XML workflow documentation

Carlos Araya

Thank You

Thanks to Laura Brady for getting this particular idea started :)

Background: HTML is the final format

In researching the technologies and tools that I use when developing digital content I've come across multiple discussions about what's the best way to create HTML for X application (ebooks, web, transforming into other formats and any number of ideas. Some people think that HTML is perfect for everyone to write, regardless of experience and comfort with the technology. We forget that HTML now is very different to HTML as it was originally created.

> *HTML —which is short for HyperText Markup Language— is the official language of the World Wide Web and was first conceived in 1990. HTML is a product of SGML (Standard Generalized Markup Language) which is a complex, technical specification describing markup languages, especially those used in electronic document exchange, document management, and document publishing. HTML was originally created to allow those who were not specialized in SGML to publish and exchange scientific and other technical documents. HTML especially facilitated this exchange by incorporating the ability to link documents electronically using hyperlinks.*
>
> *From: http://www.ironspider.ca/webdesign101/htmlhistory.htm*

The biggest issue, in my opinion, is that HTML has become a lot more complicated than the initial design. Creating HTML content (particularly when used in conjunction with CSS frameworks like Bootstrap or Zurb or with applications that use additional semantic elements like ePub) takes a lot more than just knowing markup to code them correctly. It takes knowledge of the document structure, the semantics needed for the content or the applications we are creating and the restrictions and schemas that we need to use so that the content will pass validation.

This article presents 4 different approaches to creating HTML. Two of them use HTML directly but target it as the final output for transformations and templating engines; the other two use markup like HTML without requiring strict HTML conformance. I've made these selections for two reasons:

- People who are not profesionals should not have to learn all the details of creating an ePub3 table of content or know the classes to add to elements to create a Bootsrap or Foundation layout grid
- It makes it easier for developers and designers to build the layout for the content without having to worry about the content itself; we can play with layout and content organization in parallel with content creation and, if we need to make any further changes, we just run our compilation process again

Markdown

Perhaps the simplest solution when moving content from text to HTML is Markdown.

Markdownis a text to (X)HTML conversion tool designed for writers. It refers both to the syntax used in the Markdown text files and the applications used to perform the conversion.

Markdown language was created in 2004 by John Gruber with the goal of allowing people "to write using an easy-to-read, easy-to-write plain text format, and optionally convert it to structurally valid XHTML (or HTML)" (http://daringfireball.net/projects/markdown/)

The language was designed to be readable as-is, without all the additional tags and attributes that makes it possible to covert markdown to languages like SGML, XML and HTML. Markdown is a formatting syntax for text that can be read by humans and can be easily converted to HTML.

The original implementation of Markdown is markdown.pland has been implemented in several other languages as applications (Ruby Gems, NodeJS modules and Python packages). All versions of Markdown are distributed under open source licenses and are included or available as a plugin for, several content-management systems and text editors.

Sites such as GitHub, Reddit, Diaspora, Stack Overflow, OpenStreetMap, and SourceForge use variants of Markdown to facilitate content creation and discussion between users.

The biggest weakness of Markdown is the lack of a unified standard. The original Markdown language hasn't been really supported since it was released in 2004 and all new version of Markdown, both parser and language specification have introduced not wholy compatible changes to Markdown. The lack of standard is also Markdown's biggest strength. It means you can, like Github did, implement your own extensions to the Markdown syntax to acommodate your needs.

Markdown is not easy to learn but once your fingers get used to the way we type the different elements it becomes much easier to work with as it is nothing more than inserting specific characters in a specific order to obtain the desired effect. Once you train yourself, it is also easy to read without having to convert it to HTML or any other language.

Most modern text editors have support for Markdown either as part of the default installation or through plugins.

Example Markdown document

- Markdown example form daringfireball

Asciidoctor

I only discovered Asciidoctor recently, while researching O'Reilly Media's publishing tool-chains. It caught my attention because of it's structure, the expresiveness of the markup without being HTML like HTMLbook and the extensibility of the templating system that it uses behind the scenes.

Asciidoctor has both a command line interface (CLI) and an API. The CLI is a drop-in replacement for the *asciidoc* command from the Standard python distribution. This means that you have a command line tool *asciidoctor* that will allow you to convert your marked documents without having to resort to a full blown application.

Syntax-wise, Asciidoctor is progressively more complex as you implement more advanced features. In the first example below no tables are used, for example. Tables are used in the second and thirs examples both as data tables and for layout.

The http://asciidoctor.org/docs/ provides more detailed instructions for the desired markup.

Example Asciidoc documents

- Asciidoctor planning document
- Comparison of Asciidoctor and AsciiDoc Features
- Applying Custom Themes

HTMLBook

O'Reilly Media has developed several new tools to get content from authors to readers. Atlas is their authoring tool, a web based application that allows you to create content they developed HTMLbook, a subset of HTML geared towards authoring and multi format publishing.

Given O'Reilly's history and association with open source publishing tools (they were an early adopter and promoter of Docbook and still use it for some of their publications) I found HTMLbook intriguing but not something to look at right away, as with many things you leave for later it fell off my radar.

It wasn't until I saw Sanders Kleinfeld's (O'Reilly Media Director of Publishing Technologies) presentation at IDPF Book World conference that I decided to take a second look at HTMLbook and its ecosystem.

Conceptually HTMLbook is very simple; it combines a subset of HTML5, the semantic structure of ePub documents and other IDPF specifications to create a flavor of HTML 5 that is designed specifically for publishing. There are also stylesheets that will allow you to convert Markdown and other text formats into HTMLbook (see Markdown to HTMLBook and AsciiDoc to HTMLBook (via AsciiDoctor))

If you use Atlas (O'Reilly's authoring and publishing platform) you don't have to worry about markup as the content is created visually. The challenges begin when implementing this vocabulary outside the Atlas environment.

The project comes with a set of stylesheets to convert HTMLbook content to ePub, MOBI and PDF. The intriguing thing about the stylesheets is that they use CSS Paged Media stylesheets in conjunction with third party tools such as AntennaHouse or PrinceXML.

The open source solutions offer permisive licenses that allow modification and integration into other products without requiring you to release your project under the same license like GPL and LGPL.

As with any solution that advocates creating HTML directly I have my reservations. In HTML formating in general and specialized formats like HTMLbooks in particular, the learning curve may be too steep for independent authors to use for creating content.

The user must learn not only the required HTML5 syntax but also the details regarding ePub semantic structure attributes and the other standards needed to create ePub books. While I understand that technologies such as this are not meant for independent authors or for poeple who are not comfortable or familiar with HTML but the learning curve may still be too steep for most users.

Example HTMLbook document

- Alice Adventures in Wonderland marked as HTMLbook

XML / XSLT

Perhaps the oldest solutions in the book to create HTML without actually creating HTML are XML-based. Docbook, TEI and DITA all have stylesheets that will take the XML content and convert it to HTML, PDF, ePub and other more esoteric formats.

In addition to stylesheets already available developers can create their own to adress specific needs.

Furthermore, tools like OxygenXML Author (and I would assume other tools in the same category have a visual mode that allow users to write XML content, validated against a schema in a way that is more familiar to people not used to creating content with raw XML tools.

The issues with xml are similar to those involved in creating HTML. The markup vocabulary requires brackets, attributes have to be enclosed in quotation marks and generall the syntax is as complicated as you make it. However, tools like Oxygen and smilar help alleviate this problem but don't resolve it completely.

The screenshot below shows OxygenXML Author working in a Docbook 5 document using visual mode.

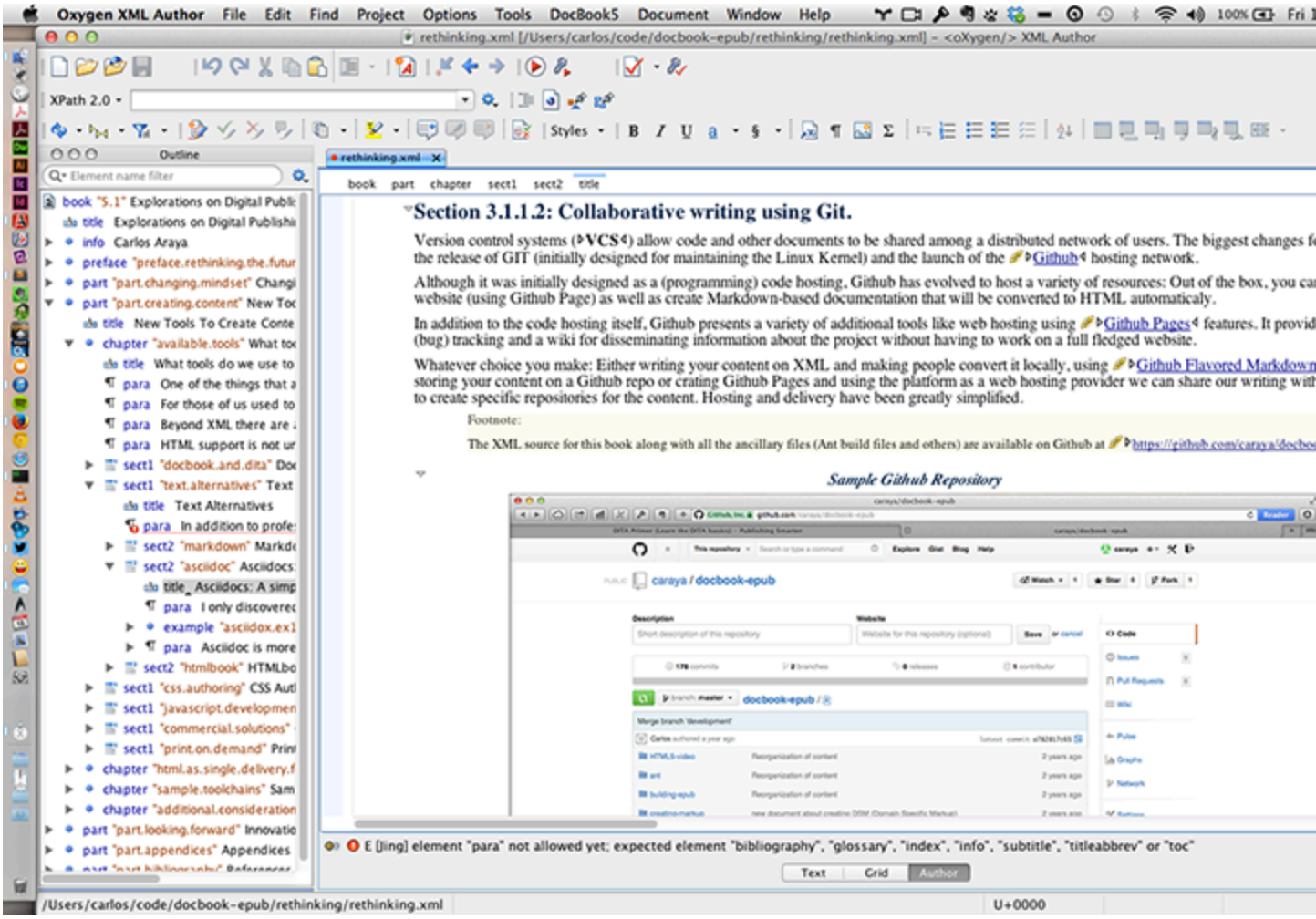

Figure 1: OxygenXML Visual Editor for XML

The positive side is that using XSLT there is no limit to what we can do with our XML content.

XML examples

- Sample Chapter marked as Docbook
- Tales by Edgar Allan Poe, marked as TEI
- Chapter from 20000 Leagues Under the Sea, marked as DITA

Conclusion

After exploring a selection of HTML conversion options the question becomes *which one is best?*

> The answer is *it depends*.

