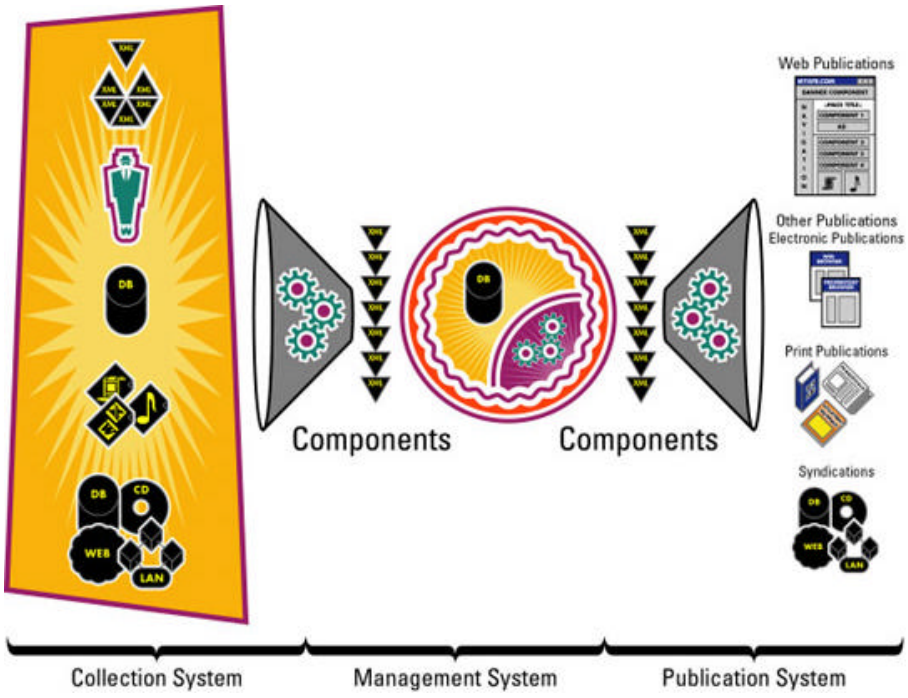
XML and Content Management

A CM Domain White Paper

By Bob Boiko



This white paper is produced from the Content Management Domain which features the full text of the book "Content Management Bible," by Bob Boiko. Owners of the book may access the CM Domain at www.metatorial.com.

Table of Contents

In this white paper I move from talking generally about markup languages to talking specifically about XML. My intention isn't to teach you the hard facts, syntax, and programming behind XML. Plenty of other resources do that. My intention is to give you a conceptual overview of XML that enables you to see how it works and how you may use it in a CMS.

I warn you ahead of time that XML gets pretty hard pretty fast. I try to keep the story at a general readership level, but a few places crop up where it gets a bit thick for the nonprogrammer. For the programmers, your challenge is to move beyond the mechanics of XML to understand the content management concept that the XML in this white paper demonstrates.

What Is XML?

As a content manager, XML should be very important to you. Within the world of markup languages, XML is a good one. It has the advantage of being open and easily read. It's taken

some strong lessons from its "also ran" parent SGML and its superficial cousin HTML. It represents structure, which endures long after format has faded, and naturally encodes the hierarchical relationships that often categorize content.

Best of all, XML is accepted by the Web community as the markup language of choice. Company after company has dropped the pretense that their language is "the one" and have instead rallied around XML. Today's poor XML tools are already being eclipsed by coming killer applications. XML may be one of the first standard markup languages that actually gets accepted as a standard. XML may be most important not for its superior functionality or for its style and sizzle but for the fact that it's a content standard that may stick.

XML addresses two separate but related areas: data interchange and content management. I briefly discuss data exchange to give you a feel for what it is, and then I discuss content management in depth.

XML and data interchange

Before jumping into the aspects of XML that are pivotal to content management, let me spend a moment on a very important aspect of XML that's not pivotal to content management (at least not directly). XML is often discussed as the coming data interchange format for data transfer between computer applications.

There's a vast tower of Babel in the computer systems world. Each applications speaks its own data representation and transfer languages that other applications may or may not understand. XML has begun to provide a lingua franca for all applications. In the past, every company that created a product also created its own scheme for storing and transferring data. (In lieu of any global standard, what else could they do?) XML is, or at least is becoming, that global standard. Now companies can give their products the capability to send and receive data in XML with the knowledge that other applications are doing the same. In the past, it's taken tremendous repeated effort to make various enterprise applications (financial, ERP, operational applications, and the like) talk to each other. And although XML doesn't help you decide what two applications need to say to each other, it does give you the tools to make the conversation that you want to happen possible.

If the Internet's proved one thing, it's that every computer that can be connected to all others will be. After two applications become connected these days, it's likely that XML has some part in how they communicate. I focus very little on the data interchange uses of XML because it plays a far less critical role in content management than does the role XML plays in representing and storing content and metadata.

XML tagging

To bring the idea of XML home, I begin by contrasting the same content marked up in very simple formats, XML, and then HTML. For the programmers, your challenge is to move beyond the mechanics of XML to understand the content management concept that the XML in this white paper demonstrates.

> **Note**
>
> I derived the code samples in this section from an old version of eXcelon Corporation's Extensible Information Server. eXcelon (at www.exceloncorp.com) makes XML database and development tools.

Imagine that you have a lot of content describing different car models. You'd like to organize it and present it on the Web. You've turned to XML rather than a standard database approach because XML offers data management down to the level of the words in a sentence and because XML enables you to produce HTML pages effectively. (You have other possible reasons to turn to XML, but this is sufficient.)

