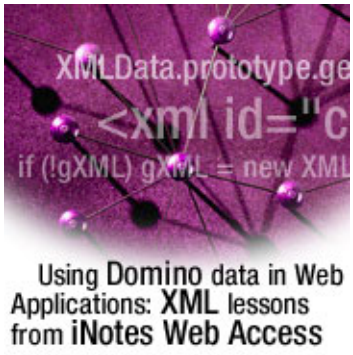
Lotus Developer Domain
The Technical Resource for Lotus software
LDD Today



Level: Intermediate
Works with: iNotes Web Access, Domino
Updated: 04-Nov-2002

by
Vinod Seraphin
and David DeJean

The development effort that went into iNotes Web Access broke new ground in the way Web technologies like DHTML and XML were used to produce dynamically interactive applications for use in a Web browser. A previous *LDD Today* article, "**Exploiting dynamic HTML: Lessons from iNotes Web Access,**" took a closer look at how the iNotes development team used DHTML to build a more interactive user interface and how what we learned can help you to create the best possible user interface for your Web applications. Another key lesson from iNotes is the way iNotes works with Domino view data, not as HTML but as XML.

In this article, we describe how to extract XML from Domino using the ReadViewEntries URL command. The code examples come from WebCal, a sample application based on a prototype that preceded iNotes Web Access. You can download these code examples from the **Sandbox**. And if you want to take a look at iNotes Web Access (and kick its tires), go to the **iNotes Web Access live demo** here on LDD.

This article assumes you have experience as a Notes/Domino application developer with some knowledge of HTML, JavaScript, and XML.

Getting Domino to speak XML

When we started working on iNotes Web Access, our goal was to develop the best possible browser-based application for accessing Domino mail and calendar data by exploiting cutting-edge browser trends. One such browser feature trend is the increased support for retrieving and manipulating XML within a browser. We felt we could exploit the browser's XML capabilities to minimize the need to reload the entire page as the user tried to display a different portion of a view. This would provide a better user experience as well as minimize the processing done on the server.

We discovered that Domino had introduced a special URL command, ReadViewEntries, to work in conjunction with the Domino 5.0 view applet and had exposed it for other developers to use in release 5.0.2. Here's an XML example of the result of a ReadViewEntries command from a view with a single calendar entry for a Staff Meeting:

```
<?xml version="1.0" encoding="UTF-8"?>
<viewentries toplevelentries="1">
<viewentry position="1" unid="F6169247625D839A85256B28000DE3AB" noteid="95A" siblings="1">
<entrydata columnnumber="0" name="$134">
<datetime>20011219T180000,00+00</datetime>
</entrydata>
<entrydata columnnumber="1" name="$149">
<number>160</number>
</entrydata>
<entrydata columnnumber="2" name="$144">
<datetime>20011219T180000,00+00</datetime>
```

```
</entrydata>
<entrydata columnnumber="3" name="$145">
<text>-</text>
</entrydata>
<entrydata columnnumber="4" name="$146">
<datetime>20011219T190000,00+00</datetime>
</entrydata>
<entrydata columnnumber="5" name="$147">
<textlist><text>Staff Meeting</text><text>Location: Gold Room</text></textlist>
</entrydata>
</viewentry>
</viewentries>
```

You can try this yourself. In a browser, enter the full path to your mail file on a Domino server and end with "yourmailfile.nsf/($Calendar)?ReadViewEntries." Don't worry, you won't get all the entries you ever added to your calendar—the default number of entries returned by ReadViewEntries is 30. You'll see <viewentry></viewentry> tags for each item, and inside those tags <entrydata></entrydata> tags for each column in the view.

We found that this URL command mostly met our needs. However, the view applet doesn't support calendar views, so the original version of the ReadViewEntries command only supported arguments that allowed the view applet to implement its designed functionality, that is:

- &Start lets you specify the 1-based absolute row within the view to begin returning entries.
- &StartKey provides a way to jump to a specific position in the view based on the view's primary sort.
- &Count lets you change the default number of items returned.
- &SortAscending and &SortDescending let you specify an alternate column number to sort the view by.

