Automated JATS XML to PDF conversion

Luke Skibinski

2018-11-09

JATS is a specification for describing articles. The specification is clear on how articles must be described however it's possible for the same article to be described in different ways by different authors and still fulfill the requirements of the specification. Authors themselves may also vary how they describe documents over time. This introduces the problem of how to treat JATS documents uniformly across authors and time so that all instances of JATS XML may benefit from common tooling, such as precise, consistent and predictable rendering to PDF.

eLife (and perhaps others) will hand-craft the final PDF version of an article because:

- automated conversions do not *quite* approach the precision of manual print design
- most articles are not read on the publisher's site in HTML form but in PDF or printed form elsewhere. Tthe quality of the PDF contributes to a good first impression.
- authors often feel the version of their submitted article to be the best formatted version and any clumsiness introduced with an automated approach to be a personal affront.
- it's cheap to outsource
- expertise in the technologies and languages that automate this process are relatively rare

This report is an evaulation of *pre-existing*, *open-source* technologies with a view to adoption during the article production process at eLife to ultimately:

- remove vendor dependency
- remove ability for authors to request changes to PDF
- share our work with other publishers

Results

The following tools were tested:

- CaSSius
- jats2latex
- jats-xslt-stylesheets
- pandoc
- peerj-jats-conversion

None of the above is a single technology and all fall in to one of these categories:

- `XML->HTML->PDF` (CaSSius, peerj-jats-conversion)
- `XML->LaTeX->PDF` (pandoc, jats2latex)
- `XML->XSL-FO->PDF` (jats-xslt-stylesheets)

The transformation from `JATS XML` to `HTML` and `LaTeX` requires an intermediate scraper that parses the `XML` and produces `HTML` or `LaTeX`.

The transformation from `JATS XML` to `XSL-FO` objects requires `XSL`, a transformation language described in `XML`.

`jats2latex` failed to produce anything for any document it was tested on. I believe it was developed for a particular vendor's flavour of `JATS XML` and wasn't robust enough to handle variations. I've excluded it from hereonout.

The results have no images in them. This is because special eLife pre-processing of the XML is required to expand image tags to actual files.

Evaluation

1. ease. how convenient is it to style the PDF?
2. precision. how closely can we affect the PDF?
3. speed. how fast was the transformation?
4. stability. how often is the technology stack changing?

All values are measured 1 to 5, with 5 being very good, 1 very bad.

	ease	precision	speed	stability
jats-xslt-stylesheets	1	5	3	5
cassius	3	2	1	2
peerj-jats-conversion	4	3	2	3
Pandoc	4	5	4	4

All values are my personal opinion. Timings for the conversions can be found in the timing.csv file and the pdf results in ./pdf

jats-xslt-stylesheets

These are official stylesheets in XSLT 1.0 and 2.0 formats and optional XPROC support. They are strict about refusing changes to the stylesheets that are presentational so there are no fancy examples here.

In this case I was attempting to generate PDF files by using the official stylesheets to generate XSL-FO files and then use Apache FOP to convert them into PDF files.

All attempts failed to generate a PDF except a relatively simple one. Debugging ended after Apache FOP stacktraces couldn't be resolved.

An example of a `.fo` file that is directly turned into a PDF can be found at ./jats-xslt-stylesheets/sample1.fo. Writing XSL transformations to generate XSL-FO files would give you very precise control over the final PDF.

CaSSius (HTML->PDF)

CaSSius is an attempt to use new web technologies such as Paged Media via Polyfills to generate print-quality PDF files.

CaSSius works by passing JATS XML through an XSL stylesheet to generate HTML. The HTML contains CaSSius stylesheets and javascript that can be modified and inspected easily on smaller articles. The HTML is then passed to `wkhtmltopdf` ("Webkit HTML to PDF"), a headless browser, that executes the javascript and generates the PDF.

Unfortunately CaSSius fails to produce anything more than the single leading page across all documents. That said, that single page is attractively styled. The HTML opened in Firefox resembles the PDF that was printed, so in this respect `wkhtmltopdf` is quite accurate. Debugging the running javascript in Firefox was difficult as the debugger often crashed and everything was so sluggish it was frustrating to inspect. Changes made in the inspector were not reflected in the browser.