The best way to see how can these text-based tools can be incorporated is to ask yourself how much work you want to do in the backend versus how much work do you want you authors to do when creating the content. This is where the value of specialists in digital formats and publishing becomes essential, we can work with clients in providing the best solution to meet their needs.

Keep in mind who your audience and what the target vocabulary you're working towards, it will dictate what your best strategy is. Are these all the solutions; definitely not. Other solutions may appear that fit your needs better than those presented here; I would love to hear if that is the case.

Striking the balance between author and publisher is a delicate one. I tend to fall on the side of making things easier for authors... The tools can be made to translate basic markup into the desired result with minimal requirements for authors to mark up the content; the same can't necessarily be said about the publisher-first strategy

Introduction

One of the biggest limitations of markup languages, in my opinion, is how confining they are. Even large vocabularies like Docbook are limited in what they can do out of the box. HTML4 is non-extensible and HTML5 is limited in how you can extend it (web components are the only way to extend HTML5 I'm aware of that doesn't require an update to the HTML specification.)

By creating our own markup vocabulary we can be as expressive as we need to be without adding additional complexity for writers and users and without adding unecessary complexity for the developers building the tools to interact with the markup.

Why create our own markup

I have a few answers to that question:

In creating your own xml-based markup you enforce separation of content and style. The XML document provides the basic content of the document and the hints to use elsewhere. XSLT stylesheets allow you to structure the base document and associated hints into any number of formats (for the purposes of this document we'll concentrate on XHTML, PDF created through Paged Media CSS and PDF created using XSL formatting Objects)

Creating a domain specific markup vocabulary allows you think about structure and complexity for yourself as the editor/typesetter and for your authors. It makes you think about elements and attributes and which one is better for the given experience you want and what, if any, restrictions you want to impose on your makeup.

By creating our own vocabulary we make it easier for authors to write clean and simple content. XML provides a host of validation tools to enforce the structure and format of the XML document.

Options for defining the markup

For the purpose of this project we'll define a set of resources that work with a book structure like the one below:

```
<?xml version="1.0" encoding="UTF-8"?>
<book
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="xsd/book-schema-draft.xsd">
<metadata>
<title>Sample Document</title>
<authors>
<author>
```

```
<first-name>Carlos</first-name>
<surname>Araya</surname>
</author>
</authors>
</metadata>
<section type="chapter">
<title>Chapter 1</title>

<para></para>
<para></para>
<para></para>
</section>

<section type="chapter">
<title>Chapter 2</title>

<para></para>
<para></para>
<para></para>
</section>
</book>
```

It is not a complete structure. We will continue adding elements afte we reach the MVP (Minimum Viable Product) stage. As usual, feedback is always appreciated.

XML Schema

The schema is defined from most general to most specific elements. We'll follow the same process to explain what the schema does and how we arrived to the choices we made.

At the beginning of the schema we define some custom types that will be used throughout the document.

The first one, *string255* is a string that is limited to 255 characters in length. We do this to prevent overtly long strings.

The second one, *isbn* is a regular expression to match 10 digits ISBN numbers. We'll have to modify it to handle ISBN-13 as well as 10.

The third custom type is an enumeration of all possible values for the *align* attribute acording to CSS and HTML. Rather than manually type each of these we will reference this enumeration and include all its values for "free".

We also allow the optional use of *class* and *id* attributes for the book by assigning `genericPropertiesGroup` attribute group as attributes to the group. We'll see this assigned to other elements so I decided to make it reusable rather than have to duplicate the attributes in every element I want to use them in.

```
<!-- Simple types to use in the content -->
<xs:simpleType name="token255">
<xs:restriction base="xs:token">
<xs:maxLength value="255"/>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="isbn">
<xs:restriction base="xs:unsignedLong">
<xs:totalDigits value="10"/>
<xs:pattern value="d{10}"/>
</xs:restriction>
</xs:simpleType>

<xs:simpleType name="align">
<xs:restriction base="xs:token">
<xs:enumeration value="left"/>
<xs:enumeration value="center"/>
<xs:enumeration value="right"/>
<xs:enumeration value="justify"/>
</xs:restriction>
</xs:simpleType>

<xs:attributeGroup name="genericPropertiesGroup">
```

```
<xs:attribute name="id" type="xs:ID" use="optional"/>
<xs:attribute name="class" type="xs:token" use="optional"/>
</xs:attributeGroup>
```

The next stage is to define elements to create our *people* types. We create a base person element and then create three role elements based on person. We will use this next to define groups for each role.

```
<!-- complex types to create groups of similar
person items -->
<xs:complexType name="person">
<xs:sequence>
<xs:element name="first-name" type="xs:token"/>
<xs:element name="surname" type="xs:token"/>
</xs:sequence>
<xs:attribute name="id" type="xs:ID" use="optional"/>
</xs:complexType>

<xs:complexType name="author">
<xs:choice minOccurs="1" maxOccurs="unbounded">
<xs:element name="author" type="person"/>
</xs:choice>
</xs:complexType>

<xs:complexType name="editor">
<xs:complexContent>
<xs:extension base="person">
<xs:sequence>
<xs:element name="type" type="xs:string"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:complexType name="otherRole">
<xs:complexContent>
<xs:extension base="person">
<xs:sequence minOccurs="1" maxOccurs="1">
<xs:element name="role" type="xs:string"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
```

Two of the derived types add attributes or elements to the base person element to make the generic person more appropriate to their role rather than repeat the content of person each time that an author, editor or other role appear.

Author is the most straight forward and only wraps person in the author element.

Editor takes the base person element and adds a `type` child to indicate the type of editor (some that come to mind are acquisition, production and managing.) The editor elements looks like this:

```
<editor>
<first-name>Carlos</first-name>
<surname>Araya</surname>
<type>Managing</type>
</editor>
```

OtherRoles takes all other roles that are not author or editor and adds a role element to specify what role they play, for example: Illustrator, Indexer, Research Assistant, among others. The element looks like this:

```
<otherRole>
<first-name>Sherlock</first-name>
<surname>Holmes</surname>
<role>Researcher</role>
</otherRole>
```

Next we create wrappers for each group as *authors*, *editors* and *otherRoles* so we can provide easier styling with XSLT and CSS later on.

```
<xs:complexType name="authors">
<xs:annotation>
<xs:documentation>Wrapper to get more than one
author</xs:documentation>
</xs:annotation>
<xs:sequence>
<xs:element name="author" type="person"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="editors">
<xs:annotation>
<xs:documentation>
extension to person to indicate editor and his/her role
</xs:documentation>
</xs:annotation>
<xs:complexContent>
<xs:extension base="person">
<xs:sequence>
<xs:element name="type" type="xs:token"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:complexType name="otherRoles">
```

```
<xs:annotation>
<xs:documentation>
extension to person to accomodate roles other than author and
editor
</xs:documentation>
</xs:annotation>
<xs:complexContent>
<xs:extension base="person">
<xs:sequence>
<xs:element name="role" type="xs:token"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
```

We now look at the elements that we can put inside a section. Some of these elements are overtly complex and deliberately so since they have to acommodate a lot of possible parameters.

We'll look at links first as it is the simplest of our content structures. We borrow the *href* attribute from HTML to indicate the destination for the link and make it required.

We also incoporate a *label* so we can later build the link and for accessibility purposes. It also uses our `genericPropertiesGroup` attribute set to add class and ID as attributes for our links.

```
<xs:element name="link">
<xs:complexType>
<xs:attributeGroup ref="genericPropertiesGroup" />
<xs:attribute name="href" type="xs:string" use="required"/>
<xs:attribute name="label" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
```

The link in our resulting book will look like this:

```
<link href="http://google.com" label="link to google"/>
```

and with the optional attributes it will look like this

```
<link class="external" id="ex01" href="http://google.com"
label="link to google">
```

Once I had the links I figured I need a way to create anchors for internal links that look like this: *<a href="#top">* and expect the target to be formated like this *<a id="top">*. To accomodate this I created an anchor element to provide the destination for internal links.

```
<!-- Named Anchor -->
<xs:element name="anchor">
<xs:complexType>
<xs:attribute name="id"/>
</xs:complexType>
</xs:element>
```

As I was working on further ideas for the project I realized that we forgot to create inline and block level containers for the content, important if you're going to style smaller portions of content within a paragraph or witin a section. Taking the names from HTML we define *section* (inline) and *div* (block) elements. Div is a secondary section, containing the same model as the section, including additional div containers.

```
<xs:element name="div">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="code"/>
<xs:element ref="para" minOccurs="1" maxOccurs="unbounded"/>
<xs:element ref="ulist"/>
<xs:element ref="olist"/>
<xs:element ref="figure"/>
<xs:element ref="image"/>
<xs:element ref="div"/>
<xs:element ref="span"/>
<xs:element ref="blockquote"/>
<xs:element ref="h1"/>
<xs:element ref="h2"/>
<xs:element ref="h3"/>
<xs:element ref="h4"/>
<xs:element ref="h5"/>
<xs:element ref="h6"/>
</xs:choice>
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="type"type="xs:token" use="optional"/>
</xs:complexType>
</xs:element>
```

Span is an inline element, therefore the model is greatly reduced to only the elements that can be inside a paragraph

Type is used in these two elements and in our sections to create data-type and epub:type attributes. These are used in the Paged Media stylesheet to decide how will the content be formated.

```
<xs:element name="span">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="strong"/>
<xs:element ref="emphasis"/>
```

```
<xs:element ref="underline"/>
<xs:element ref="strike"/>
<xs:element ref="link"/>
<xs:element ref="span"/>
<xs:element ref="quote"/>
</xs:choice>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

Next are images and figures where we borrow from HTML, again, for the name of attribute names and their functionality. We define 3 elements for the image-related tags: *figure*, *figcaption* and the *image* itself.

Figure is the wrapper around a `figcaption` caption and the `image` element itself. The `figcaption` is a text-only element that will contain the caption for the associated image

```
<!-- Figure and related elements -->
<xs:element name="figure">
<xs:complexType mixed="true">
<xs:all>
<xs:element ref="image"/>
<xs:element ref="figcaption"/>
</xs:all>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

The caption child only uses text and, because it's only used as a child of figure, we don't need to assign attributes to it. It will inherit from the image or the surrounding figure.

```
<xs:element name="figcaption">
<xs:annotation>
<xs:documentation>
caption for the image in the figure. Because it's
only used as a child of figure, we don't need to
assign attributes to it
</xs:documentation>
</xs:annotation>
</xs:element>
```

When working with the image element we start with `genericPropertiesGroup` to define *class* and *id*.

Then we require a *src* attribute to tell where the image is located. We need to be careful because we haven't told the schema the different types of images. We have at least three different locations for the image files. All three of these are valid locations for our image.png file.

```
image.png
directory/image.png
http://mysite.org/images/image.png
```

We could create branches of our schema to deal with the different locations but I've chosen to let the XSLT style sheets deal with this particular situation. The schema type for the image (*xs:anyURI*) should also help to sort out the issue.

width and *height* are expressed as integer and are left as optional to account for the possibility that the CSS or XSLT stylesheets modify the image dimensions. Making these dimensions mandatory may affect how the element interact with the styles later on.

