Team Developer and XML

Unify Corporation
By Jean-Marc Gemperle
Senior Technical Support Engineers

Abstract

This technical white paper describes how to use XML within SQLWindows/Team Developer.

Introduction

XML, or eXtensible Markup Language, is a simple and flexible text format originally derived from SGML (Standard Generalized Markup Language). Initially, XML was designed to meet the challenges of large-scale electronic publishing; now XML is plays an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere.1

SQLWindows 1.5.0 was the first version to support XML in the form of a COM object with MSXML. Then SQLWindows 3.1 introduced the ability to move data between Table Windows and XML files. In SQLWindows 2005.1 developers now have three types of XML support: Moving Table Window data to or from XML documents; serializing and de-serializing UDVs (user-defined variables); and using the XML class library to work directly with the Document Object Model (DOM). All of these methods are independent from the others.

XML Table Window Support

The XML Table Window Support allows developers to read and write Table Window data to/from XML documents. In order to support this, there are four new functions added to SQLWindows:

SalTblWriteXMLAndSchema
This writes the contents of a table window to an XML file and/or an XML schema file.

Limitations
Note that if the table window contains a column with data type Date/Time, and a cell in that column is empty, then no content will be written for that particular XML document node. Validating XML parsers will flag this situation as an invalid value. The element names used in the XML file are based on the titles of the table window columns. Spaces are replaced with underscore characters. Note that if a table window contains more than one column with the same column title, then this function will fail.

SalTblWriteXMLAndSchemaEx
This writes the contents of a table window to an XML file and/or an XML schema file and selects rows with specific flags.

Limitations
Note that if the table window contains a column with data type Date/Time, and a cell in that column is empty, then no content will be written for that particular XML document node. Validating XML parsers will flag this situation as an invalid value. The element names used in the XML file are based on the titles of the table window columns. Spaces are replaced with underscore characters. Note that if a table window contains more than one column with the same column title, then this function will fail.

SalTblSetFromXMLSchema
For an automatic-column table window this function resets the columns in the table to comply with the schema. For a static-column table window this function returns a Boolean value indicating whether the attributes in the schema are a match for the attributes of the table window columns.

SalTblPopulateFromXML
Clears a table window and then fills new rows with data from an XML document. Optionally uses an XML schema to set attributes in the table. For a static-column table window this function returns a Boolean value indicating whether the attributes in the schema are a match for the attributes of the table window columns.

SalTblPopulateFromXML
This clears a table window and then fills new rows with data from an XML document. It optionally uses an XML schema to set attributes in the table.

Limitations
This function requires that XML documents and schemas conform to GUPTA's naming conventions and structures. It is not possible to populate a table from just any well-formed generalized XML document and schema. XML documents and schemas generated from the SalTblWriteXMLAndSchema function do conform to the necessary conventions and structures. To learn more about these conventions and structures use SalTblWriteXMLAndSchema against one of your table windows in a test application and examine the output files.

When bUseSchema is set to TRUE, and if the table window's columns were created at design time, the length of each column's value will be the length assigned at design time. If the table window has no design time columns and they are being automatically created during the call to this function, then the length of each column's value will be the length that is specified in the XML schema. This raises the possibility of data truncation if the value of an element contains more characters than are specified by the design-time length or the XML schema length.

XML Table Window Sample
In order to run this sample, make sure to run the Sample Installer. The application name is TableWindowXML.app and this demonstrates the use of these new functions. The sample uses quick tabs and each tab demonstrates different functionality. Below is a description of each tab:

Connect tab
This sample requires a connection to GUPTA SQLBase's Island database.

Generate XML From Table tab
This tab contains the table window with the data populated from the Island database. This allows the user to select the Row flags that they want. The user then has the option to generate only the Schema, the Document, or both. Once the XML Generation Options and Row Flags have been selected the user can then generate either the XML from the Table Window (SalTblWriteXMLandSchema), or the Extended XML from the Table Window (SalTblWriteXMLandSchemaEx). Once the XML has been generated the user can then view the XML document by pressing the **Show XML** pushbutton.

Populate Table From XML tab
This tab populates a Table Window from an XML file using the function SalTblPopulateFromXML.

Populate New Table From XML tab
This tab also populates a Table Window from an XML file using SalTblPopulateFromXML; however the table window appearance will be updated.

XML Serialization

SQLWindows allows you to write the contents of a user-defined variable (UDV) to an XML file, and to read an XML file's contents into the variables of a UDV. This ability is extremely helpful in preserving the state of an application. A UDV can be written to a file and then read back in a later session of the application.

