Pipeline for Static Site Generation

architecture

The pipeline that I use for static site generation produces three files from a single input: 1. Plain Text 2. Responsive Hypertext Markup Language (HTML) 3. Portable Document Format (PDF) Producing three output formats solves the problem of scaling the content from mobile to desktop to print, making reading as simple as possible. A simple, single file change affects all of the content at once.

staticjinja

Jinja is the templating language for Python behind Flask. staticjinja takes the templating power of Jinja, where Flask targets dynamic web content like database backed web stacks, and re-purposes it for static content in batch mode - like jekyll but radically simpler while retaining much of the power that makes static site generators such force multipliers.

```
<!doctype html>
<html lang="en">

<head>
  {% include 'partials/_head.html' %}
</head>

<body>
  {% include 'partials/_header.html' %}
  {% block body %}{% endblock %}
  {% include 'partials/_footer.html' %}

  {% block footer_js %}{% endblock %}
</body>
</html>
```

bash

Bash provides the unix pipe for our code to go from user-contributed src to generated docs for publishing. There are four steps in the process: 1. Run staticjinja taking input templates from src and generating HTML at docs. 2. Run wkhtmltopdf taking a file at our (staging) web server and generating PDF. 3. Run html2text taking HTML as input and generating markdown as output. 4. Run a python script, taking markdown as input and generating plain text as output.

```
#!/usr/bin/env bash
staticjinja build --srcpath=src --outpath=docs
wkhtmltopdf http://localhost:8000/index.html index.pdf
html2text index.html >index.md
python ./md2txt.py
```

python

The python code just takes markdown as input, transforms the markdown to plain text using this patch from stackoverflow, and outputs plain text.

```
def main():
    md_file = open('index.md')
    txt = md_file.read()
    plain_txt = unmark(txt)
    n = open('index.txt', 'w')
```

```
n.write(plain_txt)
```

design

The presentation is simpflified by the use of a classless CSS framework called new.css, which makes achieving simple and clean output straightforward. I arrived at classless frameworks after years of working with bootstrap to achieve similar tasks, albeit with a ton of code, work, and bloat. Of course, what the term classless is getting at is that all of these CSS frameworks, much like the JS frameworks that acommpany them, end up permeating systems to the point where everyone has to be a designer, whether they like it or not. What CSS should really do is provide an abstraction, not a leaky abstraction. When CSS permeates every square inch of a web pages surface area that's what's happening. What classless CSS frameworks aspire to do is push down those style details, where they belong, in the back-end of the system, not the front-end. So if you look at a piece of HTML from my site, what you'll notice is not that there's a complete lack of CSS, but rather, the default CSS is so well-styled by the framework authors that no cascading style is necessary, even though it's readily available.

summary

Modeling the resume production problem this way has benefits for reader and writer. The design takes advantage of an automated data pipeline built on Unix pipes so that the writer can focus on writing content. The beauty of a system like this is that it is simple and reliable, depending on common open source software and free cloud resources. You get the rich content flow of a modern dynamic Content Management System (CMS), except there is no database and no application server, just a file system and some event-driven glue code, so it doesn't take an IT team to support it.