Using ConTEXt to process xml

Presentations in xml with the simpleslides module — a tutorial

Thomas A. Schmitz

xml has become the de-facto standard for information exchange. It is a highly versatile and flexible markup language which can be processed in many different ways. ConTEXt MkIV has sophisticated and powerful tools to process xml. This article will show how you can write presentations in xml and process them in ConTEXt.

1. Using xml

1.1 Basic Principles of xml

xml has become a standard for information exchange. It offers a good compromise: it can be read and written by humans, but it is also parsable by numerous applications. It is a fairly simple format; its first principles can be described in just a few sentences:

1. xml documents consist of content and markup; if you have already mastered a markup language (such as TEX), this should not be too difficult to grasp.
2. As a markup language, xml needs to reserve a few characters to distinguish markup from text. In xml, there are only three types of special characters: angle brackets < > delimit tags, the characters & and ; delimit character entities, and the quotation marks " and ' are used to quote attribute values.
3. xml documents are (at least in principle) self-contained: they describe their own structure, i.e., they define all allowed tags and their allowed content.
4. xml documents are characterized by a tree-structure: there is exactly one "root" element; all elements are delimited by opening and closing tags and descend from this singular root element.
5. Every opening tag has a corresponding closing tag; elements must be nested in proper order, and they must not overlap.

Unlike many other markup languages, xml defines only these *syntactical* or *structural* rules, but no *semantic* ones: there are no predefined tags (and almost no predefined entities). The advantage of this is: you can define your own tags and adapt them to the demands of your document. The drawback is: you must define your own tags and adapt them to the demands of your document. Which means: you have to think carefully about the structure of the information your document has to hold and communicate. xml encourages you to separate content and appearance because it forces you to develop your own structure.[1]

1.2 Should You Use xml?

xml presents a number of advantages, both because of its qualities and because of the many ways it is being used by many people:

1. It is extremely adaptable and can be used in many different ways, for a number of purposes. It is equally suitable for very structured data which is deeply nested and for "mixed" content which holds lots of text.
2. As said above, it encourages the separation of content and presentation, thus helping you write clean code.

[1] I say "encourages," because it is still possible to write horrible documents in xml and confuse visual appearance with structure.

3. xml can be read and processed by a huge number of applications. There is a variety of parsers and converters available for almost any programming language you can imagine, and there are extremely powerful libraries and toolchains which can help you set up a workflow based on xml.
4. xml provides a number of output options: with the proper toolchain, it can produce printable output (e.g., pdf), or it can be displayed on the web.
5. xml provides a good compromise between human readability and density of information.
6. One could argue that its somewhat verbose syntax is an advantage: the explicit closing of tags makes it easier to parse and read than TEX syntax with its closing braces which will not tell you exactly which group they close.

These are powerful arguments for asking yourself whether you should use xml as the input format for your files. However, xml is not a magic bullet, and it is not per se "better" than TEX input; there are also a number of drawbacks which you should not neglect when you use ConTEXt:

1. You have to learn the syntax of another language;
2. xml adds another layer between your source and your output; you will have to map xml elements to appropriate ConTEXt commands;
3. this makes writing macros somewhat cumbersome: you always have to think twice about appropriate syntax and rules, once to express your intention in XML, and then to map your xml structure to ConTEXt.
4. This can become especially annoying when something does not work: it makes debugging more difficult because the translation from xml to TEX creates an additional source of problems.

Overall, I would suggest that you make an informed decision: if you mostly write one-off documents which usually differ in structure, style and format, and if all you want from your documents is printed or pdf output, xml is most probably not for you. xml may be useful if

- your documents have a predictable, repetitive structure,
- this structure is typically translated into a repetitive style,
- you want several output formats from your source,
- there is a chance that you may wish to reuse the content of your documents in other ways and with other applications.

