Creating printable content from the web

All the stylesheets are written in SCSS and convetrted to CSS through the build process.

In early 2015 I played with the idea of creating a custom XML vocabulary and the associated style sheets (XSLT, CSS and CSS Paged Media) to convert it to HTML, EPUB and PDF. It worked, the files were fairly easy to work with (at least for me who created them) and it was a good way to learn how to write XML schemas, XSLT style sheets and hone my skills with CSS. The repository is available

While it works the project relies on a custom XML vocabulary and it would take some work to convert to use (x)HTML5. I think I would start with the CSS stylesheet, change it to suit the needs of shorter content, integrate the PDF creation system to the existing Gulp build system and host the PDF content in a third party space like Amazon S3.

Differences with @print media

As far as I understand the differences is that @print media type was designed to make content printable but not to create a printed version of the content we're working with but it can be printed directly from the browser.

In an ideal world Paged Media would be fully supported by all browsers and some of the work we're doing in this project would be unnecessary but it is not an ideal world.

In this world we create pages for each type of content we want and associate portions of HTML to each of the page types we created. We can then add additional material as needed. We then use a third party tool (all commercial unfortunately) to process the HTML with a the paged media style sheet we just created and produce PDF or other formats depending on the formatter

For a model to follow when using the stylesheet check Oreilly's HTMLBook specification

A good example of a book created with this specification and edited with Atlas is Lea Verou's CSS Secrets

Creating the paged media work

We'll break this process into three parts:

1. Creating the additional paged media style sheet
2. Modify the HTML template and add additional HTML to our markup
3. Create additional Gulp tasks to automate the process

Creating the additional paged media style sheet

This stylesheets sets up a printed stylesheet with a basic set of parameters. The stylesheet works as is but it's also meant as a starting point for printed media work and you can certainly refine it based on your needs.

The style sheet will control the print layout but not any other attribute controlled by CSS. In this case we've taken care of this in the stylesheet associated with the template, otherwise you'll have to provide your own styles.

We begin the process by defining our standard page as an 8.5 x 11 inch page (US Letter) and give it default margins of 1 inch. We also define a `footnote` at-rule to increase the count on each page.

Next we define add defaults attributes for the book defined as `body[data-type="book"]`.

In this element we define the default text color. Paged media agents (at least PrinceXML) supports CMYK colors so we use that format for opaque black. If the rendering engine doesn't support CMYK we fallback to RGBA for the same color.

If you're using Pantone colors you can use this handy converter to get the fallback RGBA color.

Lastly we define the default hyphenation

The HTML version of the document centers the `.container` element and makes it 48em (768 pixels) wide. For printed media we want to work with the full

width of the document so we remove the centering and make the width 100%. It will not affect the HTML version of the document because it's in a separate style sheet that will never be called together with the `main.scss` styles.

```
/* DEFINE THE DEFAULT PAGE */
@page {
  size: 8.5in 11in;
  margin: 1in;
  /* Footnote related attributes */
  @footnote {
    counter-increment: footnote;
    float: bottom;
    column-span: all;
    height: auto;
    }
  }

/* Default definitions for the book*/
body[data-type='book'] {
  color: cmyk(0%,0%,100%,100%);
  color: rgba(0, 0 ,0 ,1);
  hyphens: auto;
}

.container {
  width: 100%;
}
```

This is a small block to define how we'll in handle increasing the page number counter.

The first part of the book and the first article will reset the page number counter (page) because we would normally have front matter using a different numbering system and we don't want that number to carry over to the beginning of our content.

THe second selector in this block will not reset counters when a article child of body is followed by a part. We have to be explicit to use this rule because, otherwise, it would trigger the `:first-of-type` rule immediately above.

```
/* PAGE COUNTERS */
body[data-type='book'] > div[data-type='part']:first-of-type,
body[data-type='book'] > section[data-type='article']:first-of-type {
  counter-reset: page;
  counter-reset: footnote;
}
body[data-type='book'] >  section[data-type='article'] + div[data-type='pa
   counter-reset: none; }
```

Other than book (defined as an attribute of body) and part (defined as attribute of div elements) the rest of our book is defined in section elements. This is a two step process.

