Introduction to XML, Part 2: DTDs and XSL Transformations

Last time we introduced XML as a meta language that describes other languages and applications. This week we will take a deeper look into how XML describes languages, and how that data can be transformed for public consumption.

Writing a DTD

To describe a language or XML document, we need to set forth some ground rules about what is accepted. In XML this is done by creating a Document Type Definition or DTD. Writing a DTD simply involves declaring what elements, attributes and so on are going to be allowed in the document and how they will interrelate. You can use a DTD that is local, or on your system, or one that is available publicly.

System DTDs

A DTD is "declared" using a document type declaration (or a DOCYTYPE). The unfortunate similarity between the two terms can be confusing. Just remember to refer to them as DTD and DOCTYPE. XML seems stuck with this terminology, so just remember that the declaration "declares" which DTD you will use. A system DTD can also be understood as a "local" DTD.

So for the cd.xml example we used last week, we will add the DTD to the listing like so:

```
<!DOCTYPE CD_COLLECTION SYSTEM "cd.dtd">
```

The fact that there is no trailing ! is not an error (for whatever reason); you only need it on the opening tag. Breaking it down is simple. After specifying DOCTYPE for the declaration, you list the root element, which in this case is `CD_COLLECTION. SYSTEM` simply says that we are using a DTD that resides somewhere on the host computer (local), not out on the Internet. Obviously, `"cd.dtd"` is our path to the DTD.

Public DTDs

If we wanted to use a public DTD, there are a couple of differences. Instead of using `SYSTEM` you would use the keyword `PUBLIC`, to state you are using a Public DTD. Lastly you add the DTD name and the DTD URL. DTD names have some conventions that tell you some things about the document. If a DTD is an ISO standard, its name begins with ISO. If a non-ISO standards body has approved it, its name begins with a plus sign (+). If no standards body has approved it, its name begins with a hyphen (-). These initial strings are followed by a double slash (//) and the name of the DTD's owner. This is followed by another double slash telling you the type of document the DTD describes, and yet one more set of slashes followed by an ISO 639 language identifier, which just tells you the language of the document. Sounds complicated but it isn't really; look at the following example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

This line probably looks familiar – it's the DTD you see for an XHTML 1.0 Transitional document.

- "HTML" is the root element
- It is a public document (due to the keyword `PUBLIC`)
- The hyphen tells you no standards body has approved the DTD (for some reason, although the W3C specifies HTML, they do not approve it. I don't understand it either.) Oh well, we all use it anyway!
- The DTD's owner is the W3C

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

- The DTD describes transitional xHTML, version 1.0 transitional
- The DTD is in the English language (EN)

Regardless of whether you use a system or public DTD, actually building them requires the exact same process.

Element Declarations

Each tag used in a valid XML document must be declared in the DTD with an element declaration. This declaration specifies the name and possible contents of the tag (or the content specification.) DTDs are conservative: anything not explicitly permitted is forbidden.

We do not have to declare the root element in the DTD, since it is essentially declared in the `DOCTYPE` in the XML document, allowing it to contain anything. To start with an easy element, let's do `BAND` first. To start writing a DTD just fire up any text editor and start typing away – you are simply making a text document:

```
<!ELEMENT BAND (#PCDATA)>
```

This is pretty much the simplest of element declarations. After entering the opening tag and the ELEMENT keyword, you declare your element name (`BAND` in this case). Then in parenthesis you declare what the tag can contain. In this case we have said the `BAND` element can contain Parsed Character Data – PCDATA is nothing more than text that is not markup (tags). Since we are going to put the name of a band in that element (Rush, for example) we specify that it can indeed hold PCDATA. Since `BAND` will not hold any other elements, we are finished.

So you can see there are essentially three parts of the element declaration: the declaration itself, the element name, and what it can contain. That is it in a nutshell; everything else we can do is a variation on what the element can contain. Let's look at the `CD` element next. `CD` is going to have two child elements, so we will need to do a child list within the parenthesis (where you say what the element can contain.) This is very simple:

```
<!ELEMENT CD (BAND, TITLE)>
```

This tells you that every `CD` element must contain one `BAND` element and one `TITLE` element – and nothing else, and in that order. This is called a *sequence*. If for some strange reason I wanted each `CD` to have two titles, I would do this:

```
<!ELEMENT CD (BAND, TITLE, TITLE)>
```

What if we rearranged our code to be listed by band, and each band simply had a list of all the releases (by format – cd, tape or album) they ever released? `BAND` would then be the container for `FORMAT` and `TITLE` – but how would we specify how many of each? Bands put out all different numbers of cds, and we don't just want to put 30 or more `TITLE` elements in the declaration just to be safe. Luckily, there is a simple way to deal with this.