Nominal tagging

Here's how someone who doesn't know about markup languages may specify the information for one car:

```
Name: Dodge Durango
Type: Sport Utility
Doors: 4
Miles: 32000
Price: 18000
Power_Locks: Yes
Power_Windows: Yes
Stereo: Radio/Cassette/CD
Air-Conditioning: Yes
Automatic: Yes
Four-Wheel_Drive: Full/Partial
Note: Very clean
```

Notice that this content is actually marked up. The tag identifier is delimited by the beginning of the line and by a colon. The tag scope is delimited by the end of the line. In fact, this sort of markup is used fairly often by programmers who don't know XML. It's far from sufficient, however, to support managing the content. Although it supports the basic markup qualities, it has no provision for attributes or nesting

XML tagging

Here's the same content marked up with XML

```
<VEHICLES>
   <VEHICLE inventory_number="1">
      <MAKE>Dodge</MAKE>
      <MODEL model_code="USA23">Durango</MODEL>
      <YEAR>1998</YEAR>
      <PICTURE>DodgeDurango.jpg</PICTURE>
      <STYLE>Sport Utility</STYLE>
      <DOORS>4</DOORS>
      <PRICE>18000</PRICE>
      <MILES>32000</MILES>
      <OPTIONS>
         <POWER_LOCKS>Yes</POWER_LOCKS>
         <POWER_WINDOWS>Yes</POWER_WINDOWS>
         <STEREO>Radio/Cassette/CD</STEREO>
         <AIR_CONDITIONING>Yes</AIR-_CONDITIONING>
```

```
            <AUTOMATIC>Yes</AUTOMATIC>
            <FOUR-_WHEEL_DRIVE>Full/Partial</FOUR-WHEEL_DRIVE>
         </OPTIONS>
         <NOTE>Very clean</NOTE>
      </VEHICLE>
   </VEHICLES>
```

Notice that this is longer than the nominal markup that you saw first. It's much more explicit and verbose. More important, there's much more metadata in this representation. The nesting, particularly, adds a big new dimension to the content. You can tell, for example, that I'm talking here generally about vehicles and that one particular vehicle is elaborated. Further, if you need to get to all the car's options, they're all neatly wrapped within the <OPTIONS> tag. Like the first sample, it's easily readable by even the uninitiated. Unlike the first, it gives the distinct and true impression that it's well organized.

Note

XML syntax is just like HTML syntax. In fact, they're the same. They're both derived from the same mother tongue - SGML. XML is hierarchical and has no predefined set of tags. After you set up some tags, the way that you type them follows HTML precisely. That's not to say that, as an HTML savant, you know all that you need to know about XML, but you do know a lot.

XML tags are called *elements*. As in HTML, these elements can have attributes and child elements. Unlike in HTML, you get to decide whether a piece of information shows up as an attribute of its parent element or as a subelement. Consider the following two versions of the same XML description:

```
<HOUSE owner="Elizabeth Higlo" bedrooms="3"/>
```

or

```
<HOUSE>
   <OWNER>Elizabeth Higlo</OWNER>
   <BEDROOMS>3</BEDROOMS>
</HOUSE>
```

Are they the same? Yes and no. They certainly express the same information. On the other hand, the way that you'd access the information programmatically is different in each case. Is one better than the other? Not essentially, but it's hard to say. What you decide to make an attribute instead of a separate tag has more to do with the organizing principles of your tag schema than it does with any innate need to make something an attribute or a subelement. The only coherent principles that I can point to for subelements vs. attributes are as follows:

- **IDs:** IDs and references to IDs (called IDREFS in XML language) are always in attributes. XML is designed that way so that you can use the functions built into most XML validators for making sure that your IDs are unique and that your references are to IDs that exist.
- **Closed lists:** Where you need an element to have closed-list metadata associated with it, an attribute is more convenient than an element because you can specify the allowed values for the field in your Document Type Definition, or DTD. (A *DTD* is a set of rules that you define to enforce consistency in your XML documents. DTDs are discussed more fully later in this white paper.
- **Data in general:** In general, it's more space efficient and readable to have most of the numbers and small text strings (the datalike content) in attributes rather than in elements.

This is the approach taken by HTML where, for example, all the sizes and positions for an image are in the attributes of an <IMG> tag.

It's not that you can't put any information that you want in subelements; it's that, by tradition and for use with XML parsers and authoring environments, it's preferable to put some sorts of information in attributes.

HTML tagging

Look now at the following potential HTML presentation of the same content:

```
<TABLE>
   <TR>
      <TD COLSPAN="2">
         <IMG
src="../scripts/xlnisapi.dll/cars/carsdata/images/DodgeDurango.jpg"
border="1" />
      </TD>
      <TD style="font-family: Verdana;">
         <b>Dodge Durango</b>
      </TD>
   </TR>
   <TR>
      <TD style="font-family: Verdana;">
         Sport Utility
      </TD>
      <TD style="font-family: Verdana;">
         Doors: 4
      </TD>
      <TD>
      </TD>
   </TR>
   <TR>
      <TD style="font-family: Verdana;">
         Miles:32000
      </TD>
      <TD style="font-family: Verdana;">
         Price: 18000
      </TD>
      <TD>
      </TD>
```

```
   </TR>
</TABLE>
<TABLE border="2" cellpadding="2" cellspacing="3" width="380">
   <TR>
      <TD STYLE="background-color=yellow; font:10pt. Verdana;">
         Power_Locks
      </TD>
      <TD STYLE="color:blue; background-color=yellow; font:10pt.
Verdana; font-weight:bold">
         <b>Yes</b>
      </TD>
   </TR>
   <TR>
      <TD STYLE="background-color=yellow; font:10pt. Verdana;">
         Power_Windows
      </TD>
      <TD STYLE="color:blue; background-color=yellow; font:10pt.
Verdana; font-weight:bold">
         <b>Yes</b>
      </TD>
   </TR>
   <TR>
      <TD STYLE="background-color=yellow; font:10pt. Verdana;">
         Stereo
      </TD>
      <TD STYLE="color:blue; background-color=yellow; font:10pt.
Verdana; font-weight:bold">
         <b>Radio/Cassette/CD</b>
      </TD>
   </TR>
   <TR>
      <TD STYLE="background-color=yellow; font:10pt. Verdana;">
         Air-Conditioning
      </TD>
      <TD STYLE="color:blue; background-color=yellow; font:10pt.
Verdana; font-weight:bold">
         <b>Yes</b>
      </TD>
```

```
   </TR>
   <TR>
      <TD STYLE="background-color=yellow; font:10pt. Verdana;">
         Automatic
      </TD>
      <TD STYLE="color:blue; background-color=yellow; font:10pt.
Verdana; font-weight:bold">
         <b>Yes</b>
      </TD>
   </TR>
   <TR>
      <TD STYLE="background-color=yellow; font:10pt. Verdana;">
         Four-Wheel_Drive
      </TD>
      <TD STYLE="color:blue; background-color=yellow; font:10pt.
Verdana; font-weight:bold">
         <b>Full/Partial</b>
      </TD>
   </TR>
   <TR>
      <TD STYLE="background-color=yellow; font:10pt. Verdana;">
         Note
      </TD>
      <TD STYLE="color:blue; background-color=yellow; font:10pt.
Verdana; font-weight:bold">
         <b>Very clean</b>
      </TD>
   </TR>
</TABLE>
```