For iNotes Web Access, we wanted to use this URL command not only for list views, but also for calendar views. The first big hurdle we encountered was with the &StartKey argument: The value could be only text—not much good for calendar data, which is sorted based on a date/time column. For Domino 5.0.8, the first release that supported iNotes Web Access, the iNotes team worked with the Domino developers to add a few new capabilities to ReadViewEntries:

Argument	Description
&KeyType	The default is &KeyType=text. If you specify &KeyType=time, then &StartKey may be an ISO 8601 representation of a date/time value, for example &StartKey=20020101T140000,00Z or &StartKey=20020101T090000-0500 The first value represents 2 PM GMT on January 1, 2002. The second represents local time with an offset from GMT. Either format can be used to represent 9 AM Eastern Time on January 1, 2002.
&UntilKey	This argument can specify a value upon which to end the range (not inclusive). Like &StartKey, it can be either text or time. It solves one of the problems of working with calendar data—it allows you to specify a range of data without knowing exactly how many items there are in the range. You can use &StartKey and &UntilKey to specify all the calendar entries in the period beginning on or after March 1 and ending before April 1, for example.
&StartKey	This argument was enhanced to not return an error when the key specified was after the last entry in the view. Rather, some XML is returned representing that there are no entries.

With the ReadViewEntries command and these arguments, all a Web application developer has to do to extract XML data from Domino is to make a URL request. That's pretty easy. And if finer control of the data is needed, it's just a matter of tweaking the view or creating a new one.

Loading XML data into the browser

Now that we know how to retrieve Domino view data in XML format, the next challenge is properly loading and manipulating this within the browser. Microsoft Internet Explorer (IE) 5 and later has a special HTML element named XML, often referred to as an XML island, that exists for this very purpose. The browser also has a separate XML document object model (DOM) to manipulate any XML contained within this special element.

Regrettably, older browsers, such as Netscape 4.7, do not have any built-in support for loading or manipulating

XML. To load XML in such legacy browsers, you need to write an applet, such as the Domino view applet, that can retrieve and parse the XML. However, Netscape 6.0 has better support for dealing with XML documents, but it is a bit more complicated than what can be done with IE 5 or later. Mozilla 1.0, seeing the success of IE's XML loading and manipulation capabilities, added specific DOM methods to make XMLHTTP requests and parse the resultant XML.

We'll discuss the Netscape 6.0 and Mozilla 1.0 capabilities shortly, but first let's discuss IE's XML capabilities. IE 5 and later offers separate DOMs for HTML elements and XML elements. Each DOM treats each element of the appropriate type on the page as objects and provides ways to set and get various properties and to invoke methods associated with the element. This XML DOM is available in a feature called the XML island.

Welcome to the XML Island
An XML island is an invisible element, a sort of inline frame for holding XML data in an HTML document. Any well-formed XML can be used. The island is delimited by the tag <XML ID="NEWDOC"></XML>. You can use the ID property to reference the data and write script to load it and manipulate its contents. Here's an example:

```
<xml id="calData"></xml>
<script language="JavaScript">
calData.async = false; bSuccess = calData.load("/mail/juser.nsf/($Calendar)?ReadViewEntries");
// Data is loaded into island by the time execution
// reaches this point.
</script>
```

The async property used in this example is an important feature of the XML island. When it is set to false, the XML data is loaded synchronously—that is, execution is paused until the data is loaded. If it were set to true, execution would continue while the data loaded. Setting async=false means you don't have to write code that checks to see if the XML document is finished loading before you start manipulating the data.

The XMLDOM and XMLHTTP ActiveX objects
Another way to load XML in IE 5 or later, similar to the example above using an XML island, is to call the Microsoft Parser (XMLDOM) or Microsoft httpRequest (XMLHTTP) ActiveX objects using JavaScript, as in the following example:

```
<script language="JavaScript">
var xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
xmlDoc.async = false;
bSuccess = xmlDoc.load( "/mail/juser.nsf/($Calendar)?ReadViewEntries");
</script>
```

Here's another example:

```
<script language="JavaScript">
var xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
xmlHttp.open("GET", "/mail/juser.nsf/($Calendar)?ReadViewEntries", false);
// Third argument is async specifier
xmlHttp.send(null);
// xmlHttp.responseXML contains the resultant XML doc
</script>
```

The advantage of using the XML island is that it will work even when the user's browser settings disable ActiveX controls.

