Using XSLT Stylesheets in Koha

By Sam Cook, Public Services Librarian, Allen Library, U. of Hartford
Created 6/7/2012

Table of Contents

The Basics (if, for-each, value-of)

At its very basic, using XSLT to display a marc field looks like this:

```
<xsl:if test="marc:datafield[@tag=022]">
  <xsl:for-each select="marc:datafield[@tag=022]">
    <xsl:value-of select="marc:subfield[@code='a']" />
  </xsl:for-each>
</xsl:if>
```

If you are familiar with HTML, you can see that this is similar in that it uses tags. In this case, the three tags present are `<xsl:if>`, `<xsl:for-each>`, and `<xsl:value-of>`. Just like HTML, these tags either need to have corresponding closing tags, such as `</xsl:for-each>` and `</xsl:if>`, or have to be closed within the tag with a forward slash, as seen in `<xsl:value-of select="marc:subfield[@code='a']" />`.

Let's look at each of the first three lines of the above code to understand what it says.

<xsl:if test="marc:datafield[@tag=022]">

This is the start of a basic if/then statement. Like all if/then statements in any type of computer code, this statement provides a condition and, if the condition is true, indicates what to do next. In all xslt if/then statements, the condition (or the "if") is found in the test attribute. In this example, the condition being checked is `marc:datafield[@tag=022]`. This code is basically asking if the 022 field is present in this particular MARC record. If the condition is true, then everything between this and the ending </xsl:if> tag happens. If the condition is false, then it skips over everything within the tag.

<xsl:for-each select="marc:datafield[@tag=022]">

The `<xsl:for-each>` tag is basically the way to say "I am going to do something with this MARC field." In this case, it is doing something with the 022 field. In the case of a MARC record that has only one instance of that field, this "something" that happens will only occur once. In the case of a repeated field (such as subject headings), this "something" will happen with each field, in the order that they appear in the MARC record. The thing that happens is whatever lies between the initial `<xsl:if>` tag and the closing `</xsl:if>` tag.

<xsl:value-of select="marc:subfield[@code='a']" />

In this case, this is the "something" that happens. `<xsl:value-of>` retrieves the value of a particular MARC field or subfield. In this case, it is being used to insert the value of 022 subfield a into the HTML, although it can be used for other purposes, such as creating the value of a variable (more on that later). Note that this particular line of code doesn't specify that we are in the 022 field, as the `<xsl:for-each>` already takes care of that. Also note that the tag number (e.g. [@tag=022]) do not have any quotation marks around them, while the subfield letters (e.g. [@code='a']) have single quotes.

So, if you put all of the lines of code together, they say "If the 022 field is present in the MARC record, then display subfield a for each 022." Simple. 022 is the ISSN, so this code will show display the ISSN on the web page.

Making the Code More Useful (HTML, choose/when/otherwise)

The above code is obviously not very complex and is very limiting. Sticking with the ISSN example, here is how the code actually appears in the stylesheet, with new pieces of code highlighted:

```
<xsl:if test="marc:datafield[@tag=022]">
  <span class="results_summary"><span class="label">ISSN: </span>
    <xsl:for-each select="marc:datafield[@tag=022]">
      <xsl:value-of select="marc:subfield[@code='a']"/>
      <xsl:choose>
        <xsl:when test="position()=last()">
          <xsl:text>.</xsl:text>
```

```
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>; </xsl:text>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </span>
</xsl:if>
```

First off, we see the addition of some tags. One of the most important features of an XSLT stylesheet is that you can include HTML and CSS. This allows you to put in other text and take care of formatting. In this case, we see that the results will actually end up reading something like this (depending on what the ISSN actually is:

ISSN: 6187-7621

and the text "ISSN:" is within a span tag with a class of "label" and the entire text is within a span tag with a class of "results_summary". We can then use an external stylesheet to actually format these classes, such as making the label class bold (which is what we have actually done).

The other new code introduces us to another major piece of the XSLT language. What is seen in many other codes as if/then/else statements is seen in XSLT as choose/when/otherwise. This code basically takes the if/then statement one step further by allowing for multiple possible conditions, indicating what should happen if any of the conditions are true, and indicating what should happen if none of those conditions are true. These statements, as demonstrated above, always begin with a `<xsl:choose>` tag and end with a `</xsl:choose>` tag. Each condition to be tested goes within a `<xsl:when>` tag, which works the same way as the `<xsl:if>` tag. The `<xsl:otherwise>` tag indicates what to do if none of the above conditions are true, and thus does not need its own test attribute.

