Technical Writing Samples

Contents

Working with MDITA

With the advent of the upcoming Lightweight DITA (LwDITA) standard, the advantages of the docs-as-code approach can be brought to DITA. LwDITA is a simplified form of the DITA standard with only 48 tags in total and comes in 3 flavors:

- XDITA - a subset of the main DITA standard XML elements.
- HDITA - A DITA variant written in HTML5 rather than XML.
- MDITA - A DITA variant written in Markdown rather than XML. This is based on GitHub-flavored Markdown but incorporates MultiMarkdown features for tables.

MDITA is the real game changer for a docs-as-code approach for DITA but we will need all 3 variants for a complete solution.

As with normal Markdown, you can include HTML tags, and in this case HDITA tags can be used in MDITA files. The use of HDITA tags allows important DITA features such as content referencing and content filtering. Use of HDITA tags is does not add sufficient complexity to the MDITA source that they interfere with ease of editing for non-DITA savvy reviewer.

As with full-fat DITA, MDITA files are organized into maps. Although MDITA maps can be written in markdown, there is little or no tool support for this yet. Therefore maps written usign standard DITA or XDITA are used. Maps provide the organizing principle - the navigation table of contents - for a documentation site.

Specialized HDITA files holding key definitions or content reference content used in build processing only can also be added to maps along with the MDITA files.

Writing and Reviewing MDITA

MDITA *is* Markdown and so learning how to use it is much simpler than learning how to use full-blown DITA.

Writers may need a little more training on how to incorporate DITA mechanism like keys, conrefs and ditaval filters when writing in MDITA but a couple of hours at most will train any new writer with no knowledge of Markdown or DITA how to write effective MDITA.

Writers would still probably be advised to use an XML editor especially to validate any XDITA or HDITA tags they use. Reviewer however should be able to use their favorite text editor to review and make updates, up load to their VCS and issue a pull request just as if they were editing source code.

Normally, to get DITA content reviewed, writers had to transform it into another format such as PDF and circulate that to a reviewer for comments. Then, use those comments to manually update the DITA source files. Or, technical publication departments would have to invest in online XML editors like Oxygen Web Author or Xeditor, that provide a WYSIWYG front-end that hides the XML complexity from reviewers. With MDITA, DITA writers can enjoy the same easy review workflow as writers in Markdown.

MDITA topic syntax

The DITA Open Toolkit website provides a cheat-sheet for MDITA mark-up: MDITA syntax reference. Here I've provided a brief résumé of the most commonly used syntax items.

MDITA is based on CommonMark.

Headings

As DITA assigns heading level by position in the map, it's not a good idea to have more than one heading generally in an MDITA topic. If you want to have sections in your MDITA topic, assign a level 2 heading and add the *.section* class value in curly brackets as in the following example:

```
# Topic title

## Section title {.section}
```

Do not include any level three or lower headings in a MDITA topic. You can only 1 level 1 heading in a topic but you can have as many level 2 sections as you like, although you **cannot** have sections within sections.

If you a complex multi-level heading topic, break it up into smaller topics and assign the required heading levels in the map.

Paragraphs

Paragraphs are separated by 2 returns as in most flavors of Markdown

The first paragraph in an MDITA file is treated (in traditional DITA terms) as a *shortdesc* element by transform plugins.

Links

Links use the following format:

```
[Markdown](test.md)
[DITA](test.dita)
[HTML](test.html)
[External](http://www.example.com/test.html)
```

However if you are refering to a markdown file in the same document, and that file has a key defined for it in the ditamap, you can just refer to the key within square brackets. For example:

```
For more information, see [using-maps].
```

When the document is built the key is replaced by the title of the topic. The same applies for keys defined in an external links map. For more information, see Using keys on page 12.

Images

Images can be inline as well as on their own line and can be given titles and *alt* content.

```
An inline ![Alt](test.jpg).

![Alt](test.jpg)

![Alt](test.jpg "Title")
```

Inline elements

The following inline elements are possible:

```
**bold** or __bold__
*italic* or _italic_
```

Note that strikethough (as in GitHub-flavoured Markdown) is not permitted. Underline, codephrase, subscript, and superscript can added by using the relevant HDITA tags with a HDITA snippet:

```
<p>This is an <u>underline</u>.</p>
<p>This is an <sub>subscript</sub>.</p>
<p>This is an <sup>superscript</sup>.</p>
```

Alerts

There is no specific MDITA markup for Notes/Alerts/Callouts. Use the HDITA <*note*> tag wrapped in a <*p*> tag. Use the *type* attribute to define different types of alert. The options are given below:

```
<p><note>This is a note</note></p>
<p><note type="caution">This is a caution </note></p>
<p><note type="danger">This is a danger</note></p>
<p><note type="notice">This is a notice</note></p>
<p><note type="trouble">This is a trouble</note></p>
<p><note type="warning">This is a warning</note></p>
```

Lists

Multi-level ordered and unordered lists are possible.

