Working with XML in ColdFusion

ColdFusion MX is the most significant version release of ColdFusion to date. It’s significance centers around the addition of support for two key technologies, web services and XML, to the CFML programming language. In this paper we will examine the new support for XML in ColdFusion Mark-up Language.

In this paper we will:
- Take a general look at XML, including the ways in which ColdFusion represents XML
- Discuss the ways in which XML can be dynamically created in ColdFusion
- Discuss the ways in which XML can be parsed from and written to external source(s)
- Examine the ways in which XML data can be accessed and manipulated
- Learn how to search XML data in ColdFusion using XPath
- Learn how to transform XML from one “flavor” to another

XML – What is it? Why use it?

XML is the abbreviation for “extensible mark-up language” – a web initiative begun in the mid nineteen nineties. XML is based on the idea that text marked-up with tag syntax (similar to CFML and HTML) can be used to represent data and that this data can be passed between and manipulated within any application running on any environment, so long as that environment has the ability to parse the XML text and represent it in a native data format. DTD (document type definition) files are XML schema definitions used to enforce the rules of an XML “flavor”. An XML “flavor” is one XML language that two or more systems agree to “talk” using. There are hundreds of XML languages – you can create your own. That is what makes XML “extensible”… the ability to create your own “tag-based metadata language” (your own “flavor”).

XML is a very strict language – even though you can define your own mark-up language with it, there are rules that all XML languages must obey. First of all – all XML is based on tags… every tag in XML must have a closing tag. If a tag does not need to contain any content (does not need a closing tag) then it can close itself with the following syntax:

<mytag />

Note that the forward slash usually found at the beginning of a closing tag has been placed at the end of the opening tag preceded by a space. This is known as an “empty tag”. In XML, tag attributes must be enclosed in quotes. If an opening XML tag is uppercase, its closing tag must be uppercase (and vise-versa). XML attributes are also case-sensitive. XML is also very strict about tag nesting. All nested tags must be properly nested within each other. The following example is invalid in XML because the tags are not properly nested within each other:
Hello World!

Also, there are a few characters that are not allowed to be embedded in XML. In order to convert these “invalid” characters to their XML equivalent, use the CFML xmlFormat() function. The xmlFormat() function replaces all characters in a string that are not allowed in XML with a character representation that is. Characters commonly replaced are greater than and less than, single and double quotes, ampersands, etc. One other character to watch out for is a leading blank space or carriage return. XML Parsers are notorious for giving developers headaches by refusing to parse an XML string because of a hidden carriage return at the beginning of the XML packet.

<table>
<thead>
<tr>
<th>Ascii Character</th>
<th>Ascii Numeric Value</th>
<th>XMLFormat() representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>'</td>
<td>39</td>
<td>'</td>
</tr>
<tr>
<td>“</td>
<td>34</td>
<td>&quot;</td>
</tr>
<tr>
<td>&lt;</td>
<td>60</td>
<td>&lt;</td>
</tr>
<tr>
<td>&gt;</td>
<td>62</td>
<td>&amp;gt</td>
</tr>
<tr>
<td>&amp;</td>
<td>38</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

XML also requires that all XML documents have one base tag surrounding the entire packet. For example:

```xml
<employee fname="Simon" lname="Horwith" />
<employee fname="Dave" lname="Watts" />
<employee fname="Steve" lname="Drucker" />
```

is not a valid XML packet, but the following would be:

```xml
<figs>
<employee fname="Simon" lname="Horwith" />
<employee fname="Dave" lname="Watts" />
<employee fname="Steve" lname="Drucker" />
</figs>
```

XML is represented two ways. In its raw state, XML is a string, as we’ve seen in the examples so far. The fact that XML is a string makes it lightweight, easy to pass between servers via HTTP, and easy to persist to disk or database. Though XML is a string, in order to work with XML data within development environments, it is easier to represent the data as a complex data object. Every environment that has XML support uses software known as an XML parser to take an XML string and represent it as a complex data object. ColdFusion MX uses the Apache Xerxes parser internally. To you, the developer, it is fairly transparent as you will see. If a DTD exists, the parser validates that all of the rules for that language have been obeyed before parsing the XML. The data resultant of XML parsing is known as an XML DOM – a Document Object Model in
most environments. In ColdFusion (and in this paper) the term is used interchangeably with “ColdFusion XML Object”. The Document Object Model for an XML document is a tree-like data representation of the XML packet. When you look at an XML string you talk about “tags”. When you are working with an XML DOM object, these tags are commonly referred to as “nodes” or “elements” of the DOM. For example, every properly formed HTML page has a ‘body’ tag nested within the ‘html’ tag. If the HTML string were treated as XML and parsed, you would say that the ‘html’ element (you could also refer to it as the ‘XMLRoot’ since it surround all of the other html code) has a ‘body’ child node. In ColdFusion MX, XML object DOM is a new data type of sorts. There are many methods for working with XML, as well as a new tag for creating XML as well.