The *alt* attribute indicates alternative text for the image. It is not meant as a full description so we've constrained it to 255 characters.

align uses our align enumeration to indicae the image's alignment. It is not essential to the XML but will be useful to the XSLT stylesheets we'll create later as part of the process.

```
<xs:element name="image">
<xs:complexType>
<xs:attributeGroup ref="genericPropertiesGroup" />
<xs:attribute name="src" type="xs:string" use="required"/>
<xs:attribute name="height" type="xs:integer" use="optional"/>
<xs:attribute name="width" type="xs:integer" use="optional"/>
<xs:attribute name="alt" type="string255" use="required"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>
```

In order to acommodate the four basic styles available to our documents: *strong*, *emphasis*, *strike* and *underline* and their nesting we had to do some juryriging of the elements to tell the schema what children are allowed for each element. The schema look like this:

```
<xs:element name="strong">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="emphasis"/>
<xs:element ref="underline"/>
<xs:element ref="strike"/>
</xs:choice>
</xs:complexType>
</xs:element>

<xs:element name="emphasis">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="strong"/>
<xs:element ref="emphasis"/>
```

```
<xs:element ref="underline"/>
<xs:element ref="strike"/>
</xs:choice>
</xs:complexType>
</xs:element>

<xs:element name="underline">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="strong"/>
<xs:element ref="emphasis"/>
<xs:element ref="strike"/>
</xs:choice>
</xs:complexType>
</xs:element>

<xs:element name="strike">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="strong"/>
<xs:element ref="emphasis"/>
<xs:element ref="underline"/>
</xs:choice>
</xs:complexType>
</xs:element>
```

The *emphasis* element is the only one that allows the same element to be nested. When nesting emphasis elements they cancel each other

When I first conceptualized the project I envisioned one element for both numbered and bulleted lists. That proved to difficult to implement and to cumbersome to write so I reverted to having to sepratate lists, one for ordered or numbered lists (*olist*) and one for unordered or bulleted lists (*ulist*). The only difference is the type of list that we use in XSLT later on.

The list elements also require at least 1 *item* child. If it's going to be left empty why bother having the list to begin with.

They inherit class and ID from *genreicPropertiesGroup*.

```
<!-- Lists -->
<xs:element name="ulist">
<xs:complexType mixed="true">
<xs:sequence minOccurs="1" maxOccurs="unbounded">
<xs:element ref="item"/>
</xs:sequence>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

```
<xs:element name="olist">
<xs:complexType mixed="true">
<xs:sequence minOccurs="1" maxOccurs="unbounded">
<xs:element ref="item"/>
</xs:sequence>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>

<xs:element name="item">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

The *code* element wraps code and works as higlighted, fenced code blocks (think Github Flavored Markdown.)

When using CSS we'll generate a <code><pre></pre></code> block with a language attribute that will be formated with Highlight.js (the chosen package will be a part of the project tool chain)

Because of the intended use, the `language` attribute is required.

Class and ID (from *genericPropertiesGroup*) are optional

```
<xs:element name="code">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="language" use="required"/>
</xs:complexType>
</xs:element>
```

Another type of element that came up when working on the documentation were aside, blockquotes and quotes. `Blockquote` and `attribution` are for longer block level quotations (more than 4 lines of text) while `quote` is for shorter quotations usually inserted in a paragraph.

```
<xs:element name="blockquote">
<xs:complexType mixed="true">
<xs:sequence minOccurs="1" maxOccurs="unbounded">
<xs:element ref="attribution"/>
<xs:element ref="para"/>
</xs:sequence>
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>
```

```
<xs:element name="attribution">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="para"/>
</xs:choice>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>

<xs:element name="quote">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

Paragraphs (*para* in our documents) are the essential unit of content for our books. The paragraph is where most content will happen, text, styles and additional elements that we may add as we go along (inline code comes to mind).

We include 3 different groups of properties in the paragraph declaration: Styles (*strong*, *emphasis*, *underline* and *strike* to do bold, italics, underline (outside links) and strikethrough text); Organization (*span* and *link*) and our *genericPropertiesGroup* (class and id).

This model barely begins to scratch the surface of what we can do with our paragraph model. I decided to go for simplicity rather than completeness. This will definitely change in future versions of the schema.

```
<!-- Paragraphs -->
<xs:element name="para">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="strong"/>
<xs:element ref="emphasis"/>
<xs:element ref="underline"/>
<xs:element ref="strike"/>
<xs:element ref="link"/>
<xs:element ref="span"/>
<xs:element ref="quote"/>
</xs:choice>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

Like HTML we've chose to create 6 levels of headings although, to be honest, I can't see the need for more than 4.

We give all links three attributes: *class*, *id* and *align* to hint stylesheets where we want to place the heading (left, right, center)

```
<!-- Headings -->
<xs:element name="h1">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>

<xs:element name="h2">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>

<xs:element name="h3">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>

<xs:element name="h4">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>

<xs:element name="h5">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>

<xs:element name="h6">
<xs:complexType mixed="true">
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>
```

The metadata section tells us more about the book itself and can be used to build a *package.opf* manifest using XSLT as part of our transformation process. We include basic information such as *isbn* (validated as an ISBN type defined earlier in the schema), an *edition* (integer indicating what edition of the book it is) and *title*.

```
<!-- Metadata element -->
<xs:element name="metadata">
<xs:complexType>
<xs:sequence>
<xs:element name="isbn" type="isbn"/>
<xs:element name="edition" type="xs:integer"/>
<xs:element name="title" type="token255"/>
<xs:element name="authors" type="authors" minOccurs="1"
maxOccurs="unbounded"/>
<xs:element name="editors" type="editors" minOccurs="0"
maxOccurs="unbounded"/>
<xs:element name="otherRoles" type="otherRoles" minOccurs="0"
maxOccurs="unbounded"/>
<xs:element ref="para"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

Section is our primary container for paragraphs and associated content. Some of the items exclusive to sections are:

The *title* element is required to appear exactly one time.

We can have 1 or more *para* elements.

We can use 0 or more of the following elements:

- *code* fenced code blocks elements
- *ulist* unordered list
- *olist* ordered (numbered) lists
- *figure* for captioned images
- *image* without captions
- *div* block level containers
- *span* inline level container

The element inherits *class* and *ID* from genericPropertiesGroup.

Finally we add the `type` to create data-type and/or epub:type attributes. I chose to make it option and default it to chapter. We want to make it easier for authors to create content; where possible. I'd rather have the wrong value than no value at all.

```
<!-- Section element -->
<xs:element name="section">
<xs:complexType mixed="true">
<xs:sequence>
```

```
<xs:element name="title" type="xs:token" minOccurs="1"
maxOccurs="1"/>
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="code"/>
<xs:element ref="para" minOccurs="1 maxOccurs="unbounded"/>
<xs:element ref="ulist"/>
<xs:element ref="olist"/>
<xs:element ref="figure"/>
<xs:element ref="image"/>
<xs:element ref="div"/>
<xs:element ref="span"/>
<xs:element ref="blockquote"/>
<xs:element ref="h1"/>
<xs:element ref="h2"/>
<xs:element ref="h3"/>
<xs:element ref="h4"/>
<xs:element ref="h5"/>
<xs:element ref="h6"/>
</xs:choice>
</xs:sequence>
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="type" type="xs:token" use="optional"
default="chapter"/>
</xs:complexType>
</xs:element>
```

Now that we have defined our elements, we'll define the core structure of the document by defining the structure of the `book` element.

After all the work we've done defining the content the definition of our book is almost anti-climatic. We define the `book` element as the sequence of exactly 1 *metadata* element and 1 or more *section* elements.

As with all our elements we add *class* and *ID* from our genericPropertiesGroup.

```
<xs:element name="book">
<xs:complexType mixed="true">
<xs:sequence>
<xs:element ref="metadata" minOccurs="0" maxOccurs="1"/>
<xs:element ref="section" minOccurs="1" maxOccurs="unbounded"/>
</xs:sequence>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

This covers the schema for our document type. It is not completed by any stretch of the imagination. It can be further customized to suit individual needs. The current version represents a very basic text heavy document type.

There are definitely more elements to add like video, audio and others both with equivalent elements in HTML and compound elements based on your needs.

From XML to HTML

One of the biggest advantages of working with XML is that we can convert the abstract tags into other markups. For the purposes of this project we'll convert the XML created to match the schema we just created to HTML and then use tools like PrinceXML or AntenaHouse we'll convert the HTML/CSS files to PDF

Why HTML

HTML is the default format for the web and for most web/html based content such as ePub and Kindle. As such it makes a perfect candidate to explore how to generate it programatically from a single source file.

HTML will also act as our source for using CSS paged media to create PDF cotnent.

Why PDF

Rather than having to deal with XSL-FO, another XML based vocabulary to create PDF content, we'll use XSLT to create another HTML file and process it with CSS Paged Media and the companion Generated Content for Paged Media specifications to create PDF content.

In this document we'll concentrate on the XSLT to HTML conversion and will defer the from HTML to PDF to a later article.

Creating our conversion stylesheets

To convert our XML into other formats we will use XSL Transformations (also known as XSLT) version 2 (a W3C standard) and version 3 (a W3C last call draft recommendation) where appropriate.

XSLT is a functional language designed to transform XML into other markup vocabularies. It defines template rules that match elements in your source document and processing them to convert them to the target vocabulary.

In the XSLT example below, we do the following:

1. Declare the file to be an XML document
2. Define the root element of the stylesheet (xsl:stylesheet)
3. Indicate the namespaces that we'll use in the document and, in this case, tell the processor to excluse the given namespaces

4. Strip whitespaces from all elements and preserve it in the code elements
5. Create the default output we'll use for the main document and all generated pages (discussed later)
6. Create a default template to warn us if we missed anything

```
<?xml version="1.0" ?> <!-- Define stylesheet
root and namespaces we'll work with --> <xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:epub="http://www.idpf.org/2007/opf"
exclude-result-prefixes="dc
epub" xml:lang="en-US" version="2.0"> <!-- Strip whitespace from
the
listed elements --> <xsl:strip-space elements="*"/> <!-- And
preserve it from the elements below --> <xsl:preserve-space
elements="code"/> <!-- Define the output for this and all
document
children --> <xsl:output name="xhtml-out" method="xhtml"
indent="yes"
encoding="UTF-8" omit-xml-declaration="yes" /> <!-- Default
template
taken from http://bit.ly/1sXqIL8 This will tell us of any
unmatched
elements rather than failing silently --> <xsl:template
match="*">
<xsl:message terminate="no"> WARNING: Unmatched element:
<xsl:value-of select="name()"/> </xsl:message>
<xsl:apply-templates/> </xsl:template> <!-- More content to be
added --> </xsl:stylesheet>
```

This is a lot of work before we start creating our XSLT content. But it's worth doing the work up front. We'll see what are the advantages of doint it this way as we move down the style sheet.

Now onto our root templates. The first one is the entry point to our document. It performs the following tasks:

1. Match the root element to create the skeleton for our HTML content
2. In the title we insert the content of the *metadata/title* element
3. In the body we 'apply' the templates that match the content inside our document (more on this later)

```
<!-- Root template, matching / --> <xsl:template
match="book"> <html> <head> <xsl:element name="title">
<xsl:value-of select="metadata/title"/> </xsl:element>
<xsl:element name="meta"> <xsl:attribute name="generator">
<xsl:value-of select="system-property('xsl:product-name')"/>
<xsl:value-of select="system-property('xsl:product-version')"/>
</xsl:attribute> </xsl:element> <link rel="stylesheet"
href="css/style.css" /> <xsl:if test="(code)"> <!-- Use
```

```
highlight.js
and docco style --> <link rel="stylesheet" href="css/styles/
docco.css"
/> <!-- Load highlight.js --> <script
src="lib/highlight.pack.js"></script> <script>
hljs.initHighlightingOnLoad(); </script> </xsl:if> <!-- Comment
this out for now. It'll become relevant when we add video <script
src="js/script.js"></script> --> </head> <body>
<xsl:apply-templates/> <xsl:apply-templates select="/"
mode="toc"/>
</body> </html> </xsl:template>
```

We could build the CSS stylesheet and Javascript files as part of our root template but we chose not to.

Working with the stylesheet as part of the XSLT stylesheet allows the XSLT stylesheet designer to embed the style and parameterize the stylesheet, thus making the stylesheet customizable from the command line.

For all advantages, this method ties the styles for the project to the XSLT stylesheet and requires the XSLT stylesheet designer to be involved in all CSS and Javascript updates.

