OpenDoPE Flat OPC XML processing

Thursday, January 23, 2014

# Motivation

Microsoft Word 2013 supports binding "an escaped string comprised of a **flattened WordprocessingML document**" (see 2.5.1.6 dataBinding of  [MS-DOCX] Word Extensions to the Office Open XML (.docx) File Format).

A a **flattened WordprocessingML document**" is equivalent to a docx file, except that instead of being a zipped up collection of primarily XML files, it is a single XML file. Hereinafter referred to as **Flat OPC XML**.

This makes it possible to bind rich text content (tables, images etc).

However, only Word 2013 supports this.

It is therefore useful to provide a way to bind rich text which is compatible with Word 2010.

# Approach

The authoring tool (a Word AddIn) supports adding a tag in a content control (w:sdt), like so:

<w:sdt>

<w:sdtPr>

<w:tag w:val="od:xpath=HrRjw&amp;od:progid=Word.Document"/>

</w:sdtPr>

<w:sdtContent>

<w:p>

:

od:xpath points by ID to an XPath specified in the XPaths part; that XPath points to an element in your Custom XML part.

od:progid=Word.Document signifies that that element contains escaped Flat OPC XML.

At document generation run-time, docx4j recognises that tag, and if present, replaces the content in the w:sdtContent element with an altChunk element, like so:

<w:sdt>

<w:sdtPr>

<w:tag w:val="od:xpath=HrRjw&amp;od:progid=Word.Document"/>

</w:sdtPr>

<w:sdtContent>

<w:**altChunk** r:id="rId14"/>

</w:sdtContent>

</w:sdt>

If the resulting docx is opened in Word, Word resolves the altChunk, and (transparently to the user) replaces it with normal Word content, which can be edited in the usual way.

With MergeDocx, docx4j can itself convert the altChunks to normal Word content (just as Word does). This is convenient if you need to manipulate the content further, or export the docx to HTML or PDF without having first opened it in Word.

# Additional Capability

If a Word user edits the content of a normal plain text data bound content control, the new value is automatically written back to the bound element in the custom XML part.

It is desirable to have the same capability for rich text content controls. So for example, modified data can be persisted in some external system.

MergeDocx supports this capability. See further below.

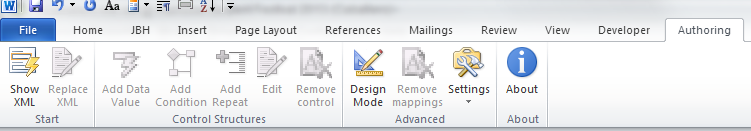
(It would also be possible to add this capability to the authoring tool, or indeed, to create a Word addin for end users which does this.)

# Authoring AddIn Walkthrough

Download and install the Word AddIn, from:

<http://www.opendope.org/downloads/authoring-friendly/setup.exe>

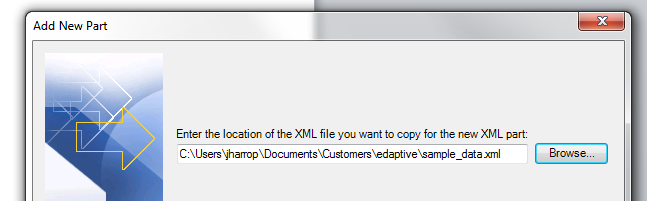
Once installed, you should see the following menu in Word:



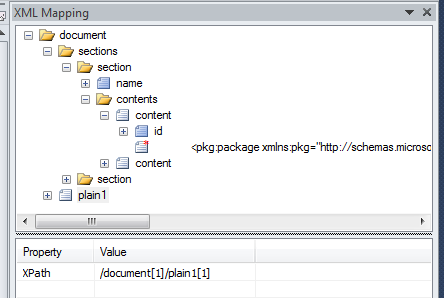
Press the “Show XML” button. This:

1. let’s you add your XML part to the docx
2. adds the other OpenDope custom XML parts
3. shows the task pane

For this walkthrough, use the provided XML file **sample\_data.xml**



You should then see the following task pane:

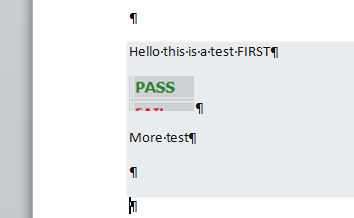


Here I’ve expanded some of the nodes (by clicking the plus symbol).

