XML workflow documentation

Carlos Araya

Table of Contents

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 toolchains. 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.

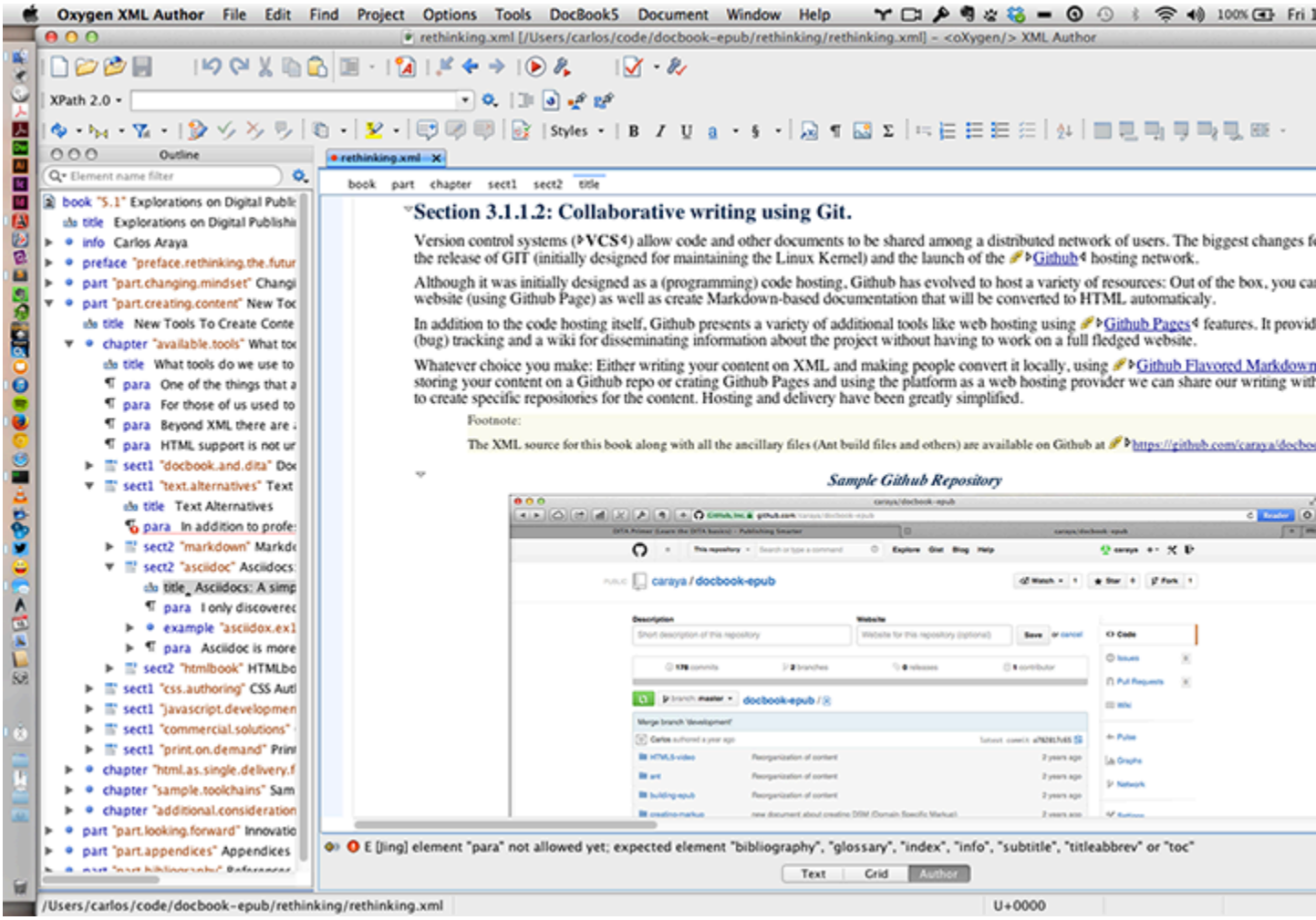

Figure 1:

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

Platforming books from a single source

The formats chosen for this project (HTML, PDF and ePub) were not selected at random. I'm following Craig Mod's model as outlined in Platforming Books with my own ideas of what we need to produce.

> The three formats are:

- (X)HTML
- PDF
- ePub3

We'll discuss the rationale for each format, what this project is not, and what we can do moving forward.

(X)HTML

PDF

ePub3

Introduction

One of the biggest limitations of markup languages, in my opinion, is how confining they are. Even large vocabularies like [Docbook](http://docbook.org) 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.

It has been suggested that Pandoc may be a better alternative than rolling your own markup and style sheet sets. While Pandoc is a very powerful tool it tries to do too much for too many. According to the website, Pandoc supports:

- HTML formats: XHTML, HTML5, and HTML slide shows using Slidy, reveal.js, Slideous, S5, or DZSlides.
- Word processor formats: Microsoft Word docx, OpenOffice/LibreOffice ODT, OpenDocument XML
- Ebooks: EPUB version 2 or 3, FictionBook2
- Documentation formats: DocBook, GNU TexInfo, Groff man pages, Haddock markup
- Page layout formats: InDesign ICML
- Outline formats: OPML
- TeX formats: LaTeX, ConTeXt, LaTeX Beamer slides
- PDF via LaTeX
- Lightweight markup formats: Markdown, reStructuredText, AsciiDoc, MediaWiki markup, DokuWiki markup, Emacs Org-Mode, Textile
- Custom formats: custom writers can be written in lua.

My problems with Pandoc are:

- You can only convert your content to PDF if you have LaTex installed. Even a minimal LaTex library requires a fairly large install
- LaTex installs are different for Macintosh, Windows and Linux. This makes a uniform support system impossible
- If you want to add custom formats you have to do so in Lua. I'd rather not learn another programming language just for one project

Why create our own markup

Rather than figure out how to use someone else tool for this proof of concept I've decided to build my own markup vocabulary and explore XML, XSL and CSS as the tools to create HTML and print-ready content.

Using XSL and CSS to manipulate XML content reduces the number of external dependencies. As it stands right now, the project depends on Saxon to convert XML to HTML and PrinceXML

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)

It reduces the ammount of external code that we have to adapt in order to acomplish a given task.

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:

```
<book>
<metadata>
<title>The adventures of SHerlock Holmes</title>

<author>
<first-name>Arthur</first-name>
<surname>Connan Doyle</surname>
</author>
</metadata>

<section type="chapter">
<para>Lorem Ipsum</para>
<para>Lorem Ipsum</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.

The XML Schema

> This version of the documentation is based on commit 542178fb21 to the Github repository. Any differences between this document and the repository should be resolved in favor of the repository (repo is always right)

The idea behind the schema is to create as clean a document as possible. The base document uses classes and IDs to avoid having to add styles directy to the document and leaving all the styling to CSS. There are 2 exceptions for addresses (discussed when we look at the XML to XHTML conversion.)

We cover all the content for the schema but will only detail the things I belive are important to understand the choices I made and why the schema was created this way.

Why a schema

There are many ways to define a schema. There is the original XML Schema from W3C, there is RelaxNG developed by OASIS in a technical committee and Schematron defined as an ISO standard.

With all these choices why did I stick with Schema?

- It's the most widely supported
- It can be converted to RelaxNG or Schematron (an experimental conversion to RelaxNG is available in the rng directory of the repository)

thor elements. At leaher the Schema or RelaxNG version. This document will refer to the Schema.

Getting started

As with all XML document the schema needs to define the XML Prologue (*<?xml version="1.0" encoding="UTF-8"?>*), the root element (*<xs:schema*), the namespaces we'll use for the project and the default forms for elements (*elementFormDefault="qualified"*) and attributes (*attributeFormDefault="unqualified"*`)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:dc="http://purl.org/dc/elements/1.1/"
```

```
elementFormDefault="qualified"
attributeFormDefault="unqualified">
```

Form defaults refer to whether we need to add the namespace prefix to our default elements. To avoid confusion I've chosen to add namespace prefixes to all our elements (using `xs:` as the namespace prefix for the schema name space.) Doing that for attributes is unnecessary; it may not always be the case.

