CoffeeXeLaTeX

(CXLTX)

How to

Extend LaTeX

with

A Real Programming Language

Contents

Conventions: In this document, (most) command line / code stuff is printed in red, while (most) remote command output is printed in green.

1 CoffeeXeLaTeX (CXLTX)

1.1 What is it? And Why?

Everyone who has worked with LaTeX knows how hard it can often be to get seemingly simple things done in this Turing-complete markup language. Let's face it, (La)TeX has many problems; the complectedness of its inner workings and the extremely uneven syntax of its commands put a heavy burden on the average user.

The funny thing is that **while TeX is all about computational text processing, doing math and string processing are often really hard to get right** (not to mention that TeX has no notion of higher-order data types, such as, say, lists).

Often one wishes one could just do a simple calculation or build a typesetting object from available data *outside* of all that makes LaTeX so difficult to get right. Turns out you can already do that, and you don't have to recompile TeX.

Most of the time, running TeX means to author a source file and have the TeX executable convert that into a PDF. Of course, this implies reading and writing of files and executing binaries. Interestingly for us, both capabilities—file access and command execution—are made available to user-facing side of TeX: writing to a file happens via the `\write` command, while input from a file is done with `\input`; command execution repurposes `\write`, which may be called with the special stream number `18` (internally, TeX does almost everything with registers that are sometimes given symbolic names; it also enumerates 'channels' for file operations, and reserves #18 for writing to the command line and executing stuff). This is how the `\exec` command is (in essence) defined in `coffeexelatex.sty`:

```
\newcommand{\CXLTXtempOutRoute}{/tmp/coffeexelatex.tex}

\newcommand{\exec}[1]{%
  \immediate\write18{#1 > \CXLTXtempOutRoute}
  \input{\CXLTXtempOutRoute}
  }
```

With some TeXs, its possible to avoid the temporary file by using `\@@input|"dir"`, but XeTeX as provided by TeXLive 2013 does not allow that. The temporary file does have an advantage: in case TeX should halt execution because of an error, and that error is due to a script with faulty output, you can conveniently review the problematic source by opening the temporary file in your text editor.

Besides `\exec`, there is also `\execdebug` which captures the `stderr` output of a command and renders it in red to the document in case any output occurred there.

1.2 TeX and NodeJS

1.2.1 Method One: Spawning a `node` process

The whole idea of CXLTX is to have some external program receive data from inside a running TeX invocation, process that data, and pass the result back to TeX.

The command given to `\exec` could be anything; for our purposes, we will concentrate on running NodeJS programs. The simplest thing is to have NodeJS evaluate a JavaScript expression and print out the result; the `\evaljs` command has been defined to do just that:

```
\newcommand{\evaljs}[1]{\execdebug{node -p "#1"}}

% will insert `16 * 3 = 48` in TeX math mode (triggered by `$`):
$ 16 * 3 = \evaljs{16 * 3} $
```

This technique is easily adapted to work with [CoffeeScript]{http://coffeescript.org} (or any other language that compiles to JS):

```
\newcommand{\evalcs}[1]{\execdebug{coffee -e "console.log #1"}}

% will insert `[1,4,9]`
\evalcs{ ( n * n for n in [ 1, 2, 3 ] ) }
```

The Command Line Remote Command Interface (CL/RCI) Of course, evaluating one-liners will give you only so much power, so why not execute more sizeable programs? That's what `\nodeRun` is for:

```
\usepackage[abspath]{currfile}
\newcommand{\CXLTXmainRoute}{../../lib/main}
\newcommand{\nodeRun}[2]{
  \exec{node "\CXLTXmainRoute" "\currfileabsdir" "\jobname" "#1" "#2"}}

% will insert `Hello, world!`
\nodeRun{helo}{world}
```

`\NodeRun` will run NodeJS with the following arguments: **(1)** the route to our custom-made executable; **(2)** the route to the parent directory where the currently processed TeX source files are; **(3)** the current job name; **(4)** a command name (first argument to `\nodeRun`); and, lastly, **(5)** optional command arguments (second argument to `\nodeRun`).

