XPath Basics

By Vikram Vaswani

Table of Contents

Shrinking Violet

It's strange but true − even if you've been working with XML for a while, it's quite likely that you've never heard of XPath prior to today. And even if you have, it's probably been in the context of another technology − XSLT or XPointer − with your experience with XPath itself limited to a cursory examination.

Now, that's a shame, because XPath is not only pretty interesting, but it forms an important component of both XML stylesheet tranformations (XSLT) and the XPointer linking language. By providing XML developers with a standard method of addressing any part of an XML document, XPath is a small yet very important piece of the whole XML jigsaw. XSLT uses it extensively to match nodes in an XML source tree, while XPointer uses it in combination with XLink to identify specific locations in an XML document.

My point? Very simple − if you're at all serious about XML, you need to understand XPath. And over the course of this tutorial, I'm going to assist you in this endeavour by explaining the fundamentals of the XPath data model, together with some examples of XPath's more complex expressions and functions.

Before we get going, though, a few disclaimers:

First, I don't claim to be an expert on XPath. Much of the material in this tutorial is based on my own experience as a developer, or gleaned from late−night email conversations over beer and stale pizza. In other words − caveat emptor!

Second, as new XML standards are proposed and disposed, the material here may become invalid; you should always refer to the most current standard or recommendation for up−to−date information (links to everything you need are included at the end of the article.) This tutorial is based on the W3C's XPath 1.0 recommendation.

Now that I have that off my chest − let's get started!

Dog Days

If you've worked with XML data, you already know that the XML specification defines certain rules that a document must adhere to in order to be well−formed. One of the most important rules is that every XML document must have a single outermost element, called the "root element" which, in turn, may contain other elements, nested in a hierarchical manner.

Now, it seems logical to assume that if an XML document is laid out in this structured, hierarchical tree, it's possible to move at will from any node on the tree to any other node on the tree. And that's where XPath comes in − it provides a standard addressing mechanism for an XML document which lays bare every element, attribute and text node on the tree, making it a snap to access and manipulate them.

In fact, XPath gets its name from the fact that node addresses look a lot like standard *NIX or Windows paths − a hierarchical list of all the branches between the current node and the tree root, separated by slashes.

XPath represents an XML document as a tree containing a number of different node types − seven of them, actually. In order to illustrate this, consider the following XML document:

```
<?xml version="1.0"?>
<movie>
<title>X−Men</title>
<!−− in case you didn't know, this is based on the comic − Ed
−−>
<cast>Hugh Jackman, Patrick Stewart and Ian McKellen</cast>
<director>Bryan Singer</director>
<year>2000</year>
<?play_trailer?>
</movie>
```

Here's how XPath would represent this:

```
[root]
| −− movie
| −− title
| −− X−Men
| −− in case you didn't know...
| −− cast
| −− Hugh Jackman, Patrick Stewart and Ian McKellen
| −− director
| -- Bryan Singer
| -- year
| -- 2000
| -- play_trailer
```

As you can see, the various nodes in the tree above are not identical − some are elements, some contain text fragments, and some simply represent comments. Since XML itself supports a limited number of constructs, the XPath specification is able to categorize these different types of nodes into:

Element nodes: Elements within the XML document are represented as element nodes in the XPath data model. Since elements can have other elements nested within them, they typically appear as branches on the tree (although so−called "empty" elements would appear as leaves.) In the example above, "title" would be an element node.

Text nodes: The character sequences that are enclosed within elements constitute text nodes on the XPath tree. If a text node contains an entity reference, it is automatically expanded to its full value. In the example above, "X−Men" would be a text node.

Attribute nodes: If an element has attributes, those attributes are also represented as nodes; however, since attributes are always linked to elements, they appear as children of the corresponding element node.

Namespace nodes: If an XML document defines one or more namespaces for the elements within it, these namespace declarations are represented as separate nodes by XPath. Like attributes, namespace nodes appear as children of the associated element node in the XPath tree − you can see this from the diagram above.

Processing instruction (PI) nodes: If a document contains a processing instruction − well, that's a separate node too. Note, however, that although the XML declaration at the top of the document is a PI, there exists no node corresponding to it.

Comment nodes: You figure this one out...

Now, in addition to these six types (which, if you look at your leather−bound copy of the XML specification, correspond rather closely with the six basic constructs available in XML), XPath also defines something called a "root node", which is unique to every XML document. This root node represents the base of the XML document tree, and encloses everything within it. There can be only one root node in an XML document, and all other elements within the document exist as children of this root node.

