Software & Tools

The Paper Path: XML to paper using TEXML

Brian E. Travis

Abstract

So, you've gotten your valuable information assets described in terms of XML schemas (either DTD or some other form of schema), and you've taken the painful step of converting your information from word processors and typesetting files into XML that adheres to your new schema. Now what?

This article describes a path that has many pieces that must fit together exactly. That's the down-side. The up-side, however, is a very powerful XML-to-paper path that will not cost you a penny, and runs on any platform that runs Java.

— — ∗ — —

So, you've gotten your valuable information assets described in terms of XML schemas (either DTD or some other form of schema), and you've taken the painful step of converting your information from word processors and typesetting files into XML that adheres to your new schema. Now what?

This article describes a path that has many pieces that must fit together exactly. That's the down-side. The up-side, however, is a very powerful XML-to-paper path that will not cost you a penny, and runs on any platform that runs Java.

As part of the information analysis phase of your project, you probably went through the task of looking for information hidden in your existing documents. This meant taking text that was in italic, for example, and tagging it as an emphasized phrase, foreign term, bibliographic reference, or legal citation. Your new XML-tagged content is rich in self-describing information objects, but contains no information about how to format those objects. For example, indicating to a formatting engine that a string of text is to be rendered in an italic font causes something to happen. That is, the formatting engine changes the font characteristics for the duration of the italicized phrase. However, tell a formatting engine to render something in "foreign phrase", and it will probably have a problem.

That is because there is an important piece missing. Sure, we know it is a phrase expressed in a language other than the default language being used in the current document, but we don't have information about how it is to be rendered on paper. We need some kind of mapping to translate from "foreign phrase" to "italic". What's more, we may even need to do more processing on the object. For example, we might want to collect all foreign phrases in the document, along with their translations in an appendix. We have even more opportunities for further processing when rendering the information in electronic form. For example, when creating an HTML rendering of the document, we may want our foreign phrase to be underlined and linked to a pop-up window with the translation. Or, we may want to build a system that causes a speech synthesizer to speak the word in its native tongue.

This is only one example of processing a single element in many different ways based on the desires for information delivery.

So, how do we make the leap from “foreign phrase” to an italicized string of text with the translation collected in an appendix? Add to that the other several dozen elements that need to be translated into some kind of deliverable.

Pagination Nation

The first thing to consider when putting information on paper is the page itself. This process is called “pagination”. Everything in your content must be rendered somehow on a two-dimensional frame bounded by the physics of the real world. A page contains a body area where the rendered text sits, plus a margin area where navigational information goes.

In the body of a page, there are blocks of text that have been rendered using centuries-old techniques. First, each line is set with text until it reaches an acceptable length, at which point the line is ended and the next word starts a new line. If the word doesn’t reach the acceptable range, and the next word causes the right margin to be overrun, the last word must be broken at a place that follows the rules of hyphenation.

Another task of the pagination program is making sure each line ends at the same place. This is called “justification”, and is preferred by some designers to make the page look symmetrical. This requires the typesetter to calculate the space left over at the end of a line, divide it by the number of spaces between words of that line, and add this new increment to each space.

These two processes are lumped together in the typesetting lingo as “hyphenation and justification”, or “H&J”. H&J is a basic requisite of any typesetting program, and all programs, from free to $100K+ systems provide this service. Higher-end typesetters will also do a sophisticated analysis of the page after it is set in memory. One thing such typesetters look for is spaces between words that line up from one line to the next. Putting too many of these spaces in a row vertically causes an effect known as “rivers”, that might be distracting to the reader. Another check these high-end devices perform is hyphenation analysis. Some page designers don’t like to see more than two hyphenated lines in a row. In order to avoid this, the typesetter may need to reset the page many different times, using different word- and character-spacing values in order to eliminate multiple hyphens. Speaking of hyphens, some typesetters check to make sure there is no hyphenation between pages or columns. This is something my third-grade grammar teacher, Miss Blankenship, would not tolerate.

Once the body area of a page is set, certain navigational features are placed on the page. The most common is the page number. The typesetter must keep track of the page number, and provide an incremental indicator on each page. This is more difficult than it looks, once you consider the many different ways pages can be numbered.

Running headers and running footers provide further navigational aids to the reader, and give the document designer a place to show off. Running headers usually provide some kind of indication of the title of the chapter, and maybe even the name of the document. Another type of running head is called a “dictionary header”. This is a header that changes depending on the contents of the page itself. The dictionary header is used in dictionaries, encyclopedias, and telephone books where the left header indicates the first entry on the page and the right header indicates the last entry on the page. This processing can be time-consuming, but leads to a better product to which consumers are accustomed.

