Publishing the Dice Word List book

Wednesday, April 18th, 2018

I recently published a small book, the Dice Word List[1], and I wanted to write down some of the technical details and approaches I took in getting that little thing out the door. Parts of this post assume some familiarity with Pollen[2] and LaTeX[3].

Although the book's cover is not very complex or impressive, creating it was the biggest detour I took. I didn't just want a cover, I wanted better tools for making covers for future books as well.

I've made a few books for fun over the years, and creating the cover was always my least favorite part of the process. I had to fire up a graphics program, calculate the size of everything, and click around until I had what I wanted. If my page count changed, I'd have to recalculate everything and manually futz with it some more. This kind of activity always felt wrong in the context of the rest of the project, the whole point of which which was to *automate* book production with code, not clicks.

So for this project, I created bookcover[4]: a Racket language for writing book covers as programs. Check out the examples[5] in the documentation to get a feel for how it works.

Writing and packaging `bookcover` was a fun publishing exercise in its own right, and it deserves its own blog post. I learned a lot about Racket in the process.[6] Also, I'm a documentation nerd; I love that Racket's documentation system is itself a full-powered Racket language, and I love the fact that they have such well-considered and accessible style guides for code[7] and prose[8]. It was great to have an excuse to use these tools and to contribute to the Racket ecosystem in some small way.

But the best part is that I now have a way to crank out book covers for future books.[9]

1. https://dicewordbook.com
2. http://docs.racket-lang.org/pollen/index.html
3. https://www.latex-project.org/
4. http://docs.racket-lang.org/bookcover/index.html
5. http://docs.racket-lang.org/bookcover/Overview.html#%28part._.Quick_start%29
6. The web book *Beautiful Racket* (https://beautifulracket.com), and the post *Languages as Dotfiles* (http://blog.racket-lang.org/2017/03/languages-as-dotfiles.html) from the Racket blog, were a huge help in learning and understanding the concepts I needed to write this package.
7. http://docs.racket-lang.org/style/Units_of_Code.html
8. http://docs.racket-lang.org/style/reference-style.html
9. I have wanted to have this kind of tool ever since I read about *Faber Finds* generative book covers

And, if you use Racket, you do too.

As with previous projects, I used LaTeX to produce the book's PDF. LaTeX is itself a programming language, but, like most people, I find its syntax grotesque and arbitrary (kind of like its name). So for just about all the interesting bits I used Pollen[10] as a preprocessor.

For example, here's the part of my LaTeX template that sets the book's measurements:

book.tex.pp

```
% Here is where we configure our paper size and margins.◊
(define UNITS "mm")◊
(define STOCKHEIGHT 152.4)◊
(define STOCKWIDTH 105.0)◊
(define TEXTBLOCKHEIGHT (* (/ 7.0 9.0) STOCKHEIGHT))◊
(define TEXTBLOCKWIDTH (* (/ 7.0 9.0) STOCKWIDTH))◊
(define SPINEMARGIN (/ STOCKWIDTH 8.0))◊
(define UPPERMARGIN (/ STOCKHEIGHT 9.0))◊
(define (c num) (format "~a~a" (real->decimal-string num 3) UNITS))

\setstocksize◊{c[STOCKHEIGHT◊]}{c[STOCKWIDTH]}
\settrimmedsize◊{c[STOCKHEIGHT◊]}{c[STOCKWIDTH]}{*}
\settypeblocksize◊{c[TEXTBLOCKHEIGHT◊]}{c[TEXTBLOCKWIDTH]}{*}
\setlrmargins◊{c[SPINEMARGIN]}{*}{*}
\setulmargins◊{c[UPPERMARGIN]}{*}{*}
\setheadfoot{13pt}{2\onelineskip} % first was originally \onelineskip
\setheaderspaces{*}{\onelineskip}{*}
\checkandfixthelayout
```

The first part of that is sensible Pollen code, the second part is LaTeX with Pollen functions wedged in.

