XSLT & XPath Tutorial

Table of Contents

1. eXtensible Stylesheet Language Transformation (XSLT)
2. XSLT is rule-based
   Worksheet 1: XSLT - Simple HTML Table
3. XPath: Context, Location Paths
4. XPath: Axis, Predicate
5. XPath: Expressions
6. XPath: Functions
   Worksheet 2: XPath - Simple Example
7. XSLT Elements: xsl:template, xsl:apply-templates
   Worksheet 3: xsl:template & xsl:apply-templates
8. XSLT Elements: xsl:value-of
   Worksheet 4: xsl:value-of
9. XSLT Elements: xsl:for-each, xsl:attribute
10. XSLT Elements: xsl:if
    Worksheet 5: xsl:if
    Worksheet 6: xsl:choose, xsl:when

1 eXtensible Stylesheet Language Transformation (XSLT)

XSL can be confusing because it can refer to several different things. When people talk about XSL, they’re usually referring to XSL Transformations (XSLT) and the Path Language (XPath). XSL also includes a formatting language, XSL FO. The formal recommendation for the XSLT transformation language was approved by the W3C on 26 November 1999.

Cascading Style Sheets (CSS)

The origins of XSL are in Cascading Style Sheets (CSS), where a “stylesheet” is used to add formatting to an HTML file. The syntax to use a stylesheet in XSLT is similar to the syntax in CSS.

XSLT stylesheets have a very different function than CSS stylesheets, however. CSS allows you to define the colours, backgrounds, and font-types for an HTML web page. XSLT allows you to transform an XML file into an HTML file or another text-based format. If you only need web page formatting, then you might still chose to use CSS because it’s more widely implemented for general HTML formatting and most people think it’s easier to learn and use.

The development of XSLT progressed in several stages as people learned more about requirements for the language.

XML Query Language

Since XML allows people to define their own markup tags for documents, transforming one XML document into another became a common requirement. Also, since browsers can’t display XML documents directly, it was necessary to transform XML into HTML to allow browsers to display it on a web page.

To meet these needs, Microsoft, Texcel, and webMethods submitted a proposal in September 1998 to the W3C called the XML Query Language or XQL. Part of that proposal was the use of the XSL pattern language as the basis for a general query mechanism for XML documents.

eXtensible Stylesheet for Transformation

In May of 1999, the W3C decided to unify all the research that had been going on in “a common core semantic model for querying” and one result was introduction of the eXtensible Stylesheet Language for Transformation or XSLT.
XPath

During the development of XSLT, another member of the XML family, known as XPointer was defined. XPointer takes the idea of anchor tags to a new level. Both XPointer and XSLT needed a way to point to various parts of a document. XSLT needed it to select the part of the document that would be transformed and XPointer for linking two documents. The solution was to provide a common syntax and semantics that both XSLT and XPointer could use. This new subset was called XPath. Although XPath is a subset of XSLT, it can also be used on its own.

XPSTL is rule-based

This tutorial will demonstrate the programming features of XSLT. XSLT is different from conventional programming languages because XSLT is based on template rules which specify how XML documents should be processed. Although conventional programming languages are often sequential, template rules can be based in any order because XSLT is a declarative language. The stylesheet declares what output should be produced when a pattern in the XML document is matched.

For example, a stylesheet could declare that when the XSLT transformation engine finds a 'NAME' element it should add markup by calling the 'NAME' template.

The template syntax is:

```xml
<xsl:template match="NAME">
...
</xsl:template>
```

XPath: Context, Location Paths

Before we go into detail on XSLT, let's first discover some basics on how to retrieve information from an XML file, using XPath. XPath is a language for finding the information in an XML document. Using XPath, we can specify the locations of document structures or data in an XML document, and then process the information using XSLT. In practice, it can be difficult to determine where XSLT stops and where XPath starts, but they were developed as two different standards in the W3C.

Context

When we are working with XSLT, the context for a query is the node in the source XML document currently being processed. So in the template `xsl:template match="/"`, we are in the context of the root of the XML document. This isn't the PEOPLE element, it's the whole document, above the PEOPLE element.