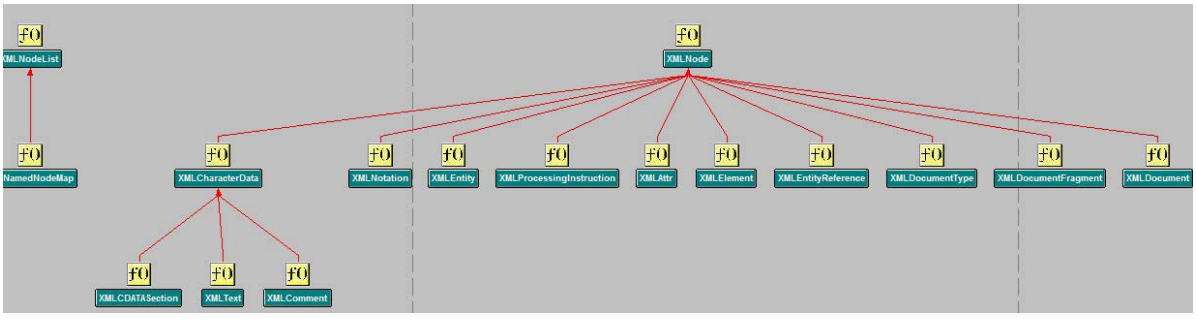
Serialization is limited to complete UDVs, not partial UDVs. If a UDV contains other UDVs nested inside it, only the outermost UDV can be serialized.

SQLWindows Document Object Model Class Library

The Document Object Model DOM is the main industry API standard for accessing XML documents. Many applications support or provide access to the DOM. In SQLWindow a Class Library called xmllib.apl, and based on the Xerces-C XML parser (see http://xml.apache.org), is provided to access XML Documents.

To add XML handling capability to your SQLWindows application you simply include the xmllib.apl library file to your application. Note that there are no external functions call definitions in xmllib.apl. All the methods of the classes are implemented in the runtime DLL CDLLI41.DLL. This means there are no extra specific DLLs for deployments.

XML support in SQLWindows is provided by a group of functional class definitions listed on the next page.



Class XMLNode
This is the abstract base class for all other XML classes, except for XMLNodeList and XMLNamedNodeMap, which are collections of XMLNode objects.

Class XMLDocument
This is the logical parent of all other XML class objects. All other objects must belong to a document. This class is responsible for the loading and creation of the XML document in memory (mapping) along with its parser settings. It allows the retrieval of the XML document root elements by retuning an Element object from which you can access the elements of your XML document. When the XMLDocument object is released all the objects attached to it are destroyed.

Class XMLElement
This class represents any XML element nodes and is derived from the XMLNode class. With it you can access all the elements of your XML document through the XMLNode.

Class XMLNodeList
This is a class that contains an array of the XMLNode type. XML methods that return a collection of nodes use this class to represent the collection. XMLNodeList is the base class for the XMLNamedNodeMap class.

Class XMLNamedNodeMap
This is derived from XMLNodeList, and adds the additional methods for Name and Attribute lookups.

All other classes deal with specifics of XML documents; i.e. XMLComment, XMLCDATASection, XMLCharacterData, XMLNotation, XMLProcessingInstruction, etc. See the XML Index in the Online Help for further details.

Basic XML Programming Tasks

The Class Library gives you full access to your XML documents. This means you can load a document and read it, modify it, or create a new XML document from scratch using the framework. Every time you need to perform such XML tasks, you create instances of the classes and call functions in those instance objects.
The first step in using XML is to map an XMLDocument object to the Document Object Model. For this, you first create an instance of the XMLDocument then you invoke from that instance the method Create or CreateNewDoc.
Window Variables:
XMLDocument: oXMLDocument
…

If NOT oXMLDocument.create ()

Error Handling

All methods in XMLLIB return a BOOLEAN. Always test the return value of the methods invoked; *never* assume the call is successful. The error description concerning XMLDocument input/output error can be retrieved simply by calling getLastError method from the XMLDocument object.

```
If NOT oXMLDocument.loadFromURI( "note.xml" )
Call oXMLDocument.getLastError( nError, sError )
Call SalMessageBox("Erro: " || SalNumberToStrX( nError, 0 ) || " " ||sError
,"Error",MB_Ok)
```

Setting Document or Parser Feature
Before loading and writing of XML documents you may want to alter the default setting for the document or the parser. The method setFeature() from XMLDocument allows you to do this. On the other hand, you can query the current feature settings by invoking getFeature(). Most of the feature set takes effect only after a document is loaded via loadFromSQL(), loadFromString() and loadFromURI(). Thus, you can typically invoke setFeature() before loading the document and after calling Create or createNewDoc() methods.

