

Level: Advanced
Works with: Notes/Domino 6
Updated: 03-Mar-2002

by
Sally Blanning DeJean
and [David DeJean](#)

In the previous *LDD Today* article, "[LotusScript: XML classes in Notes/Domino 6](#)," we introduced you to the new LotusScript XML classes and to the DOM and SAX parsers for exporting Domino data to DXL (Domino XML language). This article continues that discussion by introducing you to the XSL transformer for converting DXL to other markup languages, particularly HTML, and using the DXL importer to convert XML to NSF format. This article and the previous article are part of a series that investigates the changes to the LotusScript programming language in Notes/Domino 6.

The code samples referenced in this article are available for download from the [Sandbox](#). Before you use the sample databases, create a local dxl directory on your C: drive.

Using the XSL transformer

XSL (for "XML Stylesheet Language") is one way to control the reformatting, or transformation, of an XML file into a different format or another markup language. If you've worked with Cascading Style Sheets, XSL will look familiar. The way it works is similar as well. In fact, you can use CSS stylesheets to transform XML files into HTML for display in a Web browser.

The need to transform data is everywhere. For example, your company may keep its CRM (customer relationship management) data in a Domino database that you maintain. When your company forms a business partnership with another company, it must share part of its customer information. The partner organization wants contact information, but not historical data, and only for part of the country. And they have their own CRM system. How would you get the data to them? If the need were internal and you had full knowledge of both systems, you might use Lotus Enterprise Integrator (LEI). But the partner is external, and you only know your end of the transaction. In this case, XML is the answer, and an XSL transform is the key.

The Export Only Data—XSLT agent

In the previous article, we introduced our Hello World sample database that included agents that exported Domino data to DXL. Our DXL Hello World data isn't as complicated as a contact record; it's only one field with a default value of Hello World. But the principles of transformation are the same. An agent named 5. Export Only Data—XSLT uses an XSL stylesheet named dxlhelloworld_data.xsl to control the reformatting of the DXL Hello World database into an XML file that contains only a minimal amount of data. You can look at this agent's code in the sample database. If you want to run it, you'll have to copy the stylesheet out of the database (it's saved in the Files section under Shared Resources) and save it in the c:\dxl directory you created earlier.

Here are the relevant sections of the agent. After the initial set-up, it creates two stream objects (one to represent the stylesheet and one to represent the XML output file) and does some checking to make sure the files and pathnames are all correct:

```
Dim XSL_ss As NotesStream ' stylesheet
```

```
Set XSL_ss=session.CreateStream
If Not XSL_ss.Open(pathname$ & filename$ & ".xsl") Then
    Messagebox "Cannot open " & filename$, "XSL file error"
Exit Sub
End If

Dim XML_out As NotesStream ' output file
Set XML_out=session.CreateStream
If Not XML_out.Open(pathname$ & filename$ & ".xml") Then
    Messagebox "Cannot create " & filename$, "TXT file error"
Exit Sub
End If
XML_out.Truncate
```

The database is first slimmed down by the same NotesNoteCollection code used in the previous article to limit the exported data just to documents:

```
Dim nc As NotesNoteCollection
Set nc = db.CreateNoteCollection(False)
nc.SelectDocuments=True
Call nc.BuildCollection
```

The NotesDXLExporter object is created and its input is specified nc for the NotesNoteCollection. Notice that no output is specified:

```
Dim exporter As NotesDXLExporter
Set exporter = session.CreateDXLExporter(nc)
```

The NotesXSLTransformer object is created with three arguments: an input (exporter), the stylesheet (XSL_ss), and an output file (XML_out). Specifying exporter, the NotesDXLExporter object, as the input for the XSL transformer sets up the pipeline between the two objects:

```
Dim transformer As NotesXSLTransformer
Set transformer=session.CreateXSLTransformer(exporter, XSL_ss, XML_out)
```

When you run the agent, it transforms the DXL file using the SXL_ss stylesheet. The result is a minimal representation of the Domino data:

```
<?xml version="1.0" encoding="UTF-8"?>
  <database xmlns:dxl="http://www.lotus.com/dxl" numberofdocuments="1" dbid="85256C7500771804"
    replicaid="85256C7500771804" path="dxlhelloworld.nsf" title="DXL Hello World">
    <document unid="2650244E74784BD985256C85004F5EDA" form="Hello">
      <item name="HelloData">Hello World.</item>
    </document>
  </database>
```

When you compare this output to the file created by the agents in the previous article, you'll see that not only have elements been dropped, but attributes have been moved between elements. The numberofdocuments and dbid attributes of the database item, for example, used to be in the <databaseinfo> element. The unid attribute of the <document> element has migrated from its child element named <noteinfo>. All of the <datetime> elements for adds and updates have disappeared.

