

XML - A P r i m e r

Presented by: **Silvio Belini**

INTRODUCTION

Extensible Markup Language (XML) is a data format that allows you to create structured data. XML was derived from SGML (Standard Generalized Markup Language) and is standardized by the World Wide Web Consortium. XML is largely used to share structured data over different systems and has become particularly important in transferring data over the Web.

Among other things, XML was designed with the following goals in mind:

- XML shall be straightforwardly usable over the Internet.
- XML shall support a wide variety of applications.
- It shall be easy to write programs which process XML documents.
- XML documents should be human-legible and reasonably clear.
- The design of XML shall be formal and concise.
- XML documents shall be easy to create.

XML CONCEPTS

XML is a specification, not a language. It is a standard for creating languages that meet the XML criteria. In other words, XML describes a syntax that you use to create your own languages. Below are some properties of XML:

- XML is a markup language (similar to HTML)
- XML was designed to describe data
- XML language is self-describing
- XML tags are not predefined in XML. You must define your own tags
- XML uses a Document Type Definition (DTD) or an XML Schema to describe data
- XML with a DTD or XML Schema is designed to be self-descriptive

One main purpose of XML is to making it easier to write software that accesses the information, by giving the structure of the data. The Web is the ideal medium for many business applications and solutions, because it enables users to share documents and work processes with almost anyone. XML has been widely adopted for Web applications because of its affinity with HTML and its ability to serve as a repository for multiple types of data and data structure.

In an integrated application environment, business applications communicate seamlessly with one another. XML makes it possible for all these applications to speak the same language. Time and productivity formerly spent on translating data formats and dealing with incompatibilities in them can now be spent solving business problems. Making good use of XML in the business environment means users do not have to learn a new application or technology. They can make use of XML in the applications they work with every day.

XMLSyntax

Here are the common elements that will make up an XML document:

Processing Instructions (PI's)

PI's are commands that directly address the program processing the code with instructions on how to do so. All processing instructions begin with an open bracket and a question mark, and end with a question mark and closing bracket. Immediately following the opening tag is the name of the application that is to make use of that information. There should be no space before it.

```
<?instructionname SELECT * FROM somewhere?>
```

XML Tags

XML tags makeup the most significant portion of an XML document. XML tags are used to define the contents of a document. All markup tags in XML have an open tag and a close tag. Usually the close tag is the same as the open tag except that it has a slash in front of it. For instance, the close tag for <first> would be </first>. Here is an example of an XML document that we will use to look into the structure of tags:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contacts>
  <name>
    <first title="Mr.">Nathan</first>
    <middle>Christopher</middle>
    <last>Summers</last>
  </name>
  <name>
    <first title="Ms.">Kathrine</first>
    <middle>Emily</middle>
    <last>Pryde</last>
  </name>
  <name>
    <first title="Dr." >Henry</first>
    <middle />
    <last>McCoy</last>
  </name>
</contacts>
```

Note – the syntax has been indented for readability.

Elements

All of the information from the start of the open tag to the end of the close tag and everything in between is known as an Element. The text in between the start and end tags is called the Element content or Element value. Below is an example of an Element item.

```
<element name>element value</element name>
```

The angle brackets surrounding the tag tell the XML parser that what is inside the brackets is a command. They are command delimiters. The above structure is called an Element.

Examples:

```
<first title="Mr." suffix="Jr">Nathan</first>

<middle>Christopher</middle>

<last>Summers</last>
```

The first string in this tag indicates that the tag is a tag named "first", which in this example, describes someone's first name. This string is the name of the tag. It is also the name of the element type the tag is describing. When you combine XML tags with their

content, you get an element. The slash character (/) is a command that indicates the end of some tag. It is the end tag marker. Following it with the string first indicates that the tag being ended is a "first" tag. If there is more than one still open, then it is applied to the last one opened. For tags that aren't delimiting any content, they can be minimized. In a minimized tag, the slash is included at the end of the opening tag to tell the parser that the command is finished and there will be no closing tag. Continuing with this example, if the person does not have a middle name (third name element above), the middle tag can be represented by:

<middle />

Attributes

Something else you will often find inside an XML tag is an attribute. Attributes are special values that can be assigned to that element. Attributes provide a means for specifying additional information about the element being marked up. They occur as name="value" pairs. The XML attributes in the above example is:

<element name attribute1 ... attribute[N]>element value</element name>

Example:

<first title="Mr." suffix="Jr">Nathan</first>

The rules of the using attributes in XML are as follows:

- Attributes are included as a space separated list of name/value pairs inside the opening tag of an element.
- The order in which attributes appear in the tag is not important; however an attribute can only occur once within a given tag.
- The name and value in each pair is separated by an equals sign.
- All values must be in quotes. They can be either single or double quotes, though double quotes are preferred. In either case, the quotes must be balanced.

Entity References

Certain characters have special meaning and are illegal in XML. For example, if the character "<" is present in an element value, the parser will interpret this as the start of a new element and will generate an error. These characters can be used by way of entity references. Entity references are placeholders that represent an entity (usually text). An entity reference consists of the entity name preceded by an ampersand "&" and followed by a semicolon ";". For example:

This syntax will generate an error:

<statement>if (1 < x) </statement>

This syntax is correct:

<statement>if (1 < x) </statement>

Here is a list of the predefined entity references in XML:

Entity name	Character represented	Syntax
Lt	< or "less than"	<
gt	> or "greater than"	>

amp	& or “ampersand”	&
apos	' or “apostrophe”	'

CDATA

Another option for using illegal characters is to use a CDATA section. All text within a CDATA section will not be parsed. The text will be interpreted as only character data. For example, the following is a valid CDATA section:

```
<![CDATA[
function myFunction(a,b,c) {
    if (a < b && b > c) {
        return 'pass';
    }
    else {
        return “fail”;
    }
}
]]>
```

Well-formed XML vs Valid XML

A Well-Formed XML document meets the syntactical rules outlined in the XML 1.0 specification. A well formed document meets the minimum criteria for XML processors and validators to read the document. All true XML Documents are well formed documents (otherwise they would not be XML documents). An XML document must adhere to the following rules in order to be considered Well-Formed:

- Every start tag must have a matching end tag, unless it is a self-closing tag. For instance, in the name example above, the element <name> contains three other elements. The second element is <middle>, and <middle>, </middle> tags represent the start and end tags respectively. If the person does not have a middle name, then a self-closing tag, <middle /> would be used instead.
- Tags can't overlap. Unlike HTML, you will not be able to overlap your XML tags.

This would result in an ill-formed document:

```
<p>this <b><em>is</b> some text</em></p>
```

This is a well-formed document:

```
<p>this <b><em>is</em></b><em> some text</em></p>
```

- Only one root element in an XML document. The XML document shall only have one main element. All other elements are either its children, or descendants. It must even have a root element even if it does not have any content.
- Element names must obey XML naming conventions. Names must start with characters or "_" key, not numbers or punctuation characters
- After the first character, numbers are allowed.

- Names can't contain spaces.
- Names can't contain ":", it's a reserved character.
- Names can't start with the characters "xml" in any case what-so-ever.
- XML is case-sensitive. In HTML, <table> and <TaBlE> are the same. In XML <table>, <TABLE>, and <Table> are all different.
- XML will keep white space in your text. Unlike HTML, no white space stripping takes place in XML.

A Valid XML document is not the same thing as a Well-Formed XML document. An XML document may be Well-Formed, but that does not guarantee that it is properly written. A valid XML document, on the other hand is a more formal form of XML. Valid documents must conform not only to the syntax, but also to the DTD (Document Type Definition). An XML document that has been verified against a DTD is said to be valid. With the 4D XML commands Parse XML Source and Parse XML Variable, 4D's parser will attempt to validate the XML structure of the document based on the DTD defined, the DTD referred to in the document, or that designated by the DTD parameter.

Every SGML document requires a DTD for the application to interpret it. For HTML, the application is a Web browser, which has a DTD built right in. XML also requires a DTD, but since the DTD can be dramatically different from one application to another, you have to provide your own. Without a DTD, a program won't know how to display the XML document without further instructions.

DOCUMENT TYPE DEFINITION AND SCHEMA

Think of DTD's as a guide line or a set of rules for creating your XML document. Remember, a valid XML document is one that adheres to its DTD. It defines what tags appear in a XML document and where; so that viewers of an XML document know what all the tags mean. DTD's role is to describe the structure of an XML document. A DTD can be declared inline in your XML document, or as an external reference.

NOTE: When the parser reads the XML file, it also reads the DTD, and then checks the file to make sure it follows those rules written in the DTD.

Generically speaking, a schema is a set of information that describes the structure of another set of information. It is a set of rules or a grammar for describing some kind of data structure. An XML schema identifies the constraints on the content of the XML documents, and describes the vocabulary (rules or grammar) that compliant XML documents must follow in order to be considered schema-valid with that particular schema. DTD's are a type of schema.

XML provides an application independent way of sharing data. Independent groups of people can agree to use a common schema for interchanging data. Your application can use a standard DTD/schema to verify that data that you receive from the outside world is valid. You can also use a DTD/schema to verify your own data.

4D XML COMMANDS (BUILDING AND ACCESSING XML)

4D offers two different modes for parsing XML documents: DOM (Document Object Model) and SAX (Simple API XML). Each mode comes with its own set of commands within 4D to manipulate XML documents.

Here is the XML document from above again. It will be used in the example for each mode.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<contacts>
  <name>
    <first title="Mr.">Nathan</first>
    <middle>Christopher</middle>
    <last>Summers</last>
  </name>
  <name>
    <first title="Ms.">Kathrine</first>
```

```

        <middle>Emily</middle>
        <last>Pryde</last>
    </name>
    <name>
        <first title="Dr." >Henry</first>
        <middle />
        <last>McCoy</last>
    </name>
</contacts>

