
Managing XML Documents with Biferno

The xmlDoc and xmlNode Classes

Valerio Ferrucci, Tabasoft Sas

Table of Contents

1. xml Extension . .	1
1.1. Versions and Requirements . .	1
1.2. Installation . .	1
1.3. Compiling from sources . .	2
2. Opening and Validating . .	3
3. Navigating the tree of a document . .	4
4. Creating a New XML Document . .	6
5. SetTreeFromString Method . .	8
6. Processing Instruction . .	9
7. Errors . .	10
References . .	11

Abstract

Thank to the new Biferno xmlDoc class it is now possible to open, create, modify and validate XML documents. Besides, it is also possible to navigate their hierarchical structure and to get/set content and attribute values combining *xmlDoc* with the other Biferno XML class, *xmlNode*. *xmlDoc* and *xmlNode* are declared by the *xml* Biferno extension.

1. xml Extension

1.1. Versions and Requirements

The first version of the *xml* extension available to developers is 1.1 and is packaged together with Biferno 1.1 (usually Tabasoft extensions have the same versions of the package they are distributed with). Biferno ≥ 1.1 is required to use the *xml* extension.

xml is based on the *libxml2* cross-platform library.

The extension is available to Linux, Windows and MacOSX developers. A MacOS classic version will not be delivered.

To use Biferno *xml* you need access to a computer (Linux, Windows or MacOSX operating system) with the following installed:

- Biferno version ≥ 1.1
- on Linux and MacOSX: *libxml2* (on Windows the library is statically linked into the extension binary)

Information and downloads for *libxml2* can be found at: <http://www.xmlsoft.org>

1.2. Installation

The xml extension must be in the Extensions folder of BifernoHome. The exact location depends on the platform:

1. **Linux:** /home/BifernoHome/Extensions/xml_bfr.so. The xml RPM installation package (available from Biferno 1.1) installs it to the right location and no additional operations are needed.
2. **MacOSX:** /Users/BifernoHome/Extensions/xml_bfr.so. The Biferno (>= 1.1) installer/updater copies it to the right location and no additional operations are needed.
3. **Windows:** [Program Files]/Biferno/BifernoHome/Extensions/xml_bfr.dll. The Biferno (>= 1.1) installer/updater copies it to the right location and no additional operations are needed.

If the xml extension is installed while Biferno is running, a complete reload of Biferno is needed to cause the loading of the new extension. Instructions to do a complete (from console or BifernoCtl) reload of Biferno can be found in [BFRIA].

1.3. Compiling from sources

This section gives useful information to developers who want to recompile the xml biferno extension from sources. If you just want to use the binary extension, skip this section.

- *Linux and MacOSX:*

The Makefile is located in: External/xml/Makefile. The libxml folder containing include headers is searched (by default) in the /usr/include/libxml2 folder. You can override this setting passing the INCLUDES variable to the command line, as in:

```
make INCLUDES=/usr/local/include/libxml2
```

This assuming that your libxml folder is in /usr/local/include/libxml2.

The library used is libxml2.so on Linux and libxml2.dylib on MacOSX (directive -lxml2), and it is searched in the usual libraries search path (note that this can be overridden by the LD_LIBRARY_PATH environment variable, see "Defining Environment Variables for Biferno" in [BFRIA]).

- *Win32:*

The xml extension is not compiled by default in the Biferno.dsw workspace. To compile it open the project: External/xml/xmlVC/xmlVC.dsw and build the extension.

The libraries needed are:

```
iconv_a.lib
libxml2_a.lib
zlib_a.lib
```

and they must be in the same folder as the other VisualC libraries.

The include directories to add to the search path for your environment (in tool->options->directories) are:

```
[path containing the lib folders]\LIBXML2-[version].WIN32\INCLUDE
[path containing the lib folders]\ICONV-[version].WIN32\INCLUDE
[path containing the lib folders]\ZLIB-[version].WIN32\INCLUDE
```

for example, on my system the paths are:

```
C:\PROGRAM FILES\LIBXML2-2.6.4.WIN32\INCLUDE
C:\PROGRAM FILES\ICONV-1.9.1.WIN32\INCLUDE
C:\PROGRAM FILES\ZLIB-1.1.4.WIN32\INCLUDE
```

(I downloaded the libraries in the C:\PROGRAM FILES\ folder).