When we are in a `xsl:for-each` loop, the context is whichever node we are currently looping through. In our PEOPLE example, it's the PERSON element.

Understanding the context being processed by an XSL template is important. If you're confused, it will help you to understand why your XSL file isn't creating the output file you're expecting. When you're debugging XSL, the first question you should ask is, "What context is being processed?".

Location Paths

The location path sets the context of the node that you're trying to find. We've already seen an example of this, where we are finding all PERSON elements in the PEOPLE element. The context is set using the location path of the root (`match="/"`).

To code a location path, you can use an abbreviated or non-abbreviated syntax. You might need to check which one your parser supports. An example of the non-abbreviated syntax is shown below.

Using the example from Worksheet 1, suppose we insert a new level into the XML document so that the first address is as follows:
If we don’t change the selection in the `xsl:for-each` location path, the first address won’t be found. The following **unabbreviated** selection will work, however.

**Unabbreviated:**

```xml
<xsl:for-each select="child::PEOPLE/descendant::PERSON">
```

An equivalent way to make the selection using the **abbreviated** syntax is as follows.

**Abbreviated:**

```xml
<xsl:for-each select="PEOPLE//PERSON">
```

### Path Expressions

A path expression is an expression used for selecting a node set by following a path or steps. Although the complete set of path expressions includes a much larger group of operators, here are some of the most useful ones:

- **element**
  - Selects all element children of the context node.
  - Example: `article`
  - Select all the child nodes of the `article` element.

- `/`
  - Select from the root node of the current document, of the node in context.
  - Example: `/article`
  - Selects the `article` element, starting at the root of the document.

  - Example: `x/article`
  - Select all `article` elements that are children of `x`.

- `//`
  - Selects nodes in the document from the current context that matches the selection no matter where they are. When used with a context, this operator selects all descendant nodes in the context, no matter how many levels deep they are.
  - Example: `//article`
  - Select all `article` elements no matter where they are in the document.

  - Example: `x//article`
  - Select all `article` nodes that are descendant of the `x` element, no matter where they exist under the `x` element.

### 4 XPath : Axis, Predicate

#### Axis

An **axis** returns a list of nodes, based on the context of the original node.

The following are examples of some the axis location paths.
Expression | Abr. | Comment
--- | --- | ---
self | . | This selects the current node in the context.
Example: `<TD><xsl:value-of select="."/></TD>` if the context node contains a text value, insert this value.
ancestor |  | This selects a path of all parent and parent-of-parent nodes of the current node, starting from the first parent above the context node.
parent | .. | This selects only the single parent of the context node.
attribute | @ | This selects all the attributes of an element.
Example: `<TD><xsl:value-of select="@PERSONID"/></TD>` selects the PERSONID attribute.
child |  | Selects all the children of the current node. This is the default in the abbreviated syntax.

**Predicate**

The Axis helps us to find nodes around the current node. To be able to find a sub-node that contains a specific value, we use a predicate. It consists of a 'qualifying expression' to do the query.

The syntax for the predicate is:

```
[]
```

use square brackets around the expression.

Expression the expression is inside the brackets.

Example: `book[name='Phar Lap']`

The following are example of predicates:

- `article[position()=2]` This is syntax to find the second `article` element.
- `article[starts-with(name, "B")]` This is syntax to find all `article` nodes, whose `name` element starts with 'B'.

5 XPath: Expressions

**Expressions**

Filter patterns can consist of expressions, such as booleans (and, or, etc.), substring and others. These expressions are found as the expression in the predicate or they can stand on their own.

**Node-sets**

Expressions return a node-set selected by the operators, path locations and filters used in the query.

- `=`, `!=`, `>=`, `<=`, `>`, `>=` logical and, or
- `and`, `or` equal to
- `!=` not equal to
- `>`, `>=` greater than, greater than or equal to
- `<`, `<=` less than, less than or equal to (remember to code the `<` character as \&lt; in the XSL file)
- `+`, `-`, `*`, `div` addition, subtraction, multiply, divide
- `mod` modular, returns the integer remainder of a division
- `|` Computes (unions) two node sets

For example:

```
article[author="Jones" and author="Scott"]
```

This returns article nodes where the author is Jones and Scott.

