Generate high-quality Microsoft Word DOCX files using a simplified XML format (simple word processing XML).

Simple Word Processing XML (SWPX) makes it relatively easy to transform structured content into DOCX (or other similar word processing formats).

The Wordinator uses the Apache POI library [https://poi.apache.org/] to generate DOCX files from SWPX XML. It uses Saxon [http://saxonica.com/] for XSLT transformations when using the built-in XSLT support.

This approach provides a two-stage X-to-DOCX conversion process, where the first stage is a transform from whatever your input is into one or more SWPX documents and the second stage generates DOCX files from the SWPX files. You can think of the SWPX XML as a very abstract API to the DOCX format.

The Wordinator Java code can run an XSLT to generate the SWPX dynamically from any XML input or you can generate the SWPX documents separately using whatever means you choose and then process those into DOCX files. Word style definitions are managed using a normal Word template (DOTX) file that you create and manage normally.

The Wordinator is designed for batch or on-demand generation of DOCX files.

The Wordinator requires Java 8 or newer (because POI 4 requires it).

The Wordinator provides a generic HTML5-to-DOCX transform that can easily be adapted to your specific HTML or other XML format.

The main challenges are managing white space within text runs and mapping source elements to the appropriate paragraph and character styles. The XSLT has been designed to make the element-to-style mapping as easy as possible by using a separate XSLT mode to generate the style names for elements. This mode uses an XSLT 3 map to map HTML class values to Word style names and paragraph and run-level formatting controls (e.g., a @class token of 'bold' will result in bold runs). This makes configuring the mapping about as easy as it can be. If a simple class-to-style mapping is insufficient you can use normal XSLT templates to map elements in context to styles.

You can use your own XSLT transform to generate SWPX files from any XML (or JSON source for that matter). You may find it easier to generate HTML and then use that as input to the Wordinator.

If you need to go from Word documents back to XML, you may find the DITA for Publishers Word-to-DITA framework useful ([https://github.com/dita4publishers/org.dita4publishers.word2dita]). This packaged as a DITA Open Toolkit plugin but is really a general-purpose XML-to-DOCX framework. It does not depend on the DITA Open Toolkit in any way. While it is designed to generate DITA XML it can be adapted to produce any XML format, either directly or through a DITA-to-X transform applied

Release Notes

- 1.0.0
    - Section-specific running heads and feet, page geomentry, page numbers
    - Improved table generation:

        - Table spans should be 100% correct
        - Use table styles
        - Correct setting of row- and cell-level vertical spacing
        - Borders on tables and cells
        - Support "shade" property on cells (background color)
    - Fixed issue where runs after footnotes were dropped.
- 0.9.2
    - Control table borders for rowsep, colsep, and per-edge on individual cells
    - Handle absolute width values on tables
- 0.9.1
    - Out-of-the-box DITA HTML5 transform.
    - Handle unnamespaced HTML5
    - Added some useful documentation
    - Added command-line help
- 0.9.0

Working for XHTML input. DOCX pretty complete

- 0.8.0

Use final version of POI 4.0.0

- 0.7.0
    - Improved performance by only reading template doc once

Word feature support

The Wordinator supports generation of documents with the following Word features:

- Paragraphs and runs with specific styles
- Footnotes and end notes
- Tables with spans
- Embedded graphics
- Running heads and feet
- Bookmarks
- Hyperlinks
- Multiple sections (full support pending)

Getting Started

The Wordinator is packaged as a runable Java JAR file. It also requires an XSLT transform and a Word DOTX template in addition to your input file.

To try it you can use the basic XHTML- or HTML5-to-DOCX transform that is included in the Wordinator materials. For production use you will need to create your own transform that expresses the details of mapping from your XML or HTML to your styles. This can be pretty easy to implement though--you shouldn't normally need any significant XSLT knowledge.

Installation

Unzip the release package into a convenient location. The release includes the Wordinator JAR file and base XSLT tranforms, along with a generic Word template (as a convenience).

You need to be able run the java command using Java 8 or newer.

If you have ant installed you can run the Wordinator using the build.xml script in the root of the distributaion package (src/main/ant/build.xml in the project source).

Running the Wordinator With Ant

The build.xml file in the distribution provides two targets: html2docx and ditahtml2docx. The default target is ditahtml2docx.

If you just run the ant command from the Wordinator distribution directory it will run the ditahtml2docx target against the sample HTML file included in the distribution:

```
      c:\projects\wordinator> ant
Buildfile: /Users/ekimber/workspace/wordinator/dist/wordinator/build.xml

init:

ditahtml2docx:
     [java] + 2019-03-07 22:14:54,322 [INFO ] Input document or
directory='/Users/ekimber/workspace/wordinator/dist/wordinator/html/sample_web_page.ht
ml'
     [java] + 2019-03-07 22:14:54,324 [INFO ] Output directory
='/Users/ekimber/workspace/wordinator/dist/wordinator/out'
     [java] + 2019-03-07 22:14:54,324 [INFO ] DOTX template
='/Users/ekimber/workspace/wordinator/dist/wordinator/docx/Test_Template.dotx'
     [java] + 2019-03-07 22:14:54,324 [INFO ] XSLT template
='/Users/ekimber/workspace/wordinator/dist/wordinator/xsl/ditahtml2docx/ditahtml2docx.
xsl'
     [java] + 2019-03-07 22:14:54,325 [INFO ] Chunk level               ='root'
...
    [java] + 2019-03-07 22:14:55,759 [INFO ] Generating DOCX file
"/Users/ekimber/workspace/wordinator/dist/wordinator/out/sample_web_page.docx"
     [java] + 2019-03-07 22:14:56,249 [INFO ] Transform applied.

BUILD SUCCESSFUL
Total time: 4 seconds
```

