XML WORKFLOW DOCUMENTATION

CARLOS ARAYA

THANK YOU

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

INTRODUCTION

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

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

WHY CREATE OUR OWN MARKUP

I have a few answers to that question:

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

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

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

OPTIONS FOR DEFINING THE MARKUP

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

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

```
<title>Sample Document</title>
<authors>
<author>
<first-name>Carlos</first-name>
<surname>Araya</surname>
</author>
</authors>
</metadata>

<section type="chapter">
<title>Chapter 1</title>

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

</section>

<section type="chapter">

<title>Chapter 2</title>

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

</section>
</book>
```

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

XML SCHEMA

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
<!-- Elements inside section -->
<xs:element name="link">
<xs:annotation>
<xs:documentation>links...</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:attributeGroup ref="genericPropertiesGroup" />
<xs:attribute name="href" type="xs:string" use="required">
<xs:annotation>
<xs:documentation>
Link destination
</xs:documentation>
</xs:annotation>
</xs:attribute>
<xs:attribute name="label" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
```

The link in our resulting book will look like this:

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

and with the optional attributes it will look like this

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

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

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

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

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

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

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

```
<xs:element name="span">
<xs:complexType>
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="type" type="xs:token" use="optional"
default="chapter"/>
</xs:complexType>
</xs:element>
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

They inherit class and ID from *genreicPropertiesGroup*.

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

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

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

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

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

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

Class and ID (from *genericPropertiesGroup*) are optional

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

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

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

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

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

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

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

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

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

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

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

<xs:element name="h2">
<xs:complexType mixed="true">
```

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

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

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

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

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

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

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

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

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

We can have 1 or more *para* elements.

We can use 0 or more of the following elements:

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

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

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

```
<!-- Section element -->
<xs:element name="section">
<xs:complexType mixed="true">
<xs:sequence>
<xs:element name="title" type="xs:token" minOccurs="1"
maxOccurs="1"/>
<xs:choice minOccurs="0" maxOccurs="unbounded">
<xs:element ref="code"/>
<xs:element ref="para" minOccurs="1" maxOccurs="unbounded"/>
<xs:element ref="ulist"/>
<xs:element ref="olist"/>
<xs:element ref="figure"/>
<xs:element ref="image"/>
<xs:element ref="div"/>
<xs:element ref="span"/>
<xs:element ref="blockquote"/>
<xs:element ref="h1"/>
<xs:element ref="h2"/>
<xs:element ref="h2"/>
<xs:element ref="h3"/>
<xs:element ref="h5"/>
<xs:element ref="h6"/>
```

```
</xs:choice>
</xs:sequence>
<xs:attributeGroup ref="genericPropertiesGroup"/>
<xs:attribute name="type" type="xs:token" use="optional"
default="chapter"/>
</xs:complexType>
</xs:element>
```

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

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

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

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

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

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

FROM XML TO HTML

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

WHY HTML

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

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

WHY PDF

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

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

CREATING OUR CONVERSION STYLESHEETS

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

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

In the XSLT example below, we do the following:

1. Declare the file to be an XML document

2. Define the root element of the stylesheet (xsl:stylesheet)
3. Indicate the namespaces that we'll use in the document and, in this case, tell the processor to excluse the given namespaces
4. Strip whitespaces from all elements and preserve it in the code elements
5. Create the default output we'll use for the main document and all generated pages (discussed later)
6. Create a default template to warn us if we missed anything

```
<?xml version="1.0" ?>
<!-- Define stylesheet root and namespaces we'll work with -->
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:epub="http://www.idpf.org/2007/opf"
exclude-result-prefixes="dc epub"
xml:lang="en-US"
version="2.0">
<!-- Strip whitespace from the listed elements -->
<xsl:strip-space elements="*"/>
<!-- And preserve it from the elements below -->
<xsl:preserve-space elements="code"/>
<!-- Define the output for this and all document children -->
<xsl:output name="xhtml-out" method="xhtml" indent="yes"
encoding="UTF-8" omit-xml-declaration="yes" />

