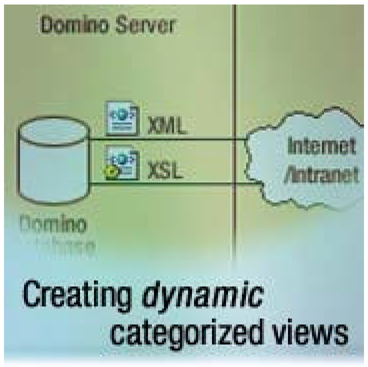


Level: Advanced
Works with: Notes/Domino
Updated: 03-Nov-2003

by
Aram
Hovhannisyan

As the number of documents in a Domino database grows, categorized views become increasingly important. By clicking on categories and drilling down to the last level, users can filter out redundant information and find the documents they want faster and easier.

When you create a categorized view, you have a choice of displaying it as a Java applet or as an HTML view. Applet views have the same features as native Notes client views. But they take a fair amount of time to load, and as the number of documents grows, view applets are known to show some problematic behavior.

HTML views, on the other hand, are quicker and work great, but there are a few things you need to bear in mind when creating them. For example, every time a user clicks a category, the Domino server has to re-render the whole view, expanding only the category requested. Multiply this by the number of users simultaneously clicking on your view and you have some idea of the level of redundancy here—not only does a big chunk of view travel back and forth between client and server, but also the screen visibly flickers every time you click a twistie. This can become annoying, especially for users who work extensively with the view. Another drawback is that you can have only one category expanded at a time because expanding another category collapses the previously opened one. The ideal way to avoid these issues is to come up with a method to dynamically refresh only the portion of the view being requested, while leaving the rest of the view intact.

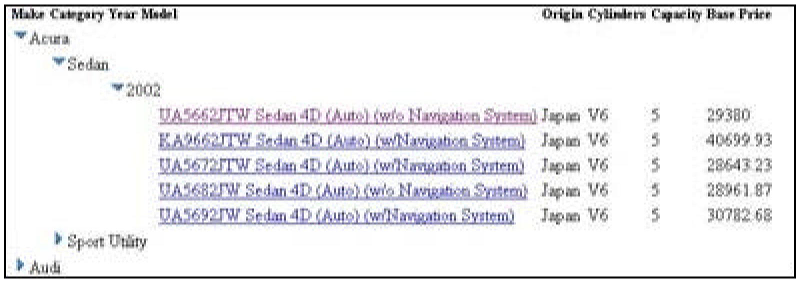
This article explains how to do this. By following our suggestions, you can create more effective views, reduce network traffic and server load, and enhance the UI experience for end-users. Our methods for creating dynamic views are fairly simple and straightforward and avoid some of the more complicated and cumbersome solutions that involve creating a separate agent or preloading whole views. This article shows you how to create and deploy your dynamic views on existing Web applications with minimal modifications to the application's current design. In addition, it explains how to synchronize the view with the currently opened document to implement a Web page consisting of two frames that show the view in one frame and the document in another. (You can download the database containing the sample code used in this article from the Sandbox.)

This article assumes that you're an experienced Notes/Domino and Web application developer. In addition, the *LDD Today* article "Building dynamic collapsible views for Web applications" provides good background information for some of the concepts discussed in this article.

The ReadViewEntries command

The key to our technique for creating dynamic categorized views is the ReadViewEntries URL command, first introduced in Domino Designer 5.0.2. This command pulls XML view information and transforms it into HTML using Microsoft's parser and an XSLT (XML transformation). In the world of Web-based applications, XML to HTML transformations are the most common use of XSLT. (For more on parsing XML data to HTML, see the *LDD Today* article "Using Domino data in Web Applications: XML lessons from iNotes Web Access.) Some developers use methods exposed by XML DOM to parse the XML data retrieved from the Domino database. However, we use an alternative way to load portions of HTML using XSLT. This article walks you through the basic elements of XSLT, giving you a good head start towards creating your own XSLT style sheets for your top-of-the line Web applications.

