Using Pandoc Lua scripts with LaTex

J. Madgwick

02/07/2021

Contents

Introduction

Pandoc is a popular utility for converting markup file formats and can also power publishing workflows. It has been around for quite some time and it's capabilities now extend far beyond approximate document format conversion[1].

This article will explain how Pandoc can be used to convert some documents while altering the structure and formatting at the same time. LaTex is then used to create polished PDF documents without any additional manual editing of LaTex markup.

[1] Though that's exactly how this page was written. I used LibreOffice to write the article, converted the ODT to markdown, added the code snippets and then converted that into HTML which is inserted into a template page using PHP.

Installing dependencies

Pandoc needs to be installed[2] and XeTex is also required[3]. Most of the LaTex covered is not specific to XeTex and will work fine with pdftex (also installed with XeTex).

Assumptions

Throughout we'll be assuming that the source document is markdown (`.md`). Depending on the input document format, certain Pandoc features are not available. For example, the way of passing in LaTex commands described later doesn't work when using ODT documents as a source[4].

Some knowledge of LaTex and to an extent Pandoc would be useful. This useful article introduced me to Pandoc.

How does Pandoc work?

In order to use Pandoc to it's full potential we need to learn how it works and how we can get involved in that process.

Pandoc essentially reads the input file and parses it, extracting elements it wants (headings, text) and ignoring what it doesn't (certain styling information, format specific metadata). These elements are then stored in the Pandoc native format. This is an intermediary format used only inside Pandoc. An example follows:

```
[Header 1 ("i-am-a-level-1-header",[],[]) [Str "I",Space,Str
 ↪  "am",Space,Str "a",Space,Str "level",Space,Str "1",Space,Str
 ↪  "header"]
,Header 2 ("im-a-level-2-header",[],[]) [Str "I\8217m",Space,Str
 ↪  "a",Space,Str "level",Space,Str "2",Space,Str "header"]
,Para [Str "This",Space,Str "is",Space,Str "a",Space,Str
 ↪  "sentence.",Space,Str "Here\8217s",Space,Str "some",Space,Code
 ↪  ("",[],[]) "inline code",Str ".",Space,Str "This",Space,Str
 ↪  "could",Space,Str "be",Space,Str "a",Space,Str "paragraph."]]
```

Conversion to the output format is done by taking a template for that format and adding in the information stored in the native format. Depending on the targetted output format, a template can be optional. E.g. for HTML the default approach is to create just text and markup without the headers required for a stand-alone page.

For PDF output, the Pandoc native representation is first converted to LaTex with a default template. This Tex is then converted to PDF with pdftex (the default) or other engines (XeLaTex).

---

[2] Version 2.14.0.3 was the latest at the time of writing, although everything here should also works with earlier versions. To install on a Debian like system, run `sudo apt install pandoc`

[3] XeTex is a modern LaTex typesetting engine. It can use normal system fonts which allow for much better handling of non Latin characters or symbols. To install on a Debian like system, run `sudo apt install texlive-xetex`

[4] This is because the 'raw_attribute' Pandoc functionality is not enabled for ODT documents, more information here: https://groups.google.com/g/pandoc-discuss/c/ffSJBNFX8q4 .

Step 1: Customizing the default conversion settings

When converting to PDF from any format[5] the results will use the default LaTex style for 'article' and a paper size of US Letter. To customize this a YAML metadata file can be included[6] (when converting from markdown, this can instead be included in the source file). A file doesn't need to be used for simple options - they can be passed on the command line (using `-M KEY [=VAL]`). Although not documented officially, passing multiple values to the same key does not appear to work on the command line, the rest of this article assumes a metadata file is used.

A lot of customization can be done using just the variables Pandoc makes available[7].

```
title: Example Title
author: Example Author
classoption:
- oneside
- a4paper
geometry:
- lmargin=20mm
- rmargin=20mm
- tmargin=20mm
- bmargin=25mm
fontsize: 12pt
```

The above YAML specifies the document title and author (in the document itself and in the metadata[8]) plus LaTex specific options for the document class to use A4 single sided, margins for the page (via the geometry package which is used by default) and the default font size.

Under the covers all of these options set LaTex headers in the template. This can be seen by changing the output format to 'tex' and using the stand-alone option (e.g. `pandoc example.md -s -o out.tex`) and looking at the resulting Tex file.