Observe that you may want to define your own routes in case the default values do not match your needs; these can be easily done using `\renewcommand`:

```
\renewcommand{\CXLTXmainRoute}{../../lib/main}
\renewcommand{\CXLTXtempOutRoute}{/tmp/CXLTXtempout.tex}
\renewcommand{\CXLTXtempErrRoute}{/tmp/CXLTXtemperr.tex}
```

1.2.2 Method Two: Spawning `curl`

Spawning a subprocess to 'outsource' a computing task is certainly not the cheapest or fastest way to do stuff, as creating and taking down a process is relatively costly.

More specifically, spawning `node` is both comparatively expensive (in terms of memory) and slow. Also, the subprocess must run on the same machine as the main process, and unless that subprocess persisted some data (in a DB or in a file), the subprocess will run in a stateless fashion.

Then again, since we're using NodeJS anyway: What's more obvious than to make TeX communicate with a long-running HTTP server?

Now i've not heard of any way to make TeX issue an HTTP request on its own behalf. But we already know we can issue arbitrary command lines, so we certainly can spawn `curl localhost` to communicate with a server. We still have to spawn a subprocess that way—but maybe `curl` is both lighter and faster than `node`!? It certainly is.

Here are some timings i obtained for running our simple echoing example, once spawning `node`, and once spawning `curl`. The server is a very simple Express application; except for the command line argument and HTTP parameter handling, the same code is ultimately executed:

```
time node "cxltx/lib/main" "cxltx/doc/" "cxltx-manual" \
  "helo" "friends" \
  > "/tmp/CXLTXtempout.tex" 2> "/tmp/CXLTXtemperr.tex"

real  0m0.140s
real  0m0.082s
real  0m0.083s

time curl --silent --show-error \
  127.0.0.1:8910/foobar.tex/cxltx-manual/helo/friends \
  > "/tmp/CXLTXtempout.tex" 2> "/tmp/CXLTXtemperr.tex"

real  0m0.010s
real  0m0.011s
real  0m0.010s
```

In so far these rather naïve benchmarks can be trusted, they would indicate that fetching the same insignificant amount of data via `curl` from a local server is around ten times as performant as doing the same thing by spawning `node`. The reason for this is partly attributable to the considerable size of the NodeJS binary (respectively whatever actions are taken by NodeJS to ready itself).

Even doing `time node -p "1"` results in execution times of over 0.07s. Add to this the `curl` timings of around 0.01s, and you roughly get the 0.08s needed to spawn `node` and get results—in other words, `curl` itself seems to be quite fast.

The upshot of this is that using our first method we can call external code at a frequency around 10 per second, but using the `curl` method, we can get closer to almost 100 per second (depending on the machine etc). The difference might matter if you plan to put out a *lot* of external calls, and since the typical way of getting TeX source code right is running and re-running TeX a lot of times, doing it faster may help greatly.

The HTTP Remote Command Interface (H/RCI)

```
\curlRaw{127.0.0.1:8910/foobar.tex/\jobname/helo/friends}
```

```
\curl{helo}{friends}
```

1.2.3 Security Considerations

Be aware that executing arbitrary code by way of mediation of a command line like

```
xelatex --enable-write18 --shell-escape somefile.tex
```

is inherently unsafe: a TeX file downloaded from somewhere could erase your disk, access your email, or install a program on your computer. This is what the `--enable-write18` switch is for: it is by default fully or partially disabled so an arbitrary TeX source gets limited access to your computer.

If you're scared now, please hang on a second. I just want to tell you *you should be really, really scared*. Why? Because if you ever downloaded some TeX source to compile it on your machine **even without the `--enable-write18` switch** you've already **executed potentially harmful code**. Few people are aware of it, but many TeX installations are quite 'liberal' in respect to what TeX sources—TeX programs, really—are allowed to do *even in absence of command line switches*, and, as a result, even people who are hosting public TeX installations for a living are susceptible to malicious code.*

It is a misconception that TeX source is 'safe' because 'TeX is text-based format' (how stupid is that, anyway?); **the truth is that by doing `latex xy.tex` you're executing code which may do malicious things**. Period. That said, the papers linked below make it quite clear that `--enable-write18` just 'opens the barn door', as it were, but in fact, there are quite a few other and less well known avenues for TeX-based malware to do things on your computer.