The ReadViewEntries command and its arguments allow you to load only the requested portion of the view. To understand how this works, let's examine the first rows of the view from our sample database, and its associated XML. First let's look at the view:



Make Category Year Model	Origin	Cylinders	Capacity	Base Price
Acura				
Sedan				
2002				
UA5662JTW Sedan 4D (Auto) (w/o Navigation System)	Japan	V6	5	29380
KA9662JTW Sedan 4D (Auto) (w/Navigation System)	Japan	V6	5	40699.93
UA5672JTW Sedan 4D (Auto) (w/Navigation System)	Japan	V6	5	28643.23
UA5682JW Sedan 4D (Auto) (w/o Navigation System)	Japan	V6	5	28961.87
UA5692JW Sedan 4D (Auto) (w/Navigation System)	Japan	V6	5	30782.68
Sport Utility				
Audi				

And here's the associated XML:

```
<viewentry position="1" noteid="80000F10" children="2" descendants="7" siblings="15">
    <entrydata columnnumber="0" name="txtMake" category="true">
        <text>Acura</text>
    </entrydata>
</viewentry>
<viewentry position="1.1" noteid="80000F14" children="1" descendants="5" siblings="2">
    <entrydata columnnumber="1" name="$1" category="true">
        <text>Sedan</text>
    </entrydata>
</viewentry>
<viewentry position="1.1.1" noteid="80000F18" children="5" descendants="5" siblings="1">
    <entrydata columnnumber="2" name="$2" category="true">
        <text>2002</text>
    </entrydata>
</viewentry>
```

Each <viewentry> element represents a row from the view holding <entrydata> elements for each column. Notice the children and position attributes. You use the children attribute to distinguish between the two types of view entries—those that have children (so are expandable/collapsible) and those that do not (the last level entries). Also, the children attribute helps you define the number of nested subcategories to retrieve. You use the position attribute to define the starting position of the nested subcategory which is composed from the position of the current category with .1 appended.

As you can see, the ReadViewEntries command retrieves the whole view data as if the view were expanded. To control this behavior, you use the optional arguments listed in the following table:

Argument	Description
Start = n	Where n is the row number to start with.
Count = n	Where n is the number of rows to display. The default number of entries returned by ReadViewEntries is 30. To return all entries, set n to –1.
Collapse = n	Where n is the row number to display in collapsed format.
CollapseView	Displays view in collapsed format.

Note that in the XML representation of a view, the row number coincides with the value of the position attribute of the <viewentry> element.

The very first command you issue is:

```
?ReadViewEntries&Count=-1&CollapseView
```

which ensures that you get the first level categories only. Let's examine the first <viewentry> element:

```
<viewentry position="1" noteid="80000F10" children="2" descendants="7" siblings="15">
    <entrydata columnnumber="0" name="txtMake" category="true">
        <text>Acura</text>
    </entrydata>
</viewentry>
```

Notice that this element is located at the first ("1") position and has two immediate children (subcategories). Suppose in your next step you want to expand this category and get only the two subcategories of the next level. To do this, start with the next <viewentry> at the position 1.1 (&Start=1.1) and use &Count=2. But because ReadViewEntries retrieves the view data in expanded format by default, calling it with just these two arguments is not enough. Instead of getting the <viewentry> elements for the positions 1.1 and 1.2, you get <viewentry> elements for the positions 1.1 and 1.1.1, which is definitely *not* what you want to do! To avoid this, you need to add one more argument: &Collapse=1.1.

At this point, you might ask: "Why do we collapse the row 1.1 only? What about the row 1.2? Shouldn't we have it collapsed too?" The answer to this question lies in an interesting behavior of the Domino view. When you collapse a given row, all siblings of that row are also collapsed. This behavior is easy to see on any categorized view. Simply open "view all expanded" and click any row inside a category. You see that all other rows that belong to the same category collapse too.