For Netscape 6
Netscape 6.0 does not support XML islands, but it does support XML in frames. You can load an XML document into a hidden frame (one with a height and width of 0 pixels) or an iframe and manipulate it there. The drawback is that frames load asynchronously.

One way to reliably detect when the XML document has finished loading is to load the document into a single frame within a frameset. The frameset element supports an onload event that executes when all frames have loaded. This allows you to load the XML into a frameset with a single frame and to use the onload event to start processing the data.

For Netscape 6.1 and later and Mozilla 1.0

Netscape 6.1 and later and Mozilla 1.0 support a new XMLHttpRequest object that allows you to load XML documents in a manner very similar to the above example using the Microsoft.XMLHTTP ActiveX object:

```
<script language="JavaScript">
var xmlHtttp = new XMLHttpRequest();
xmlHttp.open("GET", "/mail/juser.nsf/($Calendar)?ReadViewEntries", false);
// 3rd argument is async specifier
xmlHttp.send(null);
// xmlHttp.responseXML contains the resultant XML doc
</script>
```

Parsing the loaded XML

After the data has been loaded into the browser, the next challenge is parsing the XML to extract the data values of interest to your application. Once you have this data, it can then be rendered to the screen via various dynamic update mechanisms.

An XML document is always a tree structure with the outermost tag being the tree root and each nested tag being a branch (or node) off the root. The IE 5 or later XML DOM has a powerful selectNodes method that you can use to quickly extract subsets of the XML tree represented by the entire XML document. The selectNodes method, by default, takes an XSL pattern syntax but may also be told to accept the more standard Xpath query syntax. The following table shows some simple XSL pattern syntax examples:

Syntax	Description
/	The root node
*	Any element
el1	Any element named *el1*
el1|el2	Any element named *el1*or any element named *el2*
parent/child	Any element named *child* with a parent named *parent*
ancestor//child	Any element named *child* with an ancestor named *ancestor*
el[@attr="value"]	Any element named *el1* with an attribute named *attr*
@attr	Any attribute named *attr*
@*	Any attribute

Later in this article, as we explore the WebCal example code available from the **Sandbox**, we will point out specific code using XSL patterns to help extract the desired XML data values.

For Netscape 6.0 and Mozilla 1.0, the DOM does not include anything with the power of IE's selectNodes. Rather, you can use the getElementsByTagName and getAttribute methods to extract the desired XML data values. The xml_demo sample code shows how these alternate mechanisms are used to implement the XMLData JavaScript object that extracts data returned by Domino's ReadViewEntries URL command.

Both IE and Netscape 6.0 and Mozilla 1.0 support key attributes for each node within the XML tree. These include childNodes, a collection of child nodes off the current node as well as some key self-explanatory attributes: firstChild, lastChild, nodeName, and nodeValue.

The WebCal example: Fetching XML from Domino

You can see how DHTML and JavaScript work with XML data in the WebCal code examples posted in the **Sandbox** as part of the Lotusphere 2002 session. WebCal is a modified version of the original early prototype that led to the development of the actual iNotes Web Access product. Unlike the final product, WebCal makes extensive use of framesets and has most of its design in static files off the server's HTML directory. Also, the JavaScript code isn't obfuscated (comments stripped out and large identifiers remapped to very small ones to boost performance). The code for working with XML data to display the view is very similar to what is done in iNotes Web Access.

The WebCal application displays a monthly calendar in a browser. It authenticates the user and manipulates appointment records stored in a Notes mail file. You can create, edit, and delete appointments and drag-and-drop calendar entries to reassign them. The application supports only IE 5 or later. The application includes several

HTML and JS files, which are stored on the server in the path ...\Domino\Data\domino\HTML\WebCal.

