make this easier using specific XML markup, and processing software built from XProc 3.0 components.

Keywords: XML, XProc

1. A day in the life of a tutorial developer

So how do you develop a tutorial? Me, I start with a presentation, trying to find ways to divide the material into sections and add exercises to these.

Developing exercises means (for XML courses), adding input files, stylesheets, schemas, etc. This must be accompanied by some kind of instruction: what is the participant supposed to do. I also like to add a solution, so participants who don't understand it can see how to solve it. Sometimes that is quite a lot of files for one exercise. Files that are sometimes reused in a few other exercises or shown on slides.

I often use oXygen⁶ during a tutorial to make life as easy as possible for the participants. This means developing an oXygen project with, among others, pre-defined validation and transformation scenarios. The result is that for, for instance, an XSLT tutorial, participants can open a template XSLT file, write their code and press transform, without having to bother with setting up things. No distractions, keep the focus on what is being taught.

Now all these files, or at least the directories they're in, are numbered, and these numbers tie in with what's shown in the presentation.

And then the nightmare begins: You want insert another exercise and the numbering changes. This means changing things in file/directory names, instructions, oXygen project and the presentation. It's all too easy to forget it somewhere.

⁶ <https://www.oxygenxml.com/>

Or you change something in the contents of a file. But this file might be shown in the presentation or was reused a couple of times (in different exercises, so with different file/directory names).

Again, all too easy to do this inconsistently. Again and again I found myself doing boring, repetitive and error-prone tasks that had to be carried out very carefully. I noticed it, psychologically, stopped me from enhancing tutorials with, for instance, an extra exercise, because it was too much too boring work.

There was obviously something wrong, what could I do about it?

2. What are the requirements?

For an XML hacker like me, the start of answer was, of course, obvious: create some kind of markup, an XML based language, that represents the tutorial. But to do that properly, we need to know what the requirements are. So I took the “helicopter” and looked down on the way I organized and carried out my training.

First of all, what are (for me) the building blocks of a training?

- I always use a *presentation* as guideline. It supports me explaining the theory and points the participants to exercises they must do.

Sometimes the contents of files that are used in the exercises are also present in the presentation. For instance, when discussing a solution to one of the problems.

I often use tricks like appearing/disappearing text and objects, balloons and call-outs on the slides. I think this definitely helps explaining the hard parts. Therefore, the ability to use a presentation design tool, like Microsoft Powerpoint, is a requirement.

- Every tutorial contains *exercises*. An exercise consists of:
 - An exercise contains instruction about what to do. This is formatted text in a PDF.
 - Template files: files that the participants can use as a starting point for their solution. For instance a skeleton XSLT document.
 - Input files: Files that are needed for the exercise. For instance an input document to transform.
 - Solution files: I always provide a solution to the exercise, so people can have a look when they’re stuck or are just curious how I solved it. Sometimes there’s also a PDF explaining the solution.

Files are usually XML documents but can be text also (for instance XQuery or JSON).

- Provide some kind of *tool/environment support*. To keep the participants eyes on the ball, it works best to use a tool/environment that helps them filling in the exercises and run their solutions: make it as easy as possible, learning

something new is already hard enough. Most of the times I use oXygen for this, so I prepare an oXygen project that provides easy access to the exercises and enables running the solutions (for instance XLST scripts) with the click of a button.

- Sometimes there are additional learning materials, usually in the form of PDF or ZIP files.

A solution must support creating and maintaining these building blocks. What are the additional requirements?

- Everything must have a *single source*. So no duplication of, for instance, a file in an exercise and presented on a slide. The environment must take care of that.
- Directories and files for the exercises must be *consistently named*. So, for instance, an input file is always called `input.xml`, a template `template.xml`, etc.
- *Generate the numbering* for, for instance, sections and exercises. Make it easy to insert one, re-order things, etc.
- Make *writing texts* for instructions etc. easy. Usually paragraphs and bulleted lists are sufficient.
- When a text *references* an exercise ("*please do exercise 02*"), or shows the contents of a file, this must be kept consistent automatically.
- It must support using a *presentation design tool* (like Powerpoint).
- It must support using a *tool/environment* for doing the exercises (like oXygen).
- Participants get a package with the prepared exercises, additional PDFs, etc. *Create a ZIP file* for easy distribution.

The following figure illustrates what I mean with keeping a presentation automatically consistent:

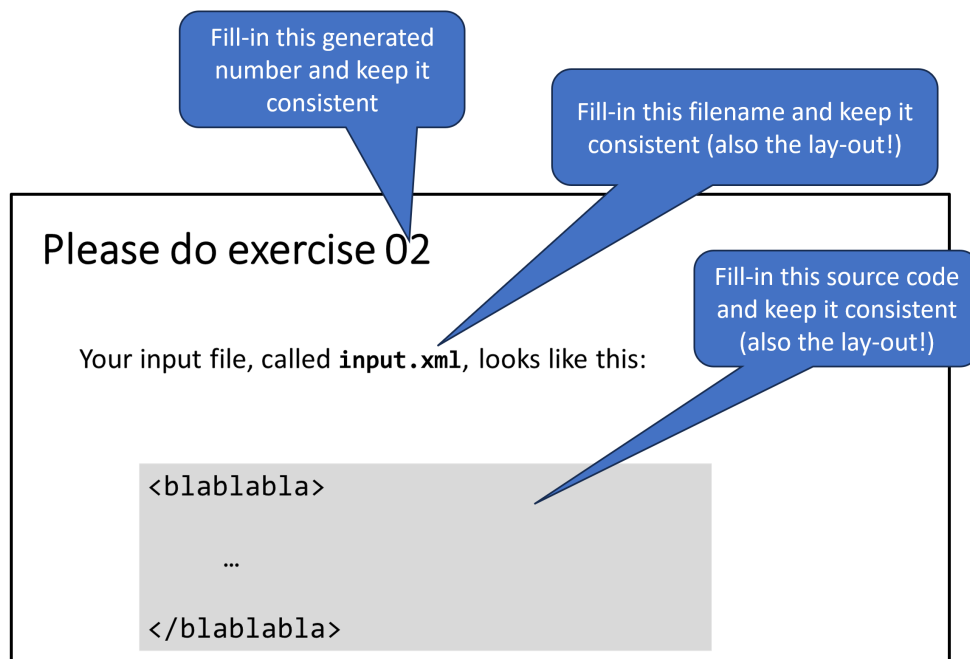


Figure 1. Requirements for automatically keeping a presentation consistent

And here is an example of a prepared oXygen project. The project view provides easy access to the exercises. Transformation scenarios are pre-defined, so people only have to click “transform” to make things happen.

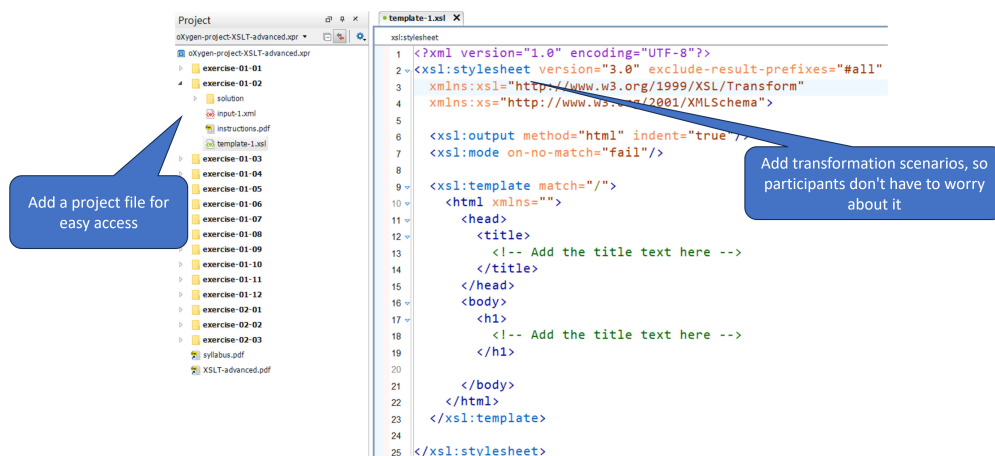


Figure 2. Example of the use of an oXygen project for a tutorial

3. Implementation using XML technology

An implementation, that uses XML technologies, looks like this:

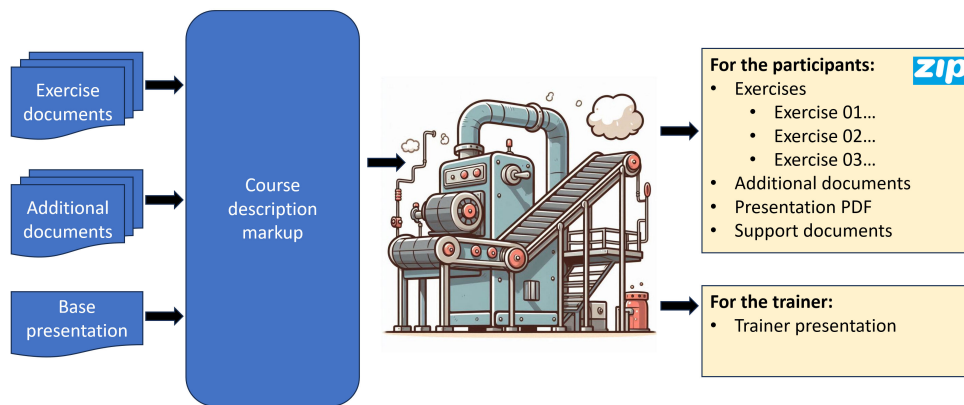


Figure 3. XML tutorial single sourcing/solution outline

- We need some kind of “*course/tutorial description markup*”. Something that ties all the separate components together.
- This markup will reference/use lots of external documents (exercise files, PDFs, presentation, etc.).
- We’re going to need some machine that takes this markup, processes it, and produces the required outputs:
 - A ZIP file with all the exercise files, usually the presentation in PDF, any additional documents, etc.
 - For the presenter a presentation where all the references to exercises and documents are filled in.

3.1. The course markup

Let’s have a look at a simplified version of the course/tutorial description markup for this:

```
<xcourse xmlns="http://www.xtpxlib.nl/ns/xcourse">

  <!-- Prologue (omitted): title, author, settings, etc. -->

  <section>
    <title>Matching templates in XSLT</title>

    <exercise id="matching-first-match">
      <title>Your first match template</title>

      <input-document id="input" href="exercise-match/in.xml"/>
      <template-document id="template" href="generic/empty.xml"/>
      <input-document id="solution" href="exercise-match/
solution.xml"/>
```

```
<instructions>
  Please open @input@:

  [[[input]]]

  Open @template and write a match template that ...
</instructions>

<oxygen-scenarios>
  <transform source-idref="input" stylesheet-idref="template"
    output-filename="build/results.xml"/>
  <transform source-idref="input" stylesheet-idref="solution"
    output-filename="build/results-solution.xml"/>
</oxygen-scenarios>

</exercise>

<!-- More exercises... -->

</section>

<!-- More sections... -->

</xcourse>
```

- I called it “xcourse”, so the root element is `<xcourse>` (in the `http://www.xtpxlib.nl/ns/xcourse` namespace).
- Then there is a prologue, which was omitted here to keep things simple. It contains information about the title, author, etc. It also contains a number of settings, like, for instance, how should exercises be numbered.
- Courses are made up of sections (`<section>` elements).
- Sections contain exercises (`<exercise>` elements).
- An exercise consists of:
 - A sequence of documents (inputs, templates and solutions) that belong to the exercise. Whatever the source filename, they get a consistent filename in the output. These files also have an (optional) identifier.
 - Exercises can contain texts. For instance, in the example above, an instruction for the exercise (`<instruction>` element). This one is written using (simple) Markdown. For more complicated texts you can use DocBook (5) also. Texts are converted to PDFs.

In these texts there are some references to the files in the exercise:

- An identifier between @ characters (for instance @input@) is replaced with the actual filename of the file in the exercise directory, for instance input-01.xml.
- A paragraph that contains an identifier between [[[and]]] (for instance [[[input]]]) will be replaced by a program-listing of the referenced file.

By the way: the syntax for representing the references was a, more or less, random choice. Other representations would also have been possible.

3.2. The processing engine

The processing engine was built using XProc 3.0. This programming language is very suitable for this application:

- XProc 3.0 is a language for document pipeline processing: in XProc you write steps (aka pipelines).
- It has a large number of standard steps as building blocks (XSLT processing, validation, XInclude, XSL-FO processing, etc.).
- It can handle (of course) XML documents, but also other kinds of files, like JSON or ZIP.
- It can do the necessary “housekeeping” surrounding an application, like deleting, creating and copying of files and directories.
- It’s (very) easy to re-use steps that you’ve written.

Everything runs on the MorganaXProc-III⁷ XProc processor.

I also had already a number of steps that came in handy implementing this:

- Markdown to DocBook
- DocBook to PDF
- Working with ZIP files

All these steps (and more) are available as open source components⁸.

3.2.1. Aside: Re-using steps in XProc

Once you’ve written an XProc step/pipeline, it’s easy to re-use it in some encompassing other step/pipeline, as is illustrated in the following figure:

⁷ <https://www.xml-project.com>

⁸ <https://www.xtpxlib.org/>

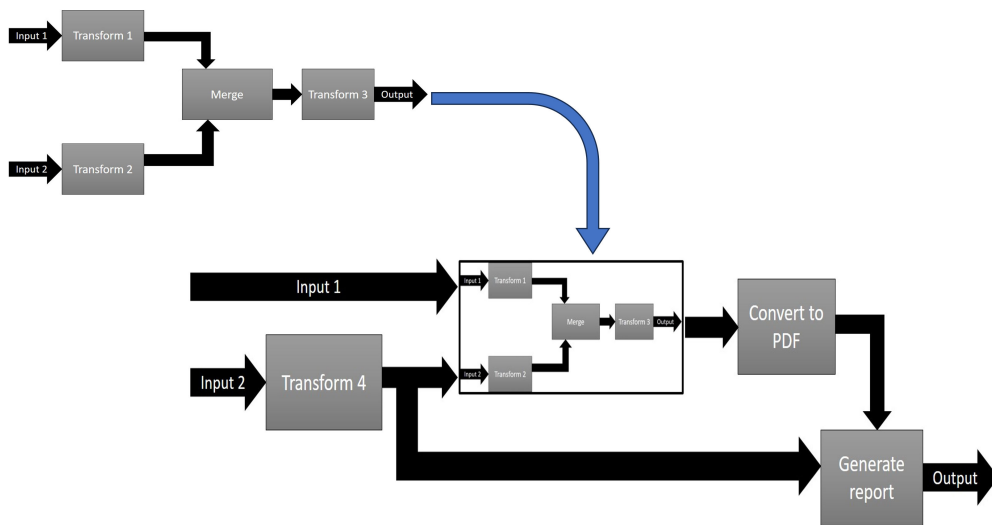


Figure 4. Re-using an XProc step

In code it looks like this:



Figure 5. Re-using an XProc step, code

3.3. Processing the markup

The following figure illustrates the different steps in processing the course markup:

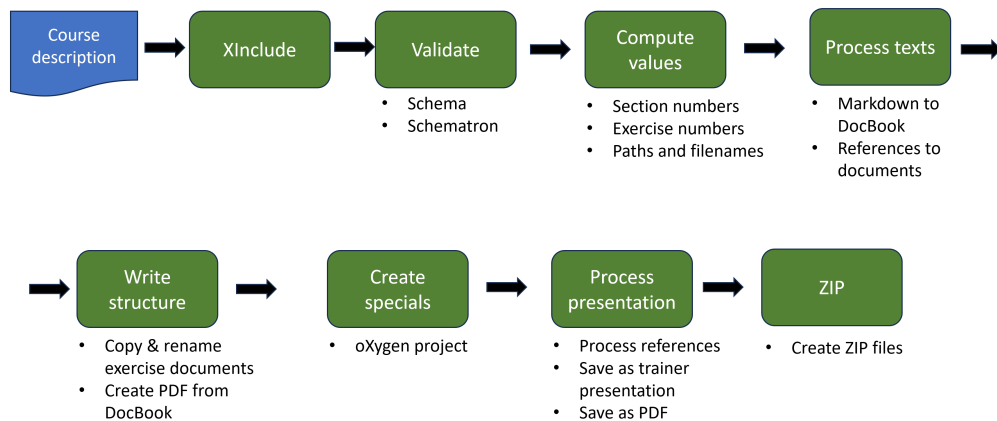


Figure 6. Processing the course markup

- The first thing that happens is run the course markup through XProc's XInclude step (`<p:xinclude>`). This de-references all `<xi:include href=...">` elements (recursively). Using XInclude enables segmenting the course markup in smaller documents (that can potentially be re-used in other courses). XProc makes doing XInclude *very* easy...
- Then the resulting document is validated against the course markup Schema and some additional rules using Schematron. Again, this is easy because XProc has standard steps for it.
- The next step is a very important one. Using XSLT, all kinds of values are computed: section and exercise numbers, directory and filenames, etc. This information is recorded in the course markup in additional attributes.
- Then the texts are processed. The Markdown is converted into DocBook. References (like `@input@` or `[[[input]]]`) are resolved. Converting from Markdown to DocBook to PDF is rather complex stuff, but the necessary XProc steps for this were already available, so I simply re-used them.
- Then the processing engine creates an output structure and copies all the files. The DocBook parts are written as PDF, again with already available XProc steps.
- The oXygen project is created.
oXygen's project files (`.xpr`) are XML documents in some proprietary format. You can inspect them and, with some trial and error, create something that defines your project structure and the necessary transformation/validation scenarios. A potential problem is that the document format is, AFAIK, not documented nor publicly defined. Just hope they will not change it anytime soon...
- The presentation is processed (see Section 3.3.1).
- And finally everything is zipped so it's easy to transport.

3.3.1. Processing the Powerpoint presentation

Microsoft Powerpoint stores its presentations in .pptx file. A .pptx file actually is a ZIP file with lots of heavily interrelated XML documents inside. Microsoft uses “Office Open XML” for this, a well-documented but complicated standard.

It’s a *very* complicated standard. Generating a Microsoft Office file from scratch (using some specification markup) is almost impossible, but modifying an existing one, just tweaking bits and pieces here and there, is very doable. Especially since, knowing it’s XML inside a ZIP, you can inspect what’s there easily.

Using a syntax as introduced before for text documents (@...@ and [[...]]), the following became possible:

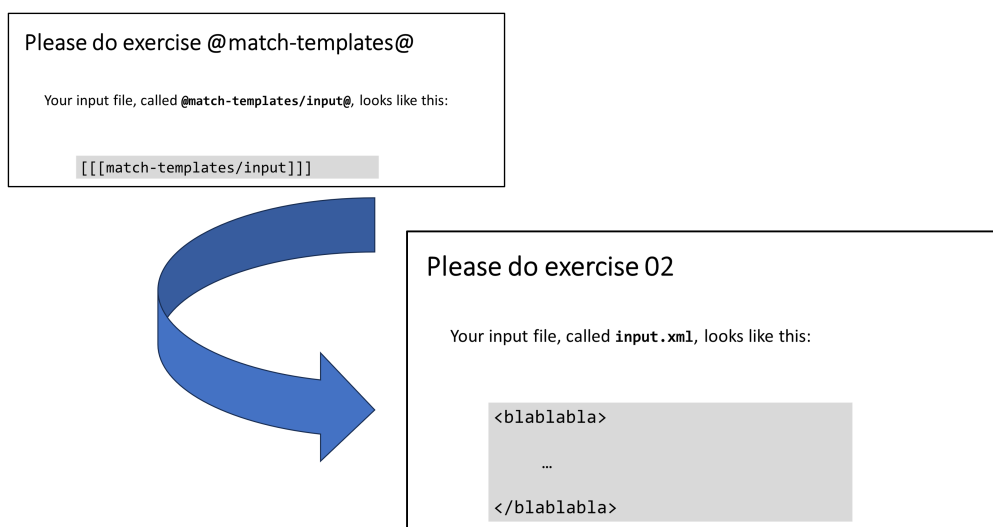


Figure 7. Powerpoint slide adaptations

This all can be done with an XProc step/pipeline (outside view):

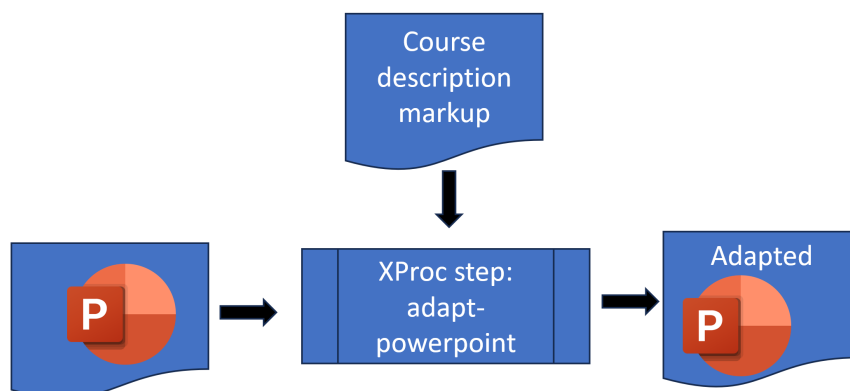


Figure 8. Powerpoint slide adaptations, pipeline (outside view)

Let's see how this is implemented here:

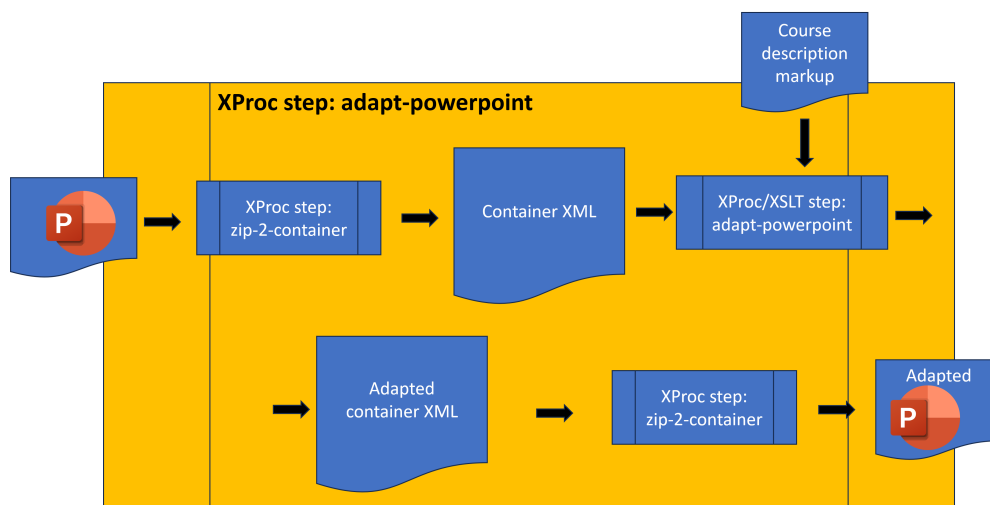


Figure 9. Powerpoint slide adaptations, pipeline (inside view)

- The .pptx file (which is a ZIP file), is turned into what I call “container” format (more details below).
- In a container, all XML from inside the ZIP is there, so we can now write XSLT to change the XML in anyway we want.
- Containers are designed for round-tripping, so after the changes we can put the .pptx ZIP file back together again. Everything we didn't touch (which is a lot) is still there, unchanged.

Working with the container format (using XProc 3.0) is open source code⁹.

There are of course more ways to do this. Handling ZIP files and changing parts of their contents is something XProc can easily handle. See for instance the following conference recording (Markup UK 2020): Dealing with ZIP Documents in XProc 3.0¹⁰.