The first and obvious points are that this version, which contains no more content than the others, is way bigger and much harder to read and interpret. Only the fully trained eye can make out what this content is or how it's designed. If I hadn't been really nice to you by indenting child tags (something rarely done on HTML pages), it would have been even harder to interpret. The size difference can be made up for by faster processors; the difficulty in reading HTML can be made up for by enough practice. What can't be compensated for is the incapability of a program to parse this HTML and decide definitively what's content and what's added formatting and layout.

How, for example, would a program pick all the Notes out of files such as this? You can't search for the word *note* because it may appear anywhere in the content. You can't search for the particular formatting around the note, because other parts of the content may be formatted the same way. Your only hope is to find the note in the file by starting from some definitely unique place (such as the beginning of the file) and tracing a series of steps to get to its location. You

can't move line by line because, as you know, white space (which includes paragraph marks) is ignored in HTML. Possibly, you can move from tag to tag if you know the expected tag sequences or from table row to table row if your content is formatted consistently. Any way that you choose to move through the content is fraught with exceptions and traps. Even a novice can imagine how you can find the content that you want in an XML file. Even an expert may not figure it out for an HTML file.

So, again, I've made the point that XML is better for management than HTML. Fine. The real issue now becomes how to use XML for management and then turn it into HTML (or another format) for publication.

Management by DTD

XML is a wonderful way to express any sort of content structure you can imagine. But, if you can make up a tag for anything, what keeps XML from getting out of hand? How can you ensure that tags are spelled correctly and that they fall in the right place under only particular parent elements? How can you ensure that only certain attributes and attribute values are allowed? In short, how can you ensure that the wonderful structure that you create can ever be enforced?

This was a major issue for the creators of XML's parent, SGML. They ruled, so to speak, that all SGML documents would follow the structure that was defined in a Document Type Definition (DTD). DTDs list, in exacting detail, all the rules behind a particular set of tags. A DTD is metamarkup; it's not the markup itself, but rather markup about markup.

> **Note**
>
> XML schemas are on the way to replacing DTDs as the major way that metamarkup is defined. I stick to a discussion of the still-standard concepts of the DTD. At my level of discussion, you can apply just about all that I say to schemas as well.

DTDs list rules such as the following:

- **Element names:** These are unique names of each element type. They're not the same as the IDs that I've discussed from time to time. Rather, they're the names of the tags that may be used. (In HTML, for example, some of the allowed names are IMG, H1, P, BR, TABLE, and so on).
- **Allowed child elements:** DTDs specify the allowed child elements of each element, including the number of times that the child can occur (once, more than once, and so on) and the order in which the children are allowed to occur.
- **Attributes:** DTDs specify the allowed attributes (if any) for each element. In addition, they specify the type of attribute that it is (a unique ID, a reference to a unique ID, a closed list, and so on). Finally, the DTD can specify whether the attribute is required or optional.

As you may imagine, the DTD for a nontrivial set of tags can be long and complex. Additionally, DTDs use an abbreviated syntax that's tough to learn and read. On the other hand, a good DTD assures you that WYWIWYG (What You Want is What You Get). There's be no room for misspelled or out of place tags.

XML validators are programs that go through XML code and ensure that it conforms to the DTD. As the name implies, the validators complain mercilessly and may not enable you to save an XML document that doesn't conform. The upshot of all this is a general rule of being organized WYGIWYPF (What You Get is What You Pay For!). In markup, as in life, it's costly to be organized; and the more organized you are, the more costly it is. But a time comes in every content system's life where the cost of being organized becomes cheaper than the cost of staying flexible and disorganized. Still, DTDs and the trouble they cause for people who "just want to get something done," may be the biggest reason that SGML never caught on.

To combat this problem, most XML systems enable you to have an XML file without a DTD. Of course, if you leave out a DTD, you're back to the problem of verifying that you have good tagging. Still, by not making you create a DTD right off the bat, XML leaves room for experimenting until you're ready to get serious. This approach avoids the high barrier to getting started that SGML had.

An XML file without a DTD isn't lacking in structure. To be XML, it still must be well formed, meaning that all the brackets and quote marks are in the right place. So interpreters (like browsers) ensure that your XML is well formed. After you load an XML file into an interpreter, it checks that all the syntax is correct. Validators check that XML is valid, meaning that the tags are the ones that you intended and are in the right relationship to each other. Obviously, to be valid, XML must first be well formed. To be well formed, however, XML doesn't need to be valid.

Note

I always think of baby talk when teasing these two concepts apart. Babies babble but not randomly. Rather, they babble in a way that's just like the language that their parents speak. The most impressive thing about baby talk is that you're always just on the verge of making out words even though you know it's basically nonsense. Baby talk is well-formed. It follows the verbal syntax of the ambient language. Its problem is that it's not valid. It doesn't follow the further semantic rules that give particular meaning to a string of sounds.

I leave the details of how to construct a DTD to the standard sources. What I hope to have conveyed here is that a DTD (or schema) is a tool for specifying and enforcing a certain structure on your XML.

Adding formatting

XML is a structural representation of content, which is a good thing for managing that content. You need to know the structure of content to track and manipulate it. Because XML isn't a presentation language, it must be translated into one as it comes time to view it.

Note

This isn't strictly true. If you send an XML file to Microsoft Internet Explorer (5.0 or later), for example, you see something that looks like an outline. It's a formatted version of an XML file. The browser simply reproduces the hierarchy of the XML file on-screen as an expandable tree. Not particularly useful to end users, this nominal formatting is the best that you get from XML without effort.