```

DOM

The DOM mode parses an XML source and builds its structure (its "tree") in memory. Because of this, access to each element of the source is extremely fast. However, since the entire tree structure is stored in memory, the processing of large XML documents may lead to the memory capacity being exceeded and thus provoke errors. One advantage of using DOM mode is that it allows accessing nodes in random order. DOM mode builds the structure in memory, uses references to element nodes to properly build the hierarchy and then save to file. DOM mode also allows you to modify elements, their values, and attributes, using the 4D DOM commands.

Here is an example of how to create the above XML file using the 4D DOM XML commands:

```

`Method: DOM_Create_XML_Doc

C_STRING(16;vRootRef;vElemRef)
C_TEXT($rootElem;$vElem1;$vElem2;$vElem3)
C_TEXT($mtitle;$myfilepath)

$rootElem:="contacts"
$myfilepath:="Macintosh
HD:Users:jessepina:Desktop:XML:contacts1.xml" ` name of the XML file
to create.
vRootRef:=DOM Create XML Ref($rootElem)

ALL RECORDS([Contacts])
DOM SET XML OPTIONS(vRootRef;"UTF-16";False) `Set encoding and stand
alone values.

`Go through all records in contacts table.
For (i;1;Records in selection([Contacts]))
    $MainElem:="/contacts/name"

    `Create name element
    vElemRef:=DOM Create XML element(vRootRef;$MainElem)

    `Create first name element which is a child of name element.
    $vElem1:="first"
    $mtitle:=[Contacts]Title
    vElemRef1:=DOM Create XML
element(vElemRef;$vElem1;"title";$mtitle)

    `Set first name value
    $fname:=[Contacts]firstname
    DOM SET XML ELEMENT VALUE(vElemRef1;$fname)

    `Create middle name element which is also child of name
element.
    $vElem2:="middle"
    vElemRef2:=DOM Create XML element(vElemRef;$vElem2)

```

```

        `Set middle name value
        $middle:=[Contacts]middle
        DOM SET XML ELEMENT VALUE (vElemRef2;$middle)

        `Create last name element
        $vElem3:="last"
        vElemRef3:=DOM Create XML element (vElemRef;$vElem3)

        ` set last name value
        $last:=[Contacts]lastname
        DOM SET XML ELEMENT VALUE (vElemRef3;$last)
        NEXT RECORD ([Contacts])

End for

    ` save XML to file.
    DOM EXPORT TO FILE (vRootRef;$myfilepath)

```

SAX

The SAX mode does not build a tree structure in memory, but handles an XML document in an event streaming approach. In this mode, "events" (such as the start and end of an element) are generated when parsing the source. This mode lets you parse XML documents of any size, regardless of the amount of memory available. SAX parses an XML document node by node in a linear fashion. It follows a top to bottom approach returning an event for every element it encounters such as begin tag, end tag, etc. It does not allow random access manipulation of an XML document. When parsing an existing XML document in SAX mode, the document needs to be opened in read-only. Because of this, you would not be able to modify elements, their values, and attributes with SAX.

