XML Serialization and De-serialization

By Daniel Tin
Field Sales Engineer

November, 2005

GUPTA™

Abstract 3
Introduction 3
XML Serialization 4
    Parameters 4
    Return Value 4
XML De-serialization 5
    Parameters 5
    Return Value 5
    New Outline Option 6
Error Handling Within Serialization And De-Serialization 6
Error Handling Within Serialization And De-Serialization 7
    Parameters 7
    Return Value 7
    Other Error Sources 7
XML Class Library Overview 8
    Basic Techniques 8
    Using Casting 9
    Avoid Problems When Using XML UDVs 10
    A UDV's State Impacts Its Usefulness 11
    Beware of Memory Leaks 12
    Avoiding Internal Mapping Side Effects 12
Team Developer Online Help 13

Abstract

This white paper describes the usability of XML serialization and XML de-serialization and how it works in Team Developer 2005.1

Introduction

Good (and essential) programming practice involves calling *SalXMLGetLastError* after any XML function call which may in theory cause an error

Serialization and de-serialization can be used as a mechanism to simplify persistency. For example, the ability to suspend transactions at the end of a day and restart them again the next morning by serializing the objects and saving them in an XML document. To resume the transactions the objects can be reconstituted from the serialized data.

XML serialization describes the process of converting the status of an object into a form which can then be kept or transported. The complement to XML serialization is XML de-serialization, to which objects can be reconstituted from the serialized data. On the basis of these two procedures, data can be stored and transmitted in an easy and problem-free way.

XML serialization allows for the conversion of user defined variables (UDV) instance variable into XML. Results of the XML serialization are classes with strict connection types that have public characteristics, and fields which are converted for the purpose of storage and transmission into an XML file.

Since XML is an open standard the XML file can be processed platform-independent from arbitrary applications.

XML Serialization

Team Developer 2005.1 allows you to write the contents of a 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.

The serializations of UDVs are handled by the SAL function:

SalXMLSerializeUDV(sUDV, sFilename, sWriteStyle)

Parameters

sUDV	Object: the UDV to be serialized.
sFilename	String: the path and name of the file to contain the serialized XML. If the file already exists it is overwritten without warning.
sWriteStyle	Number: a constant indicating whether the data, the schema, or both are to be written.

Return Value

Boolean: TRUE if successful

XML Constant	Value	Description
XML_DocAndSchema	0	Causes both the XML document and the schema files to be written
XML_Document	1	Causes only the XML document to be written
XML_Schema	2	Causes only the XML schema to be written

An error can occur midway through serializing or de-serializing leaving you with incomplete results – this can be potentially dangerous

The function SalXMLSerializeUDV() is able to handle complete UDVs (no partial UDVs). This means that the declared object, in the first parameter of this function, can't be an instance variable within another UDV. It must be declared as a window variable, global variable or local variable. Because UDVs are nested within each other, you will receive an error if you attempt to serialize partial UDVs. Therefore you should always serialize the outermost UDV.

The UDV name that was referenced in the first parameter will be the name of the root element in the output XML document, and the class definition of the UDV is included as an attribute (class="TestClass"). The names of other elements are equal to the names of the class variables and instance variables in the UDV. Arrays are defined in an element whose name is the same as the name of the array, and a series of elements named "arrayEntry", one for each array element, will have the attributes "index=0", "index=1", etc.

The UDVs can contain any mixture of primitive data types like number, string or date/time, and of course nested UDVs as instance variables. But remember, class variables are not serialized, only instance variables.

All levels of nesting will be represented in the output XML file. Whether or not a specific variable is included in the XML output is determined at design time when you edit the UDVs class definition to set the *serialize* property to TRUE or FALSE. Additionally, the built-in SAL primitive data types are less specialized than those which are supported by XML (i.e. is the declared SAL *number* variable an integer, a float, a decimal, or a double?). Because of this, you should specify the specific XML data type to be associated with each primitive variable that will be serialized during design time.

Errors are *silent* and do not interrupt the application

The XML data type and the serialize property can be set through Team Developer's Coding Assistant, Active Coding Assistant, or the right-click context menu, when focus is on an instance variable in a class definition. If an error occurs, this function will return FALSE. You can then call SalXMLGetLastError() to read the text of the error.