3.4. Real world experiences, wrap-up and conclusions

Recently I had to teach some courses for which I already had materials that needed updating: the perfect use-case for all of this. Turning everything into the newfangled markup format was a bit of work. However, once that was done, the following things became apparent:

- For the tutorial developer:
 - Simple Markdown for instruction texts etc. is very easy to write and usually sufficient. Using Markdown lowers the psychological barrier for creating this kind of content.

⁹ <https://container.xtputlib.org/>

¹⁰ <https://www.youtube.com/watch?v=6yvO4GOue6k>

- Small mistakes (in exercise documents and texts) are still easily made, the devil is in the details. But rectifying these, consistently, everywhere, has become much easier.
- After a fix it's just a push of a button and a new distribution ZIP is created. No hassle, no fuzz.
- I can use my favourite presentation design tool (Powerpoint) *and* keep references and numbering consistent.
- For the participants/students:
 - All training modules have become nicely consistent in how to interpret and use, naming conventions, etc. People appreciate this because they can keep their focus on what's being taught.
 - The generated oXygen project is much appreciated, for the same reason.

So, as general conclusions and wrap-up:

- Yes, this approach helps, really well. Both for the tutorial developer and the participants/students.
- Single sourcing is always a good idea, proven once and again.
- XProc seems the almost ideal "mash-up language" for XML processing. If you design your steps carefully it's very easy to re-use them in another context.

The implementation I used was a first version and it left me with a long list of ideas. The second version is currently work in progress. Once done I'm planning to open-source it.

Modern Benchmarking of XQuery and XML Databases

Alan Paxton

Evolved Binary

<alan@evolvedbinary.com>

Adam Retter

Evolved Binary

<adam@evolvedbinary.com>

Keywords: Benchmark, Performance, XML Database, XQuery, NoSQL

1. Abstract

The performance of XML Databases and associated XQuery implementations is dependent on many factors, not least:

- The structure of the document(s)
- The number and size of documents (a large number of small documents vs a few large documents)
- The distribution of data between documents
- The queries (workload) under use.

Previous work in XML benchmarking has focused on generating realistic synthetic XML data, and testing that data with an appropriately tailored set of queries.

XML Databases can be viewed as a very specific kind of NoSQL database. Recent advances in large distributed NoSQL databases have demanded and produced strong general-purpose benchmarking frameworks, including tools for generating and using “virtual data sets”.

Here we review previous approaches to benchmarking XQuery and XML Databases, and examine current approaches to NoSQL Database benchmarking. We show how ideas from the broader NoSQL world can be used to replicate, generalise and improve upon existing XML benchmarking, and we assess the tools and techniques we develop.

2. Introduction

XML storage systems can store XML documents as plain old files, catalogue them through some kind of third-party system (e.g. a relational database, or text

index), or store them in a native XML database which breaks them down into basic components of the structured document and organises these for efficient access and update [1]. In any case, the standard is to access the contents using XQuery (XML Query language) [2].

Information may be modelled in XML and distributed across one or many documents. The distribution choice can affect performance, but perhaps more importantly all aspects of the structure (deep vs shallow, child element fanout, element content size) and the variability within the structure can have profound effects on the ultimate performance of an application.

Previous work in XML benchmarking has dealt with this performance variability issue by positing a document or set of documents which are implied to be adequate to exercise the whole range of performance of an XML storage system. This is generally achieved either by using a range of pre-existing real document corpuses, or more commonly by generating a synthetic document or documents which contain a variety of fragments which are claimed to be realistic in both scale and structure. An XML benchmark typically consists of:

1. Data input. Most frequently this is generated by a tool, parameterised in a limited number of dimensions. Sometimes the data input is simply a fixed document corpus.
2. A set of prescribed queries, in XQuery, to be issued against the document store.

More recently, there has been a significant upsurge of interest in NoSQL databases, which have grown up in response to the difficulties encountered applying traditional relational databases to less rigidly structured, and often very large, data sets. Additionally, NoSQL databases have been promoted as having superior performance in a range of workloads. As these databases have been adopted broadly, at scale, and often in very distributed installations, the NoSQL community has found a pressing need for testing and performance frameworks which can exercise and validate the features and performance of their databases.

These requirements have yielded very powerful and general purpose “pluggable” benchmarking frameworks that can be used to flexibly generate workloads and drive highly concurrent benchmarks over multiple different products.

In *Section 3* we first review the previous approaches to benchmarking XML Databases and XQuery implementations. Then we move on in *Section 3.3* to examine the current approaches to NoSQL database benchmarking. We discover in NoSQLBench a flexible framework for driving the execution of localised and distributed testing of databases. We therefore propose an XML database adapter for NoSQLBench; the framework is built with a number of existing adapters and it is natural to extend this to XML. This gives us a powerful tool to co-ordinate general NoSQL-style benchmarks over XML databases via XQueries encapsulated in NoSQL benchmarking scripts.

Further, we find that the “virtual data generation” facilities in some NoSQL benchmarking frameworks are highly adaptable, and we borrow them to rebuild existing synthetic XML data in a structured manner. This opens the possibility of generating more and different synthetic datasets targeted to the detailed measurement and analysis of specific XML database features.

3. Related Work

3.1. XML Benchmarking

XML Benchmarking has existed for almost as long as the XML standard itself has existed. X007 [6] was an initial pragmatic attempt to translate the 007 benchmark to a number of nascent XML databases (or information retrieval systems).

XML benchmarks, like database benchmarks in general, are often broken down into application-level benchmarks and micro-benchmarks. Application-level benchmarks are intended to provide a broad characterisation of performance over a realistic workload, by analysing the performance of a range of queries over a range of documents. Microbenchmarks focus on specific performance aspects of a system, and may rely on very specific (though not necessarily small) documents/data and individual queries targeted at these parts of data to target performance measurement of isolated components of the database system.

The application-level XML Benchmark Project [3] [4], XMark for short, took a systematic approach to generating test data for XML databases and to measuring database performance with a prescribed set of queries. Its XML document generator, *xmlgen*, builds an XML file (or optionally a set of files) which describe an online auction. The files contain several different structures with different characteristics (nesting depth, text size, indexing and cross-referencing) which along with the queries exercise a broad range of XML database characteristics. Although released in 2002, *xmlgen*, which is written in portable C89 (ANSI C) is still buildable and runnable today, and the range of structures it can generate has had a strong influence on what constitutes a realistic set of XML data for benchmarking.

XBench [7] is another application-level benchmark. It outlines a diverse set of XML benchmarking requirements and emphasises the importance of update performance in realistic workloads. It characterises workloads as document-centric or text-centric, and distinguishes between multiple-document and single-document data sets. It then applies a broad range of queries to the database, so that an understanding of which areas are strengths and which are weaknesses can be acquired. Having defined its workloads along these axes, schemas are derived for the ToXGene tool ??? which is used to generate concrete test documents.

The Michigan Benchmark [8] is a micro-benchmark intended for drilling into detailed performance aspects of XML databases. It takes the interesting approach

of generating only a single dataset for this, but structured in a way where different levels have different numbers of nodes and/or different fanouts. For example, in the base case of the Michigan document:

- Level 7 has 2704 nodes, with fanout 13 (each node has 13 children)
- Each node is given an appropriate *level* attribute.
- A query which selects for level=7 can be compared for one which selects for level=9 and any difference can be attributed to fanout.

The structure of the test data therefore allows the effect of a single characteristic (fanout, node count) to be isolated. While such a data set is not characteristic of a typical application workload, it serves the precise purpose of microbenchmarking, by isolating individual characteristics which shed light on details of a data store implementation.

The Michigan Benchmark proposed a number of queries focused on:

- Approximate matching
- Order sensitivity
- Structural selection
- Traditional (SQL-esque) joins
- Aggregation
- Update (in 7 different forms)
- Document construction

The Michigan Benchmark was used to measure 3 different databases:

- Timber (an XML Database under development at Michigan)
- An anonymous commercial XML DB
- An anonymous commercial OR-DBMS (Object Relational Database Management System)

It was able to measure and demonstrate that the 3 XML database systems under test had very different profiles for the different query types, and identified detailed areas for immediate performance improvement in the Timber XML database [9] under development at that time at The University of Michigan.

XQBench [16] the useful and interesting perspective of building out a standardised environment for the execution of XML benchmarks, allowing an objective comparison of XML Systems/Databases within the parameters of the chosen system.

The subject of XSLT processing and its performance is worth mentioning. XSLT is a powerful XML-based specification for transforming documents based on stylesheets. There is crossover between the use of XSLT and XQuery, and work on XSLT performance with the XT-Speedo benchmark therefore has some bearing on the understanding of XML performance [17]

3.2. Automated XML Data Generation

A constant strand running through XML benchmarking is the problem of acquiring or generating sufficient input data with the right structure to perform realistic benchmarking with.

The *xmlgen* tool which is part of XMark [3] is designed to efficiently generate a single, well-defined XML hierarchy, either within one single XML file or distributed across multiple files; the latter case was originally envisaged for systems where loading or processing a single very large file is difficult, but also can be revealing about parallel/concurrent document access performance. The *xmlgen* output consists of an online auction site, and catalogues the major components of such a site as subtrees of:

- Items (available for sale)
- Categories (to which items belong)
- People (participants in the auctions)
- Open (in progress) auctions of particular items
- Closed (completed) auctions of items

The subtree elements intentionally differ between classes, so that some (categories) are simple and shallow, but hold significant text elements, while others (items, open auctions) are more structurally deep and complex. The sum of them approximates a full spectrum of XML document features. Elements are also cross-referenced, for example open auctions refer to the items being auctioned and to the people involved (as sellers or bidders).

xmlgen data generation can be scaled when the generator is invoked; the default (scale 1) output file is 116MB in size, and it is easy and (fairly) quick to generate a file of any scale (e.g. 10 or 100 times large). The program is written to include its own custom random number/distribution generation, and the output is entirely reproducible in all environments in which the program (written in C89 (ANSI C) for high portability) can be run.

xmlgen has been very successful in the context of the XMark benchmark, where it supports easily running the benchmark on any platform through the re-creation of the auction XML files.

The obvious significant restriction on data produced by *xmlgen* is that only a single Auction dataset can be constructed. While this dataset is designed to be all things, it is still an obvious restriction. Later projects have attempted to address this shortcoming in a number of ways.

The ToXGene [5] XML generation tool is designed to flexibly generate XML of the size, structure, and content required. The required data is specified within an extension to XML Schema, and generation will take account of schema attributes such as `minOccurs` and `maxOccurs` to specify the number of elements (or range of elements) to be generated. There are also control structures in the ToXGene ele-

ment specifications which allow some generation of irregular and/or random structure. The authors used ToXGene to generate a copy of the XMark auction dataset, and demonstrated that the resulting data was extremely similar in terms of results and performance when benchmarked with the XMark queries. They also simulated the generation of data for the TPC-H benchmark [19]

While ToXGene can be used to generate fairly sophisticated documents, the XBeGene project [11] reduces the learning curve by instead basing its generation on example documents. Given appropriate example input document(s), much larger documents exhibiting the same structures can be generated. XBeGene performs feature extraction on input documents to identify element path occurrence frequencies and content (of element or attribute) distribution frequencies. It then builds output documents by regenerating paths and values pseudo-randomly to approach the frequencies displayed by the input documents. The authors confirmed the correctness of their approach through a similarity analysis which categorised test documents into clusters generated from original documents; XBeGene generated documents were clustered very accurately with the input documents from which they were generated. From our perspective, however, it would have been illuminating to learn whether XBeGene documents could be used in (for instance) the XMark benchmark, if generated from small instances of XMark data.

3.3. NoSQL Benchmarking in the Large

OLTP-Bench [10] provides a well-argued analysis of the need for a comprehensive benchmarking testbed for relational databases. The authors argue that such a testbed needs to control the data set and workload for testing, to automate the extraction of metrics from test runs, and to ensure these runs are repeatable.

The principles put forward by OLTP-Bench are the same as those that should be applied to NoSQL database benchmarking, and therefore also to XML database benchmarking. Within the domain of NoSQL databases, the YCSB project [15] built a flexible framework for benchmarking cloud databases with a defined (but easily configurable and extensible) set of workloads. The more recent NoSQLBench [12] aims to provide a comprehensive testbed for NoSQL. As the project's own README file describes it:

- You can run common testing workloads directly from the command line. You can start doing this within 5 minutes of reading this.
- You can generate virtual data sets of arbitrary size, with deterministic data and statistically shaped values.
- You can design custom workloads that emulate your application, contained in a single file, based on statement templates - no IDE or coding required.

As with OLTP-Bench and YCSB, the architecture of NoSQLBench is pluggable; while it was designed initially for testing Apache Cassandra [14], and therefore supports a CQL (Cassandra Query Language) interface, the database interface element (the adapter) is pluggable, and a number of other adapters have already been developed and made available.

Further, the VDS (Virtual Data Set) feature of NoSQLBench is very powerful. Within a workload configuration, one may simply declare which of a broad set of VDS functions and their compositions to use to generate values which will be substituted into various operations as they are executed; NoSQLBench takes care of making the VDS value generation and substitution maximally pre-computed and efficient, so that each operation is performed with minimal overhead from the benchmark framework.

As a “battle tested” [13], NoSQLBench takes care of the coordination and dispatch of operations within workloads, selected at the desired ratio from the specified range of operations and initiated over a defined number of threads.

4. XML Generation and Testing in the NoSQLBench Framework

We have seen how benchmarking of SQL and NoSQL has benefitted from the development of overarching benchmark frameworks which deal with:

1. Test data generation/loading
2. Benchmark setup, query ramp-up, run and teardown
3. Metric collection

We want to bring these benefits into the XML benchmark domain, and it would appear that the most natural way to achieve this is to adapt an existing benchmark framework so that it can work effectively with XML data stores.

4.1. A Testbed for XML Database Testing - NoSQLBench XML:DB API Adapter

The first part of our solution was to develop a NoSQLBench adapter for the XML:DB API [24] XML:DB API is database agnostic, and may be used to access any XML database that supports it. As a proof-of-concept we have built an XML:DB adapter that is compatible with [20], [21] and [22]

The XML:DB API adapter implementation is straightforward. Both NoSQLBench and its adapters are written in the Java programming language. The current NoSQLBench release (5.17) requires Java 17, and the current development branch for the forthcoming (5.21) release requires Java 21. We implemented our adapter as a new Maven module within the NoSQLBench codebase. There are several compulsory classes to create/load an adapter and have it build and execute operations that NoSQLBench requires. Our fork of NoSQLBench with our

new XML:DB API adapter is at <https://github.com/evolvedbinary/nosqlbench>. The adapter supports compulsory parameters defining how to connect to the XML database endpoint:

- URL

It also supports optional parameters (with standard defaults) for:

- username
- password
- collection

Given these, the adapter can be used to run an example NoSQLBench YAML demo script which implements a schema phase and a write phase; the schema phase deletes and re-creates the database collection(s) and generates and inserts data. The script is https://github.com/evolvedbinary/nosqlbench/blob/master/src/main/resources/activities/existdb_crud_basic.yaml and it is run by executing

```
java -jar ./nb5/target/nb5.jar ./adapter-existdb/target/classes/activities/existdb_crud_basic.yaml
```

. This runs the demo script with the default configuration values that connects to an XML database instance running on the local host. The schema and write operations of our NoSQLBench XML:DB script:

```
scenarios:
  default:
    #
    schema:  run driver=existdb tags==block:"schema.*" threads==1
              cycles==UNDEF
              endpoint=xmldb:exist://localhost:8080/exist/xmlrpc

    # different ops in the "write" schema occur on intervals, at the
    ratios declared by the ops
    write:   run driver=existdb tags==block:"write.*"
              cycles===TEMPLATE(write-cycles,TEMPLATE(docscount,1000))
seq=interval threads=auto errors=timer,warn
              endpoint=xmldb:exist://localhost:8080/exist/xmlrpc
```

The default NoSQLBench scenario in this script runs 2 blocks; the *schema* block and the *write* block. The schema is used to re-initialise the database (the schema) before writing. Its cycle count is undefined, so that it runs only once, carrying out one-off deletion and (re)creation of the collection used for the benchmark:


```
blocks:
  schema:
    ops:
      drop-collection:
        xquery version "3.1";

        import module namespace xmldb="http://exist-db.org/xquery/xmldb";

        let $clean := if (fn:contains(xmldb:get-child-collections("/{collection}"), "testnb"))
        then
          xmldb:remove("/{collection}/testnb")
        else
          ()
        return $clean
```

The sample above shows the deletion part. Within the script, we can see that the binding for *collection* is substituted into text of the query which will be sent to the adapter.

The write block contains 2 write operation templates. Both carry out similar functions; they just write slightly different elements at different rates to illustrate the process within NoSQLBench. This write block

```
xquery version "3.1";

import module namespace xmldb="http://exist-db.org/xquery/xmldb";

declare namespace ppl = "https://evolvedbinary.co.uk/people/";

let $record :=
  <person id="{user_id}" seq="{seq_key}" creation="{created_on}">
    <name>{full_name}</name>
    <family>
      <gender>{gender}</gender>
      <married>{married}</married>
    </family>
    <address>
      <city>{city}</city>
      <country>{country_code}</country>
      <location lat="{lat}" lng="{lng}" />
    </address>
    <text>{text}</text>
  </person>
```

```
return xmldb:store("/{collection}/testnb/alpha", "{random_key}",
$record)
```

creates a *people* record in its own file. In particular note that it substitutes a number of NoSQLBench bindings (*{city}*, *{full_name}*, *{text}* etc) into the operation. Because the bindings are a function of the cycle (roughly, cycle=*n*, for the *n*-th operation being generated), each element created has different element and attribute values at places containing bindings. Thus we can perform as many instances of the operation as we choose, and all will have different parameter instances. Looking at the generated XML, we can see that we have created the files

```
<person id="0d0d165a-9978-4a04-94e2-459c729cea8d" seq="905"
creation="1294444148">
  <name>Terra Barile</name>
  <family>
    <gender>M</gender>
    <married>>false</married>
  </family>
  <address>
    <city>Powderly</city>
    <country>BA</country>
    <location lat="-143.33042469046987" lng="-129.56662632758295"/>
  </address>
  <text>"What?" asked the commander.</text>
</person>
```

and

```
<person id="0d53d1e5-39f2-40e9-a0ee-16b33e3d5f13" seq="900"
creation="1295124573">
  <name>Sydney Hruby</name>
  <family>
    <gender>M</gender>
    <married>true</married>
  </family>
  <address>
    <city>Clarksville</city>
    <country>BB</country>
    <location lat="-142.55411040458398" lng="157.01567078043186"/>
  </address>
  <text>Dólokhov slowly straightened his bent knee, looking straight
with his
clear, insolent eyes in the general's face.</text>
```

</person>

from the same *write-node-alpha* operation.

As for performance, in a run of this script on an Apple MacBook Pro with an M1 Max CPU and 64MB RAM, write cycle service times (the most relevant metric reported by NoSQLBench) have a mean of 38ms (standard deviation 32ms), with 100 concurrent threads running. This is a number which we would like to drill into further, but our strong suspicion is that a large part of this is the cost of serialising/deserialising data over the network (of the host) using XML-RPC via the XML:DB API, and repeated parsing of queries. There are lots of potential solutions to this; the easiest first step would be to switch to an embedded implementation of the XML:DB API, whereby the XML database and NoSQLBench run within the same JVM (Java Virtual Machine). However, we would first like to drill into whether we can use our driver to distinguish the time spent performing data storage operations within the database, rather than the time spent serialising/deserialising requests. It may be possible to start internal timers and return their values via XML:DB which we can then return in other metrics. In any event, non-query costs may be less important when we come to test complex and slow queries, serialisation/deserialisation overhead may then be only a small fraction of total time.

Where the performance under the test framework is problematic, though, is if we wish to use it to generate XML test data. We considered the idea of populating a database by generating NoSQLBench operations to build the entire data set under test; this idea can be seen to live in the example we have used to demonstrate our adapter. But some experimentation showed that the performance here would mean that it took many minutes or hours to populate a database of a realistic size for testing. The overhead cost of substituting each of millions of insertion query instances and passing them through the XML:DB API is probably the reason. Even were we to create an embedded implementation of the XML:DBAPI and use that for each entry generated, we still doubt that this would perform optimally.

Therefore we looked at whether we could find another way to exploit the power of VDS within the testbed, so that we could use it to generate XML test data more efficiently. We describe our solution in Section *Section 4.2*

4.2. XML Data Generation in a Benchmark Framework - using NoSQLBench Virtual Data Sets

We have discussed several previous projects which have created tools to generate XML data. Our hypothesis is that the VDS (virtual data set) functionality, which is part of NoSQLBench, is powerful enough that we can define scenarios for generating realistic synthetic XML test data with it. To test our hypothesis we attempt

ted to generate a close approximation of the *xmlgen* auction site that is generated by XMark [3].

Our first attempt at generation involved working entirely within the NoSQL-Bench scripting framework. The YAML scripting format contains and supports JSON, so it is possible to write operations using JSON containing binding substitutions which are parsed by the framework; if we write an “xmlgen adapter” for NoSQLBench, then running the framework targeting this adapter will result in it being called with parsed and substituted structures of maps and arrays which can then readily be serialised to an XML format. This takes the following form for one of the simpler parts of the auction site, the generation of categories. Categories just require a single binding:

```
bindings:
  category_name: Add(100); ListSizedHashed(Zipf(100,1.4),
CountryNames()); Join(' ')
```

Notice how we move the base for this binding generation by 100 so that it does not work in step with any other bindings used for the same operation. Then the VDS function for the category name

- creates a list which is usually (but not always) short
- generates the country name from the hash value for each item in the list
- concatenates them together to produce a simulated name like “Spain Andorra Marshall Islands” for the category.

Our operation defined for categories looks like this:

```
genauctioncategories:
ops:
  generate-category:
    file: "{xml_file_id}"
    path: ["categories", "category"]
    __attrs: {
      id: "category{{Identity()}}"
    }
    __children: {
      name: "{category_name}",
      description: {
        text: "{{MobyDick()}}"
      }
    }
  op: >2
    "generate category op - required by the system - we don't use"
```

it"

This makes use of the VDS functions *Identity()* and *MobyDick()* directly with the *op*, as well as the more complex binding *category_name* which we have just described. There are several other points, mostly that our *xmlgen* driver makes a special case of certain JSON map keys:

- *file* - we concurrently send XML data to multiple output files.
- *__attrs* - we use this at any level to indicate that the following map contains (key, value)-pairs that should be used as attributes rather than sub-elements.
- *__children* - for consistency, we are explicit about using a map as the children to create. This is however the implicit default; (key,value)-pairs go into the children map implicitly.

For the simplest case in the auction site, that of the `<category>` elements, the extensions we have provided are sufficient to allow us to generate elements which have the same form and distribution as those generated by the original *xmlgen* program from XMark.

However, problems begin to appear when we set out to generate more complex elements of the auction site, specifically `<item>` and `<person>` elements. These contain groups of sub-elements with dependencies on the synthetic values. This leads us to add another special map key:

- *__foreach* - introduces a parameter which takes a value from a VDS list, and substitutes it into a map, adding a little bit of substitution syntax.

```
watches: {
  # a list of <watch>, each with an open_auction attribute
  watch: {
    __foreach: [auction_val, "{auction_ref}"],
    __attrs: {
      open_auction: "open_auction[auction_val]"
    }
  }
}
```

which generates an element like this

```
<watches>
  <watch open_auction="open_auction2888"/>
  <watch open_auction="open_auction7264"/>
  <watch open_auction="open_auction2980"/>
  <watch open_auction="open_auction4928"/>
```

</watches>

This is already becoming complex, and next we encounter the `<open_auction>` element which contains sub-elements representing each bid in the auction. These have numerous complex relationships; the bids have dates which must be in order, and the bids record increments on the previous bid which must add up to the current bid value, which is another field of the `<open_auction>`. When we tried to address this complexity, we again found that we would need to add further special-case constructs to the JSON content beyond those (`__children`, `__attrs`, `__foreach`) which we had already added. We require specifically to generate a `<current>` bid field which contains the arithmetic sum of each of a sequence of `<bid>` fields. The amount of work that would be required to achieve this appears prohibitive, and the natural solution to the complexity of the needed extensions was to switch from the YAML scripting language to a full-fledged programming language; better allowing us to control the flow and relationships between the parts of XML being generated and their values.

