The Canvas learning management system and LaTeXML

Will Robertson

1 Introduction

In 2017 The University of Adelaide adopted *Canvas* (Instructure, Utah, USA) for its learning management system (LMS). Unlike our previous LMS, Canvas provides a programming interface to get data from and send data to its servers. In this paper I will discuss the system I have started developing to prepare coursework material using LaTeX, processed via LaTeXML, and uploaded into Canvas in both HTML and PDF formats.

1.1 What is a learning management system?

An LMS is an online system that organises students into classes and allows them to access course notes, assignments, lecture recordings, discussion boards, and so on. They usually also include collaborative features to help teamwork activities. The LMS is also how the lecturer or coordinator for the course communicates with the students, posts grades, and otherwise does their job of running the course.

Each course tends to have a fairly typical structure, with links to the syllabus, assignment submissions, grades, etc. Actual course content in Canvas is organised into modules with individual pages written in HTML via a rich text editor. The amount of technical content in each course will depend on the coordinator; most academics I know still rely largely on printed lecture notes.

1.2 Requirements for the project

As the coordinator for the mechanical engineering honours project course, I maintain a large collection of course content that is provided as reference material. After many years as a LaTeX user, I am most comfortable with the idea of a source document that is portable, version controllable, and easy to edit. For me, using any web interface to fix typos or make small changes requires a non-negligible mental effort due to having to log in, navigate to an appropriate page, and click through to an editing mode. This project was motivated by my desire for a more efficient and effective means to keep course material up-to-date.

In the past, our honours project course documentation was a single monolithic Word document, which was difficult to maintain and awkward to extract information from. With a slow move to online documentation and incremental changes over time, this comprehensive document fell out of sync with the live content and had to be dropped. However, the single reference document had specific utility in being able to be easily distributed, and being searchable; its loss was not disastrous but not ideal.

The new LMS provided the opportunity to rethink the means by which the information needed in this course was documented and communicated to students and supervisors. This project was motivated by two main requirements: (1) improving the care and maintenance of the (somewhat extensive, and growing, amount of) content, and (2) to allow the generation of a comprehensive PDF reference document for the entire course.

When I started this work, I didn't know exactly what I wanted except that I knew I needed at least a good combination of the following:

One source, multiple outputs. My need for a better system arose from wanting to be able to keep online content up-to-date, while also having this content collected in a single offline location. And it soon became clear in my thinking that it would also need to be compilable into a single PDF document for distribution via alternate means.

Macros. I knew there would be a fair amount of information that I would want to re-use and keep consistent through the document (e.g., the names of academic and professional staff members; due dates for assessment). This precluded writing in HTML or a 'plain text' format like Markdown.

Reliable and easy to set up. As much as I might like to program my own document preparation system, using this was to be part of my day job and I don't want to have to dive into the weeds if code rot occurs and the system breaks down.

LaTeX syntax. I admit, I stick with what I know. Although I could have jumped into any number of competing technologies here, I knew I would be most comfortable if I was writing in the system with which I was most accustomed. More objectively, LaTeX is a mature format with a variety of possibilities among third party support tools for creating HTML.

2 The authoring interface

After some quick trials to establish that I could programmatically send HTML content to Canvas, it was time to decide how to generate the HTML in the first place.

2.1 Which HTML converter to choose?

As discussed above, I didn't want to do anything home-grown (and hence fragile), nor inflexible such as Markdown or raw HTML. I was aware of a number of 'competing' technologies to write in LaTeX or LaTeX-like syntaxes which could be converted to XML and hence to HTML. The most actively developed of these tools, and their implementation language, appear to be:

- lwarp [1] — Lua
- LaTeXML [2, 3] — Perl
- HeVeA [8] — OCaml
- Tralics [4] — C++
- TeX4ht [5] — C and TeX
- GELLMU [6, 7] — Emacs Lisp

(Not an exhaustive list; apologies for any oversights.) To be honest I didn't thoroughly evaluate each on their pragmatic merits; I was passingly familiar with LaTeXML's philosophy, had heard it was robust, and wanted to see if it fit the bill. It did, largely speaking. Of the tools listed above, only TeX4ht and lwarp are included in TeX Live. While LaTeXML required 'manual' installation, I had no troubles doing so.

2.2 LaTeXML overview

The LaTeXML program would be better explained by someone who knows a lot more than I do about Perl, XML, and friends. My user-level understanding is that LaTeXML reimplements an extensive subset of TeX in Perl, so that input documents are literally processed as LaTeX syntax, but not by the LaTeX program. The LaTeXML parser then intercepts package loading and inserts its own understanding of the various syntaxes introduced by different packages. If needed, it is possible to write custom support for packages that it doesn't cover out of the box.