XML is good at describing data. Because you can define your own data language using XML, it is the ideal choice in how to describe data. XML allows this “described data” to be easily persisted in an RDBMS or text file, to be easily syndicated and shared with other servers and development environments over HTTP, to be searched for data-subsets using XPath, and to be easily transformed into another language or text format. In other words, it’s an extremely flexible, lightweight, portable solution for meeting data representation needs.

WDDX (Web Dynamic Data eXchange) was the first form of XML support in the CFML programming language (WDDX is one XML “flavor”). WDDX was created for and is good at describing raw data, but it is also very verbose – often resulting in huge XML packets. Not only that, but WDDX only describes raw data – the data holds no context. In other words, just from looking at a WDDX packet representation of an array, I have no way of knowing whether I’m looking at a shopping cart, employee listing, or a user’s personal preference settings. XML lets you be much more descriptive as well as more concise in describing data. WDDX is beyond the scope of this paper; for more information about WDDX, read the ColdFusion documentation for the <cfwddx> tag or visit http://www.openwddx.org.

**Dynamically Creating XML**

The question you may be asking yourself now is “Where does this XML come from?” XML comes from one of two places: it is either created dynamically with code or it comes from an external source. We will look at external sources in the next section… now let’s look at dynamically creating XML with code. One thing that does go without saying is that you could create an XML string programmatically using ColdFusion’s string manipulation functions, but we won’t waste any time talking about that in this unit. We’re interested in the new, more efficient functionality that comes with the server.

There is a new tag in ColdFusion MX called the <cfxml> tag. The <cfxml> tag gives developers a very easy to use way to create a ColdFusion XML object. The tag has two attributes: only one is required:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>Variable is the name of the ColdFusion variable that is created to hold the XML DOM. It is the required attribute.</td>
</tr>
</tbody>
</table>
The way a ColdFusion XML object is created with the `<cfxml>` tag is that an XML file (all of the tags and attributes for an XML string) is placed between the opening and closing `<cfxml>` tags. In addition to XML tags, CFML tags can be placed within the `<cfxml>` block in order to dynamically create XML tags. By way of the “variable” attribute, you specify the name of a variable that will be created whose value is the XML object. Note that only CFML tags will be evaluated – not CFML functions. If you want to use a CFML function such as an `XMLFormat()` within a `<cfxml>` block, be sure to call it within `<cfoutput>` tags.

The following code example would create an XML DOM (stored in the local “xmlEmps” variable) from a query recordset of employee information.

```cfml
<cfxml variable="xmlEmps">
<employees>
<cfoutput query="qEmployee">
<employee id="#qEmployee.empID#" fname="#xmlFormat(qEmployee.empFName)#" lname="#xmlFormat(qEmployee.empLName)#" />
</cfoutput>
</employees>
</cfxml>
```

The above code creates an XML packet with an “<employees>” root tag and one nested “<employee>” tag for each row that is returned from the query. Using the `<cfdump>` tag to look at the value of the “xmlEmps” variable created by the above code, would look like figure 1.
You’ll note from the image of diagram 1 that there are a lot of data values not represented in the code used to create the DOM but still present in the DOM. An XML DOM is a variable of the XML DOM type. An XML DOM can be detected with the new “isXMLDoc()” function, which returns a Boolean yes/no whether or not the variable passed is an XML DOM. Every valid XML DOM is essentially represented as a structure of structures. Of particular interest: each node in a DOM is a structure with an “XMLText” key that is the text between its opening and closing tags, an “XMLAttributes” key that is a structure of the attributes (name/value pairs) for this node, and an “XMLChildren” key that is an array of XML Nodes (the children of this node). The root node, which surrounds all other nodes, can be referred to by name or as “XMLRoot”. XML tag attribute names cannot begin with “xml” – as ColdFusion uses this internally to identify values.