Notice that a content element contains “<pkg:package...”; that is Flat OPC XML.

To add it to your docx, simply drag the node onto the document surface.

The result should be:



(or whatever your Flat OPC contains; the authoring tool converts the Flat OPC to ‘real’ content, which is what you see).

The result is a rich text content control, with the tag described above. You’ll see it if you press the design mode button.

When you are done, save your docx and close it.

***sample\_template.docx*** *is a docx created by dragging content from* ***sample\_data.xml*** *as described above.*

Caveats/notes:

* The content control belongs at “block” level (ie like a paragraph or table). The AddIn will try to ensure this for you. So it can be a paragraph sibling, or you can drag it into a table cell.
* Do not drag a single element to the docx in two different locations, since a user could edit the resulting content controls differently, but only the content of one of them could later be persisted back to the XML part.
* Do not drag a Flat OPC content control into another Flat OPC content control.   
  This is unsupported.
* Do not include bound content controls in your Flat OPC, since when the Flat OPC is included in the docx by docx4j, any bound controls inside that Flat OPC will be ignored by docx4j (though not by Word).

# Creating your initial XML part

There is a bit of a chicken/egg problem in getting started.

It is this. You need your XML to contain escaped Flat OPC XML, in order for the authoring addin to detect the presence of Flat OPC XML, and add a suitable content control to your docx.

But how do you produce that XML file in the first place.

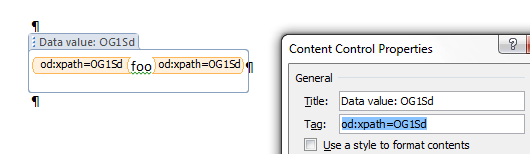
There are 2 methods.

Method 1 –

1. Create a docx file containing your desired rich text content
2. Save it as Flat OPC XML.
   1. in Word, save as “Word XML document”,
   2. OR use docx4j (see ConvertOutFlatOpenPackage sample)
3. In a text editor, manually escape it, by replacing ‘<’ with ‘&lt;’ and ‘>’ with ‘&gt;’
4. Copy the resulting content into your target XML structure

Method 1 can easily be automated.

Method 2 (not recommended) –

1. start with an XML file containing your desired element hierarchy, but with plain text content in the elements (as opposed to escaped Flat OPC XML)
2. drag an XML element to the document surface (creating a data bound plain text control, but also an XPath part entry)
3. using the Developer tab in Word (you may need to add/enable this tab), copy the content of the tag:  
     
   
4. Delete this plain text content control
5. Now, using the Developer tab again, insert a rich text control, then paste the content of the tag, then add &od:progid=Word.Document
6. Add rich text content inside the content control
7. Run the BindInverse code (see below) against the docx; this will convert your rich text content to Flat OPC XML, and put it into your Custom XML part
8. Extract the Custom XML part from the resulting docx, or copy it from the console output

The above is a bit painful; a future version of the AddIn could make this easier.

# Document Generation Run-Time Walkthrough

To bind XML containing instance data (eg created from a database query) to your template docx, run the following code with docx4j on your classpath:

String input\_DOCX = System.*getProperty*("user.dir") + "/sample\_template.docx";

String input\_XML = System.*getProperty*("user.dir") + "/**instance\_data.xml**";

// resulting docx

String OUTPUT\_DOCX = System.*getProperty*("user.dir") + "/OUT\_ContentControlsMergeXML.docx";

// Load input\_template.docx

WordprocessingMLPackage wordMLPackage = Docx4J.*load*(**new** File(input\_DOCX));

// Open the xml stream

FileInputStream xmlStream = **new** FileInputStream(**new** File(input\_XML));

// Do the binding:

// FLAG\_NONE means that all the steps of the binding will be done,

// otherwise you could pass a combination of the following flags:

// FLAG\_BIND\_INSERT\_XML: inject the passed XML into the document

// FLAG\_BIND\_BIND\_XML: bind the document and the xml (including any OpenDope handling)

// FLAG\_BIND\_REMOVE\_SDT: remove the content controls from the document (only the content remains)

// FLAG\_BIND\_REMOVE\_XML: remove the custom xml parts from the document

Docx4J.*bind*(wordMLPackage, xmlStream,

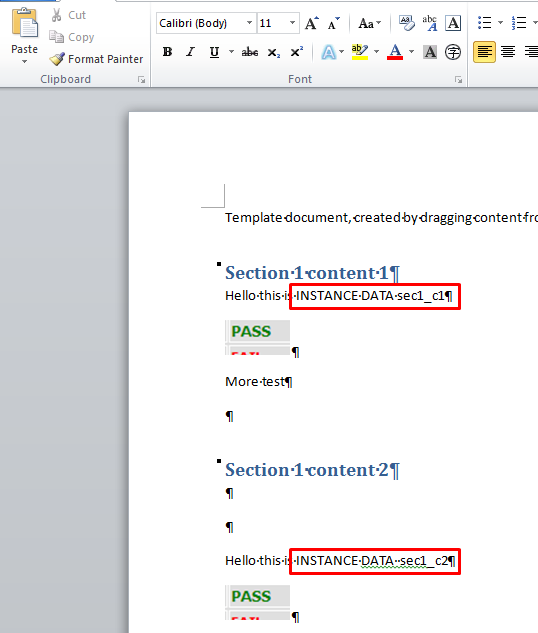
Docx4J.*FLAG\_BIND\_INSERT\_XML* | Docx4J.*FLAG\_BIND\_BIND\_XML*);

//Save the document

Docx4J.*save*(wordMLPackage, **new** File(OUTPUT\_DOCX), Docx4J.*FLAG\_NONE*);

As described above, that much will create an output document containing altChunk elements.

Opened in Word, you should see (without the red boxes):



If you’d like to resolve those altChunk elements to real WordML content, then with MergeDocx( v1.8.0 or later) on your path, run

wordMLPackage = (WordprocessingMLPackage) ProcessAltChunk.*process*(wordMLPackage);

# BindInverse: Instance document repopulating XML

If a user edits your instance docx, you can write the modified contents of the rich text controls which are bound to FlatOPC, back to the source elements in the Custom XML file.

To do that, with MergeDocx ( v1.8.0 or later) on your classpath:

String inputfilepath = System.*getProperty*("user.dir") + "/edited\_instance.docx";

WordprocessingMLPackage wordMLPackage = WordprocessingMLPackage.*load*(**new** java.io.File(inputfilepath));

**BindInverse** bindInverse = **new** BindInverse(wordMLPackage, **true**);

List<CustomXmlPart> updatedParts = bindInverse.updateCustomXmlParts();

**for**(CustomXmlPart cxp : updatedParts) {

System.*out*.println("\n\n\n");

System.*out*.println(cxp.getXML());

}

// If you want to save a docx containing the updated part(s)

String outputfilepath = System.*getProperty*("user.dir") + "/inverse\_OUT.docx";

wordMLPackage.save(**new** java.io.File(outputfilepath));

Caveats/notes:

* Only content controls in the Main Document Part are processed
* It is assumed the instance docx does not contain OpenDoPE conditions or repeats
* The MergeDocx sample OpenDopeBindInverse shows how the updated docx can itself be run through BindingHandler to produce a docx containing altChunks (a useful integrity check)
* Styles (and the numbering part) are not required in Flat OPC emitted by docx4j for that Flat OPC to be imported into another docx *based on those same styles/ndp*; but those styles & NDP **are** necessary for Word 2010 altChunk processing (if the fragment is to use those styles). Passing false to the **BindInverse** constructor gives smaller file sizes, but whether you should do this depends on your workflows