Unordered lists:

```
- Item 1
- Item 2
- Item 3

* Item 1
* Item 2
* Item 3

* Item 1
    * 2nd Item 1
* Item 2
* Item 3
    - 2nd level item 1
    - 2nd level item 2
* Item 4
```

- Item 1
- Item 2
- Item 3

Example with asterisks:

- Item 1
- Item 2
- Item 3

Example with 2nd level lists items:

- Item 1
    - 2nd Item 1
- Item 2
- Item 3
    - 2nd level item 1
    - 2nd level item 2
- Item 4

Ordered lists:

```
1. Item 1
1. Item 2
1. Item 3
    1. 2nd level item 1
    1. 2nd level item 2
    1. 2nd level item 3
1. Item 4
```

1. Item 1
2. Item 2
3. Item 3
    1. 2nd level item 1
    2. 2nd level item 2
    3. 2nd level item 3
4. Item 4

Tables

MDITA also incorporates MultiMarkdown or GitHub-flavoured Markdown style tables:

```
| First Header | Second Header | Third Header |
| ------------ | :-----------: | -----------: |
| Content      | *Long Cell*                 ||
| Content      | **Cell**      | Cell         |
```

First Header	Second Header	Third Header
Content	*Long Cell*	
Content	**Cell**	Cell

Admittedly MDITA, like any flavor of Markdown, is weak when it comes to tables. Markdown tables must be short and simple or they become virtually impossible to read. Reviewing Markdown content that content long and/or complex tables is very difficult and defeats the purpose.

Warning: If you have a documentation that contains many tables then Markdown-based systems are probably less useful. A more robust system like standard DITA is more appropriate in that case.

Video

Generally you cannot add videos to a Markdown file but by adding the HDITA HTML5 tags to the file you can access video:

```
Check out the video below for more information:

<video title="My New Video" controls autoplay loop muted>
    <source src="media/vid1.mp4" poster="vid1.jpg"/>
</video>
```

Note that you will need to provide an image to be displayed in place of the video for the user to access it.

Escaping special characters

Special characters are escaped using a backslash.

MDITA frontmatter

Frontmatter refers to the content in YAML format that can be placed at the start of an MDITA file. This content is essentially metadata that is useful when converting to other formats. This content, for example, is passed to the META tags in the HEAD of an HTML file. The content is also useful is the markdown file is ever converted to standard DITA XML.

Frontmatter is delineated by 3 dashes above and below and must be the first thing in the topic. There can be no spaces or empty lines before the frontmatter. For example:

```
---
author: Michael
---

# MDITA frontmatter

Frontmatter refers to the content in YAML format that can be placed at the start of an
 MDITA file.
This content is essentially metadata that is useful when converting to other formats.
This content, for example, is passed to the META tags in the HEAD of an HTML file.
The content is also useful is the markdown file is ever converted to standard DITA XML.

...
```

For our purposes, only 2 items of front maatter are required: an *id* which serves as a unique identifier for the MDITA file, and the *author* so the writer of the topic can be identified i nthe future. The value of the *id* should be used also as key for the topic when it is added to the ditamap.

Other possible frontmatter items include:

- publisher
- source
- permissions
- audience
- category
- keywords
- resourceid

HTML5 tags in MDITA topics

HDITA is the HTML5 flavor of LwDITA. Entire topics can be written in HDITA of course but we are interested here in those tags that can be used to augment functionality in MDITA topics. The following HDITA/HTML5 tags can be used in MDITA:

- **** - to italicize text.
- **** - to embolden text.
- **<u>** - to underline text (please avoid using underlining).
- **<dl>** - a definition list
- **<dt>** inside **<dl>** - a definition term.
- **<dd>** inside **<dt>** - a definition description.
- **<figure>** - a container for an image.
- **<img>** - an image.
- **<ol>** - to introduce an ordered list.
- **<ul>** - to introduce an unordered list.
- ** ** inside **<ul>** or **<ol>** - a list item.
- **<p>** - a paragraph. Especially important when using context references.

- **** - a span of text. Used mostly for key references within notes and context referenced text.
- **<pre>** - to indicate code or teletype text.
- **<section>** - To designate a new section in HTML content.
- **<a href>** - an anchor tag for links.
- **<sub>** - for subscript content.
- **<sup>** - for superscript content.
- **<audio>** - to embed audio content.
- **<video>** - to embed video content.
- **<source>** inside **<audio>** or **<video>** - to indicate the path to a video or audio source file.
- **<table>** - a table
- **<tr>** inside **<table>** - a table row.
- **<th>** inside **<tr>** - a table header row.
- **<td>** inside **<tr>** - a table cell.