One or More Children

We want to say that each `BAND` will have at least one set of `CD` and `TITLE` (otherwise, we would not include them in our list.) We can do this by adding a plus sign (+) to the element that must have one or more instances, like so:

```
<!ELEMENT BAND (FORMAT, TITLE)+>
```

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

Notice that the plus sign is *outside* the parenthesis – this is important. I'm saying here that I want each BAND element to have both a FORMAT and a TITLE element. If I had done this instead:

```
<!ELEMENT BAND (FORMAT+, TITLE+)>
```

This would not work. This would allow you to give BAND one or more FORMAT elements followed by one or more TITLE elements. What we want is for each BAND to have as many FORMAT and TITLE *pairs* as needed, therefore the plus sign is outside the parenthesis to state that everything *inside* the parenthesis can be repeated one or more times.

Zero or More Children

Suppose you wanted to put a placeholder in for a band you knew you were going to get some music from, but had not yet. One or more would not work – since you had zero. Simply changing the symbol to an asterisk (*) gives you zero or more children, thus:

```
<!ELEMENT BAND (FORMAT, TITLE)*>
```

Now you could enter a BAND name and leave out the following elements for the time being, since the asterisk allows you to have zero children.

Zero or One Child

The last way you can manipulate children is using zero or one. This can be useful in a variety of ways. For example, let's say that some bands re-released an album that has been digitally re-mastered. The title of the album is exactly the same – how can you differentiate this? You could add a REMASTERED element, and allow each instance of the pair FORMAT and TITLE to optionally have a REMASTERED element as well. You use the question mark sign (?) for this, like so:

```
<!ELEMENT BAND (FORMAT, TITLE, REMASTERED?)*>
```

Now the above declaration says that each BAND element will have zero or more pairs of FORMAT and TITLE, and may or may not have one REMASTERED element accompanying the pair.

As you can see, with these three symbols and using parentheses, you have a great deal of flexibility on setting the structure of your documents – and this only scratches the surface of what you can do.

Choices

Adding to your flexibility is choices. Let's say you are writing an article, and this article will have certain things, and may or may not have others. You could declare an ARTICLE tag like so:

```
<!ELEMENT ARTICLE (TITLE, (P | PHOTO | GRAPH | SIDEBAR | PULLQUOTE | SUBHEAD)*,
BYLINE?)>
```

By reading the symbols and the parenthesis we can easily see what is and is not allowed. This element declaration states:

- The element name is ARTICLE

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

- In the outer parenthesis, we see that we must have one `TITLE` element (which is first) and we may or may not have one `BYLINE` (since it has a ? symbol – zero or one)
- The inner set of parenthesis is marked by a * symbol, so we know that the `ARTICLE` can have zero or more children of `P, PHOTO, GRAPH, SIDEBAR, PULLQUOTE` and `SUBHEAD` elements.

This makes sense because many articles contain some of these things in various numbers, but never the same. Declaring the element this way gives you maximum flexibility while still retaining some structure.

Empty Elements

If you are going to use any empty elements (such as defining a `<BR>` tag) you must still declare them. You would declare an empty element like so:

```
<!ELEMENT BR EMPTY>
```

Attribute Declaration

Let's say we wanted to also mention what condition the cd was in – mint, good, etc., and we decided to do that with an attribute in the title element like so:

```
<TITLE CONDITION="MINT">Signals</TITLE>
```

To declare it is simple:

```
<!ATTLIST TITLE CONDITION PCDATA "Not graded">
```

Using the `ATTLIST` declaration, you specify the element that will carry the attribute (`TITLE` in this case), the name of the attribute (`CONDITION`), the type of attribute (there are 10 types, `PCDATA` being most common – and what we used here) and the default value of the attribute (we selected "Not graded" to indicate we had not decided what it should be yet.) You may declare as many attributes using the `ATTLIST` as you want, or do them on separate lines.

