Debugging XML with Export

By David Adams

Technical Note 06-41

Overview

4th Dimension's DOM (Document Object Model) XML commands support reading and writing valid XML. While it's natural to think of XML as text, the DOM commands treat XML as a structured tree of linked nodes. Each node has a name, may have a value, and may have attributes. Ultimately, the XML tree can be converted to text but, internally, the commands always work with a tree. Within the 4th Dimension environment, an XML tree is largely invisible until it is converted to a finished XML document. As anyone who has struggled to create XML knows, it is hard to debug something you can't see. When creating XML with the DOM commands, writing the contents of the tree out for inspection with **DOM EXPORT TO VAR** or **DOM EXPORT TO FILE** can ease debugging. Additionally, using the **DOM EXPORT** commands is a great way to learn more about what the various DOM commands do.

Creating XML with the DOM Commands

Before looking at the DOM export commands, we'll review a few lines of code that produce some simple XML:

```
C_STRING(16;$root_xmlref)
C_STRING(16;$businessPhone_xmlref)
C_STRING(16;$homePhone_xmlref)
$root_xmlref:=DOM Create XML Ref("contact")
$businessPhone_xmlref:=DOM Create XML element($root_xmlref;"/contact/business/phone")
$homePhone_xmlref:=DOM Create XML element($root_xmlref;"/contact/home/phone")
DOM SET XML ELEMENT VALUE($businessPhone_xmlref;"123 456 789")
DOM SET XML ELEMENT VALUE($homePhone_xmlref;"123 888 999")
DOM CLOSE XML($root_xmlref)
```

If you already know how 4th Dimension's XML commands work, it's easy enough to guess what the code shown above does. Otherwise, the code can look at bit dense. This may be a familiar sensation for developers who have used the 4D 2004 Web Service Wizard to create SOAP proxy methods for complex services. Below is a fragment that illustrates the type of XML production code the Web Service Wizard creates:

```
C_STRING(16;$subelem)
C_TEXT($namespace)
$namespace:="http://www.4d.com/namespace/default"
$root:=DOM Create XML Ref("SendValveSettings";$namespace)

$subelem:=DOM Create XML element($root;"/SendValveSettings/Temperature")
DOM SET XML ELEMENT VALUE ($subelem;$1)

$subelem:=DOM Create XML element($root;"/SendValveSettings/Pressure")
DOM SET XML ELEMENT VALUE ($subelem;$2)

$subelem:=DOM Create XML element($root;"/SendValveSettings/Vector_One/item")
DOM SET XML ELEMENT VALUE ($subelem;$3)

$subelem:=DOM Create XML element($root;"/SendValveSettings/Vector_Two/item")
DOM SET XML ELEMENT VALUE($subelem;$4)
```

If there is anything wrong with the XML produced by proxy code like this, it can be difficult to track down the problem, unless the XML is converted to text for inspection.

"Serializing" Incomplete Trees

The process of converting an XML tree into text is often called "serializing" the XML. The **DOM EXPORT TO VAR** and **DOM EXPORT TO FILE** commands each perform this task, differing only in where the XML output is directed. For this discussion, we'll export XML to a text variable, although **DOM EXPORT TO VAR** also supports exporting to BLOBs. Now we'll rewrite the first example to help inspect the contents of the XML tree at each step (whitespace in the code listing has been adjusted for legibility):

```
C_STRING(16;$root_xmlref)
C_STRING(16;$businessPhone_xmlref)
C_STRING(16;$homePhone_xmlref)

$root_xmlref:=DOM Create XML Ref("contact")
DOM_DumpTreeToClipboard ($root_xmlref;"DOM Create XML Ref('contact')";True)

$businessPhone_xmlref:=DOM Create XML element($root_xmlref;"/contact/business/phone")
DOM_DumpTreeToClipboard ($root_xmlref;"DOM Create XML Ref('contact/business/phone')")

$homePhone_xmlref:=DOM Create XML element($root_xmlref;"/contact/home/phone")
DOM_DumpTreeToClipboard ($root_xmlref;"DOM Create XML element('/contact/home/phone')")

DOM SET XML ELEMENT VALUE($businessPhone_xmlref;"123 456 789")
DOM_DumpTreeToClipboard
    ($root_xmlref;"DOM SET XML ELEMENT VALUE($businessPhone_xmlref;'123 456 789')")

DOM SET XML ELEMENT VALUE($homePhone_xmlref;"123 888 999")
DOM_DumpTreeToClipboard
    ($root_xmlref;"DOM SET XML ELEMENT VALUE($homePhone_xmlref;'123 888 999')")

DOM CLOSE XML($root_xmlref)
```

The sample code shown above calls the *DOM_DumpTreeToClipboard* routine five times. Each call includes an XML reference and some diagnostic information used as a header. *DOM_DumpTreeToClipboard* adds the diagnostic text and the full contents of the XML tree passed as a reference to the clipboard. Below are the contents of the clipboard after the sample routine shown above is finished:

```
DOM Create XML Ref('contact')
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contact/>

DOM Create XML Ref('contact/business/phone')
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contact>

   <business>
     <phone/>
   </business>

</contact>

DOM Create XML element('/contact/home/phone')
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contact>

   <business>
     <phone/>
   </business>

   <home>
     <phone/>
   </home>

</contact>

DOM SET XML ELEMENT VALUE($businessPhone_xmlref;'123 456 789')
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contact>

   <business>
    <phone>123 456 789</phone>
   </business>

   <home>
     <phone/>
   </home>

</contact>

DOM SET XML ELEMENT VALUE($homePhone_xmlref;'123 888 999')
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contact>

   <business>
    <phone>123 456 789</phone>
   </business>

   <home>
    <phone>123 888 999</phone>
   </home>

</contact>
```