I can't tell you how many times I had to read the LaTeX documentation and scratch my head in order to understand how to code that second half. Looking at it now, I've already forgotten how it works exactly. But thanks to the crystalline little safe space of Pollen written above it, it's easy for me to come back to it, understand what is happening, and how to change it if I want to.

Further down, there is also Pollen code that slurps up the raw raw word list text file[11] from the EFF's website, cooks up LaTeX code for each line, and inserts it all into the book's middle. This means that in future I could simply substitute a different word list and easily generate a book for it.

(https://web.archive.org/web/20081015005111/http://postspectacular.com/process/20080711_faberfindslaunch).

10. http://docs.racket-lang.org/pollen/index.html
11. https://www.eff.org/files/2016/07/18/eff_large_wordlist.txt

I WROTE a short preface to the book with some (I hope) fun and useful information in it: how to use Diceware, how secure is it really, all that sort of thing. But security and word list design are really deep topics, and I wanted some good way of referring the interested reader to more informative reads on these topics.

The problem is, if you put a URL in a printed book, sooner or later it will break: the web page it links to is going to disappear, or the information in it is going to go out of date. Plus, some URLs are really long. Who's going to bother typing in a long web address?

The gods of the internet have provided the solution to the broken link problem in the form of **PURL**: the **P**ersistent **URL**, a service of the Internet Archive[12]. PURL works like a URL shortener such as `bit.ly` or `goo.gl`: it lets you create a short URL, and say, in effect, "make this short URL redirect to this other, long URL." But unlike other URL shorteners, PURL isn't a profit center or a marketing tool: it's a service of the Internet Archive, a nonprofit whose whole *raison d'être* is preserving the Internet, and who will (likely, hopefully) be around for many more decades.

So I made some PURLs of the links I wanted and put them on the last page of the preface as QR codes.

Open the camera app on your phone and point it at one of the codes: it'll take you right to the web page. If any of those pages ever disappear, I can redirect the PURL to a snapshot of it on the Internet Archive, or to another page. This way the links will be good for at least as long as I'm alive, which is more than can be said for the URLs in 99% of the other books in the world.

I'm kind of shocked that more people don't know about and use PURLs. They could, for example, be used as your main "web presence" address, since they are more permanent than even domain names.

There is also surprisingly little guidance from the service itself about how it should be used. The "top-level directory" portion of a PURL (the `jd/` part in the last two PURLs shown above) is called a "domain". PURL domains seem like they should be an incredibly important, protected resource, since they are permanently tied to the account that created them (once you claim a domain, no one else can create PURLs in that domain)—and, once created *can never be deleted!* Despite this, creating a PURL domain is easy, too easy. Almost the first thing you see when you log in is a prompt and a form for creating new domains in one click, with no confirmation, no indication that you are doing something permanent. It's like PURL's design is begging you to mindlessly create as many domains as you can think of.