It should be noted that the root node of a document is not the same as the outermost element (sometimes referred to as the "document element"); rather, as the representation above describes, it is a hypothetical node which exists as the parent of the outermost element

The hierarchical nature of XML data itself imposes a couple of other rules, which you might think of as pretty obvious − however, they bear repeating in this context. Every node (other than the root node) has a single parent. Every node (including the root node) may have one or more children. And every dog has his day.

First Steps

Given this basic structure, XPath now makes it possible to locate a node, or set of nodes, at any level of the tree, using a thingamajig known as a "location path."

A location path may be either an absolute path, which expresses a location with reference to the root node, or a relative path, which expresses a location with reference to the current node (since this location is always in context to something else, it is also referred to as the "context node"). Location paths are made up of a series of "location steps", each identifying one level in the XPath tree and separated from each other by a forward slash (/).

A location step can be further broken down into three components: there's an "axis", which defines the relationship to use when selecting nodes; a "node test", which specifies the types of nodes to select; and optional "predicates" to filter out unwanted nodes from the resulting collection (I'll explain each of these in detail further down so that they become a little less frightening.)

The syntax of a location step is as follows

```
axis::node−test[predicates]
```

Revolving Around An Axis

Since the first component of a location step is the axis, let's deal with that first. An axis defines the relationship between the current node and the nodes to be selected − whether, for example, they are children of the current node, siblings of the current node, or the parent of the current node.

The XPath specification defines the following axes:

self − the "self" axis refers to the context node itself;

parent − the "parent" axis selects the parent of the context node;

child − the "child" axis selects the children of the context node;

attribute − the "attribute" axis refers to the attributes of the context node;

These are the most commonly−used ones; however, it's quite likely that you'll also find a use for:

ancestor − the "ancestor" axis selects the parent, grandparent, great−grandparent and all other ancestors of the context node;

descendant − the opposite of the "ancestor" axis, this axis selects the children (and children's children) of the context node;

ancestor−or−self − this variant of the "ancestor" axis selects all ancestors of the context node as well as the node itself;

descendant−or−self − this variant of the "descendant" axis selects all descendants of the context node as well as the node itself;

following−sibling and preceding−sibling − these two axes contain the nodes at the same level in the document tree as the context node. Depending on which axis you use, you will get a collection of siblings which are either after or before the context node, in document order.

following and preceding − the "following" axis selects all nodes within the document tree which follow (are placed after) the context node, while the "preceding" axis selects all nodes which come before the context node.

namespace − the "namespace" axis selects all the nodes in the same namespace as the context node;

Once the relationship to be established has been defined and an appropriate node collection obtained, a node test can be used to further filter the items in the collection. This node test is connected to the axis by a double colon (::) symbol.

Typically, you would select a node on the basis of its name without using a node test; in such a case, the node type is deduced from the axis part of the location step. If, on the other hand, you'd like to select nodes on the basis of type, XPath offers some pre−defined node tests; the text() node test selects text nodes, the comment() function selects comments, the processing−instruction() function selects PIs and the generic node() function selects any and all nodes.

Finally, in case the resulting collection needs to be further broken down, XPath allows you to add optional predicates to each location step, enclosed within square braces.

Proof Of The Pudding

By combining the axis and node test into a location step, and combining multiple location steps into a location path, it becomes possible to locate specific nodes with the document tree quite easily. Using the following XML sample, let's consider some examples.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="recipe.xsl"?>
<movie id="67" genre="sci-fi">
<title>X-Men</title>
<!-- in case you didn't know, this is based on the comic - Ed
-->
<cast>Hugh Jackman, Patrick Stewart and Ian McKellen</cast>
<director>Bryan Singer</director>
<year>2000</year>
<?play_trailer?>
</movie>
```

The path

```
/child::movie/child::cast/child::text()
```

references the text node

```
Hugh Jackman, Patrick Stewart and Ian McKellen
```

In order to make this a little easier to read (and write), XPath assumes a default axis of "child" if none is specified − which means that I could also write the above path as

```
/movie/cast/text()
```

Similarly, the path

```
/movie/comment()
```

references the comment string

```
in case you didn't know, this is based on the comic − Ed
```

while the path

```
/movie/node()[8]
```

references the string

```
Bryan Singer
```

The * character matches all child elements of the context node, while the @ prefix indicates that attributes, rather than elements, are to be matched. The path

```
/movie/*
```

would match all the children of the "movie" element, while the path

```
/movie/@*
```

would refer to all the attributes of the movie element. In case I need a specific attribute − say, "genre", I could use the path