Example DTD

So what does our DTD for the cd example look like? Here it is:

```
<!ELEMENT CD(BAND, TITLE)>
<!ELEMENT BAND(#PCDATA)>
<!ELEMENT TITLE(#PCDATA)>
```

`and optionally`

```
<!ATTLIST TITLE CONDITION PCDATA "Not graded">
```

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

Presenting XML - CSS and XSL

Because XML does not handle the presentation of the data, only the structure, you'll need a way to format your data if it is going to be seen by human eyes. CSS and XSL (Extensible Style Language) can both do the job for you, but in different ways.

There are distinct differences between CSS and XSL. XSL is much more than another kind of style sheet. CSS can only change the format of an element (tag), and only on a element-wide basis. XSL is much more powerful. XSL can reorder and rearrange elements and hide and show elements. Additionally, XSL can choose the style of the element based not only on the element, but also on the content and attributes of the tag and a variety of other criteria. Currently CSS has much broader browser support. However, XML documents with XSL style sheets can be converted to HTML documents with CSS stylesheets.

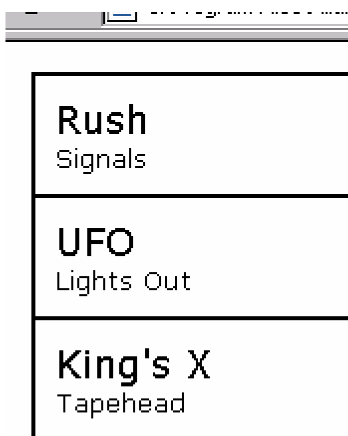
Using CSS with XML

As you saw before with our "Hello XML!" using a CSS style sheet with XML is easy. We can take a quick look at our cd example to see how we could present that to the user using a CSS style sheet as well.

Take a look at this simple style sheet for cd.xml:

```
CD {
      display:block;
      font-size: 18px;
      border: 2px solid #000;
      padding: 10px;
      margin: 10px; }
BAND {display:block; font-size: 14px;}
TITLE {display:block; font-size: 10px;}
```

By setting font sizes and manipulating the box a bit, we get the following display:



XML with basic CSS styling

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

That's pretty basic and easy to read. Keeping in mind the many browser difficulties associated with CSS, you can do all kinds of things with XML and CSS to make your documents look pretty. But XSL is a more powerful way.

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

XSL Transformations (or XSLT)

XSL includes both a transformation and a formatting language. The transformation language provides elements that define rules on how to transform one XML document into another. The formatting language is not much used yet and is incomplete. This document will focus on the transformation language.

XSL transformation language's ability to move data from one XML representation to another makes it an important component of XML-based electronic commerce, electronic data interchange, metadata exchange and any application that needs to convert between different XML representations of the same data. This transfer can be without concern for the presentation to a human, and all about the movement of the data.

With XSLT you can transform an XML document into a well formed HTML document with a CSS style sheet.

Browsers and XSLT

While XSLT can work on the browser side of things, there are reasons why you would not want to do the transformation on the browser side:

- There are browser compatibility issues and the formatting engines are not the same, while server-side XSLT converts it into 100% HTML. It is a more complex solution, however, and not one appropriate for this beginner's article.
- Sending your full XML document to the browser may be too big – and thus suffering a performance hit
- You may not want to send out and share all your information with the client
- Downloading the XML, as well as doing the actual XSLT on the browser can be too slow. Although server-side, real-time XSLT has performance issues as well, this can be somewhat mitigated by caching.

How XSLT Works

In order to do an XSL transformation, the XSL processor has to incorporate a formatting engine, which actually combines the XML and XSL and transforms it. IE5.0+ has a built-in engine for this, but deviates from the standard (It was released before the final specification.)

There are products available (in various forms) that will do this process for you apart from any browser. Xalan, which is part of the Apache XML Project, is one such application. More information about Xalan can be found at http://xml.apache.org/xalan-j/.

What is a "Tree?"

Every well formed XML document is a tree. XSLT sees every node in this tree, unlike CSS – which only sees each element. Thus, attributes, text, comments, and processing instructions are all seen and can be manipulated by XSLT. The language operates by transforming one XML tree into another tree (the transformed tree is called the "results tree".) You can select nodes, reorder nodes and output nodes. A node can be an entire tree itself or one single item, such as a block of text.