Following are three basic nonexclusive approaches to adding formatting to XML:

- **You can cheat.** Put formatting tags into the XML file.
- **You can write a custom program.** It can read the XML file and transform it.
- **You can use Extensible Stylesheet Language Transformations (XSLTs).** These are add-ons to XML that transform it into whatever other markup that you want.

The first of these approaches works well (with a big caveat) if your target is HTML. Because HTML and XML share syntax, you can just include whatever HTML tags that you want in your schema or DTD. Here's the caveat: HTML often doesn't follow its own rules. HTML tags are all supposed to be matched with an end tag (even <P> and
 tags), for example, but browsers let you get away with forgetting these. XML requires them and doesn't parse (another word for interpret) without them. So, if you try to pretend that HTML is XML, you often end up with "perfectly fine HTML" that crashes your XML interpreter. For this reason, many people now use what's called XHTML, which is simply HTML that follows all the rules of XML. In addition to the problem of syntax, HTML tags in your content are useful only if your only publications is HTML (not print, for example).

The second of the approaches is the most general, flexible, and powerful. XML offers content the way that programmers like it - well organized, well named, and very accessible. If you're a programmer, you can always write a program that opens an XML file, runs through it branch by branch, and converts all the tags to the appropriate format markup. Along the way, it can do other interesting stuff such as rearrange the sections, leave some out, and mix in sections from other sources (databases and the like). The biggest problem with this approach is that it's labor intensive and requires a programmer to go in and change code every time that you want to change formatting.

XSLT was invented as a general-purpose tool for transforming XML to other formats (primarily HTML). XSLT enables you to walk through the branches of an XML file and take actions based on the tags that you encounter. XSLT has been proposed as the way that content people, not programmers, can take control of the formatting of an XML file.

XSLT files, called *templates*, are written by using the same syntax as are XML. XSLT templates contain commands that select and transform XML tags. These commands form a sort of lightweight programming language that you use to loop, select, and format. Unfortunately, the task of converting structure to formatting is often not all lightweight. In fact, it's sometimes exceedingly difficult and requires advanced programming techniques and a real eye for efficiencies and performance. Thus there's solid doubt in the XML community that XSLT is up to the task. And far from being a nonprogrammer solution, XSLT actually requires a solid programmer to understand and use it. Moreover, the programmer must also be a formatter who can understand both computer programming and content design - not a usual combination of skills.

Finally, today's XSLT (which hasn't even been officially accepted yet) is very clumsy by programming standards and has few development tools to insulate you from its problems. The intent of XSLT is clear: Make it simple to add formatting to XML. Its goal is lofty: Make formatting a separate, nonprogrammer-driven process that can be quickly brought to bear to change the way XML is displayed. At present, it meets neither its goal nor its intent.

Using XML in Content Management

XML can stand behind most electronic information initiatives, including content management. XML enables you to add the structure that you need to content to find it and deliver it. Suppose, for example, that you're a manufacturer and have a Web site that tells your distributors about all about the products that you provide. By using XML, you can create a system behind the site that matches what you know about a distributor to all product information that distributor may want.

In XML parlance, the product information is tagged in such a way that it can be matched to a distributor's profile. If you create a strong XML framework, it not only serves this personalization feature, but it can also form the basis of knowing how to bring new content into the site, how and when to update information, and how to build a variety of outputs, not just a Web site, from your content. Obviously, as the size and complexity of your content increases, so does your need for the organization that XML gives you.

XML in collection

The collection system in a content management system handles gathering content and putting it in a form that's usable for management and publishing. Here are the ways that a CMS may use XML as a content format within the collection system:

- **XML authoring:** Almost any CMS can take input as an XML file, as it can any other file, with no particular parsing or processing to bring it into the system. A better CMS may have an interface with an XML authoring tool, which would enable authors to access the CMS directly from their authoring environments. An XML authoring environment that supports DTDs and an interface into the repository may also be built directly into the CMS.

- **XML conversion:** The CMS may be capable of converting files to XML (from Microsoft Word or other formats) as they're entered into the repository. It may also have tools that enable you to bring some of the rigor of an XML DTD into an author's home environment (for example, by checking for consistency of word processor content as it's being transformed into XML).
- **Rules enforcement:** Your CMS may be able to help enforce structure by managing and deploying DTDs and other files, such as Cascading Style Sheets (CSS), to the authoring environment. Better, the CMS may validate contributed XML files against a master DTD that enforces the content model across all types of input.
- **XML syndication:** The CMS may include tools that enable XML from external sources (syndications) to be automatically parsed and converted to components.

XML in management

The management system in a CMS is responsible for storing and administering content. CMS management systems may use XML as a repository structure. CMS repositories come in three basic types: object (or XML) repositories, relational database repositories, and file system repositories, as the following list describes:

- **Object repositories:** Most CMS products that employ an object repository store XML in the most accessible way. Any level of XML element can be found and used apart from its surrounding content. Object repositories also enable a natural tight linkage between XML elements, the versioning, and workflow systems. Any element can potentially be checked in or out and can participate independently in a workflow (such as revising or publishing). The standards for finding and manipulating XML in these systems generally follows W3C guidelines, such as XPath and XSL (although not all CMS products follow the most recent guidelines). At their best, object-based systems can treat the entire repository as XML that can be validated and fully parsed.
- **Relational database repositories:** Systems that employ a relational database typically store XML in one field of a record. The other fields of the same record store metadata such as author, create date, and the like. The XML itself most commonly isn't directly accessible through the CMS. Generally, you must retrieve the XML and then manipulate it yourself by using standard XML tools such as the Document Object Model (DOM), which I describe more fully later in this white paper The CMS may come with some tools to help work with XML code that's stored in a database field. Thus, although these systems enable you to directly manage metadata, you must manage the XML code itself on your own.
- **File systems:** Systems that employ a file system repository generally store files in their native format - that is, XML files are stored as XML; Microsoft Word files are stored in Word format; and so on. At least one system that I know of converts files to XML and stores both the XML version and the original in the repository. File-based systems generally don't give you direct access to the contents of the files that they handle. Similar to what you must do with the relational database systems, you must retrieve the XML from the file it's stored in and manipulate it yourself by using standard tools such as the DOM.