In the first step (shown below) we associate a given type (`data-type`) with a page pseudo element we'll define later in the stylesheet. In this block we also define page specific attributes for each type of page. Most of the pages break before (`page-break-before`) and after (`page-break-after`). THe only additional element that I've placed in this block is the dotted leader for the table of content's page number.

`section[data-type='toc'] nav ol li a:after` will place a dotted leader followed by the target content's page number. Because we want this to work only on the table of contents we have to be too specific. In most other circumstances I may shorten the selector.

To create additional content type, create a new definition here and a @page definition later in the document.

```
/* MATCH THE PARTS OF OUR BOOK FILE TO EACH OF OUR PAGES DEFINED LATER */
/* Title Page*/
section[data-type='titlepage'] {
  page: titlepage;
  page-break-before: always;
  page-break-after: always;
}

/* Copyright page */
section[data-type='copyright'] {
```

```
  page: copyright;
  page-break-before: always;
  page-break-after: always;
}

/* Dedication */
section[data-type='dedication'] {
  page: dedication;
  page-break-before: always;
  page-break-after: always;
}

/* TOC */
section[data-type='toc'] {
  page: toc;
  page-break-before: always;
  page-break-after: always;
}
/* Leader for toc page */
section[data-type='toc'] nav ol li a:after {
  content: leader(dotted) ' ' target-counter(attr(href, url), page);;
}

' '/* Foreword  */
section[data-type='foreword'] { page: foreword; }

/* Preface*/
section[data-type='preface'] { page: preface; }

/* Part */
div[data-type='part'] { page: part; }

/* Chapter */
section[data-type='article'] {
  page: article;
  page-break-before: always;
}
```

```
/* Appendix */
section[data-type='appendix'] {
  page: appendix;
  page-break-before: always;
}

/* Glossary*/
section[data-type='glossary'] { page: glossary; }

/* Bibliography */
section[data-type='bibliography'] { page: bibliography; }

/* Index */
section[data-type='index'] { page: index; }

/* Colophon */
section[data-type='colophon'] { page: colophon; }
```

These are the page definitions. We have two options, defining both left and right versions of the page (`toc:right` and `toc:left`) for content that changes between both sides or create a single page template (`toc`) that will apply equally to left and right sides.