The ISSN example shown above has just one test condition, although this type of code often uses multiple conditions. The condition in this case is `test="position()=last()".` This is testing to see if it is currently on the last 022 field. If so, it applies the code `<xsl:text>.</xsl:text>`. This is just puts a period at the end of the last 022. Note that the `<xsl:text>` tag can be used to insert text into the HTML document, but you cannot put HTML tags in the `<xsl:text>` tag. If it is not the last 022 (aka the condition is false), the <xsl:otherwise> tag indicates that a semi-colon and a space should be inserted instead of a period. So in the case of an item that has more than one ISSN, the resulting display would be:

ISSN: 6848-7681; 8461-7681.

You can have a choose without an otherwise, if you want nothing to happen if none of the conditions in the when tags are true. This could be the case if the when tags capture all the possible values for a particular variable.

Templates

Templates are basically pieces of code that allow for shortcuts. They allow you to reuse a piece of code as many times as you would like without rewriting the code each time. Technically most of the stylesheet is one gigantic template, but we're more concerned with the shorter templates.

There are several at the end of the stylesheets that you can modify to fit your needs, and there are some external templates that you cannot change. I have not found a case where I needed to create my own templates yet, and thus will not discuss how to create a template, as I haven't done it myself. Here is an example of using a template, however:

```
<xsl:if test="marc:datafield[@tag=770]">
    <xsl:for-each select="marc:datafield[@tag=770]">
        <span class="results_summary">
            <span class="label">Supplement: </span>
            <xsl:call-template name="subfieldSelect">
                <xsl:with-param name="codes">abtg</xsl:with-param>
            </xsl:call-template>
        </span>
    </xsl:for-each>
</xsl:if>
```

The `<xsl:call-template>` tag indicates what template will be used. In this case, it is the subfieldSelect template, which is an external template that allows for a quicker way to display multiple subfields. The `<xsl:with-param>` tag indicates what the parameters will be. In this case, the parameter "codes" is given the value abtg, which in this template means that subfields a, b, t, and g will be displayed. If you want more subfields to display, just add them to the list.

You will see the templates chopPunctuation and chopString used many times throughout the XSLT codes. Since these are external templates, I can't see exactly what they do, but I assume they just clean up the MARC fields so they display better.

One useful template is string-replace-all, which takes a piece of text and replaces it with another piece of text. This can be used to take a variable, modify a piece of the text, and save it as a different variable, such as in this code, which replaces a period with nothing (aka eliminates unwanted periods):

```
<xsl:variable name="myVar2">
  <xsl:call-template name="string-replace-all">
    <xsl:with-param name="text" select="$myVar" />
    <xsl:with-param name="replace" select="'.'" />
    <xsl:with-param name="by" select="''" />
  </xsl:call-template>
</xsl:variable>
```

Variables

Just like in any other type of code, variables are hugely valuable in XSLT. They save time, allow for better conditional statements, and so on. What is very important to realize about XSLT, however, is that variables CANNOT be changed after they are declared. Unlike most codes that basically read from top to bottom, thus allowing for variables to change throughout the course of the code, XSLT basically occurs all at once. Also, variables cannot begin with numbers. There are some other minor restrictions, so if your variable doesn't seem to be working, try just giving it a different name.

There are two ways to declare a variable, both using the `<xsl:variable>` tag. The first is very simple:

```
<xsl:variable name="title245"
select="marc:datafield[@tag=245]/marc:subfield[@code='a']" />
```

In this declaration method, the name attribute declares the name of the variable (obvious enough), and the select attribute declares the value. In this case, the variable title245 is assigned the value of whatever is in subfield a of the 245 field.

The other way to declare a variable is slightly more complex, but gives you more power in declaring the variable:

```
<xsl:variable name="count240">
    <xsl:value-of select="count(marc:datafield[@tag=240])"/>
</xsl:variable>
```

In this method, you have opening and closing `<xsl:variable>` tag and the `<xsl:value-of>` tag in between is what assigns the actual value to the variable. In this case, the name of the variable is count240, and the value is a count of the number of 240 tags that appear in the MARC record. This particular example doesn't really take advantage of this type of variable declaration, so let's look at one that is more complex:

```
<xsl:variable name="typeOfSR">
     <xsl:if test="$typeOf008='SR'">
          <xsl:choose>
               <xsl:when
               test="marc:controlfield[@tag=007][substring(text(),1,1
               )='s'][substring(text(),2,1)='d'][substring(text(),4,1
               )='f']">CD
               </xsl:when>
               <xsl:when
               test="marc:controlfield[@tag=007][substring(text(),1,1
               )='s'][substring(text(),2,1)='d'][substring(text(),4,1
               )='b']">LP
               </xsl:when>
               <xsl:otherwise>
               Other
               </xsl:otherwise>
          </xsl:choose>
     </xsl:if>
</xsl:variable>
```

I'll discuss substrings later on, but this is an excellent example of this more complex style of variable declaration, as it uses a choose/when/otherwise statement to decide whether the variable named typeOfSR should be given a value of CD, LP, or Other, based on bits of the 007 code. This would not be possible with the basic variable declaration style showed earlier.