As above, Pandoc includes the most common LaTex header configuration options as variables. For setting other options and using/configuring other packages the header-includes option can be used to place LaTex commands directly into the header[9]:

```
title: Example with header-includes
header-includes:
- |
  ```{=latex}
  \usepackage{tocloft,tcolorbox}
```

[5] E.g pandoc example.md -o out.pdf. Output format is autodetected when specifying output filename. For some format (native) and for using non standard extensions, `-t` can be used - e.g. `pandoc example.md -t pdf out.pdf.old`

[6] Using the `--metadata-file` FILENAME option: https://pandoc.org/MANUAL.html#option--metadata-file

[7] Some are output format specific. All are documented here: https://pandoc.org/MANUAL.html#variables

[8] Metadata only versions of these variables are also available. These are suffixed with -meta, e.g. title-meta.

[9] The Pandoc manual warns that commands need to be placed in a markdown code block to prevent them being interpreted as markdown. In practice I've usually got away without needing to do this, presumably because of the behaviour/implementation of markdown parsing. Commands can also be included from a separate file which doesn't have this problem: https://pandoc.org/MANUAL.html#option–include-in-header. Using a separate file also works with formats like ODF.
```

```
  \setlength{\cftsubsecnumwidth}{2.8em}
  \setlength{\cftsubsubsecnumwidth}{3.6em}
  ```
```

### XeLaTex specific features

When XeLaTex[10] is used (by specifying the `--pdf-engine=xelatex` option) system fonts can be used and not just Tex fonts. This allows for easy use of characters which aren't compatible with `[T1]{fontenc}`, such as Unicode symbols, it also makes mixing characters from different scripts straightforward. This example uses the system Sans font for text instead of the default LaTex font:

```
title: Example with XeLaTex fonts and new font
mainfont: Liberation Sans
monofont: Liberation Mono
```

Step 2: Manipulating the output using Lua scripting

Pandoc has a very useful feature where it can be given a Lua script which can be used to change the document content (in Pandoc native format) after it has been read but before it is written to the output format. Pandoc describes these scripts as 'filters'.

There are many potential uses for this, some are given as examples in the Pandoc manual. The Lua script uses a Pandoc library to manipulate the document content. The Pandoc executable contains a built in Lua interpreter for this purpose, so there is no need to have Lua installed on your system.

The various types of content (e.g. headers, paragraphs) in the source document are stored as Pandoc 'elements'. Elements are stored based on a nested hierarchy, where the document itself is an array of headers, paragraphs, etc and paragraphs are arrays of strings[11]. These elements can be accessed and changed by using one or more 'filter functions'. Filter functions target element types[12]. These functions are defined in the script and called (in a specific order) by Pandoc when the filter script is executed. The functions are called on each element matching of that type.

Pandoc filters are very powerful and could even be used to create a document from scratch without any input at all; e.g. by fetching content from the web from within the script. It's not practical to go into more detail here, the Pandoc Lua filters manual is the ultimate reference.

Below is an example of a filter function which runs on all 'Header' elements and reduces the header level by one. A level 2 header in the input document would be written as a level 1 header in the output. This is a trivial example but it does demonstrate how easy to read these functions can be.

[10] These options also seem to be compatible with LuaLaTex, but I've not tried it.

[11] Interestingly spaces are not considered as parts of strings but as separate elements. This can make reading text in the native representation difficult.

[12] Note that elements can be more than one type. This can be imagined as inheritance. A filter for Block elements will apply to paragraphs as well as headers and tables. See https://pandoc.org/lua-filters.html#lua-type-reference
```

```lua
function Header(elem)
    elem.level = elem.level - 1
    return elem
end
```

Step 3: Putting it together (a use case)

The metadata file and Lua script filter are very useful when working with a source document which has sub optimal formatting which cannot be changed because that formatting is required for another reason.

Recently I made some improvements to a script for converting the OpenIndianadocumentation (which is written in MkDocs style markdown) into PDF format. At first it seemed the resulting PDFs wouldn't be perfect as this would require making changes to the markdown source. But making these changes would break the documentation website as MkDocs uses markdown differently to how Pandoc uses it. However, by pulling in some extra LaTex packages and writing some Lua filter functions, it was possible to create a well formatted PDF with a similar style.

The rest of this article gives some examples of how extra LaTex packages/configuration and Lua filter functions are used for the OpenIndiana docs PDF conversion.

Give code blocks line wrapping, change TOC spacing, add symbol font

```yaml
header-includes:
 - \usepackage{fvextra,tocloft,tcolorbox}
 - \DefineVerbatimEnviron-
 ↪  ment{Highlighting}{Verbatim}{breaklines,breakanywhere,commandchars=\\\{
 - \DefineVerbatimEnviron-
 ↪  ment{verbatim}{Verbatim}{breaklines,breakanywhere}
 - \newfontfamily\symbolfont[]{Noto Sans Symbols2}
 - \setlength{\cftsubsecnumwidth}{2.8em}
 - \setlength{\cftsubsubsecnumwidth}{3.6em}
```

This adds some extra configuration (in the metadata file/header) which brings in additional LaTex packages. The 'fvextra' package adds a more feature rich verbatim environment, this is then configured to replace the default environments ('Highlighting' is a Pandoc specific environment) with ones which have a line breaking/wrapping capability – by default long lines overflow the page.

