Implementing DITA XML in a Production Environment

Susan G. Carpenter
IBM Corporation
4205 South Miami Blvd.
Durham, NC USA 27709
(919) 254-1599
carpnter@us.ibm.com

ABSTRACT

This paper describes one information development team's experience with implementing a prototype XML vocabulary in a production environment. This implementation included the migration of pre-existing content, the writing of XSLT and Perl scripts to direct migration and production, and the training of team members.

Categories and Subject Descriptors
I.7.2 [**Markup Languages**]: Document and text processing.

General Terms
Management, Documentation, Performance, Design, Experimentation, Human Factors, Standardization, Languages, Theory.

Keywords
DITA, XML, XSLT.

1. INTRODUCTION

In the fall of 2001, the documentation team for the IBM® WebSphere® Application Server product[i] agreed to participate in a deployment pilot activity for the Darwin Information Typing Architecture (DITA), a publicly available XML vocabulary that originated in IBM's XML Workgroup. Our primary goal was to gain productivity and flexibility by separating source and delivery formats.

The WebSphere development environment is extremely dynamic. Changes to delivery requirements as well as to product content occur right up until the product is released on IBM's Internet product site.

We face additional challenges in our current release: Improving the usability of the information center in every aspect from the layout of content through the selection of search engine. We also needed to achieve greater consistency of content coverage, which meant analyzing and restructuring nearly every bit of content. We looked to DITA to help us re-implement our content such that writers could develop content as other team members sorted out the evolving presentation and delivery requirements. In addition, we looked to XSLT to help us do fast prototyping in the planning stage and eventually to do the "heavy lifting" to transform DITA articles into finished help and information center articles.

2. A SHORT OVERVIEW OF DITA

Darwin Information Typing Architecture (DITA) is an XML vocabulary developed for article-based user assistance. DITA promotes semantic coding primarily by information type; article-level container elements include <concept>, <task>, and <reference>. An undifferentiated container element (<topic>) is also available. DITA enables its users to adapt generalized markup (for example, the reference type) for more specific uses (for example, an API reference type). For more information about DITA, see [1].

3. THE SCOPE OF EFFORT

After our previous product release, writer headcount was cut in half, but we already knew that the next release would require a significant amount of work. We needed to do much more with much less. Our product executive encouraged us to find creative ways of leveraging the resource we did have, offering us his support for a much different way of developing information.

We were also told that our Web-based information center would need an overhaul to meet usability and marketing guidelines. The new marketing guidelines affected the presentation of the information center, which took many months to work out. In part, our usability problems stemmed from uneven and dissimilar coverage of like functionality, resulting in a rather loose "term paper" like narrative structure.

In the fall of 2001, we put together a proposal that included a total overhaul of content, intending to remove as much industry information we could get away with and to focus on information directly tied to the product. The idea was that the team would pare baseline material down to the bone and restructure it before adding material for the new release. This represented a significant departure from previous releases.

Conversion of source to XML made sense for several reasons:

- Writers could focus on reducing and restructuring content while our graphic designer and human factors engineer focused on improving the interface.
- Our production tooling could evolve with the interface.

- We could use a single set of source to generate the information center, subject-based PDF compilations, and contextual help.
- Moving to XML during this overhaul period would put us in a good position to reap benefits from future improvements in Web-based technologies.

3.1 Migrating Pre-Existing Content

Pre-existing content included a Web-based information center, contextual help, and a *Getting Started* book. In addition, a few subjects (such as *Writing Enterprise Beans*) were actually books tied into the information center as offshoot grafts. By volume, most of this material was HTML; the grafted books were IBMIDDoc, an SGML vocabulary that has been in use within IBM for several years. I converted about 1300 HTML articles to DITA for writers to use as raw material. I investigated the migration of IBMIDDoc (migrating over 300 pages worth), but we used very little of it for raw material, mostly for content reasons.

Early in the process, I determined that HTML migration would be difficult, for the following reasons:

- Lack of containment. In a single file that contains several candidate topics, how does one separate and allocate elements algorithmically by topic? In addition, Web browsers are notoriously forgiving of coding such as paragraphs without end tags, but XML parsers are not, so the tagging must be fixed.
- Nonstandard coding. In the absence of strict coding guidelines, the sheer variety of ways to express content in HTML creates significant work for migration tooling.
- Presentational coding. Given the volume of material to be migrated, how does one account for utterly inappropriate coding, such as an h4 element used within a table just to get a desired font?

I set these problems aside for the moment to get a grip on IBMIDDoc migration issues. By comparison to HTML, migration of IBMIDDoc was easy: Compliance with the SGML standard requires containment, and the architecture of IBMIDDoc promotes semantic coding.

Writing and validating SGML migration tooling for our requirements took just a few days. I used a Perl script to make the markup XML-compliant and then used XSLT scripts to convert the content into an undifferentiated DITA topic structure.

That done, I turned back to HTML migration tooling. When finished, the process included the following:

1. Run the HTML Tidy tool to add missing end-tags.
2. Run Perl scripts to add topic-level containment.
3. Visually inspect the results and adjust by hand. This markup still looked enough like HTML that browsers could render it as such. Cascading Style Sheet (CSS) code visualized the different levels of nested topics with color so that writers could inspect renderings rather than code. I followed up later with a more stringent markup inspection.
4. Run a Perl script to strip CSS and make the markup XML-compliant.
5. Run the HTML Tidy tool against the XML-compliant markup for additional cleanup.
6. Run an XSLT script to migrate the XML-compliant markup to DITA <topic> articles.