And please don't think you're safe just because you're not executing anything but your own TeX source—that, in case you're using LaTeX, is highly improbable: any given real-world LaTeX document will start with a fair number of `\usepackage{}` statements, and each one of those refers to a source that is publicly accessible on the internet and has been so for maybe five or ten or more years. Someone might even have managed to place a mildly useful package on CTAN, one that has some obfuscated parts designed to take over world leadership on Friday, 13th—who knows?

The fact that TeX is a programming language that works by repeatedly re-writing itself does not exactly help in doing static code analysis; in fact, such code is called 'metamorphic code' and is a well-known technique employed by computer viruses.

I do not write this section of the present README to scare you away, just to inform whoever is concerned of a little known fact of life. The gist of this is: don't have `--enable-write18` turned on except you know what you're doing, but be aware that running TeX has always been unsafe anyway.

*) see e.g. `http://cseweb.ucsd.edu/~hovav/dist/texhack.pdf`

1.3 Sample Command Lines

To make it easier for TeX to resolve `\usepackage{cxltx}`, put a symlink to your CXLTX directory into a directory that is on LaTeX's search path. On OSX with LiveTeX, that can be achieved by doing

```
cd ~/Library/texmf/tex/latex
ln -s route/to/cxltx cxltx
```

Here is what i do to build `cxltx/cxltx-manual.pdf`:

(1) use Pandoc to convert `README.md` to `README.tex`:

```
pandoc -o cxltx/doc/README.tex cxltx/README.md
```

(2) copy the `aux` file from the previous TeX compilation step to preserve its data for CXLTX to see:

```
cp cxltx/doc/cxltx-manual.aux cxltx/doc/cxltx-manual.auxcopy
```

(3) compile `cxltx-manual.tex` to `cxltx-manual.pdf` (`--enable-write18`: allows to access external programs form within TeX; `--halt-on-error`: is a convenience so i don't have to type `x` on each TeX error; `--recorder`: needed by the `currfile` package to get absolute routes):

```
xelatex \
  --output-directory cxltx/doc \
  --halt-on-error \
  --enable-write18 \
  --recorder \
  cxltx/doc/cxltx-manual.tex
```

(4) move the pdf file to its target location:

```
mv cxltx/doc/cxltx-manual.pdf cxltx
```

1.4 Useful Links

http://www.ctan.org/tex-archive/macros/latex/contrib/perltex

http://ctan.space-pro.be/tex-archive/macros/latex/contrib/perltex/perltex.pdf

http://www.tug.org/TUGboat/tb28-3/tb90mertz.pdf

https://www.tug.org/TUGboat/tb25-2/tb81pakin.pdf

1.5 Related Work

¶ **PyTeX** (also dubbed QATeX) is a laudable effort that has, sadly, been stalling for around 11 years as of this writing (January 2014), so it is likely pretty much outdated. PyTeX's approach is apparently the opposite of what we do in CXLTX: they run TeX in daemon mode from Python,

where we have NodeJS start a server that listens to our independently running TeX.—Just for giggles, a quote from the above page: "XML is hard work to key by hand. *It lacks the mark-up minimization that SGML has*" (my emphasis). Well, eleven years is a long time.

¶ **PythonTeX** is an interesting approach to bringing LaTeX and Python together. Unfortunately, the authors are preconcerned with showing off Pygment's syntax hiliting capabilities (which are ... not really that great) and how to print out integrals using SymPy; sadly, they fail to provide sample code of interest to a wider audience. Their copious 128-page manual only dwells for one and a half page on the topic of 'how do i use this stuff', and that only to show off more SymPy capabilities. None of their sample code *needs* PythonTeX anyway, since none of it demonstrates how to interact with the document typesetting process; as such, all their formulas and plots may be produced offline, independently from LaTeX. Given that the installation instructions are too scary and longwinded for my taste, and that PythonTeX is not part of LiveTeX, i've given up on the matter.