2. Using xml with ConTEXt

2.1 The Editor

Whenever people hear about xml, one of the first questions to spring up is "what editor should I use?" There is no universal answer to this, of course; it depends on a number of factors: your operating system, the type of xml document you are authoring (do they primarily consist of mixed content and text, or are they typically some sort of data base?), the way you typically work with documents: do you prefer to have the xml syntax hidden while you edit your documents, or would you prefer to have tags and attributes in plain sight?
There are a number of specialized tools available that will facilitate editing xml; one of the most well-known applications is Oxygen, a commercial tool which runs on most platforms.[2] Most general-purpose editors have an xml mode which will at least provide syntax highlighting. The solution I prefer is pretty basic: Emacs is my editor of choice, and it has a nice xml editing

[2] See http://www.oxygenxml.com.

mode, nxml-mode.[3] It offers syntax highlighting, it has some basic functionality such as indenting, it validates xml on the fly, and if you have a proper schema connected with your xml document, it will validate your input against this schema and offer completion. But it is difficult to make a general recommendation; you should simply try out different tools and see what you feel comfortable with.

2.2 xml in ConTEXt

Processing xml in ConTEXt has been possible for some time now. However, with the advent of ConTEXt MkIV, based on LuaTEX, xml support has improved dramatically. The main difference is: xml processing in MkII was by and large a streaming parser, which means that elements were processed exactly in the order in which ConTEXt read them. You could of course reuse the content of elements, but this was a pretty difficult task which involved some extra steps. In MkIV, the content of an xml file is parsed with Lua, it is translated into a Lua table, and retained in memory. This means that you have access to every element of your xml file at every moment of typesetting. This makes it much easier to reuse, select, manipulate, filter, or test your input. It is now possible to write macros which will process the *nth* element of type <x> or an element of type <x> only if it contains the string z, or to compare the content of elements to predefined values and then do something special with it, depending on the output of the tests. All of this would have been extremely cumbersome or impossible in MkII; in MkIV, you don't have to be a programming genius to do that. Hence, in some areas, MkII may still be useful, but for processing xml, it should really be considered obsolete; MkIV is now the standard.
In order to process xml with ConTEXt, you need your xml file (of course), and a ConTEXt style sheet, which needs to be in a place where ConTEXt can find it (the easiest method is to put it in the same directory as the xml document). It contains two sorts of instructions:

- setups for every xml element that you want to process and
- setups of the layout and look of your document, exactly like any "normal" document preamble would contain.

When we have a file `document.xml` and a style file `style.tex`, you process your files with the command `context --environment=style document.xml` and you will receive the output in a file `document.pdf`.

2.3 Presentations in xml

We will now learn how to process xml in ConTEXt by looking at a real-world example. I will describe my own workflow, which I have been using now for two years in my university. I give a 90-minute lecture course every week of our teaching period. I write the text of the lectures in xml, and I have now decided that my source xml file should contain both the text that will appear on the slides and my lecture notes. This has the advantage that I can reuse these texts with other applications (e.g., creating a web page for the class) and that it is easy to produce the slides, handouts for the students, and the manuscript for myself from the same source, by using different style sheets.
For processing the xml file as a presentation, I use the `simpleslides` module, which Aditya Mahajan and myself have been developing during the last three years. This makes it easy to obtain slides with a visually appealing design, without having to worry about setups and layout; all the heavy lifting is done by the module.

2.4 Thinking about the Structure

As we have seen, we are free to define the elements of our xml file ourselves. What, then, would be a good structure for our file? It will hold the presentations for an entire semester. So we have a root element which we call, unimaginatively, "document."

[3] See http://www.thaiopensource.com/nxml-mode.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE presentation>
<document>
</document>
```

This root element will have child elements "presentation" for every lecture. In order to give every presentation a unique identifier, we add an attribute "tag" with the date that this particular presentation will be given. So the structure looks like this:

```
<document>
  <presentation tag="12_09_21">
  </presentation>
  <presentation tag="12_09_28">
  </presentation>
</document>
```

Every presentation has some meta-information such as a title and the date when it will be delivered; this will appear on a title slide. And it has content: the single slides which make up the presentation. So the structure of our first presentation will be:

```
<document>
  <presentation tag="12_09_21">
    <title>Example of a presentation
                        in XML</title>
    <author>A. U. Thor</author>
    <date>Sep. 21, 2012</date>
    <content>
      <slide></slide>
      <slide></slide>
    </content>
  </presentation>
</document>
```

Every single slide has three parts: its title, the content which will appear on the slide itself, and a comment which will not be visible on the slide, but will be printed in your manuscript for the lecture. The content of the slides is of course anything you may want to put there. For the purpose of this tutorial, we will look at a few standard elements which occur frequently in presentations: a slide with normal text, a slide with an itemization, a slide with an image, and finally a slide with a table. We will later look at these slides one by one so as not to make the xml example too long to read. But first, we will delve into the principles of writing a style sheet for processing these xml elements.

2.5 Writing your Style File

Our file `style.tex` begins with these lines:

```
\startxmlsetups xml:presentationsetups
  \xmlsetsetup{#1}{*}{-}
  \xmlsetsetup{#1}{document|
                   presentation|
                   title|
                   author|
                   date|
                   content|
                   slide|
                   slidetitle|
                   slidecontent}
              {xml:*}
\stopxmlsetups
\xmlregistersetup
  {xml:presentationsetups}
```

We define and "register" a setup for our xml processing instructions; we call it "presentationsetups." As you remember, our file contains both the text for the slides and our notes, but for the presentation itself, we only want to process the elements containing the content of the slides. In order to be able to do this, we need to define exactly which elements should be processed. For this, we first tell ConTEXt that every element should be dropped; that is what the instruction `\xmlsetsetup{#1}{*}{-}` does. After that, we list all the xml elements that *will* in fact be processed. We will later have to add every new element to this list. As you can see, we add all our elements to this list except the

<slidecomment> element, which should not be typeset for our slides.
The instruction for the root element <document> is simple: we just tell ConTEXt to "flush" its content, i.e., to pass it on to the typesetting engine for further processing:

```
\startxmlsetups xml:document
  \xmlflush{#1}
\stopxmlsetups
```

This, then, is the first and most important command that you have to learn when processing xml: \xmlflush{#1} means "take the content of the current xml element and process it." Processing here means: if it contains other xml elements, those will again have to be defined; if it contains simple text, this will be typeset.
What about the individual presentations? In our example, we have our presentations organized by date, and we want to be able to typeset and show one single presentation for every lecture. How can we achieve this? Here is a suggestion: we take the "tag" attribute and use it as the name of a ConTEXt mode:

```
\startxmlsetups xml:presentation
  \startmode[\xmlatt{#1}{tag}]
  \setupTitle
    [title={\xmltext{#1}{title}},
     author={\xmltext{#1}{author}},
     date={\xmltext{#1}{date}}]
  \placeTitle
  \xmltext{#1}{content}
  \page
  \stopmode
\stopxmlsetups
```

This is the code which will process one single presentation. We first extract the value of our "tag" attribute (this is what \xmlatt{#1}{tag} expands to) and use it to define a ConTEXt mode; so for our presentation with the attribute "12_09_21", this will define a mode 12_09_21. When we tell ConTEXt to use this mode, i.e., when we run it like this on the command line:

```
context --environment=style
          --mode=11_09_21 document.xml
```

it will only process the presentation with this particular tag.
Then, we look at the content of the child elements of our <presentation> element: \xmltext{#1}{author} takes the content of the <title> element, which is then passed on to the command \setupTitle from the simpleslides module. The title, author, and date elements are processed this way; the command \placetitle creates our title slide. Finally, we simply pass the <content> element on to the typesetting engine for further processing.
This is the general structure for our presentation. Let us now look at the individual slides. The first slide will only have a title and some text. This is what the xml would look like:

```
<slide>
<slidetitle>Our First Slide</slidetitle>
<slidecontent>
  ConTeXt is a document markup language
  and document preparation system
  based on the TeX typesetting system.
  It was designed with the same general-
  purpose aims as LaTeX of providing
  an easy to use interface to the
  high quality typesetting engine
  provided by TeX.
  However, while LaTeX insulates the
  writer from typographical details,
  ConTeXt takes a complementary
  approach by providing structured
  interfaces for handling typography,
  including extensive support for
  colors, backgrounds, hyperlinks,
  presentations, figure-text
  integration, and conditional
  compilation.
</slidecontent>
```