12. https://archive.org/services/purl/

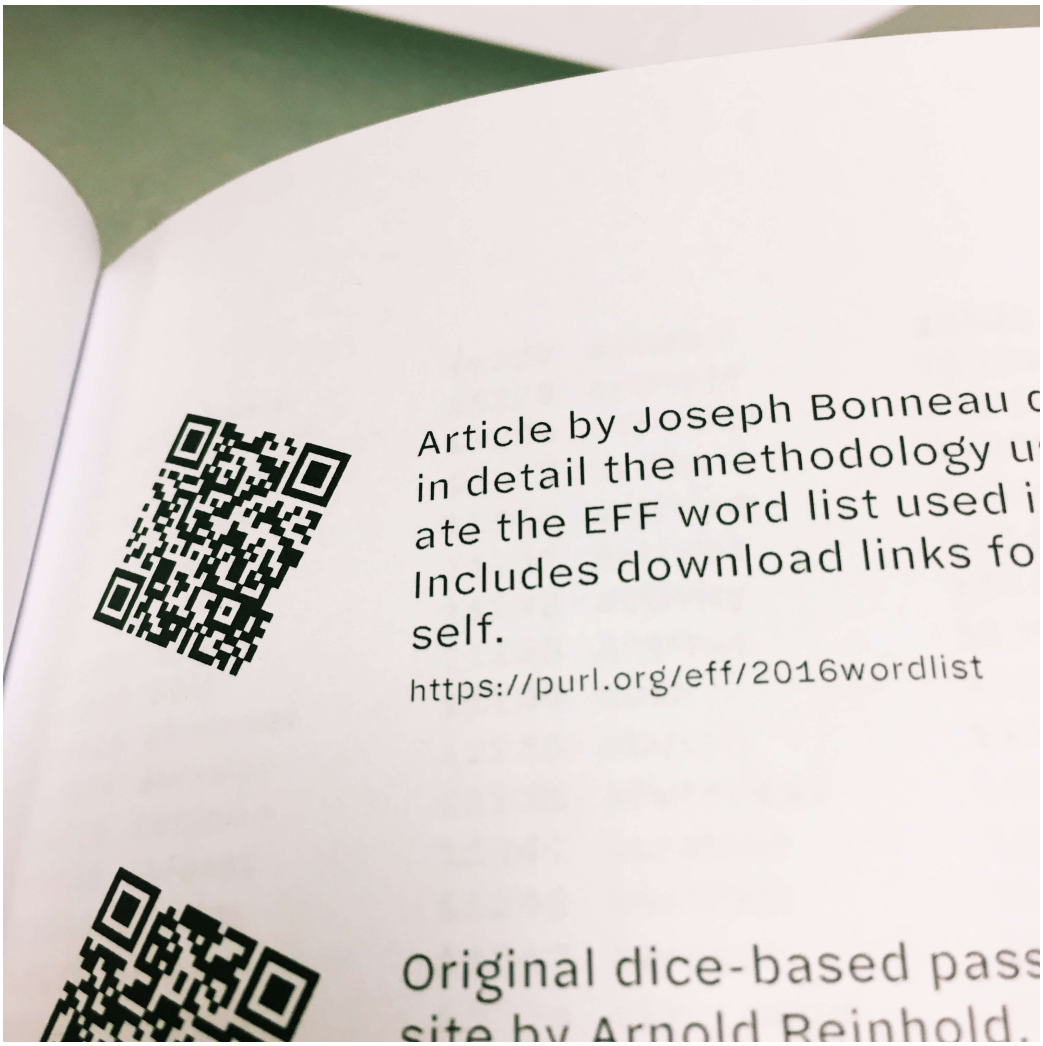

Figure 1: Page of QR codes from the first version of the book

Before I realized this, I had created two PURL domains, one each for the two sites I was linking to: `purl.org/eff` and `purl.org/stdcom`. I'm somewhat embarrassed that I now apparently have permanent irrevocable ownership of these "domains", and am still trying to find out who to contact to remedy this. Meanwhile, I did claim the `jd` domain and will probably be using solely that domain for all the PURLs I create for my own use, for the rest of my life.

Back to the book: here's the LaTeX code I used to generate and place the QR codes:

book.tex.pp

```
% In the header:
\usepackage{pst-barcode}
\usepackage{tabularx}

% …

\newcommand\qrlink[2]{%
   \begin{pspicture}(0.6in,0.6in)%
      \psbarcode[rotate=-25]{#1}{width=0.6 height=0.6}{qrcode}%
   \end{pspicture}%
   & #2 \par {\tiny \url{#1}} \\[0.55cm]%
}

% …and in the document body:

\begin{tabularx}{\textwidth}{m{0.8in} X}
   \qrlink{https://purl.org/eff/2016wordlist}{Article by Joseph Bonneau etc etc.}
   % …etc …
\end{tabularx}
```

I have no wish to explain this in detail, but if you are attempting to do something similar and are already poring over the manuals for the `pst-barcode` and `tabularx` packages, hopefully this will give you something to go on.

I LICENSED two weights of the Halyard Micro[13] typeface for the book, and wanted very much to use it for the headings on the website as well. But this one-page website has only three headings total—not enough to justify the overhead (both technical and financial[14]) of an embedded webfont.