We begin the actual work in the schema by defining some basic types that will be the basis of elements and complex types later in the schema.

Most of the simeType elements are created by restriction. We base the element in one of the default Schema data types.

For example: token255 is based on the toke schema element and is restricted to a maximum length of 255 characters. When this length is not enough we can use strings or other data types.

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

ISBN-type10 uses string as its base and creates a regular expression to match the format of ISBN codes used to identify books worldwide.

The ISBN element only matches one format. There are many formats that could match an ISBN record depending on where the book was published and which country it was registered in. There is a simple type element for ISBN numbers worldwide available at XFront. The code can be incorporated to the schema at a later time.

```
<xs:simpleType name="ISBN-type10">
<xs:restriction base="xs:string">
<xs:pattern value="0-[0-1][0-9]-\d{6}-[0-9x]"/>
</xs:restriction>
</xs:simpleType>
```

The align simple type is an enumeration. We list all the possible values for align attribute so we can reference them later without having to type them all the time.

```
<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>
```

Languages are handled using an element types as the language primitive (*xs:language*.) We can use it anywhere in the schema where we are allowed to use children elements.

Another possibility is to convert it to an attribute and move it to the generic properties attribute group discussed below.

```
<xs:element name="language" type="xs:language"/>
```

The attribute group element adds all the attributs with one statement. As it currently set up, it adds id and class attributes to the elements it's been added to. A future enhancement may be to add the language element as an attribute.

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

Organization and children

Initially there was no organization element until the question came up: **What happens when an author is not a person but a company or group?**

We keep the organization element as generic as possible to make sure we can use it in different instances. The only thing we know we'll need is the organization's name... everything else can be added when we build elements on top of organization (like publisher, discussed below.)

Address is one of those additional elements we add to organization. Addresses are string based and use the U.S. model.

Finally we build a publisher element by putting together our organization and address elements. Notice how the complex type and the element are called differently.

```
<xs:complexType name="organization">
<xs:all>
<xs:element name='name' type="xs:string"/>
</xs:all>
</xs:complexType>

<xs:element name="address">
```

```
<xs:complexType>
<xs:sequence>
<xs:element name="recipient" type="xs:string"/>
<xs:element name="street" type="xs:string"/>
<xs:element name="city" type="xs:string"/>
<xs:element name="state" type="xs:string"/>
<xs:element name="postcode" type="xs:string"/>
<xs:element name="country" type="xs:token"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="publisher">
<xs:complexType mixed="true">
<xs:all>
<xs:element name="name" type="organization"/>
<xs:element ref="address"/>
</xs:all>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

Person and children elements

We now move to define individuals and their roles.

The base class is person where we define a first-name and surname. We'll use this to create the roles for our different users with additional elements and attributes where necessary.

```
<!-- complex types to create groups of similar person items -->
<xs:complexType name="person">
<xs:sequence>
<xs:element name="first-name" type="xs:string"/>
<xs:element name="surname" type="xs:string"/>
</xs:sequence>
</xs:complexType>
```

Author, editor and otherRole use person as the base and then add additional elements to expand the person based on the type of editor (for the editor element) or the role they play in the book (for otherRole.)

During initial develpment I thought I'd just only work with plural elements (authors, editors and otherRoles) but soon realized that it took a lot of flexibility out of the schema since there may be other places where we need this information. For example, we may have an edited volume where each chapter has one or more authors.

So we have individual author (directly based on person), editor (with a *type* attribute to indicate what kind of editor the person is) and otherRole (for roles other than editor we use this with the *role* attribute.)

```
<xs:complexType name="author">
<xs:complexContent>
<xs:extension base="person"/>
</xs:complexContent>
</xs:complexType>

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

With individual roles created we can now create elements for multiple individuals.

as ones that contain multiple # characters or heast 1 author is required for the document to validate, whether we choose to use it or not. Remember that the transformation (using XSLT) doesn't have to use all the elements on the XML source.

Editors and otherRoles wrap around individual elements (editor and otherRole) to provide an easier way to work with them in XSLT later on.

Also note that we require 0 or more instances of the base type, rather than 1. This is my way of making the element optional: either we have zero, one or more than one children.

```
<!-- Wrappers around complext types -->
<xs:element name="authors">
<xs:complexType mixed="true">
<xs:choice minOccurs="1" maxOccurs="unbounded">
<xs:element name="author" type="author"/>
```

```
</xs:choice>
</xs:complexType>
</xs:element>

<xs:element name="editors">
<xs:complexType mixed="true">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="editor" type="editor"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:element name="otherRoles">
<xs:complexType mixed="true">
<xs:sequence minOccurs="0" maxOccurs="unbounded">
<xs:element name="otherRole" type="otherRole"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

Metadata and publishing information

In order to acommodate publishing information, we add multiple publishing related elements to account for publishing and publishing related information.

Most of these elements (except pubdate) are made of 1 or more paragraphs.

```
<xs:element name="releaseinfo">
<xs:complexType mixed="true">
<xs:choice minOccurs="1" maxOccurs="unbounded">
<xs:element ref="para"/>
</xs:choice>
</xs:complexType>
</xs:element>

<xs:element name="copyright">
<xs:complexType mixed="true">
<xs:choice minOccurs="1" maxOccurs="unbounded">
<xs:element ref="para"/>
</xs:choice>
</xs:complexType>
</xs:element>

<xs:element name="legalnotice">
<xs:complexType mixed="true">
<xs:choice minOccurs="1" maxOccurs="unbounded">
<xs:element ref="para"/>
</xs:choice>
```

```
</xs:complexType>
</xs:element>

<xs:element name="abstract">
<xs:complexType mixed="true">
<xs:choice minOccurs="1" maxOccurs="unbounded">
<xs:element ref="para"/>
</xs:choice>
</xs:complexType>
</xs:element>
```

The date is different as it's based on the date schema type which in turn it's based on the ISO 8601 standard.

An example of a valid ISO 8601 date is: *2015-02-28*

This is also the standard that handles time so, in theory, we could build a date/time structure including a date formated like the example above plus time and timezone offset but, unless we're required to we will avoid that much level of detail.

```
<xs:element name="pubdate" type="xs:date"/>
```

The idea behind revision and revhistory is to provide an accountability chain for the publication's history. We can have one or more paragraphs where we outline the following information:

- Revision number (cast as a string)
- Date (using the pubdate element defined earlier)
- Author's initials (short 255 character string)
- Revision description/notes (short 255 character string)

The revhistory element is a set of one or more revisions. We'll use XSLT to put them in na div tag or inside a table structure.

```
<xs:element name="revision">
<xs:complexType mixed="true">
<xs:all>
<xs:element name="revnumber" type="xs:string"/>
<xs:element ref="pubdate"/>
<xs:element name="authorinitials" type="token255"/>
<xs:element name="revnotes" type="token255"/>
</xs:all>
</xs:complexType>
</xs:element>

<xs:element name="revhistory">
<xs:complexType mixed="true">
<xs:sequence minOccurs="1" maxOccurs="unbounded">
```

```
<xs:element ref="revision"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

Links and related elements

Links are essential to the web. They allow us to connect our content to other elements inside and outside the pages we create. We add class and ID from `genericPropertiesGroup` and two new elements:

- *href* indicates the URL (local or relative) we are linking to
- *label* used as the text of the link and also the label attribute created for accessibility

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

About the xs:anyURI data type

> Some information and examples for this section taken from http://www.datypic.com/sc/xsd/t-xsd_anyURI.html

Unti I used the value xs:anyURI I wasn't sure what it did and how much work it would actually save me. I thought it was just another name for URLs but they are much more and the amount of work they save is significant.

URI (uniform resource indicator) are a superset of the web's URL. There are two main types of URIs: absolute and relative.

Absolute URIs provide the entire context for locating the resources, such as **http://datypic.com/prod.html**.

Relative URIs are specified as the difference from a base URI, such as **../prod.html**. It is also possible to specify a fragment identifier, using the # character, such as **../prod.html#shirt**. Note that when relative URI references are used as values of xsd:anyURI there is no attempt made to track the base of the URI.