By linking to external CSS and Javascript files we can leverage expertise independent of the Schema and XSLT stylesheets. Book designers can work on the CSS, UX and experience designers can work on Javascript and additional CSS areas, book designers can work on the Paged Media stylesheets and authors can just write.

Furthermore we can reuse our CSS and Javascript on multiple documents.

Table of contents

> *The table of content template is under active development and will be different depending on the desired output. I document it here as it is right now but will definitely change as it's further developed.*

There is a second template matching the root element of our document to create a table of content. At first thought this looks like the wrong approach

We leverage XSLT modes that allow us to create templates for the same element to perform different tasks. In `toc mode` we want the root template to do the following:

1. Create the section and nav and ol elements
2. Add the title for the table of contents
3. For each section element that is a child of root create these elements
    1. The `li` element
    2. The a element with the corresponding href element

3. The value of the href element (a concatenation of the section's type attribute, the position within the document and the .html string)
4. The title of the section as the 'clickable' portion of the link

```
<xsl:template match="/" mode="toc"> <section
data-type="toc"> (1) <nav class="toc"> (1) <h2>Table of
Contents</h2> <ol> <xsl:for-each select="book/section">
<xsl:element name="li"> (3.1) <xsl:element name="a"> (3.2)
<xsl:attribute name="href"> (3.2) <xsl:value-of
select="concat((@type), position(),'.html')"/> (3.3)
</xsl:attribute>
<xsl:value-of select="title"/> (3.4) </xsl:element>
</xsl:element> </xsl:for-each> </ol> </nav> </section>
</xsl:template>
```

Metadata and Section

With these templates in place we can now start writing the major areas of the document, *metadata* and *section*.

Metadata

The metadata is a container for all the elements inside. As such we just create the div that will hold the content and call `xsl:apply-templates` to process the children inside the metadata element using the apply-template XSLT instruction. The template looks like this

```
<xsl:template match="metadata"> <xsl:element
name="div"> <xsl:attribute name="class">metadata</xsl:attribute>
<xsl:apply-templates/> </xsl:element> </xsl:template>
```

Section

The section container on the other hand is a lot more complex because it has a lot of work to do. It is our primary unit for generating files, takes most of the same attributes as the root template and then processes the rest of the content.

Inside the template we first create a vairable to hold the name of the file we'll generate. The file name is a concatenation of the following elements:

- The type attribute
- The position in the document
- the string "html"

The result-document element takes two parameters: the value of the file name variable we just defined and the xhtml-out format we defined at the top of the document. The XHTML format may look like overkill right now but it makes sense when we consider moving the generated content to ePub or other fomats where strict XHTML conformance is a requirement.

We start generating the skeleton of the page, we add the default style sheet and do the first conditional test of the document. Don't want to add stylesheets to the page unless they are needed so we test if there is a code element on the page and only add highlight.js related stylesheets and scripts.

In the body element we see the first of many times we'll conditionally add attributes to the element. We use only add a data-type attribute to body if there is a type attribute in the source document. We do the same thing for id and class.

```
<xsl:template match="section"> <!-- Variable to
create section file names --> <xsl:variable name="fileName"
select="concat((@type), (position()-1),'.html')"/> <!-- An
example
result of the variable above would be introduction1.xhtml -->
<xsl:result-document href='{$fileName}' format="xhtml-out">
<html>
<head> <link rel="stylesheet" href="css/style.css" /> <xsl:if
test="(code)"> <!-- Use highlight.js and github style --> <link
rel="stylesheet" href="css/styles/docco.css" /> <!-- Load
highlight.js
--> <script src="lib/highlight.pack.js"></script> <script>
hljs.initHighlightingOnLoad(); </script> </xsl:if> <!-- Comment
this out for now. It'll become relevant when we add video <script
src="js/script.js"></script> --> </head> <body> <section>
<xsl:if test="@type"> <xsl:attribute name="data-type">
<xsl:value-of select="@type"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@class)"> <xsl:attribute name="class">
<xsl:value-of select="@class"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/> </xsl:attribute> </xsl:if> <xsl:apply-templates/>
</section> </body> </html> </xsl:result-document>
</xsl:template>
```

Metadata content

Publication information

```
<xsl:template match="isbn"> <p>ISBN:
<xsl:value-of select="."/></p> </xsl:template>
```

```
<xsl:template match="edition"> <p
class="no-margin-left">Edition: <xsl:value-of select="."/></p>
</xsl:template>
```

People groups

```
<xsl:template match="metadata/authors">
<h2>Authors</h2> <ul> <xsl:for-each select="author"> <li>
<xsl:value-of select="first-name"/> <xsl:text> </xsl:text>
<xsl:value-of select="surname"/> </li> </xsl:for-each> </ul>
</xsl:template> <xsl:template match="metadata/editors">
<h2>Editorial Team</h2> <ul class="no-bullet"> <xsl:for-each
select="editor"> <li> <xsl:value-of select="first-name"/>
<xsl:text> </xsl:text> <xsl:value-of select="surname"/>
<xsl:value-of select="concat(' - ', type, ' ',
'editor')"></xsl:value-of> </li> </xsl:for-each> </ul>
</xsl:template> <xsl:template match="metadata/otherRoles">
<h2>Production team</h2> <ul class="no-bullet"> <xsl:for-each
select="otherRole"> <li> <xsl:value-of select="first-name" />
<xsl:text> </xsl:text> <xsl:value-of select="surname" />
<xsl:text> - </xsl:text> <xsl:value-of select="role" /> </li>
</xsl:for-each> </ul> </xsl:template>
```

Titles and headings

Titles and headings use mostly the same code. We've put them in separate templates to make it possible and easier to generate different code for each heading. It's not the same as using CSS where you can declare rules for the same attribute multiple times (with the last one winning); when writing transformations you can only have one per element otherwise you will get an error.

The goal is to create as simple a markup as we can so we can better leverage CSS to style and make our content display as intended.

```
<xsl:template match="title "> <xsl:element
name="h1"> <xsl:if test="@align"> <xsl:attribute name="align">
<xsl:value-of select="@align"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@class)"> <xsl:attribute name="class">
<xsl:value-of select="@class"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/> </xsl:attribute> </xsl:if> <xsl:value-of
select="."/> </xsl:element> </xsl:template> <xsl:template
match="h1"> <xsl:element name="h1"> <xsl:if test="@align">
<xsl:attribute name="align"> <xsl:value-of select="@align"/>
```

```
</xsl:attribute> </xsl:if> <xsl:if test="(@class)">
<xsl:attribute name="class"> <xsl:value-of select="@class"/>
</xsl:attribute> </xsl:if> <xsl:if test="(@id)">
<xsl:attribute name="id"> <xsl:value-of select="@id"/>
</xsl:attribute> </xsl:if> <xsl:value-of select="."/>
</xsl:element> </xsl:template>
```

Blockquotes, quotes and asides

```
<xsl:template match="blockquote"> <xsl:element
name="blockquote"> <xsl:if test="(@class)"> <xsl:attribute
name="class"> <xsl:value-of select="@class"/> </xsl:attribute>
</xsl:if> <xsl:if test="(@id)"> <xsl:attribute name="id">
<xsl:value-of select="@id"/> </xsl:attribute> </xsl:if>
<xsl:apply-templates /> </xsl:element> </xsl:template> <!--
BLOCKQUOTE ATTRIBUTION--> <xsl:template match="attribution">
<xsl:element name="cite"> <xsl:if test="(@class)"> <xsl:attribute
name="class"> <xsl:value-of select="@class"/> </xsl:attribute>
</xsl:if> <xsl:if test="(@id)"> <xsl:attribute name="id">
<xsl:value-of select="@id"/> </xsl:attribute> </xsl:if>
<xsl:apply-templates/> </xsl:element> </xsl:template>
```

```
<xsl:template match="quote"> <q><xsl:value-of
select="."/></q> </xsl:template>
```

```
<xsl:template match="aside"> <aside> <xsl:if
test="type"> <xsl:attribute name="data-type"> <xsl:value-of
select="@align"/> </xsl:attribute> </xsl:if> <xsl:if
test="(@class)"> <xsl:attribute name="class"> <xsl:value-of
select="@class"/> </xsl:attribute> </xsl:if> <xsl:if
test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/>
</xsl:attribute> </xsl:if> <xsl:apply-templates/> </aside>
</xsl:template>
```

Div and Span

```
<xsl:template match="div"> <xsl:element
name="div"> <xsl:if test="@align"> <xsl:attribute name="align">
<xsl:value-of select="@align"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@class)"> <xsl:attribute name="class">
<xsl:value-of select="@class"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
```

```
select="@id"/> </xsl:attribute> </xsl:if> <xsl:apply-templates/>
</xsl:element> </xsl:template>
```

```
<xsl:template match="span"> <xsl:element
name="span"> <xsl:if test="@type"> <xsl:attribute
name="data-type">
<xsl:value-of select="@type"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@class)"> <xsl:attribute name="class">
<xsl:value-of select="@class"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/> </xsl:attribute> </xsl:if> <xsl:value-of
select="."/> </xsl:element> </xsl:template>
```

Paragraphs

```
<xsl:template match="para"> <xsl:element
name="p"> <xsl:if test="(@class)"> <xsl:attribute name="class">
<xsl:value-of select="@class"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/> </xsl:attribute> </xsl:if> <xsl:apply-templates/>
</xsl:element> </xsl:template>
```

Styles

```
<xsl:template match="strong">
<strong><xsl:apply-templates /></strong> </xsl:template>
<xsl:template match="emphasis">
<em><xsl:apply-templates/></em> </xsl:template>
<xsl:template match="strike">
<strike><xsl:apply-templates/></strike> </xsl:template>
<xsl:template match="underline"> <u><xsl:apply-templates/></u>
</xsl:template>
```

Links and anchors

One of the

```
<xsl:template match="link"> <xsl:element
name="a"> <xsl:if test="(@class)"> <xsl:attribute name="class">
<xsl:value-of select="@class"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/> </xsl:attribute> </xsl:if> <xsl:attribute
name="href"> <xsl:value-of select="@href"/> </xsl:attribute>
<xsl:attribute name="label"> <xsl:value-of select="@label"/>
</xsl:attribute> <xsl:value-of select="@label"/> </xsl:element>
</xsl:template>
```

When working with links there are times when we want to link to sections within the same document or to specific sections in another document. To do this we need anchors that will resolve to the following HTML:

```
<a name="target"><a>
```

The transformation element looks like this:

```
<xsl:template match="anchor"> <xsl:element
name="a"> <xsl:attribute name="name"> </xsl:attribute>
</xsl:element> </xsl:template>
```

Not sure if I want to make this an empty element or not

Empty element <anchor name="home"/> appeals to my ease of use paradigm but it may not be as easy to understand for peope who are not familiar with XML empty elements

Code blocks

Code elements create fenced code blocks like the ones from Github Flavored Markdown.

We use Adobe Source Code Pro font. It's a clean and readable font designed specifically for source code display.

We highlight our code with Highlight.js.

> *Note that the syntax higlighting only works for HTML. Although PrinceXML supports Highlight.js it is not working. I've asked on the Prince support forums and am waiting for an answer.*

```
<xsl:template match="code"> <xsl:element
name="pre"> <xsl:element name="code"> <xsl:attribute
name="class">
<xsl:value-of select="@language"/> </xsl:attribute> <xsl:value-of
select="."/> </xsl:element> </xsl:element> </xsl:template>
```

Lists and list items

When I first conceptualized this project I had designed a single list element and attributes to produce bulleted and numbered lists. This proved to difficult to implement so I went back to two separate elements: `ulist` for bulleted lists and `olist` for numbered lists.