See the Online Help "*XML document feature constants*" for the type of settings available and their defaults.

Here we will focus on the XML_VALIDATE properties:

XML_VALIDATE _NEVER
This is the default; the parser will not validate the structure of the XML document loaded against a DTD (Document Type Definition) or XSD (XML Schema definition). Thus, it will ignore any validation error against DTD or the XSD definition that maybe associated to the XLM document.

XML_VALIDATE_ALWAYS
This loads the associated DTD or XSD definitions and verifies its conformance with the XML document that is loaded.

XML_VALIDATE_AUTO
A validation is performed. But if there any mismatches in the DTD or XSD definition the parser would still load the associated XML document assuming the document is well formatted.

Assuming the following XML document and XML Schema is loaded the following script should read:

Note.xml

```
<?xml
version="1.0"?><note="http://www.xyz.com":xsi="http://www.w3.org/2001/XM
LSchema-instance":schemaLocation="http://www.xyz.com note.xsd">
<to>TechSupport</to>
<from>Mile</from>
<heading>Reminder</heading>
<body>Don't forget my problem!</body>
</note>
```

Note.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.xyz.com" targetNamespace="http://www.xyz.com"
elementFormDefault="qualified">
        <xs:element name="note">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="to"
                    type="xs:string"/>
                    <xs:element name="from"
                    type="xs:string"/>
                    <xs:element name="heading"
                    type="xs:string"/>
                    <xs:element name="body"
                    type="xs:string"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
</xs:schema>
```

And the following is the SQLWindows code:

```
If NOT oXMLDocument.create( )
If NOT oXMLDocument.setFeature(XMLDOC_VALIDATE,
XML_VALIDATE_ALWAYS)
If NOT
oXMLDocument.getFeature(XMLDOC_VALIDATE,nRetrievedFeature)
If NOT oXMLDocument.setFeature( XMLDOC_NAMESPACE, TRUE )
If NOT oXMLDocument.setFeature( XMLDOC_DO_SCHEMA, TRUE )
If NOT oXMLDocument.loadFromURI( "note.xml" )
```

The document should then load flawlessly since the XML file is correctly formatted and complies with its associated XML Schema. If the order of sequence for the elements “to”, “from”, “heading” and “body” are not respected in the XML document as per definition of the associated Schema, then the document will fail to load returning FALSE in the method call If NOT oXMLDocument.loadFromURI("note.xml").

Creating a New Document

XMLDocument is the only class that provides input and output methods for XML data. It also contains functions that create other objects such as elements, attributes and text. You start by creating a new document using the object XMLDocument, createNewDoc() method and get the root element of the newly created document using getDocumentElement() from the XMLDocument object. At this point you can start to create new elements in the root using the XMLDocument object createElement() method which returns a newly created Element object. At this point the newly created object is still not in the document tree or in memory for inserting new elements when you call the method appendChild( ) from the desired XMLNode object. Further on, you can create attributes using XMLDocument object createAttribute() method and insert it using the setAttributeNode() method from the desired XMLElement object. Text insertion is similar in this process to the element insertion.

Please note: It is important to be aware that the parser validates a document only when it is loaded; thus it’s possible to write an invalid XML from an application which won’t be caught until it’s loaded again.

GUPTA Team Developer provides in its *\Samples\New Samples\XML\XML_Editor* directory an application called xml_editor_demo.app which allows you to create a series of XML nodes visually and writes the resulting XML document to a file so that you can examine the outcome. See the file xml_editor_demo.htm for additional information on that sample.

For the purpose of the paper we will provide another sample here. The following SQLWindows application will generate an XML document. And the purpose of this sample is to show you how to create a document and the steps involved in creating elements, attributes and text in desired nodes.

address.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Address>

 <name>
 <title>Mrs.</title>
 <first-name>Sue</first-name>
<last-name>Thomson</last-name>
 </name>

 <street>23rd street</street>

<city state="NY">New-York</city>

<postal-code>2312</postal-code>

</Address>
```

