Revisting HTML To PDF with CSS Paged Media

I’ve looked at CSS Paged Media in the past. IMO it’s an awesome technology that allows you to create stunning end products from your HTML, CSS and, optionally Javascript.

The layouts for the printed material can be very simple, or they can be as complex as the book CSS Secrets by Lea Verou.

I’ve worked with a two different CSS Paged Media processors: PrinceXML and Antenna House. They are both good products but they require a license purchase before you can use their full capabilities.

The new products I want to review are:

- PDFreactor
- Weasyprint
- typeset.sh

When I discuss individual products I will look at the following:

- Can they create a PDF from the combination of HTML and CSS Paged Media?
- Do they support Javascript?
- Do they support equivalent features to PrinceXML for what the stylesheet already does?
- What license do they use and what limits does it impose on production use?

The CSS

Before we start looking at individual tools, I want to show what the CSS Paged Media stylesheet is.

We should start by poiniting out that this only deals with the structured of the printed pages. It doesn’t deal with styles, fonts and other layout elements.

Use the following links for quick references to Paged Media CSS and Generated Content for CSS:

- CSS Paged Media Module Level 3
- CSS Generated Content Module Level 3

Now, let’s dive into the code.

In order to understand the CSS that comes next we need to understand a few terms:

- Page Selectors, particularly Spread pseudo-classes to better understand the different values of page that we use
- Page-Margin Boxes to understand the boxes you have available to work with and what layout possibilities they open for you.

The table below, shows the different page boxes available per the Paged Media Specification.

top-left-corner	top-left	top-center	top-right	top-right-corner
left-top	main page area			right-top
left-middle				right-middle
left-bottom				right-bottom
bottom-left-corner	bottom-left	bottom-center	bottom-right	bottom-right-corner

The first thing we do is set up our default page type using the @page rule.

Since the @page declaration doesn’t have a name attached to it, it will apply to all pages in the document.

We’ve added the dimensions of the page, margins, using the single value syntax (all margins will be the same size) and information about the footnotes placement on the page.

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

Because we don't know if the book will be printed as duplex or single sided, we need to make sure that the margins closes to the binding are slightly larger, otherwise we might not be able to read the content. However, if you're just printing the document in a single sided format you can remove these rules and let all margins be the same on all pages.

```
@page :right {
  margin-left: 1.5in;
}

@page :left {
  margin-right: 1.5in;
}
```

We then define the page to be used for chapters. The page rule contains a positioned block of content, in this case the chapter heading or title.

The first block defines items that are common to all chapter pages:

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

We then define items specific to left and right pages. We want to alternate page numbers on right and left pages.

```
/* Right Side*/
@page chapter:right  {
  @bottom-right-corner {
    content: counter(page)
  }
  @bottom-left-corner {
    content: normal
  }
}

/* Left Side */
@page chapter:left {
  @bottom-left-corner {
    content: counter(page)
  }
  @bottom-right-corner {
    content: normal
  }
}
```

We don’t want page numbering on the title page so we make sure we reset both left and right bottom corners to be empty. Not putting the page number on a page doesn’t change the page count, the page counter will increase for each page regardless.

```
@page titlepage {
  @bottom-right-corner { content: normal }
  @bottom-left-corner { content: normal }
}
```

Most of the other types of pages (appendix, glossary, bibliography and index) use the same page defiition style as the chapter pages. We assign the page number to the corresponding bottom corner.

We could do it in the generic `page:right` and `page:left` declarations but that makes it easy to forget that we can customize each page type independently.

Some page types (toc, foreword, and preface) further customize the page numbering by using lowercase Roman numberals

```
@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
  }
}

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

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

To define the structure of the book in markup we use the `data-type` attributes. The root of the book example is in the body element.

The body element defines the default color for the document in both CMYK (for print) and HSL for online viewing and as a fallback in case the processor doesn't support CMYK.

It also defines the hyphenation default behavior. We set it up to hyphenate by default (the `auto` value)

```
body[data-type="book"] {
  color: cmyk(0%,0%,0%,100%);
  color: hsl(0,0%,0%);
  hyphens: auto;
}
```

Next we look at counters. To me, this is the trickiest part of creating paged media with CSS.

We look at the title page next. This will follow a similar pattern to other parts of the book in that we first tell it what @page we want to use and then, if needed, add any custom settings. For the title page the custom settings are:

- The h1 element with the `bookTitle` class 200% larger
- Center everything
- The h2 element or elements with the class `author` 150% of the default size and italicized.

