Transforming documents with OpenAPI pipelines

Ashley Noel Hinton
ahin017@aucklanduni.ac.nz

Paul Murrell
paul@stat.auckland.ac.nz

Department of Statistics, The University of Auckland

Many technologies now exist for writing a document in common format to be transformed into various formats for sharing. This report proposes that using markup, rather than common markdown languages, is a good fit for writing flexible human- and machine-readable transformable documents. The examples in this report use OpenAPI pipelines to perform the document transformations into various document formats. The pipelines produced show how such a task can be split into various reusable modules.

1 A markup transformable document format

When writing reports and articles for publication in journals, books, online or otherwise, an author will employ one of many document formats to describe the typographical and structural details of her document. Common formats for research publication include the LaTeX system for producing printed documents, and HTML for producing online documents. These and other formats allow an author to have low-level control over a certain type of output, but usually do not offer control over multiple outputs.

The Pandoc document converter is frequently used to allow an author to write a document in one format and then convert to another. For example an author could create an HTML document and use Pandoc to generate a LaTeX version of this document. An author will gnerally write a document using the Pandoc Markdown format, and then use Pandoc to convert the document to one (or several) formats for publication.

Pandoc Markdown is a variant of Markdown, a lightweight markup language written as a simpler way to author HTML documents. Markdown is intended to be readable without conversion, and as such is written so as not to appear to contain markup tags or formatting. While this can makes it easy for an author to write a document with common formatting it

makes it very difficult to have fine-grained control over the eventual output document(s).

As well as needing control over the appearance and structure of a document, an author will often need to embed software code to be executed to produce the final version of the document. Various tools exist for executing code embedded in a literate document to produce the desired output. For example, an author can embed R code in her document and use the Knitr package to execute the R code and produce a final document. An author embeds R code in another document format, for example HTML or LaTeX, and processes the document to produce an output document in the same format.

The R Markdown package combines the dynamic document processing of Knitr with the common authoring format of Pandoc Markdown. An author can write a document in Pandoc Markdown which contains embedded R code to be executed. The author can then execute the code and produce an output document for publication in one of the many formats available through Pandoc document conversion.

It is clear that a document author has many options, and the implementation of R Markdown in the RStudio IDE is an indication that many developers in the R world at least see Markdown as a very useful authoring tool. The following use cases will server to highlight some of the drawbacks to authoring in Markdown. These drawbacks will inform the design of a transformable document solution described in this report.

While Markdown can be very useful for authoring simple documents quickly (Markdown's creators took inspiration from the formatting conventions of plain text emails) an author also makes several sacrifices in choosing Markdown over a markup language like HTML. Two examples of the limitations of Markdown are the creating of lists and tables. List creation in Pandoc Markdown is desrcibed below. Table creation is left as an exercise for the reader.

In Pandoc Markdown a simple list is created by prepending a *, +, or - character to the beginning of each list item, as in the following example:

```
* one
* two
* three
```

If an author wishes to create an embedded list, she must use the four space rule to indent each embedded list.

```
* outer list 1
    - inner list 1
    - inner list 2
* outer list 2
```

At even two levels this process is already quite complex. As list membership depends on whitespace it can be a frustrating exercise trying to make changes to a complex list, let alone author a list in the first place.

In contrast, a author using HTML markup can indicate an unordered list using a`<ul>` element, which `<li>` elements for each item, as in the following code:

```
<ul>
  <li>one</li>
  <li>two</li>
  <li>three</li>
</ul>
```

Similarly, an embedded list simply nests the same structure inside a list structure, as in the following code:

```
<ul>
  <li>outer list 1
    <ul class="subList">
      <li>inner list 1</li>
      <li>inner list 2</li>
    </ul>
  </li>
  <li>outer list 2</li>
</ul>
```

The author does not have to count white space, and it is trivial to make changes to this list structure.

Markdown like HTML also gives a document author more control over the poutput than Markdown. The author of the HTML list example above has used the `class` attribute to indicate the inner list belongs to the subList class. Using `class` an author can apply output styles or perform other actions on subsets of elements. When a document authored in Markdown is transformed to HTML its lists will be marked up using the same HTML list tags as above, but an author of a Markdown document does not have access to these `class` attributes for customising output. She could include raw HTML in her Pandoc Markdown document, but this limits the output types to those using HTML.