- Variables
  - XMLNode: oXMLNode_current
  - XMLDocument: oXMLDocument
  - XMLElement: oXMLElement_rootaddress
  - XMLElement: oXMLElement_name
  - XMLElement: oXMLElement_title
  - XMLElement: oXMLElement_firstname
  - XMLElement: oXMLElement_lastname
  - XMLElement: oXMLElement_street
  - XMLElement: oXMLElement_city
  - XMLElement: oXMLElement_postcode
  - XMLAttr: oXMLAttr_state
  - XMLText: oXMLText
  - Number: nError
  - String: sError

```
On SAM_AppStartup
  ! Create new XML document and set the root document ,  no namespace used
  If NOT  oXMLDocument.createNewDoc( " ,"Address" )
  ! Get the document root element
  If NOT oXMLDocument.getDocumentElement(oXMLElement_rootaddress)
  !
  ! Create new elements in Address root
  If NOT oXMLDocument.createElement( oXMLElement_name, "name" )
  If NOT oXMLDocument.createElement( oXMLElement_street, "street" )
  If NOT oXMLDocument.createElement( oXMLElement_city, "city" )
  If NOT oXMLDocument.createElement( oXMLElement_postcode, "postal-code" )
  Set oXMLNode_current=oXMLElement_rootaddress
  ! The object are created but are still not part of the document
  ! tree, we MUST call appendChild() to actully insert it
  If NOT oXMLNode_current.appendChild( oXMLElement_name )
  If NOT oXMLNode_current.appendChild( oXMLElement_street )
  If NOT oXMLNode_current.appendChild( oXMLElement_city)
  If NOT oXMLNode_current.appendChild( oXMLElement_postcode )
  !
  ! Insert Child element of name element, we set or node to name element.
  Set oXMLNode_current=oXMLElement_name
  If NOT oXMLDocument.createElement( oXMLElement_title, "title" )
  If NOT oXMLDocument.createElement( oXMLElement_firstname, "first-name" )
  If NOT oXMLDocument.createElement( oXMLElement_lastname, "last-name" )
  If NOT oXMLNode_current.appendChild( oXMLElement_title )
  If NOT oXMLNode_current.appendChild( oXMLElement_firstname )
  If NOT oXMLNode_current.appendChild( oXMLElement_lastname)
  !
  ! Create a state attribute
  If NOT oXMLDocument.createAttribute( oXMLAttr_state, "state" )
  If NOT oXMLElement_city.setAttributeNode(oXMLAttr_state)
  !
  ! Finaly create TextNodes for the different elements
  Set oXMLNode_current=oXMLElement_title
  If NOT oXMLDocument.createTextNode( oXMLText, "Mrs." )
  If NOT oXMLNode_current.appendChild( oXMLText )
  Set oXMLNode_current=oXMLElement_firstname
  If NOT oXMLDocument.createTextNode( oXMLText, "Sue" )
  If NOT oXMLNode_current.appendChild( oXMLText )
  Set oXMLNode_current=oXMLElement_lastname
  If NOT oXMLDocument.createTextNode( oXMLText, "Thomson" )
  If NOT oXMLNode_current.appendChild( oXMLText )
  Set oXMLNode_current=oXMLElement_street
  If NOT oXMLDocument.createTextNode( oXMLText, "23rd street" )
  If NOT oXMLNode_current.appendChild( oXMLText )
  Set oXMLNode_current=oXMLElement_city
  If NOT oXMLDocument.createTextNode( oXMLText, "New-York" )
  If NOT oXMLNode_current.appendChild( oXMLText )
  Set oXMLNode_current=oXMLAttr_state
  If NOT oXMLDocument.createTextNode( oXMLText, "NY" )
  If NOT oXMLNode_current.appendChild( oXMLText )
  Set oXMLNode_current=oXMLElement_postcode
  If NOT oXMLDocument.createTextNode( oXMLText, "2312" )
  If NOT oXMLNode_current.appendChild( oXMLText )
  ! We want to have a neat output, so let's a Document feature.
  If NOT oXMLDocument.setFeature( XMLDOC_FORMAT_PRETTY_PRINT,TRUE)
  If NOT oXMLDocument.writeToFile( "address.xml")
  ! Finaly realease the documnent from the memory
  Call oXMLDocument.release( )
```

Loading and Modifying Documents

Xmllib class XMLDocument provides three methods for reading data to populate the document and all its child objects in memory. If validation is on and the validation fails, an error is logged and the load is stopped where the error was encountered. In this case the document is partially loaded (incomplete). Error information can be retrieved by calling getLastError() from the document object.

loadFromURI()

This loads an XML document from the specified URI location. Team Developer provides an application called LoadXMLData.app in its sample directory *\Samples\New Samples\XML\ \LoadXMLData* which allows you to choose an XML file and then loads that file. The functions of the UDVs are used to "walk" through the document tree and fill an outline list box with text representing the nodes of the document. This sample would be generally useful for anyone who has a need to do a similar “walk-through” of an XML document.

