Pandoc filters

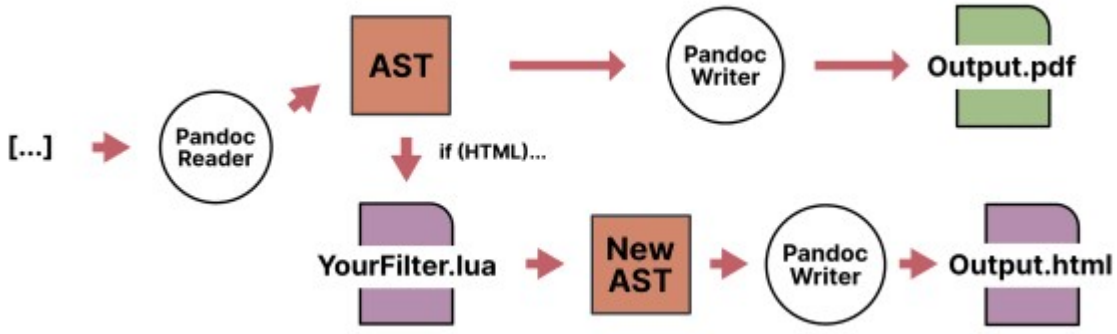
You can use JSON manipulation in any code language you prefer to do this, or write the filter in Lua. I chose to use Lua (despite not *knowing* Lua) because Pandoc includes a Lua interpreter embedded in it, so you don't need any external dependencies. Fortunately, some Lua basics weren't too hard to pick up, but please excuse my shoddy Lua coding in the examples given.

I think a diagram will help explain how filters work with Pandoc. Normally, building your book in Bookdown/Rmarkdown looks like this:

You write all your source docs, Bookdown/R processes all your code to produce plots/tables/etc and compiles all that into one mega-source doc, then ships that over to Pandoc with all your Bookdown options to produce your final output file (pdf/html/what-have-you).

With a filter, there is an extra step in the middle of the Pandoc circle. Under the hood, Pandoc reads in the input doc (`book.Rmd` in our case) and turns it into Pandoc's own interstitial format (an 'abstract syntax tree' or AST). Normally this then gets written straight back out into your desired output format.

But! With a filter, you get a chance to jump in and apply some logic to the **AST**, *before* it then gets sent over to the writer for the output format. So you end up with this:



This gives you a chance to affect elements of your document independent of any input/output foibles, and most usefully, change what output Pandoc's writer will generate based on whatever criteria you define. [1]

Writing the filter

Writing the filter is not *super* complicated, and the docs are very comprehensive, but it can take a second to wrap your head around how it works. Essentially, the filter should consist of a function named for one of the elements of Pandoc's AST. Which is just another way of saying, the elements of your source document.

So, if you want to affect all the **Strong** elements of your document, you would write a function named "Strong" and pass it the element as an argument – `Strong(elem)`. Pandoc will 'walk' the AST with your filter, and every time it finds a Strong element [2], apply your filter function to the contents of it (elem).

The only caveat is that a function must return the same type of element that went into it – so for Strong elements, the function `Strong()` must return an *Inline* element. Some elements are *Block* elements, like entire paragraphs or Divs, that must also return a Block.

This can be a bit finicky but basically, if the element can appear in the middle of a sentence, it's probably an Inline. If the element is a collection of smaller elements, it's probably a Block.