4.2.1. A Standalone XML Generation Tool using VDS

Recall that the NoSQLBench framework is written in Java. In fact, the VDS services within it were once a standalone package of their own, and while they have been rolled into the greater NoSQLBench, they still present a clean, general, and very powerful functional library for efficient data generation.

The step we needed to take, then, was to use VDS as a Java library directly and to supplement that with support (and syntactic sugar) around XML output so that while we would be writing programs for XML generation rather than scripts, the programs would still be easily comprehensible and modifiable.

We have prototyped a Java tool which uses the VDS libraries and generates XML elements, attributes and text. This tool allows us to perform XML generation of the auction site in a straightforward manner. Our tool could be re-integrated into NoSQLBench as a step in a workload scenario, however it does not take the same form as an adapter (as defined by NoSQLBench).

As an example of how the tool appears, to generate the `<people>` elements of an online auction we write a function like:

```
protected void person(final long i) {
    seed(
        attribute("id", "person" + i);
        attribute("city", citie
        element("name", fullNames);
        element("emailaddress", email);
        element("phone", phone_num);
        element("homepage", homepage);
```

```

element("creditcard", cc_num);
element("address", this::address);
if (next(optionalProfile) > 0) {
    element("profile", this::profile);

    var auctionRefs = next(auctionRef);
    if (!auctionRefs.isEmpty()) element("watches", () -> {
        for (var auctionRef : auctionRefs) {
            element("watch", () -> attribute("open_auction",
"open_auction" + auctionRef));
        }
    });
}
}

```

The `person` method is declared in a *PersonBuilder* subclass of *ElementBuilder*, and that class declares a suite of builder methods by which nodes are added to the XML tree in construction.

The core is the *element()* method calls, these generate an element with a value generated by a VDS function (*element("name", fullNames)*) or “open” an element by passing a lambda which generates the subnodes of the element (*element("watch", () -> attribute("open_auction", "open_auction" + auctionRef))*). The VDS elements used by the builder methods are declared as class variables of *PersonBuilder*. A *next()* method is provided which applies the VDS generator function at the current cycle index, and returns the result. The *element()* method can apply this implicitly in most cases.

Builders can cooperate by sharing a *BuildContext*, which holds the stack of elements being processed. In this way, the *PeopleBuilder* can use a *PeopleBuilder* which in turn uses a *TextBuilder*, which knows how to build the `<text>` element of recursively marked up text used across the elements of the auction site.

The *ElementBuilder* contains a *BuildContext* interface. We have written an implementation of *BuildContext*, using *XMLStreamWriter*, as a convenient way to stream low-level XML elements, attributes, and text nodes to a string format.

The code for the standalone XML generator is currently under development and is available on GitHub: <https://github.com/evolvedbinary/xmlgen2.git>.

4.3. Project Status

4.3.1. XML:DB Adapter

The adapter is functionally complete and has been used to run NoSQLBench scripts against FusionDB, Elemental, and eXist-db; it should be possible with little, if any, changes to do so against other systems that support the XML:DB API.

The key issue we face at present is the absence of realistic test data; while we can successfully use *xmlgen* from the original XMark project, we have chosen to

complete our own XML generation tooling first, as we believe it can be much more flexible in allowing us to run tests other than the XMark queries.

We are confident that the metrics available in NoSQLBench will allow us to effectively measure and drill into the performance of specific XML queries and benchmarks.

4.3.2. Standalone XML Generation Status

We consider our new Java XML generation tool (*xmlgen2*) to be functionally complete; we have used it to generate all of the elements in the XMark auction site, and we are able to simulate the distribution of elements with varying number and sizes of attributes, sub-elements, and text nodes as accurately as necessary (by inspection).

With the current state of our *xmlgen2*, we have compared its performance against the original *xmlgen* using various scales of the data set. The following results were obtained on an Apple MacBook Pro with an M1 Max CPU and 64MB RAM: For the current state of implementation of the new generator, performance has been compared against the original *xmlgen*-program using various scales of database. We see the following on our 64MB M1 Max Macbook Pro:

Table 1. XML generation performance

Scale 100	<i>xmlgen</i> (C89)	<i>xmlgen 2</i> (Java 21)
creation	221s	420s
size	11758MB	13235MB

The size of files created is consistent; our new generator creates text elements as random paragraphs from a book in the Gutenberg Project, and the text parts turn out a little larger than those from the existing *xmlgen*

Performance of creation is adequate; first of all, note that we are working with scale 100 of the database, which is probably much more than adequate for many tests. Second, note that the current application is a single thread. We anticipate Java performance optimization, particularly exploiting multi-threading on modern hardware, or even utilising Graal [23] compile to native code, will bring the performance deficit down significantly.

The output of the auction site generated by our new *xmlgen2* tool will not be identical to that generated by the original *xmlgen* tool. However the creation of a generated dataset is of course repeatable which is important so that others can replicate benchmarking results. We prefer to rely on query test performance from the XMark suite to confirm that queries against the database built from ingestion of each of the versions performs equivalently.

4.3.3. Future Work

We have reviewed the landscape of XML database and XQuery benchmarking, and outlined our strategy for enabling easier and more comprehensive benchmarking. Our aim is to achieve this by adopting and adapting the NoSQLBench framework to an XML context. We will do this with an XML:DB API NoSQLBench adapter and a tool for flexibly generating synthetic XML data sets using the VDS features of NoSQLBench.

The direction of future work is already partly determined by our choice of framework, and our identification of the XMark online auction data set as the exemplary data set for the purposes of data set generation.

We aim to reach the point where we can completely validate that XMark testing can be accurately replicated with our system. We plan to:

- Integrate generation and load of test data into a NoSQLBench round trip; perhaps as a new NoSQLBench adapter with a single operation which configures and runs the generation tool / code.
- Add a facility for loading an existing static XML corpus into a database as a phase of a NoSQLBench workflow, to enable tests which use existing static corpora as the basis of their operations.
- Validate our initial intuition that the `xmlgen2` generated data sets are effectively equivalent to `xmlgen` generated data sets, through application of test queries from the XMark benchmark.
- Make implementation improvements to the XML data set generation tool: Separate the auction site from the framework Build other XML examples for completeness.
- Target the testbed at other aspects of XML databases, including concurrent querying, contention, update-heavy testing. We have several reports of performance problems that XML database users have experienced with `eXist-db`; we would like to take these reports and turn them into testbed scripts.
- Investigate the loading of data into XML databases; if we modify our framework to generate multiple small files rather than one large one, much as `xmlgen` does, we can use the built in concurrency of NoSQLBench to measure how well databases can ingest files concurrently.

5. Conclusion

We have reviewed the state-of-the-art in XML Database and XQuery testing. We found that the relative scarcity of sufficiently large and complex corpora, and the demand for complex test data has spurred the development of synthetic data set generation using tools such as *xmlgen*, *ToXGene*, and *XBeGene*.

Advances in database testing frameworks in the SQL and NoSQL domains have shown a path forward for XML Database and XQuery testing. Such frameworks offer a vision of easier, more comprehensive, and automated testing.

We have built a NoSQLBench adapter for the XML:DB API to integrate XML testing into a framework ecosystem. We have completed a modernised and general rebuild of *xmlgen* using NoSQLBench's VDS services, and we have shown that we can replicate and generalise *xmlgen* within the NoSQLBench framework.

Under NoSQLBench we now have a more complete XML performance test-bed, which we hope to apply to a broad range of XML databases and XML processing performance analysis.

Bibliography

- [1] *A Brief History of XML*. Chris Collins. 2008. <https://ccollins.wordpress.com/2008/03/03/a-brief-history-of-xml/> .
- [2] *XQuery 3.1: An XML Query Language*. 2017. <https://www.w3.org/TR/xquery-31/> . Jonathan Robie, Michael Dyck, and Josh Spiegel.
- [3] *XMark: A benchmark for XML data management*. Elsevier. 974--985. 2002. VLDB'02: Proceedings of the 28th International Conference on Very Large Databases. Albrecht Schmidt, Florian Waas, Martin Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse.
- [4] *A look back on the XML Benchmark project*. Springer. 263--278. 2003. Intelligent Search on XML Data: Applications, Languages, Models, Implementations, and Benchmarks. Albrecht Schmidt, Florian Waas, Martin Kersten, and Stefan Manegold.
- [5] *ToXgene: a template-based data generator for XML*. ACM. 2002. 616--616. Proceedings of the 2002 ACM SIGMOD international conference on Management of data. Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons.
- [6] *XOO7: applying OO7 benchmark to XML query processing tool*. 2001. 167--174. Proceedings of the tenth international conference on Information and knowledge management. Ying Guang Li, Stéphane Bressan, Gillian Dobbie, Zoé Lacroix, Mong Li Lee, Ullas Nambiar, and Bimlesh Wadhwa.
- [7] *Xbench-a family of benchmarks for xml dbmss*. 2002. 162--164. Springer. Workshop on Data Integration over the Web. Benjamin B. Yao, M Tamer Özsu, and John Keenleyside.
- [8] *The Michigan benchmark: towards XML query performance diagnostics*. 2006. Elsevier. 73--97. Information Systems. 31. 2. Kanda Runapongsa, Jignesh M. Patel, H.V. Jagadish, Yun Chen, and Shurug Al-Khalifa.

- [9] *Timber: A native xml database*. The VLDB journal. 11. 2002. Springer. 274--291. Hosagrahar V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V.S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and .
- [10] *Oltp-bench: An extensible testbed for benchmarking relational databases*. VLDB Endowment. 2013. 277--288. Proceedings of the VLDB Endowment. 7. 4. Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux.
- [11] *XBeGene: scalable XML documents generator by example based on real data*. Springer. 2013. 449--460. Recent Progress in Data Engineering and Internet Technology. 1. Manami Harazaki, Joe Tekli, Shohei Yokoyama, Naoki Fukuta, Richard Chbeir, and Hiroshi Ishikawa.
- [12] *NoSQLBench - An open source, pluggable, NoSQL benchmarking suite*. Jonathan Shook. 2020. <https://www.datastax.com/blog/nosqlbench> .
- [13] *NoSQLBench v5*. NoSQLBench. 2024. <https://docs.nosqlbench.io/introduction/> .
- [14] *Apache Cassandra Open Source NoSQL Database*. 2024. https://cassandra.apache.org/_/index.html .
- [15] *Benchmarking cloud serving systems with YCSB*. 2010. 143--154. Proceedings of the 1st ACM symposium on Cloud computing. 1. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears.
- [16] *XQBench-A XQuery Benchmarking Service*. Peter M. Fischer. 341--355. XML Prague - Conference Proceedings. 2010.
- [17] *Benchmarking XSLT performance*. 2014. XML LONDON 2014. <https://www.saxonica.com/papers/xml london-2014mhk.pdf>. Michael Kay and Debbie Lockett.
- [18] *Projection and Streaming: Compared, Contrasted, and Synthesized*. 73--99. XML Prague - Conference Proceedings. Michael Kay. 2017.
- [19] *TPC-H Version 2 and 3*. <https://www.tpc.org/tpch/>.
- [20] *FUSIONDB Multi-model database*. <https://fusiondb.com/>.
- [21] *Elemental*. <https://elemental.xyz/>.
- [22] *eXist-db*. <http://exist-db.org/exist/apps/homepage/index.html>.
- [23] *Graal VM*. <https://www.graalvm.org/>.
- [24] *XML:DB Initiative for XML Databases*. <https://github.com/xml db-org/xml db-api>.

Simple Semantic Data Modeling in XML (SeMoX)

Renzo Kottmann

Coordination Office for IT-Standards

`<renzo.kottmann@finanzen.bremen.de>`

Cedric Pauken

Research Group E-Government, University Koblenz

`<cpauken@uni-koblenz.de>`

Andreas Schmitz

Research Group E-Government, University Koblenz

`<andreasschmitz@uni-koblenz.de>`

Abstract

The aim of Simple Semantic Data Modeling in XML (SeMoX) is to provide non-technical domain experts a simple model and additional tooling for capturing semantics of data with a technology-neutral approach. It is foremost designed for modeling data exchange standards between heterogeneous systems. SeMoX is simple because all it needs are five basic concepts: Terms, Semantic Datatypes, Rules, Structures and Syntax Bindings. The core artifact of SeMoX is `semo.xsd`¹. This XML Schema defines a concise and linear unfolding XML structure for users to create own SeMoX based semantic data modeling projects. In contrast to many UML based model-driven approaches in standardization, SeMoX is able to leverage the entirety of the fully interoperable XML technology stack. SeMoX is set as the modeling approach of the whole German XEinkauf and already proved to be valuable in production for the development and maintenance of procurement standards such as eForms-DE and XRechnung. SeMoX is open source under a permissive MIT Licence and invites usage and participation.

For further details see homepage <https://semo-xml.org>² and project repository <https://projekte.kosit.org/semo/semo-model>.

Keywords: XML, schema, semantic, modeling, standardization

¹ <https://projekte.kosit.org/semo/semo-model/-/blob/master/src/main/xsd/semo.xsd>

² <https://semo-xml.org>

1. Introduction

SeMoX provides domain experts a simple model and additional tooling for capturing the semantics of data with a technology-neutral approach. It is foremost specific to data standardization within certain knowledge domains and mainly focuses on the interoperable aspects of data exchange between heterogeneous systems. Hence, SeMoX defines a simple Domain-Specific Language in concise XML to structure semantics of data from and for domain experts. It is designed to separate domain knowledge and intent from technical implementations while keeping clear and close connections.

Therefore, SeMoX is not an ontology that models concepts of knowledge, but rather a semantic framework that supports the capturing and structuring of data semantics using a simple domain-specific language defined in an XML-Schema.

SeMoX can be seen as an amalgamation of years of experience in data standardization. The initial idea behind SeMoX came up based on the necessity to integrate existing international standards for public procurement such as the European Committee for Standardization (CEN) [11] or Peppol [9] into the German XSE-Standards (XStandards Einkauf). The declared goal of XSE is to summarize, harmonize and make interoperable all IT-Standards in German public procurement including the domains of ordering, delivery and payment [8]. As such, XSE needs to adopt and maintain several international standards from different standardization bodies in its own format with certain specific national features. SeMoX was designed to support this adoption and maintenance. It is very much informed by the approach of CEN EN16931 norm [5] while avoiding common challenges present in standardization approaches.

It aims to fulfill the following general goals:

1. Create a shareable and validatable model for the definition of specification and standards
2. Facilitate accessibility for domain experts without XML knowledge
3. Bridge the gap between individuals with differing business, technical and legal expertises
4. Promote and facilitate collaboration and consistency between different standards and standardization organizations

Furthermore, SeMoX must accommodate various simple and complex standardization usage scenarios:

- Writing simple specifications (see SeMoX itself model)
- Composing top-down standards (see XRechnung)
- Composing downstream standards as differential expression to upstream standards (see eForms-DE)

2. Basic concepts of SeMoX

SeMoX leverages several basic concepts for creating simple semantic data models in XML. In order to describe these basic concepts, the following continuous example is the XML representation of SeMoX concepts by SeMoX-itself. Utilizing the dogfooding principle allows fully focusing on SeMoX and avoids discussing SeMoX concepts in parallel to other concepts of another data model.

For clarity, we denote a concept by its name written in capitalization, e.g., Rule, XML representation by its tag names, e.g., rule and other names or values are in quotes if it is necessary to differentiate them from concept and tag names, e.g., an id with the value "rule".

2.1. Term

At the heart of SeMoX is the Term concept (a.k.a. Business Term (BT)). A Term is a word or a phrase that describes a concept that is used in a particular knowledge or business domain. Whatever concept a Term expresses, domain experts have to name and describe it. Hence, the basic XML representation of a Term is the following:

```
<term id="term">
  <name>Term</name>
  <description>A Term to be defined by domain experts</description>
</term>
```

This XML instance of a Term is used to define the semantics of a Term.

An excerpt from the SeMoX-itself model illustrates the definition of a list of terms:

```
<definitions>
  <term id="model" datatype="group">
    <name>Model</name>
    <description>The model describing and capturing the semantics
of data using Semantic Datatypes, Terms, Codelists etc..
    </description>
  </term>
  <term id="term" datatype="string">
    <name>Term</name>
    <description>A Term to be defined by
      domain experts</description>
  </term>
  <term id="identifier" datatype="internal-id">
    <name>Identifier</name>
    <description>An identifier</description>
  </term>
  <term id="name" datatype="string">
```

```
<name>A Name of something</name>
<description>Term for giving something a name
              (sometimes in the sense of a label)</description>
</term>
<term id="description" datatype="string">
  <name>Description</name>
  <description>Human readable explanation</description>
</term>
<term id="definitions" datatype="group">
  <name>Definitions</name>
  <description>A list of <term>term</term>s.
  </description>
</term>
</definitions>
```

This list of terms is flat and each term has to have a unique identifier with the XML representation of an `id` attribute.

A SeMoX model with only a list of Terms can be regarded as a glossary.

2.2. Semantic Datatype

Because datatypes are fundamental to software and programming languages in general, SeMoX establishes the Semantic Datatype concept.

Whatever the semantic or intent of a datatype is, domain experts have to name and describe it.

Hence, the basic XML representation of a Semantic Datatype is as follows:

```
<datatype id="string">
  <name>String</name>
  <description>Sequence of characters</description>
</datatype>
```

This XML instance of a Semantic Datatype is used to define the semantics of the String concept.

Overall, Semantic Datatypes are nothing more than high-level declarations of computer representations of data. But already, as such, they give valuable information for technical implementations.

Therefore, each definition of a term includes a declaration of a datatype.

```
<term id="description" datatype="string">
  <name>Description</name>
  <description>Human readable explanation.</description>
</term>
```

In this example, the term with `id` "description" expresses the Description concept and is declared to have the Semantic Datatype with name "String" herewith referred to by `id` of value "string".

Some more Semantic Datatypes are defined in the SeMoX-itself model:

```
<semantic-datatypes>
  <datatype id="any" minOccurs="1" maxOccurs="1">
    <name>Any</name>
    <description>The ur-type, basically no restriction
                  on data values.</description>
  </datatype>
  <datatype id="group" minOccurs="1" maxOccurs="1">
    <name>Group</name>
    <description>Group of <term>term</term>s.
    </description>
  </datatype>
  <datatype id="string" minOccurs="1" maxOccurs="1">
    <name>String</name>
    <description>Sequence of characters</description>
  </datatype>
  <datatype id="internal-id" minOccurs="1" maxOccurs="1"
            restricts="string">
    <name>Internal Identifier</name>
    <description>An id which is unique within a
                  single structure</description>
  </datatype>
</semantic-datatypes>
```

Which and how many semantic datatypes are to be defined in a SeMoX model is a design choice.

The SeMoX-itself model defines the semantic datatype "Any" which can be used as a placeholder in term definitions where name and definition are already defined, but a decision on the datatype is missing.

The sole purpose of the Semantic Datatype group is to indicate that a term of this datatype is a composition of other terms.

For example, the term "model" includes the terms term, semantic datatype and several other terms.

```
<term id="model" datatype="group">
  <name>Model</name>
  <description>The model describing and capturing the semantics
                of data using a composition of Semantic Datatypes,
                Terms, Codelists etc..</description>
</term>
```

2.3. Structure

Terms can be aggregated to form a Semantic Structure in order to express which Term belongs to which group of Terms. Such Structures can be defined according to the intents and needs of the domain as defined by experts. This is independent

of concrete implementations. A Structure also defines the cardinalities for all Terms and groups of Terms. The default cardinality is being optional, the two other cardinalities are mandatory or forbidden. Additionally, a Term or group of Terms can occur repeatably. While one Semantic Structure may specify the cardinality of a certain Term as mandatory, another structure may choose to define it as optional or forbidden - depending on the business context.

The following XML representation shows a partial semantic Structure of the SeMoX-itself model.

```
<structures>
  <structure id="semox">
    <name>Logic Structure</name>
    <type/>
    <description/>
    <message>
      <group ref="model">
        <group ref="metadata">
          <term ref="shortname"/>
        </group>
        <group ref="definitions">
          <group ref="term">
            <term ref="identifier"/>
            <term ref="name"/>
            <term ref="description"/>
            <term ref="name"/>
          </group>
        </group>
        <group ref="rules">
          <group ref="rule" maxOccurs="unbounded">
            <term ref="identifier"/>
            <term ref="name" minOccurs="0" maxOccurs="unbounded"/>
            <term ref="description" maxOccurs="unbounded"/>
            <group ref="rule-implementations" minOccurs="0">
              <term ref="schematron" minOccurs="0"/>
            </group>
          </group>
        </group>
      </group>
    </message>
  </structure>
</structures>
```

2.4. Syntax Binding

Parsing of data is a necessary operation for data exchange. Therefore, SeMoX has the concept of Syntax Binding. The explanation follows the example:

```
<syntax-bindings>
  <binding>
    <structure>semox</structure>
    <syntax id="model-syntax">
      <name>SeMoX Model</name>
      <type>xml</type>
      <query-language>xpath</query-language>
    </syntax>
    <term ref="model" path="/model"/>
  </binding>
</syntax-bindings>
```

This binds the "semox" Structure as previously defined (see Section Structure) to XML instances named "SeMoX Model". Therefore, the Term "model" can be found i.e. queried with the XPath expression `/model` in all such instances. In other words, the actual data with the semantics as defined by the `term` with the referenced `id` "model" are to be found in XML instances with the root element also named `model`.

2.5. Rule

Data Rules are means of further imposing restrictions, assertions and constraints on the use of terms and structures. On principle, these can be of any kind. Although Rules can be defined in a variety of ways, they are most useful when they can be tested via a simple yes or no answer. In SeMoX, rules should therefore be developed from a business perspective and be validatable, as business rules set the expectation levels against which the technical implementation must be measured against.

Additionally, Rule-Groups can be defined. A set of Rules may belong to several Rule-Groups. Each Rule-Group needs an `id`, a `name` and a `description`, whereas each rule only needs an `id` and a `description`. Ideally, Rules should specify a list of terms on which they apply to and provide references regarding implementations. The implementation and validation of rules works via `schema-tron`.

Provided below is another simple XML example from the SeMoX itself model. As discussed earlier, each structure contains terms and groups. A Rule may now enforce that, e.g., each structure must begin with a root group. To do this, a `rule-group` that contains rules about the use of datatypes must be defined first. Subsequently, a rule that mandates structures to begin with a root-group can now be specified. In practice, when building the model, this rule can easily be tested via a simple yes or no answer.

```
<rules>
  <rule-group id="datatype-rules">
    <name>Datatype Rules</name>
```

```
<description>Rules about datatype use</description>
</rule-group>
<rule id="root-group" groups="datatype-rules">
  <description>Any message in a structure must begin with a
    term of the datatype "group"</description>
</rule>
</rules>
```

3. Schema design of SeMoX

The previous section followed the dogfooding principle and explained the basic concepts for semantic modeling of data in XML by using XML for modeling the data and capturing the semantics of SeMoX-itself. Accordingly, an XML Schema (XSD) [6] was developed in order to describe and validate all kinds of SeMoX based data models (`semo.xsd`).

This XSD makes some more notable features of SeMoX models possible:

1. *Model Metadata*: Each model can have metadata for author attribution, lifecycle management (version, status) etc.