All of these formatting conventions have been developed over a thousand years of page creation. We all grew up learning to navigate our way around a page, so these conventions should be followed to provide your users with a familiar interface.

DSSSL and XSL

First, a little background. All of the techniques described above are oriented toward the delivery of information on paper. However, your XML documents are probably tagged according to each element’s meaning, not whether it should be italicized or placed in a dictionary header. We need to map the structure to a page layout. This requires a lot of decisions, which can be expressed in the language of the typesetting system.

Each typesetting system, however, has a different way of expressing such information. An ISO standard called DSSSL (Document Style Semantic Specification Language) was designed to normalize all of the rich formatting capabilities into a single syntax. The goal was to create a non-typesetter-specific formatting language that could be translated to any typesetter’s code. The benefits of this approach are twofold: first, a designer needn’t know the syntax specifics of a particular typesetting system to create pages using that typesetter. Second, creating stylesheets in a non-vendor-specific syntax allows a company to change their typesetters at will, without the costly process of converting from one syntax to another.

DSSSL took ten years to create, and was finally ratified as an international standard just about the time that XML was gaining momentum. DSSSL is based on SGML, and was never really implemented because software vendors were looking at XML as a replacement for SGML. The need for a vendor-neutral typesetting system was still there, however, so the DSSSL folks started work on a specification called XSL, the Extensible Stylesheet Language. XSL was intended to achieve the same goals as DSSSL, except it was expressed in XML syntax.

XSL uses a transformation process, which converts your XML document into another XML document expressed as a set of "formatting objects". These formatting objects have element names like "block" and "character", with attribute values like "bold", and "green". This formatting object XML document is exposed to the second XSL step, which translates the formatting object document into the codes of a particular typesetter.

This model assures that a designer need only be concerned with a single way of formatting a page (the formatting object model), and leaves the intricacies of the typesetter to each typesetter vendor. The biggest problem with XSL is that it is very difficult to express the full range of formatting options in a single, generic specification. The DSSSL people took ten years to do it.

XSL is still being developed, but has spawned another specification called "XSLT" (see Bob DuCharme's "XML Linking and Styling: Standards Status Report", *<TAG>* August, 1999). XSLT is only the first half of the XSL process. XSLT is designed to provide a generic tree-to-tree transformation of one document structure to another. Originally, as I mentioned, this resulting structure was the one defined by the formatting object schema. However, XSLT has been generalized to a point where it can create any arbitrary XML structure, and even non-XML structures. XSLT is truly a generic XML processing language.

TeX and LaTeX

Now that we have a way of getting our XML documents into some other form, how do we produce pages? Simple, use a pagination program. There are many different pagination systems available for virtually any price you want to pay. In the 1980s, a computer science professor named Donald Knuth at Stanford University was working on a set of textbooks to describe The Art of Computer Programming. Knuth needed a more sophisticated way of paginating his document than the current state-of-the-art paginators were able to do. At that time, scholarly works were being formatted using a rudimentary typesetting system called "troff". Knuth was a student of the art of typesetting, and felt that a computer could be taught most of the mechanics of expressing that art. So, he embarked on an effort called TeX, which he describes in his book *The TeXbook*.

Knuth used TeX to typeset his seminal multi-volume set of computer science textbooks, which has become the bible of computer science academia. Knuth also made the source of his new typesetting language available to the world to use and improve upon long before the concept of "open source" grabbed headlines. It didn't take long before TeX became the syntax used to create scholarly and technical journals. The TeX mathematical syntax is very powerful, and is used to create technical papers with an accuracy unrivaled by any commercial typesetting system.

TeX is a very powerful typesetting language, with what I think is the best H&J logic available anywhere at any price. TeX produces beautiful pages and, because of the many add-ons that people have created over the years, has great flexibility.

Of course, you need to pay for this power and flexibility. The cost is learning the terse syntax and understanding all of the different settings and the way pages are created.

One of the most successful add-ons to TeX is a package called LaTeX, which provides an easier-to-use interface to the powerful TeX language. While TeX has commands for setting the font style to bold and left-justifying paragraphs, LaTeX has directives that allow you to indicate the title of a document, or the body of a section. For example, LaTeX uses the `\section` and `\subsection` commands to indicate where such breaks are made. What happens, however, if you call your structural objects "chapter", or "lesson plan" or "appendix"? And what if your structure doesn't map directly to those hard-coded into the LaTeX spec? If this is the case, you need an intermediary translation to indicate the complex structure-to-structure translation.