<!--
Default template taken from http://bit.ly/1sXqIL8

This will tell us of any unmatched elements rather than
failing silently
-->
<xsl:template match="*">
<xsl:message terminate="no">
WARNING: Unmatched element: <xsl:value-of select="name()"/>
</xsl:message>

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

<!-- More content to be added -->
</xsl:stylesheet>
```

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

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

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

```
<!-- Root template, matching / -->
<xsl:template match="book">
<html>
<head>
<xsl:element name="title">
<xsl:value-of select="metadata/title"/>
</xsl:element>
<xsl:element name="meta">
<xsl:attribute name="generator">
<xsl:value-of select="system-property('xsl:product-name')"/>
<xsl:value-of select="system-property('xsl:product-version')"/>
</xsl:attribute>
</xsl:element>
<link rel="stylesheet" href="css/style.css" />
<xsl:if test="(code)">
<!--
Use highlight.js and docco style
-->
<link rel="stylesheet" href="css/styles/docco.css" />
<!-- Load highlight.js -->
<script src="lib/highlight.pack.js"></script>
<script>
hljs.initHighlightingOnLoad();
</script>
</xsl:if>
<!--
Comment this out for now. It'll become relevant when we add video
<script src="js/script.js"></script>
-->
</head>
<body>
<xsl:apply-templates/>
<xsl:apply-templates select="/" mode="toc"/>
</body>
</html>
</xsl:template>
```

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

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

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

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

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

TABLE OF CONTENTS

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

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

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

1. Create the section and nav and ol elements
2. Add the title for the table of contents
3. For each section element that is a child of root create these elements
    1. The `li` element
    2. The a element with the corresponding href element
    3. The value of the href element (a concatenation of the section's type attribute, the position within the document and the .html string)
    4. The title of the section as the 'clickable' portion of the link

```
<xsl:template match="/" mode="toc">
<section data-type="toc"> (1)
<nav class="toc"> (1)
<h2>Table of Contents</h2>
<ol>
<xsl:for-each select="book/section">
<xsl:element name="li"> (3.1)
<xsl:element name="a"> (3.2)
<xsl:attribute name="href"> (3.2)
```

```
<xsl:value-of select="concat((@type), position(),'.html')"/>
(3.3)
</xsl:attribute>
<xsl:value-of select="title"/> (3.4)
</xsl:element>
</xsl:element>
</xsl:for-each>
</ol>
</nav>
</section>
</xsl:template>
```

METADATA AND SECTION

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

METADATA

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

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

SECTION

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

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

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

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

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

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

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

```
<xsl:value-of select="@class"/>
</xsl:attribute>
</xsl:if>
<xsl:if test="(@id)">
<xsl:attribute name="id">
<xsl:value-of select="@id"/>
</xsl:attribute>
</xsl:if>
<xsl:apply-templates/>
</section>
</body>
</html>
</xsl:result-document>
</xsl:template>
```

METADATA CONTENT

PUBLICATION INFORMATION

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

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

INDIVIDUALS

```
<!-- complex types to create groups of similar person items -->
<xs:complexType name="person">
<xs:annotation>
<xs:documentation>
Generic element to denote an individual involved in creating the
book
</xs:documentation>
</xs:annotation>
<xs:sequence>
<xs:element name="first-name" type="xs:token"/>
<xs:element name="surname" type="xs:token"/>
</xs:sequence>
<xs:attribute name="id" type="xs:ID" use="optional"/>
```

```
</xs:complexType>

<xs:complexType name="author">
<xs:annotation>
<xs:documentation>
Author person
</xs:documentation>
</xs:annotation>
<xs:complexContent>
<xs:extension base="person">
</xs:extension>
</xs:complexContent>
</xs:complexType>

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

<xs:complexType name="otherRole">
<xs:annotation>
<xs:documentation>extension to person to accomodate roles other
than author and editor</xs:documentation>
</xs:annotation>
<xs:complexContent>
<xs:extension base="person">
<xs:sequence minOccurs="1" maxOccurs="1">
<xs:element name="role" type="xs:token"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
```

PEOPLE GROUPS