It is vital to understand exactly what a tree is to understand how XSLT works. Every well-formed XML document is a tree. A tree is a data structure composed of connected nodes beginning with a single node called a root. The root connects to all the child nodes, each of which may connect to zero or more children of its own, and so on. Nodes without children are called leaves. A diagram of a tree looks much like a family tree chart that lists descendants to a single ancestor. The most useful property of a tree is that each node and its children also form a tree. Thus, a tree is a hierarchical structure of trees in which each tree is built out of smaller trees.

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

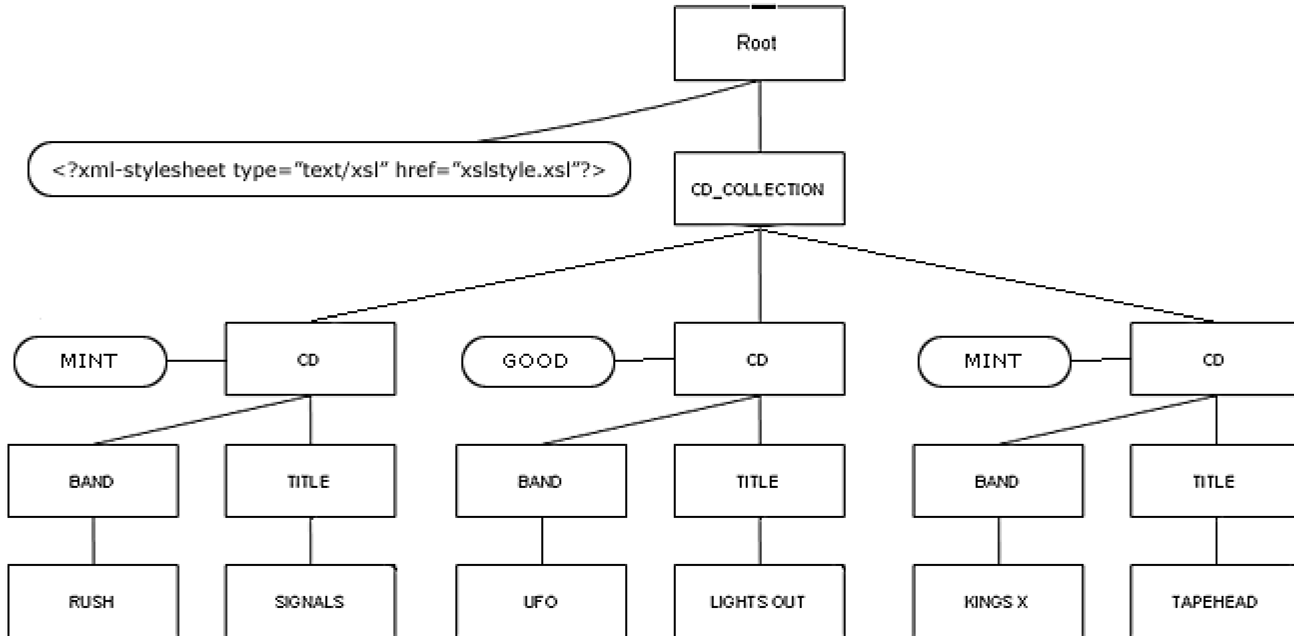
Below is a slightly modified version of our cd.xml example and its tree:

```
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="cd.xsl"?>

<CD_COLLECTION>
      <CD condition="mint">
            <BAND>Rush</BAND>
            <TITLE>Signals</TITLE>
      </CD>
      <CD condition="good">
            <BAND>UFO</BAND>
            <TITLE>Lights Out</TITLE>
      </CD>
      <CD condition="mint">
            <BAND>King's X</BAND>
            <TITLE>Tapehead</TITLE>
      </CD>
</CD_COLLECTION>
```

Note that we added a "CONDITION" attribute, which tells us what state the physical CD is in. This gives us the following tree:

CD.xml Tree



Note that the tree does not see the xml processing instruction or internal DTD subsets or DOCTYPE declarations.

The XSL transformation language operates by transforming one XML tree into another, as stated above. The language contains operators for selecting nodes from the tree, reordering nodes or outputting nodes. Keep in mind that the operators are only designed for operation on a tree, they are not a general regular

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

expression language for transforming arbitrary data. Right now we are concerned with doing a XSL transformation into well formed HTML.