The three previous examples happen to be HTTP URLs (Uniform Resource Locators), but URIs also encompass URLs of other schemes (e.g., FTP, gopher, telnet), as well as URNs (Uniform Resource Names). URIs doen't have to exist to be valid; there is no need for a resource to live at http://datypic.com/prod.html for the URI pointing to it to validate.

URIs require that some characters be escaped with their hexadecimal Unicode code point preceded by the % character. This includes non-ASCII characters and some ASCII characters, namely control characters, spaces, and the following characters (unless they are used as deliimiters in the URI): <>#%{}|\^`. For example, **../édition.html** must be represented instead as **../%C3%A9dition.html**, with the é escaped as %C3%A9. However, the anyURI type will accept these characters either escaped or unescaped. With the exception of the characters % and #, it will assume that unescaped characters are intended to be escaped when used in an actual URI, although the schema processor will do nothing to alter them. It is valid for an anyURI value to contain a space, but this practice is strongly discouraged. Spaces should instead be escaped using %20.

need</item> <item>autoplay — Hint that the media resource can be started automatically when the page is loaded</item> <item>loop — Whether to loop the media resource</item> <item>muted — Whether to mute the media resource by default</item> <item>controls — Show user agent controls</item> </ulist> <h3>Additional Resources</h3> <ulist> <item> <link href="http://www.w3.org/html/wg/drafts/characters or have % characters that are not followed by two hexadecimal digits.

For more information on URIs, see RFC 2396, Uniform Resource Identifiers (URI): Generic Syntax.

Valid URI matching xs:anyURI

- http://datypic.com **absolute URI (also a URL)**
- mailto:info@datypic.com **absolute URI**
- ../%C3%A9dition.html **relative URI containing escaped non-ASCII character**
- ../édition.html **relative URI containing unescaped non-ASCII character**
- http://datypic.com/prod.html#shirt **URI with fragment identifier**
- ../prod.html#shirt **relative URI with fragment identifier**
- urn:example:org **URN**
- an empty value is allowed

Div and span

Div and Span are subcontainers for block level elements (div) and inline elements (span). As containers I have to decide what elements are allowed as children and that's not always easy; any time we add a new element to the schema we have to decide if we're using it as a block element (that can be used inside a div) and/or an inline element (and whether that inline element can be used inside a span.)

The only special thing about div and span is the additional attribute type. We use type to build *data-type* and *epub:type* attributes. ePub and Edupub have lists of allowed values for the attribute and it's a good starting point for work with Javascript or additional functionality.

Div is a subsection of our section elements. Most of the elements that we can add to a section are also valid inside a div, including other div elements.

Span is used mostly to hold styles that apply to part of a paragraph and can be nested to build more elaborate effects and styles using class and id attributes.

```
<xs:element name="div">
<xs:complexType mixed="true">
<xs:sequence>
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="language"/>
<xs:element ref="anchor"/>
<xs:element ref="code"/>
<xs:element ref="para"/>
<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="video"/>
<xs:element ref="aside"/>
<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"/>
</xs:complexType>
</xs:element>

<xs:element name="span">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="language"/>
<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:attribute name="type" type="xs:token" use="optional"/>
</xs:complexType>
</xs:element>
```

Figure and image

Figure and image are the two ways the schema provides for working with images. Rather than make several elements and attributes optional and make the image elements brittle we allow both a figure element with caption and image children and an image without caption.

Most of the time I will use the figure/caption/image combination to give a numbering schema when performing the transformations.

```
<xs:element name="figure">
<xs:complexType mixed="true">
<xs:all>
<xs:element ref="image"/>
<xs:element ref="figcaption"/>
</xs:all>
<xs:attribute name="height" type="xs:nonNegativeInteger"
use="optional"/>
<xs:attribute name="width" type="xs:nonNegativeInteger"
use="optional"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

Captions are made of a single paragraph with all the children available to paragraphs elsewhere. This way we can add links and styles to the captions.

```
<xs:element name="figcaption">
<xs:complexType>
<xs:sequence>
<xs:element ref="para"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

One last thing to note about figures and images: Since figure and image have the same attributes we can combine them in different combinations. For example: we can have a left centered figure and caption with a right-aligned image inside.

```
<xs:element name="image">
<xs:complexType>
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="src" type="xs:token" use="required"/>
<xs:attribute name="height" type="xs:nonNegativeInteger"
use="required"/>
<xs:attribute name="width" type="xs:nonNegativeInteger"
use="required"/>
<xs:attribute name="alt" type="token255" use="required"/>
<xs:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>
```

Video

Video is a rather complex subject. According to the specification there are two ways to create video elements. I've chosen to adopt the multi-source element rather than a single src attribute to the video tag itself.

We use two children element: source and track.

The *source* element is for the actual video content. The element contains two attributes: *src* indicates the location of the video source. *Type* indicates the mimetype of the video and, optionally the codecs that were used to encode the video. This is particularly important with MP4 video as there are muliple "profiles" that may or may not be supported in all devices.

At least one of source element is required.

The track element is used to provide additional information to the video using the [VTT community specification](http://dev.w3.org/html5/webvtt/). You can use VTT to provide captioning and subtitles in multiple languages allowing the user to decide what language and what type of track they want to use with the video.

There are other uses for VTT tracks discussed in the HTML5 video captioning article listed in resources.

Attributes for video tag

- width — Horizontal dimension
- height — Vertical dimension
- poster — Poster frame to show prior to video playback
- preload — Hints how much buffering the media resource will likely need

- autoplay — Hint that the media resource can be started automatically when the page is loaded
- loop — Whether to loop the media resource
- muted — Whether to mute the media resource by default
- controls — Show user agent controls

Additional Resources

- HTML5 video specification
- WebVTT Draft Community Specification
- Video in ePub discusses HTML5 video. While written for ePub it is also applicable to HTML5 content in general
- HTML5 video captioning using VTT discusses captioning video using VTT tracks

```
<!-- Video and multimedia -->
<xs:element name="video">
<xs:complexType mixed="true">
<xs:choice>
<xs:element ref="source" minOccurs="1" maxOccurs="unbounded"/>
<xs:element ref="track" minOccurs="0" maxOccurs="unbounded"/>
</xs:choice>
<xs:attribute name="height" type="xs:nonNegativeInteger"/>
<xs:attribute name="width" type="xs:nonNegativeInteger"/>
<xs:attribute name="controls" type='xs:string' use="optional"/>
<xs:attribute name="poster" type="xs:anyURI" use="optional"/>
<xs:attribute name="autoplay" type="xs:string" use="optional"/>
<xs:attribute name="preload" type="xs:string" use="optional"
default="none"/>
<xs:attribute name="loop" type="xs:string" use="optional"/>
<xs:attribute name="muted" type="xs:string" use="optional"/>
</xs:complexType>
</xs:element>

<xs:element name="source">
<xs:complexType>
<xs:attribute name="src" type='xs:string' use="required"/>
<xs:attribute name="type" type='xs:string' use="optional"/>
</xs:complexType>
</xs:element>

<xs:element name="track">
<xs:complexType>
<xs:attribute name="src" type='xs:string' use="required"/>
<xs:attribute name="label" type='xs:string' use="required"/>
<xs:attribute name="kind" type='xs:string'/>
<xs:attribute name="srclang" type='xs:string' default='en'/>
</xs:complexType>
</xs:element>
```

Styles

Styles are hints for the transformation engine to apply rules and classes to the enclosed elements. We allow for nested styles by indetifying what styles can nest inside each of our four basic styles.

```
<!-- Style elements -->
<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="underline">
</xs:annotation>
<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>
```

Emphasis is the only style that can be nested in itself. The idea is that if an emphasis element is nested inside another emphasis it will display as normal text.

```
<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>
```

Lists

When I originally conceived the schema I had only one list element that, based on a type attribute, would generate the correct type of list (numbered or bulleted.) as I was implementing this I trealized it was too brittle and hard to work with and hard to maintain.