```
<xsl:template match="metadata/authors">
<h2>Authors</h2>
<ul>
<xsl:for-each select="author">
<li>
<xsl:value-of select="first-name"/>
<xsl:text> </xsl:text>
<xsl:value-of select="surname"/>
</li>
</xsl:for-each>
</ul>
</xsl:template>

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

<xsl:template match="metadata/otherRoles">
<h2>Production team</h2>
<ul class="no-bullet">
<xsl:for-each select="otherRole">
<li>
<xsl:value-of select="first-name" />
<xsl:text> </xsl:text>
<xsl:value-of select="surname" />
<xsl:text> - </xsl:text>
<xsl:value-of select="role" />
</li>
</xsl:for-each>
</ul>
</xsl:template>
```

TITLES AND HEADINGS

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

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

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

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

```
</xsl:attribute>
</xsl:if>
<xsl:value-of select="."/>
</xsl:element>
</xsl:template>
```

BLOCKQUOTES, QUOTES AND ASIDES

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

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

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

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

DIV AND SPAN

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

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

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

PARAGRAPHS

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

STYLES

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

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

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

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

LINKS AND ANCHORS

One of the

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

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

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

The transformation element looks like this:

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

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

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

CODE BLOCKS

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

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

We highlight our code with Highlight.js.

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

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

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

LISTS AND LIST ITEMS

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

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

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

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

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

FIGURES AND IMAGES

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

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

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

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

```
<xsl:template match="figure">
<xsl:element name="figure">
<xsl:if test="(@class)">
<xsl:attribute name="class">
<xsl:value-of select="@class"/>
</xsl:attribute>
</xsl:if>
<xsl:if test="(@id)">
<xsl:attribute name="id">
<xsl:value-of select="@id"/>
</xsl:attribute>
</xsl:if>
<!--
If the width of the figure is smaller than the width of the
```

```
containing image
we may have display problems.

If the width of the containging figure is smaller than the width
of the image,
make the figure width equal to the width of hthe image, otherwise
use the width
of the figure element
-->
<xsl:choose>
<xsl:when test="@width lt image/@width">
<xsl:attribute name="width">
<xsl:value-of select="@width"/>
</xsl:attribute>
</xsl:when>
<xsl:otherwise>
<xsl:attribute name="width">
<xsl:value-of select="image/@width"/>
</xsl:attribute>
</xsl:otherwise>
</xsl:choose>
<!--
We don't care about height as much as we do width, the caption
and image are contained inside the figure.

We only test if it exists. It's up to the author to make sure
there are no conflicts
-->
<xsl:if test="(@height)">
<xsl:attribute name="height">
<xsl:value-of select="@height"/>
</xsl:attribute>
</xsl:if>
<!--
Alignment can be different. We can have a centered image inside a
left aligned figure
-->
<xsl:if test="(@align)">
<xsl:attribute name="align">
<xsl:value-of select="@align"/>
</xsl:attribute>
</xsl:if>
<xsl:apply-templates select="image"/>
<xsl:apply-templates select="figcaption"/>
</xsl:element>
</xsl:template>

<xsl:template match="figcaption">
```

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

<xsl:template match="image">
<xsl:element name="img">
<xsl:attribute name="src">
<xsl:value-of select="@src"/>
</xsl:attribute>
<xsl:attribute name="alt">
<xsl:value-of select="@alt"/>
</xsl:attribute>
<xsl:if test="(@width)">
<xsl:attribute name="width">
<xsl:value-of select="@width"/>
</xsl:attribute>
</xsl:if>
<xsl:if test="(@height)">
<xsl:attribute name="height">
<xsl:value-of select="@height"/>
</xsl:attribute>
</xsl:if>
<xsl:if test="(@align)">
<xsl:attribute name="align">
<xsl:value-of select="@align"/>
</xsl:attribute>
</xsl:if>
<xsl:if test="(@class)">
<xsl:attribute name="class">
<xsl:value-of select="@class"/>
</xsl:attribute>
</xsl:if>
<xsl:if test="(@id)">
<xsl:attribute name="id">
<xsl:value-of select="@id"/>
</xsl:attribute>
</xsl:if>
</xsl:element>
</xsl:template>
```