See the file LoadXMLData.htm for additional details.

Nonetheless we will further discuss walking through XML documents with two samples.

Consider the following XML document:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Address>
 <name>
<title>Mrs.</title>
<first-name>Sue</first-name>
 <last-name>Thomson</last-name>
 </name>
<street>23rd street</street>
<city state="NY">New-York</city>
<postal-code>1111</postal-code>
 <name>
<title>Mr.</title>
<first-name>John</first-name>
 <last-name>Peterson</last-name>
</name>
<street>24rd street</street>
<city state="SF">San Francisco</city>
 <postal-code>2222</postal-code>
</Address>
```

This XML document as the same structure than the one we previously generated except that there is additional data: two names with there corresponding addresses. If we like to load such a document (and our goal is to extract all of the last names from that document along with the city and the attribute state), we would modify the document to make sure that the data is all in upper case. We can use a different yet similar way to achieve this.

First, it is important to understand that for accessing this information you need to start with the root element node. After loading the document use the XMLDocument.getDocumentElement(XMLElement) method. This method will return an XMLElement object. Recall that XMLElement is derived from XMLNode. Thus you can assign the returned root element object to an XMLNode object. At this point you can start searching for all child objects from the root node using XMLNode.childNodes(XMLNodeList). This method will return an XMLNodeList object as you will notice. With it you can start to navigate in the XMLNodeList collection using such methods as XMLNodeList.first(XMLNode), XMLNodeList.next(XMLNode), XMLNodeList.previous(XMLNode). These methods will return an XMLNode object that you can further use to check the node types such as Element, attributes or text (see the online help titled "*XML node type constants*"). If you would have Element or attribute returned you can drill down further to their child nodes and so on. If you have a text node you can start to extract the text node value.

So If you would extract all of the child nodes from the root Address in our address XML document, and query the node name and type using *XMLNode.nodeName()* and *XMLNode.nodeType(),* then this is the result you should achieve:

#text	XML_TEXT_NODE
name	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
street	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
city	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
postal-code	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
name	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
Street	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
city	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
postal-code	XML_ELEMENT_NODE
#text	XML_TEXT_NODE

Then further down on the node name we get:

#text	XML_TEXT_NODE
title	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
first-name	XML_ELEMENT_NODE
#text	XML_TEXT_NODE
last-name	XML_ELEMENT_NODE
#text	XML_TEXT_NODE

Finally, drilling down on the node last-name would return for you a XML_TEXT_NODE type in which you could extract its value using XMLNode.nodeValue() to retrieve that last-name value.

Below are two samples showing ways to achieve this in a SQLWindows application.

An **indirect way** is by parsing and searching for specific elements and attributes from the root node and drilling down on the Element child node and attributes found. The purpose of this sample is only to show the navigation on the nodes and child nodes. Here are the steps involved:

- Load the document.
- Search all the name element nodes from the root node and set them into an array. Then, from the name nodes retrieved in the array, search the last-name child element into another array to finally extract its value.
- Search all the city name element nodes of the document and its states attributes into arrays of nodes.
- Save and release the XML file as a new file.

- Application Description: Gupta SQLWindows Standard Application Template
  - Libraries
    - File Include: xmllib.apl
  - Global Declarations
    - Window Defaults
    - Formats
    - External Functions
    - Constants
    - Resources
    - Variables
      - XMLDocument: oXMLDocument
      - XMLElement: oXMLElement_rootaddress
      - XMLNodeList: oXMLNodeList
      - XMLNode: oXMLNode
      - XMLNode: oXMLNode_name[*]
      - XMLNode: oXMLNode_last_name[*]
      - XMLNode: oXMLNode_city[*]
      - XMLNode: oXMLNode_state[*]
      - Number: i
      - Number: nNodes
      - Number: nNodeType
      - Number: nError
      - String: sNodeName
      - String: sError
      - String: sValue