It is here where we set specific attributes for the different types of pages we're working with. For example, the front-matter pages are formatted with lower case roman numerals where the main body uses arabic numerals (this is another reason why we reset the counter on first part or article).

```
/* Comon Front Mater Page Numbering in lowercase ROMAN numerals*/
@page toc:right {
  @bottom-right-corner { content: counter(page, lower-roman);; }
  @bottom-left-corner { content: normal; }
}

@page toc:left  {
  @bottom-left-corner { content: counter(page, lower-roman); }
  @bottom-right-corner { content: normal; }
}
```

```
@page foreword:right {
  @bottom-center { content: counter(page, lower-roman); }
  @bottom-left-corner { content: normal; }
}

@page foreword:left  {
  @bottom-left-corner { content: counter(page, lower-roman); }
  @bottom-right-corner { content: normal; }
}


@page preface:right {
  @bottom-center {content: counter(page, lower-roman);}
  @bottom-right-corner { content: normal; }
  @bottom-left-corner { content: normal; }
}

@page preface:left  {
  @bottom-center {content: counter(page, lower-roman);}
  @bottom-right-corner { content: normal; }
  @bottom-left-corner { content: normal; }
}

/* Common Content Page Numbering  in Arabic numerals 1... 199 */
@page titlepage{
/* Need this to clean up page numbers in titlepage in Prince*/
  margin-top: 18em;
  @bottom-right-corner { content: normal; }
  @bottom-left-corner { content: normal; }
}

@page dedication {
/* Need this to clean up page numbers in titlepage in Prince*/
  page-break-before: always;
  margin-top: 18em;
  @bottom-right-corner { content: normal; }
```

```
  @bottom-left-corner { content: normal; }

}

@page article {
  @bottom-center {
    vertical-align: middle;
    text-align: center;
    content: element(heading);
  }
}

@page article:blank { /* Need this to clean up page numbers in titlepage
  @top-center { content: "This page is intentionally left blank"; }
  "This page is intentionally left blank"@bottom-left-corner { content: no
  @bottom-right-corner {content:normal;}
}

@page article:right  {
  @bottom-right-corner { content: counter(page); }
  @bottom-left-corner { content: normal; }
}

@page article:left {
  @bottom-left-corner { content: counter(page); }
  @bottom-right-corner { content: normal; }
}

@page appendix:right  {
  @bottom-right-corner { content: counter(page); }
  @bottom-left-corner { content: normal; }
}

@page appendix:left {
  @bottom-left-corner { content: counter(page); }
  @bottom-right-corner { content: normal; }
}
```

```
@page glossary:right {
  @bottom-right-corner { content: counter(page); }
  @bottom-left-corner { content: normal; }
}

@page glossary:left {
  @bottom-left-corner { content: counter(page); }
  @bottom-right-corner { content: normal; }
}

@page bibliography:right  {
  @bottom-right-corner { content: counter(page); }
  @bottom-left-corner { content: normal; }
}

@page bibliography:left {
  @bottom-left-corner { content: counter(page); }
  @bottom-right-corner { content: normal; }
}

@page index:right  {
  @bottom-right-corner { content: counter(page); }
  @bottom-left-corner { content: normal; }
}

@page index:left {
  @bottom-left-corner { content: counter(page); }
  @bottom-right-corner { content: normal; }
}
```

To create a running footer we'll use the running() value to use the element represented with the `rh` to do the following:

- Take it out of the regular flow of the document
- Use it as the text of the running header
- Make the text italic and center it

```
p.rh {
```

```
  position: running(heading);
  text-align: center;
  font-style: italic;
}
```

Footnotes are a little more complicated than most of what we've seen so far

Figure 1: Footnote terminology as it relates to generated page media

Now that we've the terminology let's dive into specifics

footnote element (`span.footnote`)
: The element containing the content of the footnote, which will be removed from the flow and displayed as a footnote.

footnote marker (also known as footnote number) (`::footnote-marker`)
: A number or symbol adjacent to the footnote body, identifying the particular footnote. The footnote marker should use the same number or symbol as the corresponding footnote call, although the marker may contain additional punctuation (the content of the `::fotnote-marker::after` selector).

footnote body (content inside `span.footnote`)
: The footnote marker is placed before the footnote element, and together they represent the footnote body.

footnote call (also known as footnote reference)
: A number or symbol, found in the main text, which points to the footnote body. The default is to use Arabic numbers but it can be customized to use other symbols.

footnote area
: The page area used to display footnotes. Usually located at the bottom of a page

footnote rule (also known as footnote separator)

A horizontal rule is often used to separate the footnote area from the rest of the page. The separator (and the entire footnote area) cannot be rendered on a page with no footnotes.

The footnote element will be removed from the flow of text and replaced by the footnote call symbol.

The footnote counter number increases.

The footnote body is placed in the footnote area in document order

```
/* Footnotes */
span.footnote {
  float: footnote;
}

::footnote-marker {
  content: counter(footnote);
  list-style-position: inside;
}

::footnote-marker::after {
  content: '. ';
}

'. '::footnote-call {
  content: counter(footnote);
  vertical-align: super;
  font-size: 65%;
}
```

Creating cross references is as simple as creating an internal link with the `xref` class like this: `<a href="#target" class="xref"`. The style sheet will then append the page where the reference is located using the target-counter() function.

```
/* XReferences */
```

```
a.xref[href]::after {
    content: ' [See page ' target-counter(attr(href' [See page '
}
```

The last section of the style sheet is the PDF bookmarks.