To recall a variable later in the code, you simply put a dollar sign ($) in front of the variable name. In the typeOfSR declaration code seen above, the variable typeOf008 is recalled in the if statement:

```
<xsl:if test="$typeOf008='SR'">
```

The other common place for a variable to be recalled is in a value-of statement, such as:

```
<xsl:value-of select="$title"/>
```

This could be used to insert the variable's value into the HTML code, or to assign it to another variable.

More Complex Statements

In order to dig deeper into the MARC records to create more refined results, you will need to create statements that are more complex than the examples shown above. Listed below are some of the ways (but likely not every way) to get more out of the MARC records.

Using a Subfield

If you want to access a subfield directly, either to set a variable or to use an if statement to determine if that variable exists, use `marc:datafield[@tag=245]/marc:subfield[@code='a']`. In practice, that looks like this:

```
<xsl:variable name="title245"
select="marc:datafield[@tag=245]/marc:subfield[@code='a']" />
```

Using an Indicator

You can also access the indicators with @ind. The below example looks at the 505 field and writes "Contents (as listed on the item):" if the indicator 1 is set to zero.

```
<xsl:if test="marc:datafield[@tag=505]">
<xsl:for-each select="marc:datafield[@tag=505]">
<span class="results_summary">
<xsl:choose>
<xsl:when test="@ind1=0">
<span class="label">Contents (as listed on the item):  </span>
</xsl:when>
```

Substrings

If you need to get even more precise in the MARC record, you can use a substring. A string is basically a piece of text (can include numbers), usually declared as a variable, and a substring is a smaller piece of that text. To get a substring, use the format substring(string,starting character,number of characters). For example, the following code creates a variable called controlField008-21, which is assigned a value of a four-character long string starting at the 25th character of variable contralField008.

```
<xsl:variable name="controlField008-24"
select="substring($controlField008,25,4)"/>
```

There is no position 0 in XSLT but there is in MARC field 008, the 25th character according to XSLT equates to character position 24 in the MARC record. This means that the variable controlField008-24 is actually the string of 008 characters 24, 25, 26, and 27.

You can also use a substring to broaden a MARC field search. For example, the below code tests to see if there are any 6XX fields:

```
<xsl:if test="marc:datafield[substring(@tag, 1, 1) = '6']">
```

This last one is much more complex, as it tests to see if there are any 5XX fields that are not 505, 590, 511, 500, 518, 580, 515, 520, or 510 (basically its creating a catch all for all of the other types of 5XX notes.

```
<xsl:if test="marc:datafield[substring(@tag, 1, 1) =
'5'][not(substring(@tag, 2, 2) = '05')][not(substring(@tag, 2, 2) =
'90')][not(substring(@tag, 2, 2) = '11')][not(substring(@tag, 2, 2) =
'00')][not(substring(@tag, 2, 2) = '18')][not(substring(@tag, 2, 2) =
'80')][not(substring(@tag, 2, 2) = '15')][not(substring(@tag, 2, 2) =
'20')][not(substring(@tag, 2, 2) = '10')]">
```

And/Or Statements

XSLT allows basic Boolean style operators . They work basically like you would expect, but make sure they are lowercase. They are case sensitive. Here are some examples (these lines of code are not related to each other):

```
<xsl:if test="$typeOf008='CF' and
marc:controlfield[@tag=007][substring(.,12,1)='a']">

<xsl:if test="marc:datafield[@tag=440 or @tag=490 or @tag=830]">

<xsl:value-of select="count(marc:datafield[@tag=700 or @tag=710 or
@tag=711])"/>

<xsl:value-of select="count(marc:datafield[@tag=700 or @tag=710 or
@tag=711]/marc:subfield[@code='t'])"/>

<xsl:when test="$leader6='g' or $leader6='k' or $leader6='o' or
$leader6='r'">VM</xsl:when>
```

You can even use nested Boolean operators using parentheses to create statements that are more complex:

```
<xsl:when test="($check008-23 and $controlField008-23='f') or
($check008-29 and $controlField008-29='f')">
```