It may be necessary to code the quote marks as apostrophe (single quote mark).
6 XPath: Functions

XSLT stylesheets can take advantage of a large list of functions. The following functions are just a few examples that can be used in XPath queries.

The first part of the XML document that we are going to use in our examples is shown below. A link to the full text of this document is in Worksheet 2.

```xml
<books>
  <book>
    <name link="http://www.amazon.com/exec/obidos/0789722429">XML by Example</name>
    <author>Benoit Marchal</author>
    <listprice>24.99</listprice>
    <price>17.49</price>
    <review>4.5</review>
    <publish>QUE</publish>
  </book>
  <book>
    <name link="http://www.amazon.com/exec/obidos/1861003110">Professional XML</name>
    <author>Mark Birbeck, Michael Kay, Steve Livingstone</author>
    <listprice>49.99</listprice>
    <price>34.99</price>
    <review>4</review>
    <publish>Wrox</publish>
  </book>
  ...
</books>
```

**count**

The count function takes a node-set, which returns a number of the nodes present in that node-set:

`count(nodes)`

In the following example we are looking for the number of books with a review of 3.5 and 4:

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <p>Review of 3.5 = <xsl:value-of select="count(books/book[review=3.5])"/>
    </p>
    <p>Review of 4 = <xsl:value-of select="count(books/book[review=4])"/>
    </p>
  </xsl:template>
</xsl:stylesheet>
```

The output:

Review of 3.5 = 1
Review of 4 = 2

**number**

The number function converts any value to a number. If the value being parsed is a string, it will return 'NaN' (not a number).

In the following example we examine a node-set which contains a number:

```xml
<p>The number is: <xsl:value-of select="number(books/book/price)"/>
</p>
```
The output:

The number is: 17.49

**position**

The `position()` function returns the position value of the element in its context. Some example of using this function is when you need to insert numbering next to a list of items, or when you need to test if the position of element is the last one (which you would test against the `last()` function).

If you sort your elements, it will return the position in its sorted position.

The following example inserts the number next to the book, note we are using the XSLT element, `<xsl:number>` in this example:

```xml
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <xsl:for-each select="books/book">
      <p><xsl:number value="position()"/>. <xsl:value-of select="name"/></p>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

The output:

1. XML by Example
2. Professional XML
3. Xml in Action
4. Xml Design and Implementation
5. XML: A Managers Guide

**substring**

The `substring()` function returns a part of a string, based on the parameters you pass to it. The format is:

```xml
substring(value, start)
substring(value, start, length)
```

The character position(start) begins with 1.

In the following example we are selecting the first 3 letters of the book name:

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <xsl:for-each select="books/book">
      <p><xsl:value-of select="substring(name, 1, 3)"/></p>
    </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

The output:

XML
Pro
Xml
Xml
XML
The sum function calculates totals for a node-set. You need to be aware of nodes that do not contain values, as this function will return 'NaN' if one of the items is not numeric (i.e. empty). You will need to do formatting of your code to overcome this, i.e. replacing empty values with a 0.

\[ \text{sum(node)} \]

In our XML example, if we wanted to calculate the total of all the book prices, our code could read:

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <p>Total Price = <xsl:value-of select="sum(//price)"/></p>
  </xsl:template>
</xsl:stylesheet>
```

The output:

Total Price = 146.41

Conclusion

The above functions are but a few of the functions you can use in XPath. We will be visiting a few more as we go through more examples.

In the DOMDocument, these XPath expressions can be used in the selectSingeNode and selectNodes method for finding a node in the DOMDocument tree.

For example, the following code is searching for all price elements, whose book has a child element called publish with a value of "Wrox".