```css
/* Title Page*/
section[data-type="titlepage"] {
  page: titlepage
}

section[data-type="titlepage"] * {
   text-align: center
}
h1.bookTitle {
  font-size: 200%;
}

h2.author {
  font-size: 150%;
  font-style: italic;
}
```

The copyright section just asigns the `copyright` page to the section elements with the `data-type="copyright"` attribute. We could also use an `id` attribute since we only expect to have one copyright page per publication but I like consistency and this will make the markup less error prone.

```css
section[data-type="copyright"] {
  page: copyright
}
```

The dedication section customizes the text inside by centering it and italicizing it.

```
section[data-type="dedication"] {
  page: dedication
}
section[data-type="dedication"] * {
  font-style: italic;
  text-align: center
}
```

The table of contents is associate with the `toc` page.

We make the TOC from a `section` element with a `data-type="toc"` attribute with an ordered list of `li` elements as the children. We remove the numbers for the list elements.

It also uses leaders to associate the TOC with page numbers.

The `leader` function defines a literal string, which expands to fill the available space on the line like justified text, by repeating the string as many times as necessary making it appear as if the text was left aligned and the page number was right aligned on the same line.

```
section[data-type="toc"] {
  page: toc
}

section[data-type="toc"] ol {
  list-style-type: none
}

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

The following sections associate the different page structures (defined with `data-type` attributes) with the corresponding @page rules.

We also make sure that we always have a page break before any new section.

```
/* Foreword  */
section[data-type="foreword"] {
  page: foreword;
  page-break-before: always;
}

/* Preface*/
section[data-type="preface"] {
  page: preface;
  page-break-before: always;
}

/* Part */
div[data-type="part"] {
  page: part;
  page-break-before: always;
}

/* Chapter */
section[data-type="chapter"] {
  page: chapter;
  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;
  page-break-before: always;
}

/* Index */
section[data-type="index"] {
  page: index;
  page-break-before: always;
}

/* Colophon */
section[data-type="colophon"] {
  page: colophon;
  page-break-before: always;
}
```

We now customize elements within the page. We’re not including any font-related styles. we might do that on a later iteration of the stylesheet.

All headings are aligned left and will not be hyphenated.

```
/*  Block Elements*/

h1, h2, h3, h4, h5, h6 {
  hyphens: none;
  text-align: left;
}
```

We’re not using Prism in this example so we just make the code blocks use the default monospace font. We could take some of the CSS from Prism to make the code blocks and inline code blocks look better.

```
code {
  font-family: monospace
}
```

The orphans property constrols the minimum number of lines in a block container that must be shown at the bottom of a page, region, or column.

In typography, an orphan is the first line of a paragraph that appears alone at the bottom of a page. (The paragraph continues on a following page).

The widows property sets the minimum number of lines in a block container that must be shown at the top of a page, region, or column.

In typography, a widow is the last line of a paragraph that appears alone at the top of a page. (The paragraph is continued from a prior page).

Both properties take a single integer value to indicate the number of columns to measure.

```
p {
  orphans:4;
  widows:2;
}
```

Next we tackle running headers. For all elements with `rh` class, we use `position: running(heading)` and style the content of the element.

This rule will remove the matching element from the normal document flow, to be inserted in a page margin region. See the PrinceXML documentation for Taking elements from the document for more details.

```
p.rh {
  position: running(heading);
  text-align: center;
  font-style: italic;

}
```

If an element has the property float: footnote then it will be floated into the footnote area of the page and a reference will be placed in the text.

The example stylesheet uses the `footnote` class to represent footnotes.

```
.footnote {
  float: footnote;
}
```

The `::footnote-marker` pseudo-element will style foot note markers. Footnote markers are the numbers used in front of the footnote text. They are similar to list item markers.

The example uses the `::after` pseudo element to add a period and a space after the marker.

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

::footnote-marker::after {
  font-weight: bold;
  content: '. ';
}
'. '
```

Prince will generate footnote calls using the ::footnote-call pseudo-element. Footnote calls are the numeric anchors in the text that refer to the footnotes.

The example CSS styles will display the current value of the footnote counter in square brackets and a superscript position in a slightly smaller font than the main text.