Example

Here's a practical example. I like to use Pandoc's fenced Div syntax to define little breakout boxes of information in my documents. Here's the syntax in my source Markdown file:

```
# Heading

Some text goes here.

::: notes
Here's the content of my breakout box.
:::
```

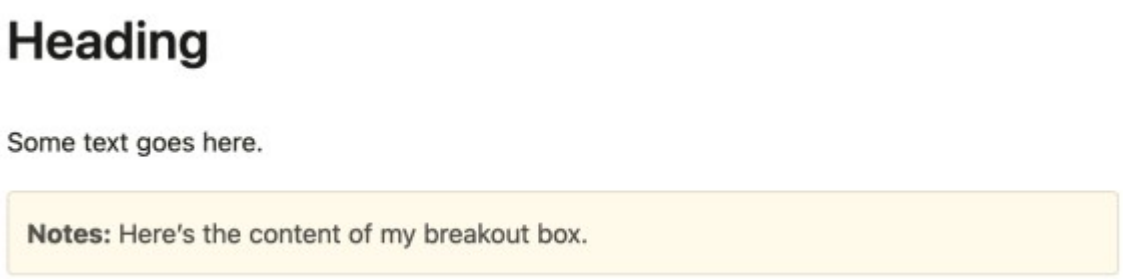
Pandoc already processes this into HTML, resulting in the following output:

Heading

Some text goes here.

Here's the content of my breakout box.

Notice how Pandoc has spotted our 'notes' label on the div and automatically applied it as a class to the div in the html. With a little CSS, this div then shows as:



Great. [3] But, if we want to output our document as anything else, this won't work. Under the hood, Pandoc is still applying the 'notes' label to our element in the AST, but that label doesn't *mean* anything to the other output formats. Instead, we can use a filter to add the required information to style these boxes in other output formats, without having to change our input syntax.

For Word/Docx output, Pandoc does let you apply a custom label to a fenced div, and will then map that onto a style with the same name in your Word reference doc. We could just change our syntax and write our source document like this:

```
# Heading

Some text goes here.

::: {.notes custom-style="notes"}
Here's the content of my breakout box.
:::
```

Here, we're still applying the '.notes' class in html (we have to include a period in front of it now though, as we've switched to use the bracketed divs syntax) but also declaring a custom-style of 'notes' that, if we have the corresponding style in our Word reference doc, Pandoc will apply to the content of the div.

This is okay, but it's nowhere near as convenient as the first example's syntax. Why not stick with that, and instead have the filter add in the extra syntax for us?

In the filter file (using the `.lua` extension), we can define a function named `Div` and pass in our element. In this case, we're passing in everything contained within the div – so a nested series of paragraphs, and within each of those, any inline elements. You can drill down and access those if you need, but for our purposes, we just want to add an extra attribute to the div:

```
function Div (elem)
  if FORMAT:match 'docx' then
    if elem.classes[1] == "notes" then
      elem.attributes['custom-style'] = 'Notes'
      return elem
    else
      return elem
    end
end
```

`FORMAT` is a global variable in a filter that just contains the output format currently being generated. Here, all we're saying is:

1. If the output is docx, look at each div within the document.
2. For each div, see if it has the class 'notes' with `elem.classes[1]`.[4]
3. If a div has the class 'notes', then *also* add a new attribute named 'custom-style' with the content 'Notes' and return the new div.
4. Otherwise, just return the original div.

Now, provided we have the style 'Notes' setup in our Word reference doc, it will automatically get assigned to our div in the output!

Importantly, if we switch our output back to HTML, it makes no difference – the class is just applied to the div as before, and the filter skips over the function as `FORMAT` doesn't match docx.

Finally, we can look at PDF output. Pandoc uses Latex as an intermediary to generate PDF files and unfortunately, there's no way (I know of) to apply custom styling to Latex output natively in Pandoc. Instead, we can use a filter to find our notes div as before, and *wrap* some custom Latex commands around the div contents

instead.

I used the Latex package tcolorbox for this. Other options are available, and the precise mechanisms of tcolorbox aren't the main point here, but if you want to use it exactly as I have you'll need to customise the Latex template to include tcolorbox as a package. Easiest way I found was to just put `\usepackage{tcolorbox}` in the Latex preamble as detailed here.