In some cases, you can omit the “and” entirely, as it is assumed to be there. The below code requires that several characters within the 007 field match the designated letters in order for the condition to be met.

```
<xsl:when
test="marc:controlfield[@tag=007][substring(text(),1,1)='s'][substring
(text(),2,1)='d'][substring(text(),4,1)='f']">CD</xsl:when>
```

There is also a similar shortcut for “or”, using a pipe instead:

```
<xsl:if
test="marc:datafield[@tag=130]|marc:datafield[@tag=730][@ind2!=2]">
```

Not Statements

You can also create “not” statements. This is basically testing to see if a condition is false. Unlike and/or, the statement following “not” needs to be in parentheses, such as in the following example.

```
<xsl:if test="marc:subfield[@code='b'] and
not(marc:subfield[@code='c'])">
```

You can also use the shortcut of != to mean “does not equal.” In this example, the code is testing to see if it is true that the variable typeOf008 does not equal SR.

```
<xsl:if test="$typeOf008!='SR'">
```

You can also test to see if a string does not contain a particular substring, by combining not with contains, discussed in the next section.

```
<xsl:if test="$typeOf008!='SR' and
not(contains($physicalDescription,'videodisc')) and
not(contains($description300,'videocassette')) and
not(contains($physicalDescription,'videocassette'))">
```

Contains

Whereas the substring function discussed above creates a variable or performs a test using a specified part of a larger string, the contains function is used to determine if a particular substring is found anywhere in a larger string.The format is `contains(string,substring)`. The below example tests to see if the variable physicalDescription contains the text *videodisc*.

```
<xsl:if test=”contains($physicalDescription, ‘videodisc’)”>
```

Below is a more complex example of the contains function, which uses both the or operator and several contains function.

```
<xsl:if test="$typeOf008='SR' or
contains($physicalDescription,'videodisc') or
contains($physicalDescription,'videocassette') or
contains($description300,'videocassette')">
```

You can test to see if a string does not include a particular substring by combining contains with not:

```
<xsl:if test=”not(contains($description300,'videocassette'))”>
```

Moving up a Layer

If you have nested for-each statements, you may need to use . or .. to go up one or two layers in order to indicate that you want to stay in the particular MARC field you are in at the moment. In the example below, the first for-each pushes you into the first 245 (there is probably only one, but that doesn’t matter). The next for-each indicates that you should do the following for each

subfield c *within* this particular 245, rather than grabbing every subfield c in the entire MARC record.

```
<xsl:for-each select="marc:datafield[@tag=245]">
<xsl:for-each select="./marc:subfield[@code='c']">
```

If you need to specify a field that is up two levels, use the double period.

```
<xsl:for-each select="..">
```

I know this concept is a little more confusing than others (and difficult to explain), but it should make more sense in context when you actually need to use it, or when you see it in the code.

More Functions

Need more options? You'll find what should be a fairly comprehensive list at http://www.w3schools.com/xpath/xpath_functions.asp. You can also find more explanations of the XSLT language at http://www.w3schools.com/xsl/default.asp.

MARCXML

The reason all of this works is because it is getting the MARC data not from a regular MARC record, but from a MARC record that has been transformed into a MARCXML file. If you were to look at the MARCXML file (which may not be possible right now), you wouldn't see regular MARC, but rather something like this:

```
<datafield tag="500" ind1=" " ind2=" ">
    <subfield code="a">One Mylar sheet included in pocket.</subfield>
</datafield>
```

Everything is wrapped up in XML tags like this. You can see a full example (from which the above code was taken) at http://www.loc.gov/standards/marcxml/Sandburg/sandburg.xml. Note that a Koha MARCXML file also includes information on the items, including call numbers, locations, etc., which is not reflected in this example.

Important Note about Going Live with XSLT Changes

The sandbox and production database stylesheets differ very slightly, so you won't be able to push code from one into the other without making a very small change. Near the very top of the code is an <xsl:import> tag that needs to be modified based on which database you are working in:

Production: <xsl:import href="/home/wxuhartford/kohaclone/koha-tmpl/opac-tmpl/prog/en/xslt/MARC21slimUtils.xsl"/>

Sandbox: <xsl:import href="/home/waldo/kohaclone/koha-tmpl/opac-tmpl/prog/en/xslt/MARC21slimUtils.xsl"/>

It is just a change from "waldo" to "wxuhartford". Without this change, though, the modified stylesheets won't work. **DO NOT skip this step.** Thanks.