Although an object repository is the best of the three choices for complete integration and access to XML, the other two technologies generally perform better and are better accepted by IT departments. The next generations of CMS repositories are likely to employ hybrids of these three basic repository systems.

A CMS may use XML to find and represent metadata. A CMS may, for example, use XML to "wrap" metadata around a non-XML file such as a Microsoft Word file or an HTML file. Thus, although the content itself isn't accessible by using XML tools, its metadata is. Better, the system may use an XML DTD to determine the structure of the metadata that's required for each type of content and to ensure that metadata standards are enforced.

Finally, a CMS may use its own proprietary format for representing workflows, or it may use an XML format. If it uses XML, integration with outside workflow players (e-mail servers, for example) becomes easier - as does integrating with or converting to other workflow systems.

XML in publishing

The publishing system of a CMS is responsible for bringing content out of the repository and formatting it into finished pages. A publishing system may use XML for templating. Templates are programs that select content, format it for presentation, and integrate the published page with other programs (with e-commerce systems, for example). Some systems employ open programming language such as ASP/COM (the Microsoft standard) or JSP/J2EE (the Java standard) as the programming platform for templates. These open systems are "XML-agnostic" in that you can choose to integrate with XML but aren't required to do so. To employ XML tools such as the DOM and XSL in the open systems, you must add code that's outside the normal realm of the CMS.

Other systems employ a proprietary programming language to accomplish templating tasks. Within this proprietary realm, some systems use XML to represent their programming constructs. A bit of a particular system's template code, for example, may look as follows:

```
<FIND type="PressRelease">
   <WHERE>
      Author=Jon Doe
   </WHERE>
</FIND>
```

Systems that use this approach can rightly claim that their templating system uses the open standards of XML. They use these standards to create a proprietary programming language, however, that you still must learn to use. You may or may not be able to integrate XML tools, such as the DOM and XSLT, into this sort of system.

Finally, some CMS products simply use the XML template standard XSLT. Because XSLT assumes valid XML as a starting place, the systems that use XSLT must also store content only in XML.

In addition to using XML for templating, your system may be capable of publishing transformed XML. Any CMS that stores XML can output the same XML. Some CMS publishing systems, however, can transform the XML that they store under one DTD into XML under a different DTD for syndication or for publishing to devices that understand a particular DTD (ones for wireless application protocol phones or e-books, for example).

XML in integration

A CMS can use XML as a data exchange format. Today, XML is most used as a data and command interchange format. XML is used to wrap data and commands in such a way as to make them understandable to other applications. Many CMS products take advantage of this fact to connect to other systems. A CMS may, for example, have a tool that imports user data from a particular XML file.

Or the CMS may employ an XML structure for accepting commands from remote servers. In fact, the preceding templating example is just such a situation. The FIND command is represented as an XML element. The CMS accepts this element and carries out the command that it represents. In at least one case, this is the only use that a product makes of XML. This product can rightly say that it uses XML as a core component of its system, even though the XML has nothing to do with the content itself!

Because XML is such an open and useful standard that can represent any structured information, you must look quite closely at the exact way that a CMS uses XML to understand how compatible that system is with your XML needs. Some products are based entirely XML while others use it in a very limited way.

Help from the rest of the XML gang

XML has the buzz, but XML alone can't a system make. I've already discussed XSLT, which fills the formatting vacuum left by XML. Here are a few other attendant abbreviations that help XML toward becoming the clear choice for content management:

- **CSS:** This stands for *Cascading Style Sheets*. These style sheets enable Web designers to do what print designers have done for a long time - create formatting definitions in one place and apply them easily to a number of files. With a CSS attached to a Web page, you can leave out all the formatting from the page. Browsers that support CSS can use the attached style sheet to decide what formatting to apply to each type of HTML tag. CSS makes managing content for the Web much easier. It enables you to leave out the enormous amount of formatting code that you'd otherwise need to include in your publication templates to create fully formatted Web pages. In addition, CSS files aren't so programlike that designers can't read and change them. Designers can even work with full-featured CSS applications that put a slick graphical user interface around the CSS text files. Although XSLT claims to put formatting in the hands of nonprogrammers, CSS really does so.
- **XPath:** This is the way you specify which piece of an XML file you're seeking. The syntax of XPath loosely follows the sort of paths that you type for a URL. XPath, however, has significant additions for specifying search conditions and ordering results. XPath is used in XSLT as well as more generally XML programming (covered in the section "Programming in XML," later in this white paper. XPath is important to content management because it's the way that you find and select components and elements in an XML repository. There was a time when the term *XQL* (for e *X* tensible *Q* uery *L* anguage) was in vogue. XQL used XPath to retrieve information from an XML file. I liked the term XQL because it really expressed the function of XPath as a querying tool.
- **XPointer:** It enables you to pinpoint a location within an XML file that you want to link to. XPointer enables you to get to any point in the file even if it has no tag associated with it. XPointer is of interest to a CMS because it offers a precise way of saying where the referent of a cross-reference is.
- **XLink:** (or XML Linking) This is a way to create the sort of cross-references that you need to make good on the promise of hyperlinks. Among other things, XLink gives you ways to make links that go back and forth between the reference and referent (blurring the distinction between the two terms completely), go to multiple referents from the same reference, and instead of jumping to a referent to expand the content of the referent in place after you click it. XLink also provides for link databases that enable you to organize and manage your linking strategy. At the time that I'm writing this, XLink isn't yet widely available.

XML is young and still exploding. The few extensions that I've listed here are nothing, I'm sure, compared with what's still to come. What you can see already in the development of XML is that the problems that it's confronting are some of the same ones that are most important to content management. XML is already a good choice for an infrastructure behind content management. In the future, it will only get better.

Programming in XML

XML extracts a high cost for all the benefit that it offers. As opposed to HTML, which you can learn in a weekend, XML requires the experience of an intermediate to advanced programmer to be really useful. In addition, the programming paradigm of XML combines concepts from object-

oriented programming with concepts of hierarchical storage and access. For programmers steeped in the older ways of linear programming and table-based access, XML can present a steep learning curve. On the bright side, the concepts of XML mesh so nicely with the concepts of content management that what you learn about XML pays off twice.

Who needs to know XML?