XML De-serialization

Team Developer 2005.1 not only allows you to write the contents of a UDV to an XML file but also vice versa. Team Developer 2005.1 allows you 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 XML file and then read back in a later session of the application

The de-serialization of UDVs is handled by the following SAL function:

SalXMLDeserializeUDV(sUDV, sFilename, sSchema)

Parameters

sUDV — Object: the UDV to receive the XML file contents.

sFilename — String: the path and name of the file that contains the serialized XML.

sSchema — String: the path and name of the file containing the schema that describes the XML file (optional).

Return Value

Boolean: TRUE if successful.

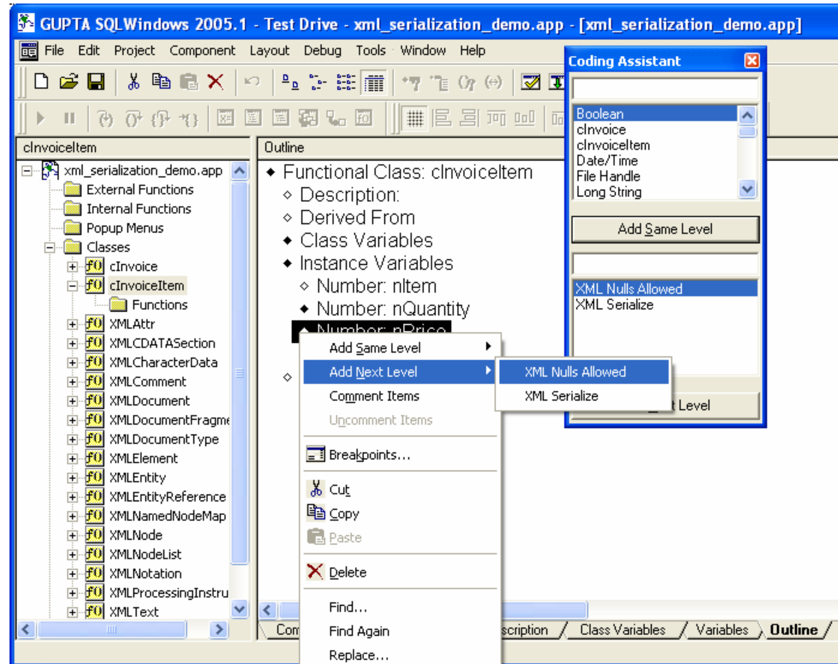
XML Constant	Value	Description
XML_DocAndSchema	0	Causes both the XML document and the schema files to be written
XML_Document	1	Causes only the XML document to be written
XML_Schema	2	Causes only the XML schema to be written

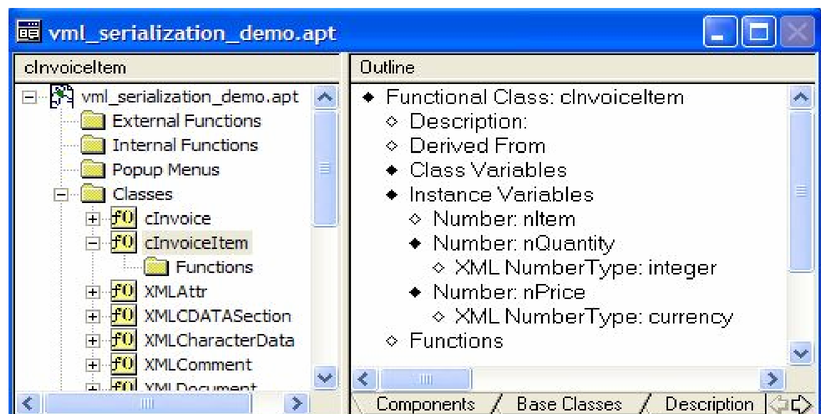
The function SalDeserializeUDV() is responsible for loading the contents of an XML file into a UDV. The UDV should be an exact match for the structure of the XML file. The easiest way to ensure that the structure matches is to create the XML file by initially calling SalXMLSerializeUDV(). If the parameter sUDV does not evaluate at runtime to a UDV or an array that is in scope, an error will occur. If such an error occurs, this function will return FALSE. You can then call SalXMLGetLastError() to read the text of the error.

New Outline Option

For instance variables of a class, you can use the Coding Assistant or right-click to specify the following:

- **XML Serialize:** If *No* then this instance variable will not participate in serializing or de-serializing. The default value is: *Yes*
- **XML Data Type:** More precise than SAL. For example *currency* instead of just *number*. The default value is: *None*
- **XML Nulls Allowed:** If *No* then an error is thrown when de-serializing an empty XML element or attribute. The default value is: *Yes*





Error Handling Within Serialization And De-Serialization

Errors are *silent* and do not interrupt the application

From the previous description we now know that errors are silent and that they won't interrupt our application. The worst case is that an error can occur during serialization or de-serialization and because of this your application leaves you with incomplete results. This can cause potential danger during developing, and last but not least during bug hunting. It is safer to assume that something has going wrong.