In addition to the <cfxml> tag, there is another, slightly more complex, method of creating an XML DOM. We already discussed the fact that XML nodes are structures and that nodes contained within other nodes exist within their parent’s “XMLChildren” array. This, coupled with two new functions, is all you need to manually create XML. The first function required to manually create an XML DOM is the “XMLNew()” function. It has but one attribute – “caseSensitive (boolean).” “caseSensitive” is optional and defaults to “No”. The other method new to ColdFusion MX that allows XML to be created is the “XMLElemNew()” function. “XMLElemNew()” takes two arguments.
The first is the DOM variable in which this node is being created. The second argument is the name of the new element. The prior `<cfxml>` example could be rewritten using XML functions in `<cfscript>` like so:

```cfscript
//create object and initialize root element
xmlEmps = XMLNew();
xmlEmps.xmlRoot = XMLElemNew(xmlEmps, "employees");

//add first child element
xmlEmps.xmlRoot.xmlChildren[1].XMLAttributes.id = "1";
xmlEmps.xmlRoot.xmlChildren[1].XMLAttributes.fname = "Simon";
xmlEmps.xmlRoot.xmlChildren[1].XMLAttributes.lname = "Horwith";

//add second child element
xmlEmps.xmlRoot.xmlChildren[2].XMLAttributes.id = "2";
xmlEmps.xmlRoot.xmlChildren[2].XMLAttributes.fname = "Dave";
xmlEmps.xmlRoot.xmlChildren[2].XMLAttributes.lname = "Watts";

//add third child element using slightly different syntax to reference the root
xmlEmps.employees.xmlChildren[3].XMLAttributes.id = "3";
xmlEmps.employees.xmlChildren[3].XMLAttributes.fname = "Steve";
xmlEmps.employees.xmlChildren[3].XMLAttributes.lname = "Drucker";
</cfscript>

<!---:: output xml object ::--->
<cfdump var="#variables.xmlEmps#">

While the `<cfscript>` syntax may seem like much more verbose code and less simple to implement, we will discuss advantages to this approach in the “Manipulating and Accessing XML Data” section of this paper.

**Using XML with External Sources**

The strength of XML is not so much as a format to store persistent data in memory, as a portable data representation. By portable, we mean it is easy to share between applications, syndicate to other servers, and persist on disk in files and/or database(s). XML is the ultimate in portable and easily repurposed storage of data. In the last section you learned how to create an XML DOM with ColdFusion code. In this section we will examine creating an XML DOM from a pre-existing XML source, and how to save an XML DOM for later use.

When XML data is used in an application, the application generally works with the data in document object model (DOM) format. In order to save that XML DOM for later use by either writing it to file or storing it in a database, it must first be converted to an XML string (the tags and attributes that are the XML packet). This is achieved in a single line of code with the `toString()` function. The `toString()` function will accept an XML DOM object as it’s first argument and will return the string representation of the XML DOM.
toString also has an optional second argument that specifies the character encoding to use for the string (the default is the encoding of the current page). In order to persist the XML DOM, after converting it to a string with toString(), write that string to file with the <cffile> tag or to a database with a <cfquery> tag.

In order to create an XML object from an external source, you must first retrieve the XML string. Since an XML string is no different from any other string in CF, many methods are available for transferring XML data. The <cffile> tag can be used to retrieve the contents of an XML file from a file on the server’s local file system. The <cfinvoke> tag, <cfobject> tag, or createObject() function can be used to connect with a remote web service or local ColdFusion Component which may have methods that return XML. The <cfhttp> tag can be used to make an HTTP request for any web accessible file. A <cfquery> tag may be used to retrieve the XML from an RDBMS. There are more ways in which XML can be retrieved from an external source and brought into an application, but these are by far the most common.

Once an external source has responded to a page’s request, a variable exists on that page whose contents are an XML string. If you attempt to view this string by outputting the variable in a <coutput> block, many browsers will show you the data but not the actual tags themselves. In order to view an XML string in a web page, use the HTMLEditFormat() function. This will replace any necessary characters with their HTML display equivalent. For example, “<” becomes “&lt;”. In order to take an XML string and convert it into a ColdFusion XML DOM Object, use the XMLParse() function. XMLParse() has one required attribute, which is the XML string itself. A second optional argument specifies whether or not the XML Document and its attributes are case-sensitive. Let’s examine some examples:

If you were to save the resulting string from the earlier examples as “employees.xml” in the “C:\myXMLDocs” directory - an example of retrieving the XML from file and parsing it into an XML DOM would look like:

```cfcaption
<!----: create xml DOM from text file :---->
<cffile action="READ" file="C:\myXMLDocs\employees.xml" variable="myXMLString">
<cfset myXMLDOM = XMLParse(myXMLString)>
```

In order to create an XML DOM by requesting a URL that returns an XML string:

```cfcaption
<cfhttp url="http://www.mysite.com/xmlfiles/foo.xml" method="GET" resolveurl="No"></cfhttp>
<cfset myXMLDOM = XMLParse(cfhttp.fileContent)>
```

When the XML is retrieved from a web service, the code will look something like:

```cfcaption
<cfinvoke webservice="http://www.mydomain.com/data/xmlFactory" method="getXML" returnvariable="myXMLString">
<cfset myXMLDOM = XMLParse(myXMLString)>
```
This is an example of retrieving the XML from a database:

```cfc
<cfquery name="qGetXML" datasource="myDSN">
    SELECT myXMLCol FROM someTable
</cfquery>
<cfset myXMLDOM = XMLParse(qGetXML)>
```

When retrieving XML from another environment rather than creating it in the current page programmatically, care must be taken to assure success. Earlier it was mentioned that XML Parsers have difficulty parsing text whose first character is a space or carriage return (linefeed). Therefore it is crucial that before passing the XML string, any leading spaces or carriage return(s) be stripped away before the XMLParse() function is called.

Because these methods of retrieving an XML sting require leveraging external resources and because accessing external resources is likely the most error prone of all of the things you can do in a ColdFusion template, be sure to implement some sort of error handling strategy when attempting to retrieve the XML string. There is also no guarantee that the string returned is going to be in proper XML notation – remember, XML is a strict language. Attempting to XMLParse() a string that does not meet the format requirements of XML such as having a base tag will result in error. For this reason, unless you are sure of your source, implement error handling when parsing a string into an XML DOM object (see my paper on exception handling at http://www.how2cf.com/files/papers/exceptions.doc for more on structured exception handling).

Putting it all together, here is an example of retrieving an XML packet from a text file, verifying that it doesn’t have any carriage returns in the beginning, parsing it into an XML DOM and dumping the DOM, and writing the DOM back to the text file as a string.