FROM XML TO PDF: PART 1: SPECIAL TRANSFORMATION

Rather than having to deal with [XSL-FO](http://www.w3.org/TR/2006/REC-xsl11-20061205/), another XML based vocabulary to create PDF content, we'll use XSLT to create another HTML file and process it with [CSS Paged Media](http://dev.w3.org/csswg/css-page-3/) and the companion [Generated Content for Paged Media](http://www.w3.org/TR/css-gcpm-3/) specifications to create PDF content.

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

```
<?xml version="1.0" encoding="iso-8859-1"?> (1)

<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format"> (2)
<fo:layout-master-set> (3)
<fo:simple-page-master master-name="my-page">
<fo:region-body margin="1in"/>
</fo:simple-page-master>
</fo:layout-master-set>

<fo:page-sequence master-reference="my-page"> (4)
<fo:flow flow-name="xsl-region-body"> (5)
<fo:block>Hello, world!</fo:block> (6)
</fo:flow>
</fo:page-sequence>
</fo:root>
```

1. This is an XML declaration. XSL FO (XSLFO) belongs to XML family, so this is obligatory.
2. Root element. The obligatory namespace attribute declares the XSL Formatting Objects namespace.
3. Layout master set. This element contains one or more declarations of page masters and page sequence masters — elements that define layouts of single pages and page sequences. In the example, I have defined a rudimentary page master, with only one area in it. The area should have a 1 inch margin from all sides of the page.
4. Page sequence. Pages in the document are grouped into sequences; each sequence starts from a new page. Master-reference attribute selects an appropriate layout scheme from masters listed inside `<fo:layout-master-set>`. Setting master-reference to a page master name means that all pages in this sequence will be formatted using this page master.

5. Flow. This is the container object for all user text in the document. Everything contained in the flow will be formatted into regions on pages generated inside the page sequence. Flow name links the flow to a specific region on the page (defined in the page master); in our example, it is the body region.
6. Block. This object roughly corresponds to `<div>` in HTML, and normally includes a paragraph of text. I need it here, because text cannot be placed directly into a flow.

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

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

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

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

MODIFYING THE HTML RESULTS

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

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

In this particular situation we want to:

- Add the data-type=book attribute to the body of the document
- Convert the multiple file book into a single file by eliminating the result-document element
- Remove filename variable. It's not needed
- Change the test for type attribute so it'll terminate if it fails (type attribute is required for our implementation of the Paged Media style sheet)
- Add the element that will build our running footer (p class="rh" with the same value as the chapter title)
- Remove the toc mode apply-template call from the book template. It's not needed, we may move it to a separate template for navigation
- Rework the metadata template so it'll match the spec on the CSS (it's a section with data-type = title page)
- Insert a link to the paged media CSS style sheet in the head of the document to make the Prince command easier (and so I won't forget it during testing and development)

Only the templates defined in this stilesheet are overriden

The style sheet is shown below (with large comments removed)

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

Makes the necessary changes to the content to work with the Paged
Media CSS stylesheet
-->
<!-- First import the base stylesheet -->
<xsl:import href="book.xsl"/>

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

<xsl:template match="book">
<html>
<head>
<xsl:element name="title">
<xsl:value-of select="metadata/title"/>
</xsl:element>
<link rel="stylesheet" href="css/pm-style.css" />
<!-- Just so I won't forget it again -->
<link rel="stylesheet" href="css/paged-media.css"/>
<!--
Use highlight.js and docco style
```

```
-->
<link rel="stylesheet" href="css/styles/docco.css" />
<!-- Load highlight.js -->
<script src="lib/highlight.pack.js"></script>
<script>
hljs.initHighlightingOnLoad();
</script>
<script src="js/script.js"></script>
</head>