```plaintext
Set xmlNL = xmlDom.documentElement.selectNodes("/books/book[publish="Wrox"]/price")
```

7  **XSLT Elements: xsl:template, xsl:apply-templates**

We have already discussed how XSLT is the transformation tool, whereby we can take an XML file (or files) and convert it to HTML, another XML file, SQL select statements - or any text-based format.

The XSLT elements are the main building blocks for the actual transformation that occurs in your XSL file.

**xsl:template**

- The template is the basis of the rule-based architecture that XSLT uses.
- You define the template to produce an outcome, so whenever you call that template (using xsl:apply-templates or xsl:call-template), it handles this output.
- The template is processed either by its match or name attributes.
  - The match attribute uses a pattern.
  - The name is whatever name you give it.
- Another important attribute of the template is the mode attribute. If you want different formatting from the general template that you have defined, you can define different modes for that node, which will ignore the general match instruction.

**Note:** You need to have at least one template in your XSLT file.

**xsl:apply-template**

- The apply-template element is always found inside a template body.
- It defines a set of nodes to be processed. In this process, it then may find any 'sub' template rules to process (child elements of element in context).
- If you want the apply-template instruction to only process certain child elements, you can define a select attribute, to only process specific nodes. In the following example, we only want to process the 'name' and 'address' elements in the root document element:
The \texttt{xsl:value-of} instruction is one of the ways of \texttt{writing} the value of text to the output. So far we have seen that the \texttt{xsl:apply-templates} also outputs the text, except that it continues to evaluate the node-set of the element in context (unless differently specified). The \texttt{xsl:value-of} instruction only outputs the node in context.

- **writing** the text in the stylesheet:
  - \texttt{<p>Here is text</p>}
- \texttt{<xsl:copy>}
- \texttt{<xsl:copy-of>}

The \texttt{xsl:for-each} instruction allows you to do looping in XSLT for a given node set. As we’ve seen from a previous example (Worksheet 4, people.xsl), these rows in an HTML table are processed for each person element in the people context:

\begin{verbatim}
<xsl:for-each select="PEOPLE/PERSON">
  <TR>
    <TD><xsl:value-of select="NAME"/></TD>
    <TD><xsl:value-of select="ADDRESS"/></TD>
    <TD><xsl:value-of select="TEL"/></TD>
    <TD><xsl:value-of select="FAX"/></TD>
    <TD><xsl:value-of select="EMAIL"/></TD>
  </TR>
</xsl:for-each>
\end{verbatim}

The \texttt{xsl:attribute} allows you to add an attribute to an element. This element is often used when you need to add a source(src) to an image. Remembering that in HTML an img tag is still an element in XML term. So when we are wanting to add any of these attributes to elements, you need to use this technique.

In this example we are adding a src attribute to an \texttt{<img>} tag:

\begin{verbatim}
<img>
  <xsl:attribute name="src">
    <xsl:value-of select="@src"/>
  </xsl:attribute>
</img>
\end{verbatim}

The output:

\begin{verbatim}
<img src="dodaa.gif">
\end{verbatim}

**Tip:** If you find that the output goes over many lines then you can either:

- Place the code over one line:

\begin{verbatim}
<img><xsl:attribute name="src"><xsl:value-of select="@src"/></xsl:attribute></img>
\end{verbatim}
• or do the following:

```xml
<img>
  <xsl:attribute name="src"><xsl:text/>
  <xsl:value-of select="@src" />  
  </xsl:attribute>
</img>
```

**Attribute Value Template**

Using the `xsl:attribute` element is useful when you want to output an attribute based on a choice (see the example in the `xsl:choose` section).

But if you want a simpler way to output an attribute for an element, you can use the Attribute Value Template, which uses curly brackets {} to surround the XPath expression that you want to output in an attribute.

Using the `xsl:attribute` element is a bit long, so, using the example above you can use the attribute value template to rather output:

```xml
<img src="{@src}" />
```

Attribute value templates can only be used with:

- `xsl:attribute`
- `xsl:element`
- `xsl:number`
- `xsl:processing-instruction`
- `xsl:sort`
- and of course all attributes (such as HTML attributes, `img`, etc) which do not belong to the XSL namespace.

10 **XSLT Elements: xsl:if**

`xsl:if`