Both elements share the *item* element to indicates the items inside the list. At least one item is required a list.

```
<xsl:template match="ulist"> <xsl:element
name="ul"> <xsl:if test="(@class)"> <xsl:attribute name="class">
<xsl:value-of select="@class"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/> </xsl:attribute> </xsl:if> <xsl:apply-templates/>
</xsl:element> </xsl:template> <xsl:template match="olist">
<xsl:element name="ol"> <xsl:if test="(@class)"> <xsl:attribute
name="class"> <xsl:value-of select="@class"/> </xsl:attribute>
</xsl:if> <xsl:if test="(@id)"> <xsl:attribute name="id">
<xsl:value-of select="@id"/> </xsl:attribute> </xsl:if>
<xsl:apply-templates/> </xsl:element> </xsl:template>
<xsl:template match="item"> <xsl:element name="li"> <xsl:if
test="(@class)"> <xsl:attribute name="class"> <xsl:value-of
select="@class"/> </xsl:attribute> </xsl:if> <xsl:if
test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/>
</xsl:attribute> </xsl:if> <xsl:value-of select="."/>
</xsl:element> </xsl:template>
```

Figures and Images

Figures, captions and the images inside present a few challenges. Because we allow authors to set height and width on both figure and the imageg inside we may find situations where the figure container is narrower than the image inside.

To avoid this issue we test whether the figure width value is smaller than the width of the image inside. If it is, we use the width of the image as the width of the figure, ootherwise we use the width of the image inside.

We didn't do the same thing for the height. It may be changed in a future iteration.

The data model for our content allows both figures and images to be used in the document. This is so we don't have to insert empty captions to figures just so we can add an image... If we don't want a caption we can insert the image directly on our document.

```
<xsl:template match="figure"> <xsl:element
name="figure"> <xsl:if test="(@class)"> <xsl:attribute
name="class">
<xsl:value-of select="@class"/> </xsl:attribute> </xsl:if>
<xsl:if test="(@id)"> <xsl:attribute name="id"> <xsl:value-of
select="@id"/> </xsl:attribute> </xsl:if> <!-- If the width of
the figure is smaller than the width of the containing image we
may have
display problems. If the width of the containging figure is
smaller than
the width of the image, make the figure width equal to the width
of hthe
image, otherwise use the width of the figure element -->
<xsl:choose>
<xsl:when test="@width lt image/@width"> <xsl:attribute
name="width"> <xsl:value-of select="@width"/> </xsl:attribute>
</xsl:when> <xsl:otherwise> <xsl:attribute name="width">
<xsl:value-of select="image/@width"/> </xsl:attribute>
</xsl:otherwise> </xsl:choose> <!-- We don't care about height as
much as we do width, the caption and image are contained inside
the
figure. We only test if it exists. It's up to the author to make
sure
there are no conflicts --> <xsl:if test="(@height)">
<xsl:attribute
name="height"> <xsl:value-of select="@height"/> </xsl:attribute>
</xsl:if> <!-- Alignment can be different. We can have a centered
image inside a left aligned figure --> <xsl:if test="(@align)">
<xsl:attribute name="align"> <xsl:value-of select="@align"/>
</xsl:attribute> </xsl:if> <xsl:apply-templates select="image"/>
<xsl:apply-templates select="figcaption"/> </xsl:element>
</xsl:template> <xsl:template match="figcaption">
<figcaption><xsl:apply-templates/></figcaption>
</xsl:template> <xsl:template match="image"> <xsl:element
name="img"> <xsl:attribute name="src"> <xsl:value-of
select="@src"/>
</xsl:attribute> <xsl:attribute name="alt"> <xsl:value-of
select="@alt"/> </xsl:attribute> <xsl:if test="(@width)">
<xsl:attribute name="width"> <xsl:value-of select="@width"/>
</xsl:attribute> </xsl:if> <xsl:if test="(@height)">
<xsl:attribute name="height"> <xsl:value-of select="@height"/>
</xsl:attribute> </xsl:if> <xsl:if test="(@align)">
<xsl:attribute name="align"> <xsl:value-of select="@align"/>
</xsl:attribute> </xsl:if> <xsl:if test="(@class)">
<xsl:attribute name="class"> <xsl:value-of select="@class"/>
</xsl:attribute> </xsl:if> <xsl:if test="(@id)">
<xsl:attribute name="id"> <xsl:value-of select="@id"/>
</xsl:attribute> </xsl:if> </xsl:element> </xsl:template>
```

From XML to PDF: Part 1: Special Transformation

Rather than having to deal with XSL-FO, another XML based vocabulary to create PDF content, we'll use XSLT to create another HTML file and process it with CSS Paged Media and the companion Generated Content for Paged Media specifications to create PDF content.

I'm not against XSL-FO but the structure of document is not the easiest or most intuitive. An example of XSL-FO looks like this:

```
<?xml version="1.0" encoding="iso-8859-1"?> (1)
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"> (2)
<fo:layout-master-set> (3)
<fo:simple-page-master master-name="my-page">
<fo:region-body margin="1in"/>
</fo:simple-page-master> </fo:layout-master-set>
<fo:page-sequence master-reference="my-page"> (4)
<fo:flow flow-name="xsl-region-body"> (5)
<fo:block>Hello, world!</fo:block> (6)
</fo:flow>
</fo:page-sequence>
</fo:root>
```

1. This is an XML declaration. XSL FO (XSLFO) belongs to XML family, so this is obligatory
2. Root element. The obligatory namespace attribute declares the XSL Formatting Objects namespace
3. Layout master set. This element contains one or more declarations of page masters and page sequence masters — elements that define layouts of single pages and page sequences. In the example, I have defined a rudimentary page master, with only one area in it. The area should have a 1 inch margin from all sides of the page
4. Page sequence. Pages in the document are grouped into sequences; each sequence starts from a new page. Master-reference attribute selects an appropriate layout scheme from masters listed inside `<fo:layout-master-set>`. Setting master-reference to a page master name means that all pages in this sequence will be formatted using this page master
5. Flow. This is the container object for all user text in the document. Everything contained in the flow will be formatted into regions on pages generated inside the page sequence. Flow name links the flow to a specific region on the page (defined in the page master); in our example, it is the body region
6. Block. This object roughly corresponds to `<div>` in HTML, and normally includes a paragraph of text. I need it here, because text cannot be placed directly into a flow

Rather than define a flow of content and then the content CSS Paged Media uses a combination of new and existing CSS elements to format the content. For example, to define default page size and then add elements to chapter pages looks like this:

```
@page {
size: 8.5in 11in; margin: 0.5in 1in;
/* Footnote related attributes */
counter-reset: footnote;
@footnote {
counter-increment: footnote;
float: bottom;
column-span: all;
height: auto;
}
}

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

The only problem with the code above is that there is no native broser support. For our demonstration we'll use Prince XML to tanslate our HTML/CSS file to PDF. In the not so distant future we will be able to do this transformation in the browser and print the PDF directly. Until then it's a two step process: Modifying the HTML we get from the XML file and running the HTML through Prince to get the PDF.

Modifying the HTML results

We'll use this opportunity to create an xslt customization layer to make changes only to the templates where we need to.

We create a customization layer by importing the original stylesheet and making any necessary changes in the new stylesheet. Imported stylesheets have a lower precedence order than the local version so the local version will win if there is conflict.

Only the templates defined in this stilesheet are overriden. If the template we use is not in this customization layer, the transformation engine will use the template in the base style sheet (book.xsl in this case)

The style sheet is broken by templates and explained below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xs="http://www.w3.org/2001/XMLSchema"
exclude-result-prefixes="xs" version="2.0">
<!--
XSLT Paged Media Customization Layer Makes the necessary
changes to the content to work with the Paged Media CSS
```

```
stylesheet
-->
<!-- First import the base stylesheet -->
<xsl:import href="book.xsl"/>

<!-- Define the output for this and all document children -->
<xsl:output name="xhtml-out" method="xhtml" indent="yes"
encoding="UTF-8" omit-xml-declaration="yes" />
```

The first difference in the customization layer is that it imports another style sheet (*book.xsl*). We do this to avoid having to copy the entire style sheet and, if we make changes, having to make the changes in multiple places.

We will then override the templates we need in order to get a single file to pass on to Prince or any other CSS Print Processor.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
exclude-result-prefixes="xs"
version="2.0">

<!-- First import the base stylesheet -->
<xsl:import href="book.xsl"/>

<!-- Define the output for this and all document children -->
<xsl:output name="xhtml-out" method="xhtml" indent="yes"
encoding="UTF-8" omit-xml-declaration="yes" />

<!-- Root template matching book -->
<xsl:template match="book">
<html>
<head>
<xsl:element name="title">
<xsl:value-of select="metadata/title"/>
</xsl:element>
<!-- Paged Media Styles -->
<link rel="stylesheet" href="css/pm-style.css" />
<!-- Load Paged Media definitions just so I won't forget it again
-->
<link rel="stylesheet" href="css/paged-media.css"/>
<!-- Use highlight.js and style -->
<xsl:if test="(code)">
<link rel="stylesheet" href="css/styles/railscasts.css" />
<!-- Load highlight.js -->
<script src="lib/highlight.pack.js"></script>
<script>
hljs.initHighlightingOnLoad();
```

```
</script>
</xsl:if>
<!-- <script src="js/script.js"></script> -->
</head>
<body>
<xsl:attribute name="data-type">book</xsl:attribute>
<xsl:element name="meta">
<xsl:attribute name="generator">
<xsl:value-of select="system-property('xsl:product-name')"/>
<xsl:value-of select="system-property('xsl:product-version')"/>
</xsl:attribute>
</xsl:element>

<xsl:apply-templates select="/" mode="toc"/>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>
```

Most of the root template deals with undoing some of the changes we made to create multiple pages.

We've changed the CSS we use to process the content. We use paged-media.css to create the content for our media files, mostly setting up the different pages based on the data-type attribute.

We use pm-styles.css to control the style of our documents specifically for our printed page application. We have to take into account the fact that Highlight.js is not working properly with Prince's Javascript implementation and that there are places where we don't want our paragraphs to be indented at all.

We moved elements from the original section templates. We test whether we need to add the Highlight.JS since we dropped the multipage output.

Overriding the section template

Sections are the element type that got the biggest makeover. What we've done:

- Remove filename variable. It's not needed
- Remove the result document element since we are building a single file with all our content
- Change way we check for the type attribute in sections. It will now terminate with an error if the attribute is not found
- Add the element that will build our running footer (p class="rh") and assign the value of the secion's title to it

```
<!-- Override of the section template.-->
<xsl:template match="section">
<section>
<xsl:choose>
<xsl:when test="string(@type)">
<xsl:attribute name="data-type">
<xsl:value-of select="@type"/>
</xsl:attribute>
</xsl:when>
<xsl:otherwise>
<xsl:message terminate="yes">Type attribute is required for paged
media. Check your section tags for missing type attributes
</xsl:message>
</xsl:otherwise>
</xsl:choose>
<xsl:if test="string(@class)">
<xsl:attribute name="class">
<xsl:value-of select="@class"/>
</xsl:attribute>
</xsl:if>
<xsl:if test="string(@id)">
<xsl:attribute name="id">
<xsl:value-of select="@id"/>
</xsl:attribute>
</xsl:if>

<!--
Running header paragraph. This will be take out of the regular
flow
of text so it doesn't matter if we add it or not
-->
<xsl:element name="p">
<xsl:attribute name="class">rh</xsl:attribute>
<xsl:value-of select="title"/>
</xsl:element> <!-- closses rh class -->
<xsl:apply-templates/>
</section>
</xsl:template>
```

Metadata

The Metadata section has been reworked into a new section with the title data-type. We then apply all children templates.

```
<!-- Metadata -->
<xsl:template match="metadata">
<xsl:element name="section">
<xsl:attribute name="data-type">titlepage</xsl:attribute>
<xsl:apply-templates/>
</xsl:element>
</xsl:template>
```

Titles and tables of content

The table of content is commented for now as I work on improving the content and placement of the table contents in the final document.

The title element has only one addition. We add an ID attribute created using XPath's generate-id function on the parent section element.