To give you a feeling for who in your organization should be most focused on XML, here are the kinds of tasks that an XML system requires and some comments on the best sorts of people to do them. I've arranged the tasks from least to most technical, as follows:

- **Authoring:** You can take two approaches to authoring XML. You can convert the author's output to XML, or you can try to get your authors to create content inside an XML authoring environment. The first option is what's usually taken today. To do so, you try to get as much of the DTD structure as you can into the author's native environment. You may use style sheets, templates, and macros in Microsoft Word, for example, to try to get authors to create Word files with as much of the rigor of XML as you can manage to simulate in that unstructured environment. The second option, using an XML authoring tool, isn't as far out as you may think. Having created this white paper entirely in one such environment, I can say that, although it's not nearly as friendly as Word, it's getting there. With enough effort, I could make this tool (XMetaL by SoftQuad, at www.softquad.com) workable for nontechnical end users. In the future, I'm sure that this tool and others will advance to the point at which it's not so much effort on your part to bring XML authoring to nontechnical contributors.
- **Rule creation:** Someone must create the logic and structure that your XML files implement. This structure is what I've called the *content model*. It includes all the component classes, their elements, and the access structures that they participate in. In my scheme of CMS jobs, the Content Analyst is clearly the person to do this work.
- **DTD creation:** Someone must design and code the DTDs or schemas that you use. Although this task often falls to programmers, I believe that it's better left to the Content Analyst. If this person isn't up to the challenge of mastering the tools and syntax of DTDs and schemas, you have the wrong person in the position or your analyst hasn't yet realized that these technologies are simply a very formal way of writing down the sort of rules that she's been defining all along.
- **XSLT:** XSLT files are programs. Contrary to some of the claims that I've heard made about XSLT, it's hard for me to imagine anyone but a trained programmer (or someone who wants to become the equivalent of one) able to create more than simple XSLT templates. In my job scheme, it's the template programmer who's best targeted to XSLT. XSLT is a bit behind many of the more graphical and ease-oriented languages in use today; if you can't find someone with experience in XSLT, look for someone who has a lot of patience and doesn't mind the lack of an integrated development environment. Also, as with all template programmers, an XSLT programmer needs to be conscious of and interact well with the needs of the designers with whom she shares her template files.
- **Tool automation:** If you release an XML authoring environment to your nontechnical contributors, you need someone to program it so that it implements your DTDs and has the kind of user interface that the contributors expect and understand. This programming likely includes CSS, XSLT, some sort of scripting, DOM programming, and distribution programs (setup programs and the like). It may also include integration programming with the author's other tools. The custom application developer in my framework ought to do well in this position as long as she's comfortable with the XML programming tasks.
- **Conversion programming:** If you intend to convert the output of authoring tools to XML or convert acquired content to XML, you need conversion programming done. In my scheme, it's the conversion analysis and the tool creator who'd most qualify for this sort of task.

- **Custom template programming:** If XSLT isn't suited to the needs of your CMS, you need to develop other programs to produce your publications. XSLT, for example, may not help you much on any non-HTML publications. In addition, if your template programmers revolt against XSLT, you may need to forgo it for more standard techniques. By using the XML DOM, programmers can do most of or all the operations of XSLT in any standard programming environment. In any case, you want to use the equivalent of an advanced template programmer or custom application developer for the task of going beyond XSLT.
- **Management programming:** If you store your content in an XML repository, you need programming to ensure that the content there can be managed correctly. You may need a program written that goes through the repository periodically, for example, and archives any components whose expiration date has been reached. In addition, you may need any number of programs written that connect to the repository, convert data to XML, and import the data per the DTD into your repository. These activities may require your most advanced programmers, who not only have a mastery of external systems, but also of XML programming using the DOM (covered in the section " Introducing DOM XML Programming," later in this white paper.

If your CMS uses XML extensively, XML skills may be distributed throughout your staff. Even staff who aren't called on directly to create XML can benefit from learning its major concepts. A metator, for example, may never directly create XML but could still really benefit from the ability to read and understand an XML file where the metadata that she creates in Web forms is stored.

Introducing DOM XML programming

The *Document Object Model* (the *DOM*) is a way of giving programs complete and comprehensive access to XML structures. The DOM consists of a set of programming objects with properties and methods for getting to and manipulating XML.

For the purpose of this discussion, you can think of the DOM objects as prepackaged code that delivers significant capabilities with little extra work. In the case of the DOM, the objects deliver the capability to manipulate XML fully. The most significant of the objects the DOM provides are as follows:

- **The Document object:** This object enables you to load XML from files and text strings, search through XML structures, and transform them by using XSLT templates.
- **The Node object:** This object holds a single branch of an XML hierarchy and enables you to operate on it (that is, find, add, update, and delete child elements and content).
- **The Node List object:** This object holds sets of nodes (or more precisely a collection of Node objects).
- **The Node Map object:** This object holds sets of attributes (see Figure 1).

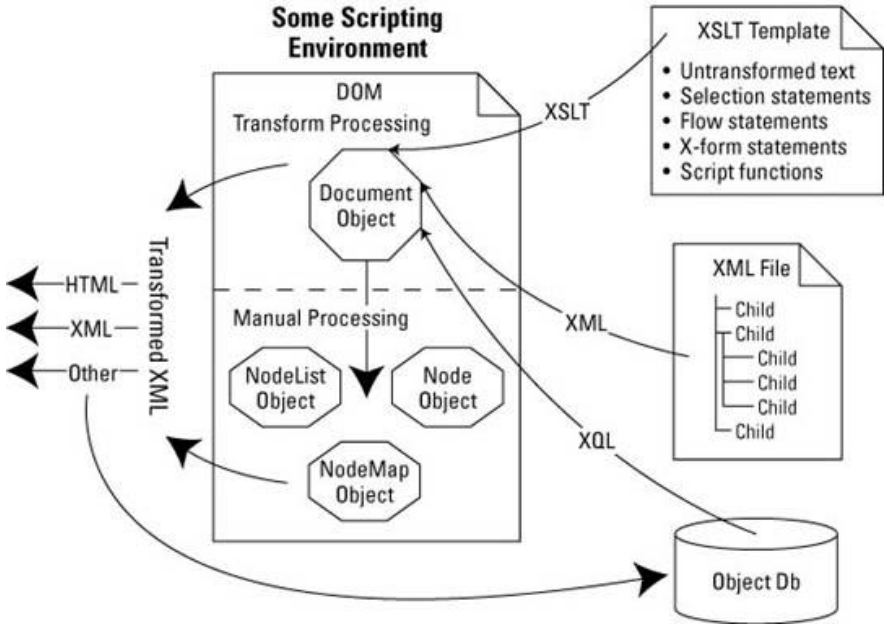


***Figure 1**: The DOM objects and how they fit with both an XSLT transformation and a program that does a transformation in code*