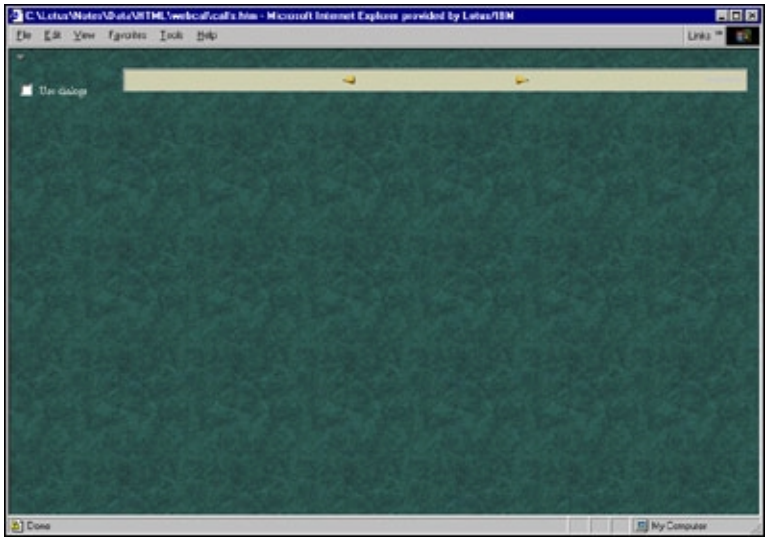
Four HTML documents are used to set up the application and log in the user. The main.htm file is a frameset and the starting page for the WebCal application. The default.htm and login.htm files are loaded into main.htm to check the user's browser and version and to force a Notes login. The main.htm document constructs two XML islands—data and action—for Internet Explorer:

```
<xml id="data"></xml>
<xml id="action"></xml>
```

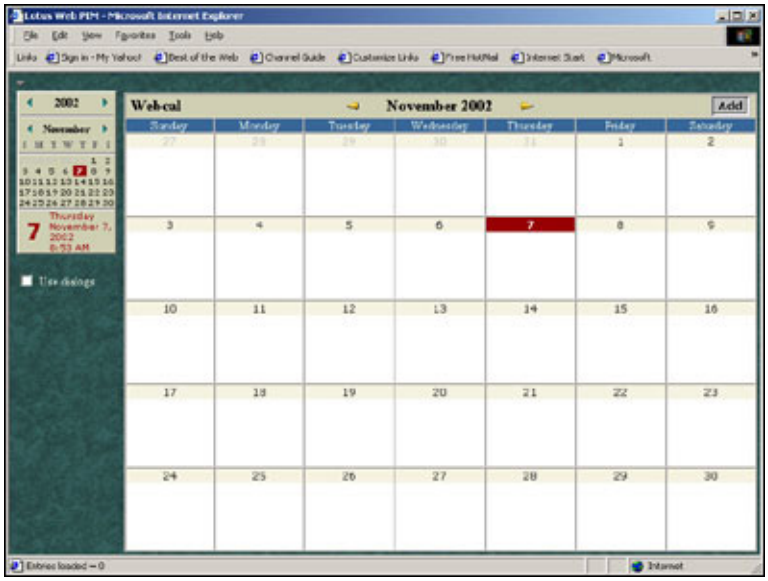
The data island is where IE loads the calendar data it receives from Domino. The action island at one point was used to submit actions (invoke agent), but is presently not used.

The visible interface for these functions is the login page:

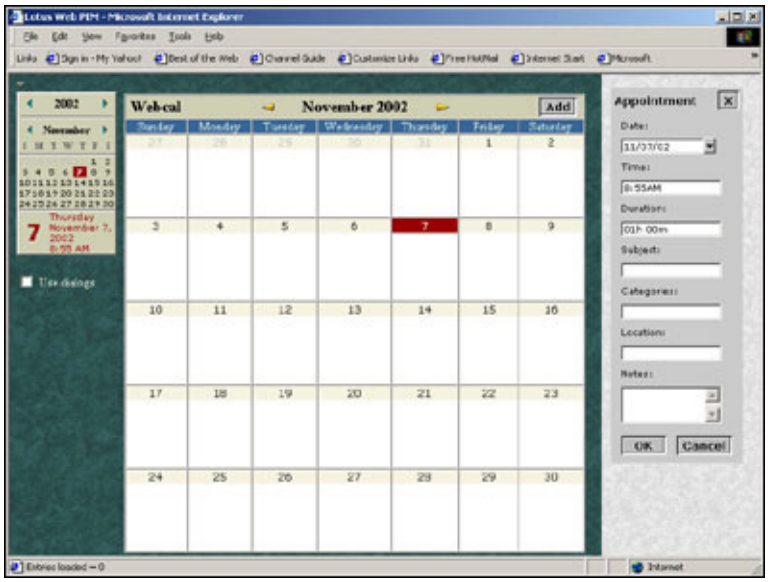
The calfs.htm file, shown in the following screen, is the main calendar page. It's a frameset that brings together four elements, including a column on the left-hand side to hold the date navigator, navigator.htm; the calendar display area that holds mview.htm, a one-month calendar; and a navigator bar at the top to change the displayed month.



Note that the basic HTML file shown previously doesn't include the date navigator and calendar table. These are created by JavaScript code on the fly. Here's the complete application:



The details frame is used to display the HTML for individual calendar entries. It can be hidden or revealed and is sometimes also used to submit actions to the server, such as requests to delete appointments:



The details frame loads either apptread.htm, a form that, when visible, displays the contents of an appointment record in read-only mode, or apptedit.htm, the editable version of the form. These HTML documents make use of several external JavaScript files:

- apptcommon.js
- button.js
- datepick.js
- datetime.js
- dominodata.js
- dropdown.js
- main.js
- utilities.js

Loading the calendar's XML data

One of the most basic functions of the application is to update the displayed calendar in response to a change either in the data displayed—the addition or deletion of an appointment, for example—or in the range of dates displayed as the user switches from month to month and year to year.

Whatever the action, WebCal uses the same function in mview.htm, called UpdateMonth, to refresh the calendar display. Clicking on the back arrow to move the calendar display back a month, for instance, invokes the prevMonth function:

```
function prevMonth()
{
	gcalfs.hideDetailPane();
	groot.setStatus("");
	groot.adjustDate( groot.gDay, 0, -1, 0, 0, 0, 0 );
	groot.gfCacheValid = false;
	updateMonth();
}
```

The function calls some housekeeping functions to clean up the screen—it hides the details frame, blanks the status line, sets the selected date back a month, and invalidates the cache. Then it calls UpdateMonth. UpdateMonth draws the month grid for the desired month, creating a table within each daycell that will later be used to add individual calendar entries. Each of these daycell tables is given a unique ID, which includes a YYYYMMDD representation of the cell's date. If necessary, it displays a Please Wait message while it calls updateMonthData to display the specific data associated with the retrieved XML data representing calendar entries.

UpdateMonthData, in turn, calls yet another function, checkData (in main.js), which creates an XMLControl object that abstracts the usage of the XML island for IE or iframe for Netscape 6 with the ID data:


```
if (!gXML) gXML = new XMLControl("data", window);
if (!gXML) {alert("No XML Control"); return;}
```

Next, it builds the ReadViewEntries URL command within a string variable named sURL:

```
sURL = gDBURL + "/($Calendar)?ReadViewEntries";
sURL += "&KeyType=time&StartKey=" + sDRStart + "&UntilKey=" + sDREnd + "&Count=" + iCacheSize;
```

Then it asks the XMLControl to retrieve the data using this constructed URL:

```
gXML.load( sURL );
```

The XMLControl and its subfunctions are grouped in dominodata.js. This code is split: There are duplicate functions for loading and parsing XML in an IE XML island (which takes advantage of the async property) and using a frame for Netscape 6.0. However, the prototype itself doesn't support Netscape 6.0. To see similar functionality in Netscape 6.0, see the xml_demo code.