Migrating 1284 files took about three weeks of full-time work, including the scripting and visual inspection. Post-migration cleanup haunted us for a bit longer; it was managed with content restructuring and reduction.

3.2 Managing the Effects of a Moving Baseline

There were two moving baselines to be managed: that for DITA, and that for the information.

DITA underwent significant change between fall and winter. Colleagues involved with the activity kept me abreast of the most significant changes as they happened.

At first, I updated our copy of the DTD as DITA evolved, but I learned quickly how much of my time that strategy required. Moreover, we were in the midst of constructing baseline content for the information center through a significant amount of restructuring, and the writers could not be distracted from that activity by markup that changed every week

Ultimately, we “froze” our version of the DTD and then “thawed” it at two points that were carefully chosen. Each time, the DITA source had to be converted to the new vocabulary. The “thaw” points represented our best guess at stability and adequacy for the moment: At each point, we believed the proposed changes unlikely to be replaced by another round of future changes, and we decided that the target DTD would be adequate for our needs if no further thaws were needed.

DITA itself helped us deal with evolving delivery requirements. As long as the team held fast to markup that described meaning rather than presentation, we could confine changes to processing scripts.

4. AUTHORING TOOLS AND TRAINING

Most of the team uses the Epic editor from Arbortext. Two writers decided to go with text editors.

The writers wound up needing education on several fronts:

- Information typing
- Article-based writing
- DITA tagging (most writers had HTML backgrounds)
- Working with Epic (most writers had worked only with WYSIWYG editors)

The team leader and I conducted a number of orientation sessions to cover these topics. We also set aside a portion of our weekly team meeting to cover authoring questions or issues. For future reference, we posted job aids in a Lotus Notes® database. This especially came in handy when my manager was able to borrow the use of two writers from another department midway through the release: Orientation information was already waiting for them.

Besides conducting one-time orientation and follow-up sessions, we wanted to reinforce the concepts of information typing and article-based writing over the many months that writers added articles. To that end, we developed a set of information templates in DITA that writers used to assemble new articles. These included the following:

- Concept
- Task overview
- Basic task
- System-file reference
- Command-line interface reference
- Best-practices reference
- Field help and settings reference
- Troubleshooting reference
- Resources for learning (reference)

5. PROCESSING TOOLS

Our processing scripts relied on the following:

- A Perl interpreter that was part of our existing SGML tooling performed global string manipulations during HTML and IBMIDDoc migrations.
- The Xalan-Java tool from the Apache Software Foundation performs XSLT transformations. The binary distribution for Xalan also includes Xerces, an XML parser. For more information about Xalan or Xerces, see [2].

The build tooling is composed of a series of XSLT scripts (one per process step). This is necessary because of a basic characteristic of XSLT 1.0: One can operate on the input document many times in a single invocation but never on the output. As a result, the output from one step becomes the input for the next.

A command-line interface written in the Java™ language facilitates the setting of production parameters in the scripts, incorporates limited catalogue support, and does limited locale processing for internationalization.

Certain support in the XSLT scripts required the use of Xalan-specific extensions to the XSLT recommendation. Whenever possible, Xalan-specific code was segregated and kept to a minimum, keeping open our options for another XSLT engine.

6. LESSONS LEARNED

Given the team's lack of experience with XML and SGML, I expected that they would require lots of education. They didn't. This isn't to say that they didn't have plenty of questions or use some tagging in ways that I didn't anticipate. All in all, they had more trouble with article-based writing than with DITA tagging.

Managing the effects of changing delivery requirements involved trading off impacts to processing with those to authoring. The extended writing team consists of approximately 12 people, so changes to authoring rules have broad impact. Changes to processing tools involve significant testing but affect the work of fewer people. Regardless of the nature of the impact, timing the introduction of changes was important.

Basing a build process on XSLT makes the process very flexible. This is good and bad: Good because it can respond quickly to changes, and bad because it can respond quickly to changes! I rewrote some scripts as many as six times in response to changes in delivery requirements.

Tooling has its limitations. A certain type of recursion fails because of overeager document caching. A kluge to release the cached documents works well but results in less than optimal code.

Our first attempt at adding the build to an automation environment yielded extremely poor performance. A process that took 1.5 hours on the local file system took 11 hours in the first automated build environment. Moving to another automation environment shrank the build time to one comparable with local results. The reason for this behavior is still under investigation.

7. ACKNOWLEDGMENTS

Many thanks go to Don Day for his advice and support. I am also grateful to my team leader, Tricia York, for her wholehearted embrace of XML.

8. REFERENCES

[1] Day, Don R, Priestley, Michael, and Schell, David A. *Introduction to the Darwin Information Typing Architecture* (March 2001, updated May 2002). http://www-106.ibm.com/developerworks/library/x-dita1/index.html.

[2] Apache Software Foundation. *The Apache XML Project*. http://xml.apache.org/index.html.

[i] IBM and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries, or both.