To simplify earlier discussions, I contrast DOM programming with XSLT. I said that you can either use XSLT or DOM programming to transform XML into some final output. In reality, there's only DOM programming. If you use XSLT to transform your XML, what you're really doing is using the XSLT transform methods of the DOM objects. Quite often, you can ignore this fact because the DOM is hidden by some other program that you use. Many XML tools, for example, enable you to choose an XSLT file and "apply" it to an XML file to see how the results look. The tool is simply creating the appropriate DOM objects behind the scenes and invoking the appropriate methods. So whatever method you use to manipulate your XML boils down to DOM programming at some point. Still, it's worthwhile to distinguish between XSLT and DOM programming, because the commands that you can use in an XSLT template are matched (and exceeded) by the commands that you can use within a standard programming environment to directly work with XML through the DOM.

In the sections that follow, I overview the concepts that I just introduced and give you a feel for how they all fit together to enable you to use the DOM to program with XML. I break the discussion into the three standard headings of all programming: input, processing, and output.

Input to the DOM objects

Before you can work with XML, you must load it into a DOM object from some storage area. After it's loaded, you can begin to manipulate the XML directly, or you can load an XSLT file into a DOM object as well and then use the XSLT to transform the XML.

All DOM operations start from the Document object. You load XML into the Document object from the following:

- **An XML file:** This file is located somewhere on the hard drive or at a Web location (that is, accessible from a URL). You can also load XML from a text string that contains well-formed XML.
- **An XML repository or database:** In this case, some form of XQL (or more correctly, XPath) retrieves the XML that you want to load.

XSLT files use the same syntax as XML files. In addition, you use the same methods to load them into the DOM.

DOM processing

After you've loaded an XML structure (and possibly XSLT structure) into a DOM Document object, you can begin to process the XML. You have the following two basic ways to process XML:

- **XSL Transform processing:** You create a Document object for the XML and another for the XSLT. You then tell the first Document object to transform its XML by using the XSLT in the second. The result is a text string that holds the transformed XML. You can then save that text string as a publication page (an HTML file, for example) or send it on to another program (a Web server, for example) that sends it to an end user's browser.
- **Manual processing:** You load an XML structure into a Document object and then use the variety of DOM objects and your own code to walk through the XML and do what you need to do. What you need to do may include converting it to HTML, adding or modifying nodes, or anything else that you can conceive of doing with structured content.

Regardless of whether you use an XSL Transform or your own custom manual processing, the point of the processing is the same — you apply a set of manipulations on an input XML structure to select the appropriate elements, structure them per the requirements of the publication that you're producing, and add the formatting and layout that the target publication requires.

DOM output

First you load XML and then you transform it into a publication page or section by using XSLT or your own code. Finally, you output it to the appropriate file type. Here are the most common of the variety of outputs that you may be interested in obtaining from an XML process:

- **HTML:** Obviously, HTML is a big output. As in the my cars example, HTML is a preferred display format. In this case, the output is destined for the Web and, as often as not, the entire XML process is triggered from some Web link or Web form.
- **XML:** XML, too, is a usual output from an XML process. In fact, it's common to convert from one XML schema to another. As more of your partner organizations bring XML online, for example, you no doubt are required to take your XML content and deliver it to them with their XML tag names and structures. The XML output that you create may be destined for some distant server, for a local hard drive as an XML file, or for some outside repository or XML database.
- **Any manner of other output:** If your XML is well-enough structured, you can probably transform it into just about any other structured form. Of course, a fully automatic process can go only so far. It can't, for example, create an author tag in the output if there's no mention of an author in the input XML. Still, within the constraints of what's possible, you can create just about any type of output that you want. A common output is to a relational database. In this case, the XML is mapped to a set of database rows and columns. The same code that does the mapping to these rows and columns then connects to the target database and stores the content there.

XML programming is just like any other kind of programming. You start from an input, you perform some sort of processing, and the result is a kind of output that's useful to you. XML differs from other types of programming in that it revolves completely around the objects and methods of the DOM, is concerned only with transforming XML, and can use either the language or XSLT or of custom DOM programming to accomplish the transformation.

Server-side XML

So far, I've described in general *how* XML is manipulated. Now I describe *where* it's manipulated. If you're familiar with Web programming, you know that programming code can run either on the Web server or in the Web client. (The Web client is also called the *browser*.) In this section and the next, respectively, I look at how the DOM is integrated into a Web server and a Web client.

You can apply anything that you already know about client and server programming to XML. You'd choose server programming over client programming in XML, for example, for the same reasons that you may choose CGI over client scripting — namely, that you're better off on the server side because you don't need to worry about what type of browser a user has and because, in the often flaky world of Internet programming, you're better off having software run on the one computer that you know rather than the millions that you don't know. On the other hand, if you distribute and run your code on the client, you reduce the number of transactions with the server that you must support and can increase the performance of your code significantly.

In either the server or client scenario, you can assume that you have a Web server, a browser, and whatever HTML pages you need to get your task accomplished. To see how XML processing proceeds on a server that includes the Microsoft product set, look at Figure 2.