Visit <http://www.zlatkovic.com/libxml.en.html> to obtain the needed libraries for Windows building.

2. Opening and Validating

We can open an existing and valid xml document just using the constructor of `xmlDoc`:

```
void xmlDoc(string filepath)
```

The only parameter needed is a file path (following Biferno conventions) of an XML document. For example, suppose that we have the following document (named "jclass.xml") in the same folder of the script calling the xml constructor (as an example we use a Biferno man page as our first XML document):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE bifdoc SYSTEM "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd">
<bifdoc>
<class implem="c">
  <name>jclass</name>
  <purpose><par>Calling java code from Biferno</par></purpose>
  <descr><par>Documentation will be soon available</par></descr>
  <note><par>Available for Linux,
  MacOSX and Windows from Biferno version 1.1</par></note>
</class>
<method static="no" visibility="public" varargs="yes">
  <pclass>jclass</pclass>
  <name>jclass</name>
  <prototype>
    <param_list>
      <param>
        <pclass>string</pclass>
        <name>className</name>
        <descr><par>n.a.</par></descr>
      </param>
    </param_list>
  </prototype>
  <descr><par>n.a.</par></descr>
</method>
</bifdoc>
```

Calling:

```
<?
myDoc = xmlDoc("jclass.xml")
?>
```

can lead us to two different results:

- If the document *validates*, the call completes successfully. In this way we have instantiated a new variable of class `xmlDoc` pointing to the "jclass.xml" document.

- If the document doesn't validate an error `Err_DocumentParseFailed` (of the class `xmlDoc`) is thrown.

(for the notion of *Validation*, please refer to [XMLNS]).

To avoid validation errors stopping the script execution we could use the `error.Resume` and `error.Suspend` biferno commands (see [BFRLG]). A better solution, nevertheless, is to verify validation before calling the constructor. This can be obtained using the static method `ValidateFile`:

```
static string ValidateFile(string path)
```

The `path` parameter is the location of the document that must be checked for validity and the return value depends on the validation result. If the return value is empty (`" "`), it means that the document is valid, otherwise the error message returned by the validator is returned.

As an example, let us remove the tag `</pclass>` at line 12 of our document (so that the tag `<pclass>` doesn't close properly). We can check that the return string of `ValidateFile` in this case is:

```
/Library/WebServer/Documents/jclass.xml:24: parser error :
Opening and ending tag mismatch: pclass line 12 and method

/Library/WebServer/Documents/jclass.xml:25: parser error :
Opening and ending tag mismatch: method line 11 and bifdoc

/Library/WebServer/Documents/jclass.xml:26: parser error :
Premature end of data in tag bifdoc line 3

Could not parse document /Library/WebServer/Documents/jclass.xml
```

To avoid the script breaking in case of validation failure, we could write code similar to the following:

```
<?
if NOT(valMsg = xmlDoc.ValidateFile("jclass.xml"))
    myDoc = xmlDoc("jclass.xml")
else
    $"File doesn't validate. Error: " + valMsg
?>
```

3. Navigating the tree of a document

After we have created an `xmlDoc` variable, we can navigate the document tree starting from the root element using the `root` property:

```
<?
myDoc = xmlDoc("jclass.xml")
$myDoc.root
?>
```

The output of the script is:

```
bifdoc
```

This tells us that the root node of the document is the `bifdoc` node. Note that the `root` property, as expected, returns a variable of class `xmlNode`.

Each node has a name and a content. The name can be accessed by the `name` property and it is used as the string representation of an `xmlNode` object. The content is stored in the `content` property.

As the DOM (Document Object Model, see [XMLNS]) dictates, each *node* in a document can have 0 or more nodes in a deeper level (*children*) and 0 or more nodes at the same level. These can be reached in Biferno using the `children` and `next` properties of `xmlNode`. In particular, the `children` property returns the first child of the node it is called upon. The other children can be obtained calling `next` on the node returned by the `children` property.

As an example, we examine the following code that prints the names of all the nodes contained in our "jclass.xml" document:

```
<pre>
<?
function Print(string str, int indent)
{
    for (i = 0; i < indent; i++)
        $'\t'
    $str + "\r\n"
}
function WalkOnNodes(xmlNode cur, int indent)
{
    do
    {
        Print(cur.name, indent)
        if (cur.children)
            WalkOnNodes(cur.children, indent + 1)
    } while(cur = cur.next)
}
myDoc = xmlDoc("jclass.xml")
Print(myDoc.root, 0)
WalkOnNodes(myDoc.root.children, 1)
?>
</pre>
```

The program starts from the root node, then calls the `WalkOnNodes` function on the root's children. `WalkOnNodes` prints the names of the children. Besides, it calls itself recursively if the current node has other children. The indenting is only for display purpose.

When there are no more nodes, the `next` property returns a null node. The code above exploits this behaviour in order to know when to exit from the while loop (`xmlNode.children` also returns a null node when no more nodes are available).

Despite the simplicity of the code (due to the recursive structure), its output looks a little more complicated than one would expect:

```
bifdoc
text
class
    text
    name
        text
        text
        purpose
            par
                text
            text
            descr
                par
                    text
                text
                note
                    par
                        text
        text
text
method
    text
    pclass
```


added

It's time to see an example:

```
<?
myNewDoc = xmlDoc.New("1.0")
root = myNewDoc.NewRoot("bifdoc")
myNewDoc.Save("jclass_new.xml", "ISO-8859-1", true)
?>
```

This program creates an empty XML doc, adds a root element to it and saves it to disk. The Save method:

```
int Save(string path, string encoding, boolean indent)
```

writes the file on disk in the *path* location. The value of the *encoding* parameter (optional) is written in the xml declaration. If the *indent* parameter is true, the xml tags are written to the file indented with N tabulator, where N is the children-level of the tag. The file created by this code ("jclass_new.xml") is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bifdoc/>
```

We can add a *DTD* ("Document Type Definition") declaration to the XML document using the following method:

```
void ExtSubset(string rootName, string publicID, string systemID)
```

The three parameters are: the root name ("bifdoc" in our case), the public ID and the system ID of the DTD (see [XMLNS] for details about public IDs and system IDs). Let's add the DTD declaration to our code:

```
<?
myNewDoc = xmlDoc.New("1.0")
root = myNewDoc.NewRoot("bifdoc")
myNewDoc.ExtSubset("bifdoc", ,
    "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd")
myNewDoc.Save("jclass_new.xml", "ISO-8859-1", true)
?>
```

Consequently, our document becomes:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE bifdoc
    PUBLIC "" "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd">
<bifdoc/>
```

Now let's add a root child to our document with name `class`, with an attribute `implem` with value `c` and four children: `name`, `purpose`, `descr` and `note`. The last three also have one children each named `para` (yes, we are trying to rebuild our initial file "jclass.xml" from scratch!).

```
<?
myNewDoc = xmlDoc.New("1.0")
root = myNewDoc.NewRoot("bifdoc")
myNewDoc.ExtSubset("bifdoc", ,
    "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd")

classNode = root.NewChild("class")
classNode.NewAttr("implem", "c")

nameNode = classNode.NewMixedChild("name", "jclass")
```

```

purposeNode = classNode.NewChild("purpose")
purposeContentNode = purposeNode.NewMixedChild("par",
    "Calling java code from Biferno")

descrNode = classNode.NewChild("descr")
descrContentNode = descrNode.NewMixedChild("par",
    "Documentation will be soon available")

noteNode = classNode.NewChild("note")
noteContentNode = noteNode.NewMixedChild("par", "Available for Linux,
    MacOSX and Windows from Biferno version 1.1")

myNewDoc.Save("jclass_new.xml", "ISO-8859-1", true)
?>

```

With the `xmlNode` method:

```
void NewAttr(string attrName, string attrContent)
```

We can add an attribute to a node. The two `xmlNode` methods:

```
xmlNode NewChild(string name)
xmlNode NewMixedChild(string name, string content)
```

add new nodes to the document. The new nodes are children of the node the method is called upon. The first method adds a node with no text content (it contains only other nodes), the second can be used to add a node together with text content. In the same way we can add a root element to a document (having text content or not) using one of the two methods:

```
xmlNode NewRoot(string name)
xmlNode NewMixedRoot(string name, string content)
```

The document saved by our last code is:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE bifdoc PUBLIC ""
    "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd">
<bifdoc>
  <class implem="c">
    <name>jclass</name>
    <purpose>
      <par>Calling java code from Biferno</par>
    </purpose>
    <descr>
      <par>Documentation will be soon available</par>
    </descr>
    <note>
      <par>Available for Linux,
        MacOSX and Windows from Biferno version 1.1</par>
    </note>
  </class>
</bifdoc>

```

5. *SetTreeFromString* Method

The `SetTreeFromString` method from the `xmlNode` class allows a string containing xml markup to be added to an XML document, as another node's child.

Let's return to the initial simple code:

```
<?
```

```
myNewDoc = xmlDoc.New("1.0")
root = myNewDoc.NewRoot("bifdoc")
myNewDoc.ExtSubset("bifdoc", ,
    "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd")
myNewDoc.Save("jclass_new.xml", "ISO-8859-1", true)
?>
```

This code creates the document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE bifdoc
    PUBLIC "" "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd">
<bifdoc/>
```

Now suppose that we already have the string:

```
<class implem="c">
  <name>jclass</name>
  <purpose>
    <par>Calling java code from Biferno</par>
  </purpose>
  <descr>
    <par>Documentation will be soon available</par>
  </descr>
  <note>
    <par>Available for Linux,
      MacOSX and Windows from Biferno version 1.1</par>
  </note>
</class>
```

and we want to add it to the document in one step. The following code will do just that:

```
<?
myNewDoc = xmlDoc.New("1.0")
root = myNewDoc.NewRoot("bifdoc")
myNewDoc.ExtSubset("bifdoc", ,
    "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd")
xmlstring = '<class implem="c">
  <name>jclass</name>
  <purpose>
    <par>Calling java code from Biferno</par>
  </purpose>
  <descr>
    <par>Documentation will be soon available</par>
  </descr>
  <note>
    <par>Available for Linux,
      MacOSX and Windows from Biferno version 1.1</par>
  </note>
</class>'
classNode = root.SetTreeFromString(xmlstring, &errorMsg)
myNewDoc.Save("jclass_new.xml", "ISO-8859-1", true)
?>
```

The content of the `xmlstring` string is processed and the corresponding nodes are added to the document, obtaining the same document we obtained at the end of the previous section.

In order to obtain this, the content of `xmlString` must validate in the current document context. If this is not the case an error `Err_ParseMemoryFailed` is thrown and the `errorMsg` variable contains a message explaining the reason of validation failure.

6. Processing Instruction

The `xmlNode` method:

```
xmlNode NewPI(string name, string content)
```

adds a Processing Instruction node (see [XMLNS]) to the document and returns an `xmlNode` variable pointing to the node just added.

As an example, the following code adds a processing instruction node (an *xsl-stylesheet*) to our document:

```
<?
myNewDoc = xmlDoc.New("1.0")
root = myNewDoc.NewRoot("bifdoc")
piNode = root.NewPI("xml-stylesheet",
    'href="../web/sp2html.xsl" type="text/xsl"')
myNewDoc.ExtSubset("bifdoc", ,
    "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd")
myNewDoc.Save("jclass_new.xml", "ISO-8859-1", true)
?>
```

so that our document becomes:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE bifdoc PUBLIC ""
    "http://www.tabasoft.it/biferno/bfdocdtd/bifdoc.dtd">
<xml-stylesheet href="../web/sp2html.xsl" type="text/xsl"?>
<bifdoc/>
```