Now you are ready to issue your next command:

```
?ReadViewEntries&Start=1.1&Count=2&Collapse=1.1
```

This displays the data for the next level only (without nested subcategories). Maintaining the previously described logic, repeat the pattern until you get to the bottom level of a given category where <viewentry> elements have no children attribute at all. Instead, note the UNID attribute holding the document's unique identifier as its value, which you use later on.

Now that you know how to control the XML returned from a categorized view, let's discuss the HTML that you are going to build from the XML data.

HTML

For each view category, you have two <SPAN> elements and one <DIV> element. The first <SPAN> holds the plus (+) or minus (–) signs, indicating whether the category is collapsed or expanded. The second <SPAN> holds the actual category title. Finally, we need a <DIV> element to hold the next level category data, consisting of two <SPAN> and one <DIV>. The pattern repeats itself until you expand the last category. In this case, things become a bit different because you have to format the last level data using columns—the same way Domino does. For this purpose, the last level <DIV> element holds an HTML table.

The following is the HTML design structure sample for categories:

```
<DIV><SPAN> + </SPAN><SPAN>View Title 1 </SPAN></DIV> // category collapsed
<DIV></DIV>
<DIV><SPAN> + </SPAN><SPAN>View Title 2 </SPAN></DIV> // category collapsed
<DIV></DIV>
<DIV><SPAN>  - </SPAN><SPAN>View Title 3 </SPAN></DIV> // category expanded!
```

```
<DIV>
  <DIV><SPAN> + </SPAN><SPAN>View Title 3.1 </SPAN></DIV> // category collapsed
  <DIV></DIV>
</DIV>
```

And here's the HTML design structure sample for the last level:

```
<SPAN> - </SPAN><SPAN>View Title 1.1.1.1 </SPAN></DIV> // category expanded
<DIV>
  <TABLE>
    <TR>
      <TD>..Car Model Info  </TD>
      <TD>..Country of origin Info  </TD>
      <TD>..Amount of cylinders Info  </TD>
      <TD>..Citing capacity Info  </TD>
      <TD>..Base Price Info  </TD>
    </TR>
    …..
  </TABLE>
</DIV>
```

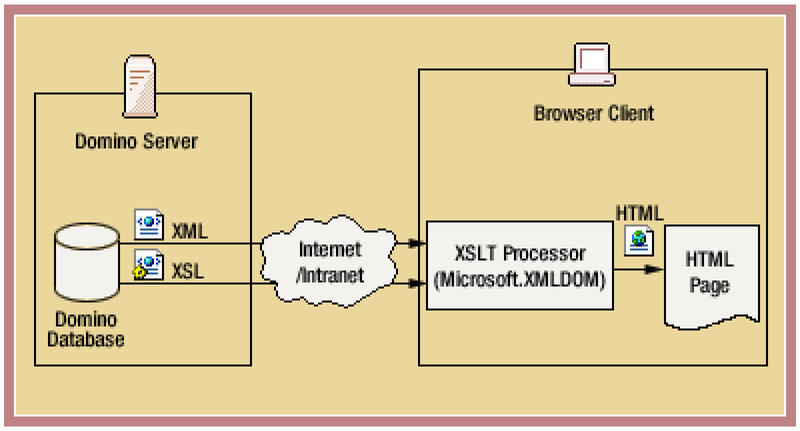
XSLT and XPATH

Now that we have discussed what our resulting HTML is going to look like, let's move our focus to the successor to XSL style sheets, XSLT—a transformation language used to transform XML to another XML or to HTML. The process of transformation is handled by an XML parsing engine (XSLT Processor), which takes both XML and XSL documents and transforms the XML based on the rules outlined in the XSLT. The process works as follows:

1. An XML file and an XSLT are read and parsed into memory, where the resulting parsed XML document is specified as a source tree and the XSLT as a style sheet tree.
2. The template rules declared in the XSLT are applied to each matching element in the source tree, thus producing an output, which is also defined as a tree.
3. The resulting tree is output in a desired format.

You can think of XSLT as tree-to-tree transformations.