So I broke the element into two separate list elements; one for bulleted (*ulist*) and one for numbered (*olist*) lists. They can also nest other list elements (of either tupe) inside.

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

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

They share a common *item* element that contains a reduced subset of the paragraph element content. We don't want a full paragraph in each item because they'll mess up the display when we convert them to HTML or PDF. We do want the ability to style the content, link or provide span and quotes elements.

```
<xs:element name="item">
<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>
```

Fenced code blocks

I love the way Github displays fenced code blocks and wanted to get my HTML and, if possible, the PDF generated from XML to look similar to it. After researching the issue I found out code highlighting libraries for the web. The one I chose (highlight.js) also works with PrinceXML (with some extra work.)

The schema defines the container for the code, the XSLT transformations, discussed in a later chapter, add the appropriate HTML tags and scripts to highlight the code and PrinceXML will honor the code highlight in the HTML and display it in the resulting PDF.

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

Blockquotes, asides and marginalia

> Element descriptions taken from the http://www.w3.org/html/wg/drafts/html/master/semantics.html

Blockquotes, asides and quotes provide related or parenthetical content.

The blockquote element represents content that is quoted from another source, optionally with a citation which must be within a footer or cite element, and optionally with inline changes such as annotations and abbreviations.

Attribution provides a way to give credit for the quote.

```
<!-- Blockquotes, asides and marginalia -->
<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="blockquote">
<xs:complexType mixed="true">
<xs:choice>
<xs:element ref="language" minOccurs="0"/>
<xs:element ref="anchor"/>
<xs:element ref="attribution"/>
<xs:element ref="para" minOccurs="1"/>
</xs:choice>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

The aside element represents a section of a page that consists of content that is tangentially related to the content around the aside element, and which could be considered separate from that content. Such sections are often represented as sidebars in printed typography. It can be used for typographical effects like pull quotes or sidebars, for advertising, for groups of nav elements, and for other content that is considered separate from the main content of the page.

It's not appropriate to use the aside element just for parentheticals, since those are part of the main flow of the document.

```
<xs:element name="aside">
<xs:complexType mixed="true">
<xs:sequence>
<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"/>
</xs:complexType>
</xs:element>
```

The quote element represents some inline content quoted from another source.

Content inside a quote element must be quoted from another source, whose address, if it has one, may be cited in the cite attribute. The source may be fictional, as when quoting characters in a novel or screenplay.

If the cite attribute is present, it must be a valid URL potentially surrounded by spaces. To obtain the corresponding citation link, the value of the attribute must be resolved relative to the element. User agents may allow users to follow such citation links, but they are primarily intended for private use (e.g. by server-side scripts collecting statistics about a site's use of quotations), not for readers.

The quote element must not be used in place of quotation marks that do not represent quotes; for example, it is inappropriate to use the quote element for marking up sarcastic statements.

```
<xs:element name="quote">
<xs:complexType mixed="true">
<xs:all>
<xs:element ref="language"/>
</xs:all>
<xs:attribute name="cite" type="xs:anyURI" use="optional"/>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

Paragraphs

The paragraph is our main unit of content and where we do most, if not all, our inline styling of elements (indicated by what children elements are allowed to the paragraph)

In addition to the genericPropertiesGroup attributes, we also use align to control the horizontal alignment of the paragraph. This is useful when the paragraph is inside an attribution and I'd like the attribution to be right aligned while keeping the paragraph aligned to the left.

```
<xs:element name="para">
<xs:complexType mixed="true">
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="language"/>
<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:attribute name="align" type="align" use="optional"
default="left"/>
</xs:complexType>
</xs:element>
```

Headings

Headins should be self explainatory. They are used to indicate headings and structure in the document. While we only show the level 1 heading, the other levels (h2 thought h6) share the same structure.

```
<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>
```

Metadata element

The metadata element contains all the elements for our publication that are not content. Information such as copyright, legal notices, publisher, author and staff information.

I placed it in a separate element rather than incorporate it in a section with the appropriate type because keeping it separate makes it easier to transfor and pick the order and the types of elements we'll use for each type of document we generate.

Having a separate metadata element makes it easier to add or take away elements as needed.

```
<!-- Metadata element -->
<xs:element name="metadata">
<xs:complexType>
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element name="isbn" type="ISBN-type10"/>
<xs:element name="edition" type="xs:string"/>
<xs:element ref="authors"/>
<xs:element ref="editors"/>
<xs:element ref="otherRoles"/>
<xs:element ref="publisher"/>
<xs:element ref="releaseinfo"/>
<xs:element ref="copyright"/>
<xs:element ref="legalnotice"/>
<xs:element name="title" type="xs:string"/>
<xs:element name="subtitle" type="xs:string"/>
<xs:element ref="language"/>
<xs:element ref="revhistory" minOccurs="0"/>
<xs:element ref="para"/>
</xs:choice>
</xs:complexType>
</xs:element>
```

Section element

Sections are the primary container for our documents. It is at this level that we will create separate files when I start looking at multiple output.

I also use the *type* attribute here to indicate the type of section it is. The default value for it is chapter since it's the one I use most often. I've been thinking whether I want to create an enumeration of all the different values for type (based on epub and edupub profiles) to make it easier to create (oXygen gives me a list of all possible values for type and it currently only shows the default.)

Other than the title element, defined as a string schema type, all other elements reference reference other elements in the schema.

It derives class and id from our genericPropertiesGroup and the type attribute.

```
<!-- Section element -->
<xs:element name="section">
<xs:complexType mixed="true">
<xs:sequence>
<xs:element name="title" type="xs:string" minOccurs="0"
maxOccurs="1"/>
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="anchor"/>
<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="video"/>
<xs:element ref="aside"/>
<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>
```

Table of contents placeholder

Even though we're generating the table of contents in the transformation stage of the process I still need an element to tell the script where to place the generated table of contents.

Since this is a placeholder element we don't assign attributes or children elements.

```
<!-- place holder for generated toc element -->
<xs:element name="toc"/>
```

Puting it al together: The book element

The last thing to do is to put the structure of the book together. As currently outlined, the structure is metadata, followed by the table of contents placeholder and 1 or more sections where only 1 section element is required.

The idea for making 1 section required and nothing else is that, sometimes, we're writing something quick where I don't really want metadata information or a table of contents. Since the section element already has a title I can get away with using only that for the document.

```
<!-- Base book element -->
<xs:element name="book">
<xs:complexType mixed="true">
<xs:choice maxOccurs="unbounded">
<xs:element ref="metadata" minOccurs="0" maxOccurs="1"/>
<xs:element ref="toc" minOccurs="0" maxOccurs="1"/>
<xs:element ref="section" minOccurs="1" maxOccurs="unbounded"/>
</xs:choice>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
</xs:schema>
```

Converting XML content 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 Antena-House 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 programmatically from a single source file.

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

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.

Where there is a direct equivalent between our mode and the HTML5.1 nightly specification I've quoted the relevant section of the HTML5 spec as a reference and as a rationale of why I've done things the way I did.

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

Creating our conversion style sheets

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 template below, we do the following:

1. Declare the file as an XML document
2. Define the root element of the style sheet (xsl:stylesheet)
3. Indicate the namespaces that we'll use in the document and, in this case, tell the processor to exclude the given namespace
4. Strip whitespace from all elements and keep 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" />
<xsl:template match="*">
<xsl:message terminate="no">
WARNING: Unmatched element: <xsl:value-of select="name()"/>
</xsl:message>

<xsl:apply-templates/>
</xsl:template>
</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 doing it this way as we move down the style sheet.

Now to 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. We include the name and version of our XSLT processor as a meta element
3. In the title we insert the value of the *metadata/title* element
4. In the body we *apply* the templates that match the content inside our document (more on this later)

```
<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)">
<link rel="stylesheet" href="css/styles/docco.css" />
<!-- Load highlight.js -->
<script src="lib/highlight.pack.js"></script>
<script>
hljs.initHighlightingOnLoad();
</script>
</xsl:if>
</head>
<body>
<xsl:apply-templates/>
<xsl:apply-templates select="/" mode="toc"/>
</body>
</html>
</xsl:template>
```

We could build the CSS style sheet and JavaScript files as part of our root template but we chose not to.