```cfc
<cfscript>
    thisDir = expandPath('.
    targetXMLFile = variables.thisDir & "\mydata.xml"
</cfscript>
<cfif not fileExists(variables.targetXMLFile)>
    <h3>Can't find target xml file</h3>
    <cfabort>
</cfif>
<cftry>
    <cffile action="read" file="#variables.targetXMLFile#" variable="originalXMLString">
    <cfcatch>
        <h3>error opening XML file!</h3>
        <cfabort>
    </cfcatch>
</cftry>
```
Manipulating and Accessing XML Data

We have already discussed why XML is sometimes an ideal format to store data in, but how do you actually use this XML in a web application? Similar to data being stored in other complex variables such as arrays or structures, an application may need to output values that are contained within the XML DOM. Sometimes data needs to be added to or deleted from an XML DOM, or existing data needs to be changed. As mentioned before, XML DOM nodes are represented as structures, their attributes are stored in the node “XMLAttributes” key, the text contained within the opening and closing tag is stored in the node “XMLText” key, and if this node has children (nested tags), they are contained within its “XMLChildren” array as structures, themselves. Not only can nodes be referenced as “parentNode.XMLChildren[indexPosition]” where “indexPosition” is the nested tag depth from the parent tag, but also as “parentNode.xmlElementName[indexPosition]” where “indexPosition” is the nested tag depth from the parent tag among all nodes with the same name. What this means is that array and structure syntax and the built-in CFML functions for manipulating these two types of data, can be used to add/edit/delete XML data. The best way to understand this is to look at examples.

Our first example shows how to add a new attribute to DOM node(s). We will start with the same XML DOM shown in the earlier example… the xml file looks like the following:
Suppose that we want to add an “active” attribute to each “employee” tag in this simple XML packet. The code would look like:

<cfscript>
// add an 'active' attribute to all XML Children off of the DOM root
for (i = 1; i LTE arrayLen(variables.myXMLDOM.XMLRoot.XMLChildren); i =
    i + 1){
    structInsert(variables.myXMLDOM.XMLRoot.XMLChildren[i].XMLAttributes,"active",1,1);
}
</cfscript>

The above code loops over the array containing all children of the XMLRoot (XMLChildren) and inserts an “active” key into each child’s “XMLAttributes” structure.

If we wanted to add a new <employee> node to the DOM, the syntax would look something like the following:

<cfscript>
    // get array position of next XML child to add
    newNodePos = arrayLen(variables.myXMLDOM.XMLRoot.XMLChildren) + 1;
    // add new node then set it's attributes
    variables.myXMLDOM.XMLRoot.XMLChildren[variables.newNodePos] =
        XMLElemNew(myXMLDOM,"employee");
    structInsert(variables.myXMLDOM.XMLRoot.XMLChildren[variables.newNodePos].XMLAttributes,"fname","Dianna",1);
    structInsert(variables.myXMLDOM.XMLRoot.XMLChildren[variables.newNodePos].XMLAttributes,"lname","Courchesne",1);
    structInsert(variables.myXMLDOM.XMLRoot.XMLChildren[variables.newNodePos].XMLAttributes,"active",1,1);
</cfscript>

The preceding code example determines the array index position in the XMLRoot’s XMLChildren array for the new node to add. It then sets that array position equal to a new XML node with the XMLElemNew() function, and adds attributes to it’s XMLAttributes structure.

In order to modify existing data in an XML DOM, simply use the same array/structure syntax to access the node or attribute you wish to modify, and assign a new value. The
following example changes the ‘lname’ attribute of the first node from “Horwith” to “Badhwar”.

```cfc
<cfscript>
  //set the first employee's last name to 'Badhwar'
  variables.myXMLDOM.XMLRoot.XMLChildren[1].XMLAttributes.lname = "Badhwar";
</cfscript>
```

Just as array and structure functions can be used to add data, so can they be used to remove data. Supposing that the fourth “employee” node needs to be removed from the XML DOM. The following example would remove the node:

```cfc
<cfscript>
  //remove the fourth employee
  arrayDeleteAt(variables.myXMLDOM.XMLRoot.XMLChildren, 4);
</cfscript>
```

An example of removing the “active” attribute from all “employee” nodes would look like the following:

```cfc
<cfscript>
  // remove 'active' attribute from all XML Children off of the DOM root
  for (i = 1; i LTE arrayLen(variables.myXMLDOM.XMLRoot.XMLChildren); i = i + 1){
    structDelete(variables.myXMLDOM.XMLRoot.XMLChildren[i].XMLAttributes, "active",0);
  }
</cfscript>
```

Alternatively, structure “dot syntax” can be combined with array notation to access the nodes nested within a DOM. The following is an example of using alternative syntax to set the “active” attribute of the second “employee” node to 0:

```cfc
<cfscript>
  // use structure syntax to set the "active" attribute of the second node to "0"
  variables.myXMLDOM.figs.employee[2].XMLAttributes.active = 0;
</cfscript>
```

Not specifying the node array position that is to be manipulated results in the first node being accessed. Here we see an example of using this shorthand syntax to set the “lname” attribute of the first “employee” node to “Badhwar”:

```cfc
<cfscript>
  // use structure syntax to set the "Lname" attribute of the first node to "Badhwar"
  variables.myXMLDOM.figs.employee.XMLAttributes.lname = "Badhwar";
</cfscript>
```
In addition to the properties of an XML DOM and its nodes that we’ve been using here, there are many other pieces of information that can be accessed within an XML DOM. The following table shows these:

<table>
<thead>
<tr>
<th>XML DOM Data Members</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Member</strong></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td><strong>XML Root Keys</strong></td>
</tr>
<tr>
<td>XMLComment</td>
</tr>
<tr>
<td>XMLRoot</td>
</tr>
<tr>
<td><strong>Element Keys</strong></td>
</tr>
<tr>
<td>XMLName</td>
</tr>
<tr>
<td>XMLNSPrefix</td>
</tr>
<tr>
<td>XMLNSURI</td>
</tr>
<tr>
<td>XMLText</td>
</tr>
<tr>
<td>XMLComment</td>
</tr>
<tr>
<td>XMLAttributes</td>
</tr>
<tr>
<td>XMLChildren</td>
</tr>
<tr>
<td>XMLParent</td>
</tr>
<tr>
<td>XMLNodes</td>
</tr>
</tbody>
</table>

Of the data members shown in the previous table, all but “XMLParent” and “XMLNodes” will be displayed by passing an XML DOM to the `<cfdump>` tag. When a DOM is “dumped”, `<cfdump>` gives you the option of viewing the DOM in long mode or short mode. The current mode being displayed is at the top of the `<cfdump>` table. Click on this label to switch between modes. Some of the data members only display in “long version”, so get in the habit of viewing XML DOMs in “long version” mode if you plan to work with all of an XML DOMs data members. XML is capable of very complex operations and data representations. As such, you may want to pick-up a book on XML such as Practical XML for the Usable Web (ISBN: 1904151086), or try visiting http://w3c.org to learn more about what some of these data members represent and how they can and should be used. The complexities of XML are beyond the scope of this paper.

In addition to the functions already discussed, ColdFusion MX also has functions for determining whether a node is the root node, whether a variable is an XML node (an XML element to be exact), to extract a specific element based on name and order within the XML children of that name, etc. These functions are shown in the following table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isXMLDoc()</td>
<td>accepts object name and returns boolean to indicate whether or not it is a properly formed XML DOM</td>
</tr>
<tr>
<td>isXMLElem()</td>
<td>accepts element name and returns boolean to indicate whether or not it is a DOM element</td>
</tr>
<tr>
<td>isXMLRoot()</td>
<td>accepts element name and returns boolean to indicate whether</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
</tr>
<tr>
<td>toString()</td>
<td>Will convert an XML DOM to a string – thus allowing the XML to be output or written to disk</td>
</tr>
<tr>
<td>XMLParse()</td>
<td>Converts a string to a DOM and accepts two arguments: An XML string to create the DOM from (required) Casesensitive – optional (boolean) to indicate case sensitivity (default is false)</td>
</tr>
<tr>
<td>XMLChildPos()</td>
<td>Retrieves the ‘n’th element of a specific name - accepts three arguments: The element to retrieve the child node from The name of the element to retrieve The element child position to return</td>
</tr>
</tbody>
</table>

Most of the functions are fairly straightforward, save for “XMLChildPos()”. The XMLChildPos() function is useful for determining where in the array of all XMLChildren a node sits, based on the name of the node as well as which occurrence in the sequence of nodes of that name. When all nodes have the same name as in our example, this isn’t very useful. What if the elements within a DOM do not all have the same name? For example, the “figs” XML packet may contain “employee” and “contractor” nodes to represent employees and contractors. The XML packet may look like:

```xml
<figs>
  <employee fname="Simon" lname="Horwith" />
  <employee fname="Dave" lname="Watts" />
  <contractor fname="Charles" lname="Arehart" />
  <contractor fname="Branden" lname="Hall" />
  <employee fname="Steve" lname="Drucker" />
</figs>
```

In order to determine the position in the XMLChildren array of the third “employee”, you first have to determine how many “employee” nodes there are (you don’t want to attempt to determine the XMLChildren position for the third “employee” if there are only two), and then use the XMLChildPos() function to determine its position. The code to do this is:

```cfscript
// find out how many employee elements there are
numContractors = arrayLen(variables.myXMLDOM.figs.employee);
// if there are at least 3 employees, find out where the third employee is in the child elements array
thirdEmployeePos = XMLChildPos(variables.myXMLDOM.figs,"employee",3);
thirdEmployeeNode = variables.myXMLDOM.XMLroot.XMLChildren[variables.thirdEmployeePos];
</cfscript>
Though array and structure functions coupled with the “XMLChildPos()” function does help in finding and extracting the data you want, sometimes these solutions don’t perform efficiently enough or at all, to meet the needs of an application. XPath addresses many of these limitations.