(the below taken from http://get-software.net/macros/latex/contrib/pythontex):

¶ **SageTeX** allows code for the Sage mathematics software to be executed from within a LaTeX document.

¶ Martin R. Ehmsen's `python.sty` provides a very basic method of executing Python code from within a LaTeX document.

¶ **SympyTeX** allows more sophisticated Python execution, and is largely based on a subset of SageTeX.

¶ **LuaTeX** extends the pdfTeX engine to provide Lua as an embedded scripting language, and as a result yields tight, low-level Lua integration.

LuaTeX is one of the most interesting projects in this field as it represents an attempt to provide a close coupling of a real programming language with LaTeX. Unfortunately, that language is Lua, whose designers believe that Unicode strings should be stored as UTF-8 bytes (Go does the same, btw). Equally unfortunately, LuaTeX uses pdfTeX, which can't compare to XeLaTeX when it comes to using custom TTF/OTF fonts.

2 Examples

2.1 Spawning NodeJS

(1) The original technique to execute an arbitrary command:

```
\immediate\write18{node
  "\CXLTXcliRoute"
  "\currfileabsdir"
  "\jobname"
  ","
  "helo"
  "readers (one)"
  > /tmp/temp.dat}\input{/tmp/temp.dat}
```

(2) With ugly details largely hidden, the `\exec{}` command is still fully general:

```
\exec{node
  "\CXLTXcliRoute"
  "\currfileabsdir"
  "\jobname"
  ","
  "helo"
  "readers (two)"}
```

(3) `\nodeRunScript{}` will execute NodeJS code that adheres to the call convention established by CXLTX:

```
\nodeRunScript
  {\CXLTXcliRoute}
  {\currfileabsdir}
  {\jobname}
  {,}
  {helo}
  {readers (three)}
```

(4) Like the previous example, but with standard values assumed as shown above. This is the form that you will want to use most of the time (if you want to use the CL/RCI at all):

```
\nodeRun{helo}{readers (four)}
```

Outputs:

Hello, readers (one)!

Hello, readers (two)!

Hello, readers (three)!

Hello, readers (four)!

2.2 Evaluating Expressions

The commands `\evalcs{}` and `\evaljs{}` allow you to evaluate an arbitrary self-contained expression, written either in CoffeeScript or in JavaScript:

```
$23 + 65 * 123 = \evalcs{23 + 65 * 123}$
```

$23 + 65 * 123 = 8018$

2.3 Spawning cURL

```
\curlRaw{127.0.0.1:8910/foobar.tex/\jobname/,/helo/friends}
```

Hello, friends!

2.4 Character Escaping

The XeLaTeX command `show-special-chrs` demonstrates that it is easy to include TeX special characters in the return value. The simple rule is that whenever the output of a command is meant to be understood literally, it should be `@escape`d:

```
\nodeRun{show-special-chrs}{}
```

opening brace	{
closing brace	}
Dollar sign	$
ampersand	&
hash	#
caret	^
underscore	_
wave	~
percent sign	%

2.5 Unicode

The next few examples demonstrate that Unicode characters—even ones from outside the Unicode Basic Multilingual plain, which frequently cause difficulties—can be transported to and from the server without losses or Mojibake / squiggles:

```
\curl{helo}{äöüÄÖÜß}
```

Hello, äöüÄÖÜß!

Chinese characters from the Unicode BMP ('16 bit'):

```
\curl{helo}{ 黎永強 }
```

Hello, 黎永強!

Chinese characters from the Unicode SIP ('32 bit'—these needed a little trick to make XeLaTeX choose the right font; see `coffeexelatex.sty`):

```
\curl{helo}{ 乛虫丽 }
```

Hello, 乛虫丽!

2.6 The `aux` Object

To facilitate data exchange between the TeX process and the server, CXLTX provides facilities to read and write data from and to the `aux` file assoated with the current job:

- the TeX commands `\aux`, `\auxc`, `\auxcs`, and `\auxpod`, which write to the `aux` file;
- the method `CXLTX.main.read_aux = ( handler ) ->`, which reads (and parses) data written with one of the above commands.