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

ult CSS (or no Ces, this method ties the styles for the project to the XSLT stylesheet and requires the XSLT stylesheet designer to remain 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 style sheets. Book designers can work on the CSS, UX and experience designers can work on JavaScript and other CSS areas, book designers can work on the Paged Media style sheets and authors can just write.

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

Table of contents

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 document that will hold the table of content
2. Build the HTML document
3. Create the sectiion and assign its data-type attribute
4. Add the title for the table of contents
5. 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 href attribute
    4. The value of the href attribute (a concatenation of the section's type attribute, the position within the document and the .html string)
    5. The title of the section as the 'clickable' portion of the link

```
<xsl:template match="/" mode="toc">
<xsl:result-document href='toc.html' format="xhtml-out"> (1)
<html> (2)
<head>
<link rel="stylesheet" href="css/style.css" />
<!-- Load Normalize library -->
<link rel="stylesheet" href="css/normalize.css"/>
</head>
<body>
<section data-type="toc"> (3)
<h2>Table of Contents</h2> (4)
<nav>
<ol>
<xsl:for-each select="//section"> (5)
<xsl:element name="li"> (5.1)
<xsl:element name="a"> (5.2)
<xsl:attribute name="href"> (5.3)
<xsl:value-of select="concat(@type, position(),'.html')"/>
(5.4)
</xsl:attribute>
<xsl:value-of select="title"/> (5.5)
</xsl:element>
</xsl:element>
</xsl:for-each>
</ol>
</nav>
</section>
</body>
```

```
</html>
</xsl:result-document>
</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 section 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="section">
<xsl:attribute name="data-type" select="'metadata'"/>
<xsl:apply-templates/>
</xsl:element>
</xsl:template>
```

Section

The section template, on the other hand, is more complex because it has a lot of work to do. It is our primary unit for generating content fifiles, 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 add a section element, the main wrapper for our content.

For the section we 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)">
<link rel="stylesheet" href="css/styles/docco.css" />
<!-- Load highlight.js -->
<script src="lib/highlight.pack.js"></script>
<script>
hljs.initHighlightingOnLoad();
</script>
</xsl:if>
</head>
<body>
<section>
<xsl:if test="string(@type)">
<xsl:attribute name="data-type">
<xsl:value-of select="@type"/>
</xsl:attribute>
</xsl:if>
<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>
<xsl:apply-templates/>
</section>
```

```
</body>
</html>
</xsl:result-document>
</xsl:template>
```

Metadata content

Since the style sheets were oginally written I've added more content to the metadata element to acommodate different publishing requirements. The first additional element I've added is publisher, an organization or person that has a name (pubname element) and an address (modeled after a United States address)

Most of the metadata templates use two conditional statements. The first one tests to see if the author gave the element a class attribute. If there is one, we use that as the value of the class attribute, otherwise we assign a default value based on the element name or its function.

The second test is simple. If there is an ID attribute use it as the value for the ID attribute, otherwise leave it blank.

I coded it this way because of the difference in using classes versus ID values.

Unless you're using the IDs to structure your content there is no real reason to use IDs instead of classes. The consensus is to avoid IDs as much as possible as explained in the CSSLint rule that disallows IDs in selectors

> <xsl:attribute name="class"> <xsl:value-of select="@class"/> </xsl:attribute> & CSS at all) or create their own CSS for the metadata element, the choice is yours.

```
<xsl:template match="publisher">
<xsl:element name="div">
<xsl:choose>
<xsl:when test="string(@class)">
<xsl:attribute name="class" select="@class"/>
</xsl:when>
<xsl:otherwise>
<xsl:attribute name="class" select="'publisher'"/>
</xsl:otherwise>
</xsl:choose>
<xsl:if test="string(@id)">
<xsl:attribute name="id">
<xsl:value-of select="@id"/>
</xsl:attribute>
</xsl:if>
<xsl:value-of select="name"/>
<xsl:apply-templates select="address"/>
```

```
</xsl:element>
</xsl:template>
```

Right now the address is a list of paragraphs with the information of the children elements. We may change the structure to a single paragraphy with span elements to indicate where each child element happens. I'm not 100% sure which way is better.

```
<xsl:template match="address">
<xsl:element name="div">
<xsl:choose>
<xsl:when test="string(@class)">
<xsl:attribute name="class" select="@class"/>
</xsl:when>
<xsl:otherwise>
<xsl:attribute name="class" select="'address'"/>
</xsl:otherwise>
</xsl:choose>
<xsl:element name="p"><xsl:value-of
select="recipient"/></xsl:element>
<xsl:element name="p"><xsl:value-of
select="street"/></xsl:element>
<xsl:element name="p"><xsl:value-of
select="city"/></xsl:element>
<xsl:element name="p"><xsl:value-of
select="state"/></xsl:element>
<xsl:element name="p"><xsl:value-of
select="postcode"/></xsl:element>
<xsl:element name="p"><xsl:value-of
select="country"/></xsl:element>
</xsl:element>
</xsl:template>
```

These elements: releaseinfo, copyright, legal notice, pubdate and abstract share a similar structure so, instead of creating one template we match all of them and we do a little magic. We replace the class name in the otherwise clause and use the name() xpath expression that will match the name of the element that matched the template.

```
<xsl:template match="releaseinfo | copyright | legalnotice |
pubdate | abstract">
<xsl:element name="div">
<xsl:choose>
<xsl:when test="string(@class)">
<xsl:attribute name="class" select="@class"/>
</xsl:when>
<xsl:otherwise>
<xsl:attribute name="class" select="name()"/>
</xsl:otherwise>
</xsl:choose>
```

```
<xsl:apply-templates/>
</xsl:element>
</xsl:template>
```

The revhistory and revisions attribute provides a way to do authority and change tracking. I'm still working on these templates. As they are right now they are paragraphs inside a div inside another div holding the full history.

Normally I'd consider using a table but want to explore other options before I do so.

```
<xsl:template match="revhistory">
<xsl:element name="div">
<xsl:attribute name="class" select="'revhistory'"/>
<xsl:value-of select="revision"/>
</xsl:element>
</xsl:template>

<xsl:template match="revision">
<xsl:element name="div">
<xsl:attribute name="class" select="'revision'"/>
<p><xsl:value-of select="revnumber"/></p>
<p><xsl:value-of select="pubdate"/></p>
<p><xsl:value-of select="authorinitials"/></p>
<p><xsl:value-of select="revnotes"/></p>
</xsl:element>
</xsl:template>
```

For authors we do the following:

1. For each individual in the group we take the first name and the surname
2. Wrap the name around an li element to build an unnumbered list. We can style the list with CSS later

For editors and other roles we do the same thing

1. For each individual in the group we take the first name and the surname
2. We concatenate the type/role to create a full title (production editor for example)
3. Wrap the name and the title with an li element that we can style with CSS later

```
<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')"/>
</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

Headings are primarily used to create sections of content. We use the same heading levels as HTML with the addition of a *title* tag that also maps to a level 1 heading. We've put *title* and *h1* in separate templates to make it possible and easier to generate different code for each heading.

Working with XSLT is 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 or a different result to the one you expected.

According to the spec:

> These elements [h1 to h6] represent headings for their sections.
>
> The semantics and meaning of these elements are defined in the section on headings and sections.
>
> These elements have a rank given by the number in their name. The h1 element is said to have the highest rank, the h6 element has the lowest rank, and two elements with the same name have equal rank.
>
> h1–h6 elements must not be used to markup subheadings, subtitles, alternative titles and taglines unless intended to be the heading for a new section or subsection. Instead use the markup patterns in the Common idioms without dedicated elements section of the specification.
>
> *- 4.3.6 The h1, h2, h3, h4, h5, and h6 elements, Berjon et al. 2013*

All elements have the same attribute set: align, class and id.