The bookmarks use heading tags (h1 through h6) as the indicators for where they will land and what content to display as the label for the bookmark. The first three levels of headings will also be open so you can see what the content is. Since I seldom use headings beyond h3 I don't think it's necessary to show them; however they are available in case I need them for a specific project.

We also use vendor prefixes for the bookmark attributes. Even though bookmarks are part of the CSS Generated Content for Paged Media Module vendors have impemented the feature under prefix (boo!)

One enhancement I definitely want to do is to change the selector for the bookmarks. It currently only works for article content. I definitely want to make all headings (particularly those in front matter and bibliography sections) visible as well. It should be a matter of removing the attribute selector.

```
/* PDF BOOKMARKS */
section[data-type='article'] h1 {
  -ah-bookmark-level: 1;
  -ah-bookmark-state: open;
  -ah-bookmark-label: content();
  prince-bookmark-level: 1;
  prince-bookmark-state: open;
  prince-bookmark-label: content();
}

section[data-type='article'] h2 {
  -ah-bookmark-level: 2;
  -ah-bookmark-state: open;
  -ah-bookmark-label: content();
  prince-bookmark-level: 2;
```

```
  prince-bookmark-state: open;
  prince-bookmark-label: content();
}

section[data-type='article'] h3 {
  -ah-bookmark-level: 3;
  -ah-bookmark-state: open;
  -ah-bookmark-label: content();
  prince-bookmark-level: 3;
  prince-bookmark-state: open;
  prince-bookmark-label: content();
}

section[data-type='article'] h4 {
  -ah-bookmark-level: 4;
  prince-bookmark-level: 4;
}

section[data-type='article'] h5 {
  -ah-bookmark-level: 5;
  prince-bookmark-level: 5;
}

section[data-type='article'] h6 {
  -ah-bookmark-level: 6;
  prince-bookmark-level: 6;
}
```

Making articles instead of books

The style sheet we discussed in the previous section works well enough for most documents. We can create book-like and article-like content. Rather than use the generic stylesheet I looked at what would it take to minimize the stylesheet for articles since I think articles better represent the content of the blog posts.

The style sheet is an abbreviated version of the stylesheet we discussed in the last section. I'll point out the differences where necessary.

In the first section we do all our imports. We import variables and mixins, create our font-face declarations and import the rest of our SCSS rules.

```
// PRELIMINARY IMPORTS
//
// Import variables and mixins
@import 'partials/variables';
'partials/variables'@import
//fonts
@import 'partials/fonts';

/* Monospaced font f'partials/fonts'/* Monospaced font for code samples */
@include, '../fonts/notomono-regular-webfont', normal, normal);
/* Regular font */
@include font-dec'../fonts/notomono-regular-webfont'/* Regular font */
@includet', normal, normal);
/* Bold font */
@include font-declaration('notosans_bold', '../fonts/no'notosans_bold'/* B
@include Italic Font */
@include font-declaration('notosans_italics', '../fonts/notosans-ita'notos
@includeic font */
@include font-declaration('notosans_bolditalic', '../fonts/notosans-boldit
@includeport 'partials/pm/columns';
@import 'partials/pm/marginalia';
@import 'partials/pm/paragraphs';
@'partials/pm/columns'// Import other partials
@importes';
@import 'partials/pm/headings';

.container {
  width'partials/pm/headings'@import 'partials/pm/paragraphs';
@import 'partials/pm/lists';
@import 'partials/pm/images';
@import 'partials/pm/headings';

.container {
  width: 100%;
}
```

The following section is similar to the default page setup on the previous section. We set up the default page as a US Letter page with 1 inch margins. We then do the footnote-related work.