LaTeXML does an excellent job emulating TeX, and it covers an impressive array of both TeX and LaTeX programming constructs.[1] Therefore, including in my preamble a construct like

```
\newcommand\honourscoord{Will Robertson}
```

simply worked out of the box with LaTeXML; indeed, simple LaTeX 2_ε programming using counters and so on worked without a hitch.

To run LaTeXML on my document involves the somewhat complex command:

```
latexml tex/$FILENAME.tex | latexmlpost - \
  --xsltparameter=SIMPLIFY_HTML:true \
  --sourcedirectory=tex \
  --format=html5 \
  --destination=html/$FILENAME.html \
  --splitat=chapter \
  --splitnaming=label
```

This setup ensures that for each 'chapter' of my LaTeX document a separate self-contained HTML file is created, which is the starting point for getting my content into Canvas.

Using a degree of consistency in the naming and structure of my document, each of my source LaTeX files with an `\input` for each chapter is therefore converted into a similarly-named HTML file. The structure of the source document is as follows:

```
\documentclass{report}
  ...
\begin{document}
  \title{...}\author{...}\date{...}
  \maketitle\tableofcontents
  \input{../texdata/course-data.tex}
\part{Introduction}\label{part-intro}
  \input{../pages/introduction}
  \input{../pages/course-schedule}
  \input{../pages/week-planner}
  ...
```

The file `course-data.tex` is the source for various data (such as names of staff members, weightings of assessment, due dates, etc.) in basic LaTeX data structures.

Each `.tex` file contains one chapter, with a convention that the `\label` of each chapter matches its file name:

```
% file: pages/introduction.tex
\chapter{Introduction}
\label{introduction}
...
```

This is because LaTeXML does not convert file by file; rather, with the `--splitat=chapter` option, the output HTML is split by chapter.

3 The Canvas programming interface

Canvas has a so-called 'REST API'[2] that was easy enough for me to get started with, with a little trial and error. Initially, I used `curl` commands wrapped into Bash scripts like this:

```
curl -X GET -H "$CANVASAUTH" $CANVASCOURSE/$1
```

where `$CANVASAUTH` is set up in my `.bash_profile` as a secret 'token' to avoid needing a manually-input password, and `$CANVASCOURSE` is defined essentially as the institution-specific URL to the Canvas course. Finally, `$1` is the parameter to send through to the Canvas API, such as `assignments` or `rubrics` or `users`, etc. The parameters passed in the `curl` argument `$1` can include additional options, such as

```
assignments?search_term=charter
```

would return just the one assignment for my honours project students called their 'Project Charter'.

The Canvas API can also be used to send or upload data to a course using `PUT` and `POST`. This

[1] Even David Carlisle's `xii.tex` can be successfully run through LaTeXML.

[2] `https://canvas.instructure.com/doc/api/`

can range from relatively small and simple pieces of information, to entire 'content pages' in HTML, to linked files that students can access and download (see Section A).

The results from any communication are sent back from the Canvas API in JSON format, for which there are a number of useful standard tools. For Bash scripts I use `jq`, a nice tool which is designed in the philosophy of Unix tools such as `awk` and `sed`.

After getting comfortable with this Bash script approach of sending and receiving data using `curl` and friends, I more recently started programming small Lua interfaces for more advanced operations. The main impetus for this switch was that the Canvas API will not send arbitrarily large amounts of information in one request. For example, if I request a list of students, it will send me just ten and expect me to ask again with `page=2` as an option. Then repeat until no data is returned. This level of iteration was beyond what I wanted to invest in a Bash script.

To write the Lua scripts I needed a number of third-party utilities. For Lua there are in fact very many tools to convert JSON data into Lua's 'table' data structure. I have been using the library json-lua (installed via luarocks) and it has been fine. After installing an SSL library (namely luasec, since unencrypted HTTP wouldn't cut it), I was able to convert the `curl` command above into an equivalent statement in Lua:

```
local http  = require("ssl.https")
local ltn12 = require("ltn12")
local body, code, hdrs, status = http.request{
  method = "GET",
  url = canvas_url .. req .. "?" .. opt,
  headers = {
   ["authorization"] = "Bearer "..canvas_token,
   ["content-type"]  = "application/json"
  },
  sink = ltn12.sink.table(canvas_result),
}
```

where `canvas_result` is the name of the table where the data will be stored. It is then 'decoded' from JSON using json-lua.

4 The generated HTML page

For pages delivered via Canvas, I am not writing self-contained HTML files; rather, Canvas constructs the page within its main interface, and within the page includes what I call 'snippets' of HTML to display the actual course content.

The HTML that is generated by LaTeXML is not intended to be used for 'snippets' to be transferred into a separate system, but the good thing about HTML is that this isn't really a problem, since any additional markup in the HTML is silently ignored.