Alphabet Soup

IBM, through an effort called AlphaWorks, is working on a number of projects to support XML and related standards. The AlphaWorks site makes available an XML parser, written in Java. On top of this, they provide, under their Lotus brand name, a product called LotusXSL. LotusXSL is an XSL processor that uses the transformation part of XSL (XSLT) to transform one type of XML to another.

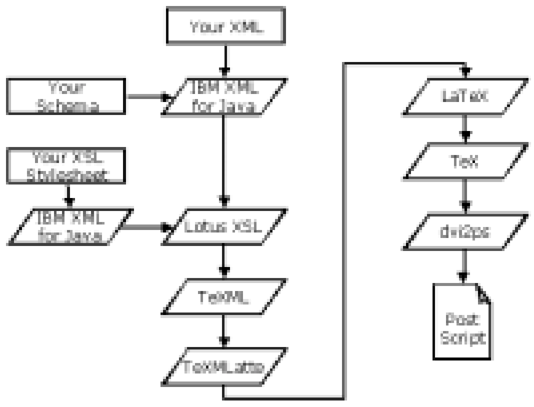


Figure 1: Paper chase: XML to Paper using XML, XSL and TEX

Lotus engineers have created an XML schema that is designed to express the structure of a paper document in terms of the LATEX markup language. They have also created a processor to read this XML document and transform it to LATEX codes.

The path I am describing has many pieces that must fit together exactly. That's the down-side. The up-side, however, is a very powerful XML-to-paper path that will not cost you a penny, and runs on any platform that runs Java.

The map is shown in Figure 1.

The XML document describes our information in terms of the information itself, not in terms of some eventual delivery platform. Therefore, we need to run a transformation step that translates this into some kind of format that can be interpreted as a two-dimensional, static form to be rendered to paper.

Translating directly to TEX is difficult because of the complexity of the TEX typesetting language. One of the problems is that TEX uses the "<" and "&" characters to mean certain things. These characters are sacred to the XML parser, and shouldn't be used because they might end up being interpreted as markup characters.

One solution, then, is to transform our XML document, which describes our information structure, into another XML document that describes the desired formatting characteristics of the information contained therein.

The TEXML system does this by defining an XML document that provides the full capabilities of the TEX typesetting language. Actually, the TEXML system creates documents that can be expressed in LATEX, the TEX add-on that is used to provide a high-level interface to TEX.

Structure to Pages

There are many steps involved. In this article, we will follow a fairly simple document through the steps required to go from XML to paper.

1. **Create a directory to contain all programs and data.**
   Select some directory anywhere on your machine. I will describe all processes in relation to that root. You should be able to move the contents of the directory anywhere using the relative paths explained here.
2. **Load the IBM4J parser.**
   Go to `http://www.alphaworks.ibm.com/tech/XML4J` and download IBM's parser written in Java. The examples in this article use version 2.0.15 of IBM's parser. Extract the files to the `xml4j_2_0_15` directory under the directory set above. If you are using a different version, you might need to change some environment variables and batch-file commands.
3. **Load the Lotus XSL processor.**
   Go to `http://www.alphaworks.ibm.com/tech/LotusXSL` and download the Lotus XSL processor. The examples in this article use version 0.18.5 of the XSL processor. Extract the files to the `lotusxsl_0_18_5` directory under the directory set above. If you are using a different version, you might need to change some environment variables and batch-file commands.
4. **Load the TEXML Processor.**
   Go to `http://www.alphaworks.ibm.com/tech/texml` and download the IBM TEXML processor. The examples in this article use version 1.4 of the TEXML processor. Extract the files to the `TeXML_V1R4` directory under the directory set above. If you are using a different version, you might need to change some environment variables and batch-file commands.
5. **Load a TEX Processor.**
   There are many excellent TEX processors available for free or by commercial license. Check the TEX Users Group at `http://www.tug.org` for a list of pointers to sites with TEX implementations. I used the MiKTEX implementation for this article, which can be found at `http://www.miktex.de/`. Most TEX implementations come with the TEX processor, which will also read LATEX files. A TEX processor creates a device-independent (DVI) output file. Most implementations also come with a program for converting the DVI to printable forms, like PostScript. The batch files in this article assume that tex, latex, and dvi2ps are in the system path, and that

there is a program registered to view DVI files. You might need to change some environment variables and batch-file commands.