7. Errors

`xmlDoc` methods and properties can throw the following errors:

- `Err_DocumentParseFailed`: Is returned by the `xmlDoc` constructor when the XML document pointed to by the `path` parameter is not a valid XML file (this error is related to `libxml2`'s `xmlParseFile` failing).
- `Err_EmptyDocument`: If an XML document is empty, accessing the `root` property causes this error.
- `Err_SaveFailed`: The error is thrown when the `Save` method fails. Generally this is related to folder permissions, make sure that the `biferno` user has permission to create files in the folder containing the file specified by the `path` parameter. This error is related to `libxml2`'s `xmlSaveFormatFileEnc` failing.
- `Err_NewDocumentFailed`: the `New` method can throw this error in case of failure. This error should arise only when not enough memory remains on the system to create the new document in memory (this error is related to `libxml2`'s `xmlNewDoc` failing).
- `Err_NewDTDFailed`: this error is thrown by the `ExtSubset` method when the DTD declaration can not be added to the document (this error is related to `libxml2`'s `xmlNewDtd` failing).
- `Err_AddChildFailed`: can be caused by `NewRoot`, `NewMixedRoot` or `ExtSubset` when the corresponding children node (the root or the external subset, i.e. the DTD node) can not be added (this error is related to `libxml2`'s `xmlAddChild` failing).
- `Err_AddPrevSiblingFailed`: can be thrown by `ExtSubset` when `libxml2`'s `xmlAddPrevSibling` method, used to add the node as first node of the document, fails.
- `Err_NewDocNodeFailed`: can be thrown by `NewRoot` or `NewMixedRoot` when the new

root node could not be added (this error is related to libxml2's `xmlNewDocNode` failing).

The `xmlNode` methods and properties can throw the following errors:

- `Err_NullNode`: is thrown when a method is applied on a null node or a property of a null node is requested.
- `Err_CantCreateAttribute`: is thrown by `NewAttr` when libxml2's `xmlNewProp` fails.
- `Err_CantCreateNode`: thrown by `NewChild`, `NewMixedChild` or `NewPI` when libxml2's `xmlNewChild` or `xmlNewPI` fails.
- `Err_UTF8ConversionFailed`: strings are always encoded to UTF-8 before being passed to the libxml2 library. This error is thrown when the encoding fails (the `islat1ToUTF8` function is used)
- `Err_IsolatInConversionFailed`: strings returned by the libxml2 library are always decoded to ISOLATIN before being returned to the user. This error is thrown when the decode fails (the `UTF8Toislat1` function is used)
- `Err_DocGetRootElementFailed`: this error is thrown by `SetTreeFromString` if an error occurred while attempting to get the root element of the XML tree created from the `xmlstring` parameter (libxml2's `xmlDocGetRootElement` failed).
- `Err_AddChildFailed`: this error is thrown by `SetTreeFromString` if an error occurred while attempting to add the new tree (created from the `xmlstring` parameter) to the node the method is applied to (libxml2's `xmlAddChild` failed).
- `Err_CopyNodeFailed`: this error is thrown by `SetTreeFromString` when an error occurred while attempting to create a copy of the root node of the tree (created from the `xmlstring` parameter) in order to add it to the node the method is applied to (libxml2's `xmlCopyNode` failed).
- `Err_ParseMemoryFailed`: this error is thrown by `SetTreeFromString` when an error occurred while validating the `xmlstring` parameter (libxml2's `xmlParseMemory` failed).
- `Err_AddPrevSiblingFailed`: can be thrown by `NewPI` when the libxml2's `xmlAddPrevSibling`, used to add the processing instruction node as first node, failed.

References

[BFRLG] Biferno Language Guide Tabasoft S.a.s.

[BFRIA] Biferno Installation and Administration Guide Tabasoft S.a.s.

[XMLNS] XML in a nutshell A Desktop Quick Reference Elliotte Rusty Harold W. Scott Means O'Reilly

[XMLAS] XML The Annotated Specification Bob DuCharme Prentice Hall PTR