You can prevent such potential danger by calling the function SalXMLGetLastError when calling any XML-related function in Team Developer 2005.1. The function SalXMLGetLastError is able to retrieve the last error code and the appropriate error description.

***SalXMLGetLastError* is also used by the XML class library to monitor errors**

SalXMLGetLastError(nErrorCode, sErrorText)

Parameters

nErrorCode	Receive Number: The code returned by the XML parser.
sErrorText	Receive String: Description of the error condition.

Return Value

Boolean: TRUE if successful.

Other Error Sources

To give you a short insight whereby other errors during serialization or de-serialization can occur, just imagine the following scenario:

> You have serialized a UDV into a XML file. If you then edit the output *.xml file and change, for example, the tag name, the application runs and there will occur no error.

What will happen is that the dependant field in your application will be empty. This illustrates that XML values are written to UDV instance variables based on an exact match between the XML tag name and the instance variable name. If you change the first character of the tag name from lowercase to uppercase in the XML file so the match is no longer exact then the value in that tag will be ignored.

Although very important, error handling is sometimes forgotten

Another scenario is to edit the output *.xml file and change a number value from 399.99 to *Test*. The application runs and you will get no error message. To avoid this behavior you should place a message box in your application to make sure that the value *Test* could not be assigned to a numeric field. The XML processing in Team Developer will normally *not* pop up any error message boxes. Error conditions are made available to the programmer by calling SalXMLGetLastError().

An error during XML processing will not interrupt your application, and this could be a potential hazard that you must be prepared for. When de-serializing an XML document many instance variables may be populated successfully before an error occurs.

You are then left with a partially populated UDV and this is why it is important to monitor the error status after a call to a de-serializing function.

You should be aware of the XML class library

XML Class Library Overview

To provide your Team Developer 2005.1 application XML handling capability you must include the library file xmllib.apl. XML support in Team Developer 2005.1 is provided by a group of functional class definitions that you should be aware of.

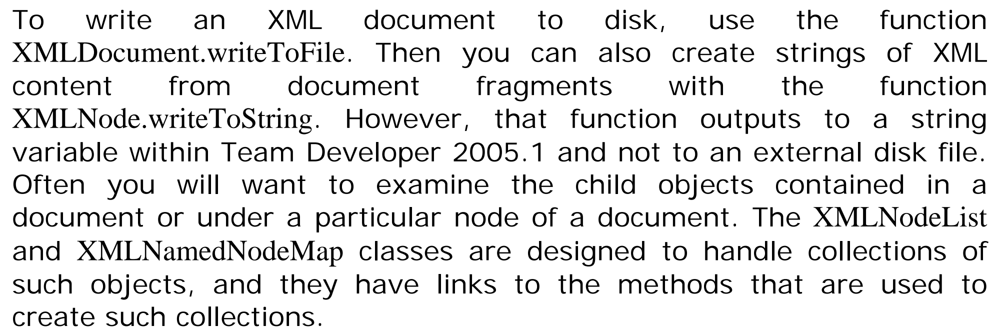
When you need to perform XML tasks you must create instances of the classes and call functions in those instance objects. The class XMLNode is the parent of nearly all other classes, except for XMLNodeList and XMLNamedNodeMap which contain collections of XMLNode objects. A large number of constants are contained in xmllib.apl to support specific methods within these classes.

Basic Techniques

The first step to using XML is to map an XMLDocument object to the Document Object Model (DOM). To do this, call up the functions create or createNewDoc in that class. 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* and *attributes*. All of the internal objects created through calls to a document's function will persist in memory until the application exits or unless you call the *release* function from the document. The release function will free up the document itself and all its owned objects, so therefore it is imperative to call the release function when you are finished with a document, or your application may encounter memory leak issues.

Reading data in through the loadFromSQL, loadFromString, or the loadFromURI methods in XMLDocument will populate the document and all of its child objects. Other objects can be initiated by calls to XMLDocument methods such as createElement and createAttribute. It is important to note that simply creating such an object doesn't make it a part of the document for purposes of parsing, i.e. you can create an element using a call to function createElement, but it doesn't become a part of the document tree until you reference the element in a call to XMLNode.appendChild, XMLNode.insertBefore or XMLNode.replaceChild.