```
Application Actions
  On SAM_AppStartup
    ! Instantiates a document object and load the address XML doc
    If NOT  oXMLDocument.create( )
    If NOT oXMLDocument.loadFromURI( "address.xml" )
    If NOT oXMLDocument.getDocumentElement(oXMLElement_rootaddress)
    !
    ! We start to set our node to the root address
    Set oXMLNode=oXMLElement_rootaddress
    ! The function searches for name nodes and will return the number of nodes found
    ! initializing our dynamic array with the node found. Last Parameter FALSE
    ! is for processing elements, not attributes as there processing is slightly different than elements.
    Set nNodes=ParseNode(oXMLNode_name, oXMLNode, "name",FALSE)
    While i< nNodes
      ! we found some nodes name, we continue drilling down searching for last name element...
      Call ParseNode(oXMLNode_last_name, oXMLNode_name[i], "last-name",FALSE)
      ! The function will get and set the value of last name text node setting to UPPERCASE in the DOM mapping the value found for last_name
      Set sValue=GetSetNodeValue( oXMLNode_last_name[0], SalStrUpperX( GetSetNodeValue( oXMLNode_last_name[0], STRING_Null )) )
      ! To finaly display last-name value
      Call SalMessageBox(sValue,"Info",MB_Ok)
      Set i=i+1
    !
    ! We continue accessing other elements, like city
    Set i=0
    ! We start again to set our node to the root address
    Set oXMLNode=oXMLElement_rootaddress
    ! We search  the element city
    Set nNodes=ParseNode(oXMLNode_city, oXMLNode, "city", FALSE)
    While i< nNodes
      ! we found some nodes city, we extract the city value
      Set sValue=GetSetNodeValue( oXMLNode_city[i], SalStrUpperX( GetSetNodeValue( oXMLNode_city[i], STRING_Null )) )
      ! and also the state attribute
      Call ParseNode(oXMLNode_state,  oXMLNode_city[i], "state",TRUE)
      Set sValue=sValue || "    " || GetSetNodeValue( oXMLNode_state[0], SalStrUpperX( GetSetNodeValue( oXMLNode_state[0], STRING_Null )) )
      ! To finally display the city and the state
      Call SalMessageBox(sValue,"Info",MB_Ok)
      Set i=i+1
    !
    ! we want a neat XML file so we set the feature before writing
    If NOT oXMLDocument.setFeature( XMLDOC_FORMAT_PRETTY_PRINT,TRUE)
    If NOT oXMLDocument.writeToFile( "address1.xml")
    !
    ! We finally release the document mapping from DOM, thus freeing the document from the memory
    Call oXMLDocument.release( )
```

- Internal Functions
  - Function: ParseNode
    - Description: The function searches for name nodes and will return the number of nodes found
      initializing our dynamic array with the node found. Last Parameter FALSE
      is for processing elements, not attributes as there processing is slightly different than elements.
    - Returns
      - Number:
    - Parameters
      - XMLNode: pAoXMLNode[*]
      - XMLNode: poXMLNode
      - String: psNodeName
      - Boolean: bAttr
    - Static Variables
    - Local variables
      - XMLNodeList: oXMLNodeList
      - XMLNode: oXMLNode
      - XMLNamedNodeMap: oXMLNamedNodeMap
      - Number: nNodeType
      - String: sNodeName
      - Number: i
    - Actions
      - ! Do we process Attributes or Elements ?
      - If bAttr
        - ! Retrieves the list of attributes for this node
        - ! XMLNamedNodeMap is derived from XMLNodeList, so we get the first attribute...
        - ! In this example we expect only one attribute...
        - If poXMLNode.attributes(oXMLNamedNodeMap)
          - If oXMLNamedNodeMap.first(oXMLNode)
            - ! Usefull for checking if we are in the righ place and get what we expect...when debugging ....
            - Set nNodeType = oXMLNode.nodeType( )
            - Set sNodeName= oXMLNode.nodeName( )
            - ! We check we have an attribute node type that match our search
            - If nNodeType=XML_ATTRIBUTE_NODE AND sNodeName=psNodeName
              - ! We set the array with the node of that attribute for future reading/setting
              - Set pAoXMLNode[i]=oXMLNode
      - Else
        - ! We process element, notice the slight difference with the above...
        - If poXMLNode.childNodes(oXMLNodeList)
          - If oXMLNodeList.first(oXMLNode)
            - ! In this example there maybe more elements with the same search criteria...
            - While TRUE
              - Set nNodeType = oXMLNode.nodeType( )
              - Set sNodeName= oXMLNode.nodeName( )
              - If nNodeType=XML_ELEMENT_NODE AND sNodeName=psNodeName
                - Set pAoXMLNode[i]=oXMLNode
                - Set i=i+1
              - ! Are there further child? if not we exit
              - If NOT oXMLNodeList.next(oXMLNode)
                - Return i