Example 1. Metadata section of the current eForms-DE model.

```
<m:meta>
  <m:shortName>eForms-DE</m:shortName>
  <m:name>eForms-DE Standard</m:name>
  <m:id>eforms-de</m:id>
  <m:extends version="1.10.1" id="eforms-eu">
    <m:name>eforms-eu</m:name>
  </m:extends>
  <m:version>1.2.0</m:version>
  <m:status>active</m:status>
  <m:date>2024-02-02</m:date>
  <m:abstract>Deutsche Anpassung und Erweiterung der
    "DURCHFÜHRUNGSVERORDNUNG (EU) 2022/2303 DER KOMMISSION
    vom 24. November 2022 [...].
  </m:abstract>
</m:meta>
```

2. *Multilingual support*: Every basic concept has a name and a description, each of which can be repeated in different languages denoted by the standard `xml:lang` attribute

```
<m:term id="t1" datatype="sdt-1">
  <m:name>Nonsense term 1</m:name>
  <m:name xml:lang="de">Nonsense in Deutsch</m:name>
  <m:description>A term using a semantic datatype.</m:description>
  <m:description xml:lang="de">Eine deutsche Beschreibung.</
```

```
m:description>  
</m:term>
```

3. *Extensible*: Each element allows attributes from different namespaces. Almost all elements with a complex content model may have additional sub-elements from different namespaces.

4. Usage scenarios in practice

In order to showcase some aspects of SeMoX's versatility and utility, some real world usage scenarios of SeMoX are discussed. Common to all scenarios is the requirement to have a book publication made available in PDF format which serves as the standard publication. Each standard has varying requirements on content, style and layout in terms of cover page design, additional chapters, and other aspects. Therefore, a custom publishing pipeline was developed. It is based on Docbook and the Ant build tool. The primary source of content (in publishing lingo) is the SeMoX model, whereas Docbook fragments and markdown files serve as secondary input.

The results are publicly available open standards like eForms-DE (specification³ and model⁴) and XRechnung (specification⁵).

4.1. SeMoX-itself

The actual design of the SeMoX-itself model instance serves two purposes. First, during development, it illustrates how to best represent the SeMoX concepts in XML. The XSD is manually developed in parallel to the development of the XML representation of SeMoX concepts. This separate design process forces clear separation of what is semantically important and needs description in the model as well as what are technical necessities, trivialities and specifics of XSD. This parallel process as exercised with SeMoX-itself includes iterations and considerations going back and forth between a semantic description and design of actual XML representation. This is and will be common to all data standard developments from scratch. Second, it also demonstrates the two common standard artifacts of SeMoX as a result of writing a specification a validation artifact: Here, having a validatable specification in a SeMoX model instance and an XSD for validation.

³ <https://xeinkauf.de/app/uploads/2024/02/specification-eforms-de-v1.2.0.pdf>

⁴ <https://projekte.kosit.org/eforms/eforms-de-specification/-/blob/master/src/main/model/eforms-de-model.xml>

⁵ <https://xeinkauf.de/app/uploads/2023/09/301-XRechnung-2023-09-22.pdf>

4.2. XRechnung

The key characteristics of the development of the XRechnung standard are that XRechnung is a top-down standard and has one semantic data model which has to be bound to two vastly different XML syntaxes.

Here, top-down standard means practically that there is the European Norm CEN 16931 for B2G (Business-to-Government) electronic invoicing published by CEN which includes several specifications in PDF format of which the CEN 16931-1 document is normative. This norm is on top because all European member states are required to adhere to the norm by law. However, because there are too many nationally specific requirements on electronic invoices, the norm allows "down"stream standards to deviate from the norm by certain normative rules. Hence, XRechnung is the German downstream standard with some additional requirements on electronic invoices in the German market compliant to the CEN Norm. At the heart of the CEN Norm is a single and explicit semantic model of a core invoice. This core model is then bound to the XML invoice representation as defined by the Universal Business Language (UBL) and also Cross Industrie Invoice (CII) by UN/CEFACT. Therefore, the major achievement of CEN EN-16931 is to semantically integrate UBL and CII in a way that both syntaxes can practically be used for invoicing while ensuring semantic interoperability.

In the following the Term designated "BG-30" is used to show how this semantic integration is declared in the XRechnung SeMoX model:

```
<m:group ref="BG-30">
  <m:term ref="BT-151" />
  <m:term ref="BT-152" minOccurs="0" />
</m:group>
```

then the Syntax Binding with UBL for the above Terms is as follows:

```
<m:term ref="BG-30"
  path="/Invoice/cac:InvoiceLine/cac:Item/
cac:ClassifiedTaxCategory"/>
<m:term ref="BT-151"
  path="/Invoice/cac:InvoiceLine/cac:Item/
cac:ClassifiedTaxCategory/cbc:ID"/>
<m:term ref="BT-152"
  path="/Invoice/cac:InvoiceLine/cac:Item/
cac:ClassifiedTaxCategory/cbc:Percent"/>
```

and the Syntax Binding with CII for the above Terms is as follows:

```
<m:term ref="BG-30"
  path="/rsm:CrossIndustryInvoice/rsm:SupplyChainTradeTransaction/
ram:IncludedSupplyChainTradeLineItem/ram:SpecifiedLineTradeSettlement/
ram:ApplicableTradeTax">
```

```
<m:term ref="BT-151"
  path="/rsm:CrossIndustryInvoice/rsm:SupplyChainTradeTransaction/
ram:IncludedSupplyChainTradeLineItem/ram:SpecifiedLineTradeSettlement/
ram:ApplicableTradeTax/ram:CategoryCode"/>

<m:term ref="BT-151"
  path="/rsm:CrossIndustryInvoice/rsm:SupplyChainTradeTransaction/
ram:IncludedSupplyChainTradeLineItem/ram:SpecifiedLineTradeSettlement/
ram:ApplicableTradeTax/ram:TypeCode"/>

<m:term ref="BT-152"
  path="/rsm:CrossIndustryInvoice/rsm:SupplyChainTradeTransaction/
ram:IncludedSupplyChainTradeLineItem/ram:SpecifiedLineTradeSettlement/
ram:ApplicableTradeTax/ram:RateApplicablePercent"/>
```

The Syntax Binding of CII includes to different XPath for "BT-151" which is ambiguous and might be of disadvantage in certain situations. But, here it shows that SeMoX allows to declaratively make explicit the ambiguity of certain bindings. Moreover, it is conceivably simple to use a SeMoX for semantic integration of different data format standards such as UBL and CII.

4.3. eForms-DE

Currently, the Coordination Office for IT-Standards in Germany (CoSIT) develops and maintains the German downstream standard eForms-DE of the upstream EU eForms regulation ("Commission Implementing Regulation (EU) establishing standard forms for the publication of notices in the field of public procurement and repealing Implementing Regulation (EU) 2015/1986 ("eForms")"⁶). Similar to the above XRechnung setting, this regulation also allows national deviations from the EU regulation in order to allow member states to implement national specific requirements. The EU requirements and interoperability with national requirements on notices in the domain of public procurement are rather complex. Therefore, the regulation defines several hundreds of Terms for more than 40 different notices (Structures in SeMoX) and 3 different syntaxes where each Term has different cardinality requirements and sometimes slightly different semantics. Currently, the upstream definition and technical implementations change on a bi-monthly basis.

Hence, the CoSIT implemented a "tailoring process" based on SeMoX. The idea is to upstream create a SeMoX model of each new version of "eForms-EU" and then to downstream always re-create a national SeMoX eForms-DE model. For this, SeMoX was extended to declare only the semantically specified deviations from - or in other words, tailoring of the eForms-EU model. The creation of

⁶ <https://ec.europa.eu/docsroom/documents/43488>

the eForms-EU SeMoX model is a complex integration of several sources of information and hence an own converter was developed and is available at <https://projekte.kosit.org/eforms/eforms-converter>.

In order to explain the actual tailoring step in the process an excerpt of the Model Tailoring is shown here:

```
<m:model-tailoring version="0.4.0" xml:lang="de" xmlns:m="semantic-model"
  xmlns:e="eforms">
  <m:meta>
    <m:shortName>eForms-DE</m:shortName>
    <m:name>eForms-DE Standard</m:name>
    <m:id>eforms-de</m:id>
    <m:extends version="1.10.1" id="eforms-eu">
      <m:name>eforms-eu</m:name>
    </m:extends>
    <m:specification />
    <m:version>1.2.0</m:version>
    <!-- ... for brevity -->
  </m:meta>

  <m:change-datatypes>
    <m:datatype tailor="code">
      <m:name mode="add" xml:lang="de">Code</m:name>
      <m:description mode="add" xml:lang="de"
        >Werte aus einer definierten Codeliste.</m:description>
    </m:datatype>
    <!-- ... for brevity -->
  </m:change-datatypes>
  <!-- ... for brevity -->
  <m:change-terms>
    <m:term tailor="BG-7141">
      <m:description xml:lang="de" e:eg="3"
        >Einzelheiten zu den Fahrzeugen im
        Anwendungsbereich des Saubere-Fahrzeuge-Beschaffungs-
        Gesetzes zur Umsetzung der Richtlinie 2009/33/EG.
      </m:description>
    </m:term>
    <!-- ... for brevity -->
  </m:change-terms>
  <m:rules>
    <m:rule id="BR-DE-20" on-terms="BT-06 BT-775 BT-776" e:eg="prea-6">
      <m:description>Wenn BT-06 (Strategische Beschaffung) einen Wert
        ungleich "entfällt" enthält, ist mindestens eines der Felder BT-774
        (Grüne Beschaffung BT-775 (Soziale Beschaffung) oder BT-776
        (Beschaffung von Innovation) zu füllen.</m:description>
    </m:rule>
```



```
<!-- ... for brevity -->
</m:rules>
<m:model-tailoring>
```

The XML representation of the Model Tailoring follows the structure of a normal model but named `model-tailoring` to distinguish the purpose. The tags `change-datatype` and `change-term` to declare the changes to be made w.r.t. the upstream model. The nested elements are of same structure as in a normal model except that each element has a `tailor` attribute instead of an own `id` attribute which identifies the respective element in the upstream model:

```
<m:datatype tailor="code">
  <m:name mode="add" xml:lang="de">Code</m:name>
  <m:description mode="add" xml:lang="de"
    >Werte aus einer definierten Codeliste.</m:description>
</m:datatype>
```

here `tailor="code"` states that the Semantic Datatype "code" is to be changed with the nested information. The attribute `mode="add"` of the name element tells a processor that first the element `name` is not to be expected to exist in the upstream model and second to add it in the downstream model. Currently, a simple XSLT script exists which has an upstream model and a model-tailoring instance as input and generates the "tailored" model.

Overall this allows maintaining a stable semantic definition of specific German changes to eForms-EU with an own independent life cycle of the upstream changes and rate of changes. This approach to semantic data modeling in a complex and highly changing setting proves to be very useful in many ways. First, by creating an upstream SeMoX model from different sources and then validating this model made possible to find several mistakes and inconsistencies in the upstream standard. Second, being able to define German deviations and re-apply them allowed a German expert group - involving dozens of people from many different institutions and ministries - to work on its own pace with its own quality control, and procedures independent from the rate of changes. Practically, Germany does skip several upstream versions while maintaining own "model-tailoring" versions and only re-apply model-tailoring at own defined time points. Extensive automatic validating and testing throughout this process also ensures full interoperability to upstream eForms-EU.

The current status of eForms-DE is available: eForms Model⁷

5. Conclusion and future work

The way SeMoX is conceptualized and technically implemented proved to be viable in different real world usage scenarios.

⁷ <https://projekte.kosit.org/eforms/eforms-ts/-/blob/master/eforms-model.xml>

The SeMoX-itself usage scenario exemplifies writing specifications in XML for XML instance validation.

The domain knowledge is captured in a model instance separated from the technical implementation. The model instance is shareable (see `semox-itself-model.xml`⁸). The model is validatable using the `semo.xsd` implementation of SeMoX.

The XRechnung usage scenario demonstrates that the separation allows for the binding of common and harmonized semantics to two different XML Syntaxes which already existed and have completely different XML Schemas.

Last, the eForms scenario demonstrates that the extensible design of the SeMoX implementation makes it possible to only capture the semantic difference of a downstream standard and create a full SeMoX instance as a tailored version of the upstream standard. This makes it possible to accommodate any rate of change of an upstream standard.

Overall, this demonstrates the versatility and utility of SeMoX in even rather complex standardization scenarios which are modeling data exchange standards between heterogeneous systems. By no means it attempts to provide a general approach to semantic data modeling. Capturing and making explicit semantics of data is a long-term research topic. A plethora of discussions of what semantics actually is, including capturing and modeling approaches and solutions, have been held, published and deployed. Among which are Resource Description Framework ([15]), Simple Knowledge Organisation System ([16]) and Topic Maps ([17]). Whereas RDF has a very broad scope and a generalized "subject-predicate-object" graph model for describing resources in general, the scope of Topic Maps and SKOS is on modeling and capturing real world knowledge. A full and detailed discussion of commonalities and differences between the aforementioned approaches and SeMoX is beyond the limits of this paper. However, the differentiation is mainly driven by the narrow scope of modeling data for interoperable data exchange scenarios between heterogeneous systems. SeMoX' scope is neither on general descriptions of resources nor on knowledge and concepts and relations between these. Hence, the concept Term only includes a name and description. And consequently, the concept of Structure is not meant to express semantic relations between knowledge concepts, it just only groups Terms in order to describe that a group of data elements together represent a more complex data description such as an address that consists of a street name and a city name. Therefore, SeMoX is dubbed simple.

As demonstrated SeMoX is capable of achieving its outlined goals. It is in productive use and a core component of XEinkauf (XSE). However, the further facilitation of data standardization and interoperability between different standards

⁸ <https://projekte.kosit.org/semox/semox-model/-/blob/master/src/main/model/semox-itself-model.xml>

will be the focus of SeMoX's ongoing development and is discussed in the following.

5.1. Facilitate more effective data standardization

Many XML based data standards already exist today. Most of which have specification documents and validation artifacts of various sorts. The content and comprehensibility vary for non-technical domain experts, whereas SeMoX based standards have a clear focus on only including semantics and related business intent in specifications. Thus, technical implications which can be solved and implemented in validation artifacts and related tools are left out. This makes SeMoX based standard developments more amenable to non-technical domain experts and therefore facilitates more effective data standardization w.r.t. correctness and accuracy of a knowledge domain's semantics. To illustrate this: In a group of experts, it only needs one person who can encode the results of expert discussions in a SeMoX model and generate a specification document. All others only need to know the knowledge domain and can verify the correctness based on the specification. Or, the other way around: SeMoX allows limited options to "dilute" the model with technicalities and gives full freedom to technical implementations.

Moreover, every data standard based on SeMoX can validate its SeMoX model instance with `semo.xsd`. Consequently, many aspects of semantic consistency (e.g. does every term use a defined datatype? Are all terms used in a certain structure?) can be validated. For example, to ensure consistency in eForms-DE, the project includes dozens of additional Schematron rules which are specific to the context of this domain and standard.

Although, SeMoX models are not validation artifacts by themselves, the models can be used as input data for tests which check consistency between semantic model and validation artifacts during standard development. A very simple consistency check can be implemented if a SeMoX rule has the same `id` as a Schematron rule or assertion which implements the corresponding SeMoX rule.

A SeMoX model instance with `syntax-bindings` and hence having information on an `xpath` for any given term can be used to generate simple occurrence/existence constraints. It is also conceivable to generate simple XSDs from models based on the information of semantic datatypes and structures including occurrences.

Taken together, SeMoX already facilitates more quality controlled and effective data standardization. Further design and tooling will seek to increase effectiveness.

5.2. Enhancing interoperability between different standards and standard organisations

Other standardization approaches such as Peppol and CEN encounter common challenges that hinder their effectiveness and interoperability. One such challenge is their tendency to prioritize either technical or business perspectives, creating difficulties for editors from the opposing viewpoint to participate effectively. This imbalance can lead to misunderstandings, inefficiencies, and barriers to collaboration, especially when onboarding new people. Additionally, these approaches necessitate vastly different production environments with external dependencies. For instance, Peppol relies on a rigidly defined project and file structure based on different XML and YAML technologies. While such requirements may ensure consistency within each standard, they pose significant hurdles for individuals with varying backgrounds and skill sets. Furthermore, changes in these dependencies can lead to delays and complications, disrupting implementation efforts and hindering progress. Moreover, the lack of interoperability between standards further amplifies these challenges. Without seamless integration and compatibility between different standardization approaches, organizations face difficulties in mapping between data and standards across diverse standardization environments. This limitation stifles innovation and hampers the potential benefits of standardization efforts. Addressing these challenges requires a concerted effort to bridge the gap between technical and business perspectives, harmonize production environments, and enhance interoperability between standards.

SeMoX is addressing these gaps by providing an interoperable methodology for standard definition, acting as a bridge between various different actors, organizations or processes that need to collaborate. First, SeMoX endeavors to bridge the gap between individuals with technical and business/legal expertise that often have challenges in communicating on the same level. By accommodating both perspectives, it promotes understanding and cooperation among diverse stakeholder groups. Second, SeMoX seeks to bridge the gap between maintenance systems associated with different standards. By adopting a technology-neutral approach, it enables integration and collaboration across various systems, ensuring adaptability and compatibility with evolving external technologies. Finally, SeMoX aims to bridge the gap between the technical formats of different standards, enhancing interoperability between different standardization approaches. By offering a common foundation for standard definition, it promotes consistency and facilitates collaboration between different standardization organizations.

One way to further foster collaboration would be to integrate the concept of transactions into SeMoX. European Peppol [10] and CEN [12] standards commonly use transactions to standardize the XML-message-flow between a contracting body and an economic operator in different phases of the procurement process. Most notably, transactions are used in the pre-award phase of the pro-

curement process to express the concept of a request and a response between a contracting body and an economic operator. Within a pre-award process, one actor might need to request information from another actor via a standardized request-transaction. The other actor can then fill out the required information from the request-transaction and transmit a standardized response-transaction back to the requester. Integrating the concept of transactions into SeMoX would lead to full interoperability between SeMoX-standards and European Peppol/CEN standards in the domain of public procurement. Therefore, SeMoX aims to develop fully EU-interoperable procurement standards in the future by adjusting the current framework to include transactions. In practice, this may work by allowing for the definition of several transactions within a single SeMoX-instance (similar to structures).

Additionally, more practical research focused on the interoperability between SeMoX and other standardization methodologies is planned.

First, conceptional mappings between SeMoX and Peppol / CEN should be created. Based on these mappings, transformations from Peppol / CEN to SeMoX can be created and tested. A first step may be an XSL transformation from Peppol to SeMoX. More such technical studies can provide further insights into how SeMoX can act as a bridge between standardization methodologies.

5.3. Further research

Overall, this paper demonstrates how SeMoX can be utilized to provide a semantic standardization model. Many current issues in the domain of standardization can be specifically addressed by SeMoX. This research provides an introduction to the concepts and benefits of SeMoX and is supposed to be the foundation for further research. Literature shows a clear need for methodologies and frameworks in the domains of standardization and public procurement in particular, that SeMoX can address [14]. Further SeMoX should also be considered as a part for more holistic frameworks regarding interoperability, for example as an extension of the Framework for Interoperable Service Architecture Development [13]. Besides the technical additions to SeMoX discussed above in detail, further generic research needs on SeMoX include:

1. Comprehensive classification of SeMoX in the status quo of scientific literature to ensure rigor
2. Empirical verification of SeMoX's applicability in practice by conducting a structured evaluation of its effectiveness across a wide range of different use-cases
3. Development of methodological frameworks for best-practice applications of SeMoX

Bibliography

- [1] Kay, Michael: *XSLT 2.0 and XPath 2.0*. Wiley Publishing, 2008.
- [2] Gao, Shudi– Sperberg-McQueen, C.M. – Thompson, Henry S.: *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C Recommendation, 5 April 2012. <https://www.w3.org/TR/xmlschema11-1/>
- [3] Jelliffe, Rick. *Schematron*, 1999. Retrieved from <http://xml.ascc.net/schematron>
- [4] *XRechnung*. Retrieved from <https://www.xoev.de/de/xrechnung>
- [5] *Electronic invoicing - Part 1: Semantic data model of the core elements of an electronic invoice; German version EN 16931-1:2017*. Retrieved from <https://www.din.de/de/mitwirken/normenausschuesse/nia/normen/wdc-beuth:din21:274990963>
- [6] *Validation artefacts for the European eInvoicing standard EN 16931*. Retrieved from <https://github.com/ConnectingEurope/eInvoicing-EN16931>
- [7] *Universal Business Language Version 2.1.. 04 November 2013*. OASIS Standard. <http://docs.oasis-open.org/ubl/os-UBL-2.1/UBL-2.1.html>
- [8] *XStandards Einkauf - Die Standardfamilie des öffentlichen Einkaufs*. <https://xeinkauf.de/>
- [9] *Peppol (Pan-European Public Procurement OnLine)*. <https://peppol.org/>
- [10] *Peppol Pre-Award Profiles Overview*. Retrieved from <https://docs.peppol.eu/pracc/>
- [11] *About the European Committee for Standardization (CEN)*. <https://www.cencenelec.eu/about-cen/>
- [12] *CEN/TC 440 'Electronic Public Procurement'*. <https://www.cencenelec.eu/areas-of-work/cen-cenelec-topics/public-procurement/cen-tc-440-electronic-public-procurement/>
- [13] *Framework for interoperable service architecture development*. <https://www.sciencedirect.com/science/article/abs/pii/S0740624X23000692>
- [14] *Closing the gap: Leveraging data for seamless integration between pre-award and post-award in public procurement*. <https://scholarspace.manoa.hawaii.edu/items/be322fe7-6932-4132-baea-af702e6081a5>
- [15] *RDF 1.1 Concepts and Abstract Syntax*. <https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/>
- [16] *SKOS Simple Knowledge Organization System Reference*. <https://www.w3.org/TR/skos-reference/>

- [17] *The TAO of Topic Maps*. <https://www.ontopia.net/topicmaps/materials/tao.html>

GEDCOM to RDF

Transforming Genealogical Data for use in a Personal Knowledge Graph

Robert Walpole
Devexe Limited

<rob.walpole@devexe.co.uk>

Abstract

This paper describes a process for converting genealogical data in GEDCOM format to RDF suitable for loading into a Personal Knowledge Graph (PKG)[1]. As part of this process, we aim to retain as much metadata about individuals and their relationships as possible. Once loaded into the PKG, this dataset will provide an excellent base for further enrichment of the PKG following the principle of the Open World Assumption. New individuals and additional information about existing individuals can be added to the PKG over time.

It should be noted that the PKG being created is intended purely for personal or household activities under the terms of GDPR and is not intended for use in a public space.

Keywords: RDF, XML, GDECOM, RML, YARRRML, Linked Data

1. Background

The background to this piece of work is an ongoing project to build a secure, linked data personal diary system, backed by a Personal Knowledge Graph (PKG)[1] which permits the linking of words and phrases in the text to resources within the PKG, as well as to public resources, such as those in DBpedia or Wikidata. The purpose of this linking is to make clear what is meant by the words and phrases within the text of the diary. For example, if someone writes “my grandfather” which grandfather do they mean?

Why would we want to do this? After all, diaries are normally private and the writer will normally know exactly what they mean. That may be so, but most people keep a diary as an aid to memory and if someone keeps a diary for many years it can be incredibly difficult to find specific pieces of information about an event or person, unless you know exactly when something happened. If the diary is kept in a digital format, it should be possible to search the text for key phrases, but this will likely produce a lot of noise. By linking the text to a PKG it would be possible to find every mention of a particular individual in the text and also access additional information about the person that the writer is referring to. In

our grandfather example, this could be easily achieved by linking the text at the time of writing to the grandfather resource stored in a PKG.