More recent articles at Orbit are an example of a journal using CaSSius.

peerj-jats-conversion (HTML->PDF)

PeerJ have a repository of tools used for validating and converting their own JATS XML to HTML. The conversion to PDF is done using `wkhtmltopdf`, just like CaSSius.

The results have no special styling but it demonstrates that basic HTML to PDF *can* work well, that PDF bookmarks are present, links work, etc and that `wkhtmltopdf` isn't choking on larger articles.

Pandoc (LaTeX->PDF)

Pandoc is well established software that converts text documents to any other format. It does this by parsing a text document into it's own internal representation of a document using 'readers' and then writing that document out using a 'writer'. Pandoc can also generate PDF documents by using LaTeX as an intermediate format. Pandoc has recently added a 'jats' reader and writer so that JATS XML can now be read in and then a PDF written out by first generating LaTeX.

The outputs are only as good as the internal representation of the document, which *may be incomplete or lossy*. Pandoc is a Haskell language program and the 'jats' reader is also written in Haskell, rather than XSL or Java.

The template that writers use can be seen with `pandoc -D <writer>`. These templates can be customised and used instead during the generation process. A customised LaTeX template can be found at `./pandoc/latex-template.tex` and an example of a populated template is `./pandoc/pandoc--31543-intermediate.tex`. It uses a straightforward templating language very similar to `mustache` or `jinja`.

After the document has been read but before it has been written, the document can be transformed in a fashion similar to XSL using languages like Lua or Python or Haskell. These transformations are called 'filters' and an example can be found at `./pandoc/abstract.py`. The process becomes `Reader->Filters->Writer`.

Conclusions

There are two very large problems that can't be solved together: multitude of idiomatic JATS XML usage and a complete and precise rendering to PDF. I erred on simply producing an article PDF than coercing eLife JATS XML into a more compatible shape but I was necessarily constrained by the generalised JATS handling in the software. A single strict specification that all idiomatic JATS XML can be transformed to would allow the subsequent conversion to PDF to be available to everybody. This is a whole other epic undertaking.

My next biggest concern is completeness. We absolutely should not have existential worries that bits of the final PDF are 'missing' because they were not parsed. XSL neatly avoids this problem most of the time, but Haskell (Pandoc) and Java (jats2latex) and Python (bot-lax-adaptor) and other 'code' approaches do not. They target specific sections, extract and transform. Unhandled elements will silently be ignored. For example, Pandoc's Haskell 'jats' reader doesn't extract the 'abstract' element. There is support in the output LaTeX template for displaying an abstract but to get at it we would need to extend or fork the Haskell code. Or wait for somebody else to fix it.

LaTeX felt wonderfully precise, stable and sophisticated. Pandoc as well, but perhaps too constrained in it's internal model for us.

The state of a HTML+CSS+JS, Paged Media and headless PDF printing stack felt like a hodge-podge of good intentions, much enthusiasm, sluggish W3C proposals, literal 'polyfill' javascript and hamfisting a traditionally non-print medium into a print one. Development and debugging on this stack will be frustrating. This stack will feel like the easiest route to most of us with web backgrounds and there *is* a lot of enthusiasm out there for a 'web native' approach, but overall it's terribly imprecise, slow and immature. The print-css.rocks website has a number of 'lessons' that demonstrate various techniques of HTML->PDF, unfortunately most of them don't appear to very flattering when rendered with a headless browser. You can find my rendered outputs in the `./paged-media-output/` and the `./build-print-css-rocks-examples.sh` script to render your own.

The XSLT stack feels like the 'proper' approach, but the XSL idiom of transforming XML into other XML including XSL-FO is often alien to developers. It requires more than just learning another programming language, it requires learning a different programming paradigm altogether whose tools are unpopular, scorned and worst of all: old. The precision and speed of this stack would be second to none however, but so would the learning curve and expertise required to maintain it all.

Suggestions

If eLife is prioritising a community effort to improve JATS->PDF tooling for all, I suggest pouring time and effort into Pandoc and JATS4R.

If the priority is more practical, I would suggest none of the above tools and that eLife use the article-json as generated by the bot-lax-adaptor with a LaTeX template.

The benefits are:

- we are certain the eLife article-json is *complete*
- LaTeX will give us very precise control over the PDF presentation
- LaTeX can be templated like mustache or jinja, which is very familiar to the average developer
- LaTeX to PDF conversion is quick making the feedback loop short, which is important to development