```
<!-- Create Table of Contents Work in progress-->
<xsl:template match="/" mode="toc"/>
<xsl:template match="title">
<xsl:element name="h1">
<xsl:attribute name="id">
<xsl:value-of select="generate-id(..)"/>
</xsl:attribute>
<xsl:if test="string(@align)">
<xsl:attribute name="align">
<xsl:value-of select="@align"/>
</xsl:attribute> </xsl:if>
<xsl:if test="string(@class)">
<xsl:attribute name="class">
<xsl:value-of select="@class"/>
</xsl:attribute>
</xsl:if>
<xsl:value-of select="."/>
</xsl:element>
<!-- closes h1 -->
</xsl:template>
</xsl:stylesheet>
```

With all this in place we can now look to the CSS Paged Media file.

HTML To PDF, Part 2: CSS Paged Media

With the HTML ready, we can no look at the CSS stylesheet to process it into PDF.

The extensions, pseudo elements and attributes we use are all part of the CSS Paged Media or Generated Content for Paged Media specifications. Where appropriate I've translated them to work on both PDF and HTML.

Book defaults

The first step in creating the default structure for the book using `@page` at-element.

Our base definition does the following:

1. Size the page to letter (8.5 by 11 inches), width first
2. Use CSS notation for margins. In this case the top and bottom margin are 0.5 inches and left and right are 1 inch
3. Reset the footnote counter.
4. Using the @footnote attribute do the following
    1. Increment the footnote counter
    2. Place footnote at the bottom using another value for the float attribute
    3. Span all columns
    4. Make the height as tall as necessary

```
/* STEP 1: DEFINE THE DEFAULT PAGE */
@page {
size: 8.5in 11in; (1)
margin: 0.5in 1in; (2)
/* Footnote related attributes */
counter-reset: footnote; (3)
@footnote {
counter-increment: footnote; (4.1)
float: bottom; (4.2)
column-span: all; (4.3)
height: auto; (4.4)
}
}
```

In later sections we'll create named page templates and associate them to different portions of our written content.

Page counters

We define two conditions under which we reset the page counter: When we have a book followed by a part and when we have a book followed by the a first chapter.

We do **not** reset the content when the path if from book to chapter to part.

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

Matching content sections to page types

The next section of the style sheet is to match the content on our book to pages in our style sheet.

The book is broken into sections with data-type attributes to indicate the type of content; we match the section[data-type] element to a page type along with some basic style definitions.

We will further define the types of pages later in the style sheet.

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

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

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

/* TOC */
section[data-type='toc'] {
page: toc;
page-break-before: 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='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 }

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

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

Front matter formatting

For each page of front matter contnt (toc, foreword and preface) we define two pages: left and right. We do it this way to acommodate facing pages with numbers on ooposite sides (for two sided printout)

For the front matter we chose to use Roman numerals on the bottom of the page

```
/* 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 }
}
```

Pages formatting

We use the same system we used in the front matter to do a few things with our content.

We first remove page numbering from the title page and dedication by setting the numbering on both bottom corners to normal.

```
/* 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 }

}
```

Now we start working on our chapter pages. The first thing we do is to place our running header content in the bottom middle of the page, regardless of whether it's left or right.

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

We next setup a blank page for our chapters and tell the reader that the page was intentionally left blank to prevent confusion

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

Then we number the pages the same way that we did for our front matter except that we use narabic numerals instead of Roman.

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

@page chapter: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 }
}
```

Running footer

We now style the running footer.

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

Footnotes and cross references

Footnotes are tricky, they consist of two parts, the footnote-call and the footnote content it-self. I'm still trying to figure out what the correct markup should be for marking up footnotes.

We've also defined a special class of links that appends a string and the the destination's page number.

```
/* 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]::after {
content: ' [See page ' target-counter(attr(href), page) ']'
}
```

PDF Bookmarks

PDF bookmarks allow you to navigate your content form the left side bookmark menu as show in the image below

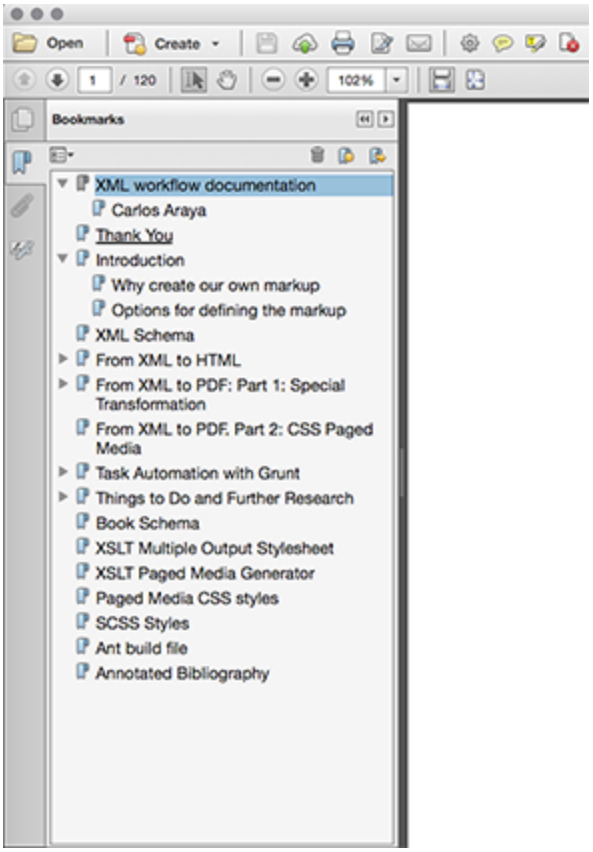

Figure 1: Example of PDF bookmarks

For each heading level we do the following things for both Antenna House and PrinceXML:

- Set up the bookmark level
- Set up whether it's open or closed
- Set up the label for the bookmark

Only heading 1, 2 and 3 are set up, level 4, 5 and 6 are only set up as bookmarks only.

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

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();
}

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

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

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

Running PrinceXML

Once we have the HTML file ready we can run it through PrinceXML to get our PDF using CSS stylesheet for Paged Media we discussed above. The command to run the conversion for a book.html file is:

```
$ prince --verbose book.html test-book.pdf
```

Because we added the stylesheet link directly to the HTML document we can skip declaring it in the conversion itself. This is always a cause of errors and frustratoins for me so I thought I'd save everyone else the hassle.

CSS Styles for Paged Media

This is the generated CSS from the SCSS style sheets (see the scss/ directory for the source material.) I've chosen to document the resulting stylesheet here and document the SCSS source in another document to make life simpler for people who don't want to deal with SASS or who want to see what the style sheets look like.

Typography derived from work done at this URL: http://bit.ly/16N6Y2Q

The following scale (also using minor third progression) may also help: http://bit.ly/1D-dVbqK

Feel free to play with these and use them as starting point for your own work :)

The project currently uses these fonts:

- Roboto Slab for headings
- Roboto for body copy
- Source Code Pro for code blocks and preformated text

Font Imports

Even though SCSS Lint throws a fit when I put font imports in a stylesheet because they stop asynchronous operations, I'm doing it to keep the HTML files clean and because we are not loading the CSS on the page, we're just using it to process the PDF file.

Eventually I'll switch to locally hosted fonts using bulletproof font syntax (discussed here and available for use at Font Squirrel.)

At this point we are not dealing with font subsetting) but we may in case we need to.

```
@import url(http://fonts.googleapis.com/
css?family=Roboto:100italic,100,400italic,700italic,300,700,300it
alic,400);
@import url(http://fonts.googleapis.com/
css?family=Roboto+Slab:400,700);
@import url(http://fonts.googleapis.com/
css?family=Source+Code+Pro:300,400);
```

Defaults

```
html {
font-size: 16px;
overflow-y: scroll;
-ms-text-size-adjust: 100%;
-webkit-text-size-adjust: 100%;
}

body {
background-color: #fff;
color: #554c4d;
color: #554c4d;
font-family: Adelle, Rockwell, Georgia, 'Times New Roman', Times,
serif;
font-size: 1em;
font-weight: 100;
line-height: 1.1;
padding-left: 10em;
padding-right: 10em;
}
```

Blockquotes, Pullquotes and Marginalia

```
aside .lowlight {
opacity: .3;
}
aside .lowlight :hover {
opacity: 1;
}

aside h3 {
padding-left: 20px;
text-align: center;
}

aside {
border-bottom: 3px double #ddd;
border-top: 3px double #ddd;
color: #666;
font-size: 1.8em;
line-height: 1.4em;
padding-bottom: .5em;
padding-top: .5em;
```

```
width: 100%;
}
aside .pull {
margin-bottom: .5em;
margin-left: -20%;
margin-top: .2em;
}

.opening {
border-bottom: 3px double #ddd;
border-top: 3px double #ddd;
font-size: 2em;
margin-bottom: 10em;
padding-bottom: 2em;
padding-top: 2em;
text-align: center;
}

.margin-notes,
.content-left {
font-size: .75em;
margin-left: -230px;
margin-right: 20px;
text-align: right;
width: 230px;
}

.margin-notes-right,
.content-right {
font-size: .75em;
margin-left: 760px;
margin-right: -20px;
position: absolute;
text-align: left;
width: 230px;
}

.content-right {
font-size: .75em;
margin-left: 760px;
margin-right: -20px;
position: absolute;
text-align: left;
width: 230px;
}

.content-right ul,
.content-left ul {
```

```
list-style: none;
}

blockquote {
color: #222023;
font-size: 1.5em;
font-style: italic;
font-weight: 100;
margin-bottom: 2em;
margin-left: 4em;
margin-right: 4em;
margin-top: 2em;
}
blockquote p {
padding-left: .5em;
}

.pullquote {
border-bottom: 18px solid #000;
border-top: 18px solid #000;
font-size: 36px;
font-weight: 700;
letter-spacing: -.02em;
line-height: 38px;
margin-right: 100px;
padding: 20px 0;
position: relative;
width: 200px;
}
.pullquote p {
color: #00298a;
font-weight: 700;
position: relative;
text-transform: uppercase;
z-index: 1;
}
.pullquote p:last-child {
line-height: 20px;
padding-top: 2px;
}
.pullquote cite {
color: #333;
font-size: 18px;
font-weight: 400;
}
```

Paragraphs

```
p {
font-size: 1em;
margin-bottom: 1.3em;
}
p + p {
text-indent: 2em;
}

.first-line {
font-size: 1.1em;
text-indent: 0;
text-transform: uppercase;
}

.first-letter {
float: left;
font-size: 7em;
line-height: .8em;
margin-bottom: -.1em;
padding-right: .1em;
}
```

Lists

```
ul li {
list-style: square;
}

ol li {
list-style: decimal;
}
```

Figures and captions

```
figure {
counter-increment: figure_count;
margin-bottom: 1em;
margin-top: 1em;
}
```

```
figure figcaption {
font-weight: 700;
padding-bottom: 1em;
padding-top: .2em;
}
figure figcaption::before {
content: "Figure " counter(figure_count) ": ";
}
```

Headings

```
h1,
h2,
h3,
h4 {
font-family: 'Roboto Slab', sans-serif;
font-weight: 400;
line-height: 1.2;
margin: 1.414em .5em;
text-transform: uppercase;
}

h1 {
font-size: 3.157em;
margin-top: 0;
}

h2 {
font-size: 2.369em;
}

h3 {
font-size: 1.777em;
}

h4 {
font-size: 1.333em;
}
```

Different parts of the book

```
section[data-type='titlepage'] h1,
section[data-type='titlepage'] h2 {
text-align: center;
}
section[data-type='titlepage'] p {
text-align: left;
}
section[data-type='titlepage'] p + p {
text-indent: 0 !important;
}

section[data-type='dedication'] h1,
section[data-type='dedication'] h2 {
text-align: center;
}
section[data-type='dedication'] p {
text-indent: 0 !important;
}
```

Preformatted code blocks

```
pre {
overflow-wrap: break-word;
white-space: pre-line !important;
word-wrap: break-word;
}

pre code {
font-family: 'Source Code Pro', monospace;
font-size: 1em;
line-height: 1.2em;
page-break-inside: avoid;
}
```

Columns and miscelaneous classes

```
.justified {
text-align: justify;
}

video {
border: 1px solid #c5c5d0;
}

blockquote {
color: #222023;
font-size: 1.5em;
font-weight: 300;
margin: 2em 1.5em;
}

.columns2 {
column-count: 2;
column-gap: 3em;
column-fill: balance;
column-span: none;
line-height: 1em;
width: 100%;
}
.columns2 p:first-of-type {
border-top: 0;
margin-top: 0;
}
.columns2 p + p {
text-indent: 2em;
}

.columns3 {
column-count: 3;
column-gap: 10px;
column-fill: balance;
column-span: none;
width: 100%;
}
```

Task Automation with Grunt

Because we use XML we can't just dump our code in the browser or the PDF viewer we need to prepare our content for conversion to PDF before we can view it. There are also front-end web development best practices to follow.