The remaining headings, h2 through h6, all have the same attributes and the templates are structured the same way.

```
<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>
```

Blockquotes, quotes and asides

Blockquotes, asides and quotes provide sidebar-like content on our document. According to the W3C:

> The blockquote element represents content that is quoted from another source, *optionally* with a citation which must be within a footer or cite element, and *optionally* with in-line changes such as annotations and abbreviations.
>
> Content inside a blockquote other than citations and in-line changes *must* be quoted from another source, whose address, if it has one, *may* be cited in the cite attribute. [emphasis mine]
>
> *- 4.51 the Blockquote element , Berjon et al. 2013*

The cite HTML provides attribution to the blockquote it is used in. To prevent confusion and to make it's meaning clear the document model uses the *attribution* tag instead, their purpose is identical and during the transformation the attribution will become a *cite* element. According to spec:

> The cite element represents a reference to a creative work. It must include the title of the work or the name of the author (person, people or organization) or an URL reference, which may be in an abbreviated form as per the conventions used for the addition of citation metadata.
>
> *- 4.51 the Cite element , Berjon et al. 2013*

```
<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>
```

The *q* element is the inline equivalent to `blockquote` and has been replaced in our markup by the *quote* element. As stated in the HTML5 specification:

> The q element represents some phrasing content quoted from another source.
>
> Quotation punctuation (such as quotation marks) that is quoting the contents of the element must not appear immediately before, after, or inside q elements; they will be inserted into the rendering by the user agent.
>
> Content inside a q element must be quoted from another source, whose address, if it has one, may be cited in the cite attribute. The source may be fictional, as when quoting characters in a novel or screenplay.
>
> If the cite attribute is present, it must be a valid URL potentially surrounded by spaces. To obtain the corresponding citation link, the value of the attribute must be resolved relative to the element. User agents may allow users to follow such citation links, but they are primarily intended for private use (e.g. by server-side scripts collecting statistics about a site's use of quotations), not for readers.
>
> The q element must not be used in place of quotation marks that do not represent quotes; for example, it is inappropriate to use the q element for marking up sarcastic statements.
>
> The use of q elements to mark up quotations is entirely optional; using explicit quotation punctuation without q elements is just as correct.
>
> *- 4.5.7 The q element, Berjon et al. 2013*

```
<xs:element name="quote">
<xs:complexType mixed="true">
<xs:attribute name="cite" type="xs:anyURI" use="optional"/>
<xs:attributeGroup ref="genericPropertiesGroup"/>
</xs:complexType>
</xs:element>
```

Asides are primarily used for content realted to the main flow of the document. I use it mostly for notes indirectly related to the main content, for example, to explain that there are other ways to generate Schemas apart from W3C's Schema. It is good to know this but it won't change the information in the main content flow.

Per Spec:

> The aside element represents a section of a page that consists of content that is tangentially related to the content around the aside element, and which could be considered separate from that content. Such sections are often represented as sidebars in printed typography.
>
> The element can be used for typographical effects like pull quotes or sidebars, for advertising, for groups of nav elements, and for other content that is considered separate from the main content of the page.
>
> *- 4.3.5 The aside element, Berjon et al. 2013*

```
<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

div and span elements are neutral, they don't have meaning on their onw but they can get their meaning from attributes such as *class*, *data-**, and *id*. Divs are meant as block level elements and siblings or children to sections where `span` is used inline, like a child to our *para* elements.

According to the specification:

> The div element has no special meaning at all. It represents its children. It can be used with the class, lang, and title attributes to mark up semantics common to a group of consecutive elements.
>
> Authors are strongly encouraged to view the div element as an element of last resort, for when no other element is suitable. Use of more appropriate elements instead of the div element leads to better accessibility for readers and easier maintainability for authors.
>
> *- 4.4.14 The div element, Berjon et al. 2013*

```
<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>
<tribute name="id">

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

The span element on its own is meaningless. We can give it meaning with the attributes we pass to it. We can give it a class or id for CSS styling or a type to generate semantic meaning for the contained text.

When we start working on an ePub implementation we can also add the epub:type attribute to create an even more detailed semantic map of our content.

> The span element doesn't mean anything on its own, but can be useful when used together with the global attributes, e.g. class, lang, or dir. It represents its children.
>
> *- 4.5.28 The span element, Berjon et al. 2013*

```
<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

The paragraph is our basic unit of content. Paragraphs are usually represented as blocks of text but they can be styled anyway we choose with the proper CSS.

```
<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

Styles are used to indicate typograhical styles such as strong, emphsis, strikethrough and underline.

```
<xsl:template match="strong">
<strong><xsl:apply-templates /></strong>
</xsl:template>
```

Since we're working with print and visual media only we use only *strong* to indicate bold elements. I've never understood how do strong and *b* work in screen and printed pages or in screen displays.

May have to implement *b* when developing the accessibility component for the document schema.

```
<xsl:template match="emphasis">
<em><xsl:apply-templates/></em>
</xsl:template>
```

As with strong, I've decided to only use *emphasis* to indicate italics and save `*i*` for a future revision when, and if, it becomes necessary.

```
<xsl:template match="strike">
<strike><xsl:apply-templates/></strike>
</xsl:template>
```

Although the strikethrough element has been deprecated in the HTML5 standard, it's still worth having as it can also be the target for CSS that accomplishes the same goal.

The CSS way is to assign a *text-decoration: line-through* instruction to the strike selector.

```
<xsl:template match="underline">
<u><xsl:apply-templates/></u>
</xsl:template>
```

While there is a *u* element it has different semantics than underline. Like strike the correct way to do it is with CSS; in this case using `text-decoration: underlike` for the chosen element.

Links

Links are the essence of the web. They allow you to navigate within the document you're in or move to external documents. I've taken shortcuts and made the label attribute (used for accessibility) and the content of the link the same text. This reduced the ammount of typing we have to do but run the risk of becoming too inflexible.

```
<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 or to specific sections inside a paragraph or to a figure. To do this we need anchors that will resolve to the following HTML:

```
<a id="#target"><a>
```

Since all our structural elements can have an id, it's better to target the id in the element rather than create a separate element to link to. Because we have the facilitie to accomplish the same task I've removed the anchor element from the document model.

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. This makes the class attribute mandatory as we need it to tell highlight.js what syntax library to use

```
<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="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>
<xsl:apply-templates/>
</xsl:element>
</xsl:template>

<xsl:template match="olist">
<xsl:element name="ol">
<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>
<xsl:apply-templates/>
</xsl:element>
</xsl:template>

<xsl:template match="item">
<xsl:element name="li">
<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>
<xsl:value-of select="."/>
</xsl:element>
</xsl:template>
```

Figures and Images

> The figure element represents some flow content, optionally with a caption, that is self-contained (like a complete sentence) and is typically referenced as a single unit from the main flow of the document.
>
> The element can thus be used to annotate illustrations, diagrams, photos, code listings, etc.
>
> A figure element's contents are part of the surrounding flow. If the purpose of the page is to display the figure, for example a photograph on an image sharing site, the figure and figcaption elements can be used to explicitly provide a caption for that figure. For content that is only tangentially related, or that serves a separate purpose than the surrounding flow, the aside element should be used (and can itself wrap a figure). For example, a pull quote that repeats content from an article would be more appropriate in an aside than in a figure, because it isn't part of the content, it's a repetition of the con-

> tent for the purposes of enticing readers or highlighting key topics.
>
> *- 4.4.11 The figure element, Berjon et al. 2013*

Figures, captions and the images inside present a few challenges. Because we allow authors to set height and width on both figure and the image 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, otherwise we use the width of the image inside.

We do the same thing for height in order to avoid squished images of captions that draw over the image because it's too small. If the height of the figure is smaller than the height of the image we use the height of the image, otherwise we use the height of the figure element.

For both height and width we concatenate the attribute value with the string 'px' to make sure that it works in both straight CSS and with Prince and other CSS PDF generators

Alignments can be different, it is possible to have a right-aligned image to live inside a centered container.

Contrary to the HTML specification we use figure only to display images. We have a specialized template to address code blocks for program listings and can create additional elements

```
<xsl:template match="figure">
<xsl:element name="figure">
<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>
<xsl:choose>
<xsl:when test="string(width) and (@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>
<xsl:choose>
<xsl:when test="string(@height) and (@height lt
image/@height)">
<xsl:attribute name="width">
<xsl:value-of select="@height"/>
</xsl:attribute>
</xsl:when>
<xsl:otherwise>
<xsl:attribute name="width">
<xsl:value-of select="image/@height"/>
</xsl:attribute>
</xsl:otherwise>
</xsl:choose>
<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>
```

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="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
type
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

I'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">
<!-- 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.