DOM_DumpTreeToClipboard Code Listing

Below is the code of the *DOM_DumpTreeToClipboard* method:

```
C_STRING(16;$1;$xmlref)
C_TEXT($2;$introduction_text)
C_BOOLEAN($3;$clearClipboard_b)

$xmlref:=$1
$introduction_text:=Char(Carriage return )+$2+Char(Carriage return )
$clearClipboard_b:=False
If (Count parameters>=3)
   $clearClipboard_b:=$3
End if

C_TEXT($existing_text)
$existing_text:=""
If ($clearClipboard_b=False)
   $existing_text:=Get text from clipboard
End if

C_TEXT($xml_text)
$xml_text:=""
DOM EXPORT TO VAR($xmlref;$xml_text)

SET TEXT TO CLIPBOARD($existing_text+$introduction_text+$xml_text)
```

Clearly the code is very quick-and-dirty. For example, it is limited to contents that fit within a 4th Dimension text variable. However, even in the simple form presented here, *DOM_DumpTreeToClipboard* is a useful tool. From the output shown above it's easy to see how each command modifies the XML. As an example, review the following command:

```
$root_xmlref:=DOM Create XML Ref("contact")
```

This is a simple line of code but produces an entire XML tree, as shown in the output produced by *DOM_DumpTreeToClipboard*:

```
DOM Create XML Ref('contact')
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contact/>
```

Special Note: SET WEB SERVICE OPTION

When 4th Dimension serializes an XML tree, it always produces a full and complete XML document. Therefore, the first line of output is always an XML prolog, such as the following:

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

An XML document must have one and only one prolog on the first line. It is illegal for an element within the body of the XML to contain a prolog. This rule creates a situation where dumping the contents of an XML tree may be misleading. The **SET WEB SERVICE OPTION** command expects an XML tree when using the Web Service SOAP Header option. In this context, 4th Dimension takes the contents of the XML tree supplied as an argument and inserts them into the next SOAP request message. However, no illegal prolog is inserted. If you're debugging an XML tree being prepared for use as a SOAP header, don't be concerned about the prolog produced when exporting the tree for debugging.

The situation just described is easiest to understand by looking at some sample SOAP messages. First, consider the XML of a simple SOAP request without a SOAP header (whitespace adjusted for clarity):

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <SOAP-ENV:Body>
      <mns:SayHelloWorld xmlns:mns="http://www.4d.com/namespace/default">
      </mns:SayHelloWorld>
   </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

The code below prepares a custom SOAP header for insertion into the request:

```
C_STRING(16;$root_xmlref)
C_STRING(16;$username_xmlref)
C_STRING(16;$password_xmlref)
$root_xmlref:=DOM Create XML Ref("credentials")
$username_xmlref:=DOM Create XML element($root_xmlref;"/credentials/username/")
$password_xmlref:=DOM Create XML element($root_xmlref;"/credentials/password/")
DOM SET XML ELEMENT VALUE($username_xmlref;"Donald")
DOM SET XML ELEMENT VALUE($password_xmlref;"Quack1")
SET WEB SERVICE OPTION(Web Service SOAP Header ;$root_xmlref)
```

Inserting this header alters the SOAP request, as highlighted in the XML listing below (whitespace adjusted for clarity):

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP-ENV:Envelope
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <SOAP-ENV:Header>
     <credentials>
       <username>Donald</username>
       <password>Quack1</password>
     </credentials>
   </SOAP-ENV:Header>

   <SOAP-ENV:Body>
      <mns:SayHelloWorld xmlns:mns="http://www.4d.com/namespace/default">
      </mns:SayHelloWorld>
   </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

The XML prolog is the same in both versions of the message and, in both cases, there is one and only one prolog. However, dumping the XML tree suggests there might be a redundant prolog. The code fragment below shows the tree being dumped to the clipboard after being prepared and inserted in the SOAP message:

```
SET WEB SERVICE OPTION(Web Service SOAP Header ;$root_xmlref)
DOM_DumpTreeToClipboard ($root_xmlref;"Custom SOAP header";True)
```

Below is the XML included on the clipboard:

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

  <username>Donald</username>

  <password>Quack1</password>

</credentials>
```

The output starts with a prolog because **DOM EXPORT TO VAR** and **DOM EXPORT TO FILE** always convert an XML tree into a full, legal XML document. However, as already noted, no redundant prolog is generated by **SET WEB SERVICE OPTION**. Therefore, if dumping the contents of an XML tree being passed to **SET WEB SERVICE OPTION**, don't worry about the XML prolog.

Summary

4th Dimension's DOM commands create XML trees which are difficult to inspect before they're turned into completed XML documents. Dumping trees is a quick way to help solve problems, debug proxy code produced by the Web Services Wizard, and learn more about the underlying behavior of the DOM commands. This technical note describes a simple utility for dumping XML to text and onto the clipboard that can be enhanced and modified for use in your own work.