The 'newfontfamily' command is specific to XeLaTex and is used to create a font family for a specific font. In this case 'Noto Sans Symbols2' is used because it's one of only a few fonts to support certain Unicode characters (as used in a later section below).

The 'tocloft' package allows for additional customization of the table of contents. In this case the default spacing wasn't enough for some of the section numbers and they were overlapping with the section name text. The 'tcolorbox' package is covered in a later section below.

Convert level 1 header into a title page, adjust level of other headers

```
function Header(elem)
  if (elem.level == 1) then
    documentTitle = pandoc.utils.stringify(elem.content)
    table.insert(elem.content,1,pandoc.RawInline('latex',
     ↪  '\\title{'))
    table.insert(elem.content,pandoc.RawInline('latex',
     ↪  '}\\maketitle\\tableofcontents\\newpage'))
    return pandoc.Plain(elem.content)
  else
    elem.level = elem.level - 1
    return elem
  end
end
function Meta(m)
  m['title-meta'] = documentTitle
  return m
end
```

This Lua snippet defines a filter function (which will be run on all header elements) which checks if a header is level 1, if it is it stores it's contents as a regular string and replaces the header (but not the text content) with LaTex for creating a title page[13]. The string is then used to set the `title-meta` meta value. For PDFs this becomes the 'Title' metadata field in the PDF file. If the header isn't 1, then the level is reduced by one, as seen the the simple example earlier.

Wrap HTML 'div' with LaTex 'tcolorbox'

In MkDocs markdown the HTML 'div' block is used to create information boxes to highlight specific content. These are always preceded by an 'i' tag which MkDocs uses to insert a symbol – either an informational or warning symbol. MkDocs styles the 'div' with CSS to appear inside a colored box. An example of the HTML is below.

```
<i class="fa fa-info-circle fa-lg" aria-hidden="true"></i>
 ↪  **NOTE:**
<div class="well">
Content to highlight to readers as a note.
</div>
```

This separation of information is lost during the Pandoc conversion. Pandoc stores the div and it's contents as a 'Div' element, but in the PDF output there's nothing to differentiate it from other text. While the 'i' tag is stored by Pandoc, it is lost in the process of converting to LaTex[14].

The Lua below wraps the Div contents with LaTex to create a 'tcolorbox' and adds a Unicode

[13] Note this is perhaps more manual than it needs to be. Setting the 'title' meta value causes Pandoc to automatically add the LaTex for a title page and table of contents can be added with a configuration setting. However, this method is required if a page break is required afterwards.

[14] If HTML had been used as the output instead it would be included. Pandoc cannot of course convert plain HTML markup directly into LaTex markup as the context (stylesheets etc.) is missing.

symbol to represent the type of note.

```
function Pandoc(thedoc)
  local blocks = thedoc.blocks
  local i = 1
  local noteBegin = pandoc.RawBlock('latex',
   ↪  '\\begin{tcolorbox}{\\symbolfont ▫} \\textbf{NOTE:}')
  local cautionBegin = pandoc.RawBlock('latex',
   ↪  '\\begin{tcolorbox}{\\symbolfont ▫} \\textbf{CAUTION:}')
  local noteEnd = pandoc.RawBlock('latex', "\\end{tcolorbox}")
  while i <= #blocks do
    if (blocks[i].t == "Div" and blocks[i].classes[1] == "well" and
     ↪  blocks[i-1].t == "Plain" and blocks[i-1].content[1].t ==
     ↪  "RawInline") then
      if (blocks[i-1].content[1].text == "<i class=\"fa
       ↪  fa-info-circle fa-lg\" aria-hidden=\"true\">" and
       ↪  blocks[i-1].content[4].content[1].text == "NOTE:") then
        table.remove(blocks,i-1)
        i = i - 1
        table.insert(blocks[i].content,1,noteBegin)
        table.insert(blocks[i].content,noteEnd)
      elseif (blocks[i-1].content[1].text == "<i class=\"fa
       ↪  fa-exclamation-triangle fa-lg\" aria-hidden=\"true\">"
       ↪  and blocks[i-1].content[4].content[1].text == "CAUTION:")
       ↪  then
        table.remove(blocks,i-1)
        i = i - 1
        table.insert(blocks[i].content,1,cautionBegin)
        table.insert(blocks[i].content,noteEnd)
      end
    end
    i = i + 1
  end
  return thedoc
end
```

Unlike the other filter functions, this one operates on the whole Pandoc document in one go, instead of running once for each element of a specified type. This is done because elements need to be removed based on their index and thus sight of the whole overarching block element array for the document is required.

Conclusion

This is only a very quick look at some simple things which can be done with Pandoc to reformat files when converting them. As this article is about Pandoc and mentions PDFs, it would be a mistake not to also include a Pandoc PDF version which uses some of the techniques mentioned.

I've got some other ideas for Pandoc, if I do write anything else I'll link it here.