Here is an example of how to create the above XML file using the 4D SAX XML commands:

```

`Method: SAX_Create_XML_Doc

C_TIME ($docref)
C_TEXT ($rootElem;$vElem1;$vElem2;$vElem3)
C_TEXT ($mtitle;$myfilepath)

ALL RECORDS ([Contacts])

$myfilepath:="Macintosh
HD:Users:jessepina:Desktop:XML:contacts2.xml" ` name of the XML
file to create.
$DocRef:=Create document ($myfilepath)

SAX SET XML OPTIONS ($DocRef;"UTF-16";False) ` set encoding and
stand alone values

`Create root element "contacts". A begin tag of the same name
is added.
$rootElem:="contacts"
SAX OPEN XML ELEMENT ($DocRef;$rootElem)

`Go through all the records in table
For ($i;1;Records in selection ([Contacts]))

    `Create parent element "name". A begin tag "name" is
    added.
    $MainElem:="name"
    SAX OPEN XML ELEMENT ($DocRef;$MainElem)

    `Create child element "first". Begin tag "first" is

```

```

created.
    $vElem1:="first"
    $mtitle:=[Contacts]Title
    SAX OPEN XML ELEMENT($DocRef;$vElem1;"title";$mtitle)

    `Set value for the first tag.
    $fname:=[Contacts]firstname
    SAX ADD XML ELEMENT VALUE($DocRef;$fname)

    `Close last element created. An end tag "first" is added.
    SAX CLOSE XML ELEMENT($DocRef)

    `Create a child element "middle". Begin tag "middle" is
created.
    $vElem2:="middle"
    SAX OPEN XML ELEMENT($DocRef;$vElem2)

    `Set value for middle tag.
    $middle:=[Contacts]middle
    SAX ADD XML ELEMENT VALUE($DocRef;$middle)

    `Close the "middle" tag. An end tag "middle" is added.
    SAX CLOSE XML ELEMENT($DocRef)

    `Create a child element "last". Begin tag "last" is
created.
    $vElem3:="last"
    SAX OPEN XML ELEMENT($DocRef;$vElem3)

    `Set value for last tag.
    $last:=[Contacts]lastname
    SAX ADD XML ELEMENT VALUE($DocRef;$last)

    `Close the "last" tag. An end tag "last" is added.
    SAX CLOSE XML ELEMENT($DocRef)

    `Close the "name" tag. An end tag "name" is added.
    SAX CLOSE XML ELEMENT($DocRef)

    NEXT RECORD([Contacts])

End for

    `Close the "contacts" tag.
    SAX CLOSE XML ELEMENT($DocRef)

CLOSE DOCUMENT($DocRef)

```

Aside from the commands in these examples, there are numerous other commands available within the XML theme for both DOM and SAX. For a complete list of the commands and descriptions, please see the Language Reference manual.

USES WITHIN 4D

XML is used extensively within 4D. Here are a few examples:

Import/Export Structure

A new feature in 4D v11 SQL allows Importing and Exporting the structure of databases, which consists of the tables, fields, indexes, relations, structure editor settings, and their attributes. These new built in options allow developers to generate an XML structure file, which can later be used to create new databases. This functionality can also be used to move tables, fields and their attributes between databases.