The most common use of XSLT is XML to HTML transformations. The parsing engine that you use in your solution comes integrated with Internet Explorer. Both XML and XSL are delivered to the browser, where the built-in XML parser handles XSL transforms, producing the desired HTML:



You are now ready to proceed to your main step—the creation of the XSLT. The main challenge that many developers face while learning XSLT is that the coding logic is not procedural, but rather declarative. This is because the logic of transformation is based on matching. The XSLT parser looks for patterns in XML data as indicated by the match = ”expression” instruction. The match pattern determines where this template applies based on the XPath expression.

XPath is a language for addressing parts of an XML document, designed to be used by XSLT and XPointer. It is also used to provide basic facilities for manipulation of strings, numbers, and booleans. For example, the instruction:

```
< xsl:template match = "viewentry[@children]" >
```

directs the parser to look for a tag that has the children attribute.

An XSLT basically consists of a set of templates with each template matching some set of elements in the source tree. Non-XSL elements within a template (elements that do not start with the xsl: prefix) are placed in the result tree. For example, the following code snippet from our XSL:

```
<xsl:template match="viewentry[@children]">
<div><span>+</span>
<span>
   <xsl:attribute name="NextEntrySrc"> ... </xsl:attribute>
   <xsl:value-of select="entrydata"/>
</span>
</div>
<div></div>
</xsl:template>
```

applied to the following XML:

```
<viewentry position="1" children="2" …>
   <entrydata columnnumber="0" name="txtMake" category="true">
      <text>Acura</text>
   </entrydata>
</viewentry>
```

results in the following HTML:

```
<div><span>+</span><span NextEntrySrc=" ...">Acura</span></div>
<div></div>
```

This instruction creates a NextEntrySrc attribute node and attaches it to the <SPAN> element. The value for the NextEntrySrc attribute is constructed using the logic described in the previous section on the ReadViewEntries command. (See also the sample code for more details.) The other instruction inserts the value of the selected node as text.

Here are the remaining instructions used in our solution:

Instruction	Syntax	Description
<xsl:apply-templates>	<xsl:apply-templates select=expression> <!-- Content --> </xsl:apply-templates>	Invokes the template rules against the node-set returned by the select expression.
<xsl:if>	<xsl:if test=boolean expression> <!-- Content --> </xsl:if>	Evaluates the template if and only if the test expression evaluates to true.

As mentioned earlier, the resulting HTML is going to be a little different depending on which view entry you transform (category or last level entry). So in your XSL, you need another matching template for viewentries that do not have the children attribute. In this case, the resulting HTML is a <TABLE> holding <TR> for each document with column values in each <TD>:

```
<xsl:template match="viewentries">
<TABLE>
    <TR>
        <TD></TD>
    </TR>
</TABLE>
</xsl:template>
```

See the sample database for the full listing of the XSLT.

JavaScript support

The Microsoft XML parser is a COM component featuring a programming model that supports JavaScript along with other programming languages. Microsoft Internet Explorer (5.0 and higher) integrates the XML parser into the browser, which allows you to manipulate it using JavaScript where an instance of the XML parser can be created with the following code:

```
var obj = new ActiveXObject("Microsoft.XMLDOM")
```

Please refer to the MSDN Web Products page for more details on the XML parser.

At this point, all you have left to do is to add two short JavaScript functions for handling the expand/collapse mouse clicks. Here is the first one called toggleSection(obj, bSync):

```
function toggleSection(obj, bSync)
    {
        objLink = obj.nextSibling
        objCont = objLink.nextSibling
        if(objCont.style.display == 'block' && !bSync)
        {
            objCont.style.display = 'none';
            obj.innerText = '+ '
        }
        else
        {
            objCont.style.display = 'block';
            obj.innerText = '- '
        }
            if(objCont.children.length == 0)
        {
            objCont.innerHTML = getHtmlFromXml(objLink.NextEntrySrc)
        }
    }
```

This function is pretty straightforward, as you can see. It processes the onClick event triggered when a user clicks the plus (+) or minus (–) signs or a category title. At line 3 (objLink = obj.nextSibling), you obtain a reference to the <SPAN> object holding the NextEntrySrc attribute that you use to retrieve the next level data.