**Searching for and Extracting XML Data with XPath**

Often in the real world, a simple XML schema with one DOM level of tags, all of which with the same name, is not capable of meeting the needs of an application. This is particularly true of applications storing data with complex relationships with other data. Sometimes it is necessary to nest tags within tags… sometimes the XML schema definition is already made for us. XPath can help to quickly retrieve data nested at various levels in a DOM hierarchy. Another common issue when working with any XML, particularly large XML packets, is the performance limitation of looping over every node in an XML DOM and performing conditional logic to validate whether this is one of the elements you are looking for. XPath helps tremendously with this type of functionality.

XPath is the language of representing XML Document Object hierarchies with expressions. You may already be familiar with using regular expressions for text pattern matching in CFML, JavaScript, or some other programming language. Think of XPath as regular expressions for XML hierarchies of data. XPath may syntactically look more like regex to many folks, but it’s actually more similar to SQL in terms of functionality. Like SQL, the sole purpose of XPath is to search through a large amount of data and return whatever data it finds to meet your criteria. Unlike SQL, XPath is only used for searching and not for inserting and updating data, however, XPath can be used with other XML tools to update or transform data. In essence, it is a tool for searching XML.

Just like HTML, CSS, and XML, XPath is an official specification of the World Wide Web Consortium (http://w3c.org/). As has already been mentioned, XPath is a tool for searching XML using expressions. More than that, XPath is a tool for mapping XML nodes. You can use XPath to map the nodes of one XML packet to the nodes of another when performing transformations.(transformations are covered in the next section of this paper). XPath is not only good at defining elements (branches in the hierarchy) but is also good at mapping node text values and attribute values.

XPath is implemented in ColdFusion MX pages with the XMLSearch() function. XMLSearch() accepts two arguments: the first is the DOM Object to search and the second is the XPath expression that you wish to apply to the XML DOM. In order to use XMLSearch(), you must be somewhat familiar with XPath. There are a few rules that will help you in creating XPath expressions:

- Single slashes separate nested elements in a DOM—similar to how slashes in a URL represent nested folders
- Double slashes are a wildcard: they represent “any parent element – any number of nested levels”
• XPath comparison statements are always enclosed within square brackets
• Any non-numeric attribute or XMLText values to match are wrapped in single quotes
• Precede attribute names with an ‘@’ symbol
• XPath searches are case-sensitive

If you follow the basic rules above, and experiment a little bit, you can get up and running with XPath in very little time. In order to practice with XPath, we’ve made the XML schema a little bit more complex. The schema now contains both “<employee>” and “<contractor>” tags just beneath the root level. Nested below that level are “<active>” elements which contain the text 1 or 0 between the two tags (XMLText). The sample XML looks like this:

```xml
<figs>
  <employee id="1" fname="Simon" lname="Horwith">
    <active>1</active>
  </employee>
  <employee id="2" fname="Dave" lname="Watts">
    <active>1</active>
  </employee>
  <contractor id="1" fname="Charles" lname="Arehart">
    <active>1</active>
  </contractor>
  <contractor id="2" fname="Branden" lname="Hall">
    <active>1</active>
  </contractor>
  <employee id="3" fname="Steve" lname="Drucker">
    <active>1</active>
  </employee>
</figs>
```

Using the XML packet shown above, if you wanted to retrieve all of the “<contractor>” tags directly off of the “figs” root. It would look like this:

```cfscript
//get all contractors off of the 'figs' root element.
aAllContractors = XMLSearch(myXMLDOM,"figs/contractor");
</cfscript>
```

If, on the other hand, you wanted all “<contractor>” tags regardless of where they are in the hierarchy (no matter how many levels deep they are.) the code would look like:

```cfscript
//get all contractors off of the 'figs' root element.
aAllContractors = XMLSearch(myXMLDOM,"//contractor");
</cfscript>
```
The syntax to retrieve all employee nodes found immediately off of the 'figs' root element that have a nested "<active>" node with the value 0 between it’s opening and closing "<active>" tags

<cfscript>
    //get all employees off of the 'figs' root element that have a nested "<active>" node with the value 0 between the opening and closing "<active>".
    aActiveEmployees = XMLSearch(myXMLDOM,"/figs/employee[active=0]");
</cfscript>

An example of retrieving the “<employee>” that has an “lname” of “Horwith” follows:

<cfscript>
    //get all employees that have an lname of "Horwith"
    aHorwiths = XMLSearch(myXMLDOM,"//employee[@lname='Horwith']");
</cfscript>

This is a sample of how to find all nodes that have an “fname” of “Dave” AND that have their nested “<active>” tag set to 1.