```xml
<xsl:if test=boolean expression>
  template body
</xsl:if>
```

- The `xsl:if` element allows an instruction to be run if it meets a criteria, like the `if` statement in most languages, except that there is no `else` statement. The criteria for entering the `if` statement is based on a `test`.
- Criteria: The criteria is evaluated by a `test` expression, which returns a `true` boolean for the following:
  - if the expression returns a node-set, there must be a node in it,
  - if the expression returns a string, the string cannot be empty, and
  - if the expression returns a number, it must not equal zero (0).
- There is no `else` instruction

11 **XSLT Elements: xsl:choose / xsl:when / xsl:otherwise**

`xsl:choose / xsl:when / xsl:otherwise`

The `xsl:if` element was pretty restrictive in that it only allows for one test. The `xsl:choose` element allows multiple choices in a query.
<xsl:choose>
  <xsl:when test=boolean expression>
    template body
  </xsl:when>
  <xsl:otherwise>
    template body
  </xsl:otherwise>
</xsl:choose>

**xsl:choose**

- The *xsl:choose* element is a conditional template which defines a *choice* between alternatives.
- It is similar to *if-then-else* statements, *switch* statements or *select* statements found in other languages.
- There are no attributes for the *xsl:choose* statement.

**xsl:when**

- The *number* of *xsl:when* instructions is limitless.
- There must always be at least one in the body of the *xsl:choose* element.
- The *criteria* for the *test* expression are the same rules found in the *xsl:if* element.
- The first *xsl:when* statement met will be processed. Thereafter it breaks out of the *xsl:choose* conditional template.

**xsl:otherwise**

- If none of the above *xsl:when* criteria are met, then follow the *xsl:otherwise* instructions.
- It is optional in the body of the *xsl:choose* element.
- It has to be the last child of the *xsl:choose* element.

**12 XSLT Elements: xsl:sort, xsl:variable, xsl:param**

**xsl:sort**

```xml
<xsl:sort select = string-expression
        data-type = {"text" | "number" | QName}
        order = {"ascending" | "descending"}
        case-order = {"upper-first" | "lower-first"}
        lang = {language-code} />
```

- The *xsl:sort* instruction defines a sort criteria for a given expression.
- It can only be used in the body of the *xsl:apply-templates* or *xsl:for-each* elements.
- The default sort key is *text*.

From our books.xml example the following demonstrates sorting the books by listprice, using a *number* datatype:

```xml
<xsl:template match="/books">
  <html><basefont face="Verdana" size="2"/>
  <body>
    <xsl:for-each select = "book">
      <xsl:sort select="listprice" data-type="number" />  
      <p>
        <xsl:value-of select="name" /> -  
        <xsl:value-of select="listprice" />
      </p>
    </xsl:for-each>
  </body>
  </html>
</xsl:template>
```
**xsl:variable**

```xml
<xsl:variable name="Qname" select="expression" /></xsl:variable>
```

- The `xsl:variable` is an element that allows you to set a variable in your XSL file.
- The variable can either be global (above the top `xsl:template` section) or local (within an `xsl:template` section).
- You give the variable a name and then define its context (or string) in the `select` attribute.
- The content of the `xsl:variable` can be:
  - a **node set**, which enables you to evaluate the node set later in your document:
    ```xml
    <xsl:variable name="articles" select="/articles" />
    ```
  - a **string** which enables you to use the same string throughout the template in context.
    ```xml
    <xsl:variable name="planet" select="'earth'" />
    ```
    **Note**: The single quote is used in the `select` attribute for a string. This helps you to differentiate between an element and a string.
- Variables are very useful when you want to reference the same data repeatable in your stylesheet and for holding context.
- Variables can only be written to when they are declared, therefore 'write-once', similar to constants in Visual Basic.
- When you reference a variable, you insert a dollar ($) in front of it:
  ```xml
  <xsl:for-each select="$articles/headlines" />
  ```

**xsl:param**

The `xsl:param` works in a similar way to the `xsl:variable`, i.e. the way its declared, using the $ sign to access it, etc.

Some of the differences are:

**xsl:variable:**
- variables can be declared anywhere in your XSL stylesheet.

**xsl:param:**
- param's can only appear as top-level elements (global), for example the child of a `xsl:stylesheet` or as an immediate child of a template (local to the template).
- when declared globally, the value can only be set externally from the XSLT file. This is done via the DOM parser by passing parameters to the XSLT file before doing the transformation.
- the `xsl:call-template` and `xsl:apply-templates` element allows you to pass a parameter value into the template in these functions.