We associate the `[data-type="article"]` with our article @page instead of `[data-type="book"]`. We also assign the font we want to use to the article, including a fallback font and a generic font. We do the same thing with our `[data-type="bibliography]` and the bibliography @page. I kept bibliography because this is the only section I think I may use in my posts.

```
@page {
  size: 8.5in 11in;
  margin: 1in;
  /* Footnote related attributes */
  counter-reset: figure;
  @footnote {
    counter-increment: footnote;
    float: bottom;
    column-span: all;
    height: auto;
  }
}

body[data-type="article"] {
  font-family: notosans_regular, Verdana, sans-serif;
  font-size: 12pt;
  line-height: 1.375;
  // Widow and orphan control
  orphans: 4;
  widows: 2;
  // counter resets
  counter-reset: footnote;
  counter-reset: figures;
  // page related
  page: article;
  page-break-before: always;
  page-break-after: always;

}
```

```
section[data-type='bibliography'] {
  font-family: notosans_regular, Verdana, sans-serif;
  font-size: 12pt;
  line-height: 1.375;
  // Widow and orphan control
  orphans: 4;
  widows: 2;

  page: bibliography;
  page-break-before: always;
  page-break-after: always;

  p {
    text-align: left;
    text-indent: 0 !important;
  }
}
```

For some reason, I believe markup related, Prince is not picking up the fonts correctly. It sometimes picks the web font (`notosans_regular`)and some times it picks the fallback (`Verdana`).

I've fixed the issue by specifying the font to use for most tags. It's not a show stopper but it's important to research the cause and I wonder if it's the way the styles sheet is written.

```
h1, h2, h3, h4, h5, h6,
p, li {
  font-family: notosans_regular, Verdana, sans-serif;
}

pre, code {
  font-family: notomono_regular, Consolas, Monaco, 'Andale Mono', 'Ubuntu
}

'Andale Mono'strong, b {
```

```
  font-family: notosans_bold, Verdana, sans-serif;
}

em, i {
  font-family: notosans_italic, Verdana, sans-serif;
}
```

When defining the pages I've included both page numbering and any generated text that I want to use beyond the content of the Markdown/HTML content.

```
@page article:left {
  @bottom-right {
    content: counter(page)
  }
}

@page article:right {
  @bottom-right {
    content: counter(page)
  }
}

@page article:blank {
  /* Need this to clean up page numbers in titlepage in Prince*/
  @top-center {
    content: "This page is intentionally left blank"
  }
  "This page is intentionally left blank"@bottom-left-corner {
    content: normal;
  }
  @bottom-right-corner {
    content: normal;
  }
}

@page bibliography:left {
  @bottom-right {
```

```
    content: counter(page)
  }
}

@page bibliography:right {
  @bottom-right {
    content: counter(page)
  }
}
```

The definition of footnotes and cross references remain the same as it did for our book-like content.

```
/* Footnotes */
span.footnote {
  float: footnote;
}

::footnote-marker {
  content: counter(footnote);
  list-style-position: inside;
}

::footnote-marker::after {
  content: '. ';
}

'. '::footnote-call {
  content: counter(footnote);
  vertical-align: super;
  font-size: 65%;
}

/* XReferences */
a.xref[href][href]::after content: ' [See page ' target-counter(attr(href
}
' [See page '
```

We use the same code for bookmarks. I only included the bookmarks for h1 to h3. The full stylesheet also include the code for h4 to h6.

```
/* PDF BOOKMARKS */
section[data-type='article'] h1 {
  -ah-bookmark-level: 1;
  -ah-bookmark-state: open;
  -ah-bookmark-label: content();
  prince-bookmark-level: 1;
  prince-bookmark-state: open;
  prince-bookmark-label: content();
}

section[data-type='article'] h2 {
  -ah-bookmark-level: 2;
  -ah-bookmark-state: open;
  -ah-bookmark-label: content();
  prince-bookmark-level: 2;
  prince-bookmark-state: open;
  prince-bookmark-label: content();
}

section[data-type='article'] h3 {
  -ah-bookmark-level: 3;
  -ah-bookmark-state: open;
  -ah-bookmark-label: content();
  prince-bookmark-level: 3;
  prince-bookmark-state: open;
  prince-bookmark-label: content();
}
```