An XSL Style Sheet for CD.xml

Here is our XSL style sheet for transforming this data **on the client** to well formed HTML:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

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

<xsl:template match="CD_COLLECTION">
  <html>
  <head>
        <title>XSL Transformation</title>
  </head>
  <body>

  <h1>A Huge CD Collection</h1>

  <xsl:apply-templates />

  </body>
  </html>
</xsl:template>

<xsl:template match="CD">

   <h2>CD Title</h2>
      <xsl:value-of select="TITLE" />

   <h3>Band</h3>
      <xsl:value-of select="BAND" />

      <h3>Condition:</h3>
      <xsl:value-of select="@condition" />
      <hr/>

   <xsl:apply-templates />

</xsl:template>

</xsl:stylesheet>
```

Matching Templates

Template rules defined by the `xsl:template` element are the most important part of the XSL style sheet. These rules associate particular output (to the results tree) with particular input (from the original XML.) Because all XSL instructions are in the `xsl` namespace (they begin with `xsl:`) it is easy to distinguish between the elements and the literal data to be sent to the results tree and XSL instructions.

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

Remember, XSLT works by matching a tree, or a part of a tree, or a node to a template, processing that instruction and outputting a results tree. Think of the card game "memory." You pick up one card that has a boat on it. You find the other card that has the boat – they match. Once we have the match, the instructions are carried out and you move on.

The XSL Style Sheet: Step by Step

When this style sheet is applied to our cd.xml example, here is what happens:

1. The *root level node* of cd.xml is compared with all template rules in the style sheet. It matches the first one (<xsl:template match="/">) since the "/" designates the root.

   NOTE: Don't mistake **the root element** with the root of the **root level node** of the XML document. The root element is CD_COLLECTION, but the root level node is an imaginary pair of tags surrounding the entirety of your XML document which must be addressed before you can get to the actual tags.

2. There are no instructions to output after the match, so the next element in the style sheet is <xsl:apply-templates />. This element tells the formatting engine **to process the child nodes of what was just matched** (in this case, the root node of the cd.xml). In other words, look at the children of the root node and see if there are any template matches.

   A. The first child of the root node is the <xsl:stylesheet...> processing instruction, which is compared with the template rules. There is no match so there is no output.
   B. The second child of the root node is the root element CD_COLLECTION, and it matches the second template rule <xsl:template match="CD_COLLECTION">.
   C. The <html>, <head>, </head>, <title> (with text), <body>, <h1> (with text) and </h1> are all output, since we had a match.
   D. The <xsl:apply-templates /> element in the <body> tag causes the formatting engine to process the child elements of CD_COLLECTION.
      a. The first child of the CD_COLLECTION element is CD. It matches the third template, <xsl:template match="CD">.

      b. CD has two children and one attribute. The <H#> tags are output – along with a new element named xsl:value-of. This simple element simply says take the value of a tag specified in the "select" attribute and output its contents. For example, in this code the value of the first <BAND> element is the pcdata "Rush." So that text will be output by the engine. You are able to grab the data from these elements even though you have not used a template to match them since they reside in from the currently matched CD element. Therefore we get the <TITLE> of the CD as well.

      c. Lastly, we can grab the attribute value of the CD condition. The <h4> tags and "Condition:" are output, followed by the value of the CONDITION attribute. Putting the "at" sign (@) in front of the element in the select attribute of value-of tells the engine that it is an attribute, not an element. Again, we can grab the value of the attribute since it is a child element of the already matched element.

      At this point the end of the template, and style sheet is reached. The formatting engine now runs back through the templates and closes them.

3. In the CD_COLLECTION match, </body> and </html> are output and the template ends. Processing complete.

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com

We can run this in IE6 - other browsers return mixed results – remember what was said earlier. Using the browser here is only to show the concept of XSLT. Opening cd.xml using IE6 (and ensuring that the .xsl file is where we said it would be) produces this result:

A resulting file of well formed HTML! It will be interesting to note, however, that if you view source you will only see the XML. The transformation takes place behind the scenes.

Conclusion

This article merely exposes you to the "root" concepts (pardon the pun) of XSLT. There is much more under the hood. Eventually, you will also need to learn something called XPath, which is a syntax for defining parts of an XML document. XPath is a W3C standard and is a major part of real world XSLT.

If you have comments about this article, or any issues you would like to see covered in this JavaScript column, please send them to Tom at pixelmech@yahoo.com

Copyright © 2004 DMXzone.com All Rights Reserved
to get more go to DMXzone.com