The second function called getHtmlFromXml(xmlsrc) is where you transform the XML:

```
function getHtmlFromXml(xmlsrc)
    {
        oXml = new ActiveXObject("Microsoft.XMLDOM")
        oXml.async = false
        oXml.load(xmlsrc);
        strRet = oXml.transformNode(oXsl);
        oXml = null;
        return strRet
    }
```

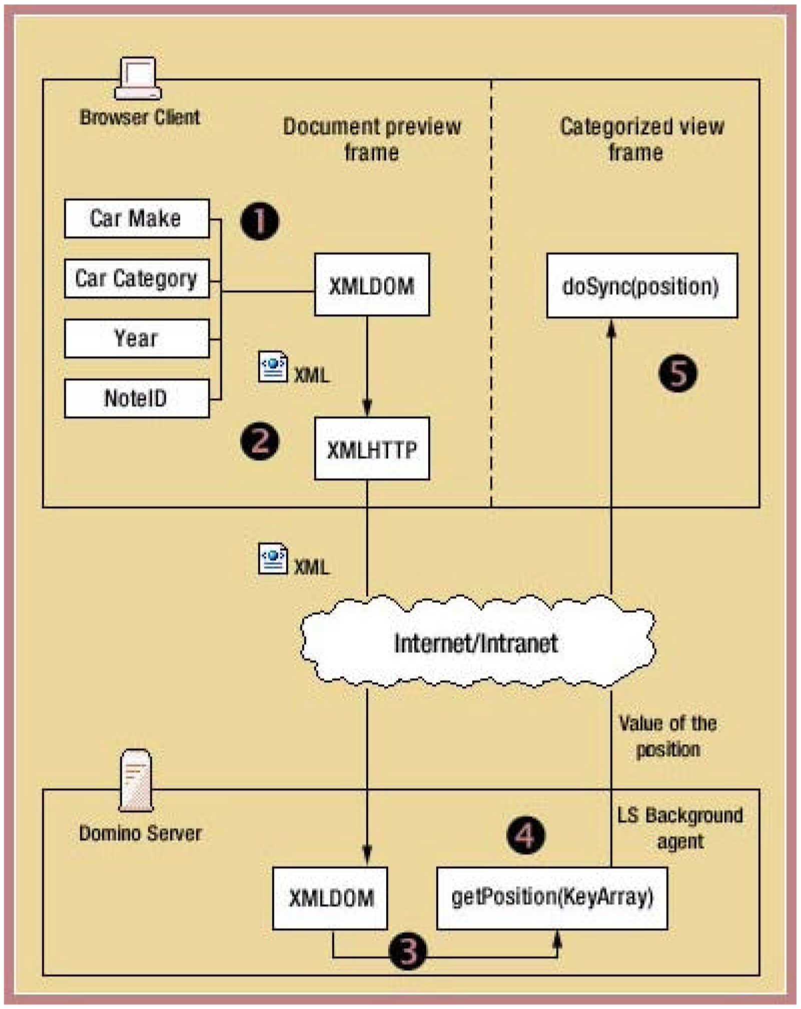
After you obtain a reference to the parser at line 3, in the next line you have to make sure that the subsequent load of the XML data into the parser is synchronous, otherwise transformation at line 6 (strRet = oXml.transformNode(oXsl);) may start before the download is complete. Reference to the XSL object (oXsl) has been obtained during the onLoad event of the page. (See the sample code for more details.)

