Chapter

14

A Quick Intro to XML with ShiVa

In this Chapter I'm going to give you a very quick overview of how XML (Extensible Markup Language) can be used from within ShiVa. XML is very useful as, not only is it human readable, but it is also simple for a computer to parse. Sorry, what does "parse" mean? On Wikipedia, the definition of the word "parse", as it applies to computing, is as follows:

"the process of analyzing a text, made of a sequence of tokens (for example, words), to determine its grammatical structure with respect to a given (more or less) formal grammar"

Well, that's about as clear as mud! What it all means, is that parsing is the process whereby you can take a file full of some sort of data (such as a "High Score Table" where you may have the Name, Date & Score of the top XX scores for your Game) and go through it line by line to get at that data (sort of like a database). As long as you know how the data is structured, you can easily get at the bits you need. This is where XML comes in handy, due to its inherent structure. Or, as the official StoneTrip demo states:

"Extensible Markup Language (XML) is a human-readable, machine-understandable, general syntax for describing hierarchical data, applicable to a wide range of applications (databases, e-commerce, Java, web development, etc.)."

XML documents are treated as a type of Variable within ShiVa (however, they must be AI MEMBER Variables), and as such can be treated just like any other AI Variable (i.e. they can be sent, received, parsed and built directly from within ShiVa).

An XML document looks something like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="Dino_Hunter.xsl"?>

<ExportReport name="Dino_Hunter" Compress_Size="7440932"
UnCompress_Size="11112246" FileCount="262" >
      <Resource Name="Particles" Compress_Size="73" UnCompress_Size="83"
FileCount="1" >
            <File Name="Weapon0Explosion" UnCompress_Size="83" Compress_Size="73"
Extension="par" />
      </Resource>
      <Resource Name="Fonts" Compress_Size="205" UnCompress_Size="284"
FileCount="1" >
            <File Name="HLDArial" UnCompress_Size="284" Compress_Size="205"
Extension="fnt" />
      </Resource>
</ExportReport>
```

It may look a bit complicated, but in reality it's not! The first couple of lines of the document set the Version of XML to be used ("1.0"), the Encoding of the XML to be used ("ISO-8859-1"), and a StyleSheet to be used when displaying the XML (don't worry too much about this, as it is outside the scope of this book. If you are really interested in learning more about XML, then I would suggest getting a good book, or searching on-line for the multitude of tutorials that are out there).

The next lines are the interesting bit (from our point of view anyway!). You can see that the above lines are indented, which, just like good programming, means that it is easier to see where each "block" starts and finishes. The "blocks" in the above are "ExportReport" and "Resource". You'll also probably have noticed that there is one "ExportReport" block, and two "Resource" blocks (with different "Names"). This is because the "ExportReport" is the main block (similar to the naming of a Script), and the "Resource" blocks are the actual data.

So, what we have above can be set out as follows:

1) the whole document is a "ExportReport" with the following overall parameters:

   a) Name – Dino_Hunter
   b) Compress_Size – 7440932
   c) UnCompress_Size– 11112246
   d) FileCount – 262

2) This report then has two "Resources":

   a) Name - Particles
      Compress_Size - 73
      UnCompress_Size - 83
      FileCount - 1
   b) Name - Fonts
      Compress_Size - 205
      UnCompress_Size - 284
      FileCount - 1

3) Within each "Resource", there is one "File":

   a) Name - Weapon0Explosion
      UnCompress_Size - 83
      Compress_Size - 73
      Extension - par
   b) Name - HLDArial
      UnCompress_Size - 284
      Compress_Size - 205
      Extension - fnt

As you can see, each layer of the XML doc includes more information than the previous layer (which summarises the sub-layers below). Each layer can have multiple sub-layers, making the laying out of data a relatively simple task (if you are methodical!).

For instance, if your Game used multiple Particle Systems, you would have one "File" for each Particle System within the "Particles" Resource.

But, how does this help? Well, if you wanted to create a "High Score Table", you could define your XML document something like this:

```
<HSTable name="HST" >
    <HSEntry number= "1" Name="John Smith" Score="100000" Date="01/07/2009" />
                                                    :
    <HSEntry number= "10" Name="Fred Bloggs" Score="10000" Date="03/07/2009" />
</HSTable>
```

Not that complicated really. You can see that each entry in the table is called a "HSEntry", and is numbered "1" to "10", with a "Name", "Date" and "Score" for each. Don't worry about all the "<", ">" and "/" characters, as they are just ways of showing the beginning and end of each block within the XML document. Another great thing about XML docs are that they can be stored pretty much anywhere (and aren't usually that big!).

That's all well and good, but how does ShiVa handle XML?

ShiVa has the ability to do pretty much whatever you want to do with XML docs. It has over 30 Functions dedicated to XML processing.

As examples, I'll just quickly run through the StoneTrip XML Tutorial:

Load an XML document

An XML document can be loaded from a local or distant location using the following Function:

bOK = xml.receive (hXML, sURI).

From a local location, the URI parameter must be prefixed by "file://":

```
xml.receive ( this.hMyXML ( ), "file://C:/Program Files/MyXML.xml" )
```

From distant locations, the URI parameter must be prefixed by "http://":

```
xml.receive ( this.hMyXML ( ), "http://developer.stonetrip.com/tuto/test.xml" )
```

From both locations, this operation is asynchronous, so you will have to test the status of the XML file before using it.

The waiting loop can be done in a State, as shown below:

```
-------------------------------------------------------------------------
function AIXMLManager.GetXML_onEnter ( )
-------------------------------------------------------------------------

xml.receive ( this.hMyXML ( ), "http://developer.stonetrip.com/tuto/test.xml" )

-------------------------------------------------------------------------
end
-------------------------------------------------------------------------
```

The status of the XML document can be retrieved by using the following Function:

nStatus = xml.getReceiveStatus (hXML)

Where "*nStatus*" will be one of the following:

-3	Parse failed. The XML has arrived but is not valid.
-2	The request failed. The server hasn't responded, or the URI is wrong (take care of capitals in the URI!)
-1	The XML is not in the cache. You haven't used the "*receive*" Function for this XML.
0 to 0.99	Receiving. (nStatus * 100) is the progress percentage.
1	The XML is ready to be used.

For example:

```
-------------------------------------------------------------------------------
function AIXMLManager.GetXML_onLoop ( )
-------------------------------------------------------------------------------

local nXMLStatus = xml.getReceiveStatus ( this.hMyXML ( ) )
```

Here, we are retrieving the status of the "Receive", and storing it in a LOCAL Variable ("*nXMLStatus*").

```
if ( nXMLStatus == -3 )
then
   log.warning ( "XML parse failed." )

   --Leave GetXML state for another state
   this.Idle ( )
```

Firstly, we check for a parse failure. If there is one, then we put a message in the Log, and set the State to "idle".

```
elseif ( nXMLStatus == -2 )
then
   log.warning ( "XML not found." )

   --Leave GetXML state for another state
   this.Idle ( )
```

Next, we check that the XML exists. If it doesn't, we put a message in the Log file, and set the State to "idle".

```
elseif ( nXMLStatus == 1 )
then
   log.message ( "XML completed." )
   log.message (  xml.toString ( xml.getRootElement ( this.hMyXML ( ) ) ) )

   --Process XML
   this.DoSomethingWithXML ( )
```

```
    --Leave GetXML state for another state
    this.Idle ( )
end

-------------------------------------------------------------------------
end
-------------------------------------------------------------------------
```

Finally, we check to see if the "Receive" is successful. If it is, then we put a message in the Log file, along with the name of the Root element of the XML document (using the "*toString*" Function), and process the XML document. Lastly, we set the State to "idle".
NOTE: "*this.DoSomethingWithXML ( )*" is NOT a real ShiVa Function!

Send an XML document

An XML document can be sent to a local or distant location using the following Function:

bOK = xml.send (hXML, sURI).

To a local location, the URI parameter must be prefixed by "file://":

xml.send (this.hMyXML ( ), "file://C:/Program Files/MyXML.xml")

To a distant location, the URI parameter must be prefixed by http://:

xml.send (this.hMyXML ( ), "http://developer.stonetrip.com/tuto/Test_GetXML.php")

Again, this operation is asynchronous. However, most of the time, you don't need to wait until the end of this operation as it will not alter the Variable.

You can use the following Function to check the progress of the "*Send*" operation:

nStatus = xml.getSendStatus (hXML)

If you need to wait until the file has been sent, you can use the same code as shown for the "*Receive*" operation.

If you are sending the XML file to a local location, it will be written to the file and location you specified. If, however, you are sending the XML file to a distant location, it will be sent in a POST request to the URL that you specified.

An example of the php script is shown below:

```
<?php
if ( isset($_POST['STContent'] ) )
{
    $xml = $_POST['STContent'];
    echo $xml;
}
?>
```

Parse an XML document

A XML document always has only one Root Element which is the entry point for the document. The Root node can have one or many Child nodes, which can also have one or many Child nodes etc. etc.

To be able to extract the XML elements, you will have to retrieve the Handle of their parent.

Below is an example of navigating through an XML document (this has been taken form Wikipedia):

```
<recipe name="bread" prep_time="5 mins" cook_time="3 hours">
   <title>Basic bread</title>
   <ingredient amount="8" unit="dL">Flour</ingredient>
   <ingredient amount="10" unit="grams">Yeast</ingredient>
   <ingredient amount="4" unit="dL" state="warm">Water</ingredient>
   <ingredient amount="1" unit="teaspoon">Salt</ingredient>
   <instructions>
      <step>Mix all ingredients together.</step>
      <step>Knead thoroughly.</step>
      <step>Cover with a cloth, and leave for one hour in warm room.</step>
      <step>Knead again.</step>
      <step>Place in a bread baking tin.</step>
      <step>Cover with a cloth, and leave for one hour in warm room.</step>
      <step>Bake in the oven at 180(degrees)C for 30 minutes.</step>
   </instructions>
</recipe>
```

So, for example, if you want to extract only the amount of each "ingredient" element and its name, you would do something like this:

```
local hRootelement = xml.getRootElement ( this.hMyXML ( ) )
```

This first line creates a LOCAL Variable ("*hRootelement*"), which is set to contain the name of the Root element from the AIModel Variable "*hMyXML*".

```
if ( hRootelement )
then
   local hXMLEntry = xml.getElementFirstChildWithName ( hRootelement, "ingredient" )
```

If "*hRootelement*" is successfully created, we then create another LOCAL Variable "*hXMLEntry*", which is set to contain the value of the first Child of the Root element with a "Name" of "ingredient".

```
   while ( hXMLEntry ~= nil )
   do
```

Now, we set up a loop to go through the XML document.

```
      local sIngredientName = xml.getElementValue ( hXMLEntry )
      local sIngredientAmount = "None"
      local hAmountAttribut = xml.getElementAttributeAt ( hXMLEntry, 0 )
```

For each step of the loop, we create three LOCAL Variables ("*sIngredientName*", "*sIngredientAmount*" and "*hAmountAttribut*") to hold the following information:

sIngredientName	-	the Value of the Child element (i.e. its Name)
sIngredientAmount	-	set to "None"
hAmountAttribut	-	a Handle to the first element of the Child element*

*the elements of each Child are zero-based, hence the "0" representing the first element.

```
    if ( hAmountAttribut )
    then
       sIngredientAmount = xml.getAttributeValue ( hAmountAttribut )
    end

    log.message ( "Ingredient : ", sIngredientName, " amount : ", sIngredientAmount )
```

Next, we check to see if "*hAmountAttrib*" exists and, if so, we set "*sIngredientAmount*" to be equal to the value of the element pointed to by "*hAmountAttrib*". We also write an appropriate Log message, detailing the information we have extracted.

```
    --next element
    hXMLEntry = xml.getElementNextSiblingWithName ( hXMLEntry, "ingredient" )

  end
end
```

Finally, we get the next element that has the name "ingredient". Note the use of the "*getElementNextSiblingWithName*", which means getting the next element at the **SAME LEVEL**, in the hierarchical tree, as the element that was found initially.

The above Script outputs the following to the Log file:

Ingredient : Flour amount : 8
Ingredient : Yeast amount : 10
Ingredient : Water amount : 4
Ingredient : Salt amount : 1

If you wished to extract all of the Child elements of "instructions", you would use something like this:

```
local hRootelement = xml.getRootElement ( this.hMyXML ( ) )
if ( hRootelement )
then
  local hXMLEntry = xml.getElementFirstChildWithName ( hRootelement, "instructions" )
```

Up to this point, this is the same as the code above (except for the Name of the Child of course!).