The last block is styles that do not affect the page layout. The only style that is interesting is the automatic figure numbering that we do for the `figcaption element`. We use the `figures` counter and increase it for every figure element in the page. The `figcaption` element then takes that counter attribute and adds text around it to create the counter that will be prepended to the image.

```
small,
```

```
.font-small {
  font-size: .833em;
}

.justified {
  text-align: justify;
}

pre {
  background-color: #ddd;
  padding: 1em;
  overflow-wrap: break-word;
  white-space: pre-wrap;
  word-wrap: break-word;
}

.footnote {
  font-family: notosans_regular, verdana, sans-serif ;
}

figure {
  counter-increment: figures;
  img {
    width: min-content;
  }

  figcaption::before {
    content: 'Fig. ' counter(figures) ' - ';
    font-size: 75%;
  }

  'Fig. 'figcaption {
    font-size: 75%;
    max-width: min-content;
    color: #ddd;
  }
}
```

```
/* Informational messages */
.message {
  border: 1px solid black;
  border-radius: 10px;
  display: block;
  padding: .5em;
  margin: 1em 0;

  &.info {
    background-color: lightblue;
    font-weight: bold;
  }

  &.warning {
    background-color: lightyellow;
    font-weight: bold;
  }

  &.danger {
    background-color: indianred;
    font-weight: bold;
  }
}
```

Changes we need to make to our HTML document (or template)

As I was working on implementing the style sheets for printed media I realized a few things that would require changes on the HTML template. These things are:

- The styles on `main.css` do not display properly in a printed style sheet. Haven't figured out if it's an issue with the way I set up the styles for web content
- Prism styles are not fully supported in PrinceXML and entire selectors are ignored as a result
- Still need a way to do syntax highlighting

- I need to add an attribute (`data-type="article"`) to the body tag

```
<html lang="en">
<head>
<head><!--<link rel="stylesheet" href="css/main.css">--> <!-- 1 -->
<!--<link rel="stylesheet" href="css/prism.css">--> <!-- 2 --><script asyn
<script src="paged-media/highlight/high<!-- 3 --><!-- 3 -->ipt> <!-- 3 -->
<!--<script async src="scripts/vendor/fontf<!-- 3 --><!-- 3 -->
<!--<script async src="scripts/vendor/fontfaceobserver.standalone.js"></sc
<!--<script async src="scripts/load-fonts.js"></script>-->  <!-- 4 -->dth
<title></title>
</head>

<body data-type="article"> <!-- 5 -->
<div class="container">
    <%= contents %>
</div>

</body><title><!-- 5 -->
<div class="container">
    <%= contents %>
</div>

</body>
</html>
```

The new template performs the following tasks:

1. Removes `main.css` and the styles in it
2. Removes Prism style sheet and the Prism script tag
3. Adds Highlight.js script and the solarized-light theme style sheet
4. Removes `fontfaceobserver.js` and `load-fonts.js` scripts. Since we don't have to worry about font loading timing out and I'm not sure if PrinceXML supports the way we observe font loading it's better if we remove them altogether
5. Adds `data-type="article"` as an attribute to the body tag. This will match with selectors on the paged media style sheet and make it work on PDF

The template is located in the template directory as `template-pm.html`

Additional Gulp tasks

Now that we know what we want to do and have the paged media template and style sheets we need to look at how to integrate them into the build process. Just to recap, we want to do the following:

- Convert markdown into an HTML fragment
- Insert the fragment into an HTML template customized for paged media output
- Run a Paged Media Processor (PrinceXML) on the output
- Copy the files to a specific directory for the fact that none of my tools will let me easily do it
- Upload the PDF files to an Amazon S3 bucket

We've already got tasks for the first two items in our list. The first task, converting Markdown files to HTML fragments doesn't need to change so we'll leave it as is.