<cfscript>
    //find all nodes that have an “fname” of “Dave” AND that have their nested “<active>” tag set to 1
    aActiveDave = XMLSearch(myXMLDOM,"//employee[@fname='Dave' and active=1]");
</cfscript>

This last sample shows how to find all nodes that have an “fname” of “Simon” OR that have their nested “<active>” tag set to “0”

<cfscript>
    //get all employees that have an fname of "Simon" OR that have a nested "<active>" child
    aSimonOrInactive = XMLSearch(myXMLDOM,"//employee[@fname='Simon' or active=0]");
</cfscript>

While the example XML packet is not the most complex in the world, the sample syntax and rules laid out in this paper can be applied to XML of any size in order to search through the XML data. It is important to note that XPath generally executes very fast – much faster than Query of Queries in tests. This makes it an excellent alternative to query of queries, when the desired data is easily represented as XPath expression. When multiple data sources (xml files or queries) need to be cross-referenced with each other (similar to a table join in SQL) or wildcard functionality like the SQL “LIKE” keyword needs to be implemented, XPath is not usually going to be the recommended solution.
For retrieving data from a single xml source based on the values of tags and their attributes, XPath is often the best answer.

One last thing to discuss about XPath is its return value. You may have noticed that all of the XPath samples set the results to a variable that begun with a lowercase “a”. This is not required, but was done for naming convention reasons - XPath returns the data it finds as an array of elements. What this means is that you will have to get that result set into proper XML DOM format in order to perform operations like XMLSearch(), toString(), etc. Unfortunately, you will have to loop over the array result elements and manually copy their data members into another DOM node. Attempting to insert the actual XPath results array into a second DOM will not work. ColdFusion does not allow elements from one DOM to exist in another, which makes sense when you take into account the “XMLParent” pointer that every element contains. Unfortunately, the “XMLParent” cannot be deleted from an element, either.

**Transforming XML with XSLT**

We have seen how to create, access, and manipulate the data in an XML DOM. We have also examined the syntax used to search an XML DOM for nodes that match search parameters using XPath expressions. One last thing we need to examine is transforming XML from one XML version to another, or (more useful) from XML to HTML for display as a table or other HTML content. Like XPath and XML in general, there are entire books devoted to XSLT. If you need to heavily leverage XSLT in your applications, we recommend picking up a book at your local bookstore or visiting the w3c at http://w3c.org.

XSLT stands for “eXtensible Stylesheet Language Transformation”. It is the process of combining XML with a stylesheet written to transform that data into some other format. The result of the combining of the two is that the XML becomes “transformed” into some other text. The stylesheet used to perform the transformation generally is stored in a file and given a “.xsl” extension. You will have to open this file programmatically using <cffile>. If the XSL file is web accessible it can be retrieved using <cfhttp>, and it’s contents can also come from a database or from any variable who’s value meets the requirements for extensible stylesheet content.

Extensible Style Sheets use a lightweight scripted programming language to define conditional logic, loops, variable declaration and output, data sorting, and other simple common tasks required to take XML content and redefine its structure. Implementing XSLT in ColdFusion MX is a trivial task – it’s writing the stylesheet that most developers find challenging.

In order to implement XSLT in ColdFusion pages, use the “XMLTransform()” function. XMLTransform() accepts two arguments, both of which are required. The first argument is an XML DOM variable, and the second is an XSL stylesheet – not a URL but the actual stylesheet contents.
Before we can implement a stylesheet, we have to examine the details of writing extensible stylesheets. The first line of a stylesheet looks the same as the first line of most XML packets: “<?xml version="1.0"?>” – this identifies the remaining text in the page as an XML packet. This first line, like in other XML files, requires no closing tag. The second tag in an XSL document is: “<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
xmlns="http://www.w3.org/TR/REC-html40"
xmlns:xlink="http://www.w3.org/1999/xlink">”. This identifies the XML contents as an XSL stylesheet and also identifies the URLs for linking, transformation, and namespace specifications. This tag and all proceeding tags will have end tags. The next tag (with accompanying closing tag) in an XSL stylesheet is the “<xsl:template match="/”” tag. This tag tells the browser at what point in the XML packet used for the transformation to start XPathing – with the “match” attribute. A values of “/” as shown, specifies that all XPath mappings in this document begin at the document root. Everything between “<xsl:stylesheet><xsl:template>” and “</xsl:stylesheet></xsl:template>” comprises of the actual text that makes up the transformation output, and other “<xsl:…”” tags which represent stylesheet instructions. All extensible stylesheet commands are tags with the “xsl:” namespace. The complete set of allowed XSL commands are:

<table>
<thead>
<tr>
<th>XSLT Tag</th>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xsl:apply-templates</td>
<td>Select (xpath expression)</td>
<td>Calls one template from another. Attribute determines which node is acted upon</td>
</tr>
<tr>
<td>xsl:choose</td>
<td>None</td>
<td>Choose one of many possible options – equivalent to a switch statement</td>
</tr>
<tr>
<td>xsl:for-each</td>
<td>Select (xpath expression)</td>
<td>For looping through the set of elements specified by the attribute</td>
</tr>
<tr>
<td>xsl:if</td>
<td>Test (boolean expression)</td>
<td>For “if” logic</td>
</tr>
<tr>
<td>xsl:otherwise</td>
<td>None</td>
<td>Executed when no xsl:when statement is executed. Similar to the default in a switch-case statement</td>
</tr>
<tr>
<td>xsl:sort</td>
<td>Order (“ascending” or “descending”)</td>
<td>Specifies sort order for xsl:apply-templates and xsl:for-each statements</td>
</tr>
<tr>
<td>xsl:stylesheet</td>
<td>Version (“1.0” – optional)</td>
<td>Root element for the stylesheet</td>
</tr>
<tr>
<td>xsl:template</td>
<td>Match (xpath expression)</td>
<td>Specifies set of xslt tags to execute as a single unit. Applies to node specified by the attribute</td>
</tr>
<tr>
<td>xsl:value-of</td>
<td>Select (xpath expression)</td>
<td>Generates a text string with the value of the attribute. Similar to cfoutput</td>
</tr>
<tr>
<td>xsl:variable</td>
<td>Name (value)</td>
<td>Defines a variable and it’s value. Once a variable is set in XSLT, it’s value cannot be changed</td>
</tr>
</tbody>
</table>
| xsl:when        | Test (boolean expression) | Represents on option in an xsl:choose block. Similar to a
One thing worth noting is that unlike traditional programming languages, once an XSL variable is created with the “<xsl:variable>” command, its value cannot be changed. Let’s put this all together in a simple example where our previous XML packet is transformed into an HTML table of active employees.

First, let’s look at the code that creates the XML DOM, retrieves the XSL stylesheet, and performs the actual transformation:

<!--:: create XML DOM ::--->
<cfxml variable="myXMLDOM">
<figs>
<employee id="1" fname="Simon" lname="Horwith">
  <active>1</active>
</employee>
<employee id="2" fname="Dave" lname="Watts">
  <active>1</active>
</employee>
<contractor id="1" fname="Charles" lname="Arehart">
  <active>1</active>
</contractor>
<contractor id="2" fname="Branden" lname="Hall">
  <active>1</active>
</contractor>
<employee id="3" fname="Steve" lname="Drucker">
  <active>1</active>
</employee>
<employee id="4" fname="Dave" lname="Gallerizzo">
  <active>0</active>
</employee>
</figs>
</cfxml>
<cffile action="read" variable="myXSL" file="#expandPath('.')#igsXSL.xsl">
<cfset transformedXML = XMLTransform(variables.myXMLDOM, variables.myXSL)>
<cfoutput>#variables.transformedXML#</cfoutput>

You will note that the above code creates an XML DOM with two nested active “<employee>”s, then two active “<contractor>”s, then another active “<employee>” and an inactive “<employee>”. All we want to return is an HTML table with active “<employee>”s in it.

The “figsXSL.xsl” XSL stylesheet contents look like:
The resulting output when browsing the ColdFusion page containing the XML DOM declaration, `<cffile>` tag that opens the XSL document, and the code that performs the transformation and outputs the results is shown in figure 2.

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simon</td>
<td>Horwith</td>
</tr>
<tr>
<td>Dave</td>
<td>Watts</td>
</tr>
<tr>
<td>Steve</td>
<td>Drucker</td>
</tr>
</tbody>
</table>

**Figure 2 – Results of transforming the “figs” XML**

Stylesheets are a very powerful tool for transforming XML from one “flavor” into another. You can use it to transform XML into HTML, XHTML, another XML flavor,
CSV, or any other text format imaginable. Stylesheets allow for looping, conditional logic, variable declarations, etc. In order to implement more complex stylesheets, you may need to study and practice.

Summary

The `<cfxml>` tag makes it very easy to create an XML object without having to really understand the intricacies of XML and the new XML functions, though they do require a little more knowledge of XML, allows a developer to manipulate existing XML objects and to create them using an alternative syntax. The XMLSearch() function adds full support for the retrieval of data using XPath, and the XMLTransform() function makes it very easy to programmatically apply styles in order to transform one version of XML to another.

The new XML functionality in ColdFusion MX allows for the easy creation, manipulation, transformation, and filtering of XML data. Developers are only limited by their lack of knowledge about XPath, XSL, and XML – so play around with it and have fun. When you begin to figure out and get comfortable with the basics, pick up an intermediate level book – one that explains not only XML but also XPath and XSL, and enjoy! Once you begin down a path of using XML to represent application data, you may very well never want to store data in any other format again!