DBL Metadata 2.3 Documentation

Mark Howe

Feb 18, 2019

CONTENTS:

CHAPTER

ONE

AN INTRODUCTION TO XML AND DBL METADATA

1.1 XML STYLE

1.1.1 HUMAN READABILITY

XML, like SGML, was designed to be readable and editable by non-technical users. It is of course possible to produce utterly opaque XML (and some companies specialize in doing this), but, within the Bible publishing world, non-programming archivists routinely skim XML for content and, when necessary, modify it using a text editor.

XML values readability over conciseness (within reason). Tag names tend to use unabbreviated words. Perhaps because of Java's domination of the XML world, composed names tend to be camelCase.

1.1.2 ATTRIBUTES VS ELEMENTS

Choices need to be made between

```
<parentElement>
    <childElement1>content</childElement1>
    <childElement2>content</childElement2>
</parentElement>
```

vs

```
<parentElement childAtt1="content" childAtt2="content"/>
```

or maybe

```
<parentElement>
    <child n="1">content</child>
    <child n="2">content</child>
</parentElement>
```

There are some fairly solid rules to follow, such as

- Do not use attributes where the value may eventually need to be structured
- Do not use attributes for anything that needs localizing (since some locales require markup, CDATA does not work with attributes, etc)
- Do not put long text strings in attributes (because it looks ugly and is hard to read and format)
- Do not put large numbers of attributes in any one element (because it looks ugly and is hard to read and format)
- Do not use attributes where order matters (because not all technologies preserve attribute order)

There are also lots of strong opinions and habits that make for lively arguments at XML conferences.

1.1.3 DOCUMENT ORDER

This is a basic XML concept, respected by all W3C XML technologies, which requires XML processors to respect the order in which elements appear. So, eg, in

```
<parent>
    <child>foo</child>
    <child>baa</child>
    <child>frob</child>
</parent>
```

it will always be possible to discover that “baa” appears after “foo” and before “frob”. In other words, XML does not have a clear distinction between lists/arrays and dictionaries/objects. Thus, regarding

```
<structure>
    <content name="book-gen" src="release/USX_2/GEN.usx" role="GEN"/>
    <content name="book-exo" src="release/USX_2/EXO.usx" role="EXO"/>
    <content name="book-lev" src="release/USX_2/LEV.usx" role="LEV"/>
    <content name="book-num" src="release/USX_2/NUM.usx" role="NUM"/>
    <content name="book-deu" src="release/USX_2/DEU.usx" role="DEU"/>
</structure>
```

XML technologies can iterate efficiently over the content elements, in the expected “document” order, or access one or more content element efficiently using any combination of attribute values. There is no need for tricks like “001_GEN” to preserve order.

(It is obviously possible to parse XML into an unordered dictionary, at which point the elements will be… unordered. Also, XML does not announce when order is considered to be significant, since significance is somewhat in the eye of the XML producer and/or consumer.)

1.2 PROCESSING XML

1.2.1 SAX

SAX is one of the oldest XML processing models that is supported by most languages. It is a streaming model that uses callbacks for every syntactic element, such as a start tag, an end tag, text…

SAX can be very fast - in a C implementation the limiting factor is often the bandwidth of the storage device. It is often used to bootstrap other models, and has become more popular recently on mobile devices because of its low memory usage. It can be an elegant way to “cherry pick” a few features the document while ignoring most of the content and/or structure.

However, many developers dislike SAX because

- there is no context for callbacks unless the application code saves it
- there is no way to look forward
- for complete processing of complex documents, the SAX callback handlers tend to become nested conditionals that are hard to debug and potentially fragile when faced with unexpected document content.

1.2.2 DOM

DOM is the original XML tree model, and is probably one of the most popular ways for applications to process XML. (Like SAX, DOM is a model rather than a standard, so implementational details vary.) Basic DOM operations allow typical tree manipulation functionality such as

- find parent, children
- remove, add or move a node
- discover the type of a node
- count nodes

The main challenge with DOM is that it does little to hide the complexity of XML documents. For example, changing the whitespace between elements in a document can drastically change the DOM representation of that document.