```
*::footnote-call {
  content: "[" counter(footnote) "]";
  font-size: inherit;
  vertical-align: inherit;
}
```

This code will be used to generate crossreferences to the specified link destination. This requires having links with href attributes. The rule will take the location of the link in the URL and use it to generate a page number to use in the cross reference generated text.

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

The final section is PDF-specific and deals with bookmarks. These are the links that appear on the bookmarks pannel in Acrobat products that will help you navigate to different places of the document.

There are three parts to a PDF bookmark:

- The bookmark level (in our case an integer from 1 to 6)
    - Lower numbered bookmark levels can contain higher levels bookmarks
- The bookmark state (open or close)
    - Hides (if closed) or shows (if open) any child bookmarks
- The bookmark label
    - What the PDF reader will display when showing the bookmark

Yes, there is a lot of repetition in the code. Prince and AntennaHouse have their own vendor prefixes for bookmarks so, in order to remain as compatible as possible, we use the vendor prefixed versions and an unprefixed. The hope is that all vendors will eventually move to the unprefixed version.

```
section[data-type="chapter"]  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();
  bookmark-level: 1;
  bookmark-state: open;
  bookmark-label: content();
}
```

```
section[data-type="chapter"]  h2 {
  -ah-bookmark-level: 2;
  -ah-bookmark-state: closed;
  -ah-bookmark-label: content();
  prince-bookmark-level: 2;
  prince-bookmark-state: closed;
  prince-bookmark-label: content();
  bookmark-level: 2;
  bookmark-state: closed;
  bookmark-label: content();
}

section[data-type="chapter"]  h3 {
  -ah-bookmark-level: 3;
  -ah-bookmark-state: closed;
  -ah-bookmark-label: content();
  prince-bookmark-level: 3;
  prince-bookmark-state: closed;
  prince-bookmark-label: content();
  bookmark-level: 3;
  bookmark-state: closed;
  bookmark-label: content();
}

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

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

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

The HTML

Rather than disecting the HTML, I’ve linked to the Github repository associated with this post:

> peter-pan-clean.html

Testing with multiple processors

Below are notes from running the CSS and HTML files against different processors, PrinceXML as a baseline and the new processors as comparison.

Baseline: PrinceXML

Because the stylesheet was originally developed to test PrinceXML and AntennaHouse formatters, the first thing to do is to make sure that the current version of PrinceXML is installed and works as intended.

```
prince -s paged-media.css -o peterpan.pdf peter-pan-clean.html
```

Weasyprint

Weasyprint bills itself as an open source alternative to PrinceXML and AntennaHouse but it appears to be lacking on some features. It works with the basic test document and CSS stylesheet but it appears to do some kind of black box magic to add bookmarks to the document.

There is no support for footnotes. The issue is known and documented in issue 296 and issue #298 on their issue tracker.

For testing I created a Homebrew formula for Weasyprint and added it to a custom tap I use for research and development.

Typeset .sh

typeset.sh is another paid service written in PHP, unlike PrinceXML, which as far as I understand, requires a single license payment, Typeset requires an anual subscription that can be for one year or permament.

The product has a very restrictive free tier that ties in with very limited use of their API. If I understood correctly, If you want to use command line tools then you must purchase either the anual (€89 per year) or perpetual (€449, one time) license. This is similar to what PrinceXML charges but you can download and test Prince before comitting to license and future maintenance costs.

In looking at the documentation all the CSS is inline of the file we're converting to, if we're converting many files with the same stylesheet, it adds unnecessary code duplication.

PDFReactor

PDFreactor seems like an interesting tool but, since it's not open source, I haven't been able to tell if it'll work with the current CSS and HTML to create PDF like PrinceXML and AntennaHouse do.

The first issue is that they require a trial license for testing and a full license for production use.

It is also a Java appliaction so it assumes that you have a JDK or JRE installed on your system. The command is:

```
java -jar path/to/pdfreactor.jar \
-i example.html \
-o example.pdf
```

This should work but, for some reason, it fails to find the jar file. The application doesn't set a CLASSPATH to follow and expects you to run the jar file from the Application directory in macOS or to specify the full path to the jar file if running it from somewhere else in the file system.

The PDF generating command assumes that the CSS is linked to the HTML file. It leads to a lot of code duplication.

Conclusion

Because of the choices commercial products make, they are not suited my use case. Even though it doesn’t meet all my needs (and there is no timeline for when the footnote support will be added) Weasyprint is my second option behind PrinceXML.

In a future post I’ll look at more complex stylesheets, including custom fonts, figures, tables, flexbox, Grid layouts (where supported) and other features.