Furthermore, the act of linking text in a diary to representative resources in a knowledge graph is one way to learn more about yourself and the people around you. For example, when talking about a cousin it may be difficult to recall exactly how this cousin is related to you, whether they are 1st or 2nd cousins and how many times they are removed. Linking the mention of a cousin to a PKG would allow you to quickly see the relationship at any time.

There are other benefits to having a PKG-backed diary as well. Future generations coming across your diary - or perhaps your biographers, should you become famous (or infamous) - will be able to understand exactly who or what you were talking about straight away.

Finally, creating and maintaining a PKG has benefits beyond diarying. In future, it will be possible to have an AI-based personal assistant, which can be given access to your PKG and thereby provide you with personal insights and information which may otherwise not be immediately apparent to you. For example, let's imagine that the aforementioned grandfather once lived in Istanbul and this was recorded in your PKG. You might make a visit to Istanbul and your PKG-backed personal assistant could remind you of this fact about your grandfather, and perhaps even guide you to the house he used to live in.

2. Creating a Personal Knowledge Graph

Creating a Personal Knowledge Graph from scratch is potentially a long and tedious process. If you come from a large family then adding information about everyone in it could be particularly tiresome. One solution is to look around at existing data sources to see if they could be used to populate the PKG.

One potential source of information is genealogical data. Many people are interested in family history these days and there are some excellent tools out there for creating family trees in an easy and enjoyable way. Ancestry.com is one example of a commercial website which allows you to build family trees using a friendly GUI. Furthermore, once you have created a family tree you can download it in GEDCOM[1] file format. This GEDCOM file can contain not only names and relationships, but also a lot of other metadata that would be useful in a PKG, such as dates of birth, marriage dates, children's names, and relationships between otherwise slightly obscure cousins. It seems like a good place for us to start to see if we can convert this genealogical data into an RDF format, which could then form the start of a PKG.

3. GEDCOM

GEDCOM or FamilySearch GEDCOM, to give it its full name, is a proprietary and de facto open plain text file format specification used for the exchange of

genealogical data. It was developed by the Church of Jesus Christ of Latter-Day Saints as an aid to genealogical research and is maintained today by the non-profit FamilySearch organisation. As well as Ancestry.com, it is supported by other online genealogical services such as FindMyPast.co.uk and MyHeritage.com. There are a number of different versions of GEDCOM in circulation, however GEDCOM 5.5 which was first published in 1996 is widely supported by genealogical software suppliers. The latest version is GEDCOM 7.0.

There are a number of existing open source tools which offer to convert GEDCOM files to RDF, including a Java-based tool called `gedcom2sem`[2] and a PHP-based tool called `GedcomRDF`[3]. However, both of these projects have been neglected for a number of years and a fresh approach to the challenge now seems appropriate. There may be other chargeable tools available but this was not investigated.

For demonstration purposes we have created a family tree in Ancestry for the fictional Walton family, as portrayed in the 1970's America TV show "The Waltons". Character information for this tree was obtained from The Waltons Wiki[4]. This tree will be downloaded in GEDCOM 5.5 format.

4. GEDCOM X

In 2012, FamilySearch launched a new genealogical standards project called GEDCOM X[5]. GEDCOM X is not a newer version of GEDCOM but rather an alternative representation which includes specifications for an open data model and open serialisation format. It has a key advantage over standard GEDCOM in that it can be serialised to a JSON or XML format. FamilySearch has also made available libraries for converting GEDCOM 5.5 to GEDCOM X. Converting our GEDCOM file to an XML format would be a key first step in converting it to RDF.

5. GEDCOM to GEDCOM X conversion

The `gedcom5-conversion`[6] command line tool can be used to convert a GEDCOM 5.5 file to GEDCOM X. Following the instructions provided in the tool README we can convert our Waltons GEDCOM file to GEDCOM X using a command similar to the following:

```
java -jar /tmp/gedcom5-conversion-full.jar -i /tmp/WaltonFamilyTree.ged -o /tmp/WaltonFamilyTree.gedx
```

The output file `/tmp/WaltonFamilyTree.gedx` is based on the ZIP format so it can be unpacked as follows:

```
unzip -d /tmp/Waltons /tmp/WaltonsFamilyTree.gedx
```

Examining the `/tmp/Waltons` directory we will now find a `tree.json` file containing our family tree for the Waltons in GEDCOM X JSON format. Unfortu-

nately the conversion utility does not currently allow us to choose whether to output JSON or XML format so a further step is required to convert the file to XML.

6. GEDCOM X JSON to XML conversion

FamilySearch has created a number of open source libraries which are suitable for reading and writing GEDCOM X, including `gedcomx-java`[7]. As part of this project, a command line wrapper has been created around the Java API library which is available from `gedcom-transformer`[8]. Following the instructions provided in the README of this project we can convert the original `WaltonsFamilyTree.gedx` file to XML using a command similar to the following:

```
java -jar /tmp/gedcom-transformer-full.jar -i /tmp/WaltonFamilyTree.gedx
-o /tmp/WaltonXmlTree.gedx
```

As before, the output file `/tmp/WaltonFamilyTreeXml.gedx` can be unpacked using the `unzip` command:

```
unzip -d /tmp/WaltonsXml /tmp/WaltonFamilyTreeXml.gedx
```

Now, within the `/tmp/WaltonsXml` directory specified in the command above, you will find our GEDCOM X file in XML format named as `tree.xml`. You can open this file in your favourite XML editor to examine it in more detail.

The first thing you will notice is that the XML is in the `http://gedcomx.org/v1/` namespace and the root element is `gedcomx`. Within the root there are multiple `person` elements, which contain the details we have for each individual within the tree. For example John (John-Boy) Walton would appear as follows:

```
<person id="I112573407540">
  <gender type="http://gedcomx.org/Male"/>
  <name>
    <preferred>true</preferred>
    <nameForm>
      <fullText>John Walton</fullText>
      <part type="http://gedcomx.org/Given" value="John"/>
      <part type="http://gedcomx.org/Surname" value="Walton"/>
    </nameForm>
  </name>
  <fact type="http://gedcomx.org/Birth">
    <date>
      <original>1916</original>
    </date>
    <place>
      <original>Virginia, USA</original>
    </place>
  </fact>
</person>
```

```
</fact>
</person>
```

There are also a number of relationship elements which are either type `http://gedcomx.org/ParentChild` or type `http://gedcomx.org/Couple`. Within each of these elements there is a `person1` and `person2`. Each person element has a `resource` attribute which identifies the person concerned by the identifier. For example, the following snippet links John-Boy Walton's parents, his father John Walton Senior and his mother Olivia:

```
<relationship type="http://gedcomx.org/Couple" id="F7-I112573407541-I112573407611">
  <person1 resource="#I112573407541"/>
  <person2 resource="#I112573407611"/>
</relationship>
```

In the case of couples, the relationships are symmetrical whereas in the case of parent-child relationships the parent appears as `person1` and the child as `person2`. In the following snippet John-Boy Walton is identified as `person2` and his mother as `person1`:

```
<relationship type="http://gedcomx.org/ParentChild" id="F7-I112573407611-I112573407540">
  <person1 resource="#I112573407611"/>
  <person2 resource="#I112573407540"/>
</relationship>
```

This will be important later when we come to convert the XML to RDF.

7. RML

The RDF Mapping Language (RML)[9] is currently a draft specification for a mapping language based on and extending the RDB to RDF Mapping Language (R2RML)[10] which is an existing W3C recommendation. While R2RML is targeted at mapping relational databases to RDF, RML is intended to be more generic and allows conversion of other formats to RDF, including XML.

8. YARRRML

YARRRML[11] is another draft specification for a more human-readable text-based representation of RML rules. It is a subset of YAML and in our case can be used to express the mapping from XML elements to the RDF subjects, predicates and objects that we want to apply when converting our GEDCOM X XML file to RDF.

For the Walton's example, we will name our YARRRML file `gedcomx-rules-waltons.yml`, and the first part of the file which includes the prefixes and mappings declaration will also specify the mappings we have chosen for people:

```
prefixes:
  dbo: http://dbpedia.org/ontology/
  wexp: http://waltons.example.org/person/
  grel: http://users.ugent.be/~bjdmeest/function/grel.ttl#
mappings:
  people:
    sources:
      - ['WaltonsXml/tree.xml~xpath','/gedcomx/person']
    subjects: wexp:$(@id)
    predicateobjects:
      - [a, dbo:Person]
      - [a, foaf:Person]
      - [foaf:givenName, "$(name/nameForm/part[@type='http://gedcomx.org/
Given']/@value)"]
      - [foaf:surname, "$(name/nameForm/part[@type='http://gedcomx.org/
Surname']/@value)"]
      - [foaf:gender, "$(gender/@type)"]
      - [dbo:birthDate, "$(fact[@type='http://gedcomx.org/Birth']/date/
original)"]
      - [dbo:birthPlace, "$(fact[@type='http://gedcomx.org/Birth']/place/
original)"]
      - [dbo:deathDate, "$(fact[@type='http://gedcomx.org/Death']/date/
original)"]
      - [dbo:deathPlace, "$(fact[@type='http://gedcomx.org/Death']/place/
original)"]
```

The set of prefixes we wish to use in our RDF output are defined at the start of the file. In line with the Linked Data Best Practice principle using standardised vocabularies as far as possible, we have chosen to map our GEDCOM X XML element and attribute values to existing vocabularies whenever possible. The `grel` prefix is used to attach the GREL functions[16] which we will make use of shortly.

The prefixes are followed by the mappings. The first mapping we see in the file is for people. The sources of people is given as follows:

```
- ['WaltonsXml/tree.xml~xpath','/gedcomx/person']
```

This statement includes our GEDCOM X XML file path suffixed with `~xpath`. This means that we are going to use XPath[12] to process the file. The next statement provides the XPath statement to locate people within the XML file. The `subjects` key defines the subjects IRI in the RDF triple we are going to generate. In this case we use the `wexp` prefix and append the value of the person id attribute. This means that the IRI for our John-Boy resource would be as follows:

```
http://waltons.example.org/person/I112573407540
```

Next we see a `predicateobjects` section. This defines the predicates and objects for each subject resource defined previously. For example, we see given above:

```
- [foaf:givenName, "$(name/nameForm/part[@type='http://gedcomx.org/
Given']/@value)"]
```

This means that the predicate will be `foaf:givenName` and the object will be derived from the relative XPath statement:

```
name/nameForm/part[@type='http://gedcomx.org/Given']/@value]
```

In the case of our Walton's example we would expect this to result in a triple as follows for John-Boy:

```
<http://waltons.example.org/person/I112573407540> foaf:givenName "John" .
```

The next part of the YARRRML file deals with the mappings for relationships. Firstly couples where `person1` is connected to `person2` via the `dbo:spouse` predicate:

```
couples1to2:
  sources:
    - ['WaltonsTreeXml/tree.xml~xpath', "/gedcomx/
relationship[@type='http://gedcomx.org/Couple']"]
  subjects:
    - function: grel:string_replace
      parameters:
        - [grel:valueParameter, wexp:$(person1/@resource)]
        - [grel:p_string_find, '#']
        - [grel:p_string_replace, '']
  predicateobjects:
    - predicates: dbo:spouse
      objects:
        - function: grel:string_replace
          parameters:
            - [grel:valueParameter, wexp:$(person2/@resource)]
            - [grel:p_string_find, '#']
            - [grel:p_string_replace, '']
          type: iri
```

And then the inverse where `person2` is connected to `person1` in the same way:

```
couples2to1:
  sources:
    - ['WaltonsXml/tree.xml~xpath', "/gedcomx/
relationship[@type='http://gedcomx.org/Couple']"]
  subjects:
    - function: grel:string_replace
      parameters:
        - [grel:valueParameter, wexp:$(person2/@resource)]
        - [grel:p_string_find, '#']
        - [grel:p_string_replace, '']
  predicateobjects:
```

```
- predicates: dbo:spouse
  objects:
    - function: grel:string_replace
      parameters:
        - [grel:valueParameter, wexp:$(person1/@resource)]
        - [grel:p_string_find, '#']
        - [grel:p_string_replace, '']
      type: iri
```

In this case both the subjects have a function applied. This is a GREL string replacement function to remove a hash which is added to the person identifier value of the resources attribute of the relationship element in the GEDCOM XML.

Next we look for parent-child relationships from the point of view of the parent. In other words to find an individual's children. In this case `dbo:child` is used as the predicate:

```
children:
  sources:
    - ['WaltonsXml/tree.xml~xpath', "/gedcomx/
relationship[@type='http://gedcomx.org/ParentChild']"]
  subjects:
    - function: grel:string_replace
      parameters:
        - [grel:valueParameter, wexp:$(person1/@resource)]
        - [grel:p_string_find, '#']
        - [grel:p_string_replace, '']
  predicateobjects:
    - predicates: dbo:child
      objects:
        - function: grel:string_replace
          parameters:
            - [grel:valueParameter, wexp:$(person2/@resource)]
            - [grel:p_string_find, '#']
            - [grel:p_string_replace, '']
          type: iri
```

Next we look for parent-child relationships from the point of view of the child. In other words, to find an individual's parents. In this case `dbo:parent` is used for the predicate:

```
parents:
  sources:
    - ['WaltonsXml/tree.xml~xpath', "/gedcomx/
relationship[@type='http://gedcomx.org/ParentChild']"]
  subjects:
    - function: grel:string_replace
```



```
parameters:
- [grel:valueParameter, wexp:${person2/@resource}]
- [grel:p_string_find, '#']
- [grel:p_string_replace, '']
predicateobjects:
- predicates: dbo:parent
  objects:
  - function: grel:string_replace
    parameters:
    - [grel:valueParameter, wexp:${person1/@resource}]
    - [grel:p_string_find, '#']
    - [grel:p_string_replace, '']
  type: iri
```

9. YARRRML Parser

The `yarrml-parser`[13] is a parser library and command line interface written in JavaScript which is used to generate an RML rules file in Turtle[14] format from our more human-readable YARRRML file. After following the installation instructions in the README, we can run the parser, giving the YARRRML file described above as the input parameter and an output filename for the result:

```
yarrml-parser -i /tmp/gedcomx-rules-waltons.yml -o /tmp/gedcomx-rules-
waltons.rml.ttl
```

10. RML Mapper

The `rmlmapper-java`[15] is a Java command line tool for generating Linked Data from RML rules. By downloading the latest release version from <https://github.com/RMLio/rmlmapper-java/releases> and following the usage instructions shown in the project README, we can use the tool to generate our knowledge graph of the Walton family:

```
java -jar rmlmapper-all.jar -m /tmp/gedcomx-rules-waltons.rml.ttl -
o /tmp/Waltons.ttl
```

The resulting `Waltons.ttl` file contains all of the members of the Walton's family, their relationships and other available metadata in RDF Turtle format. The following is a snippet of the file, showing the relationship between John-Boy Walton, his father John and mother Olivia. Other relationships have been trimmed from the example for the sake of brevity:

```
@prefix wexp: <http://waltons.example.org/person/> .
@prefix dbo: <http://dbpedia.org/ontology/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
```

```
wexp:I112573407540 rdf:type dbo:Person , foaf:Person ;
                    dbo:birthDate "1916" ;
                    dbo:birthPlace "Virginia, USA" ;
                    dbo:parent wexp:I112573407541 ;
                    dbo:parent wexp:I112573407611 ;
                    foaf:gender "http://gedcomx.org/Male" ;
                    foaf:givenName "John" ;
                    foaf:surname "Walton" .

wexp:I112573407541 rdf:type dbo:Person , foaf:Person ;
                    dbo:birthDate "1896" ;
                    dbo:child wexp:I112573407540 ;
                    dbo:spouse wexp:I112573407611 ;
                    foaf:gender "http://gedcomx.org/Male" ;
                    foaf:givenName "John" ;
                    foaf:surname "Walton" .

wexp:I112573407611 dbo:birthDate "1899" ;
                    dbo:child wexp:I112573407540 ;
                    dbo:spouse wexp:I112573407541 ;
                    rdf:type dbo:Person ;
                    rdf:type foaf:Person ;
                    foaf:gender "http://gedcomx.org/Female" ;
                    foaf:givenName "Olivia" .
```

11. Conclusion and future work

This piece of work has been extremely successful in establishing a toolset for being able to repeatedly generate Linked Data from genealogical records held in GEDCOM format. When run across a more realistic family tree it has generated over 5000 RDF triples, something that would take a great deal of time to do by hand. Furthermore there is opportunity to extract more data from the GEDCOM files such as residence and employment information. It is also foreseen that the steps described in this paper will in future be orchestrated into a workflow which can be run at the click of a button, removing the need for issuing any shell commands. A framework such as Apache Hop would be an ideal candidate for such a platform.

As mentioned this work forms part of a larger ongoing project around creating and persisting Personal Knowledge Graphs. It is anticipated these PKGs will be stored in Solid Pods[17], enabling the owner to provide controlled access to the resources within. This could be particularly useful for sharing genealogical information with family members, saving them the requirement to generate their own records for the same family members and also allowing the data to be supplemented with information from other users.

Bibliography

- [1] Ivo Velitchkov, George Anadiotis: *Personal Knowledge Graphs*. 2023. <https://personalknowledgegraphs.com/>
- [2] FamilySearch: *GEDCOM: The Genealogical Data Standard*. 2019. <https://www.gedcom.org/index.html>
- [3] Jo Pol: *gedcom2sem*. 2019-11-06. GitHub <https://github.com/jo-pol/gedcom2sem>
- [4] Bruce Wheaton: *GedcomRDF*. 2013-06-03. GitHub <https://github.com/BruceMWhealton/Gedcom-RDF>
- [5] Zmario: *The Waltons Wiki*. 2015-11-08. https://thewaltons.fandom.com/wiki/The_Waltons_Wiki
- [6] FamilySearch: *GEDCOM X*. 2023. <http://gedcomx.org/Specifications.html>
- [7] FamilySearch: *GEDCOM 5.5 to GEDCOM X Converter*. 2024-04-01. GitHub <https://github.com/FamilySearch/gedcom5-conversion>
- [8] FamilySearch: *gedcomx-java*. 2024. GitHub <https://github.com/FamilySearch/gedcomx-java>
- [9] HyperDiary: *GEDCOM Transformer*. 2024. GitHub <https://github.com/hyperdiary/gedcom-transformer>
- [10] Ben De Meester, Thomas Delva, Pieter Heyvaert: *RDF Mapping Language (RML)*. 2022-11-16. <https://rml.io/specs/rml/>
- [11] Souripriya Das, Seema Sundara, Richard Cyganiak: *R2RML: RDB to RDF Mapping Language*. 2012-09-17. <https://www.w3.org/TR/r2rml/>
- [12] Dylan Van Assche, Ben De Meester, Pieter Heyvaert, Anastasia Dimou: *YARRRML*. 2023-01-26. <https://rml.io/yarrrrml/spec/>
- [13] Jonathan Robie, Michael Dyck, Josh Spiegel: *XML Path Language (XPath) 3.1*. 2017-03-21. <https://www.w3.org/TR/xpath/>
- [14] RDF Mapping Language (RML): *YARRRML Parser*. 2024. GitHub <https://github.com/RMLio/yarrrrml-parser>
- [15] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, Gavin Carothers: *RDF 1.1 Turtle*. 2014-02-15. <https://www.w3.org/TR/turtle/>
- [16] RDF Mapping Language (RML): *RMLMapper*. 2023-12-06. GitHub <https://github.com/RMLio/rmlmapper-java>

[17] OpenRefine: *GREL Functions*. 2024-01-12. <https://openrefine.org/docs/manual/grelfunctions>

[18] The Solid Project: *Solid*. 2024. <https://solidproject.org/about>

Bridging XDM types in multiple native type systems

O'Neil Delpratt

Saxonica

<oneil@saxonica.com>

Matt Patterson

Saxonica

<matt@saxonica.com>

Abstract

We explore the relationship of XDM types and the native types in the host language of an XML processing system. In multi-tier language systems such as SaxonC we find that there is not always that one size fits all approach to representing data of a native type to what we require in the XDM type system. Secondly, we look at the complexities of handling numbers and strings; it seems simple to represent them across languages and within the XDM system, but this we found can get cumbersome and complicated. Thirdly, we consider the more complex XDM Map, and how issues of implicit and explicit type conversion meet issues of XDM and local idiom. Lastly, we dive into the use case of handling XDM Node objects such as traversing, cross language memory management (i.e. from Java to C++ and vice versa). On top of that we discuss how we add further complexity in layering C++ extension to support APIs in Python and PHP which operate in a managed code environment again.

Keywords: XML, XSLT, XQuery, XPath, C++, Python, PHP, GraalVM, XDM

1. Introduction

The XML processing languages XPath, XSLT and XQuery have in common and at their core the XQuery and XPath Data Model (XDM). Over the years, the XDM has been through several iterations, with increased complexity along the way. Firstly, we had the XPath data model 1.0, which primarily focused on the tree structure of the XML document. The data model was composed of seven types of nodes (root, element, text and attribute). Strings were determined from the type of the node. At the same time we had XSLT 1.0, which was based on this data model with additional features.

Secondly, we have the XQuery 1.0 and XPath 2.0 Data Model given as a specification. Here the 1.0 and the XSLT 1.0 data models were combined with additional data types to support more than just trees. These come from the XML schema simple types and what we call atomic values, primitive types such as `xs:string`, `xs:boolean`, `xs:integer`, and `xs:date`. We also have the notion of a sequence which is a collection of zero or more items. These items can be nodes, atomic values or a combination of both.

Thirdly, we have the XQuery and XPath Data Model 3.1 which is based on the 2.0 specification. What is new in this specification is the addition of array and map types, which are derived from the function type, which is also derived from an item. At the time of writing, the XQuery and XPath Data Model version 4.0 specification is currently at draft status.

In this paper, we look at the challenges of providing multiple language APIs to Saxon, in terms of both type system and API design constraints, and some of the lower-level concerns necessitated by the ways GraalVM manages bridging of managed Java objects with the unmanaged world of C/C++ and beyond.

As a background, it is important to mention here the building blocks of SaxonC. We use GraalVM, a JVM implementation that provides the ability to compile Java down to native code ahead-of-time, and allows that code to be distributed as a shared library. Effectively a tiny VM implementation runs in a thread in the library and allows your native code to call into it. We create a native library using GraalVM's native-image technology, which we call SaxonC. This is compiled from Saxon-J (a pure java implementation) so that we can make calls from other languages such as C/C++ to the Java code which is now native. The binary library comes packed with all the compiled JDK libraries, and a VM, required to run the Saxon-J image for execution. Creating a native image has the benefits of having faster startup time, low resource usage, flexibility in extending the application with other languages, and being easier to package.

2. Exploring the gap between XPath and native types

APIs that permit working directly with XDM data – that is, objects within the XDM type system – outside an XQuery/XPath processor require bridging XDM and native types. XDM has a different approach to data at a fundamental level from most of the languages that Saxon runs on, the languages you will use to bridge into XDM. This makes things hard. Apple's AppleScript programming language was designed to be easy to understand for non programmers, and leant heavily into an english-like syntax. One consequence of this was that although it was easy to read, in many ways that made it very hard to write. Author Matt Neuberg called this the 'English-likeness monster' [4]. XDM's pervasive use of Sequences is one of the main points of disconnect between the XDM type system and most native type systems. I call this the 'Sequence-likeness monster'.