```
  if ( hXMLEntry ~= nil )
  then
```

This time round, we check that "*hXMLEntry*" contains a value and, if so, we continue:

```
        for i = 0, xml.getElementChildCount ( hXMLEntry ) - 1
        do
```

Now, we set up a loop based on the Count of the number of Child elements of "*hXMLEntry*". Remember that these are zero-based, and hence the "*0*" as the start of the loop, and the "*- 1*" after the number of Child elements (since the actual Count is NOT zero-based!).

```
            local hXMLChildEntry = xml.getElementChildAt ( hXMLEntry, i )
            if ( hXMLChildEntry ~= nil )
            then
                log.message ( "Step " .. (i+1) .. " : " .. xml.getElementValue ( hXMLChildEntry ) )
            end
        end
    end
end
```

Finally, we create another LOCAL Variable ("*hXMLChildEntry*"), which will contain the value of the element of the Child at the identifier specified by the loop. If this value is other than "*nil*", then we output a message to the Log file with the required information.

This Script outputs the following to the Log file:

Step 1 : Mix all ingredients together.
Step 2 : Knead thoroughly.
Step 3 : Cover with a cloth, and leave for one hour in warm room.
Step 4 : Knead again.
Step 5 : Place in a bread baking tin.
Step 6 : Cover with a cloth, and leave for one hour in warm room.
Step 7 : Bake in the oven at 180(degrees)C for 30 minutes.

Build or Modify an XML document

You can build an XML document from scratch by adding Children to the Root element, or you can easily modify the value of an existing element.

NOTE: the name of the element, and of any attributes, **MUST NOT** contain spaces or symbols.

To add a Child to an existing element, you would use the following code:

```
local hRootelement = xml.getRootElement ( this.hMyXML ( ) )
if ( hRootelement )
then
    local hXMLEntry = xml.getElementFirstChildWithName ( hRootelement, "instructions" )
    if ( hXMLEntry ~= nil )
    then
```

All pretty much the same to here.

```
    --insert a new element at first place
    xml.insertElementChildAt ( hXMLEntry, 0, "step", "Buy the ingredients" )
```

This line simply uses a value of "*0*" to specify the first element of the Child, and inserts a new element at that location. I would point out that the name of the element (i.e. "step") must NOT have any of the "<" or ">" characters added to it, as these are added internally by ShiVa.

```
    --add a new element after all children
    xml.appendElementChild   ( hXMLEntry, "step", "Eat" )
```

This line demonstrates a way of attaching a new element to the end of the list, by using the "*appendElementChild*" Function.

```
    for i = 0, xml.getElementChildCount ( hXMLEntry ) - 1
    do
       local hXMLChildEntry = xml.getElementChildAt ( hXMLEntry, i )
       if ( hXMLChildEntry ~= nil )
       then
          log.message ( "Step " .. (i+1) .. " : " .. xml.getElementValue ( hXMLChildEntry ) )
       end
    end
 end
end
```

The rest of the code is identical to that above, so I shouldn't have to make any comments.

This code will output the following to the Log file:

```
Step 1 : Buy the ingredients
Step 2 : Mix all ingredients together.
Step 3 : Knead thoroughly.
Step 4 : Cover with a cloth, and leave for one hour in warm room.
Step 5 : Knead again.
Step 6 : Place in a bread baking tin.
Step 7 : Cover with a cloth, and leave for one hour in warm room.
Step 8 : Bake in the oven at 180(degrees)C for 30 minutes.
Step 9 : Eat
```

Well, that's it for this intro to XML. Obviously, there is a lot more to the XML API (and XML in general!) that you would need to know to fully utilise its power, but, as I have already stated, that is beyond the scope of this book! In the next Chapter I'll be introducing the HLDL Framework, which we will use to develop our "Dino Hunter" Game.