```
/movie/@genre
```

or the path

```
/movie/attribute::genre
```

both of which would reference the value

```
sci−fi
```

Finally, the path

```
/*
```

would reference the first element under the document root, which also happens to be the outermost element, while the path

```
//*
```

selects all the elements in the document.

Playing Chicken

Let's try a slightly more complex example:

```
<?xml version="1.0"?>



<recipe>



<name>Chicken Tikka</name>
<author>Anonymous</author>
<date>1 June 1999</date>



<ingredients>



<item>
<desc>Boneless chicken breasts</desc>
<quantity>2</quantity>
</item>



<item>
<desc>Chopped onions</desc>
<quantity>2</quantity>
</item>



<item>
<desc>Ginger</desc>
<quantity>1 tsp</quantity>
</item>



<item>
<desc>Garlic</desc>
```

```
<quantity>1 tsp</quantity>
</item>



<item>
<desc>Red chili powder</desc>
<quantity>1 tsp</quantity>
</item>



<item>
<desc>Coriander seeds</desc>
<quantity>1 tsp</quantity>
</item>



<item>
<desc>Lime juice</desc>
<quantity>2 tbsp</quantity>
</item>



<item>
<desc>Butter</desc>
<quantity>1 tbsp</quantity>
</item>
</ingredients>



<servings>3</servings>



<process>
<step>Cut chicken into cubes, wash and apply lime juice and
salt</step>
<step>Add ginger, garlic, chili, coriander and lime juice in a
separate
bowl</step>
<step>Mix well, and add chicken to marinate for 3−4
hours</step>
<step>Place chicken pieces on skewers and barbeque</step>
<step>Remove, apply butter, and barbeque again until meat is
tender</step>
```

```
<step>Garnish with lemon and chopped onions</step>
</process>


</recipe>
```

Now, if I wanted to get to the third ingredient, I would use the path

```
/recipe/ingredients/item[3]/desc/text()
```

whish references the text string

```
Ginger
```

Note the predicate used to get the third item in the list.

If I needed to get the number of servings, I could use

```
/recipe/servings/text()
```

or

```
//servings/text()
```

The // shortcut will select elements of that name anywhere below the current context node, and is equivalent to the "descendant−or−self" axis. So the path

```
//item
```

would select all the "item" nodes in the document, while the path

```
//item[7]/quantity
```

would reference the "quantity" element under the seventh "item" element and the path

```
//item[7]/quantity/text()
```

would reference the text node

```
2 tbsp
```

If I wanted to get really funky, I could use a different predicate to identify the appropriate node − the paths

```
//item[7]/quantity/text()
```

and

```
//item[desc/text()='Lime juice']/quantity/text()
```

are equivalent.

More examples of this nature are available in the XPath specification at http://www.w3.org/TR/xpath.html

Operating With Extreme Caution

In addition to what you've just seen, XPath also allows you to build expressions using simple logical and comparison operators. Consider the following table, which illustrates the common ones with examples:

Operator	What It Means	Example
=	is equal to	item[desc = 'Ginger']
!=	is not equal to	item[desc != 'Ginger']
	is greater than	servings > 2
<	is less than	servings < 5
=	is greater than or equal to	servings >= 2
<=	is less than or equal to	servings <= 10

Most of these comparison operators are used in conjunction with XSLT's "if" and "choose" tests (note that if you use them in an XSLT stylesheet, the < symbol must be replaced with the pre−defined XML entity < to avoid XML errors.).

You can combine expressions using the "and" and "or" operators,

Operator	Example
and	desc = 'Ginger' and quantity = '1 tsp'
or	desc = 'Cinnamon' or servings >= 3

Here's an example of how you could apply this in an XSLT stylesheet:

```
<xsl:template match="/">
<xsl:if test="//item/desc = 'Cinnamon' or //servings >= 3">
<xsl:value-of select="//name"/>
</xsl:if>
</xsl:template>
```

Finally, you can perform arithmetic operations with the various arithmetic operators:

```
Operator What It Means Example
---------------------------------------------------------------------------
--

+ Addition quantity + 5

- Subtraction quantity - 5

* Multiplication quantity * 5

div Division quantity div 5

mod Modulo quantity mod 5
```

Here's an example of how you could apply this in an XSLT stylesheet:

```
<xsl:template match="/">
Current servings: <xsl:value-of select="//servings"/>
Updated servings: <xsl:value-of select="//servings * 6"/>
</xsl:template>
```

Be Cool

In addition to these operators, XPath also comes with some useful arithmetic and non−arithmetic functions − here's a list of the most useful ones::