This is where Pollen came in handy again. [15] The fonts' standard EULA says it's fine to

13. http://halyard.dardenstudio.com/
14. http://dardenstudio.com/license/web-addendum
15. I used Pollen as a static site generator for the book's website (https://dicewordbook.com) as well as for the pre-processing of the book itself.

use *images* of the font on a website; so I wrote a function that takes a string, sets it in Halyard Micro, and saves it as an SVG. I then wrote some tag functions that make use of that function. In case it can be of use to anyone, here's all the relevant code:

pollen.rkt

```
(define heading-font (make-parameter "Halyard Micro"))
(define heading-size (make-parameter 20))
(define heading-color (make-parameter "orangered"))
(define heading-sizes '(42 30 24))

(define (normalize-string str)
  (let ([str-nopunct (regexp-replace* #rx"[^0-9a-zA-Z ]" str "")])
    (string-join (string-split (string-foldcase str-nopunct)) "-")))

(define (make-heading-svg level txt)
  (define heading-filename
    (format "img/h~a-~a.svg" level (normalize-string txt)))
  (define heading-pict
    (colorize (text txt (heading-font) (heading-size)) (heading-color)))
  (define the-svg (new svg-dc%
                       [width (pict-width heading-pict)]
                       [height (pict-height heading-pict)]
                       [output heading-filename]
                       [exists 'replace]))

  (send* the-svg
    (start-doc "useless string")
    (start-page))

  (draw-pict heading-pict the-svg 0 0)

  (send* the-svg
    (end-page)
    (end-doc))

  heading-filename)

(define (make-heading level attrs elems)
  (define h-size (list-ref heading-sizes (sub1 level)))
  (define h-str (apply string-append (filter string? (flatten elems))))
  (define h-tag (format-symbol "h~a" level))
  (parameterize ([heading-size h-size])
    `(,h-tag ,attrs (img [[src ,(make-heading-svg level h-str)]
                          [class "heading-svg"]
                          [alt ,h-str]]))))

(define-tag-function (h1 attrs elems) (make-heading 1 attrs elems))
(define-tag-function (h2 attrs elems) (make-heading 2 attrs elems))
```

Let's get into the weeds for a bit here:

The first section sets up some parameters[16] to use as defaults.

The `normalize` function transforms "`My Heading!`" into a string like "`my-heading`", making it ready to use as the base part of a filename—just tack `.svg` at the end.

The `make-heading-svg` function creates the SVG file and saves it in the `img/` subfolder.

This in turn is used by the next function, `make-heading`, as it generates what becomes the `<h1>` or `<h2>` tag.

Finally, Pollen's `define-tag-function` sets up the `h1` and `h2` tags to call `make-heading` with the appropriate heading level.

The upshot is that when, in my source markup, I write:

```
◊
h2{Need Dice? Get the Good Dice.}
```

…it becomes, in the output:

```
<h2><img src="img/h2-need-dice-get-the-good-dice.svg" class="heading-svg" alt="Need Dice? Get the
    Good Dice."></h2>
```

…and of course, when the site is generated, the `.svg` file magically appears in the `img` folder, and everything looks awesome.

Maybe this seems like a lot of code for three headings.[17] As with the book's cover, I could have just made the images by hand in a graphics editor like Pixelmator. But, as with the book cover, since I did it with code rather than by farting around with a mouse, it's very easy to change the headings or make new ones if I ever want to.

There you have it! A little book produced entirely with code. If you have an idea for another one, let me know[18].

16. Parameters (https://docs.racket-lang.org/reference/eval-model.html#(part._parameter-model)) are kind of like Racket's thread-safe equivalent of a global variable, although they work a little differently. You can use `parameterize` to change their value temporarily for a given scope; at the end of that scope the parameter automatically reverts to its previous value.
17. The basic technique of using *an image of text* instead of just the text is basically how we used to use non-standard fonts on the web before `@font-face`. It's bad and dumb in most cases. Don't do it.
18. mailto:comments@thenotepad.org