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.

Click here to read the full documentation

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 (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.