Synchronization

Suppose your Web application consists of two frames. The first frame holds the categorized view and the second one is used to preview the actual documents. In similar designs, you often must perform the task of synchronizing the view with the document currently being previewed. The main obstacle in solving this task is to figure out the position of the current document in a given view's hierarchy. After obtaining the document's position, use it to toggle all sequential categories that comprise the full path to the document. In other words, for a document with the position 5.2.3.1, you need to toggle the categories at the positions 5, 5.2, and 5.2.3 in the same order. The following doSync function implements this logic described and at line 12, ensures that the expanded branch is in the user's view:

```
function doSync(strPos)
    {
        ar = strPos.split(".")
        temp = ""
        for(i = 0; i < ar.length - 1; i++)
    {
        temp = temp + ar[i]
        obj = document.getElementById(temp)
        toggleSection(obj.children(0), true)
        temp = temp + "."
    }
        obj.scrollIntoView()
    }
```

The only remaining question is: "How do we get the position of the document in the JavaScript?" To solve this, you need to create a background agent using the getPosition method of the NotesViewEntry class. You get the NotesViewEntry object itself from the NotesView object using the getEntryByKey(keyArray) method, where the KeyArray parameter is composed from the values of fields used in the categorized columns of the view plus the NoteID of the document. These values are collected on the browser and sent to the agent using the XMLHTTP object (Microsoft.XMLHTTP), which allows sending an arbitrary HTTP request and receiving the response. The following is a step-by-step description of the solution in which XML-formatted data is sent to the server, where a background agent processes it and returns the result to the calling script:



This involves the following steps:

1. From the currently opened document, you collect the necessary field values and create an XML DOM object.
2. Using the XMLHTTP object, the XML data is sent to the agent.
3. Inside the agent, the XML is parsed using the already familiar XMLDOM (note that in Domino 6, you can use LotusScript's built-in XML parser), and the data is used to construct the KeyArray parameter to be used by the getEntryByKey method.
4. After finding the position of the viewentry, it is returned to the calling object (XMLHTTP) through the Print statement.
5. Finally, the upper frame (the one that contains the view) is reopened with the document's position value passed to the doSync function.

Here is the full listing of the function from our sample code:

```
function getDocPosition(strDocNoteID)
   {
      f = document.forms[0]
      objHTTP = new ActiveXObject("Microsoft.XMLHTTP")
      strUrl = "/" + f.dbName.value + "/getDocumentPosition?OpenAgent"
      objHTTP.open("POST", strUrl, false, "", "")
      xmlDom = BuildRequest(strDocNoteID)
      objHTTP.send(xmlDom)
      pos = objHTTP.responseText
      parent.frames.frmViewTop.location = "/" + f.dbName.value + "/vwMainView?openview&pos=" + pos
```

```
        objHTTP = null
    }
```

After you get the position at line 9, the script constructs a new URL (passing the position value in a Query_String) and reopens the frame containing the categorized view, which calls the doSync function with the respective position parameter during the onLoad event of its supporting form.

Conclusion

We have shown you how optional parameters for the ReadViewEntries command can be used to load only a portion of a view in XML. After that, we applied XSLT and used the resulting HTML to update the portions of the initial HTML page using DHTML. In addition, we demonstrated how the Microsoft.HTTP ActiveX object, in conjunction with the LotusScript agent, can be used to synchronize the categorized view with the document currently being previewed in multi-framed Web applications. We have seen how the union of these technologies allows us to expand the functionality of categorized views on the Web.

During the course of the article, we walked you through the basics of developing an XSLT and manipulating the parsers integrated into Internet Explorer. (For more detailed information on XML support in browsers other than Internet Explorer, we recommend the article "Using Domino data in Web Applications: XML lessons from iNotes Web Access.") Because developing XSLT is somewhat independent from the transforming code, introducing XSLT to your Web applications allows for more flexible, scalable, and easily maintained design architecture.

ABOUT THE AUTHOR

Aram Hovhannisyan is a Senior Consultant/Developer at IB Systems Inc. in Clearwater, Florida, focusing on Domino and WebSphere. He began his Lotus career in 1997. Since that time, Aram has been involved in numerous complex solutions involving Domino with other technologies, along with extensive experience in Domino-based Web applications. Aram hold an MS degree in Theoretical Physics from Yerevan State University, Armenia with specialization in numerical modeling in physics.