6. **Select an XML document as a source.** Create or find an XML document that is suitable to transformation. XSL provides a powerful engine to transform from any XML structure to any other. For this example, we picked a straightforward example. Ours is shown in Figure 2.

```
<?xml version="1.0"?>
<bill-o-rights>
 <section>
  <title>Amendment I (1791)</title>
  <para>Congress <emph>shall make no law</emph>
  respecting an establishment of religion, or
  prohibiting the free exercise thereof; or
  abridging the freedom of speech, or of the
  press; or the right of the people peaceably
  to assemble, and to petition the government
  for a redress of grievances.</para>
 </section>
 <section>
  <title>Amendment II (1791)</title>
  <para>A well regulated militia, <emph>being

 ... {rest of bill of rights here} ...

  others</emph> retained by the people.
  </para>
 </section>
 <section>
  <title>Amendment X (1791)</title>
  <para>The powers not delegated to the United
  States by the Constitution, nor prohibited by
  it to the states, <emph>are reserved to the
  states respectively, or to the people</emph>.
 </para>
 </section>
</bill-o-rights>
```

Figure 2: XML document to be processed

7. **Write an XSL style sheet.** This document is processed using an XSL stylesheet that transforms it into an XML document adhering to the TEXML schema. Instead of using TEX directly, this system uses LATEX, because it has a higher-level interface. LATEX requires the creation of environments (env), commands (cmd), and parameters (parm). The XSL stylesheet identifies elements in the input XML document and outputs an XML document that consists of these env, cmd, and parm elements, plus some others to create the output

```
<?xml version="1.0"?>
<xsl:stylesheet
    version="1.0"
    xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
    <xsl:output method="xml" indent="yes"
        encoding="UTF-8" xml-declaration="yes"/>

    <xsl:template match="/">
        <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="bill-o-rights">
        <TeXML>
            <cmd name="documentclass">
                <parm>article</parm>
            </cmd>
            <cmd name="title">
                <parm>U.S. Bill of Rights</parm>
            </cmd>
            <env name="document">
                <cmd name="date">
                    <parm>1791</parm>
                </cmd>
                <cmd name="maketitle"/>
                <xsl:apply-templates/>
            </env>
        </TeXML>
    </xsl:template>

    <xsl:template match="section">
        <cmd name="section*">
            <parm>
                <xsl:value-of select="title"/>
            </parm>
        </cmd>
        <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="section/title"/>

    <xsl:template match="para">
        <xsl:apply-templates/>
        <cmd name="par"/>
    </xsl:template>

    <xsl:template match="emph">
        <cmd name="emph">
            <parm>
                <xsl:apply-templates/>
            </parm>
        </cmd>
    </xsl:template>

</xsl:stylesheet>
```

Figure 3: XSL Stylesheet to create TEXML output

```
<DIR>    {\LotusXSL}_0_18_5
<DIR>    TeXML_V1R4
<DIR>    xml4j_2_0_15
  4,486  bill-o-rights.xml
  1,356  BOR2TeXML.xsl
    285  xml2tex.bat
```

Figure 4: Directory structure

```
java -cp xml4j_2_0_15\xml4j.jar;
 lotusxsl_0_18_5\lotusxsl.jar
    com.lotus.xsl.Process -in
      %1.xml -xsl %2.xsl -out %1.texml
java -cp TeXML_v1r4\TeXML.jar;
 xml4j_2_0_15\xml4j.jar
    com.ibm.texml.TeXMLatte
      %1.texml %1.tex
latex %1
dvips %1
start %1.dvi
```

Figure 5: Commands to run all processes

document. The LaTeX language is described by the inventor, Leslie Lamport, in his book, *LaTeX: A Documentation Preparation System User's Guide and Reference Manual.*

Our XSL stylesheet is shown in Figure 3.

The output from the XSL transform is an XML document that expresses the contents of the document as a series of environments, commands, and parameters, along with the text that is to be displayed according to the parameters. A program called TeXMLatte processes this XML document and creates a TeX input file. This TeX file is processed by the TeX processor, which creates a DVI file. The DVI file is transformed into a PostScript document with the DVI2PS program, and voila!, you've got paper!

8. **Process your files.**
As we have seen above, several processes need to be executed to go from your XML to paper using the TeX path. You should create a batch file or shell script that executes each one in turn to make the process automated. Using the directory structure shown in Figure 4, your commands look like those shown in Figure 5.