By modifying the generated XML structure file, a developer can basically make any change that can be made in the structure editor.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE base SYSTEM "http://www.4d.com/dtd/2007/base.dtd" >
<base name="CustomResources" uuid="E08BFDD53C03F84082635EB84E74D0EB"
collation_locale="en">
  <table name="Contacts" uuid="A485C1E298674F49B6A50C8A965BE071">
    <field name="contact_ID" uuid="C61DD50D9E34C845B521033D38507CCB"
type="3" unique="true" never_null="true">
      <index_ref uuid="27F3A0C00CAF154293820F5F04291D7E"/>
    </field>
    <field name="First_Name" uuid="213C26D9CA281F43AB1AA922C24A8AA9"
type="10" limiting_length="100" never_null="true"/>
    <field name="Last_Name" uuid="3A6ED85D83C04E45ABE453D21CA1C311"
type="10" limiting_length="100" never_null="true">
      <index_ref uuid="F4EE492A5D809343A1D4317A2093C966"/>
    </field>
    <field name="phone" uuid="863F7F6DBC90494DADE6B283DDF7CECD"
type="5" never_null="true"/>
    <field name="email" uuid="B68FE630A3D743479B99EB317E25E4C7"
type="10" limiting_length="255" never_null="true"/>
    <table_extra>
      <editor_table_info>
        <color red="224" green="234" blue="104" alpha="255"/>
        <coordinates left="84" top="9" width="157"
height="192"/>
      </editor_table_info>
    </table_extra>
  </table>
  <index kind="regular" unique_keys="true"
uuid="27F3A0C00CAF154293820F5F04291D7E" type="7">
    <field_ref uuid="C61DD50D9E34C845B521033D38507CCB"
name="contact_ID">
      <table_ref uuid="A485C1E298674F49B6A50C8A965BE071"
name="Contacts"/>
    </field_ref>
  </index>
  <index kind="regular" uuid="F4EE492A5D809343A1D4317A2093C966" type="1">
    <field_ref uuid="3A6ED85D83C04E45ABE453D21CA1C311"
name="Last_Name">
      <table_ref uuid="A485C1E298674F49B6A50C8A965BE071"
name="Contacts"/>
    </field_ref>
  </index>
  <base_extra data_file_path="\CustomResources.4DD"
source_code_stamp="48">
    <temp_folder folder_selector="data"/>
  </base_extra>
</base>
```

Note – Most of the XML code for a table can also be obtained simply by copying the table from the structure editor and pasting into a text editor. Certain information like the database name and the index definitions do not get generated when copying this way. However, the relations information can be obtained along with the table information as long as both tables involved in the relation are copied together.

Import/Export data

From within 4D, all data can be exported and imported as XML. Within the “Import From File” and “Export to File” options that are built into 4D, you have the option of importing/exporting in XML. Also, Tech Note 06-07 has an example of a generic Import/Export method that can be used to programmatically import or export all data from a database.

Building Applications (XML Keys)

When building customized applications within 4D, an XML file is used to control various aspects of the building process. There are over 80 different XML keys that can be used to do things such as: set the application name, set the data file path, include a license file, build a server application, specify that a client be upgradeable, set an IP address for a built client to use when connecting to a server, and to set a version number. The XML file containing these tools can be created/modified using 4D commands or using any Text editor. Some settings can be modified within both the Build Application dialog and directly in the XML document, but many settings can only be modified by editing the XML file. Here is a quick example:

The following BuildApp.xml file was generated by using the Save Settings button within the Build Application dialog.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Preferences4D>
  <BuildApp>
    <BuildCompiled>False</BuildCompiled>
    <IncludeAssociatedFolders>True</IncludeAssociatedFolders>
    <BuildComponent>False</BuildComponent>
    <BuildApplicationSerialized>True</BuildApplicationSerialized>
    <BuildApplicationLight>False</BuildApplicationLight>
    <SourcesFiles>
      <RuntimeVL>
        <RuntimeVLIncludeIt>True</RuntimeVLIncludeIt>
        <RuntimeVLWinFolder>C:\Program Files\4D\4D v11 SQL (11.2)\4D
Volume Desktop\</RuntimeVLWinFolder>
      </RuntimeVL>
      <CS>
        <ServerIncludeIt>False</ServerIncludeIt>
        <ClientWinIncludeIt>False</ClientWinIncludeIt>
        <ClientMacIncludeIt>False</ClientMacIncludeIt>
      </CS>
    </SourcesFiles>
    <BuildApplicationName>app_building_test</BuildApplicationName>
    <BuildWinDestFolder>..\app_building_test_Build</BuildWinDestFolder>
    <CS>
      <BuildServerApplication>True</BuildServerApplication>
      <BuildCSUpgradeable>False</BuildCSUpgradeable>
      <CurrentVers>1</CurrentVers>
      <HardLink>app_building_test</HardLink>
    </CS>
    <ArrayExcludedPluginName>
      <ItemCount>0</ItemCount>
    </ArrayExcludedPluginName>
    <ArrayExcludedPluginID>
      <ItemCount>0</ItemCount>
    </ArrayExcludedPluginID>
    <ArrayExcludedComponentName>
```

```

        <ItemCount>0</ItemCount>
    </ArrayExcludedComponentName>
    <Licenses>
        <ArrayLicenseWin>
            <ItemCount>0</ItemCount>
        </ArrayLicenseWin>
    </Licenses>
</BuildApp>
</Preferences4D>