This chapter will discuss tools to accomplish both tasks from one build file.

What software we need

For this to work you need the following software installed:

- Java (version 1.7 or later)
- Node.js (0.10.35 or later)

Once you have java installed, you can install the following Java package

- Saxon 9.0.6.4 for Java

A note about Saxon: OxygenXML comes with a version of Saxon Enterprise Edition. We'll use a different version to make it easier to use outside the editor.

Node packages are handled through NPM, the Node Package Manager. On the Node side we need at least the `grunt-cli` package installed globally. TO do so we use this command:

```
$ npm install -g grunt-cli
```

The -g flag will install this globally, as opposed to installing it in the project directory.

Now that we have the required sotfware installed we can move ahead and create our configuration files.

Optional: Ruby, SCSS-Lint and SASS

The only external dependencies you need to worry about are Ruby, SCSS-Lint and SASS. Ruby comes installe in most (if not all) Macintosh and Linux systems; an installer for Windows.

SASS (syntactically awesome style sheets) are a superset of CSS that brings to the table enhancements to CSS that make life easier for designers and the people who have to create the stylesheets. I've taken advantage of these features to simplify my stylesheets and to save myself from repetitive and tedious tasks.

SASS, the main tool, is written in Ruby and is available as a Ruby Gem.

To install SASS, open a terminal/command window and type:

```
$ gem install sass
```

Note that on Mac and Linux you may need to run the command as a superuser. If you get an error, try the following command:

```
$ sudo gem install sass
```

and enter your password when prompted.

SCSS-Lint is a linter for the SCSS flavor of SASS. As with other linters it will detect errors and potential erors in your SCSS style sheets. As with SASS, SCSSLint is a Ruby Gem that can be installed with the following command:

```
$ sudo gem install scss-lint
```

The same caveat about errors and installing as an administrator apply.

> *Ruby, SCSS-Lint and SASS are only necessary if you plan to change the SCSS/SASS files. If you don't you can skip the Ruby install and work directly with the CSS files*
>
> *If you want to peek at the SASS source look at the files under the scss directory.*

Installing Node packages

Grunt is a Node.js based task runner. It's a declarative version of Make and similar tools in other languages. Since Grunt and it's associated plugins are Node Packages we need to configure Node.

At the root of the project there's a `package.json` file where all the files necessary for the project have already been configured. All that is left is to run the install command.

```
$ npm install
```

This will install all the packages indicated in configuration file and all their dependencies; go get a cup of coffee as this may take a while in slower machines.

As it installs the software it'll display a list of what it installed and when it's done you'll have all the packages.

The final step of the node installation is to run bower, a front end package manager. It is not configured by default but you can use it to manage packages such as jQuery, Highlight.JS, Polymer web components and others.

Grunt & Front End Development best practices

While developing the XML and XSL for this project, I decided that it was also a good chance to test front end development tools and best practices for styling and general front end development.

One of the best known tools for front end development is Grunt. It is a Javascript task runner and it can do pretty much whatever you need to do in your development environment. The fact that Grunt is written in Javascript saves developers from having to learn another language for task management.

Grunt has its own configuration file (`Gruntfile.js`) one of which is provided as a model for the project.

As currently written the Grunt file provides the following functionality in the assigned tasks. Please note that the tasks with an asterisk have subtasks to perform specific functions. We will discuss the subtasks as we look at each portion of the file and its purpose.

```
Available tasks
autoprefixer Prefix CSS files. *
clean Clean files and folders. *
coffee Compile CoffeeScript files into JavaScript *
copy Copy files. *
jshint Validate files with JSHint. *
sass Compile Sass to CSS *
uglify Minify files with UglifyJS. *
watch Run predefined tasks whenever watched files change.
gh-pages Publish to gh-pages. *
gh-pages-clean Clean cache dir
mkdir Make directories. *
scsslint Validate `.scss` files with `scss-lint`. *
shell Run shell commands *
sftp Copy files to a (remote) machine running an SSH daemon. *
sshexec Executes a shell command on a remote machine *
uncss Remove unused CSS *
lint Alias for "jshint" task.
lint-all Alias for "scsslint", "jshint" tasks.
prep-css Alias for "scsslint", "sass:dev", "autoprefixer" tasks.
prep-js Alias for "jshint", "uglify" tasks.
generate-pdf Alias for "shell:single", "shell:prince" tasks.
generate-pdf-scss Alias for "scsslint", "sass:dev",
```

```
"shell:single",
"shell:prince" tasks.
generate-all Alias for "shell" task.
```

Setup

The first thing we do is declare two variables (module and require) as global for JSLint and JSHint. Otherwise we'll get errors and it's not essential to declare them before they are used.

We then wrap the Gruntfile with a self executing function as a deffensive coding strategy.

When concatenating Javascript files there may be some that use strict Javascript and some that don't; With Javascript vriable hoisting the use stric declaration would be placed at the very top of the concatenated file making all the scripts underneat use the strict declaration.

When concatenating Javascript files there may be some that use strict Javascript and some that don't; With Javascript vriable hoisting the use stric declaration would be placed at the very top of the concatenated file making all the scripts underneat use the strict declaration.

The function wrap prevents this by making the use strict declaration local to the file where it was written. None of the other templates will be affected and they will still execute from the master stylesheet. It's not essential for Grunt drivers (Gruntfile.js in our case) but it's always a good habit to get into.

```
/*global module */
/*global require */
(function () {
'use strict';
module.exports = function (grunt) {
// require it at the top and pass in the grunt instance
// it will measure how long things take for performance
//testing
require('time-grunt')(grunt);

// load-grunt will read the package file and automatically
// load all our packages configured there.
// Yay for laziness
require('load-grunt-tasks')(grunt);
```

The first two elements that work with our content are *time-grunt* and *load-grunt-tasks*.

Time-grunt provides a breakdown of time and percentage of total execution time for each task performed in this particular Grunt run. The example below illustrates the result when running multiple tasks.

```
Execution Time (2015-02-01 03:43:57 UTC)
loading tasks 983ms ████████████████████████████ 12%
scsslint:allFiles 1.1s ████████████████████████████ 13%
sass:dev 441ms ████████████ 5%
shell:html 1.5s ████████████████████████████████████████
18%
shell:single 1.2s ████████████████████████████ 14%
shell:prince 2.9s
████████████████████████████████████████
██ 36%
Total 8.1s
```

Load-grunt-tasks automates the loading of packages located in the `package.json` configuration file. It's specially good for forgetful people like me whose main mistake when building Grunt-based tool chains is forgetting to load the plugins to use :-).

Javascript

```
grunt.initConfig({

// JAVASCRIPT TASKS
// Hint the grunt file and all files under js/
// and one directory below
jshint: {
files: ['Gruntfile.js', 'js/{,*/}*.js'],
options: {
reporter: require('jshint-stylish')
// options here to override JSHint defaults
}
},

// Takes all the files under js/ and selected files under lib
// and concatenates them together. I've chosen not to mangle
// the compressed file
uglify: {
dist: {
options: {
mangle: false,
sourceMap: true,
sourceMapName: 'css/script.min.map'
},
files: {
'js/script.min.js': ['js/video.js', 'lib/highlight.pack.js']
}
}
},
```

JSHint will lint the Gruntfile itself and all files under the js/ directory for errors and potential errors.

```
$ grunt jshint
Running "jshint:files" (jshint) task
Gruntfile.js
line 9 col 33 Missing semicolon.
line 269 col 6 Missing semicolon.

? 2 warnings

Warning: Task "jshint:files" failed. Use --force to continue.
Aborted due to warnings.
```

Uglify allow us to concatenate our Javascript files and, if we choose to, further reduce the file size by mangling the code (See this [page](http://lisperator.net/uglifyjs/mangle) for an explanation of what mangle is and does). I've chosen not to mangle the code to make it easier to read. May add it as an option for production deployments.

SASS and CSS

As mentioned elsewhere I chose to use the SCSS flavor of SASS because it allows me to do some awesome things with CSS that I wouldn't be able to do with CSS alone.

The first task with SASS is convert it to CSS. For this we have two separate tasks. One for development (dev task below) where we pick all the files from the scss directory (the entire files section is equivalent to writing *scss/*.scss*) and converting them to files with the same name in the css directory.

```
// SASS RELATED TASKS
// Converts all the files under scss/ ending with .scss
// into the equivalent css file on the css/ directory
sass: {
dev: {
options: {
style: 'expanded'
},
files: [{
expand: true,
cwd: 'scss',
src: ['*.scss'],
dest: 'css',
ext: '.css'
}]
},
production: {
options: {
```

```
style: 'compact'
},
files: [{
expand: true,
cwd: 'scss',
src: ['*.scss'],
dest: 'css',
ext: '.css'
}]
}
},
```

There are two similar versions of the task. The development version will produce the format below, which is easier to read and easier to troubleshoot (css-lint, discussed below, tells you what line the error or warning happened in.)

```
@import url(http://fonts.googleapis.com/
css?family=Roboto:100italic,100,400italic,700italic,300,700,300it
alic,400);
@import url(http://fonts.googleapis.com/
css?family=Montserrat:400,700);
@import url(http://fonts.googleapis.com/
css?family=Roboto+Slab:400,700);
@import url(http://fonts.googleapis.com/
css?family=Source+Code+Pro:300,400);
html {
font-size: 16px;
overflow-y: scroll;
-ms-text-size-adjust: 100%;
-webkit-text-size-adjust: 100%;
}

body {
background-color: #fff;
color: #554c4d;
color: #554c4d;
font-family: Adelle, Rockwell, Georgia, 'Times New Roman', Times,
serif;
font-size: 1em;
font-weight: 100;
line-height: 1.1;
padding-left: 10em;
padding-right: 10em;
}
```

The production code compresses the output. It deletes all tabs and carriage returns to produce cod elike the one below. It reduces the file size by eliminating spaces, tabs and carriage returns inside the rules, otherwise both versions are equivalent.

```
@import url(http://fonts.googleapis.com/
css?family=Roboto:100italic,100,400italic,700italic,300,700,300it
alic,400);
@import url(http://fonts.googleapis.com/
css?family=Montserrat:400,700);
@import url(http://fonts.googleapis.com/
css?family=Roboto+Slab:400,700);
@import url(http://fonts.googleapis.com/
css?family=Source+Code+Pro:300,400);
html { font-size: 16px; overflow-y: scroll; -ms-text-size-adjust:
100%; -webkit-text-size-adjust: 100%; }

body { background-color: #fff; color: #554c4d; color: #554c4d;
font-family: Adelle, Rockwell, Georgia, 'Times New Roman', Times,
serif; font-size: 1em; font-weight: 100; line-height: 1.1;
padding-left: 10em; padding-right: 10em; }
```

I did consider adding cssmin but decided against it for two reasons:

- SASS already concatenates all the files when it imports files from the modules and partials directory so we're only working with one file for each version of the project (html and PDF)
- The only other file we'd have to add, normalize.css, is a third party library that I'd rather leave along rather than mess with.