2.1. The Sequence-likeness monster

The introduction of Sequences with the introduction of the XPath Data Model and XPath 2.0 is extremely important. Extending the XPath 1.0 Nodeset concept to things that weren't XML Nodes gives XPath much of its power, but also introduces the biggest gap in native and XPath type systems. Let's look quickly at a diagram showing XDM Values, Items, and Atomic Values:

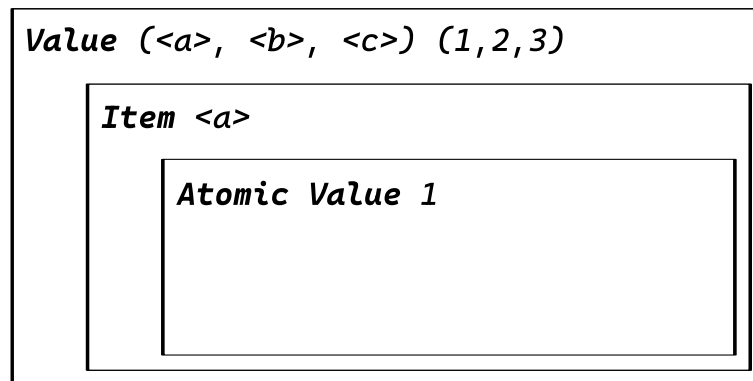


Figure 1. The XDM supertypes

A Sequence is a **Value**, an XML Node is an **Item**, and a primitive type, like a String, is an **Atomic Value**. But, Atomic Values are also Items, which are, in turn, also Values. This is a kind of inversion of more traditional object and type hierarchies, where a primitive type may be a kind of **Object**, but is certainly not a kind of meta-collection type (like Value):

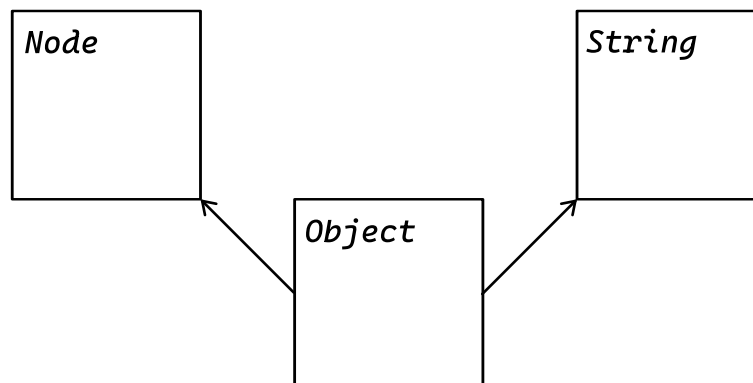


Figure 2. A 'Traditional' Object-based type system

Because of this type hierarchy all primitive types are also a kind of Value – a Sequence – even if they can only ever contain 1 item (themselves). Thought about in terms of the type system of a dynamic language like Python, all Integers in XDM are also a kind of List-ish thing containing a single member. In fact, everything in XDM is also a kind of List-ish thing, in addition to whatever else it is. In

some ways Sequences are the simplest XDM type, since, unlike a String, they don't have a concrete value, they're just a collection.

For obviously alien objects like XML Nodes, there is, perhaps, less dissonance when dealing with these objects in a non-XPath language. They are obviously unlike a String, or other native primitive type, and so it's more obvious that interacting with, and manipulating, them is unlike working with a core native type.

Dealing with creating Atomic Values (such as parameters required by an XSLT), or processing Atomic Values returned by an XPath evaluation, XQuery, or XSLT invocation is much more slippery. Unless you can guarantee what return type, perhaps from the type signature of an XPath function or XSLT template, you have to figure it out at runtime. More critically for this paper, the implementer of the native bindings for the XDM processor, have to figure it out as well, and then they need to provide meaningful mappings between XDM Atomic Values and native types.

XPath 1.0 had a significantly simpler data model, but even then this was a complex issue. JAXP's XPath evaluation methods return String or Object, and the user needs to know in advance what kind of thing their XPath will return (a node, a sequence, a primitive value) and pass it to the evaluation method as an additional argument.

Purely from the point of view of the user, XPath 3's richer data model, and increased dynamism (higher order functions and `fn:transform`, for instance) make it hard for implementers to expect users to know the return types of XPath expressions they might need to execute in advance, and, from the implementer's point of view, those same issues make it impossible to avoid dealing with the complexity of dynamic return types, whether the host language is dynamically or statically typed.

2.2. Dynamic and Static typing

XPath is a dynamically typed language. It's possible to specify strong requirements for parameter and return types, but it's not required in all situations. Many host languages, such as Java or C++, are statically typed. A situation where code in the host language is used to invoke an XSLT that is specified at runtime, and whose parameters (and their types) are therefore also not known until runtime is not an unusual one. Neither is a situation where an XSLT's return value differs in type based on the content it's processing unusual.

With a dynamically typed language there's an expectation of, and language support for, returning multiple possible types from any given function or method. A user is likely to be comfortable with the idea that they'll have to determine exactly what they got back at runtime, and process accordingly.

With a statically typed language, that's much more difficult. Without knowing the type in advance, a user must use a native type that represents an XDM super-

type like `Value`, and an implementer, like `Saxon`, can only return more specific types if they add extra type-specific invocation methods.

2.2.1. Creating XDM objects

The primary need for users of an XML processing system to create XDM objects is to hand them to the processor for processing, usually as parameters, as when invoking an XSLT transformation. Of course, parsing an XML document itself into an XDM Node is also an example of this, but we're concerned here primarily with XDM objects other than document trees.

There are two ways to create an XDM object from a native object. The XDM type can be inferred from the native type, implicit conversion. At it's simplest, this could mean converting a native `String` to an Atomic Value of type `xs:string`. A complex example is converting a JavaScript object to an XDM Map. Here's an example using `SaxonJ`'s implicit conversion:

```
import net.sf.s9api.XdmAtomicValue;
new XdmAtomicValue("string"); // => An xs:string
int v = 1; new XdmAtomicValue(v); // An xs:int
double v = 42.0; new XdmAtomicValue(v); // An xs:double
new XdmAtomicValue(LocalDate.now()); // An xs:date
```

And using `Saxon JS`:

```
SaxonJS.atom(42.0); // An xs:double
SaxonJS.atom(true); // An xs:boolean
SaxonJS.atom("string"); // An xs:string
```

The other approach is explicit conversion, where the XDM type is specified along with the native value to be converted or a function that generates a specific type is called. Here's an example using `SaxonC`:

```
#include "XdmAtomicValue.h"
double d = 42;
XdmAtomicValue *value = XdmAtomicValue::makeDoubleValue(d);
```

And using `SaxonC Python`:

```
from saxonche import PySaxonProcessor
proc = PySaxonProcessor()
value = proc.make_double_value(42.0) # as xs:double
value = proc.make_atomic_value("date", "2024-06-08") # an xs:date
```

The last example above illustrates a variant of explicit conversion whereby the type is specified and the value is given using its XML Schema lexical string representation (like XML element or attribute content that was typed using an XML Schema).

2.2.2. Converting XDM object into native objects

Converting XDM objects into native objects is where the differences between static and dynamic typing are more apparent. Given an XPath evaluation that returns an XDM Value that we know will return a single Item, but the exact return type of the Item is not known, what can be done to convert that into a native object of an appropriate type?

With a statically typed language, about the best we can do is to provide methods on the various Value and Item objects / subclasses that report what kind of thing they are, along with methods to produce a specific native type, like a long or boolean. Then, we can use those methods in combination with branching logic to get values with correct native types out:

```
SaxonProcessor *processor = new SaxonProcessor(false);
XPathProcessor *xpathProc = processor->newXPathProcessor();
xpathProc->setLanguageVersion("3.1");
xpathProc->setContextItem((XdmValue*) input);
XdmItem *result = xpathProc->evaluateSingle(".[last()]");
if (result.isAtomic()) {
    switch(result.getPrimitiveTypeName()) {
        case "Q{http://www.w3.org/2001/XMLSchema}int":
            long num = result.getLongValue();
    }
}
```

This is complex and verbose. With a dynamically typed language there's more scope for API convenience for the user. Saxon JS is designed for very close integration in a JavaScript runtime, so calls to `SaxonJS.XPath.evaluate` and `SaxonJS.transform` will return results where the XDM objects are converted into JavaScript values, following the rules set out in its documentation (see [1]). These rules are complex, and neatly illustrate the trade off between convenience for the user and complexity for the implementer.

```
Object.is(SaxonJS.XPath.evaluate("true()"), true) // => true
SaxonJS.XPath.evaluate("map { 'a': 1 }") // => {a: 1}
SaxonJS.XPath.evaluate("map { 'a': 1 }")['a'] // => 1
```

2.3. Numbers

We've talked about the issues of moving between native types in a Saxon API host language, and native XDM types in terms of differences in the type systems, but the other issue is simply the different ways most programming languages and XDM think about how their primitive types work, and what shape they are. It's worth looking in more depth at the conversion of numeric types. Numeric types are (perhaps surprisingly) complex to deal with. XDM's numeric types owe a lot to C and Java's numeric types. As a result there are a lot of different numeric

types. The following two tables shows XDM's numeric types, and the types for Java, C++, Python, and JavaScript.

Table 1. XDM numeric types

```
xs:float
xs:double
xs:decimal
  xs:integer
    xs:nonPositiveInteger
      xs:negativeInteger
xs:long
  xs:int
    xs:short
      xs:byte
xs:nonNegativeInteger
  xs:unsignedLong
    xs:unsignedInt
      xs:unsignedShort
        xs:unsignedByte
xs:positiveInteger
```

Table 2. Other languages

Java	C++	Python	JavaScript
NumericType	bool	int	Number
IntegralType	char	float	
byte	signed char	complex	
short	unsigned char	fraction.Fraction	
int	wchar_t	decimal.Decimal	
long	short		
char	int		
FloatingPointType	long		
float	long long		
double	unsigned short		
	unsigned		
	unsigned long		
	unsigned long long		
	long		
	float		
	double		
	long double		

The main thing that sticks out is that while there's some overlap between the type systems, the XDM numeric types are more numerous even than the C++ types, if you take into account the four C++ types that are integers representing characters, and comically more numerous than JavaScript's single `Number` type. There's no simple, 100% reliable type conversion possible. Distinctions like `xs:nonNegativeInteger` and `xs:positiveInteger` are effectively impossible to capture in any other type system, and, of course, what could you do if the number you need to pass into a processor is a Python `complex` number?

Converting between numeric types of different bit length is called *widening*, when converting from a smaller to a larger type (short to long, for example), and *narrowing* when converting from a larger to a smaller type, or from a floating point to an integer type (long to short, float to long). Widening doesn't lose information, but narrowing can. Java specifies how these kinds of conversions should work (see [2]), and C++'s *Core Guidelines* have a section on avoiding narrowing conversions (see [3]). The XQuery and XPath Data Model ([5]) and XML Schema Part 2: Datatypes ([6]) specifications don't really talk about these conversions at all. Casting from one type to another numeric type is covered briefly in

([7]), but these are largely implementation dependent questions. (Casting to `xs:integer` is mentioned, but not `xs:short` or `xs:byte`.)

Many conversions would require runtime checking of values to ensure that a given non-XDM numeric object's value fits within the allowed range of, say, an `xs:positiveInteger`, or an `xs:byte` when converting from a Python integer or JavaScript number. If the values are not checked then strange behaviours in the XPath / XQuery code could happen if there are dramatic silent truncations of input values, and implementations should raise exceptions if lossy narrowings occur.

2.4. How long is a (piece of) String?

Let's continue to look at the different ways XDM and other type systems shape their primitive values. Take the example of XDM's `xs:string` Atomic Value. Strings are interesting in this case because they are a data type that has intrinsic length (the number of characters they contain).

In the XDM type system, the following statements are all true:

- The length of the XDM Atomic Value representing the string "Hello World" is 11.
- The length of the XDM Atomic Value representing the string "Hello World" is 1.
- The length of the XDM Atomic Value representing a string containing 2^{10} characters is 1.

On its face, this seems ridiculous, but it shows up one of the ways XDM differs in the way it views the shape of data. In XDM, all Atomic Values are a kind of Item, but all Items are also Values, which are collections – Sequences. An Atomic Value is an XDM Value containing a single Item, a sequence of length 1, in other words. So, the Atomic Value is both a Sequence containing a single item, and a primitive value containing a number of codepoints.

In XPath this isn't a particular problem - to get the length of a string in XPath you use `fn:string-length()`, and `fn:count()` to get the length of a sequence. In host languages which are more traditionally object-oriented, you would expect to query the Atomic Value itself to find out that information.

In Python, for example, the length of a string is simply the number of codepoints it contains. There is no sense that this primitive type could also be thought of as a container collection of itself.

XDM strings are Unicode strings, which is not true in C and C++ (and used to not be true in Python). Wanting to pass an XDM string into a processor and wanting to get a non-unicode string (perhaps just the string's representation as UTF-8 encoded bytes) makes working with XDM string objects trickier and the design of an API for wrapped XDM objects which supports making use of the them as

XDM objects (iteration over sequences, perhaps) and as containers of native or native-like objects (wanting to get substrings or string lengths) needs implementers to consider the idioms of both XDM and the host language carefully.

2.5. How do you get a value from an XDM Map in Python?

Maps in XDM are also Atomic Values, and Maps are widely used in other languages. Wanting to expose an XDM Map in a host language involves making choices about what idioms to use, and which to discard. XDM Maps are immutable, unlike maps in most languages, which introduces API friction when creating an XDM map, especially in languages where creating a map all-at-once with a literal syntax is either not especially idiomatic, or just plain impossible.

When fetching a value, we need a key. What is the key? In the most trivial case, paralleling a JSON Object, the key is an XDM Atomic Value with type `xs:string`. Next, how do we construct that Atomic Value? Can we allow a Python user to call a method on the Map and pass in a Python string? Do we need to require explicit construction of an `xs:string` Atomic Value? If we allow native Python primitives, do we require separate API methods for getting values using Python types versus using XDM Atomic Values?

As we can see, even in this trivial example there are a host of questions which need answering, none of which are themselves trivial.

If we allow Python primitive values to be used in place of XDM Atomic Values in the Python API equivalent of `map.get()`, how do we convert them?

The end goal is to have Saxon construct an Atomic Value – a string – whose contents match the Python string we started with. We need to extract the bytes of the Python string, pass that C-level byte array into the Java internals using the GraalVM C API, and construct an Atomic Value from that Java string.

Once we have an Atomic Value, we can call the `get()` method of the XDM Map with the key as its argument, and hand back the resulting value.

But what is that value? When invoking an XSLT via the API it's very easy to overlook this question: the result of the transformation is written to a file, or perhaps serialized to a string, neither of which require dealing with an XDM Value. When retrieving a value from an XDM Map, the result is an XDM Value. Should this be converted to an equivalent primitive type in Python, so that an `xs:string` value becomes a Python string? What about more exotic XDM Value types? What do you do with an XDM Function Item?

If we consider Pythonic idioms, we would expect the map to respond to the `[]` operator to return values, as with `value = xdm_map[key]`. Of course, considered as a sequence, we would expect the map to respond to the `[]` operator to return items from its 1-item sequence: itself in other words.

```
xdm_map is xdm_map[0] # => true
```

This is a problem for the API designer.

XPath 1.0 had a significantly simpler data model, but even then this was a complex issue. JAXP's XPath evaluation methods return `String` or `Object`, and the user needs to know in advance what kind of thing their XPath will return (a node, a sequence, a primitive value) and pass it to the evaluation method as an additional argument. XPath 3's richer data model, and increased dynamism (higher order functions and `fn:transform`, for instance) make it harder to get away with requiring the user to have to know the return types of their expressions in advance.

2.6. Unicode strings

It's also worth looking at the issue of Unicode strings more closely. In Python and XDM we can simply say that a string is a sequence of Unicode codepoints. (This wasn't always true in Python, but with Python 3 it is). How bytes get translated into Unicode codepoints is a matter of encoding. Conceptually at least, passing strings from Python into Saxon should be straightforward.

As we mentioned above, SaxonC is implemented using GraalVM's native-image feature. One of the things that means is that to pass data between our API host language (Python in this case) and the Saxon internals we have to use GraalVM's C API. We are used to thinking of strings purely in terms of Unicode, and the encoding of those strings into bytes is an internal concern of the XML processor, or of Python, but here we are forced to deal with it: C byte arrays have no intrinsic encoding, just bytes, and while you can get lucky most of the time by assuming that the byte arrays you're passing around contain UTF-8-encoded strings, this isn't always the case, and some library functions in Java vary the encoding they use based on OS or other locale settings, and of course Unicode representations in C are heavily dependent on locale settings or environment variables.

What does that mean for the case where Python strings (Unicode) need to be passed across the GraalVM API to construct XDM strings (also Unicode)?

Effectively every C-based call needs to pass the byte array and the name of the encoding used, or the GraalVM Java API functions that reconstruct Java Strings from byte arrays passed in will assume UTF-8, even if it isn't. The main SaxonC API provides XDM wrappers as C++ classes, but even there you need to specify as the GraalVM API is C, not C++.

The intrinsic Unicode-awareness of all strings within Saxon internals (simply as a side-effect of being implemented in Java) means that even strings you might not think twice about, like filenames, that need to be passed through the GraalVM C API must have their encoding specified explicitly to avoid bugs where problems appear, but only in certain circumstances, like on one OS, or with environment variables set to certain values. These can be a nightmare to deal with because they can be so hard to reproduce for debugging purposes. Likewise,

any Saxon APIs provided in another host language, like the Python API, need to make sure they supply encoding with string data when calling across GraalVM into Saxon proper.

3. Traversing XDM Nodes via the API

The navigation of XML trees in XPath is fundamental to most things we do with XML languages like XQuery and XSLT. The non-XML syntax in XPath is both powerful and succinct, but at the same time the XPath expressions for many tasks are simple to use for navigation. For example, traversing forward in XPath can be done as follows: `/doc, //person, /doc/person[2]/firstname`. Likewise backward traversal can be done as follows: `/expr/ ../ ..`, where the `expr` is a valid XPath expression.

Navigation of XML trees via an API to do the same simple task as an XPath expression is verbose and error prone, but is very powerful and useful because it provides a way to integrate into other multi-tier systems, extensions for programming languages and simply the flexibility to support user requirement where performance is a criteria. Often what we are dealing with is accessing the vendor's XPath implementation API at some lower level. There are a lot of comparisons and relationships that can be drawn with when navigating XML tree between the XPath languages and APIs. See below a C++ code snippet to access the child nodes and get the string represented at that node from a parent node object. Note that in this implementation it is the users requirement to delete the associated memory when finished.

```
int childCountA = node->getChildCount();
XdmNode **childrenA = node->getChildren();
XdmNode *child = childrenA[0];
XdmNode **children = child->axisNodes(EnumXdmAxis::CHILD);
int childCount = child->axisNodeCount();
for (int i = 0; i < childCount; i++) {
    const char *childStr = children[i]->toString();
    cout << "child node:" << (childStr) << endl;
    operator delete((char *)childStr);
}
for (int i = 0; i < childCount; i++) {
    delete children[i];
}
delete[] childrenA;
delete node;
```

There are more elegant ways to traverse XML trees in APIs even in C++, but these features are often restricted or best suited to certain programming languages, such as Python, Java and C#. For example, in Java we have functional interfaces

like the following streams-based APIs (similar techniques we can find in Python APIs and Linq in C#):

```
for (XdmNode pack : testInput.select(
    child("package").where(
        attributeEq("role", "secondary"))).asListOfNodes() {
    ...
}
```

In SaxonC on Python such a query we can write as follows:

```
packs = (pack for pack in testInput.children
    if (pack.name == 'package' and
        pack.get_attribute_value('role') == 'primary'))
```

This is equivalent to iterating over the result of XPath expression `package[@role='secondary']`, but it saves the cost of compiling the expression, which is often much greater than the execution cost. When writing XPath expressions the focus is the XML data we are querying, providing valid expression and its efficiency, but there is a lot happening under the hood by the implementation without the user needing to care much about how its done. For APIs your focus is not just on the XML data, but also efficiency in coding, memory management of the XML node objects created and how you represent the data as discussed in the previous sections on handling strings. This is in addition to the XPath and host language syntax and the actual data and paths you are navigating itself.

We now look at how we implement support for XML tree navigation in a C++ API. We chose C++ as the programming language here because its the path we have chosen for providing the basis for extension APIs in languages such as PHP and Python. The C++ API also sits on top of a Java API compiled to native using GraalVM. A traversal of an XDM Node as in a SaxonC application is in the context of XPath, XQuery and XSLT processing.

For something as simple as traversing node objects it becomes difficult to support when implemented in multi-tier languages. Multi-tier languages means interfacing with an application written in one programming language and its environment from another language. There is often further interfacing of languages. For example, in SaxonC, the porting of the application written in Java to C++ is achieved using GraalVM. PHP and Python APIs are built on top of the C++ API. We have some important questions that need answering: how do we make callbacks between the C++ and Java environments? and how do we hold on to objects in C++ that have been created in the Java space? Also, how could we prevent Java's garbage collection getting in the way when a Java object is still in use in C++?

The first question we will discuss with an example of getting a node and its child nodes. The second and third question we will discuss later as we explore the GraalVM's APIs, specifically the `ObjectHandle` and `ObjectHandles`.

3.1. Get a node, get its children

In the SaxonC C++ API, `XdmNode` objects are created by either parsing from some XML string, file or as a result of firing an XPath, XQuery or XSLT execution. As mentioned above we are calling back to the Saxon Java internals to create an in-memory representation of the XML document, which is a Java `XdmNode` object. This Java object is not directly returned to C++. Instead we use GraalVM's `ObjectHandle` API to return to C++. We will come back to talk about `ObjectHandle`, but for now, keeping things simple, we have a reference in C++, which allows us to get to the underlying Java object when we need to do some processing in the Java world. This we use to make callbacks to get child nodes in the following ways:

- `getChild(ith)` - Gets the *ith* child node from a current node
- `getChildren()` - Get all the child nodes from the current parent node
- `axisNodes(axis)` - Get the array of nodes reachable from this node via a given an XPath axis.

In the C++ code, for each case above we call back to Java, for example, to get the child node `getChild(ith)` we get the Java `XdmNode` child object from its parent node. Then we return a representation of that object to the C++ code (i.e. an `ObjectHandle` object from GraalVM). We have wrapper classes in C++ containing the `ObjectHandle` reference to the Java `XdmNode` object. At this stage, note that it does not matter how we represent the Java object in the C++ code, just realise that it can be complex. There are many other techniques for passing objects between languages. Some more efficient than others. For example, the Java native interface (JNI) is a common option to access the C API, but it can be slow and error prone. Another method is via native COM components, also web services via HTTP requests and other methods which are more complicated. If languages involved are in the same runtime (i.e. .NET, JVM), then it is possible to pass objects from one language to the other. For sure, implementers who are interfacing with an XPath implementations in multi-tier languages have to make the choice between these different methods.

3.2. GraalVM's `ObjectHandle` and `ObjectHandles` pool

GraalVM compiles a Java application into a native application that has its own executable. It also has the capability to create a native library which can be called from other programming languages, for example, directly using C/C++, or using the Truffle API for other programming languages such as Python, etc. For C/C++, GraalVM supports callbacks into the Java code using two different interface mechanisms. The first one is the Java native interface (JNI), which is a standard JDK API and secondly, directly through GraalVM's Native Image C API [9]. For

the latter, GraalVM exposes Java methods by marking the method with `EntryPoint` annotations which GraalVM interprets and creates C like interfaces to the methods. These methods are now available as export methods from the native library which can be used by C/C++ code. We make callbacks on the wrapped Java objects in the C/C++ API layer using the entry points, which we consider as being more efficient than using JNI.

We can easily make callback to the Java object from C++ and work with primitive Java/C++ types. But to manage Java objects from C++ is not so easy and preventing objects getting garbage collected (GC) by the Java JVM ahead of time before the object use is finished in the unmanaged C++ code is an issue. Java objects are held as references in an intermediate C++ interface. To achieve this we use GraalVM's `ObjectHandle` API. A `ObjectHandle` is an opaque representation of a handle to a Java object which is given out to unmanaged code (i.e. C/C++). We also use GraalVM's `ObjectHandles` API as a pool of `ObjectHandle`, which is a managed set of `ObjectHandles` to keep alive the Java objects in memory and prevent them from being garbage collected.