Edit the build.xml file to see the properties you can set to specify your own values for the command-line parameters.

You can create a file named build.properties in the same directory as the build.xml file to set properties statically or you can specify them using -D parameters to the ant command:

```
c:\projects\wordinator> ant -Dditahtml2docx.dotx=myTemplate.dotx
```

Running the Wordinator From OxygenXML

You can set up an Oxygen Ant transformation scenario and apply it against HTML files to generate DOCX files from them.

To set up a transformation scenario follow these steps:

1. Open an HTML file in OxygenXML
2. Open the Configure Transformation Scenarios dialog
3. Select "New" and then "Ant transformation"
4. Give the scenario a meaningful title, i.e. "DITA HTML to DOCX"
5. In the "Build file" field put the path and name of the build.xml file. Take the defaults for the other fields in this tab.
6. Switch to the "Parameters" tab and add the following parameters:
   - input.html: ${cfd}/${cfne}
   - output.dir: ${cfd}/out
   - html2docx.dotx: Path to your DOTX file
   - html2docx.xsl: Path to your XSLT (if you have one, otherwise omit)
7. Switch to the "Output" tab and set the Output field to ${cfd}/out/${cfn}.docx. Make sure that "Open in system application" is selected.

You can omit any of the parameters that you have set using a build.properties file.

You should now be able to run the scenario against any HTML file and have the resulting DOCX file open in Microsoft Word.

Running the Wordinator From The Command Line

1. Open a command window and navigate to the directory you unzipped the Wordinator package into:

   ```
   cd c:\projects\wordinator
   ```

2. Run this command:

   ```
   java -jar wordinator.jar -i html/sample_web_page.html -o out -x
xsl/html2docx/html2docx.xsl -t docx/Test_Template.dotx
   ```

You should see a lot of messages, ending with this:

```
+ 2019-03-07 16:58:33,873 [INFO ] Generating DOCX file
"/Users/ekimber/workspace/wordinator/dist/wordinator/out/sample_web_page.docx"
```

```
+ 2019-03-07 16:58:34,406 [INFO ] Transform applied.
```

3. Open the file out\sample_web_page.docx in Microsoft Word

It's not a very pretty test but it demonstrates that the tool is working.

Wordinator Commandline Options

- -i The input XML file or directory
- -o The output directory
- -t The DOTX Word template
- -x The XSLT transform to apply to the input file to generate SWPX files.

If the -i parameter is a directory then it looks for *.swpx files and generates a DOCX file for each one.

Adapting Wordinator To Your Needs

The base HTML-to-DOCX transform is very basic and is not intended to be used as is.

To create good results for your content you will need the following:

- A Word template (DOTX) that defines the named styles you need to achieve your in-Word styling requirements. For many documents the built-in Word styles will suffice. You may also have existing templates that that you need to map to. The important thing for the mapping to Word is the style names: the mapping from your input XML to Word is in terms of named paragraph, character, and table styles.
- A custom XSLT style sheet that implements the mapping from your input XML to Simple Word Procesing XML that is then the input to the DOCX generation phase. A a minimum you need to provide the mapping from element type names and @class values to paragraph and character style names. This can be done with a relatively simple XSLT module that overrides the base HTML-to-DOCX transform.
- The XML from which you will generate the Word documents. This can be any XML but the Wordinator-provided transforms are set up for XHTML and HTML5, so if you are either authoring in HTML5 or you can generate XHTML or HTML5 from your XML then the transform is relatively simple. For example, the provided ditahtml2docx transform handles the HTML5 produced by the DITA Open Toolkit.

Java Integration

The release package uses a jar that contains all the dependency jars required by the Wordinator.

However, if you want to include the Wordinator in a larger application where the dependencies should be managed as separate JAR files, you can build the JAR from the project source.

The Wordinator project is a Maven project.

SimpleWP XML (SWPX)

The Simple Word Processing XML format is the direct input to the DOCX generation phase of the Wordinator.

It is essentially a simplification of Word's internal XML format.

The SWPX format is defined in the simplewpml.rng file in the doctypes/simplewpml directory. The RNG file includes documentation on the SWPX elements and attributes and how to use them.

The XSLT file xsl/html2docx/baseProcessing.xsl does most of the work of generating SWPX from HTML and it also serves to demonstrate how to generate SWPX if you want to implement direct generation from some other XML format.