LaTeXML runs a two-stage process, where the latexml program itself generates generic XML from the LaTeX input, and then latexmlpost converts this generic XML into one of several output formats. In theory I could write my own XSL stylesheet to format the information in the generic XML document in a customised way. Instead, I simply postprocess the HTML output from latexmlpost; because LaTeXML creates highly structured and machine friendly HTML, the output from latexmlpost makes this an easy process with some ad hoc tools (with plans for more robust Lua processing in the future).

According to the options passed to latexmlpost, each chapter is converted into its own HTML file; this means I have a one-for-one correspondence between files `\included` as chapters in the main document. Each of these chapters has a standard structure, with 'top matter' that is not needed:[3]

```
<!DOCTYPE html><html>
<head>
 <title>
  ⟨Title of chapter⟩
 </title>
 <meta http-equiv="Content-Type"
       content="text/html; charset=UTF-8">
 <link rel="stylesheet" href="LaTeXML.css"
       type="text/css">
  [...]
</head>
<body>
```

The chapters end with 'bottom matter' which is also not needed in my application:

```
<footer class="ltx_page_footer">
  [...]
</footer>
</div>
</body>
</html>
```

Finally, the 'content' of each HTML file has a structure along the following lines:

```
<div class="ltx_page_main">
<header class="ltx_page_header">
  [...]
</header>
<div class="ltx_page_content">
 <section class="ltx_chapter ltx_authors_1line">
  <h1 class="ltx_title ltx_title_chapter">
   <span class="ltx_tag ltx_tag_chapter">
    ⟨Chapter number⟩
   </span>
   ⟨Title of chapter again⟩
```

[3] All of the HTML examples have been reformatted for ease of reading.

```
  </h1>
  <div class="ltx_date ltx_role_creation"></div>
  <section id="S1" class="ltx_section">
   ⟨Contents of chapter⟩
  </section>
 </section>
</div>
```

The un-greyed text is the part of the HTML that is retained for upload into Canvas. A single line of `awk` is used to extract everything contained between the outer `<section>` tags:

```
  awk '/\<section.*\>/,/\<\/section\>/' \
    html/$BASE >> snip/$BASE
```

The LMS doesn't do anything in particular with the `<section>` tags, but it doesn't mind them, either. Similarly, all of the CSS structure (the `class` and `id` tags) is not used by Canvas, but also, thankfully, doesn't cause any problems.

It is worth noting that while plain LaTeX and HTML share some superficial similarities in the types of document structures they can produce, in some cases LaTeXML really has to work hard to replicate certain LaTeX structures in HTML+CSS. The example that I ran into was related to lists. In order to cope with LaTeX syntax such as:

```
\begin{enumerate}
  \item      aaa
  \item[1b.] bbb
  \item      ccc
\end{enumerate}
```

the HTML generated by latexmlpost looks like:

```
<ol id="I1" class="ltx_enumerate">
 <li id="I1.i1" class="ltx_item"
                style="list-style-type:none;">
  <span class="ltx_tag ltx_tag_enumerate">
   1.
  </span>
  <div id="I1.i1.p1" class="ltx_para">
   <p class="ltx_p">aaa</p>
  </div>
 </li>
 <li ...>...</li>
 <li ...>...</li>
</ol>
```

(The second and third items have identical structure and are elided.) This is carefully structured HTML source intended to be styled with specific CSS provided by LaTeXML. However, without latexml's custom CSS files to control its layout, this HTML produces output something like this:

> 1.
>   aaa
> 1b.
>   bbb
> 2.
>   ccc

While understandable to a reader, this is not ideal from a formatting perspective. LaTeXML's default to match as much of LaTeX's functionality as possible is notable in the overall design of LaTeXML, but for my needs it still needed a workaround. In an early stage of this project, I used more cumbersome regex code to adapt the LaTeXML output to something that didn't need CSS, but the developer of LaTeXML kindly added the `--xsltparameter=SIMPLIFY_HTML:true` option to account for my use case here.

5 Future work

- I do not yet have an automated approach to linking images and files. This has been okay for the time being, since for a small number of items doing a manual upload is no real imposition. In the long run, it would be nice to have this automated; compiling the main document could create a list of files, and a Lua script could check which files were already present in Canvas and only upload those missing. (And possibly even delete any existing files no longer used in the document.)
- What about mathematics? Luckily, for this course I don't need to include mathematical content. LaTeXML, naturally, can produce appropriate mathematical output in a variety of modes (MathML, etc.). Currently Canvas is in the middle of having its support for mathematical content improved and I'm holding off considering this further until their platform has stabilised.
- In time I will transition away from Bash scripts entirely to make the system more portable and robust. Since a TeX platform is required to typeset the PDF documentation, Lua is the natural choice as a scripting language. I explicitly do not wish to develop a full-featured Canvas interface in Lua, but I hope this system will become general enough that other users could deploy it for their courses.
- Currently these documents use a largely 'one-way' communication in that the LaTeX source is compiled and delivered to Canvas. However, for certain types of information (assignment rubrics in particular), the best source of this information is within the LMS itself. Therefore, I will be

building data processors to typeset information from Canvas within the PDF documentation (with a simple link in the online version).