Given our node traversal example: We get a child node in Java, create an `ObjectHandle` for it, add it to the `ObjectHandles` pool, and then return the `ObjectHandle` reference to C++ which will then get wrapped in a C++ `XdmNode` object. Another layer of wrapping will take place if we are in a Python or PHP application. This child node object will stay alive until we remove it from via `ObjectHandles` set.

3.3. Telling GraalVM you are not using an object

When working with XDM node objects in processors such as XSLT, XQuery, XPath and Schema Validator from C++/Python/PHP languages it is not always obvious when the object is no longer needed. This is because references can be held as external variables or structures, internally in the processors or a combination of both. As mentioned before the `ObjectHandles` API prevents the GC deleting the object.

We must make sure that deallocating the C++ `XdmNode` object which wraps an `ObjectHandle` cascades down to remove the associated Java object held in the `ObjectHandles` pool, else we have memory leaks with objects that never get deleted. Removing an object from the `ObjectHandles` pool will make the object available for GC. Languages such as Python and PHP have their own GC. These also need to inform the internal program when objects are to be freed.

3.4. How do you know when you're not using an object?

Processors hold and then release XDM objects that were used in execution, for example, used as a parameter or context item. In the C++ environment it is mostly

the users responsibility to deallocate the memory associated with objects they no longer need. To solves this we add internal hooks (i.e. reference counting) on objects if they are still indirectly required by processor. We then hand over the deallocation responsibility to the internals of the processor object.

In the PHP and Python world it is somewhat more complicated because we are in a managed code environment which has its own GC. Here we have to add our own reference counting techniques to ensure that PHP or Python's GC doesn't cause GraalVM's GC to either be called too early, or not to be called at all. Likewise, if for example an XDM node object is no longer used in the Python script the reference count for that object at that point should be zero. Therefore we can safely delete that object. At the same time we expect that child nodes which have been created exist independent of the parent node and therefore are relinquished to the Python GC, which will have its own reference counting checks.

4. Memory Management

Managed code, in simple terms, is code that is managed at execution time. In Java's JVM the code is translated into an intermediate language which is interpreted and executed. The entire memory management is taken care of by the runtime using the garbage collection feature.

Unmanaged code is when the code is compiled to native code or machine code and executed by the operating system. The whole memory management of the program is unmanaged; therefore it is the user's responsibility to handle the memory allocated by the program throughout its span, and delete it when no longer needed and when the program is terminated. Correct memory management is important to avoid errors and memory access violations such as segmentation faults.

The management of memory in C++ is fundamental. There are two aspects we would like to talk about here: Firstly, how we keep alive objects such as XDM Node objects in main memory. And secondly, how we create and return strings from the internal Java code which needs to be returned in C++. For example, serialising nodes, returning the XSLT transform as a string and executing an query to string.

In SaxonC, for XDM node objects we have our own memory management where we keep a reference counter in the objects for where the object is used. We use this to prevent the object from being deleted too early. This is particularly important in extension languages like Python and PHP which have their own GC. If the XDM object is internally referenced by processors and variables then we keep alive these XDM node objects using the reference counter, until they are no longer referenced.

For the textual data that we create in the Java code and which needs to be returned to the C++ API we create this data directly in the C++ memory space. In essence, we are allocating the C++ memory space from managed code (i.e. Java). This we found to be the more efficient because we are not having to make unnecessary copies of the data. The pointer to the string is returned to the user with the responsibility to deallocate the associated memory when finished with the data.

5. Conclusion

Providing meaningful API access to XML processing with Saxon requires that API users can construct and make use of XDM Values. In host languages other than Java or JavaScript, this means directly or indirectly using SaxonC, with its GraalVM-C-API-based bridge between Java and C/C++.

The XDM view of the world is very different to the way that most host languages structure data. XDM wrappers that bridge between the worlds need to be a first-class part of any API, otherwise it's extremely difficult, or impossible, to do anything other than simply invoking a transform or query. Passing parameters or arguments in to a transform or XPath evaluation as lexical XPath strings, as in the XPath 1.0 days, is no longer viable.

Providing first-class XDM wrappers to API clients in other languages for Saxon through SaxonC requires wrangling the expectations of managed code in the core Java runtime, and the managed code in the host language runtime through the unmanaged layer of GraalVM's C API and SaxonC's own C++ classes. Correctly managed, this allows, for example, a Python user to make use of XSLT 3.0, and XPath/XQuery 3.1 in a way that feels idiomatically Pythonic, and more generally expands the base of developers who can make use of modern XML technologies. We still have a lot of room to improve, and we hope that this survey of some of the higher-level challenges and lower-level engineering will be useful to other implentors and users, as well as ourselves.

Bibliography

- [1] Saxonica: *Type Conversion between JavaScript and XDM*, <https://www.saxonica.com/saxon-js/documentation2/index.html#!xdm/conversions>
- [2] James Gosling, Bill Joy, et al. *The Java Language Specification, Chapter 5. Conversions and Contexts*, <https://docs.oracle.com/javase/specs/jls/se22/html/jls-5.html>
- [3] Bjarne Stroustrup and Herb Sutter, editors *C++ Core Guidelines, ES.46 Avoid lossy (narrowing, truncating) arithmetic conversions*, <https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es46-avoid-lossy-narrowing-truncating-arithmetic-conversions>

- [4] Matt Neuberg *Applescript: The Definitive Guide 2nd edition*, <https://www.oreilly.com/library/view/applescript-the-definitive/0596102119/ch04s03.html>
- [5] Norman Walsh, John Snelson, and Andrew Coleman, editors *XQuery and XPath Data Model 3.1*, <https://www.w3.org/TR/xpath-datamodel-31/>
- [6] Paul V. Biron and Ashok Malhotra, editors *XML Schema Part 2: Datatypes Second Edition*, <https://www.w3.org/TR/xmlschema-2/>
- [7] Michael Kay, editor *XPath and XQuery Functions and Operators 3.1 § 19.1.2 Casting to numeric types*, <https://www.w3.org/TR/xpath-functions/#casting-to-numeric>
- [8] GraalVM, <https://www.Graalvm.org/>
- [9] Kevin Menard: *Embedding Truffle Languages*, on May 9, 2022 <https://nirvdrum.com/2022/05/09/truffle-language-embedding.html>
- [10] Abel Braaksma (Exselt): *Writing more robust XSLT stylesheets by understanding and leveraging the XDM data model*. XMLPrague 2019, Prague, 7-9 February 2019 <https://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf> <https://youtu.be/Y2sRDh-ymBU?si=jgLeE269SwxbYd97>
- [11] Rennau, Hans-Jürgen, and David A. Lee. *XDML - an extensible markup language and processor for XDM*. Presented at Balisage: The Markup Conference 2011, Montréal, Canada, August 2 - 5, 2011. In *Proceedings of Balisage: The Markup Conference 2011*. Balisage Series on Markup Technologies, vol. 7 (2011). <https://doi.org/10.4242/BalisageVol7.Rennau01>

natural-xml-diff: an XML Diffing Library

Martijn Faassen

Paligo

<faassen@startifact.com>

Abstract

natural-xml-diff is a software library written in the Rust programming language that produces a structure-aware difference between two XML documents. The diff is aimed to be human-readable and is produced efficiently for typical document-style content. natural-xml-diff creates a document describing the difference, not just a sequence of edits, and this diff document is post-processed to further improve the human readability of the diff.

1. Introduction

natural-xml-diff is a software library written in the Rust programming language that produces a structure-aware difference between two XML documents. The diff is aimed to be human-readable and is produced efficiently for typical document-style content. Here is a simple example:

We have a source document:

```
<doc>
  <a/>
  <b/>
</doc>
```

And a target document:

```
<doc>
  <a/>
  <b/>
  <c/>
</doc>
```

As we can see, the `c` element has been inserted compared to the source document.

natural-xml-diff expresses the difference as follows:

```
<doc xmlns:diff="http://paligo.net/nxd">
  <a/>
  <b/>
  <c diff:insert=""/>
</doc>
```

The target document can be constructed from this diff document by applying the insertion (simply removing the `diff:insert` attribute). The source document can also be constructed by removing the element with the `diff:insert` attribute.

`natural-xml-diff` supports all types of XML content, elements, text and attributes, and also comments and processing instructions. Information about cdata sections and custom entity references is not retained: these are normalized to normal text nodes during parsing.

`natural-xml-diff` reuses the JNDiff algorithm as its core algorithm, but improves on it in various ways. `natural-xml-diff` has a well-tested, robust implementation.

2. Goals

The goal of `natural-xml-diff` project was to create a production-ready library that supports high-quality human-readable diffs for document-style XML, in particular Docbook, though `natural-xml-diff`'s implementation is XML format agnostic.

The diffing algorithm should perform well for reasonably-sized documents, and not crash given correct XML input. It should produce correct output.

Not all correct diffs are equal. It is easy to create a correct diff between document A and B by deleting all the content of A and then inserting the complete content of B, but this is not very practical or semantically useful. What human authors and editors want is a diff that is useful at a human level. This means the diff should be similar to how a human author would express it, like “I inserted a word after this word” or “I deleted this paragraph”. Diffs should also be minimal, but not at the cost of semantic usefulness: inserting and removing of entire words can for instance be easier to understand than more compact character-level diffs.

3. Diffing Annotations

Diffing operations are represented inline in an XML diff document. Both the source document as well as the target document can be reconstructed from this XML diff document. These reconstructions have the same normalized XML content, though they may differ in details of serialization. All annotations are in the diff namespace <http://paligo.net/nxd> and take the form of elements and attributes. What follows should give an impression of what `natural-xml-diff` supports.

3.1. `diff:same`

Attribute (with empty value) added to the document element if the documents contain no differences at all.


```
<doc diff:same="">...</doc>
```

3.2. **diff:insert** and **diff:delete**

Attributes (with empty value) added to an element if it is either inserted into source, or deleted from source.

```
<doc><p>One</p><p diff:delete="">two</p><p>three</p></doc>
```

3.3. **diff:text-insert** and **diff:text-delete**

Marks insertions and deletions of text. Their text content is the inserted or deleted text from the source compared to the target. It can be configured whether this is a character-level or word-level diff.

```
<section>
  <p>This is a paragraph with <diff:text-insert>new text</diff:text-
insert> and
  <diff:text-delete>deleted text</diff:text-delete>.</p>
</section>
```

3.4. **diff:wrap** and **diff:unwrap**

If enabled during diffing, diffing is marked up with inserted elements to wrap an existing element. For instance:

```
<p>Hello <b diff:wrap="">world</b>!</p>
```

means that a new `b` tag was inserted around the text “world”. `diff:unwrap` is the inverse operation, where an element is removed.

```
<p>Hello <b diff:unwrap="">world</b>!</p>
```

If this feature is turned off, wraps and unwraps are represented as delete/insert combinations.

3.5. **diff:attributes**, **diff:attr-update**, **diff:attr-insert**, **diff:attr-delete**

`diff:attributes` is an element that represents inserts, updates and deletes of attributes in its parent element.

Here is an example of an update, where the element marked with `attr-update` has the name of the attribute that was updated, and its content is the

value of the attribute before the update. The attribute itself contains the updated value:

```
<section a="A!" b="B">
  <diff:attributes>
    <a diff:attr-update="">A</a>
  </diff:attributes>
  <p>Txt</p>
</section>
```

3.6. diff:pi-insert, diff:pi-delete, diff:comment-insert, diff:comment-delete

These are custom diff elements that let you express a deleted or added comment or processing instructions:

```
<diff:pi-insert><?my pi?></diff:pi-insert>
```

3.7. diffing a diff document

You can diff the output of `natural-xml-diff`. If a document is detected to be a `natural-xml-diff` document, a new `diff-meta` namespace prefix is used for the diffing annotations, with the `http://paligo.net/diff-meta` namespace. If you diff a diff of a diff (read that carefully!), the prefix `metametadiff` and the URI `http://paligo.net/diff-meta-meta` is used, and so on.

4. Prior Art

How to implement semantically useful diffing for structured documents is not obvious. When we set out to create `natural-xml-diff` we wanted to learn about the best approaches to implement this algorithm.

We had previously examined `xmldiff`¹, a Python project that can produce diffs between XML documents. Unfortunately we had identified a number of cases where it produced correct but far from human-readable diffs. We also found that its performance degraded significantly with even moderate document size.

The `xmldiff` project has an interesting history - while it has been in existence since 2004, it had been completely rewritten in 2018. Both original and reimplementation are based on Chawathe et al. (1996) , a paper written before the emergence of XML. This paper describes a diffing algorithm for hierarchical information. It does not aim for human-readable diffs or document-style content, and the focus is on the performance characteristics of the algorithm itself and the

¹ <https://pypi.org/project/xmldiff/>

minimality of the produced diff Barabucci et al. (2016) . Minimality is only one aspect of a useful diff, and the easiest one to measure, but does not indicate much about usefulness to human authors.

We then discovered the overview paper Cuculovic et al. (2020) . The authors performed a comparison study of 12 different XML diff implementations that were state of the art in 2020, with a focus on semantically useful diffs. This was exactly the kind of survey we needed. A previous comparison study Cobena, Abdessalem and Hinnach (2002) was less useful, as its quality comparison focused on minimality, remarking that the semantic value cannot be easily captured.

From the Cuculovic et al. (2020) , it became clear that the JNDiff² algorithm, described by Ciancarini et al. (2016) and Di Iorio et al. (2009) , was the most promising candidate for further evaluation. The same group produced several studies into a qualitative analysis of diff algorithms, Barabucci et al. (2013) , Barabucci et al. (2014) , and Barabucci et al. (2016).

A more recent implementation, *jats-diff*³, is described by Cuculovic et al. (2022). This implementation is based on the original Java source code of JNDiff (including comments in Italian) and adds a new layer for further refinement and cleanup of the diff.

5. Implementation Considerations

JNDiff's implementation status was not conducive to adoption for production use: the source was last updated close to 10 years ago at the time of writing, it has comments and API documentation in Italian, a language we do not speak. In addition, later on in the project we discovered some XML structures for which JNDiff produced the wrong output, as we will discuss later.

The far more recent *jats-diff* tool at first glance looked more promising, but unfortunately in our testing this algorithm failed to complete in a reasonable time on some tests inputs, and thus did not appear to have the performance characteristics we required. The maintainability status of the *jats-diff* software is also in question given its reliance on the original JNDiff source code, including Italian comments. The fact that *jats-diff* extends JNDiff with additional semantically rich operations such as a move operation is interesting, but representing these advanced operations in a user interface provides significant challenges. We decided therefore to focus on the smaller subset of operations as described previously.

The JNDiff algorithm as described by Ciancarini et al. (2016) as well as the JNDiff Java source code still provided the best starting point for our efforts. In order to make this algorithm useful for production, we decided to modernize and improve the JNDiff algorithm by reimplementing it from the ground up in the

² <https://sourceforge.net/projects/jndiff/>

³ <https://github.com/milos-cuculovic/jats-diff>

Rust programming language. We chose Rust because the algorithm is computationally intensive, and Rust lends itself well to performant software that is robust. We found the Java implementation difficult to follow due to a reliance on class inheritance. In our Rust implementation we have implemented it in a more procedural fashion, which we believe is easier to understand and maintain. The reimplementation is extensively documented.

6. JNDiff Algorithm

We give a sketch of the JNDiff algorithm here, but for more details see Ciancarini et al. (2016).

The key insight of JNDiff is that long subsequences of identical content can be used to construct semantically useful diffs. Given an XML document linearized in document order, JNDiff finds the longest subsequence of XML nodes that are the same in both the source and target documents, and then partitions the document in two halves, one before the common sequence, and one after. It then recursively applies this algorithm to the two halves. In the end, a number of partitions are found where the two documents are identical, favoring the longer stretches of identical text.

Between these partitions there are sequences of nodes that are not identical, but may still be considered updates of each other. JNDiff assumes that a gap in the source and a gap in the target both placed between the same common subsequences are likely to contain updates. The motivation is that updates are likely to remain in the same relative location in an XML document, so that updates can best be found between ranges of equal nodes.

An update pair of nodes is found by going through all text nodes in the gap in the source, and then looking for the best match in the gap in the target by looking at the levenshtein distance of the text nodes, a common text similarity algorithm. After update pairs are found, the update range is partitioned once again in two halves, and the update algorithm is applied recursively.

We now have obtained a collection of equal and update node pairs between the source and the target document. We propagate this information upwards in the tree to ancestor nodes - if two nodes have equal or updated children, we consider the parent nodes as equivalent. Nodes with more text ("weight") are favored over nodes with less text, so it's more likely equivalent nodes are matched up.

Any remaining nodes are considered different. If a different node exists in the source, it is taken as deleted. If a different node exists in the target, it is taken as inserted.

7. **natural-xml-diff** Improvements

7.1. JNDiff Problems

As previously mentioned, we found a fundamental problem in the JNDiff algorithm that we needed to resolve.

Given these two documents, with source document:

```
<doc>
  <a>
    A<b>B</b>
  </a>
</doc>
```

and target document:

```
<doc>
  <a>A</a>
  <b>B</b>
</doc>
```

the JNDiff algorithm considers these to be identical, even though there is a clear structural difference, where the `b` element was deleted in `a` and inserted inside of `doc`, because the values (text and element nodes) are the same, and appear in the same order.

Because the values are the same, the JNDiff algorithm propagates this information upwards to the parent node, `doc`, and this is considered to be the equivalent node as a result - it is not inserted or deleted. As a result no differences are left between the two documents and thus source and target are considered to be identical!

`natural-xml-diff` detects such structural differences by looking at parent consistency: if a node in the source document is considered equivalent to a node in the target document, but those nodes do not have the same parent as determined by the algorithm, this is detected and the node is reconsidered to be different after all.

7.2. Ignorable whitespace

Whitespace changes to an XML document are common. Often, whitespace changes happen to make the XML document easier to read, but is not part of the semantic content of the document. Other whitespace changes are significant, such as that which occurs in mixed XML content. `natural-xml-diff` can be configured to ignore insignificant whitespace, and therefore can produce easier to understand diffs.

7.3. Creating a Diff Document

While JNDiff has an optional feature to create the equivalent of the `natural-xml-diff` diff document through its JNMerge tool, the JNDiff algorithm stops once it has created a sequence of edits to produce the target from the source. `natural-xml-diff` also produces this sequence of edits, applies them to produce the diff document, and then proceeds to do further post-processing on the produced diff document to improve its semantic quality. This post-processing can produce significantly better diffs.

7.4. Element Wraps and Unwraps

The JNDiff algorithm has a way to detect wraps or unwraps, but `natural-xml-diff` employs different strategies to detect them. Here we discuss a strategy focused on detecting structural wraps and unwraps. This is supplemented by a strategy to detect style wraps and unwraps, described later.

- If an element in the source document is set to be different (and thus does not exist in the target), and all its children in the target document are the parent of this element, and have an equivalent node in the source, the element is considered to be an unwrap element.
- Similarly if the element only exists in the target document, and all its children also exist in the source document and their parent is the parent of the element in the target, the element is considered to be a wrap element. At least one of the children must also exist in the source.

The asymmetry between wrap and unwrap is that unwrap requires *all* children to be equivalent, while wrap only requires a single equivalent child to consider all children a wrap. This difference exists because reconstructing the sibling order for unwrap is very difficult - it is unclear where the unwrap starts and where an element was simply inserted or removed.

Note that the wrapping algorithm is not perfect: in cases multiple candidates for the wrapping node exist, and the algorithm has to decide between them. It has a bias towards elements that are higher up in the tree, closer to the root node. It can also miss some wrapping opportunities the context of mixed text, which are later found during text cleanup.

7.5. Delete/insert Pair Merging

If an element is deleted, and then an element with the same name is inserted immediately after it in the diff document, we can consolidate this element into a single element that is not inserted or deleted, but contains attribute changes and child node updates.

Example:

```
<section>
  <p diff:delete="">Hello</p>
  <p diff:insert="" a="A">Hello world</p>
</section>
```

can be merged into:

```
<section>
  <p a="A">
    <diff:attributes>
      <a diff:insert=""/>
    </diff:attributes>
    <diff:text-delete>Hello</diff:text-delete>
    <diff:text-insert>Hello world</diff:text-insert>
  </p>
</section>
```

Before we merge delete/insert pairs, we first reorder nodes to make more merges possible.

A key insight is that a sequence of node deletes mixed with node inserts may be safely reordered, as long as deletes keep their relative ordering, as well as inserts keep their relative ordering. For example:

```
<section>
  <p diff:insert="">A</p>
  <p diff:delete="">B</p>
  <p diff:insert="">C</p>
  <p diff:delete="">D</p>
</section>
```

This diff, when applied to produce the target document, produces this:

```
<section>
  <p>A</p>
  <p>C</p>
</section>
```

which may be reordered to:

```
<section>
  <p diff:delete="">B</p>
  <p diff:insert="">A</p>
  <p diff:delete="">D</p>
  <p diff:insert="">C</p>
</section>
```

This produces the same target document - it does not matter where the deletes are placed relative to the inserts.

The reordered diff document can then be simplified by merging delete/insert pairs:

```
<section>
  <p><diff:text-delete>B</diff:text-delete><diff:text-insert>A</
diff:text-insert></p>
  <p><diff:text-delete>D</diff:text-delete><diff:text-insert>C</
diff:text-insert></p>
</section>
```

This again, when applied, produces the same target document.

The process of delete/insert reordering and merging produces more candidates for the text cleanup procedure, described later.

7.6. Text cleanup

The diffing algorithm only creates update relationships between individual text nodes. In the case of document-style XML, text nodes often are part of mixed content, where there are non-text children as well, such as in this case:

```
<p>Hello <i>world</i>!</p>
```

For mixed content we can find a more human-readable diff by considering mixed content as plain text content, and applying a text diffing algorithm. Text diffing algorithms are already geared to produce human readable diffs, and if we can use them for mixed content, we can benefit from this quality.

In general, for any content which has a text insert or delete, we may be able to produce a better diff by reapplying a text diffing algorithm.

We find all elements that have a text insert, or text delete child, and have at least an element child, or have a text delete/insert pair in them. These are candidates for text cleanup.

An example of mixed content in a diff document:

```
<p>The <diff:text-delete>pink</diff:text-delete><i diff:delete=""> big</
i><diff:text-insert> pink</diff:text-insert> elephant</p>
```

Text cleanup then reconstructs the mixed element and its descendants in the source document:

```
<p>The pink<i> big</i> elephant</p>
```

as well as the target document:

```
<p>The pink elephant</p>
```

To ensure we have plain text so that we can apply the text diff algorithm, it replaces any non-text child node of the mixed element with a special text placeholder. For the source document the text becomes:

```
The pink{placeholder} elephant
```

And for the target document, the text becomes:

```
The pink elephant
```


The placeholder consists of unicode characters from unicode supplementary private use area A, where unicode explicitly does not define characters so that they may be used for custom algorithms. The placeholder is deliberately constructed out of multiple repeating characters, as a single character is rather too likely to be moved by a text diffing algorithm.

We were inspired to implement this placeholder strategy by a study of the source code of the Python xmldiff implementation, where the XMLFormatter uses a similar strategy.

We now have two text strings (perhaps with placeholders): one for the mixed content of the source element, and one for the mixed content of the target element.

We then produce a text difference using a text diffing algorithm. We now get a text diff described by text inserts, deletes and equal parts. This algorithm could decide that the best diff is by deleting the text represented by placeholder.

```
The pink<del>{placeholder}</del> elephant.
```

We reconstruct the mixed content from this again, replacing text diff items with the regular `natural-xml-diff diff:text-insert` and `diff:text-delete`. Placeholders are treated especially - if they are found within a text insert, they are inserted as an element, and if they are found within a text delete, they are deleted as an element. If they are in the equal part, they are left in place.