```
Function: GetSetNodeValue
  Description: Get and Set the node value from a node element or attribute, if second parameter is <> than STRING_Null
  Returns
    String:
  Parameters
    XMLNode: poXMLNode
    String: strSet
  Static Variables
  Local variables
    XMLNodeList: oXMLNodeList
    XMLNode: oXMLNode
    Number: nNodeType
  Actions
    ! We are on an element or attribue node, we check is there are some text node child...
    If poXMLNode.childNodes( oXMLNodeList )
      If oXMLNodeList.first( oXMLNode )
        While TRUE
          ! Usefull  for checking if we are in the righ place and get what we expect...when debugging  ....
          ! For text node type the node name is #text...
          Set nNodeType = oXMLNode.nodeType(  )
          Set sNodeName= oXMLNode.nodeName(  )
          ! Do we have a text node?
          If nNodeType=XML_TEXT_NODE
            ! Do we need to set a new value?
            If strSet
              Call oXMLNode.setNodeValue( strSet )
            ! we return the value
            Return oXMLNode.nodeValue(  )
          ! Are there further child? if not we exit
          If NOT oXMLNodeList.next( oXMLNode )
            Break
```

A more **simpler** & **direct way** is by parsing and searching for specific elements and attributes from the root node, but accessing directly the specific Element node using the XMLDocument.getElementsByTagName() method that returns a collection of XMLNodeList containing our searched Tag Name. In this case we rely on the XMLNodeList to retrieve our child node, and therefore we will not need an array. This sample does not modify the loaded document; it just shows another way to navigate within our node and child nodes:

- Load the document.
- Retrieve the last-name node element directly from the root element and extract its child text node to retrieve its value.
- Search the city name element nodes directly from the root element and extract its child text node to retrieve its value. On the same node retrieve the child state attribute.
- Releasing the XML document.