We'll customize the task to complete the second item on the list and change the template file and destination for the custom HTML content. We still make this task depend on the Markdown converter task to make sure we have the newest content to work. The new template looks like this:

```
gulp.task('build-pm-template', ['markdown'], () =>{
  gulp.sr'markdown'tml-content/*.html')
    .pipe($.wrap({src: './src/templates/template-pm.html'}))
    .pipe(gulp.dest('./src/pm-content'))
});
```

I originally thought about using `gulp-shell` to run the PrinceXML command that I run to dogfood the code I present in these posts. The command is shown below:

```
prince \
--verbose \
--input=html \
--javascript \
--style css/article-styles.css \
```

```
paged-media-revisit.html -o doc-name.pdf
```

Where you see a backslash treat it as a continuation character, the command and all the parameters could be written in a single line. Also note that we run this command from inside the `src` directory. We'll see how this is different when we run the Gulp task

`gulp-shell` presented too many unknowns. Since it's blacklisted by Gulp so I'm not 100% sure how well it would work. So I chose to use gulp-exec instead. I took the template from the plugin's readme page and plugged in the data for the prince command.

This is a new plugin so we need to install it and save it as a development dependency. The command should look familiar:

```
npm install --save-dev gulp-exec
```

then require the task at the top of the gulpfile.

```
var exec = require('gulp-exec');
```

The `build-pdf` task depends on `build-pm-template` to make sure we have the latest content to convert to PDF. We then run the same Prince command we used to test the content from the command line with one difference. The path to the style sheet needs to be modified because we are running Gulp from the root of the repository. The full path to the style sheet becomes `src/css/article-styles.css`.

```
gulp.task('build-pdf', ['build-pm-template'],'build-pm-template'ons = {
    continueOnError: false,
    pipeStdout: false,
  };
  let reportOptions = {
    err: true,
    stderr: true,
    stdout: true
```

```
  };
  return gulp.src('./src/pm-content/*.html')
    .pipe(exec('prince --verbose --in'./src/pm-content/*.html'tyle ./src/
    .pipe(exec.reporter(reportOptions));
});
```

Because I can't figure out how to move the output of `build-pdf` I'm creating a workaround task to copy the PDF content form `pm-content` to a dedicated `pdf` directory. It uses Gulp built-in methods. It takes the source as the list of files that we want to copy and pipes it to the destination directory we want to put the files in.

```
gulp.task('copy-pdf', ['build-pdf'], () => 'build-pdf'c('src/pm-content/*
    .pipe(gulp.dest('src/pdf')'src/pm-content/*.pdf'
```

The last task I want to build is one to upload the PDF files to an Amazon S3 bucket. I will link to the bucket items from my blog or from other places where I want to use this information.

I've chosen to use `gulp-s3-upload` for the first iteration of the task. Whenever I use Amazon I struggle with how to configure the task with Amazon Web Services is how to configure the login credentials. I've managed to stay under the radar for a while and have yet to pay for AWS hosting. I want to keep it that way.

So let's see if it works. First install the `gulp-s3-upload` plugin and save it to your package.json's dev dependencies.

```
npm install --save-dev gulp-s3-upload
```

Requiring the plugin is unusual, to me, because you attach the configuration to the require statement. I've left the credentials blank here but they are in the gulpfile, for now.

```
var s3 = require('gulp-s3-upload')({
  accessKeyId: "enter yours",
```

```
  secretAccessKey: "enter yours"
})
```

The task itself is fairly simple. we take all the PDF files located in `src/pm-content` and pipe them through S3 to upload them to the `blog-post-pdf` bucket with a public-read ACL. If I understood the settings correctly this would allow me to link to the files from external servers and not have to worry about CORS or any other types of restriction.

```
gulp.task('s3-upload', () => {
  gulp.src("./src/pm-content/*.p"./src/pm-content/*.pdf"cket: 'blog-post-p
    ACL: 'public-read'
  }))
});
```

Another option I've been considering is moving all my externally stored files from AWS to Google Cloud and their storage system. This is totally experimental and may not be my long term solution; I'm presenting it here as an alternative to AWS.