Notice that all of the paths are relative to the directory that contains directory structures for each component. Notice, also, that I have included the jarfiles directly in the java call using the `-cp (classpath)` command-line argument. If you have these in your CLASSPATH environment variable, you don't need to indicate them here.

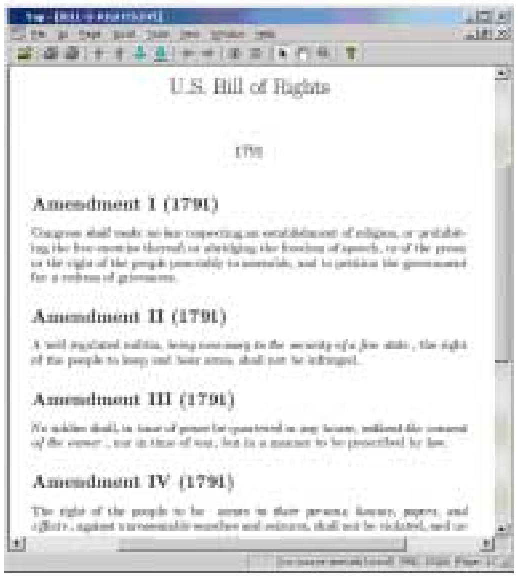
```
<figure>
<verbatim>
<?xml version="1.0" encoding="UTF-8"?>
<TeXML>
 <cmd name="documentclass">
  <parm>article</parm>
 </cmd>
 <cmd name="title">
  <parm>U.S. Bill of Rights</parm>
 </cmd>
  <env name="document">
   <cmd name="date">
    <parm>1791</parm>
   </cmd>
   <cmd name="maketitle"/>
   <cmd name="section*">
    <parm>Amendment I (1791)</parm>
   </cmd>
    Congress
     <cmd name="emph">
      <parm>shall make no law</parm>
     </cmd>
    respecting an establishment of
    religion, or prohibiting the free
    exercise thereof; or abridging the
    freedom of speech, or of the press;
    or the right of the people peaceably
    to assemble, and to petition the
    government for a redress of grievances.
     <cmd name="par"/>
     <cmd name="section*">
      <parm>Amendment II (1791)</parm>
     </cmd>
    A well regulated militia,
     <cmd name="emph">
      <parm>being necessary to the security
        of a free state</parm>
     </cmd>
    , the right of the people to keep and
    bear arms, shall not be infringed.
    <cmd name="par"/>
    ...
  </env>
</TeXML>
```

Figure 6: TeXML document

9. **Check the output.**
When the XSL stylesheet shown here is run against the XML document shown, it produces the TeXML file shown in Figure 6. When this document is processed with the TeXMLatte

```
\def\TeXMLmath#1{\ifmmode#1{}\else$#1{}$\fi}
\def\TeXMLnomath#1{\ifmmode\hbox{#1{}}\else#1{}\fi}
\documentclass{article}
\title{U.S. Bill of Rights}
\begin{document}
\date{1791}
\maketitle
  \section*{Amendment I (1791)}
  Congress \emph{shall make no law} respecting an
  establishment of religion, or prohibiting the
  free exercise thereof; or abridging the freedom
  of speech, or of the press; or the right of the
  people peaceably to assemble, and to petition
  the government for a redress of grievances.\par
  \section*{Amendment II (1791)} A well regulated
  militia, \emph{being necessary to the security
  of a free state}, the right of the people to
  keep and bear arms, shall not be infringed.
  \par
  ...
\end{document}
```

Figure 7: LaTeX document



U.S. Bill of Rights

1791

Amendment I (1791)

Congress shall make no law respecting an establishment of religion, or prohibit-
ing the free exercise thereof; or abridging the freedom of speech, or of the press;
or the right of the people peaceably to assemble, and to petition the government
for a redress of grievances.

Amendment II (1791)

A well regulated militia, *being necessary to the security of a free state* , the right
of the people to keep and bear arms, shall not be infringed.

Amendment III (1791)

No soldier shall, in time of peace be quartered in any house, without the consent
of the owner , nor in time of war, but in a manner to be prescribed by law.

Amendment IV (1791)

The right of the people to be secure in their persons, houses, papers, and
effects , against unreasonable searches and seizures, shall not be violated, and no

Figure 8: The final document (shown in YAP DVI Viewer)

program, the result is a LaTeX file shown in Figure 7. After the LaTeX file is processed with TeX, it can be viewed using a DVI viewer, as shown in Figure 8. <end/>

◇ Brian E. Travis
btravis@architag.com