The XSL stylesheet

If you watch the LotusScript debugger when you run the Export Only Data agent, you won't see any of these things happen. When it gets down to the line exporter.Process, the debugger might as well put up a Messagebox that says, "Magic happens here."

What actually happens is that the NotesXMLProcessor class's XSL Transformer engine applies the stylesheet to the DXL pipelined from the exporter. An in-depth tutorial on XSL stylesheets is not what this article is about, but the Hello World stylesheet will give you some code you can use as a starting point, and it makes clear some of the basics of stylesheet writing.

The basics start with the declarations—an XML declaration, an XSL declaration, and an output declaration:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
    xmlns:dxl="http://www.lotus.com/dxl">
    <xsl:output method="xml" indent="yes"/>
```

The most interesting of these is the XSL declaration, which specifies two XML namespaces—one for the XSL objects used in the transformation and one for the DXL objects. The namespace for DXL looks like a URL, <http://www.lotus.com/dxl>, but it isn't. It looks like a URI, but in fact, it's not anything at all. If you enter it in a browser, you get back an error 404-File not found. The XSL namespace URI does a little better: It points you to a couple of W3C Web documents about XML and XSL namespaces.

A namespace is something like a DTD. It's an optional (and usually hypothetical) document that is where you would define exactly what meanings you apply to the names you give to the elements and attributes in your XML. The most important practical benefit of namespaces is that stylesheets use namespace abbreviations as identifiers. In the Hello World stylesheet, the preface *xsl:* tells the transformer that it owns the item with this name, which may actually be a processing instruction, while the preface *dxl:* tells the transformer it should look for the following name in the DXL file.

Stylesheets are written in a syntax that includes very few verbs in the code. The verbs that do appear (such as *match* and *select*) make a stylesheet look more like a database search than a programming language. In fact, XSL works more like the *Find* function in a word processor or a Web browser. You tell it what to look for in the file. If it finds it, it sets the current context there. The element `<xsl:apply-templates select="dxl:database"/>` might be translated into English as, "Find an element in the DXL file named 'database'." The action it should take in that location is specified by a template: The element `<xsl:template match="dxl:database">` in English would be, "When you locate an element named 'database,' perform the following actions."

Some templates are built into the transformer. The stylesheet begins execution at the root element, for example, so that the first instruction in the stylesheet, `<xsl:template match="/">`, is not a *Find* instruction, but the start of a templated action. (The element you may expect to start a stylesheet with, `<xsl:apply-templates match="/">`, is likely to invoke some of the built-in templates and produce unexpected results in the output.)

As you write XSL templates, it may help to think of the stylesheet in terms of a sort of chess game. The *select* elements move you across the board to a new location. The *match* elements start processing actions in the context of this new location. The first *select/match* pair in the stylesheet shows this:

```
<xsl:template match="/">
  <xsl:apply-templates select="dxl:database"/>
</xsl:template>

<xsl:template match="dxl:database">
  <xsl:element name="database">
    <xsl:apply-templates select="dxl:databaseinfo"/>
    <xsl:attribute name="replicaid"><xsl:value-of select="@replicaid"/></xsl:attribute>
    <xsl:attribute name="path"><xsl:value-of select="@path"/></xsl:attribute>
    <xsl:attribute name="title"><xsl:value-of select="@title"/></xsl:attribute>
    <xsl:apply-templates select="dxl:document"/>
  </xsl:element>
</xsl:template>
```

The *select* takes you from the root element to the first element in the DXL data named *database* (in fact, they're probably the same element, but that's OK). The *match* element is actually an *if*: "If a match was made, apply this template to do the following things." The first action, `<xsl:element name="database">`, creates a new element in the output. The next action is another *select* statement that moves you to another square on the board: The context switches from the *database* element to the first element in the DXL data named *databaseinfo*. Because the search starts within the context of the element named *database*, its scope is all the child elements of the *database* element. It finds an element named *databaseinfo*, moves the context there, and because a match has been made, it executes another template:

```
<xsl:template match="dxl:databaseinfo">
  <xsl:attribute name="numberofdocuments"><xsl:value-of select="@numberofdocuments"/></xsl:attribute>
  <xsl:attribute name="dbid"><xsl:value-of select="@dbid"/></xsl:attribute>
```

```
</xsl:template>
```

The first instruction again creates a new element in the output—this one an attribute named `numberofdocuments`. The next element, `<xsl:value-of select="@numberofdocuments"/>`, looks for a value in an attribute (signified by the `@` symbol) of the current element, `databaseinfo`, and writes what it finds to the output, then follows the next instruction, `</xsl:attribute>` to do just what you might expect: It ends the attribute. (It's important to remember that the value-of statement always returns a child of the selected element, and the text of the value of any element is considered by XSL to be a child element.) The final instruction creates and populates a `dbid` attribute, and then the template terminates.

Just as with a LotusScript subroutine, execution returns to the point in the stylesheet where the template was called. In our chessboard analogy, we are returned to the square where we landed when we searched for items named `database`. If there were more child elements of `database` named `databaseinfo` the template would be applied against them as well, but there are none, so the context remains set to the `database` element and the next element in the template that matches to `database` is executed. Three more attributes are written to the output—`replicaid`, `path`, and `title`. Then the template calls another template to execute against an element named `document`:

```
<xsl:template match="dxl:document">
<xsl:element name="document">
<xsl:apply-templates select="dxl:noteinfo"/>
<xsl:attribute name="form"><xsl:value-of select="@form"/></xsl:attribute>
<xsl:apply-templates select="dxl:item"/>
</xsl:element>
</xsl:template>
```

This template creates a new element in the output also named `document`, then looks for a child element named `noteinfo`. Another template executes when `document` is found to have a child named `noteinfo`:

```
<xsl:template match="dxl:noteinfo">
<xsl:attribute name="unid"><xsl:value-of select="@unid"/></xsl:attribute>
</xsl:template>
```

This template writes the `unid` of the document to the output as an attribute—not of `noteinfo`, but of `document`. When no other `noteinfo` fields are found, context passes back to `document`, and the `form` name is written as an attribute to the output. Then the template searches for children named `item`—the data fields in the Notes document:

```
<xsl:template match="dxl:item">
<xsl:element name="item">
<xsl:attribute name="name"><xsl:value-of select="@name"/></xsl:attribute>
<xsl:value-of select="."/>
</xsl:element>
</xsl:template>
```

The template matches each element named `item` and creates a new item in the output called *item* with an attribute called *name* that holds the name of the item (in this case `HelloData`, the field name assigned in Notes) and copies the item's value to the output using the instruction `<xsl:value-of select="."/>`. The `"."` is a contextual referent that means this or here. The value-of element returns only the value of the first instance of a match, so this element works because `item` has only one child element—the value `"Hello World."`

With that, the `item` template is finished. Context passes back to the `document` element. Because there are no more `item` children of `document`, context passes back to the `database` element. Because there are no more `document` children of `database`, context passes back to the root element. Because there are no more `database` children of the root, we're done:

```
</xsl:stylesheet>
```

The XSL stylesheet we used to get this result wasn't particularly sophisticated. XSL is a language capable of much more than we can get into here. But even with these few lines of example code and the ability to export an NSF file as DXL and look at it, you can see how you might turn your CRM database into an address list sorted by region, or whatever your application might be.

Turning Notes into HTML

The output declaration in our XSL stylesheet said `method = 'xml'`, so the transformer created the output as well-formed XML. We could specify `method = 'text'` and write a stylesheet that outputs text, or we could specify `method = 'html'` and create a stylesheet that outputs a Web page. This is particularly interesting because it means you can write LotusScript that turns Domino data into Web content that can be served to browser-based users without requiring that they access a Domino server—something that may be a plus in your environment for security or performance reasons.

The Create HTML agent

Here's an example. Open the example database `dxlofficesupplies.nsf` in Domino Designer and copy the stylesheet `dxlofficesupplies.xml` from the Files section of Shared Resources into your `c:` directory. Then open the database in a Notes client and in the view named Office Items, click in the left column to select half a dozen documents. Run the agent named 1. Create HTML from the Actions menu. The agent turns the selected documents into a document collection, exports the collected documents as DXL, and applies the stylesheet named `dxlofficesupplies.xml` to create an HTML page that is saved in `c:\dxl` as `dxlofficesupplies.html`. You can see the complete LotusScript code for the Create HTML agent in Domino Designer. Most of it looks very much like the previous examples we've seen. The major difference is the code that creates the document collection:

```
Dim db As NotesDatabase
Set db = session.currentDatabase
Dim dc As NotesDocumentCollection
Set dc = db.UnprocessedDocuments
Dim exporter As NotesDXLExporter
Set exporter = session.CreateDXLExporter(dc)
```

The documents marked in the view are gathered into the document collection by the `UnprocessedDocuments` method of `NotesDatabase`, and this collection is made the input for the DXL exporter—an elegantly simple way to export a single document or just a few documents selected by a user.

The XSL stylesheet, `dxlofficesupplies.xml`, starts processing this DXL with a template that writes the tags required for an HTML document to the output stream. Notice that we don't have to do anything to signal that certain lines in the stylesheet are to be copied to the output. Because we specified HTML in the output declaration, the XSL transformer treats any tag that doesn't begin with `xsl:` or `dxl:` as HTML to be output:

```
<xsl:template match="/">
  <html>
    <head>
      <title>XSL-XML Demo</title>
    </head>
    <body bgcolor="#ffffff" marginheight="0" marginwidth="0" leftmargin="0" topmargin="0">
      <br />
      <center>
        <xsl:apply-templates select="dxl:database"/>
      </center>
    </body>
  </html>
</xsl:template>
```

The `<apply-templates>` element is matched in the DXL by an element named `<database>`, so the next template writes a set of `<table>` tags and column headings to the output, then looks for DXL document elements:

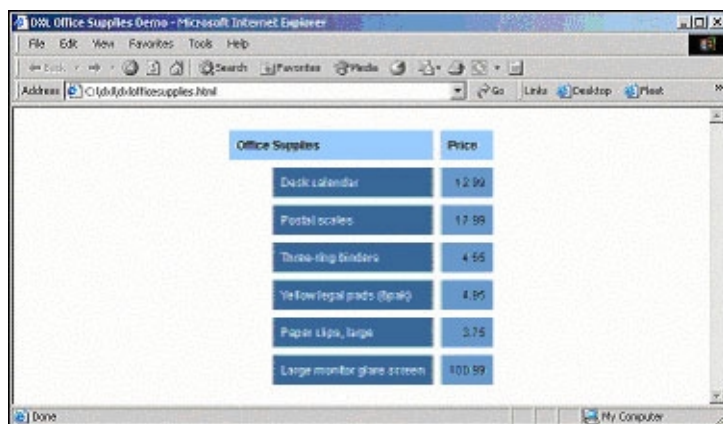
```
<xsl:template match="dxl:database">
  <table width="40%" cellpadding="8" cellspacing="8" border="0">
    <tr>
      <td colspan="2" bgcolor="99ccff">
        <font face="arial" size="2">
          <b>Office Supplies</b>
        </font>
      </td>
      <td bgcolor="99ccff">
        <font face="arial" size="2">
          <b>Price</b>
        </font>
      </td>
    </tr>
```

```
</td>
</tr>
<xsl:apply-templates select="dxl:document"/>
</table>
</xsl:template>
```

The template for DXL document elements creates a row within the table for each document found and cells for two items from the document, name and cost, then it writes the proper end tags for each HTML element:

```
<xsl:template match="dxl:document">
  <tr>
    <td width="5%"></td>
    <td width="20%" bgcolor="336699">
      <font face="arial" size="2" color="ffffff">
        <xsl:value-of select="dxl:item[ @name='Item']/dxl:text"/>
      </font>
    </td>
    <td width="5%" bgcolor="6699cc" align="right">
      <font face="arial" size="2">
        <xsl:value-of select="dxl:item[ @name='Cost']/dxl:number"/>
      </font>
    </td>
  </tr>
</xsl:template>
</xsl:stylesheet>
```

As context passes back through the other templates, other end tags are added to complete the HTML document. The stylesheet terminates, the XSL transformer closes the output stream, and the agent is done. Depending on how many Notes documents you selected in the view, when you open the HTML document in a browser, it will look something like this:



Office Supplies	Price
Desk calendar	12.99
Postal scales	17.99
Three-ring binders	4.55
Yellow legal pads (500)	8.95
Paper clips, large	3.75
Large monitor glare screen	100.99

Creating multiple HTML pages

Combining data from multiple Notes documents into a single HTML page is a technique that's especially useful for reporting. But perhaps even more useful is the ability to write documents out of a Domino database as individual HTML pages.

This might have benefits for both security and performance, and can provide an easy way to distribute Web content creation through your organization. An agent like the Create Multiple Pages agent in the DXL Office Supplies database makes it possible to create sets of HTML pages from Notes data that can be saved to a drive accessible by a Web server. You could create and update the pages on a regular schedule with a timed agent, serve them to the Web without the involvement (or exposure) of your Domino server, and control access to the content creation and editing through the ACL of the underlying database.

The agent named 2. Create Multiple Pages uses an XSL stylesheet, dxlofficesupplies_multi.xsl, that is slightly modified from the one used by Create HTML Page. Copy it from the Shared Files directory of the DXL Office

Supplies database and save it to c:\dxl before you run the agent.

The agent first creates a NotesViewEntryCollection that contains all the documents in the current view, which is named Office Items:

```
Dim view As NotesView
Set db = session.CurrentDatabase
Set view = db.GetView("Office Items")
```

It creates two counters: DocNumb, a sequence number for the current document, which is used to create a unique filename for the HTML page, and DocCount, the total number of documents in the view. The Doc object represents the current document:

```
Dim DocNumb As Integer
Dim DocCount As Integer
DocCount = view.EntryCount

Dim Doc As NotesDocument
Set Doc = view.GetFirstDocument
DocNumb = 1
```

Then it uses a Do While loop to iterate the process of creating the input and output streams, the exporter and XSL transformer, and to write the HTML file for the document. When the file is written, it decrements DocCount, increments DocNumb, gets the next document, and loops.

```
Do While DocCount > 0
```

```
....
```

Call exporter.process

```
    DocCount = DocCount - 1
    DocNumb = DocNumb + 1
    Set Doc = view.GetNextDocument(Doc)
Loop
Exit Sub
```

The XSL stylesheet called by the agent, dxlofficesupplies_multi.xsl, is only slightly modified from the version used to combine all the items into a single HTML page. It omits the template match to database because all the data we need is contained by the <document></document> tags. It writes out exactly the same HTML code, which this time produces a table with just one entry line—the item and price in the current document.

Using the DXL Importer

So far, we've focused on getting data out of Domino and into other formats and applications. But the XML features of LotusScript are equally useful for getting data from other sources into NSF files, using the new NotesDXLImporter class.

The Import DXL agent

The DXL importer is straightforward to use. To try it out, copy the brief XML file below and save it to your c:\dxl directory as additems.xml. Then in the DXL Office Supplies database, run the agent named 1. Import DXL from the Actions menu:

```
<?xml version="1.0" encoding="utf-8" ?>
  <database xmlns="http://www.lotus.com/dxl" version="6.0">
    <document form="OF">
      <item name="Item">
        <text>Saddle stapler</text>
      </item>
      <item name="Cost">
        <number>38.85</number>
      </item>
    </document>
  </document form="OF">
```

```
<item name="Item">
  <text>Computer cleaning kit</text>
</item>
HYPERLINK "C:\dxl\" <item name="Cost">
  <number>21.95</number>
</item>
</document>
</database>
```

The Office Items view should update immediately to display the two new documents included in this data: a saddle stapler and a computer cleaning kit. The Import DXL agent creates a NotesStream object to represent the data file and calls the DXL importer with the stream as the input and the current database as the output.

This works so smoothly because the XML data above conforms to the DXL DTD; the element names and parent-child relationships are all exactly the same as they would be if the same two items had been exported from the database. (The exported data would have included all the standard metadata, but its absence here doesn't cause the importer any problem—it supplies defaults for whatever it doesn't find.)

But what if the data you want to import looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<newitems>
  <item sku="123456">
    <description>Stainless steel stapler</description>
    <manufacturer>Acme Staplers</manufacturer>
    <price>38.85</price>
  </item>
  <item sku="6544321">
    <description>Computer screen cleaning kit</description>
    <manufacturer>Squeegee Manufacturing</manufacturer>
    <price>21.95</price>
  </item>
</newitems>
```