2.6.1 The `\aux*` Commands

Because they're quite straightforward, let's have a look at the actual definitions of the `aux*` commands:

```
\makeatletter
\catcode`\%=11
\newcommand{\aux}[1]{\immediate\write\CXLTX.auxout{#1}}
\newcommand{\auxc}[1]{\immediate\write\CXLTX.auxout{% #1}}
\newcommand{\auxcs}[1]{\immediate\write\CXLTX.auxout{% coffee #1}}
\newcommand{\auxpod}[2]{\immediate\write\CXLTX.auxout{% coffee #1: \{ #2 \}}}
\catcode`\%=14
\makeatother
```

We see our old friend `\immediate\write` here, this time accessing channel `\CXLTX.auxout`. All commands will write a single line to the `aux` file.

- `\aux` is the most basic command and will write text as-is to the `aux` file;
- `\auxc` puts whatever is written behind a `%` (percent sign), so it appears as a comment when TeX re-reads the `aux` file;
- `\auxcs` writes text behind a `% coffee` marker, facilitating recognition on the server side;
- `\auxpod` takes a name and a CoffeeScript Plain Old Dictionary literal (*without* the braces) to the `aux` file; to the server, this will become available as `CXLTX.aux[ name ]`;

The `\auxgeo` / `@\curl{show-geometry}{}` command pair is a good example how to use `\auxpod`.

2.6.2 Geometry

Use geometry data from aux file to render a table of layout dimensions into the document; note the we could have used the `\auxgeo` command anywhere in the document and that this currently only works for documents with a single, constant layout.

Also note we're using a dash instead of an underscore here—in TEX, underscores are special, so we conveniently allow dashes to make things easier. The CXLTX command `show-geometry` does not take arguments, which is why the second pair of braces has been left empty:

```
\curl{show-geometry}{}
```

firstlinev	15.00 mm
footskip	10.54 mm
headheight	4.22 mm
headsep	8.79 mm
marginparsep	3.51 mm
marginparwidth	12.30 mm
paperheight	297.00 mm
paperwidth	210.00 mm
textheight	246.36 mm
textwidth	155.00 mm
topmargin	−23.40 mm
voffset	25.40 mm

After `show-geometry` has been performed, the `CXLTX.aux` object has been populated with data from the `aux` file; it then looks like this for the current document:

```
\curl{show-aux}{}
```

```
{ 'is-complete': false,
  texroute: '/Volumes/Storage/cnd/node_modules/cxltx/doc/',
  auxroute: '/Volumes/Storage/cnd/node_modules/cxltx/doc/cxltx-
manual.auxcopy',
  jobname: 'cxltx-manual',
  splitter: ',',
  'method-name': 'show_aux',
  parameters: [],
  labels:
   { 'coffeexelatex-cxltx':
      { name: 'coffeexelatex-cxltx',
        ref: 1,
        pageref: 3,
        title: 'CoffeeXeLaTeX (CXLTX)' },
     examples: { name: 'examples', ref: 2, pageref: 9, title: 'Examples' } },
  geometry:
   { paperwidth: 210.0001,
     paperheight: 297,
     textwidth: 155.0001,
     textheight: 246.3597,
     headheight: 4.2176,
     headsep: 8.7865,
     footskip: 10.5438,
     marginparsep: 3.5146,
```

```
    marginparwidth: 12.3011,
    voffset: 25.4,
    topmargin: -23.404,
    firstlinev: 15.0001 } }
```

2.6.3 Labels

As it stands, CXLTX will try and collect all pertinent data from the `CXLTX.aux` file when `@read_aux` is called; this currently includes

labels from the `*.aux` file associated with the current job; from inside your scripts [redacted]

```
\curl{clear-aux}{}
\curl{show-aux}{}
```

```
{ 'is-complete': false,
  texroute: '/Volumes/Storage/cnd/node_modules/cxltx/doc/',
  auxroute: '/Volumes/Storage/cnd/node_modules/cxltx/doc/cxltx-
manual.auxcopy',
  jobname: 'cxltx-manual',
  splitter: ',',
  'method-name': 'show_aux',
  parameters: [] }
```

2.7 Implementing and Calling Functions