You can change some of the behaviors in a document by using the setFeature function. It is important to note that many features don't take immediate effect in a document. The features are related to the loading and writing of documents, and thus they are dependent on when you perform those operations. For best results you should call setFeature after calling create or createNewDoc, and before calling loadFromSQL, loadFromString, or loadFromURI.


To write an XML document to disk, use the function XMLDocument.writeToFile. Then you can also create strings of XML content from document fragments with the function XMLNode.writeToString. However, that function outputs to a string variable within Team Developer 2005.1 and not to an external disk file. Often you will want to examine the child objects contained in a document or under a particular node of a document. The XMLNodeList and XMLNamedNodeMap classes are designed to handle collections of such objects, and they have links to the methods that are used to create such collections.

Using Casting

Many of the XML classes contain functions with the word "cast". The reason that such functions exist is because many XML operations involve populating a collection of objects. Those objects are always a type of XMLNode. When you make a collection of elements using XMLDocument.getElementsByTagName, the collection is stored in the first parameter of that function which is data type XMLNodeList.

That means that every function that collects more than one object uses either XMLNodeList, or its subclass XMLNamedNodeMap, to hold that collection. The objects inside that collection are always instances of the class XMLNode. Now suppose you want to perform some work on the collection and that work is related to elements? You can't simply retrieve one of the XMLNode objects and call XMLElement.getAttribute() on it because that object is not an instance of XMLElement. Therefore, you must create a UDV which is an instance of XMLElement and then initialize that UDV with the information contained in the XMLNode object, like the following example:

```
XMLElement: TestWorkElement
XMLNodeList: theList
XMLNode: TestRetrievedNode

bOk = TestDocument.getElementsByTagName(theList, "customer")
bOk = theList.first(TestRetrievedNode)
bOk = TestWorkElement.castToElement(TestRetrievedNode)
```

This will allow you to make XMLElement calls from TestWorkElement which will be successful.

Casting won't transform one kind of node into another. In the example above, the object in TestRetrievedNode was originally an XMLElement, and we are simply casting it back to its original type to perform work upon it.

Avoid Problems When Using XML UDVs

To avoid unexpected results in your applications which you should be aware of, UDVs, created from XML classes, are mapped to internal objects at runtime in the DOM itself. This mapping technique allows XML UDVs to exist in various states.

To get an understanding of the possible states of XML UDVs please refer to the following example:

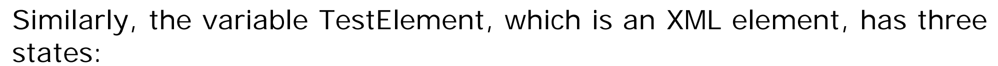
```
Function: fTestFunc
        Description:
        Returns
        Parameters
        Static Variables
        Local variables
          XMLDocument: TestDocument
          XMLElement: TestElement
          XMLElement: rootElement
        Actions
            ! at this point, TestDocument and TestElement exist, but they are
            not mapped to internal DOM objects
        Call TestDocument.create( )
          ! now TestDocument is mapped: a legal, but empty, XML document
        Call TestDocument.loadFromURI( 'file//usr/local/TestFile.xml' )
          ! now TestDocument actually contains XML data
        Call TestDocument.createElement(TestElement 'TestElementName' )
          ! TestElement is now mapped, but it is not yet part of the document tree
            Of TestDocument.
        Call TestDocument.getDocumentElement( rootElement )
        Call rootElement.appendChild(TestElement )
          ! Now TestElement is actually part of the document tree, the
          last child of the root element.
```

If we follow the UDV variable TestDocument in this example we can see that it has three states:

- **First state:**
  The variable TestDocument exists because it is in scope when the function is called.

- **Second state:**
  The variable TestDocument is mapped to an internal DOM object by the calling create(). Now it is a proper XML document (although empty).

- **Third state:**
  The variable TestDocument is loaded with data through a call to one of the load functions.



Similarly, the variable TestElement, which is an XML element, has three states:

- **First state:**
  The variable TestElement exists because it is in scope when the function is called.

- **Second state:**
  The variable TestElement is mapped to an internal DOM object through the call to createElement. At this point the variable TestElement is a proper XML UDV and many function calls made against this UDV will succeed, but it is not yet part of the tree structure of the document, instead it is an *orphan*.