```
<xsl:template match="book">
<html>
<head>
<meta charset="utf-8"/>
<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>

<!-- Load Typekit Font -->
<script src="https://use.typekit.net/qcp8nid.js"></script>
<script>try{Typekit.load();}catch(e){}</script>
<!-- Paged Media Styles -->
<link rel="stylesheet" href="css/pm-style.css" />
<!-- Paged Media Definitions -->
<link rel="stylesheet" href="css/paged-media.css"/>
<xsl:if test="(code)">
<link rel="stylesheet" href="css/styles/railscasts.css" />
<script src="lib/highlight.pack.js"></script>
<script>
hljs.initHighlightingOnLoad();
</script>
</xsl:if>
</head>
```

```
<body>
<xsl:attribute name="data-type">book</xsl:attribute>
<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.

I'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:

1. Remove filename variable. It's not needed
2. Remove the result document element since we are building a single file with all our content
3. Change way we check for the type attribute in sections. It will now terminate with an error if the attribute is not found
4. 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.
-->
<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 set up the container section and assign title to the data-type attribute. We then apply all children templates.

It is important to note that we've increased the number of children inside metadata so we'll have to be careful in making sure that we're using only one page for the title and another for the rest of the metadata if needed.

```
<!-- Metadata -->
<xsl:template match="metadata">
<xsl:element name="section">
< <para>Now we start working on titlepage</xsl:attribute>
<xsl:apply-templates/>
```

```
</xsl:element>
</xsl:template>
```

Table of contents

The table of content creates anchor links (a href='#id') to the title h1 tags we create in the step below. We can do it this way because XSLT guarantees that all calls to generate-id for a given element (in this case the section/title elements) will return the same value for a given execution.

This code depends on the empty toc placeholder element to place it. If the element is not present then the table of contents will be placed at the end of the document.

```
<xsl:template match="toc">
<section data-type="toc">
<h1>Table of Contents</h1>
<nav>
<ol>
<xsl:for-each select="//section">
<xsl:element name="li">
<xsl:element name="a">
<xsl:attribute name="href">
<xsl:value-of select="concat('#', generate-id(.))"/>
</xsl:attribute>
<xsl:value-of select="title"/>
</xsl:element>
</xsl:element>
</xsl:for-each>
</ol>
</nav>
</section>
</xsl:template>
```

Titles

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

```
<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.

From XML 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.

```
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 }
}

@page preface:left {
@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.

```
/* 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 }

}
```

is>body</emphasis> selecour 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. This is the same footer that we creataed when converting the XML to HTML.

```
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 itself. 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

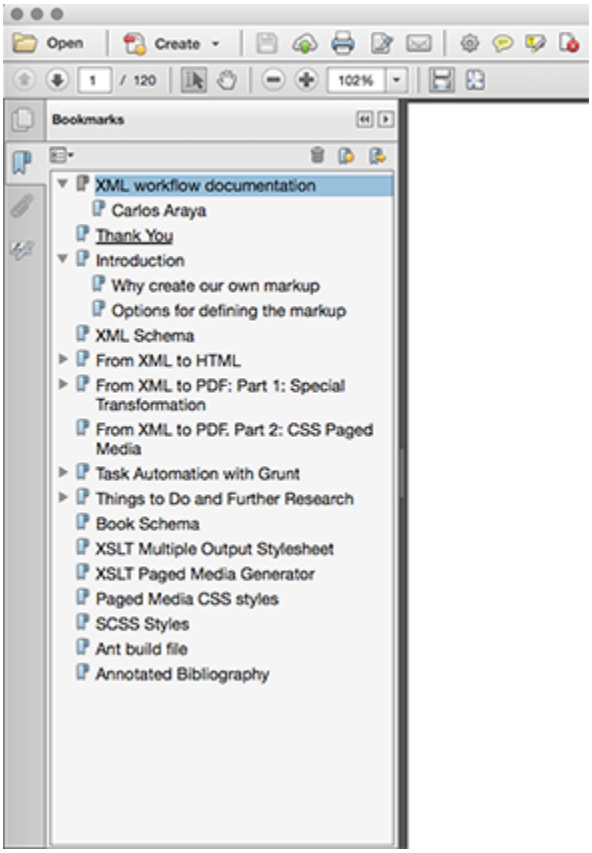

Figure 1:

Exam ple 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.

```
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.

From XML to PDF Part 3: 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 in this post by Paul Irish 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 depending on the font licensing or file size.

```
@import url(http://fonts.googleapis.com/
css?family=Roboto:100italic,100,400italic,700italic,300,700,300
italic,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

Now that we've loaded the fonts we can create our defaults for the document. The `html` element defines vertical overflow and text size adjustment for Safari and Windows browsers.

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

The *body*ling ptor will handle most of the base formatting for the the document.

The selector sets up the following aspects of the page:

- background and font color
- font family, size and weight
- line height
- left and right padding (overrides the base document's padding)
- orphans and widows

```
body {
background-color: #fff;
color: #554c4d;
font-family: 'Roboto', 'Helvetica Neue', Helvetica, sans-serif;
font-size: 1em;
font-weight: 100;
line-height: 1.1;
orphans: 4;
padding-left: 0;
padding-right: 0;
widows: 2;
}
```

Blockquotes, Pullquotes and Marginalia

It's fairly easy to create sidebars in HTML so I've played a lot with pull quotes, blockquotes and asides as a way to move the content around with basic CSS. We can do further work by tuning the CSS

```
aside {
border-bottom: 3px double #ddd;
border-top: 3px double #ddd;
color: #666;
line-height: 1.4em;
padding-bottom: .5em;
padding-top: .5em;
width: 100%;
}

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

The *magin-notes** and *content** move the content to the corresponding side of the page without having to create specific CSS to do so. The downside is that, as with many things in CSS, you are stuck with the provided values and will have to modify them to suit your needs.

```
.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;
}
```

The opening class style creates a large distinguishing block container for opening text. This is useful when you have a summary paragraph at the beginning of your document or some other opening piece of text to go at the top of your document

```
.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;
}
```

Blockquotes present the enclosed text in larger italic font with a solid bar to the left of the content. Because the font is larger I've added a little extra padding to the paragraphs inside the blockquote

```
blockquote {
border-left: 5px solid #ccc;
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;
}
```

The pullquote classes were modeled after an ESPN article and look something like this:

She demurs for all of 15 seconds when asked to tell tales about Krzyzewski, then grabs a chair and joins right in.

The stories, passed along with the pierogies and potato pancakes at the White Eagle, carry with them the idyllic whiff of nostalgia. Stories of a multiplication contest in which ultracompetitive-but-impatient Krzyzewski completed his first but made so many mistakes he lost the coveted Sacred Heart trophy to Kolpak; of Sister Lucinda, the seventh-grade teacher, who gave everyone dashes in the conduct portion of their report course because, as Kolpak remembers, "We were so bad, we didn't even earn an F. Not *bad* bad. We all got straight A's, but we fooled around, passing notes and talking in the back of the room."

A lifetime has passed since those innocent days -- marriages and deaths and children and grandchildren. They've all done well for themselves. Wrobel is a retired mechanical engineer, Stepek a former accountant. Mlynski still works as a financial controller, and Stanislawski is a risk manager.