6 Benefits of scripting Canvas

This is unrelated to the LaTeX side of things, but coming to terms with Canvas's programming interface has opened the door for me to perform quite a number of additional tasks that were previously impossible with our older LMS.

For example, our honours project reports are assessed by their academic supervisors, and directing supervisors to each report and following up in a timely manner were both difficult tasks to automate. Using a Lua script I now dynamically create a list of project reports that have not yet been marked. This allows me to automatically construct personalised emails for each project supervisor to remind them when marks are due with direct links to their assessments to mark.

As with LaTeX itself, once you have an interface that can be programmed it opens the door to extending the ways in which one uses the system.

7 Conclusion

I have satisfied the following use cases in this work:

- A typo fix or quick addition can be done by editing the source in a text editor, synced via the cloud, and uploaded with a one-line command. No need to open a browser, log in, and click through.
- Information can be 'programmed' using macros for greater consistency. This is straightforward in LaTeX and basically unheard of in a web-based editing approach.
- HTML and PDF output are kept in sync at all times; the PDF provides an archivable document for the entire course.

Although I haven't (yet) produced the most elegant system, I have created a solution without too much elbow grease beyond standard LaTeX ecosystem tools that does quite a bit more than I think anyone would otherwise consider possible. The choice of LaTeXML worked well, although few design decisions rely on it; switching to another tool for the HTML conversion would be possible with a little additional work.

The ability to develop these solutions is truly a testament to the flexibility of LaTeX, and an example of why I think it will remain relevant indefinitely. Once you program your first document, you can never go back to manually keeping track of all the bits and pieces.

References

[1] B. Dunn. Producing HTML directly from LaTeX — the lwarp package. *TUGboat* 38(1), 2017. tug.org/TUGboat/tb38-1/tb118dunn-lwarp.pdf

[2] D. Ginev and B. R. Miller. LaTeXML 2012 — A Year of LaTeXML, 2014. nist.gov/publications/latexml-2012-year-latexml

[3] D. Ginev, B. R. Miller, and S. Oprea. E-books and Graphics with LaTeXML, 2014. arxiv.org/pdf/1404.6547v1

[4] J. Grimm. Tralics, a LaTeX to XML translator. *TUGboat* 24(3), 2003. tug.org/TUGboat/tb24-3/grimm.pdf

[5] E. M. Gurari. TeX4ht: HTML production. *TUGboat* 25(1), 2004. tug.org/TUGboat/tb25-1/gurari.pdf

[6] W. F. Hammond. GELLMU: A bridge for authors from LaTeX to XML. *TUGboat* 22(3), 2001. tug.org/TUGboat/tb22-3/tb72hammond.pdf

[7] W. F. Hammond. Dual presentation with math from one source using GELLMU. *TUGboat* 28(3), 2007. tug.org/TUGboat/tb28-3/tb90hammond.pdf

[8] HeVeA. hevea.inria.fr

A Uploading a file to Canvas via its API

Evidently in a fit of late night fervour I concocted the following monstrosity for uploading a file to Canvas using a Bash function and few helper commands:

```
curl -X POST  -H "$CANVASAUTH" \
                 "$CANVASCOURSE/files" \
     -F "name=$1" \
     -F "parent_folder_path=upload" > tmp.json ;
URL=‘cat tmp.json | jq ’.upload_url’‘ ;
KEYS=‘cat tmp.json | jq ’.upload_params’ |\
      jq -r -j "to_entries | \
        map(\"-F \(.key)=\(.value|tostring)\
            \")|.[]"‘ ;
echo curl -D response.tmp\
                 $URL $KEYS -F file=@$1 | bash ;
LOC=‘sed -n -e ’s/Location: \
                     \(.*\)/\1/p’ response.tmp‘;
LOC=${LOC%$’\r’}
curl -X POST -H "$CANVASAUTH" "$LOC" | jq ;
```

I think it's fair to say that a Lua implementation would be rather more maintainable.

◇ Will Robertson
School of Mechanical Engineering
The University of Adelaide, SA
Australia
will.robertson (at) adelaide dot edu dot au
https://gitlab.adelaide.edu.au/wspr/canvas-tools