```
Variables
  XMLDocument: oXMLDocument
  XMLNode: oXMLNode
  XMLNodeList: oXMLNodeList
  XMLNamedNodeMap: oXMLNamedNodeMap
  XMLElement: oXMLElement_rootaddress
  XMLNode: oXMLNode_temp
  Number: nNodeType
  Number: nError
  String: sNodeName
  String: sError
  String: sValue
Internal Functions
Named Menus
Class Definitions
Application Actions
  On SAM_AppStartup
    ! Instantiates a document object and load the address XML doc
    If NOT  oXMLDocument.create( )
    If NOT oXMLDocument.loadFromURI( "address.xml" )
    If NOT oXMLDocument.getDocumentElement(oXMLElement_rootaddress)
    !
    ! We start to set our node to the root address
    Set oXMLNode=oXMLElement_rootaddress
    ! We get a collection of XMLNodeList for our Tag Name last-name
    If oXMLDocument.getElementsByTagName( oXMLNodeList, "last-name" )
      ! We go through the list
      If oXMLNodeList.first( oXMLNode )
        While TRUE
          ! Usefull  for checking if we are in the righ place and get what we expect...when debugging  ....
          Set nNodeType = oXMLNode.nodeType( )
          Set sNodeName= oXMLNode.nodeName( )
          ! We know there is only a TEXT node beneath
          If oXMLNode.firstChild( oXMLNode )
            ! We get its value and display it
            Set sValue=oXMLNode.nodeValue( )
            Call SalMessageBox(sValue,"Info",MB_Ok)
          ! and continue for the next node in the lost of last-name nodes
          If NOT oXMLNodeList.next( oXMLNode )
            Break
      ! We get a collection of XMLNodeList for our Tag Name "city"
    If oXMLDocument.getElementsByTagName( oXMLNodeList, "city" )
      If oXMLNodeList.first( oXMLNode )
        While TRUE
          ! Usefull  for checking if we are in the righ place and get what we expect...when debugging  ....
          Set nNodeType = oXMLNode.nodeType( )
          Set sNodeName= oXMLNode.nodeName( )
          ! We know there is only a TEXT node beneath
          ! But we use another vriable to retrieve the first child as we know there is also an attribute
          ! See Avoiding Internal Mapping Side Effects on DEVPDF p 19-7 for more details
          If oXMLNode.firstChild( oXMLNode_temp )
            Set sValue=oXMLNode_temp.nodeValue( )
          ! the current oXMLNode object is on the city node, we know there is an attribute there
          If oXMLNode.attributes( oXMLNamedNodeMap )
            ! We get a collection of attributes and get the first one, we know there is only one...
            If oXMLNamedNodeMap.first( oXMLNode )
              If oXMLNamedNodeMap.getNamedItem( "state", oXMLNode )
                Set sValue= sValue || "     " || oXMLNode.nodeValue( )
            ! And we display both the City and the state value
          Call SalMessageBox(sValue,"Info",MB_Ok)
          If NOT oXMLNodeList.next( oXMLNode )
            Break
    Call oXMLDocument.release( )
```

loadFromString()

Same as above, except that the location of the XML document is in a SAL string. Note that currently there is no direct way to write the document in memory to a string.

loadFromSQL()
This loads an XML document using the supplied SQL query. The data is loaded as children of the root element of that document. The last parameter of this function (bCreateAsElement) allows you to create separate elements for every column fetched from the result set or attribute instead. A similar function (XMLElement.loadFromSQL()) works the same way as this function does, but it loads the data as children of a designated element in the document, not necessarily the root element.
Team Developer provides an application called LoadXMLData.app in its sample directory *\Samples\New Samples\XML\ \CreateXMLData* which allows you to designate an XML file and writes to that file with the result sets of SQL queries. This sample would be generally useful for anyone who has a need to construct XML documents from SQL queries.
See LoadXMLData.htm for additional information on that sample.

Casting

As in any other framework *casting* is sometimes required to convert object types to another type. In the case of the XMLLIB framework do not expect the cast method to transform one kind of node into another. This would not work. All classes in the framework contain casting methods, except for XMLNode, XMLNodeList, XMLNamedNodeMap and XMLCharacterData. This is because it is a base class for all other text data classes.

Many XML operations involve populating a collection of objects whose type is XMLNode (the abstract base class). All methods returning more than one object use either the XMLNodeList or XMLNamedNodeMap types.

Thus, it is assumed that you have an XMLNodeList object containing many XMLNode objects and that you retrieve its first node calling XMLNodeList.first() with the intention to work on the “element level” of this retrieved node. This is possible only if you have an instance of an XMLElement on which you would cast the retrieved XMLNode such as the following:

bOk = myWorkElement.castToElement(myRetrievedNode)

Conclusion

The use of Table Window XML is fairly effortless and allows the user to easily generate XML files.

XML Serialization is relatively straightforward, although the developer needs to be aware of some of the limitations.

The use of the XMLLIB framework to access or create XLM documents is relatively simple in its usage, but there are areas on wish you need to be extra careful:

Memory mapping
SAL UDVs are mapped to internal objects created through Xerces. Mapping occurs through calls such as XMLDocument.create() and XMLDocument.createElement(). It is legal but does not make any sense to call UDV functions before the object is mapped. If you fail to call an XMLDocument.release() before the SAL object goes out of scope all the internal Xerces objects will remain in memory, thus a possible memory leak will occur. If two or more XML UDVs reference each other the internal Xerces mapping can become inadvertently damaged .

Even after an XML UDV is mapped to a Xerces internal object it may not be part of the document tree until an additional call (such as XMLElement.appendChild) is positioned in that tree.

Validation
Xerces only validates a document when it is loaded through a call like XMLDocument.loadFromURI. Thus, it is possible to create and write invalid XML documents from an application.

With all this in mind you should be able to successfully integrate XML processing in your Team Developer applications.

About Unify

Unify is a global provider of software development technology and solutions that helps IT customers participate in Service-Oriented Architecture (SOA). Unify's productive and re-liable development tools, migration solutions and databases enable organizations to build and modernize business essential applications for SOA. Composer for Lotus Notes offers a complete, like for like, production to production migration solution for Lotus Notes applications. Unify's award-winning NXJ Developer enables IT teams to be extremely productive, learn new technologies fast and deliver Web services-based applications on time and on budget. The Team Developer, SQLBase, DataServer, ACCELL and VISION product families enable cross-platform rapid development on Java/J2EE, Linux or Windows. Unify has a rich heritage in delivering rich, cost-effective technologies to its thousands of IT customers and ISV, VAR and distributor partners. Unify is headquartered in Sacramento, Calif., and can be reached at (916) 928-6400 or by visiting www.unify.com.

Unify Corporation
2101 Arena Blvd., Suite 100
Sacramento, CA 95834
USA
Phone: 1.916.928.6400
Toll Free: 1.800.468.6439
Fax: 1.916.928.6404
Munich: +49 8 115 55430
United Kingdom: +44 (0)1753 245 510
France: +33 (0)1 34 58 28 30