```
Function What It Does
-----------------------------------------------------------------
name() returns the name of the element

count (collection) returns the number of nodes in the
collection

last() returns the number of the last node in the current
collection

position() returns the position of the context node in the
current
collection

sum (collection) returns the sum of the node values in the
collection

round (num) returns a rounded integer value

concat (str1, str2, ...) concatenates its arguments into a
single string

contains (str, substr) returns true if string contains
substring

substring (str, start, len) returns a substring of length
(len) from
position (start)

string−length (str) returns the number of characters in the
string
```

I'll illustrate this with reference to the following XML document:

```
<?xml version="1.0"?>

<bookstore owner="me" location="East Fifty-Third Street">

<title>Be Cool</title>
<author>Elmore Leonard</author>
<price>7.97</price>
<quantity>150</quantity>

<title>Mystic River</title>
<author>Dennis Lehane</author>
<price>25.00</price>
<quantity>86</quantity>

<title>Hit List</title>
<author>Lawrence Block</author>
<price>23.76</price>
<quantity>26</quantity>

<title>Silent Joe</title>
<author>T. Jefferson Parker</author>
<price>24.99</price>
<quantity>268</quantity>

<title>The Travel Detective</title>
<author>Peter Greenberg</author>
<price>34.87</price>
<quantity>9</quantity>

</bookstore>
```

Here are some examples of how these functions might be used.

The rule

```
<xsl:template match="/">
Total number of items in stock = <xsl:value-of
select="sum(//quantity)"/>
</xsl:template>
```

returns the sum of all the available quantities

```
Total number of items in stock = 539
```

The rule

```
<xsl:template match="/">
Total number of unique titles available for sale =
<xsl:value-of
select="count(//title)"/>
</xsl:template>
```

returns the number of titles available

```
Total number of unique titles available for sale = 5
```

The rule

```
<xsl:template match="/">
<xsl:value-of select="//title[last()]"/>
</xsl:template>
```

returns the last title in the collection.

```
The Travel Detective
```

The rule

```
<xsl:template match="/">
Total value of inventory = <xsl:value-of
select="round((//quantity[1] *
//price[1]) + (//quantity[2] * //price[2]) + (//quantity[3] *
//price[3]) +
(//quantity[4] * //price[4]) + (//quantity[5] *
//price[5]))"/>
</xsl:template>
```

multiplies price by quantity and adds it all up.

```
Total value of inventory = 10974
```

The rule

```
<xsl:template match="/">
There are <xsl:value-of select="string-length(//title[1])"/>
characters in
<xsl:value-of select="//title[1]"/>
</xsl:template>
```

returns the length of the selected text node

```
There are 7 characters in Be Cool
```

The rule

```
<xsl:template match="/bookstore">
<xsl:for-each select="title">
<xsl:value-of select="self::title" /> - <xsl:value-of
select="following-sibling::author" />;
</xsl:for-each>
</xsl:template>
```

returns a list of title−author combinations

```
Be Cool - Elmore Leonard;
Mystic River - Dennis Lehane;
Hit List - Lawrence Block;
Silent Joe - T. Jefferson Parker;
The Travel Detective - Peter Greenberg;
```

And finally, the rule

```
<xsl:template match="/">
<xsl:value-of select="name(//*[@owner])"/>
</xsl:template>
```

first selects any element containing the attribute["owner"] and then returns the element name.

```
bookstore
```

The Next Step

And that just about concludes this little tour of XPath. In this article, you learnt a little about the history, need and rationale for XPath, together with an in−depth look at the various components of an XPath location reference. You also found out how to use XPath operators to build complex expressions, and XPath functions to perform simple processing tasks.

I hope you found this tutorial useful and informative, and that the material discussed here offers you some assistance in navigating the treacherous rapids of XML and XSL. You should now pop open your favourite XML document and begin playing with XPath to better understand how it works − the more you experiment with it, the easier it will become and, before long, you'll be constructing location paths without thinking twice.

In case you're looking for more information, you might like to check out the following links:

XML Basics, at http://www.devshed.com/Client_Side/XML/XMLBasic1/

XSL Basics, at http://www.devshed.com/Client_Side/XML/XSLBasics1/

The XPath specification, at http://www.w3.org/TR/xpath.html

General examples of XPath usage, at http://www.zvon.org/HTMLonly/XPathTutorial/General/examples.html

A list of useful XSL and XPath resources, at http://www.stars.com/Authoring/Languages/XSL/

Till next time....stay healthy

Note: All examples in this article have been tested on Microsoft Internet Explorer 5.5 and Saxon 6.4.3. Examples are illustrative only, and are not meant for a production environment. YMMV!