A document author has various methods for embedding chunks of code in her document. For example, an author of a Knitr HTML or LaTeX document can enclose code chunks to be executed in specially formatted comments in the respective document languages. For example, a document author can embed and R code in a Knitr HTML document as in the following code:

```
<!-- begin.rcode
x <- rnorm(n = 10)
plot(x)
end.rcode-->
```

Similarly an author can embed R code in a Knitr LaTeX document as in the following code:

```
%% begin.rcode
% x <- rnorm(n = 10)
% plot(x)
%% end.rcode
```

A document author using the R Markdown package can enclose R code chunks in special fenced code blocks as in the following code:

````
```{r}
x <- rnorm(n = 10)
plot(x)
```
````

While the use of these methods for including code makes it quick and easy to write a document it makes it more difficult for an author to do extra processing to chunks of code before producing the a final document. In contrast, an author creating an HTML document might wrap R code in `<code>` elements as in the following code:

```
<code class="R">
  x <- rnorm(n = 10)
  plot(x)
</code>
```

If an author marked up code chunks in this fashion she could, for example, make use of tools which employ the XPATH query language to locate `<code>` elemenents and perform transformations. If the author gives R code chunks the class R, as in the above example, she could perform transformations on just the R code chunks.

If an author uses Pandoc Markdown to write a document she can include raw HTML or raw TeX language elements to control the document output. These raw code sections are only processed by Pandoc when creating the associated output formats, and would otherwise be ignored. There is no simple method for creating custom sections or formats within Pandoc Markdown.

While an author using HTML is also unable to expect new and custom elements to be recognised by a web browser, the fact that HTML is a form

of XML means an author can invent her own XML elements for document writing. These custom elements could then be processed using an XML transformation tool like XSL Transformations to convert the custom elements to valid HTML code. A markup document format has the benefit of providing a simplified authoring format without sacrificing fine control when required.

Markdown has proven itself to be very useful for document authors, and it is not the suggestion of this report that a markup format replace Markdown entirely. Rather this report proposes that in situation where Markdown is not powerful enough for a document author a markup format like the one described in the next section might provide the solution. Importantly, a well designed markup document should allow an author to recover a Markdown document as *output*, thus providing a readable plain text document. While a format like Markdown is designed to satisfy a set of *known* transformations a format based on markup can also satisfy future *unknown* transformations, e.g. extracting subsets of elements.

2 The `document` markup format

The transformable document format described in this report is an XML file with `document` as the root element. This document has two child elements: `metadata` and `body`.

The `metadata` element contains the document metadata, with elements for the document `title` and `subtitle`, `author` information, `date` of publication, and a `description` section. An example `metadata` element follows:

```
<metadata>
  <title>Today should be a holiday</title>
  <author>
    <name>Ashley Noel Hinton</name>
    <email>ahin017@aucklanduni.ac.nz</email>
  </author>
  <date>25 December 2015</date>
</metadata>
```

The `body` element contains the document's main content. The following elements are used in the same way as they are used in HTML (`https://www.w3.org/TR/html-markup/elements.html`):

- `a` – hyperlink
- `code` – code fragment
- `em` – emphatic stress

- `figcaption` – figure caption
- `figure` – figure with optional caption
- `h1` – heading
- `h2` – heading
- `h3` – heading
- `img` – image
- `li` – list item
- `ol` – ordered list
- `p` – paragraph
- `pre` – preformatted text
- `q` – quoted text
- `section` – section
- `strong` – strong importance
- `ul` – unordered list

The `<url>` element is introduced in the `document` format to indicate a hyperlink where to enclosed URL is both the href and the value. The following code block demonstrates the use of the `url` element:

```
<ul>
  <li>modular</li>
  <li>reusable</li>
  <li>shareable</li>
  <li><url>https://github.com/anhinton/conduit</url></li>
</ul>
```

The resulting output:

- modular
- reusable
- shareable
- `https://github.com/anhinton/conduit`

The `document` XML format uses `<code>` elements to indicate blocks of computer code, just as in HTML. Dynamic code chunks which are to be executed are marked using the `class` attribute to `code`. For example chunks of R code which are to be executed used the Knitr package are wrapped in a `<code>` element with `class="knitr"`. An author can also provide a `name` attribute for the knitr code chunk, as well as knitr `options`. A document author can also use `CDATA` sections to wrap code with reserved XML characters. The following code demonstrates how to include an R code chunk to be executed with Knitr:

```
<code class="knitr" name="knitrDemo" options="tidy=FALSE"><![CDATA[x <- rnorm(n = 10
mean(x)]]></code>
```

And the following is the result of executing this code chunk:

```
x <- rnorm(n = 10)
mean(x)
```

```
## [1] 0.06679083
```

The `document` format also makes use of the `include` element from XInclude (`http://www.w3.org/2001/XInclude`) namespace to include XML content from external files. This allows `document` authors to embed other documents which may be authored separately from the main document. There is no simple method of doing this directly in either HTML or Pandoc Markdown.

The next sections describes some simple transformations which can be performed on the `document` markup format using freely available open source tools. This report was itself written in the `document` markup format—the source code is available at `report.xml`. replace with final full URL

3 Transforming the `document` markup format

This section describes how freely available open source tools can be used to transform the `document` markup format in different ways. The examples include creating an HTML document, preparing R code chunks for executing using Knitr, processing XInclude elements, and some steps towards creating a PDF document. The command line tools `xsltproc` is used in the following examples, but many other tools are available to do the transformations described.

3.1 Transforming to HTML

The `document` markup format can be easily transformed into HTML using XSL Transformations (`https://www.w3.org/TR/1999/REC-xslt-19991116`).

XSLT stylsheets are XML documents which describe how another XML document can be transformed. They can be used to produce new XML documents, such as HTML, or other plain text formats. The command line XSLT processor `xsltproc` (`http://www.xmlsoft.org/`) can be used to apply an XSLT stylesheet to an XML document to produce an HTML output document.

As a large part of the `document` format is based on HTML already we do not have to desrcibe very many transformations in our XSLT stylesheet. We will need to transform the `metadata` section of the `document` to HTML `head` elements and an appropriate title section. We also need to transform our custom `<url>` elements to HTML hyperlinks.

The full XSLT stylesheet used in this example can be found at `examples/documentToHtml.xsl`. The XSLT code used to transform `url` elements to HTML hyperlinks is as follows:

```
<xsl:template match="url">
  <xsl:element name="a">
    <xsl:attribute name="href">
      <xsl:value-of select="node()"/>
    </xsl:attribute>
    <xsl:value-of select="node()"/>
  </xsl:element>
</xsl:template>
```

The source `document examples/toHtml.xml` contains the following XML:

```
<?xml version="1.0" encoding="UTF-8"  ?>
<document>
  <metadata>
    <title>Today should be a holiday</title>
    <author>
      <name>Ashley Noel Hinton</name>
      <email>ahin017@aucklanduni.ac.nz</email>
    </author>
    <date>25 December 2015</date>
  </metadata>

  <body>
    <p>For many years I have believed that 25 December should be a
    public holiday, and I am now prepared to provide evidence for
    this.</p>

    <ol>
      <li>There aren't any other holidays in December.</li>
```

```
      <li>Schools are usually closed anyway.</li>
    </ol>

    <p>More information can be found at
    <url>https://en.wikipedia.org/wiki/December_25</url>.</p>
  </body>
</document>
```

This document can be transformed to HTML using the following call to `xsltproc`:

```
xsltproc -o examples/toHtml.html examples/documentToHtml.xsl examples/toHtml.xml
```

The resulting HTML document can be viewed at `examples/toHtml.html`.

3.2 Process `xi:include` elements

The `document` markup format uses XInclude elements (`http://www.w3.org/2001/XInclude`) to embed text from external documents. These elements can be processed by the command line tool `xsltproc` (`http://www.xmlsoft.org/`) can process the embedded documents into the output document.