Parsing the XML
When the data is loaded, XMLControl parses it into an array of named elements to make it available to the script that builds the calendar table. Again, the parsing code is browser-specific. Internet Explorer's XML island provides an XML DOM that is the equivalent of the HTML DOM. Netscape 6.0 does not, so the code parses the data using standard DOM methods and properties such as getElementsByTagName(...) and getAttribute(...).

IE's XML object model provides built-in methods for working with XML data that simplify the writing of code. The following example uses IE's selectSingleNode method if the browser is IE (the test is the variable glsIE) or the longer segment based on the HTML DOM method getElementsByTagName if the browser is Netscape.

```
XMLData.prototype.getItemData = function( i_row, s_columnName, i_offset ){
      // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      // return the text value of a row-cell
      // by programmatic name (columnName)
      // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      var row  = this.items.length? this.items.item(i_row) : this.items;
      var item;
      if (gIsIE)
            item = row.selectSingleNode("entrydata[@name='" + s_columnName +"']");
      else{
            var aEntry = row.getElementsByTagName("entrydata");
            for (var i=0; i< aEntry.length; i++){
            if (s_columnName == aEntry.item(i).getAttribute("name")){
            item = aEntry.item(i)break;
            }
      }
}
      if (item)
            return( getItemText(item, i_offset) )
      return "";
}
```

The second argument to the getItemData method above is the programmatic name for the column of the data you want to retrieve. We strongly recommend using programmatic names, rather than column numbers, to retrieve specific data items because if the view design is later modified in a way that changes the column numbering, the code will still function.

Updating the calendar table
When XMLControl returns the parsed data, the checkData function verifies the existence of the cached data and sets the status line to display the number of items loaded:

```
gfCacheValid = true;
      setStatus( "Entries loaded = " + gXML.getCount() );
```

Execution passes back to the function updateMonthData. This function iterates through each of the calendar

entries returned for the desired range and then extracts the relevant data associated with the entry and displays it within the HTML page, if the table cell representing the specific date where this entry should be displayed is present. When the updateMonthData has finished updating the HTML DOM to reflect the various calendar entries, it hides the Please Wait message and gives up control to IE so that all of the HTML DOM updates are displayed.

The value of using XML and DHTML

When you're developing Web applications that depend on data stored in Domino, XML is the universal data description language. Domino knows how to speak it and when to speak it, thanks to the ReadViewEntries URL command; and as we discussed above, the new generation of browsers know how to load and parse it. This, then, is all you need to implement view pages that allow viewing different portions of data without reloading the entire page. iNotes Web Access uses these mechanisms to allow users to traverse to different dates within a calendar view and as the mechanism for getting different chunks of data within its virtual list component.

An alternative way to update portions of HTML on the page would be to use XSL Transforms (XSLT). IE 5 and later offers a transformNode method that returns a string that results from transforming a specific node of the XML tree with an XSL stylesheet. Mozilla 1.0 also has an XSLT transformation module. XSLT is not presently used by iNotes Web Access and is a much more complex topic beyond the scope of this article.

XML and DHTML—JavaScript, style sheets, and the DOM—make it much easier for developers to create more sophisticated browser-based applications. There's no middleware, no conduits or connections to build or maintain. And because Domino is well integrated with other data stores by NotesPump and LEI and similar connectivity applications, ReadViewEntries can be used to reach beyond Domino to get data into a browser. Best of all, because you don't have to focus on the plumbing; you can devote your efforts to building the best possible user interface for your application.

ABOUT THE AUTHOR

Vinod Seraphin is a Senior Technical Staff Member and lead architect for iNotes Web Access. iNotes Web Access was "born" from Vinod's prototyping efforts to develop a very compelling Personal Information Manager (PIM) within a browser. Vinod was recently recognized with the IBM Outstanding Technical Achievement Award for his significant role on this project. He has been with Lotus/Iris/IBM for over 10 years. Prior to iNotes Web Access, Vinod was the Software Architect for Lotus Organizer. He has an MS in Computer Information Systems from Boston University and a BS in Computer Science and Engineering from MIT. He also is an avid professional sports fan and enjoys softball, skiing, traveling, and spending quality time with his family.