```

In the above example, no licenses will be included when the application is built. This can easily be changed by updating the <Licenses> tag to reflect the following:

```

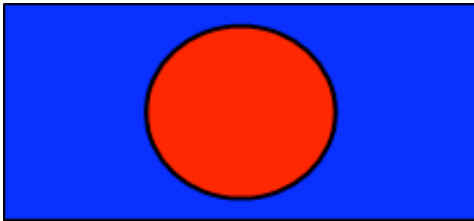
<Licenses>
    <ArrayLicenseWin>
        <ItemCount>1</ItemCount>
        <Item1>C:\MyLicenses\4DFAKELICENSENUMBER0123.html</Item1>
    </ArrayLicenseWin>
</Licenses>

```

SVG

Scalable Vector Graphics (SVG) is an XML-based language used to create rich, two-dimensional graphics. SVG is typically used to draw statistical data. While SVG documents can be modified using a text editor or by using the 4D XML commands, 4D v11 SQL has an integrated SVG engine and this allows developers to create SVG directly within the 4D method editor. Here is a simple example of an SVG file that creates a red circle within a blue rectangle.

Image:



SVG file:

```

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<svg xmlns="http://www.w3.org/2000/svg" id="ID_svg" viewBox="0 0 328
500" viewport-fill="white" viewport-fill-opacity="0"
xmlns:exslt4D="http://www.4D.com" xmlns:math="http://exslt.org/math"
xmlns:xlink="http://www.w3.org/1999/xlink">

    <rect width="200" height="100" style="fill:rgb(0,0,255);stroke-width:1;
stroke:rgb(0,0,0)"/>

    <circle cx="100" cy="50" r="40" stroke="black" stroke-width="2"
fill="red"/>

</svg>

```

Also, SVG is one of the supported picture types in 4D v11 SQL. So an SVG picture file can be used and displayed just like any other picture file.

XLIFF and the 4D Pop component

XLIFF is an XML based format that was developed to standardize localization. In 4D v11 SQL, all strings, strings lists, and text resources should now be stored in XLIFF files. Per the Sun Developers website, the XLIFF format helps to:

- Separate localizable text from formatting.
- Enable multiple tools to work on source strings and add to the data about the string.
- Store information that is helpful in supporting a localization process.

The following is a simple example of an XLIFF document that can be used to translate a few words into French

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE xliiff PUBLIC "-//XLIFF//DTD XLIFF//EN" "http://www.oasis-
open.org/committees/xliiff/documents/xliiff.dtd">
<xliiff version="1.0">
  <file datatype="x-STR#" original="undefined" source-language="en"
    target-language="fr">
    <body>
      <group id="1000" resname="Interface">
        <trans-unit id="1" resname="Button_OK">
          <source>OK</source>
          <target>Valider</target>
        </trans-unit>
        <trans-unit id="2" resname="Button_Cancel">
          <source>Cancel</source>
          <target>Annuler</target>
        </trans-unit>
      </group>
    </body>
  </file>
</xliiff>
```

XLIFF files are XML documents that conform to the XLIFF DTD. Line 2 in this example specifies the location of the DTD and the parser within 4D uses this to validate the XLIFF file. XLIFF files can be modified by commands from the 4D XML theme or from any text editor. The structure of an XLIFF file is relatively straightforward and makes adding/modifying elements relatively simple. For example, to add another string to the above file, “recherche” for the English word “search”, simply add the following block to the <group> tag:

```
  <trans-unit id="3" resname="Button_Search">
    <source>Search</source>
    <target>Recherche</target>
  </trans-unit>
```

While you can use the 4D XML commands to modify XLIFF files, a solution has already been developed and made into a component as part of 4D Pop. 4D Pop is a series of components that are free to use and distribute. The source code is also provided with each component. The 4D Pop XLIFF component provides a localization editor to manage XLIFF resources and helps migrate string resources from the 2004 resource architecture to XLIFF files.

Macros

A macro can be thought of as an instruction that automates a series of tasks. Macros can be used within 4D to carry out tasks such as: automatically adding comments to methods, automatically adding timestamps to methods, inserting code blocks, and writing to log files. Here is an example from the Web 2.0 pack that uses a macro to insert a block of code within a method.

Here is the XML file:

```
<?xml version="1.0" encoding="windows-1252" standalone="no" ?>
<!DOCTYPE macros SYSTEM "http://www.4d.com/dtd/2007/macros.dtd" >
<macros>
    <macro name="-">
        <text>
        </text>
    </macro>
    <macro name="DAX On Web Connection">
        <text>
C_TEXT($1;$2;$3;$4;$5;$6)
C_TEXT($url_t;$header_t;$clientIP_t;$serverIP_t;$user_t;$pass_t)

$url_t:=$1
$header_t:=$2
$clientIP_t:=$3
$serverIP_t:=$4
$user_t:=$5
$pass_t:=$6

Case of

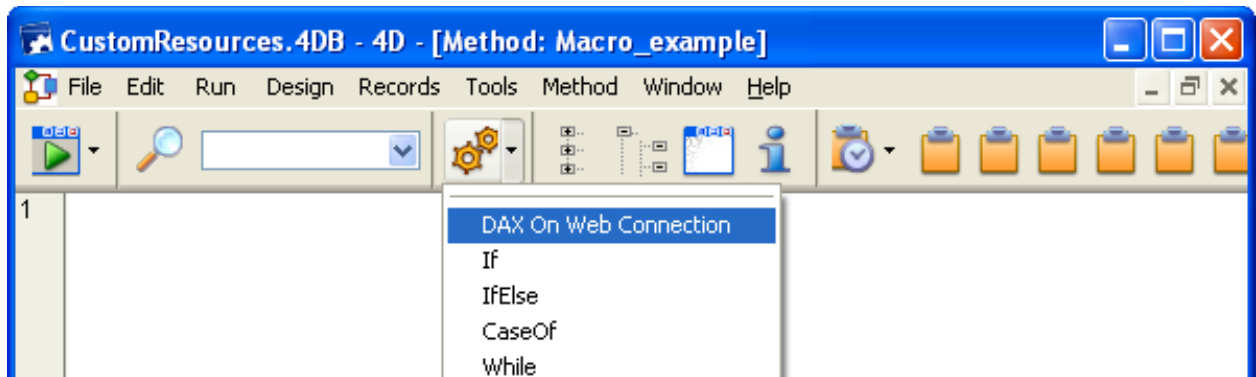
: (($url_t="DAX@") | ($url_t="/DAX@"))
DAX_Dev_OnWebConn
($url_t;$header_t;$clientIP_t;$serverIP_t;$user_t;$pass_t)