<body>
<xsl:attribute name="data-type">book</xsl:attribute>
<xsl:element name="meta">
<xsl:attribute name="generator">
<xsl:value-of select="system-property('xsl:product-name')"/>
<xsl:value-of select="system-property('xsl:product-version')"/>
</xsl:attribute>
</xsl:element>
<xsl:element name="meta">
<xsl:attribute name="vendor">
<xsl:value-of select="system-property('xsl:vendor')" />
</xsl:attribute>
</xsl:element>
<xsl:element name="meta">
<xsl:attribute name="vendor-URL">
<xsl:value-of select="system-property('xsl:vendor-url')" />
</xsl:attribute>
</xsl:element>
<xsl:apply-templates/>
</body>
</html>
</xsl:template>

<xsl:template match="book/section">
<section>
<xsl:choose>
<xsl:when test="@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
</xsl:message>
</xsl:otherwise>
</xsl:choose>
<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 name="p">
<xsl:attribute name="class">rh</xsl:attribute>
<xsl:value-of select="title"/>
</xsl:element>
<xsl:apply-templates/>
</section>
</xsl:template>

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

<xsl:template match="book" mode="toc"/>
</xsl:stylesheet>
```

FROM XML TO PDF. PART 2: CSS PAGED MEDIA

This stylesheets sets up a printed stylesheet with a basic set of parameters. It is meant as a starting point for printed media work.

TASK AUTOMATION WITH GRUNT

Figure 1: Grunt.js, a Javascript task runner

Grunt.js is a task runner written in Javascript.

```
/*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
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);

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']
}
}
},

jscs: {
options: {
'standard': 'Idiomatic'
},
all: ['js/']
},
```

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

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

// This task requires the scss-lint ruby gem to be installed on
your system
// If you choose not to install it, comment out this task and the
prep-css
// and work-lint tasks below
//
// 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
}
},
```

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

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

// 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']
}
}
},
```

```
// 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/']
},
```

```
// GH-PAGES TASK
// Push the specified content into the repositories gh-pages
branch
'gh-pages': {
options: {
message: 'Content committed from Grunt gh-pages',
base: './build/app',
dotfiles: true
},
// These files will get pushed to the `
// gh-pages` branch (the default)
// We have to specifically remove node_modules
```

```
src: ['**/*']
},
```

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

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


}); // closes initConfig
```

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

// This task should run last, after all the other tasks are
completed
grunt.task.registerTask(
'generate-pdf',
[
'shell:single',
'shell:prince'
]
);

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


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

THINGS TO DO AND FURTHER RESEARCH

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

DEEP LINKING USING EMPHASIS

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

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

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

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

EXPLORE HOW TO ADD OTHER PARTS OF A BOOK STRUCTURE

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

BUILD MEDIA QUERIES

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

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

USING XSLT TO BUILD NAVIGATION

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

CREATE A BETTER WAY TO GENERATE FILENAMES

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

EXPAND THE USE CASES FOR THIS PROJECT

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

EXPLORE IMPLEMENTING A SERVICEWORKER SOLUTION

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

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

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

LIMITATIONS

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

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

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

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

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

IMPLEMENTING GRAPHIC CALLOUTS

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

It would take the following:

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

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

ANNOTATED BIBLIOGRAPHY

For more information, see: Building books with css3

For another example, see: HTMLbook stylesheet to generate pdf which is the basis for this project.

For the W3C specifications: Paged Media level 3 and Generated Content for Paged Media

Antenna House Formatter Onine Manual

Prince XML User Guide

BESbswy`?´?ˆ?˘BESbswy`?´?ˆ?˘BESbswy`?´?ˆ?˘BESbswy`?´?ˆ?˘BESbswy`?´?ˆ?˘BESb-
swy`?´?ˆ?˘BESbswy`?´?ˆ?˘BESbswy`?´?ˆ?˘BESbswy`?´?ˆ?˘BESbswy`?´?ˆ?˘BESb-
swy`?´?ˆ?˘