- **Third state:**
  The variable TestElement becomes part of the document tree through the call to appendChild.

A UDV's State Impacts Its Usefulness

When these UDVs are in their first state they are useless for any practical XML purpose because in this state they are not yet mapped to an internal DOM object. Therefore, all of the UDVs class functions will fail when called. A useful way of determining whether a UDV is in its first state is to call XMLNode.NodeType() against it. Watch now for a return value of zero.

When UDVs are in their second state, and now mapped to an internal DOM object, most of their class functions will succeed. In the example above we used a createElement call to associate the UDV TestElement with an internal DOM object. At this point we could add attributes to the TestElement and those function calls now will succeed.

But because the UDV TestElement is not a part of the document tree yet, the contents of the UDV TestElement and its attributes would not be included in the output if you write the document to a file. Because of this it is often necessary that XML UDVs reach their third state before they are of any practical value in the application. That means for the UDV TestElement to be a part of the document tree you will need to call appendChild(), or calling either of the two similar functions insertBefore() or replaceChild().

Memory leaks

Beware of Memory Leaks

In the above scenario, with the example fTestFunc, we have illustrated potential dangers for applications that use XML UDVs because these UDVs are mapped to internal DOM objects. But what happens with these mapped UDVs when the function fTestFunc ends? The XML UDVs are declared as local variables and they will lose their scope and their mapping to internal DOM objects. Yet the internal DOM objects persist in memory.

Obviously, if the function fTestFunc is called multiple times the accumulation of unmapped internal DOM objects can be a potential danger. Therefore, it is very important that you call the release function in a document, when you are finished using it, and certainly before its UDVs loses its scope. It is easy to locate a document from any of the other UDVs. Nearly every class inherits from the class XMLNode, and XMLNode has a function called ownerDocument that will provide a link to that document.

If you are certain that you are finished with a node you can also call the release function of the class XMLNode to free just that individual node from memory. Once you have done this all other function calls against that UDV will return FALSE. The node expected to be successfully released, if it is in the document tree, can't be released so it must be an orphan. Nodes can become orphans by calling the functions removeChild, removeAttribute, or replaceChild. Nodes are also orphans when they are in the second state as described above.

Avoiding Internal Mapping Side Effects

Much more importantly, and where special caution is needed, is where you have two or more UDVs mapped to the same internal DOM object. The mapping itself behaves like an instance variable because it can be changed and it is the same as with an instance data. A change in the mapping affects all the UDVs associated with the internal DOM object.

If you wish to add a new element, such as a child, to the root element of a document follow this example:

```
Local Variables
  XMLNode: TestNode
  XMLElement: TestElement
Actions
  Set bOk = TestDocument.getDocumentElement(TestElement)
  Set TestNode = TestElement
```

Now both the elements TestNode and TestElement are mapped to the same internal DOM object (the root element of the document) and we now need to create a new element so that we can add it. To do this we need to have an instance of XMLNode available so that we can call appendChild() against it.

```
Set bOK = TestDocument.createElement(TestElement,
'someElementName')
```

With this function call we have just changed the DOM mapping of TestElement from the root element of the document to the newly created element.

```
Call TestNode.appendChild(TestElement)
```

The TestNode was originally associated with the root element of the document, but when the UDV TestElement changes its mapping it causes TestNode to change its mapping also. Now TestNode also points to the newly created element. This means that the appendChild() call will attempt to add TestElement to itself as a child which will cause the function to fail and an error to be logged. You can see that having two UDVs mapped to the same internal DOM object can be dangerous if you call a function against one of the UDVs which has changed the mapping.

Is there a way to avoid this danger? Yes, of course! Obviously you can use a lot of different UDVs in an attempt to avoid ever having two UDVs mapped to the same DOM object. You also can use the *new* keyword to erase the mapping in one UDV without affecting the mapping of the other:

```
Set TestElement = new XMLElement
Set bOK = TestDocument.createElement(TestElement,
'someElementName')
```

At this point TestNode will retain its original mapping to the document's root element while TestElement will refer to the newly created element. Then the appendChild() will work properly.

Online Help

Team Developer Online Help

You can also refer to Team Developer's Online Help for specific XML functions that will indicate when a function is the particular type that causes a change in the DOM object mapping.

Copyright © 2005 Gupta Technologies LLC. GUPTA, the GUPTA logo, and all GUPTA products are licensed or registered trademarks of Gupta Technologies, LLC. All other products are trademarks or registered trademarks of their respective owners. All rights reserved.