1.2.3 XPath

XPath may be viewed as a way to describe a route to find specific parts of a document. It consists of one or more step, separated by slashes, eg

/DBLMetadata/identification/systemId[@type='gbc']

which can be read as

- Start with the root element which is “DBLMetadata”
- Get all the child elements with a tag of “identification” (in practice there’s only one of those)
- Get all the child elements with a tag of “systemId” with an attribute called “type” that has a value of “gbc”

The result is a list of zero or more nodes which can then be processed by various technologies.

Most modern implementations of DOM include XPath 1.0, as do some DBMSs such as PostGreSQL. XPath 2.0 is more powerful and consistent, but is not well-supported, especially in the non-JVM open-source world.

1.2.4 XQuery

XQuery is a superset of XPath 2.0 which provides something like SQL functionality for working with one or more XML document.

1.2.5 XSLT

XSLT is a turing-complete, functional XML vocabulary for transforming XML documents. The basic unit of an XSLT stylesheet is the template. Parsing can be directed explicitly, much as with XQuery, or templates can be matched as part of all of the document is traversed. XSLT is particularly useful for making a limited number of changes to a complex document, most of which should be copied. (This is based on an identity transform.) XSLT tends to polarize developers, between those who consider it to be Lisp with pointy parentheses and those who consider it to be arcane and verbose.

XSLT 1.0 is available for most languages. XSLT 2.0 and 3.0 are not well-supported in the open-source world.

1.2.6 Other Processing Models

There are many language-specific processing models. Most of these models attempt to make XML simpler, or more like something else such as nested objects or a database. This tends to work well for the simple cases and not at all for the hard ones. (Scala is one example, where namespace-broken XML support is baked into the core language.)

Regular expressions are the XML processing model that no-one admits to using, but that most people end up using at some point. It can work remarkably well (I've seen conversion of OSIS to USX using cascading regexes in PHP), but it tends to be fragile and struggles with recursive structures. Regexes are one example of reinventing the XML parser, which is generally considered to be A Bad Idea (since handling all the XML edge cases is remarkably hard, and since the whole point of XML is to provide a consistent syntax so that application code doesn't need to worry about character-based parsing at all.)

1.3 XML VALIDATION

1.3.1 Validity vs Well-Formedness

This is an important distinction in XML, which is less clear with other document formats (perhaps because XML has unusually good validation support.)

Well-formed means that the document is XML, eg the tags match and are correctly constructed. Most XML processors will stop dead at the first sign that the document is not well-formed.

Valid* means that the document conforms to a particular schema.

There are therefore three levels of correctness:

- Not well-formed, ie it isn't XML at all
- Invalid, ie it's XML but not the XML we were expecting
- Valid, ie it's the XML we were expecting

1.3.2 The case for strict schema

Validation errors can be frustrating and, in some cases, hard to pinpoint. However, strict validation also has major benefits, including

- the schema provides a machine-executable, formal definition of the "shape" of a document. This is hugely preferable to a verbose description in a human language, which is inevitably ambiguous and which then needs to be implemented anyway.
- validated documents can be processed with very little defensive code, because there are no surprises on the level of missing data, unexpected data or data of an unexpected type. This leads to shorter, cleaner, more maintainable programs.

(The corollary of this is that *Bad Things Will Happen* if systems built to assume valid documents receive invalid documents and do not perform their own validation.)

1.3.3 DTDs

This was the first attempt to describe the structure of XML documents. DTDs are still used, but have generally been replaced by schemas, which describe XML using XML (or something that can be losslessly converted into XML).

1.3.4 XML Schema (XSD)

This is the original W3C schema specification. v1.0 is widely implemented. It is powerful, but has been criticised for its sprawling spec and its inability to validate some document features. v1.1 was intended to address these concerns, and can now validate everything that any other schema language can validate… but is even more sprawling. Open-source support for v1.1 is patchy.