Figure 1:

Example pullquote

The original was hardcoded to pixels. Where possible I've changed the values to em to provide a more responsive

```
.pullquote {
border-bottom: 18px solid #000;
border-top: 18px solid #000;
font-size: 2.25em;
font-weight: 700;
letter-spacing: -.02em;
line-height: 2.125em;
margin-right: 2.5em;
padding: 1.25em 0;
position: relative;
width: 200px;
}
.pullquote p {
color: #00298a;
font-weight: 700;
text-transform: uppercase;
z-index: 1;
}
.pullquote p:last-child {
line-height: 1.25em;
padding-top: 2px;
}
.pullquote cite {
color: #333;
```

```
font-size: 1.125em;
font-weight: 400;
}
```

Paragraphs

The paragraph selector creates the default paragraph formatting with a size of 1em (equivalent to 16 pixels) and a line height of 1.3 em (20.8 pixels)

```
p {
font-size: 1em;
margin-bottom: 1.3em;
}
```

To indent all paragraphs but the first we use the sibling selector we indent all paragraphs that are the next sibling of another paragraph element (that is: the next child of the same parent).

The first paragraph doesn't have a paragraph sibling so the indent doesn't happen but all other paragraphs are indented

```
p + p {
text-indent: 2em;
}
```

Rather than use pseudo elements (*:first-line* and *:first-letter*) we use classes to give authors the option to use these elements.

```
.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

The only thing we do for list and list items is to indicate what type of list we'll use as our default square for unordered list and Arabic decimals for our numbered lists.

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

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

Figures and captions

The only interesting aspect of the CSS we use for figures is the counter. The *figure figcaption::before* selector creates automatic text that is inserted before each caption. This text is the string "Figure", the value of our figure counter and the string ": ".

This makes it easier to insert figures without having to change the captions for all figures after the one we inserted. The figure counter is reset for every chapter. I'm researching ways to get the figure numbering across chapters.

```
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

Headings are configured in two parts. The first one sets common attributes to all headings: *font-family*, *font-weight*, *hyphens*, *line-height*, margins and *text-transform*.

It's this attribute that needs a little more discussion. Using text-transform we make all headings uppercase without having to write them that way

```
h1,
h2,
h3,
h4,
h5,
h6 {
font-family: 'Roboto Slab', sans-serif;
font-weight: 400;
hyphens: none;
line-height: 1.2;
margin: 1.414em 0 .5em;
text-transform: uppercase;
}
```

In the second part of our heading styles we work on rules that only apply to one heading at a time. Things such as size and specific attributes (like removing the top margin on the h1 elements) need to be handled need to be handled individually

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

h2 {
font-size: 2.369em;
}

h3 {
font-size: 1.777em;
}

h4 {
font-size: 1.333em;
}

h4,
h5,
h6 {
```

```
text-align: inherit;
}
```

Different parts of the book

There are certains aspects of the book that need different formatting from our defaults.

We use the element[attribute=name] syntax to identify which section we want to work with and then tell it the element within the section that we want to change.

For example, in the bibliography (a section with the *data-type='bibliography* <para>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 aragraphs within the bibliography section)

```
section[data-type='bibliography'] p {
text-align: left;
}
section[data-type='bibliography'] p + p {
text-indent: 0 !important;
}
```

The same logic applies to the other sections that we're customizing. We tell it what type of section we are working with and what element inside that sectin we want to change.

```
section[data-type='titlepage'] h1,
section[data-type='titlepage'] h2,
section[data-type='titlepage'] p {
text-align: center;
}

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

Preformatted code blocks

A lot of what I write is technical and requires code examples. We take a two pronged approach to the fenced code blocks.

We format some aspects our content (wrap, font-family, size, line height and wether to do page breaks inside the content) locally and hand off syntax highlighting to highlight.js with a style to mark the content differently.

```
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;
}
```

Miscelaneous classes

Rather than for people to justify text we provide a class to make it so. I normally justify at the div or section level but it's not always necessary or desirable.

Code will be used in a future iteration of the code to highlight inline snippets (think of it as an inline version of the <pre><code> tag combination)

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

Columns

The last portion of the stylesheet deals with columns. I've set up 2 set of rules for 2 and 3 column with similar attributes. In the SCSS source these are created with a column mixin.

```
.columns2 {
column-count: 2;
column-gap: 3em;
column-fill: balance;
column-span: none;
line-height: 1.25em;
width: 100%;
}
.columns2 p:first-of-type {
margin-top: 0;
}
.columns2 p + p {
text-indent: 2em;
}
.columns2 p:last-of-type {
margin-bottom: 1.25em;
}

.columns3 {
column-count: 3;
column-gap: 10px;
column-fill: balance;
column-span: none;
width: 100%;
}
.columns3 p:first-of-type {
margin-top: 0;
}
.columns3 p:not:first-of-type {
text-indent: 2em;
}
.columns3 p:last-of-type {
margin-bottom: 1.25em;
}
```

Converting to ePub

> This chapter is a draft! This style sheet is under development

Perhaps the most ambitious part of the proof of concept is to convert the XML content to an ePub book that would pass epubcheck validation. To accomplish this we'll create another customization layer to acommodate the special syntax and additional content needed to create the ePub 3 package.

This is more complicated than the creation of the single file for generating PDF. The epub specification is more complex and requires a lot more auxiliary files and directories with very specific formatting. I wonder if some of them need to be XML files at all.

Take the *package.opf*, for example. As portable as XML is and as many XML parsers are there in the wild, it would work much better if we do the same code in JSON, YAML, or other similar formats.

Tools and dependencies

Because we use XML we can't just dump our code in the browser or the PDF viewer and expect it to appear just like HTML content.

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 for OxygenXML users: 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 is also available.

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:

```
$ gem install sass
```

If you get an error, you probably need to install the gem as an administrator. 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 and 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.

```
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.
```

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 variable 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.

Setup

```
/*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 (bars reduced in length for formatting.)

```
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.

```
[20:58:14] carlos@rivendell xml-workflow 13902$ 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', dotfiles: true
.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,300,700,300
italic,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,300,700,300
italic,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 repository's 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:dev']
}
},
```

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
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
```

Ideas and outstanding items

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.

This is what the ServiceWorker code looks like in the demo application:

```
// ATHENA DEMO SERVICE WORKER
//
// @author Carlos Araya
// @email carlos.araya@gmail.com
//
// Based on Paul Lewis' Chrome Dev Summit serviceworker.

importScripts('js/serviceworker-cache-polyfill.js');
```

```
var CACHE_NAME = 'athena-demo';
var CACHE_VERSION = 8;

self.oninstall = function(event) {

event.waitUntil(
caches.open(CACHE_NAME + '-v' +
CACHE_VERSION).then(function(cache) {

return cache.addAll([
'/athena-framework/',
'/athena-framework/bower_components/',
'/athena-framework/css/',
'/athena-framework/js/',
'/athena-framework/layouts/',

'/athena-framework/content/',
'/athena-framework/index.html',
'/athena-framework/notes.html',

'http://chimera.labs.oreilly.com/books/1230000000345/
ch12.html',
'http//chimera.labs.oreilly.com/books/1230000000345/apa.html'
]);
})
);
};

self.onactivate = function(event) {

var currentCacheName = CACHE_NAME + '-v' + CACHE_VERSION;
caches.keys().then(function(cacheNames) {
return Promise.all(
cacheNames.map(function(cacheName) {
if (cacheName.indexOf(CACHE_NAME) == -1) {
return;
}

if (cacheName != currentCacheName) {
return caches.delete(cacheName);
}
})
);
});

};

self.onfetch = function(event) {
var request = event.request;
```

```
var requestURL = new URL(event.request.url);

event.respondWith(

// Check the cache for a hit.
caches.match(request).then(function(response) {

// If we have a response return it.
if (response)
return response;

// Otherwise fetch it, store and respond.
return fetch(request).then(function(response) {

var responseToCache = response.clone();

caches.open(CACHE_NAME + '-v' + CACHE_VERSION).then(
function(cache) {
cache.put(request, responseToCache).catch(function(err) {
// Likely we got an opaque response which the polyfill
// can't deal with, so log out a warning.
console.warn(requestURL + ': ' + err.message);
});
});

return response;
});

})
);
};
```

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.

Annotated Bibliography of links and resources