We will use the same XSLT stylesheet we used in the previous example. The source `document examples/processXinclude.xml` contains the following XML:

```
<?xml version="1.0" encoding="UTF-8"  ?>
<document xmlns:xi="http://www.w3.org/2001/XInclude">
  <metadata>
    <title>Today should be a holiday</title>
    <author>
      <name>Ashley Noel Hinton</name>
      <email>ahin017@aucklanduni.ac.nz</email>
    </author>
    <date>25 December 2015</date>
  </metadata>

  <body>
    <p>For many years I have believed that 25 December should be a
    public holiday, and I am now prepared to provide evidence for
    this.</p>

    <xi:include href="evidenceList.xml" parse="xml"/>
```

```
    <p>More information can be found at
    <url>https://en.wikipedia.org/wiki/December_25</url>.</p>
  </body>
</document>
```

The element `<xi:include href="evidenceList.xml" parse="xml"/>` indicates that the XML included in `examples/evidenceList.xml` is to be included in the output document.

We will add the `--xinclude` tag to our call to `xsltproc` to process the XInclude elements when we do our transformation:

```
xsltproc --xinclude -o examples/processXinclude.html examples/documentToHtml.xsl  ex
```

The resulting HTML document can be viewed at `examples/processXinclude.html`.

3.3 Subsetting elements: prepare R code chunks for Knitr

The Knitr package lets document authors embed chunks of R code in special comment code and execute these chunks to produce an output document. The use of comments to indicate code makes it difficult to perform custom actions on R code marked up in this way. The `document` markup format wraps chunks of R code to be executed by Knitr in `<code class="knitr">` elements. This allows an author using the `document` markup format to perform any operations she likes on chunks of R code. An XSLT stylesheet can be used to transform chunks of code marked up in this fashion into Knitr R code chunks in a Knitr HTML document.

The full XSLT stylesheet used in this example can be found at `examples/documentToRhtml.xsl`. The XSLT code used to transform `<code class="knitr">` elements to Knitr R code chunks is as follows:

```
<xsl:template match="code[@class='knitr']">
  <xsl:comment><xsl:text>begin.rcode </xsl:text><xsl:value-of select="@name"/><xsl:
  <xsl:text>&#xA;</xsl:text>
  <xsl:value-of select="node()"/>
  <xsl:text>&#xA;end.rcode</xsl:text></xsl:comment>
</xsl:template>
```

The source `document examples/knitrChunk.xml` contains the following XML:

```
<?xml version="1.0" encoding="UTF-8"  ?>
<document>
  <metadata>
    <title>Plotting in R</title>
```

```
    <author>
      <name>Ashley Noel Hinton</name>
      <email>ahin017@aucklanduni.ac.nz</email>
    </author>
    <date>25 December 2015</date>
  </metadata>

  <body>
    <p>A plot to celebrate 25 December:</p>

    <code class="knitr"><![CDATA[x <- rnorm(n = 10)
plot(x)]]></code>

    <p>More information can be found at
    <url>https://en.wikipedia.org/wiki/December_25</url>.</p>
  </body>
</document>
```

This document can be transformed to Knitr HTML using the following call to `xsltproc`:

```
xsltproc -o examples/knitrChunk.Rhtml examples/documentToRhtml.xsl examples/knitrChu
```

The resulting Knitr HTML document can be viewed at `examples/knitrChunk.Rhtml`.

3.4 Extended transformations

The previous three examples have shown single transformations on documents written in the `document` markup format. Authors of `document` files are not limited to just one transformation, however. In the previous example demonstrated how a module author can convert a document with marked up chunks of R code into a document for processing with the Knitr package in R. The Knitr HTML document produced by the previous example can be converted to HTML using the following code in R:

```
library(knitr)
oldwd <- setwd("examples")
knit(input = "knitrChunk.Rhtml")
```

```
##
##
## processing file:  knitrChunk.Rhtml
```

```
##
  |
```

```
  |                                                                      |   0%
  |
  |......................................................................| 100%
##   ordinary text without R code

## output file:  knitrChunk.html

## [1] "knitrChunk.html"

 setwd(oldwd)
```

The resulting HTML document can be viewed at `examples/knitrChunk.html`.

This output document is the result of two transformation steps, using two different tools (`xsltrproc`, and the Knitr package in R), each producing an output document. The `document` format can be used in this wat to authoring documents which require several transformation steps, and several transformation tools. The result of each transformation can be provided as a source document for the following transformation. In the following discussion we discuss how a `document` author might manage multiple transformations on a `document`.

4 Discussion

The `document` markup format provides a reasonably simple authoring format in which an author can write documents for one or several output formats. Like Markup, the `document` format allows an author to target HTML and PDF as outpu formats, among others. Unlike

5 Technical requirements