The *scsslint* task is a wrapper for the scss-lint Ruby Gem that must be installed on your system. It warns you of errors and potential errors in your SCSS stylesheets.

We've chosen to force it to run when it finds errors. We want the linting tasks to be used as the developer's discretion, there may be times when vendor prefixes have to be used or where colors have to be defined multiple times to acommodate older browsers.

```
// I've chosen not to fail on errors or warnings.
scsslint: {
allFiles: [
'scss/*.scss',
'scss/modules/_mixins.scss',
'scss/modules/_variables.scss',
'scss/partials/*.scss'
],
options: {
config: '.scss-lint.yml',
force: true,
colorizeOutput: true
}
},
```

Grunt's autoprefixer task uses the CanIUse database to determine if properties need a vendor prefix and add the prefix if they do.

This becomes important for older browsers or when vendors drop their prefix for a given property. Rather than having to keep up to date on all vendor prefixed properties you can tell autoprefixer what browsers to test for (last 2 versions in this case) and let it worry about what needs to be prefixed or not.

```
autoprefixer: {
options: {
browsers: ['last 2']
},

files: {
expand: true,
flatten: true,
src: 'scss/*.scss',
dest: 'css/'
}
},
```

The last css task is the most complicated one. Uncss takes out whatever CSS rules are not used in our target HTML files.

```
// CSS TASKS TO RUN AFTER CONVERSION
// Cleans the CSS based on what's used in the specified files
// See https://github.com/addyosmani/grunt-uncss for more
// information
uncss: {
dist: {
files: {
'css/tidy.css': ['*.html', '!docs.html']
}
}
},
```

This is not a big deal for our workflow as most, if not all, the CSS is designed for the tags and classes we've implemented but it's impossible for the SASS/CSS libraries to grow over time and become bloated.

This will also become and issue when you decide to include third part libraries in projects implemented on top of our workflow. By running Uncss on all our HTML files except the file we'll pass to our PDF generator (docs.html) we can be assured that we'll get the smallest css possible.

We skip out PDF source html file because I'm not 100% certain that Uncss can work with Paged Media CSS extensions. Better safe than sorry.

Optional tasks

I've also created a set of optional tasks that are commented in the Grunt file but have been uncommented here for readability.

The first optional task is a Coffeescript compiler. Coffeescript is a scripting language that provides a set of useful features and that compiles directly to Javascript.

I some times use Coffeescript to create scripts and other interactive content so it's important to have the compilation option available.

```
// OPTIONAL TASKS
// Tasks below have been set up but are currently not used.
// If you want them, uncomment the corresponding block below

// COFFEESCRIPT
// If you want to use coffeescript (http://coffeescript.org/)
// instead of vanilla JS, uncoment the block below and change
// the cwd value to the locations of your coffee files
coffee: {
target1: {
expand: true,
flatten: true,
cwd: 'src/',
src: ['*.coffee'],
dest: 'build/',
ext: '.js'
},
```

The following two tasks are for managing file transfers and uploads to different targets.

One of the things I love from working on Github is that your project automatically gets an ssl-enabled site for free. Github Pages work with any kind of static website; Github even offers an automatic site generator as part of our your project site.

For the puposes of our workflow validation we'll make a package of our content in a build directory and push it to the gh-pages branch of our repository. We'll look at building our app directory when we look at copying files.

```
// GH-PAGES TASK
// Push the specified content into the repositories gh-pages
branch
'gh-pages': {
options: {
message: 'Content committed from Grunt gh-pages',
base: './build/app',
dotfiles: true
},
```

```
// These files will get pushed to the `
// gh-pages` branch (the default)
// We have to specifically remove node_modules
src: ['**/*']
},
```

There are times when we are not working with Github or pages. In this case we need to FTP or SFTP (encrypted version of FTP) to push files to remote servers. We use an external json file to store our account information. Ideally we'd encrypt the information but until then using the external file is the first option.

```
//SFTP TASK
//Using grunt-ssh (https://www.npmjs.com/package/grunt-ssh)
//to store files in a remote SFTP server. Alternative to gh-pages
secret: grunt.file.readJSON('secret.json'),
sftp: {
test: {
files: { "./": "*.json" },
options: {
path: '/tmp/',
host: '<%= secret.host %>',
username: '<%= secret.username %>',
password: '<%= secret.password %>',
showProgress: true
}
}
},
```

File Management

We've taken a few file management tasks into Grunt to make our lifes easier. The functions are for:

- Creating directories
- Copying files
- Deleting files and directories

We will use the mkdir and copy tasks to create a build directory and copy all css, js and html files to the build directory. We will then use the gh-pages task (described earlier) to push the content to the repository's gh-pages branches

```
// FILE MANAGEMENT
// Can't seem to make the copy task create the directory
// if it doesn't exist so we go to another task to create
// the fn directory
mkdir: {
build: {
```

```
options: {
create: ['build']
}
}
},

// Copy the files from our repository into the build directory
copy: {
build: {
files: [{
expand: true,
src: ['app/**/*'],
dest: 'build/'
}]
}
},

// Clean the build directory
clean: {
production: ['build/']
},
```

Watch task

Rather than type a command over and over again we can set up watchers so that, any time a file of the indicated type changes, we perform specific tasks.

As currentlly configured we track Javascript and SASS files.

For Javascript files anytime that the Gruntfile or any file under the Javascript directorie we run the JSHint task to make sure we haven't made any mistakes.

For our SASS/SCSS files, any files under the scss directory, we run the sass:dev task to translate the files to CSS.

```
// WATCH TASK
// Watch for changes on the js and scss files and perform
// the specified task
watch: {
options: {
nospawn: true
},
// Watch all javascript files and hint them
js: {
files: ['Gruntfile.js', 'js/{,*/}*.js'],
tasks: ['jshint']
```

```
},
sass: {
files: ['scss/*.scss'],
tasks: ['sass']
}
},
```

Compile and Execute

Rather than using Ant, I've settled on Grunt's shell task to run the compilation steps to create HTML and PDF. This reduces teh number of dependecies for our project and makes it easier to consolidate all the work.

We have three different commands:

- html will create multiple html files using Saxon, a Java XSLT processor
- single will create a single html file using Saxon
- prince will create a PDF based on the single html file using PrinceXML

We make sure that we don't continue if there is an error. Want to make sure that we troubleshoot before we get all the resulting files.

```
// COMPILE AND EXECUTE TASKS
// rather than using Ant, I've settled on Grunt's shell
// task to run the compilation steps to create HTML and PDF.
// This reduces teh number of dependecies for our project
shell: {
options: {
failOnError: true,
stderr: false
},
html: {
command: 'java -jar /usr/local/java/saxon.jar -xsl:xslt/book.xsl
docs.xml -o:index.html'
},
single: {
command: 'java -jar /usr/local/java/saxon.jar -xsl:xslt/
pm-book.xsl docs.xml -o:docs.html'
},
prince: {
command: 'prince --verbose --javascript docs.html -o docs.pdf'
}
}


}); // closes initConfig
```

Custom Tasks

The custom task uses one or more of the tasks defined above to accomplish a sequence of tasks.

Look at specific tasks defined above for specific definitions.

```
// CUSTOM TASKS
// Usually a combination of one or more tasks defined above
grunt.task.registerTask(
'lint',
[
'jshint'
]
);

grunt.task.registerTask(
'lint-all',
[
'scsslint',
'jshint'
]
);

// Prep CSS starting with SASS, autoprefix et. al
grunt.task.registerTask(
'prep-css',
[
'scsslint',
'sass:dev',
'autoprefixer'
]
);

grunt.task.registerTask(
'prep-js',
[
'jshint',
'uglify'
]
);

grunt.task.registerTask(
'generate-pdf',
[
'shell:single',
'shell:prince'
```

```
]
);

grunt.task.registerTask(
'generate-pdf-scss',
[
'scsslint',
'sass:dev',
'shell:single',
'shell:prince'
]
);

grunt.task.registerTask(
'generate-all',
[
'shell'
]
);


}; // closes module.exports
}()); // closes the use strict function
```

Things to Do and Further Research

These are the things I want to look at after finishing mvp.

Deep Linking using Emphasis

The NYT developed a deep linking library called Emphasis (code - writeup that would allow us to create links to specific areas of our content.

Downside is that it uses jQuery and I'm not certain I want to go through the pain in the ass process of converting it to plain JS or ES6 (and if it's even possible)

Still, if we use jQuery for something else (video manipulation?) it may be worth exploring both as a sharing tool and as a technology.

One thing it doesn't do is handle mobile well, if at all. How do we make this tool work everywhere? Pointer Events?

Explore how to add other parts of a book structure

Right now we're working with chapters and chapter-like structures. What would it take to add parts? Do we need to add them to the schema and let them trickle from there? Do we really need them?

Build Media Queries

Particularly if we want to use the same XSLT and CSS for mutliple projects we need to be able to tailor the display for different devices and viewports.

Media Queries are the best solution (or are they?)

Using XSLT to build navigation

The same way we build the table of content should allow us to build navigation within the pages of a publication using preceeding-sibling and following-sibling logic

Create a better way to generate filenames

The current way to create filenames doesn't take into account that different *section/@type* elements have different starting values. Can I make it start from 1 for every @type in the document?

Expand the use cases for this project

The original idea was for text and code-heavy content. Is there a case to be made for a more expressive vocabulary? I'm thinking of additional elements for navigation and content display such as asides and blockquotes

Explore implementing a serviceworker solution

The core of the proposed offline capabilities is a scoped service worker that will initially handle the caching of the publication's content. We take advantage of the multiple cache capabilitity available with service workers to create caches for individual unitts of content (like magazine issues) and to expire them within a certain time period (by removing and deleting the cache).

For publications needing to pull data from specific URLs we can special case the requests based on different pieces of the URL allowing to create different caches based on edition (assuming each edition is stored in its own directory), resource type or even the URL we are requesting.

Serviceworkers have another benefit not directly related with offline connections. They will give all access to our content a speed boost by eliminating the network roundtrip after the content is installed. If the content is in the cache, the resource's time to load is only limited by the Hard Drive's speed.

Limitations

As powerful as service workers are they also have some drawbacks. They can only be served through HTTPS (you cannot install a service worker in a non secure server) to prevent man-in-the-middle attacks.

There is limited support for the API (only Chrome Canary and Firefox Nightly builds behind a flag will work.) This will change as the API matures and becomes finalized in the WHATWG and/or a recommendation with the W3C.

Even in browsers that support the API the support is not complete. Chrome uses a polyfill for elements of the cache API that it does not support natively. This should be fixed in upcoming versions of Chrome and Chromium (the open source project Chrome is based on.)

We need to be careful with how much data we choose to store in the caches. From what I understand the ammount of storage given to offline applications is divided between all offline storage types: IndexedDB, Session Storage, Web Workers and ServiceWorkers and this amount is not consistent across all browsers.

Furthermore I am not aware of any way to increase this total amount or to specifically increase the storage assigned to ServiceWorkers; Jake Archibald mentions this in the offline cookbook section on cache persistence

Implementing Graphic Callouts

What would it take to get callouts to use graphics like Docbook?

It would take the following:

- Create a new element (callout)
- Add the callout element to the code data model

This would work for code, but not necessarily for images or other elements that need callouts

Annotated Bibliography