So in this case we get the following:

```
The pink<i diff:delete=""> big</i> elephant.
```

which is a nicer diff than we had before, replacing an insert and a delete with a single deletion.

7.7. Text Cleanup Wrap/Unwrap detection

We previously described an algorithm that detects some element wrap and unwrap operations within the document. In mixed content, wrap and unwrap is commonly used to add or remove “style” elements, like `em` or `strong`, known as an *inline style edit*.

Let's consider the following source document:

```
<p>hello world!</p>
```

and this target document:

```
<p>hello <em>world</em>!</p>
```

We can consider the edit operation as a wrap: the `em` element *wraps* the `world` text:

```
<p>hello <em diff:wrap="">world</diff:wrap>!</p>
```

During text post-processing we can detect wrap/unwrap of elements within mixed content.

Let's consider the wrap example above. Without wrap, we can describe the diff XML as follows (indented for ease of reading):

```
<p>
  hello
  <diff:text-delete>world</diff:text-delete>
  <em diff:insert="">world</em>
  !
</p>
```

We can detect this situation: if the text content of a newly inserted element is deleted just before it, we can transform it into a wrap:

```
<p>
  hello
  <em diff:wrap="">world</em>
  !
</p>
```

The inverse situation produces this diff:

```
<p>
  hello
  <em diff:delete="">world</em>
  <diff:text-insert>world</diff:text-insert>
  !
</p>
```

In this inverse situation, an element is deleted and its text content is inserted just after it. We can transform this into an unwrap:

```
<p>
  hello
  <em diff:unwrap="">world</em>
  !
</p>
```

7.8. Cumulative Effect of Post-Processing

Combining mixed content text diffs with delete/insert merges creates a cumulative effect. This is especially noticable in edits where the target contains a translated version of the source document. In a translated document the structure is often very similar, but the text content is quite different. By merging delete and insert pairs, translated mixed content can be represented as a fine-grained text difference as opposed to more course grained element-level deletions and insertions. This makes the diff significantly more easy to comprehend.

8. Testing strategy

To produce robust software it is important to engage in extensive testing. Testing should be automated if possible to assure that the diffs are correct, and to prevent regressions, but manual testing was also important to verify that the diffs produced are human-readable.

In order to test that our implementation behaves correctly, we incrementally created many test cases where we compare two small documents, and record the expected correct result. When we worked on a new feature we would create test cases for it, including for important edge cases. During development we re-run these test cases repeatedly to verify that we have not introduced any regressions.

We also automatically verify upon each test run that for each produced diff document we can successfully reconstruct both the source document as well as the target document.

We then added property testing⁴, where we verify that the diff can be applied correctly for two randomly generated XML documents.

To further validate *natural-xml-diff* we implemented a bulk testing tool where can verify diffs for a large amount of real-world data. We applied this bulk check to the public testing corpus⁵ published to github by the authors of the diff comparison paper Cuculovic et al. (2020), as well as a collection of over 2000 documents we used in production.

To verify that a diff is not only correct but also readable, we implemented a visualization tool which creates a HTML representation of the diff document, pretty printed, with differences marked with green for insertions, red for deletions.

In order to measure the creation of minimal diffs, we created a weight measurement tool, where we created an estimate of the size (weight) of a diff by checking how much text is inside all the inserts and deletes, combined. We could then determine whether tweaks to the algorithm would produce smaller total differences given our corpus of test data.

9. Conclusion

We believe *natural-xml-diff* provides a significant practical improvement over existing XML diff algorithms for comparing structured documents. It produces high quality diffs and does so robustly over a wide selection of documents.

Many XML diff algorithms stop when they have produced a sequence of edits. A sequence of edits is less useful for the practical use case where a user interface presents the differences to a human author. It is important to produce a diff document, as that is far more human-readable. This has an additional benefit: we

⁴ https://en.wikipedia.org/wiki/Property_testing

⁵ <https://github.com/milos-cuculovic/XMLDiffAnalyzer>

found that taking the diff document as a basis provided many opportunities for further refinements of the diff that are much more difficult to implement on a sequence of edits.

Bibliography

- [1] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. *Change detection in hierarchically structured information*. 1996.
- [2] G. Cobena, T. Abdessalem, and Yassine Hinnach. *A comparative study for XML change detection*. 2002.
- [3] Angelo Di Iorio, Michele Schirinzi, Fabio Vitali, and Carlo Marchetti. *A Natural and Multi-layered Approach to Detect Changes in Tree-Based Textual Documents*. 2009.
- [4] G. Barabucci, Paolo Ciancarini, Angelo Di Iorio, and Fabio Vitali. *Towards a qualitative analysis of diff algorithms*. 2013.
- [5] Gioele Barabucci, P. Ciancarini, A. Iorio, and F. Vitali. *Measuring the domain-oriented quality of diff algorithms*. 2014.
- [6] Gioele Barabucci, Paolo Ciancarini, Angelo Di Iorio, and Fabio Vitali. *Measuring the quality of diff algorithms: a formalization*. 2016.
- [7] Paolo Ciancarini, Angelo Di Iorio, Carlo Marchetti, Michele Schirinzi, and Fabio Vitali. *Bridging the gap between tracking and detecting changes in XML*. 2016.
- [8] Milos Cuculovic, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. *Change Detection on JATS Academic Articles: An XML Diff Comparison Study*. 2020.
- [9] Milos Cuculovic, Frederic Fondement, Maxime Devanne, Jonathan Weber, and Michel Hassenforder. *Semantics to the rescue of document-based XML diff: A JATS case study*. 2022.

It's Useful After All — VIN Numbers, DITA, and iXML

Ari Nordström

Creative Words

<ari.nordstrom@gmail.com>

Abstract

Vehicle Identifier Numbers (VIN) are a way to uniquely identify not only a vehicle configuration — model year, engine, type of vehicle, etc — but also the individual. VIN numbers tend to be stamped on engine blocks and are very useful when identifying the configuration at hand when a vehicle arrives in a repair shop, helping plan a service in advance, with the right spare parts, tools, diagnostic procedures, etc.

DITA is a topic-based information standard, expressed as an XML vocabulary. It is designed to completely describe a given product and its variants in small building blocks (“topics”) that can be “profiled” to identify variant-specific information. A variant topic might use attributes to define a topic’s applicability by stating that the topic applies to product A and variant B for audience C, which by convention means that any topic describing product D for audience E would be excluded.

VIN numbers and DITA profiles, then, are two ways to describe applicability, but in very different ways.

Invisible XML (iXML) is a way to express implicit structures as grammars, allowing an instance of any given structure following the grammar to be serialised as XML. It turns out that expressing VIN semantics in an iXML grammar is exactly what we need to marry VIN numbers and DITA profiles.

Keywords: Invisible XML (iXML), Vehicle Identifier Number (VIN), DITA, Content profiling

1. Intro

1.1. VIN Numbers

Cars today are identified using so-called Vehicle Identifier Numbers (VINs for short). These are usually alphanumeric strings between 11 and 17 characters and are capable of identifying the vehicle's configuration as sold - model year, engine, body - but also the vehicle as an individual. The VIN has a variety of uses, from

identifying compatible accessories to vehicle ranges affected by a technical bulletin, and so on. For repair shops, then, the use cases are obvious - it becomes easy to plan a service, from understanding what needs to be serviced and when to acquiring spare parts and consumables in time for the service occasion.

1.2. Service Documentation

But the VIN can also help locate the service documentation applicable to the vehicle if the documentation is marked up accordingly.

Most car manufacturers today have been around for a long time, which means an ever-growing variety of available models, variants, and configurations, much of it changing from one year to the next and thus requiring a growing set of spares, tools and consumables. Similarly, the required service documentation grows in size as the applicable service procedures and parts lists change or are replaced for each passing model year.

2. DITA

Much of today's service documentation is authored in XML format, allowing for the reuse of shared procedures while also identifying model- and variant-specific tasks. This is done by "profiling" the XML - using markup to state what a specific element applies to. For example, a car model may come in petrol and diesel engine variants, meaning that the service information specific to the engines need to be marked up accordingly but much of the other documentation is shared between the variants.

The DITA specification uses a topic-based approach where each topic is meant to cover a single set of information such as a description of a component, a disassembly of that component, or a parts list applicable to that component. The idea then is that topics are authored for all components that are required to fully build a product. The topics are assembled to full "books" using DITA maps, essentially a method to assemble the topics into the required chapters and sections to provide, say, a complete illustrated parts catalogue.

2.1. Profiling in DITA

But importantly, DITA topics - and their individual elements - can be profiled using profiling attributes available for every DITA structure. The specification includes properties such as "product", "platform", and "audience" but can also be extended to, say "model year" or "engine type". We can identify specific topics as applying only to a whitespace-separated list of engine variants, model years, or markets. Or something else, including a combination of profiles. And of course, if a topic is not profiled, it applies to everything.

Technical documentation that is profiled using model and variant information also identifiable using a VIN can then be made to match a VIN that expresses a specific configuration. For example, a disassembly task might have this root element:

```
<task
  modelYear="2022"
  engineType="petrol-EU6"
  drive="LHD"
  platform="commercial"
  product="station-wagon">
  ...
</task>
```

This matches a commercial vehicle, a MY 2022, left-hand drive station wagon with a petrol engine for EU. The matching VIN number is a 17-character string that looks something like this: AB1GN1BA?NF123456.

3. Profiling and VINs

Comparing the VIN - AB1GN1BA?NF123456, for example - with a DITA profile is where it gets interesting.

3.1. The VIN As A Grammar

The VIN string is position-driven, with the various positions encoding vehicle information:

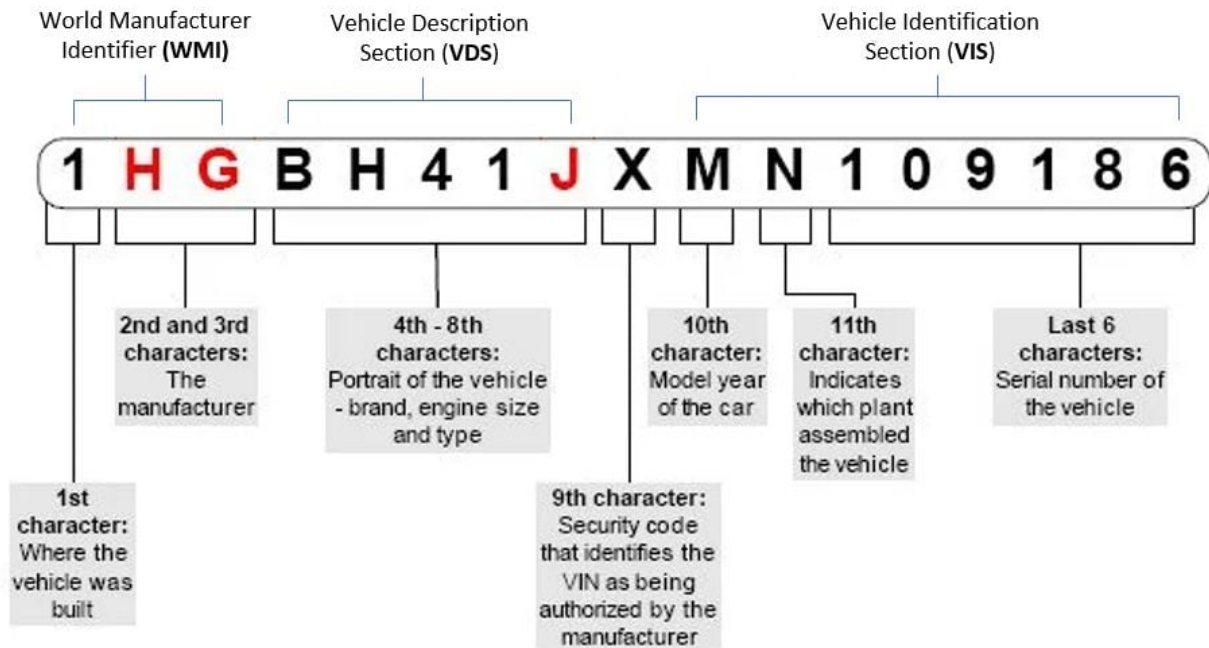


Figure 1. VIN Decoder

The allowed values are specified for each group and can be expressed as a grammar. EBNF notation, for example, has been used by manufacturers in the past, even though most of them default to an Excel spreadsheet or an ERP system.

3.2. iXML

For pointy-bracket professionals, however, there is Invisible XML¹, iXML for short:

Invisible XML is a language for describing the implicit structure of data, and a set of technologies for making that structure explicit as XML markup.

iXML allows us to write *grammars* for those various implicit structures and then use an iXML implementation to serialise an instance of that grammar in XML format.

```
vin: manufacturer-id, vehicleType, platform, engineType, drive, check-  
digit, modelYear, plant, sequence-number .  
-question-mark: "?" .  
-zero-or-many: "*" .  
manufacturer-id: "AB1" .  
commercial-vehicle: "GN" .  
private-vehicle: "GM" .  
vehicleType: commercial-vehicle; private-vehicle; question-mark,
```

¹ <https://invisiblexml.org/>


```
question-mark .
sv: "1" .
uv: "2" .
pu: "3" .
cc: "4" .
platform: sv; uv; pu; cc; question-mark .
petrol: ["A"-"C"] .
diesel: ["D"-"E"] .
avf: "F" .
engineType: petrol; diesel; avf; question-mark .
lhd: "A" .
rhd: "B" .
drive: lhd; rhd; question-mark .
check-digit: "?" .
y22: "N" .
y23: "P" .
y24: "R" .
y25: "S" .
modelYear: y22; y23; y24; y25; question-mark .
plant: "F"; question-mark .
-number: ["0"-"9"] .
-seq1: "00000", ["1"-"9"] .
-seq2: "0000", ["1"-"9"], number .
-seq3: "000", ["1"-"9"], number, number .
-seq4: "00", ["1"-"9"], number, number, number .
-seq5: "0", ["1"-"9"], number, number, number, number .
-seq6: ["1"-"9"], number, number, number, number, number .
sequence-number: seq1; seq2; seq3; seq4; seq5; seq6; zero-or-many; () .
```

Thus, the VIN string as defined for a specific manufacturer can be expressed as an iXML grammar, meaning that a string such as AB1GN1BA?NF123456 can be output in XML:

```
<vin>
  <manufacturer-id>AB1</manufacturer-id>
  <vehicleType>
    <commercial-vehicle>GN</commercial-vehicle>
  </vehicleType>
  <platform>
    <sv>1</sv>
  </platform>
  <engineType>
    <petrol>B</petrol>
  </engineType>
  <drive>
    <lhd>A</lhd>
  </drive>
```

```
<check-digit>?</check-digit>
<modelYear>
  <y22>N</y22>
</modelYear>
<plant>F</plant>
<sequence-number>123456</sequence-number>
</vin>
```

The XML is still little more than an XML version of the VIN, but already much more usable.

4. Publishing

But let's briefly go back to that technical documentation. The days of publishing the documentation on paper are long gone. Instead, the documentation is commonly published electronically in a portal application and made accessible to potentially any repair shop, as long as they have an internet connection and a browser.

4.1. Online Portal

The author has designed and helped implement one such portal application for an automotive manufacturer. The vehicle service documentation is authored in DITA and exported as-is to an XML database, eXist-db. A portal application built on top of eXist-db lists the available documentation, provides search and filtering, and allows for on-the-fly publishing of the DITA content. Importantly, the topics are listed and filtered as XML, unchanged, providing direct access to the DITA profiling information.

4.2. Listing Topics

The user interface for listing and filtering the DITA is done in XForms, with the raw file listings in XML format like so:

```
<collection
  name="/db/test/content"
  created="2023-01-03T10:19:57.112+01:00"
  owner="admin"
  group="dba"
  permissions="rwxr-xr-x"
  uri="/db/test/content">

  ...

  <file
    selected=""
```

```
    type="topic"
    uri="/db/test/content/dita-examples/01/second_portal_topic.dita"
    name="second_portal_topic.dita"
    created="2023-01-03T10:20:13.072+01:00"
    last-modified="2023-01-03T10:20:13.072+01:00"
    id="my_second_portal_topic"
    outputclass=""
    dita-content-type="content"
    product="A B"
    audience="D E"
    root-profiles="product(A B) audience(D E)"
    include="true"
    include-profiles="product(B)">
    <title>Topic 2</title>
</file>

<file
    selected=""
    type="topic"
    uri="/db/test/content/dita-examples/02/topic_3.dita"
    name="topic_3.dita"
    created="2023-01-03T10:20:12.595+01:00"
    last-modified="2023-01-03T10:20:12.595+01:00"
    id="topic_3"
    outputclass=""
    dita-content-type="content"
    product="A"
    audience="novice"
    root-profiles="product(A) audience(novice)"
    include="false"
    exclude-profiles="product"
    include-profiles="">
    <title>Topic 3</title>
</file>

...

<profiles>
    <product>
        <value>B</value>
    </product>
    <platform/>
    <audience/>
</profiles>
</collection>
```

This lists two topics (the `file` element is exactly one listed topic). If you look carefully, you'll note that the first one includes the attribute `product="A B"`. This says that the topic represented by the `file` element and referenced in `uri="/db/test/content/dita-examples/01/second_portal_topic.dita"` applies to products A and B.

The second `file` identifies a topic (`uri="/db/test/content/dita-examples/02/topic_3.dita"`) that applies to product A only, as stated in `product="A"`.

The UI that presents the file list looks like this:

The screenshot displays a web-based file browser interface. On the left side, there is a vertical panel with various search and filter controls. At the top of this panel are 'Search' and 'Reset' buttons. Below them is a 'Search String' input field. Further down are dropdown menus for 'Element', 'Book Type', and 'Document Type'. Below these are labels for 'Language: English' and 'Country: UNITED STATES'. There is a checkbox labeled 'Show Filtered'. Below that is a 'Customer VIN Number' input field. At the bottom of the filter panel is a 'Profiling Attribute' dropdown menu and a 'Show Available Profiles' button. On the right side of the interface, there is a pagination bar showing '<<' '<' 'Page 1 of 46' '>' '>>' and 'Displaying 1 to 10 of 458'. Below the pagination bar is a list of document entries. Each entry consists of a 'Browse' button followed by the document title. The titles are: 'Engine and Motor', 'Front Left Wheel Bearing Test Book', 'Owners Handbook', 'Rear Brake Pads Book', '1A. Engine (Petrol)', '7A. Front End', 'BEFORE DRIVING', 'CLIMATE CONTROL', 'DRIVER SUPPORT AND FEATURES', and 'INTRODUCTION'.

Figure 2. File Browser UI

This is simply an XForm representation. Let's have a look at the filtering capabilities on the left-hand side.

4.3. Filtering the List

The above file listing XML fragment concludes with a `profiles` structure:

```
<profiles>
  <product>
    <value>B</value>
  </product>
</platform/>
```

```
<audience/>
</profiles>
```

This is an XML fragment that is the result of a filter selected elsewhere in the XForm and then applied to the file listing. It essentially says "show only the topics applicable to product B" and if you now check the first of the two file elements, you'll spot the attribute `include="true"`. The second file has `include="false"`. These are both inserted as a result of applying the `profiles` filter.

The XForm has a standard way of entering filtering values to the `profiles` structure. The profile attributes actually in use in the database are generated as a list and presented to the user in a dropdown list. Selecting a profiling attribute from the list then provides its available values:

The image shows a web form interface. At the top, there is a section titled "Profiling Attribute" in a light blue box. Below this title is a dropdown menu with a white background and a thin border. The text "Audience" is displayed in the dropdown, and a small downward-pointing chevron is visible on the right side. Below the dropdown menu, there is another section titled "Available Profiles" in a light blue box. Under this title is a list of ten items, each consisting of an unchecked checkbox followed by a label: "Australia", "Bahrain", "EU", "Ireland", "Kenya", "NAFTA", "New_Zealand", "ROW", "UK", and "USA".

Figure 3. Profiling Values

Here, the profiling attribute `Audience` is used to list market regions.

4.4. Filtering with a VIN

Now, that filter - the `profiles` was created using an XForm dropdown list, but can easily be created by other means. For example, given a VIN string, we can serialise that string as XML using an iXML grammar and an iXML engine, and then convert the XML to the `profiles` format above and apply that to the file listing.

The screenshot shows a web form with the following elements:

- Customer VIN Number**: A text input field containing the string `GN1BA?NF123456`.
- Look up VIN**: A button with a light gray background and rounded corners.
- Profiling Attribute**: A dropdown menu that is currently empty, with a downward arrow icon on the right.
- Resolved VIN Data**: A list of attributes and their values, displayed in a light gray box:
 - `vehicleType`: `CV`
 - `Platform`: `LWB_SW`
 - `engineType`: `Petrol_EU4`
 - `drive`: `LHD`
 - `modelYear`: `2022`

Figure 4. VIN Entry Field with Resolved VIN

Above, Figure 4 shows the VIN field with a resolved VIN number (the entries are matching DITA profiling attributes and their values). The VIN number was serialised as XML and then converted to the `profiles` XML format used for filtering.

4.5. Granularity

The above filters the file listing, meaning that we limit the list of DITA topics (and maps; they are not shown above but part of the XML format, too) to only those

topics that match the filter. DITA, however, can be profiled on any element, allowing us to mark up fragments of the topic. For example, we might identify a step as only applicable to a specific engine type. If the topic was then published using that profile, the step would be included. If a conflicting profile was used, the step would be excluded.

4.6. DITAVAL Publishing

DITA defines an XML-based filtering format known as "DITAVAL", used by including the filter alongside the topic when publishing. When publishing a listed topic, the portal application converts the `profiles` XML to the DITAVAL format and provides the DITAVAL filter to the DITA publishing process alongside the topic.

Similarly, we can convert the XML-serialised VIN iXML to DITAVAL format and bypass the file listing and filtering functionality.

5. Additional Features

And finally, we can go in the "other direction" - we can use portal functionality to create a DITA filter, a set of profiles, and convert that set to a series of matching VIN strings. This is useful when wanting to know what VIN ranges match vehicles with specific variants, model years, and so on.

A somewhat contrived example is when service bulletins are issued, for example, for a service recall. Let's say that the manual gear boxes for the large diesel engine for non-EU MY 2022 vehicles all require servicing because of a problem with the oil used. We can then enter the relevant profiling filters and generate the matching VIN ranges, which can then be looked up in an ERP system so mail is only sent to customers who are actually affected.

But we can also use the profiling data to quickly look up the relevant workshop manual instructions and parts data, as well as inject the data to diagnostic software, simplifying the shop floor procedures and thus the time required to service the vehicle.

6. In Conclusion

The functionality described in this paper is live in a production setting and works beautifully. Surprisingly, though, the underlying DITA is currently limiting the usefulness of the portal because the content is not yet rich enough. Simply put, profiles are not used to the extent they can (and should) be. This is an obvious future improvement.

Technically, there's certainly more to do with the portal itself, but that's a topic for another paper.

For the iXML solution specifically, the next step is very likely going to be to expand the VIN grammar to include more data. For example, another manufacturing plant will open, requiring identification, and other vehicle models with additional capabilities will be introduced. These will all require tweaks to the processing of the serialised XML.

In closing, iXML has proved to be surprisingly robust and easy to work with, and I have other iXML projects lined up.

Bibliography

- [1] “Invisible XML Specification” [online, fetched on 4 April] <https://invisiblexml.org/1.0/>
- [2] “Invisible XML” [online, fetched on 4 April 2024] <https://invisiblexml.org/>
- [3] “jwiXML processor” [online, fetched on 4 April 2024] <https://johnlumley.github.io/jwiXML.xhtml>

Jiří Kosek (ed.)

**XML Prague 2024
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 AH Formatter.

1st edition

Prague 2024

ISBN 978-80-907787-2-6 (pdf)
ISBN 978-80-907787-3-3 (ePub)