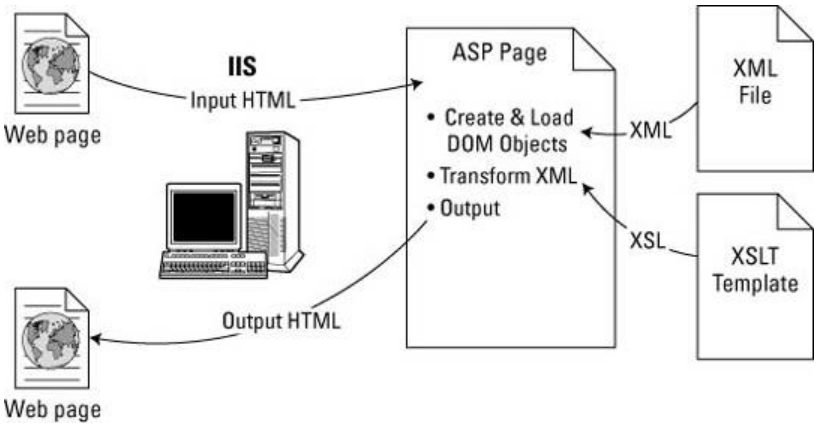


***Figure 2**: XML processing on the server using a Microsoft product set*

To run your XML code on a server, you need the following pieces:

- **Server input:** The XML process is started and supplied with whatever parameters and data it needs by an HTML page of some sort. Suppose, for example, that you wanted to use the car XML that I introduced earlier in this white paperto create an automobile search function. You may create an HTML page that's a car search form where the user can choose a car style (Sports Utility, for example) and hit a Submit button.
- **Server processing:** The parameters and data are submitted through a Web server to the program that performs the XML processing. In my current example (Microsoft software and a car search page), the Internet Information Server (IIS) passes the data onto an Active Server Pages (ASP) page that does the work. The work that the ASP page does is to retrieve the parameters sent by the search page, create DOM objects, do some processing, and pass a result back to the Web server. In the car example, the ASP page would do the following:
  - Retrieve the Car style attribute that the user selected in the form.
  - Load the XML content file into a Document object.
  - Select the appropriate set of XML elements (the ones where the <STYLE> element matches the user's choice).

- Load an XSLT template for the search results page.
- Transform the returned nodes by using the loaded XSLT template.
- Send the result through the server and to the user's browser.
- **Server output:** The output of the XML process is a Web page, usually with XML content transformed into HTML. In my current example, the output is the search-results page with standard surrounding material (logos, banners, navigation buttons, and so on) and the list of links to car pages for each car that matches the style that the user chooses. After the user clicks a link for a car page, you can expect the whole process to run again. The link triggers an XML transformation on the Web server, which selects the XML for the right car and transforms it into a well formatted page.

Your CMS may use server-side XML programming to access content components in a central XML repository and transform then into publication pages. The transformation may be in real time as users click links on a live site. Conversely, you may choose to do the transformation all at once to produce a flat HTML site from the XML that's stored in the repository.

Client-side XML

XML on the client side follows the same basic pattern as does its use on the server side. The biggest difference is that your XML programming code is client-side script. Microsoft Internet Explorer (IE) 5.0 supports XML, as shown in Figure 3.

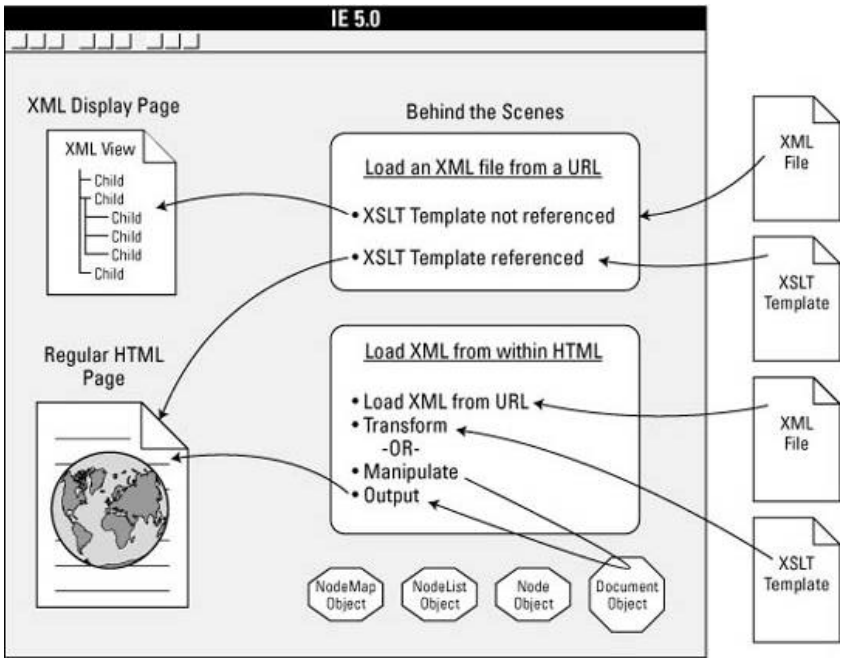


***Figure 3**: The IE 5.0 XML processing model*

Again, the input to the process is an XML file and possibly an XSLT file. IE can load the XML file directly from a URL. The loaded XML file can reference an XSLT file, also by URL, that IE automatically loads and uses to transform the XML file. The result in this case is an HTML page. IE creates the page automatically from the XML and XSLT files that you specify. If you load an XML file into a version of IE that doesn't reference an XSLT template, the browser displays it as a tree. This generally isn't what end users should see but is quite convenient for developers to browse or debug XML.

If you don't want to use this automatic XML loading and XSLT transformation, you can also write some Java or Visual Basic script to do the following:

- Load an XML file and XSLT file and then perform a transformation.
- Load an XML file and then use DOM objects to manipulate it manually and produce an HTML result that's displayed in the browser.

I'm always a bit leery of running code within a Web browser. The code often fails and leaves the end user in complete confusion. Still, at certain times, client code is the only way to provide adequate functionality. If you expect that users may not have a connection to your Web server, for example, you can't expect them to execute code there. Client-side XML plays a fairly small role in commercial content management products. It may be a valuable technique for custom systems, however, but only if you can specify that all users run a browser that can run XML programming script.

Summary

XML is the first markup language that I've seen that has both the power and acceptance to play a central role in content management. XML is all the following:

- A markup language with enough flexibility to represent whatever kind of content structure that you may want to throw at it.
- A potential key player in every part of a CMS.
- A skill that a large range of your staff may need to possess.
- A programming model that's well suited to dealing with the complexities of content storage, access and publication.