1.3.5 RelaxNG Schema (RNG/RNC)

This was an attempt to provide an alternative to XSD. It is generally agreed that RelaxNG schema are easier to write (and to read!) than XSD, especially when the schema is expressed in the "compact" syntax that bears a passing resemblance to Bachus Naur notation. In addition, RelaxNG can validate documents that cannot be parsed statically, typically because the permitted high-level structure depends on the low-level structure. DBL Metadata is one such document, which is one reason why the current schema is written in RelaxNG.

RelaxNG 1.0 is well-supported in the open source community. One drawback, which is a corollary of dynamic parsing, is that error messages are not always very informative.

1.3.6 Schematron

In contrast to XSD and RelaxNG schema, schematron can be considered to be a way to write unit tests for XML documents. Schematron would be a very clumsy way to validate every aspect of a document. However, it is particularly useful for checking consistency between parts of a document, or for detecting duplicate values. Various versions of Schematron are available, with reasonable support for Schematron 1.6 via libxml2.

1.4 DBL METADATA CONVENTIONS

1.4.1 Dublin Core

DBL Metadata was inspired by Dublin Core, a set of standard metadata names. (DC does not provide everything needed for our domain, and in some cases the DC approach seemed overly clumsy for our purposes.)

1.4.2 Elements vs Attributes

DBL Metadata generally uses child elements for most content, reserving attributes for qualifiers and machine-readable keys, eg

```
<systemId type="ptreg">
    <id>kK6ASA9ScumywfT9v</id>
</systemId>
```

1.4.3 Order

In most places within DBL Metadata, order is unimportant. (One exception is publication structure which describes something like a contents page.) DBL tends to stick to a well-known order of top-level elements, and sometimes rearranges documents to follow this order, but this is purely to make eyeballing easier, and no technology should rely on this order.

1.4.4 Optional, non-empty elements

As of v2.0, the DBL Metadata schema has many optional elements and few elements that may be empty. So, for example, in

```
<identification>
    <name>Malayalam Bible [mal] India (BCS 2017)</name>
    <nameLocal>&#3374;&#3378;&#3375;&#3390;&#3379;&#3330; &#3372;&#3400;&#3372;&#3391;
↪&#3379;&#3405;&#8205;</nameLocal>
    <description>The Holy Bible in the Malayalam language of India (BCS 2017)</
↪description>
</identification>
```

- name and description are required
- nameLocal is optional but present
- descriptionLocal is optional and not present (but never present and empty)

```
<identification>
    <name>Malayalam Bible [mal] India (BCS 2017)</name>
    <nameLocal/>
    <description>The Holy Bible in the Malayalam language of India (BCS 2017)</
↪description>
    <descriptionLocal></descriptionLocal>
</identification>
```

is invalid because two elements are empty. (The example shows two equivalent ways of writing an empty element in XML.) There are two arguments for this approach:

- in some cases there is a semantic difference between “no value” and “a value of ‘’”
- it is not uncommon for bugs such as incorrect xpaths to result in empty elements, and it is good to detect such errors.

1.4.5 Inheritance

Some information may be specified at several levels. For example, the name of entry could be obtained from (in order)

- the nameLocal element of the selected publication
- the name element of the selected publication
- the nameLocal element in the identification section
- the name element of the identification section

In the interests of consistency, the optional elements should only be provided where the value differs from a more general value. So, eg there is no need to provide a name for a publication if the required name (or nameLocal) is identical to the name in the identification section. This is one of the less popular design decisions, but it seems important to avoid copy-and-paste fixing of empty fields, eg

```
<identification>
    <name>Malayalam Bible [mal] India (BCS 2017)</name>
    <nameLocal>Malayalam Bible [mal] India (BCS 2017)</nameLocal>
</identification>
```

where there is no easy way to decide if nameLocal is a placeholder value or the actual desired value. The correct way to represent this would be

```
<identification>
    <name>Malayalam Bible [mal] India (BCS 2017)</name>
</identification>
```

and, ideally, a nameLocal element with a localized value would be added at some point.

1.4.6 Schema

DBL Metadata is currently validated using a RelaxNG schema for structure plus a Schematron schema for constraints (mainly checking for hanging references).

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search