```
<slidecomment>
  Here are the notes for this slide.
  They will not be typeset on the
  slide itself, just in our manuscript
  for the lecture.
</slidecomment>
</slide>
```

This is not difficult: we just have to write rules which will retrieve the content of the "slidetitle" and "slidecontent" element and pass it to ConTEXt:

```
\startxmlsetups xml:slide
  \xmldoiftext{#1}{/slidetitle}
    {\SlideTitle
            {\xmltext{#1}{slidetitle}}}
    \start
      \xmltext{#1}{slidecontent}
      \par
    \stop
\stopxmlsetups
```

In the first line of this setup command, we test whether the "slidetitle" element has any content (not all slides do have titles, after all); if it does, this content is given to the `\SlideTitle` command of the `simpleslides` module. As you can see, the content of the slide itself is wrapped into a `\start ... \stop` pair so that all font and size switches within a slide remain local. The slidecontent itself is then passed on to ConTEXt. In the place of our simple text slide, this is enough; ConTEXt will simply typeset the text.
Next up: a slide with a list of numbered items. Here is the xml:

```
<slide>
  <slidetitle>
    Our Second Slide
  </slidetitle>
  <slidecontent>
    <numberedlist>
      <item>Our first item</item>
      <item>Our second item</item>
      <item>Our third item</item>
      <item>Our fourth item</item>
    </numberedlist>
  </slidecontent>
  <slidecomment>
    More notes.
  </slidecomment>
</slide>
```

In this case, then, we will have to define setups for two new xml elements, "numberedlist" and "item" (and remember that you have to add these names to the list of elements at the top of the `style` file). This is pretty straightforward; we just have to pass them on to a ConTEXt itemization:

```
\startxmlsetups xml:numberedlist
  \startitemize[n]
    \xmlflush{#1}
  \stopitemize
\xmlsetups

\startxmlsetups xml:item
  \startitem
    \ignorespaces\xmlflush{#1}
  \stopitem
\stopxmlsetups
```

We first define that ConTEXt should "flush" the content of the element `<numberedlist>` inside a `\startitemze[n]` environment and then each single `<item>` inside a `\startitem` environment. Our next slide will have an image. Again, the `simpleslides` module already has code for including images and presenting them in an appealing manner, so we simply have to think of a way of expressing this in xml and passing the information on to ConTEXt. Here is what this might look like:

```
<slide>
  <slidecontent>
    <includeimage type="vertical"
                  resource="cow"
                  height="0.4">
      Caption (title) for your image
    </includeimage>
  </slidecontent>

  <slidecomment>
    More notes.
  </slidecomment>
</slide>
```

Here is the setup for this xml structure:

```
\startxmlsetups xml:includeimage
  \IncludePicture
    [\xmlatt{#1}{type}]
    [\xmlatt{#1}{resource}]
    [height=
        \xmlatt{#1}{height}\textheight]
    {\xmlflush{#1}}
\stopxmlsetups
```

As you may remember, `\xmlatt{#1}{type}` means "the value of the attribute type of the current xml element." An xml element can have several attributes, and we can retrieve their values with this ConTEXt command. And don't forget to add the new element `<includeimage>` to your list at the beginning of the style file!
Finally, we want a table on one of our slides. Here, we make use of the new "Extreme Tables" mechanism which Hans Hagen introduced in October 2011. To show one way of mapping tables from xml to TEX, we add some cells which use more than one row or column. Here is how we express this in xml:

```
<slide>
  <slidetitle>A table</slidetitle>
  <slidecontent>
    <table>
      <tablerow>
        <tablecell>Have</tablecell>
        <tablecell>you</tablecell>
        <tablecell>ever</tablecell>
        <tablecell>seen</tablecell>
       </tablerow>
       <tablerow>
       <tablecell>what</tablecell>
       <tablecell nx="2" ny="2">
         &CONTEXT;
       </tablecell>
       <tablecell>can</tablecell>
      </tablerow>
      <tablerow>
        <tablecell>do</tablecell>
        <tablecell>for</tablecell>
      </tablerow>
      <tablerow>
        <tablecell>all</tablecell>
        <tablecell>of</tablecell>
        <tablecell>your</tablecell>
        <tablecell>
          documents?
        </tablecell>
      </tablerow>
    </table>
  </slidecontent>
  <slidecomment>
    Even more notes.
  </slidecomment>
</slide>
```

The mapping to ConTEXt commands is pretty simple in this case:

```
\startxmlsetups xml:table
  \placefigure[here,force,none]{}
   {\startembeddedxtable
    \xmlflush{#1}
  \stopembeddedxtable}
```

```
\stopxmlsetups
\startxmlsetups xml:tablerow
       \startxrow
         \xmlflush{#1}
       \stopxrow
\stopxmlsetups
\startxmlsetups xml:tablecell
  \startxcell[nx=\xmlattdef{#1}{nx}{1},
              ny=\xmlattdef{#1}{ny}{1},
              align=middle,
              top=\vss,bottom=\vss]
    \xmlflush{#1}
  \stopxcell
\stopxmlsetups
```

Here, you see another way to process xml attributes: `\xmlattdef{#1}{nx}{1}` means: "the value of the nx attribute of the current element; if there is no such attribute, take the default value '1.'" Moreover, we have used an xml entity: we want the logo ConTEXt to be typeset properly, so we have expressed it in xml as `&CONTEXT;`. Now we need to add a definition to our style file

```
\xmltexentity{CONTEXT}{\CONTEXT}
```

We are almost there! Our style now needs to define the look of our document. In this case, this is fairly easy because the `simpleslides` module does all the heavy lifting for us, so all we need is this:

```
\usemodule[simpleslides]
          [style=BigNumber,
           font=Helvetica,size=17pt]
```

When we process our xml file with this command:

```
context --environment=style
          --mode=11_09_21 document.xml
```

we get output which looks like this:

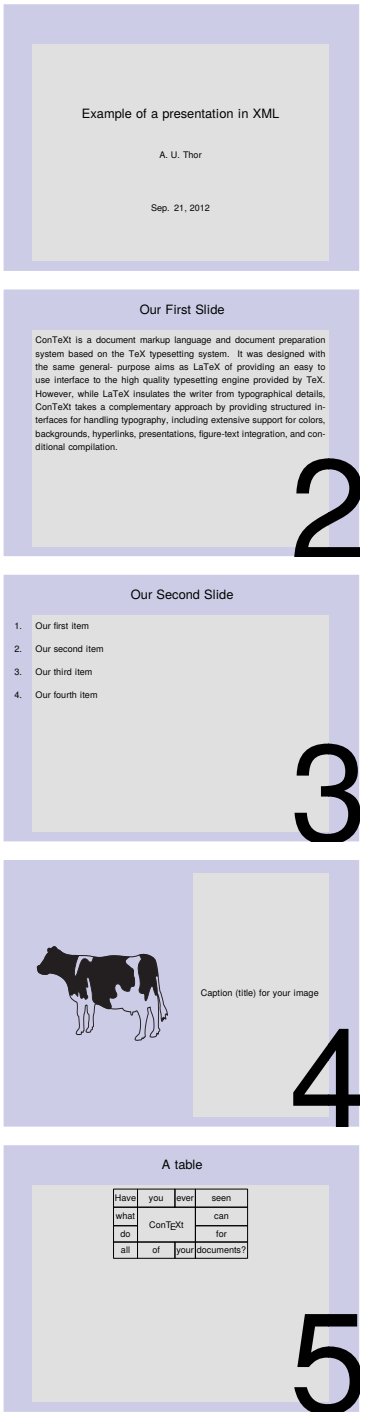

Figure 1: Output of our XML presentation

2.6 Producing a Manuscript

This, then, is the setup to produce a pdf presentation from our xml file. We will now look at ways of producing different output from the same file, using different styles. Our first exercise will be to typeset a manuscript. Here, the result we want to obtain is the reverse of what we had in the presentation: this time, we only want the content of the <slidecomment> elements typeset and all the rest dropped. As we don't have any special commands in the comment section (yet), all we have to do is make sure that these elements are typeset. We will insert an eye-catching counter for our slides so it is easy to know exactly where in your presentation you are at any given moment. This is fairly easy. Since we have understood the basic principles of processing xml we can simply look at this second style sheet:

```
\startxmlsetups xml:manuscriptsetups
  \xmlsetsetup{#1}{*}{-}
  \xmlsetsetup{#1}{document|
                   presentation|
                   content|
                   slide|
                   slidecomment}{xml:*}
\stopxmlsetups

\xmlregistersetup{xml:manuscriptsetups}

\startxmlsetups xml:document
  \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:content
  \xmlflush{#1}
\stopxmlsetups

\startxmlsetups xml:presentation
  \startsubsection
     [title={\xmltext{#1}{date}:
             \hfill
             \xmltext{#1}{title}},
      bookmark={\xmltext{#1}{date}:%
                \xmltext{#1}{title}}]
  \xmlflush{#1}
  \stopsubsection
\stopxmlsetups

\startxmlsetups xml:slide
  \NewSlide \xmlflush{#1}
\stopxmlsetups
\startxmlsetups xml:slidecomment
  \xmlflush{#1}\par
\stopxmlsetups

\definecounter
  [SlideNumber]
  [way=bytext,prefix=no]

\setuplayout
  [marking=off,
   width=fit,
   height=fit,
   header=0.6cm,
   footer=0cm]

\setuphead
  [section]
  [style=normal,
   number=yes,
   after={\resetcounter[SlideNumber]},
   expansion=yes,
   page=yes]

\setuppapersize
  [A6,landscape][A6,landscape]

\define\NewSlide%
  {\incrementcounter[SlideNumber]%
   \color[red]%
     {[{\rawcounter[SlideNumber]}]}}

\setupinteraction
  [state=start,
   title={XML Presentations},
   author={A. U. Thor}]
```

```
\placebookmarks[section][all]

\setupuserpagenumber
  [state=start,way=bysection]

\setupheadertexts
  [{\getmarking[sectionnumber]}
                        \pagenumber ]
```

Most of what you see here should be pretty obvious by now: we "flush" the parts we want typeset. In this case, we want one pdf for the entire lecture course, with every single presentation as a ConTEXt section. These sections will begin on a new page and have the `title` element as their respective title. We also put bookmarks into our pdf so it will be easier to find the single presentations. We have defined a counter `\SlideNumber` which is incremented and displayed in red print for every new slide. The header displays the number of the presentation in our lecture course and the page number within this presentation. The rest of the code contains some more setups for the visual appearance of the manuscript which is typeset on a landscape A6 paper, a format which is well adapted for printing on index cards or for reading from a tablet device.

2.7 Producing a Handout

I used to put the slides on the web for downloading in exactly the form that I showed them in the classroom, of which figure 1 is an example. But the students rightly complained that this format was not suitable for printing. Hence, I defined a layout which would typeset the contents of the slides without any colored background and arranged four slides in a vertical row on the left side of an A4 sheet, leaving the right-hand side blank for notes so students could print out the handouts, bring them to class, and take their notes next to the slides they were seeing.
Most setups for the elements of the single slides are exactly like their counterparts for the presentation setup. Since we're no longer using the `simpleslides` module in these examples, I had to copy a few definitions from the module (e.g., the setups for placing images), but I will not bore you with these details. The most important part is the arranging of the slides, which is achieved with this code:

```
\setuppapersize[A7,landscape][A4]

\setuppaper
  [topspace=3mm,
   backspace=1.5mm,
   bottomspace=0mm,
   dx=0mm,
   dy=0mm,
   nx=1,
   ny=4]

\setuparranging[XY]
```

This will give the desired arrangements, ideal for printing.

3. Conclusion

Setting up a workflow to produce presentations from xml takes some time, and it took me several attempts to reach a form which I hope will continue toserve me well in the future. Now that everything is in place, the advantages of the xml format are evident: it is convenient to have both the slides and the lecture notes in one file; this makes reusing the material easier. The material is now readily available for other output formats; it would not be difficult to put the slides on the web via some xslt transformation. And since there is a clear separation of content and appearance, I hope this format should be reusable even if the underlying mechanisms change (e.g., Aditya has been secretly working on a succesor to `simpleslides`).