It's well formed XML, but it's not DXL. There's no database tag, and there are data elements that have no counterparts in the Notes database, like sku and <manufacturer>. So you'll have to parse it and reformat it.

The Import Data - SAX agent

In this case, the data structure is relatively close to DXL, so we can write a SAX parser agent. (The SAX parser was discussed in detail in the first installment of this article. See the previous *LDD Today* article in this series, "[LotusScript: XML classes in Notes/Domino 6](#)" for a thorough explanation of the SAX parser and examples of how the NotesSAXParser class is used.) You can run this example agent by first copying and saving the data above to your c:\dxl directory as newitems.xml. Then run the agent named 4. Import Data - SAX from the Actions menu. The two new items should pop into the view.

The agent doesn't save the data in DXL format; it just pipelines it from the parser to the importer. But if it did, the saved data would look like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<database xmlns="http://www.lotus.com/dxl" version="6.0">
  <document form="FO">
    <item name="SKU">
      <number>123456</number>
    </item>
    <item name="Item">
      <text>Stainless steel stapler</text>
    </item>
    <item name="Manufacturer">
      <text>Acme Staplers</text>
    </item>
    <item name="Cost">
      <number>38.85</number>
    </item>
  </document>
</database>
```



```
</document>
<document form="FO">
  <item name="SKU">
    <number>6544321</number>
  </item>
  <item name="Item">
    <text>Computer screen cleaning kit</text>
  </item>
  <item name="Manufacturer">
    <text>Squeegie Enterprises</text>
  </item>
  <item name="Cost">
    <number>21.95</number>
  </item>
</document>
</database>
```

This follows the structure of the DXL DTD. The <database></database> tags contain all the other data. Each document is defined by <document></document> tags, and each data field within the document by a nested set of tags—<item name="n"></item> and within that a pair of tags that define the data type—in this example <number></number> or <text></text>.

The Import Data - SAX agent opens the data file, newitems.xml, as a NotesStream object and submits it to a SAX parser which reformats the data as DXL, then pipelines it to a DXL importer.

```
'Open data file as NotesStream
Dim stream As NotesStream
Set stream = session.CreateStream
If Not stream.Open("c:\dxl\newitems.xml") Then
  MessageBox "Cannot open c:\dxl\newitems.xml. Check to make sure this directory exists.", "Error"
  Exit Sub
End If

'Create the SAX parser
Dim saxParser As NotesSAXParser
Set saxParser=session.CreateSAXParser(stream)

'Create the DXL importer
Dim importer As NotesDXLImporter
Set importer = session.CreateDXLImporter(saxParser, db)
```

The number of event handlers has been reduced to a bare minimum:

```
On Event SAX_StartElement From saxParser Call SAXStartElement
On Event SAX_EndElement From saxParser Call SAXEndElement
On Event SAX_Characters From saxParser Call SAXCharacters
On Event SAX_EndDocument From saxParser Call SAXEndDocument
On Event SAX_StartDocument From saxParser Call SAXStartDocument

On Event SAX_Warning From saxParser Call SAXWarning
On Event SAX_Error From saxParser Call SAXError
On Event SAX_FatalError From saxParser Call SAXFatalError
```

The SAXStartDocument handler copies the XML declaration and <database> tag into the output:

```
Sub SAXStartDocument (Source As NotesSAXParser)
  Source.Output(<?xml version='1.0' encoding='utf-8'?> + Chr(13)+Chr(10))
  Source.Output(<database xmlns="http://www.lotus.com/dxl" version="6.0">)_
  + Chr(13)+Chr(10))
End Sub
```

The corresponding SAXEndDocument handler closes both these tags.

The key events are those that start and end elements. The SAXStartElement handler does most of the heavy lifting. It uses a Select Case statement to process the items returned by the parser based on the element name:

```
Select Case ElementName
```

If the element is named newitems, the container element for the data, it is ignored:

```
Case "newitems"  
Exit Sub
```

If the element is a named <item>, then the subroutine writes a <document> tag that includes the alias of the Notes form the document displays in, "OF." Then it transforms the attribute sku into a named item, <item name="SKU">, and wraps the attribute's value in a <number> tag to set its data type:

```
Case "item":  
Dim i As Integer  
Source.Output({<document form="OF">})  
Dim attrname As String  
For i = 1 To Attributes.Length  
attrname = Attributes.GetName(i)  
If Attrname="sku" Then  
Source.Output({<item name="SKU"><number> } + Attributes.GetValue(attrname) + {</number></item>})  
End If  
Next
```

The element named <description> is transformed into its counterpart in DXL, an item named Item, and an opening <text> tag is written to prepare for the character data that will follow. The data type tag is written here rather than in the SAXCharacters handler because the parser doesn't distinguish between data types, and we only know what the type of character data should be because of the item name that precedes it:

```
Case "description":  
Source.Output({<item name="Item"><text>})
```

The price element is handled separately because its value has a <number> data type:

```
Case "price":  
Source.Output({<item name="Cost"><number>})
```

And finally the Catch-all Else handles other items that appear in the data (in this data file the only match to Else is the item named manufacturer). The element name is converted to proper case and used as the name attribute of an item of data type <text>:

```
Case Else  
nameProper$ = Strconv(ElementName, SC_ProperCase)  
Source.Output({<item name="} + nameProper$ + {"><text>})  
End Select  
End Sub
```

The SAXEndElement handler uses another Select Case statement to write the proper closing tags to each element:

```
Select Case ElementName  
Case "newitems"  
Exit Sub  
Case "price":  
Source.Output({</number></item>})  
Case "item":  
Source.Output({</document>})  
Case Else  
Source.Output({</text></item>})  
End Select  
End Sub
```

The output of the SAX parser is pipelined to the DXL importer, which creates new documents in the database for the two new items, Computer screen cleaning kit and Stainless steel stapler. Note that the Notes documents for these two items include two fields that came from the XML input aren't present in the other documents in the database—SKU and Manufacturer.

Producing other formats

The examples in this article have focused on transforming DXL to and from other XML languages. But it should be obvious by now that you can use all three of the parser tools—SAX, DOM, and XSL—to produce text in any format.

You can see some glimpses of what the new LotusScript XML support makes possible in the Designer help file code examples. A search for Examples: NotesSAXParser class in help will bring up an agent that displays a MessageBox for each SAX event as it processes an XML file. The example code for the NotesDOMParser class is an agent that produces a text report on the nodes it finds in the DOM tree it constructs from an example XML file.

We have done very little with the metadata that DXL exposes, but there's a wealth of data available as DXL that can be mined for purposes like knowledge management: Who is the most active contributor to the database? Who has contributed most recently? What trend do the access dates for the documents show—customer activity, employee productivity, revenue generation? The parser tools have great potential for reporting applications.

Conclusion

The XML support in LotusScript gives Domino developers an important new set of tools for using the most important existing standard for exchanging data between applications and systems. This article couldn't possibly teach you everything you need to know about DOM and SAX and XSL. But it can provide you with an introduction to how to use LotusScript to work with XML, and some code examples to get you started—importing and exporting data as DXL, the Domino XML language, and translating DXL to and from other XML languages using the DOM parser, SAX parser, and XSL transformer. The XML classes, together with the already rich features of LotusScript and the access DXL provides to the elements of the Domino database—both data and metadata—make LotusScript a powerful tool for data exchange.

ABOUT THE AUTHORS

Sally Blanning DeJean and David DeJean have been working with and writing about Lotus Notes and Domino for as long as they've existed. They were co-authors of the very first book about Notes, *Lotus Notes at Work*. Sally, a CLP Principal, has written other books about Notes and is a full-time Notes/Domino developer. David, a CLP, has been an editor and writer for several computer publications. He is a partner in DeJean & Clemens, a firm that develops Notes and Internet applications and technical and marketing communications.