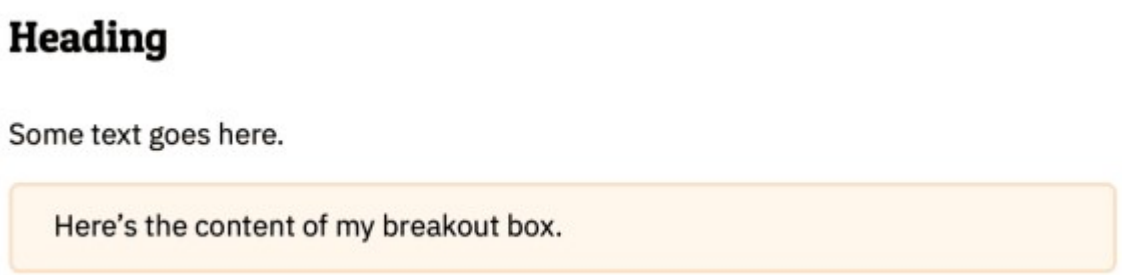
Here's the filter code:

```
function Div (elem)
  if FORMAT:match 'latex' then
    if elem.classes[1] == "notes" then
      return {
        pandoc.RawBlock('latex', '\\begin{tcolorbox}[colframe=Apricot!20!white,
colback=Apricot!8!white]'),
        elem,
        pandoc.RawBlock('latex', '\\end{tcolorbox}')
      }
    else
      return elem
    end
  end
end
```

This works much like the Word filter:

1. If the output is Latex, look at each div in the document.
2. See if each div has the class 'notes'.
3. If it does, create and return a *new* Block element.
    - This Block element will contain some raw Latex code, followed by the entire original div itself (elem), followed again by some more raw Latex.
4. Otherwise, return the original div.

`pandoc.RawBlock` is what allows us to write some raw code in the language our output format uses – in this case, Latex. So the function basically ends up wrapping a `\begin{tcolorbox}` and `\end{tcolorbox}` around our div and, when output to PDF, the div is now surrounded by the Latex code required to create a coloured box!



And again, if we switch back to HTML, Pandoc just ignores this function and our output looks the same as before.

We can combine these two functions into one:

```
function Div (elem)
  if FORMAT:match 'docx' then
    if elem.classes[1] == "notes" then
      elem.attributes['custom-style'] = 'Notes'
      return elem
    else
```

```
      return elem
    end
  elseif FORMAT:match 'latex' then
    if elem.classes[1] == "notes" then
      return {
        pandoc.RawBlock('latex', '\\begin{tcolorbox}[beforeafter skip=1cm, ignore
nobreak=true, breakable, colframe=Apricot!20!white, colback=Apricot!8!white,
boxsep=2mm, arc=0mm, boxrule=0.5mm]'),
        elem,
        pandoc.RawBlock('latex', '\\end{tcolorbox}')
      }
    else
      return elem
    end
  end
end
```

(GitHub Gist)

Now we can use fenced divs with `::: notes` with abandon in our original Markdown doc, and not worry about how it's going to come out in our output format – no matter which we choose, it'll be formatted as a coloured box.

Using with Bookdown/Rmarkdown

To use the filter with Bookdown/Rmarkdown, save it as a `.lua` file and place it somewhere convenient within your Bookdown project. I use `filters/shortcodes.lua`.

The most straightforward way I found to use the filter was to just tell Pandoc about it directly when we issue the command to build our document. In your `_output.yml`, just pass the filter as an extra argument to Pandoc using `pandoc_args` and `--lua-filter`. You'll need to include it for each output you want, for example:

```
bookdown::pdf_document2:
        toc: false
        pandoc_args: ["--lua-filter", "filters/shortcodes.lua"]
bookdown::word_document2:
        reference_docx: "assets/ref-doc.docx"
        pandoc_args: ["--lua-filter", "filters/shortcodes.lua"]
```

Obviously you'll also need to modify any of the surrounding files affected – maybe some custom CSS, or the Word reference doc, or the Latex template