For Google Cloud I've chosen to use `gulp-gzip` and `gulp-gcloud-publish`. I've chosen to gzip the files on upload to reduce the size of the payload being uploaded and the storage requirements... Google Cloud will unzip the files on download.

To install the plugins we follow the standard Node procedure. Make sure you save them as dependencies with a command like the one below

```
npm install --save-dev gulp-gcloud-publish gulp-gzip
```

The plugins takes all the PDF files in the specified directory, gzips them and then uploads them to Google Cloud Storage with the information provided. Because this requires a private key, it is essential that you keep the key referenced by `keyFileName` private. I've added the file to `.gitignore` and will not share than information. Please respect that and get your own account.

```
gulp.task('gcs-publish', () => {
```

```
  return gulp.src('src/pdf/*.pdf')
    .p'src/pdf/*.pdf'optional
    .pipe(gcPub({
      bucket: 'use your own',
      keyFilename: 'use your own',
      projectId: 'use your own',
      base: '/pdf',
      public: true
    }));
});
```

Whatever infrastructure you use is up to you. Most of it will depend on your existing infrastructure you use and whatever you're comfortable with.

.gitignore

To make sure I don't upload product files to Github I've created a `.gitignore` file that removes most of the generated files from the Git upload process.

If you use commercial fonts you definitely dont' want to upload them to a public repository as it would break all sorts of licenses and make you liable.

We don't want our Node modules uploaded to Github to make sure that the modules match your system rather than mine. This is particularly important for binary compiled modules... don't be lazy and install your own to make sure they work as intended.

```
# Project specific files #
##########################
.idea
node_modules
.sass-cache
fonts/
src/**/*.html
dist/**/*.html
src/pm-content/
src/pdf
```

```
src/md-content/*.md
src/html-content/*.html
```

Conclusion

We've taken the same Markdown code, converted it to an HTML fragment, inserted the fragment into a template, used PrinceXML to process the file and a special printed media style sheet to generate PDF and upload the resulting PDF files to an Amazon S3 bucket or to a Google Cloud Storage bucket.

We can further customize the project by creating new CSS stylesheets for HTML and Paged Media content to take advantage of additional functionality available to either the printed or HTML versions of the content.

Let me know how this works out for you. If you have bugs or ideas to report, use the repository's issue tracker for communication.

Known issues

- There is at least one instance where the fallback font is used where the primary (web) font is used in other parts of the document
- PrinceXML doesn't support videos. This is problematic since Youtube and Vimeo use iframes and doesn't offer a way to add a poster frame and causes either a blank or black space (depending on the video provider). Only way to deal with the issue is to swap the video for an image as a pre-processor step or using Javascript
- Everything gets rebuilt every time we run the commands. There are ways to only work on files that have changed. This is particularly important when uploading content to the S3 bucket, we don't want to use excessive bandwidth and have to pay for it
- S3 credentials are in the clear. Figure out a way to either use a config file in the local file system or some other way to hide the credentials from users other than me
- Haven't quite figured out if the tool will overwrite the content already in the S3 bucket
- The first time you run make-pdf or pdf-make (the full task) it may produce no result. Running it a second time fixes the issue
- When running pdf-make (the full task) copy-pdf runs before the PDF files

are generated even when running it in sequence. If make-pdf runs synchronously i need to figure out why and how to solve the problem

links and resources

- References
    - Building books with CSS3
    - HTMLBook pdf stylesheet;
- W3C specifications
    - CSS Paged Media Module Level 3
    - CSS Genearated Content for Paged Media Module
- Formatter information
    - Antenna House Formatter Onine Manual
    - Prince XML User Guide
    - Viviostyle formatter release notes