If you are generating SWPX files be sure to validate them against the simplewpml.rng grammar. One easy way to do this is to use Oxygen to associate the RNG with the the SWPX file using the Document -> Schema -> Associate Schema menu.

Table Spans and Column Widths

The Wordinator supports complex table spans and will correctly calculate the width of cells that span multiple columns.

However, there is a limitation in how the table's column widths are specified: All the values involved in calculating the width of a spanned cell must be of the same type, either all explicit widths or all percentage widths.

This is because at the time the table is generated Wordinator does not know how wide the table will be and therefore cannot convert a mix of absolute and percentage values into absolute values.

When all the values are percentages the resulting Word is generated with percentage values, allowing Word to correctly calculate the widths of the cells. When all the values are absolute then calculation of the spanned width is simple math.

As a rule, it is best to use percentages for table column widths.

If you have tables with a mix of percentage and absolute values for column widths and you have cells that span columns where the widths involved are mixed, Wordinator issues a warning message. The resulting table will likely not be correct.

Customizing the HTML-to-SWPX Transforms

The module xsl/html2docx/get-style-name.xsl implements the default mapping from HTML elements to style names. It uses a variable that is a map from @class attribute values to Word style names:

```
        <xsl:variable name="classToStyleNameMap" as="map(xs:string, xs:string)">
    <xsl:map>
      <xsl:map-entry key="'p1'" select="'Paragraph 1'"/>
    </xsl:map>
```

```
  </xsl:variable>
```

Each <xsl:map-entry> element maps a @class name (key="'p1'") to a style name (select="'Paragraph 1'").

You can override this variable in a custom XSLT to add your own mapping.

Note that the values of the @key and @select attributes are XSLT string literals: 'p1' and 'Paragraph 1'. Note the straight single quotes (') around the strings. If you forget those your results will be strange.

The map variable is used like so:

```
        <xsl:template mode="get-style-name" match="xhtml:span[@class] |
xhtml:p[@class]" as="xs:string?">
    <xsl:param name="doDebug" as="xs:boolean" tunnel="yes" select="false()"/>

    <xsl:variable name="tokens" as="xs:string*" select="tokenize(@class, ' ')"/>
    <xsl:variable name="key" select="$tokens[1]"/>
    <xsl:variable name="styleName" as="xs:string?"
      select="map:get($classToStyleNameMap, $key)"
    />
    <xsl:sequence select="if (exists($styleName)) then $styleName else ()"/>
  </xsl:template>
```

Here, the @class attribute of the element that matches the template is tokenized on blank spaces and then the first value is used to look up an entry in the $classToStyleNameMap variable.

TBD: More guidance on customizing the mapping. Would also be easy to implement using a JSON file to define the mapping as a separate configuration file.

Managing Word Styles

The Wordinator requires a Word template document (DOTX) that defines the styles available in the generated Word document.

To create and manage styles use this general procedure:

1. Create a Word document with the styles you need. For every style, whether built-in or custom, create at least one object (paragraph, character run, table, etc.) that uses the style.
2. Save the document as a DOTX (Word template document). This will be the template you provide to the Wordinator.
3. To add or modify styles, create a new document from the DOTX file. Going forward you will use this new file to create new styles or modify existing styles.
4. When you create or modify styles in the document, be sure to check the "Add to template" check mark on the style dialog. This will cause the template document

to be updated with the new style information when you save the document you are editing.

Using the Style Organizer

If you forget to do "Add to template" or you want to copy styles from an existing Word document, you can use the style organizer.

To get to the style organizer:

1. Select Tools->Templates and Add-ins to bring up the Tools and Add-ins dialog The dialog shows the template associated with the document. If your template is not attached, use the Attach button to attach it. Make sure the "Automatically update document styles" check box is checked.
2. Click the "Organizer" button to open the Organizer dialog. Select the "Styles" tab
3. The right side of the Organizer dialog shows the template document to which you will copy styles. It probably shows the default template. If so, click "Close file" and then click "Open file" and select your DOTX file.
4. Use the Organizer dialog to copy styles from the left side to the right side.
5. Click "Close" to save your changes to the template document.

Support, New Feature Development, and Contributing

The Wordinator project is supported primarily by paying clients who fund development of the features they need. Initial development was funded by Municode [http://municode.com].

Please use this project's issue tracker to report bugs or request new features.

I (Eliot Kimber) will attempt to fix bugs as quickly as possible.

For new features, it is unlikely that I will be able to implement them outside of a paid engagement, but if it's something generally usable or something one of my clients needs I may be able to implement it.

If you would like to contribute new features, I welcome all contributions. Use normal GitHub pull requests to submit your contributions. If you'd like to be more heavily involved or even take over primarily development, please contact me directly.

Building

This is a Maven project.

NOTE: POI 4.x and this project require at least Java 8.

Maven dependency:

```
<dependency>
  <groupId>org.wordinator</groupId>
  <artifactId>wordinator</artifactId>
  <version>1.0.0</version>
</dependency>
```

Release

To deploy to the public Maven repository use this command line:

```
mvn clean deploy
```

(Note: Only the project owner can do this as it requires being able to sign the jar.)