Else
` handle non-DAX requests in other cases or here
`-----

End case
        </text>
    </macro>
</macros>
```

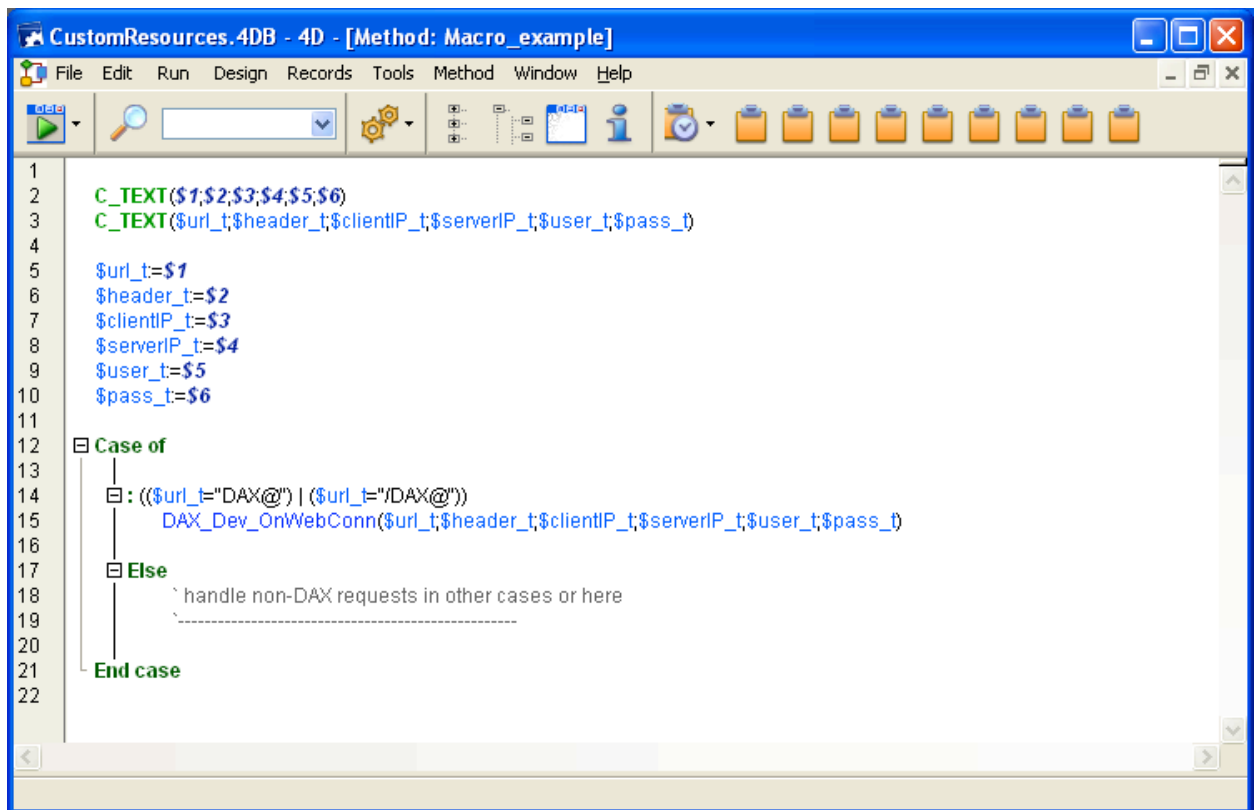
As in the previous example, the XML document must be validated by a DTD. When this specific macro is present, the code block can be inserted into any method by either:

- 1) Selecting the “DAX On Web Connection” from the macros object in the method toolbar.



- 2) Typing “DAX On Web Connection” and then typing the Tab key.

After following either one of these methods, the following will be inserted into the method:



4DLink file

A 4DLink file is a new XML based file introduced in 4D v11 SQL. 4DLink files are database access files that allow developers to control different options of how a database is opened. For example, developers can change: the structure file used, the data files used, whether to create a new data file, whether to connect to a remote 4D server, and what IP and port to use. Here is an example from a recent Tech Note on 4DLink,

The following example opens the local database using the structure file "C:\4D\Test DB's\contacts\Contacts final\Contacts.4DB" and the data file "C:\4D\Test DB's\contacts\Contacts final\Contacts.4DD", in the Maintenance and Security Center (MSC):

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<database_shortcut data_file="C:\4D\Test DB's\contacts\Contacts
final\Contacts.4DD" open_tools="true" structure_file="C:\4D\Test
DB's\contacts\Contacts final\Contacts.4DB"/>
```

The 4DLink file include just one root element that can have any number of different attributes to specify how the database is opened.

CONCLUSION

XML is a very broad topic, with many subtopics, and numerous examples, that can fill volumes.

This session is meant to provide a brief introduction of XML and how to take advantage of XML within 4D. The samples provided are meant show you how knowledge and use of XML within 4D can enhance your applications and operations. For more information, here are a few resources that were consulted in creating this document, as well as some 4D resources:

External resources:

- Beginning XML 2nd Edition by David Hunter
- www.w3.org/XML
- www.xml.com

4D resources:

Tech Notes

- 50815: 4D v11 SQL Database Access Files (4DLINK)
- 50517: XML Structure Import/Export in 4D v11 SQL
- 48977: Building Applications Examples
- 47934: Macros in 4D v11 SQL
- 47733: SVG Charts in 4D v11 SQL
- 46866: The XML Tools Component
- 42553: Difference between DOM and SAX
- 41862: Generic Full Data Export-Import Routine
- 30583: XML – An Introduction to Extensible Markup Language

Components and Sample Databases

- 4D Pop: www.4d.com/support/learning/databasesamplev11.html
- XML: www.4d.com/support/learning/howdoi-v11.html