Each tag also has one or more attributes but please refer to the latest LwDITA specification doc for those details.

Many of the tags do have Markdown counterparts so why would you use them? Well, the short answer is for encapsulating text you want to reuse or filter. You can find out about that in more detail further on in Using content references with MDITA on page 15 and Using filters with MDITA on page 16.

When to use HTML5 tags in MDITA files

There are 4 main occasion when you need to include HTML code in your MDITA markdown content:

1. For notes/alerts/admonitions (for more information, see MDITA topic syntax on page 5).
2. For context reference (conref) content pulled in from another file (for more information, see Using content references with MDITA on page 15).
3. For text to which a filter is to be applied (for more information, see Using filters with MDITA on page 16).
4. For embedding video (for more information, see MDITA topic syntax on page 5.

Organizing topics with maps

A collection of MDITA topics is organized using a ditamap. Ditamaps are XML files that function like a table of contents to determine the order and the heading level of a topic within a document. Ditamaps are written in XML.

Although the Lw-DITA specification does allow you to create ditamaps in MDITA, it is more practical to use XDITA or standard DITA for the following reasons:

- Ditamaps are not something that you need reviewers to update so there is no need to keep them in Markdown.
- Ditamaps can be outputted in Markdown format as an *index.md* file by the DITA Open Toolkit Markdown transform so there is no requirement to keep the map in Markdown.

As with standard DITA, you can have a root map that can contain content submaps as well as external links or key definition maps that have a processing-only role. As is also standard best practice, you can add "warehouse" topics to hold images and content that will used in multiple locations and accessed via the context reference (or "conref") mechanism.

A sample ditamap:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="intro-to-MDITA">
    <title>DITA-as-code<title>
    <topicref format="markdown" href="Dita-dac.md" keys="ddac"/>
    <topicref format="markdown" href="DITA4dac.md" keys="d4dac">
        <topicref format="markdown" href="write-review-MDITA.md" keys="write-MDITA"/>
        <topicref format="markdown" href="Ditamaps4dac.md" keys="MDITA-maps"/>
        <topicref format="markdown" href="MDITA-keys.md" keys="MDITA-keys"/>
        <topicref format="markdown" href="MDITA-conrefs.md" keys="MDITA-conrefs"/>
        <topicref format="markdown" href="MDITA-filters.md" keys="MDITA-filters"/>
        <topicref format="markdown" href="publish-MDITA.md" keys="publish-MDITA"/>
    </topicref>
    <mapref href="externallinks.ditamap" keys="extlinksmap" processing-role="resource-
only"/>
    <mapref href="keydefs.ditamap" keys="keydefsmap" processing-role="resource-only"/>
    <topicref format="markdown" href="conrefs.md" keys="conrefs" processing-
role="resource-only"/>
    <topicref format="mardown" href="images.md" keys="images" processing-
role="resource-only"/>
</map>
```

Ditamap tags

The first tag provides information the version of XML used. The second tag references the Document Type Definition (DTD) for the map. This DTD defines what XML tags can appear in a map and in what order. These first 2 tags appears in all ditamaps of this type and, luckily, we don't ever really need to concern ourselves with them.

The other tags, though are of interest to us:

- **map** - The <map> tag contains all the other tags and always requires an *id* attribute as a unique identifier for the map.
- **title** - The <title> tag gives the document as a whole its title.
- **topicref** - <topicref> tags are used to refer to topics. <topicref> tags can contain other <topicref> tags. The following attribute can be used with the <topicref> tag:
    - *href* - The *href* attribute contains the path to the MDITA topic.This attribute is mandatory.
    - *format* - This attribute defines the format of the file referenced. For our purposes, this will usually have a value of *markdown*. This attribute is mandatory.
    - *keys* - The *keys* attribute contains a value that functions as an alternative name when refering to the topic. Keys in general are discussed at greater length in Using keys on page 12, but bascially using a key allows you to change the name or location of a topic (in the *href* attribute) without breaking any links or references to it elsewhere in the document.
    - *processing-role* - The *processing-role* attribute is only used for certain special topics (and maps) that usually containing content that is reused elsewhere in the document. For that reason you do not want that topic to appear *as-is* in the document. The *processing-role* attribute usually has a value of *resource-only*.
- **mapref** - You can also references other maps within your map. The <mapref> is used to reference whole maps. It uses the same attributes as the <topicref> tag, but the *format* attribute is not mandatory as the format will always be *ditamap*.

There are quite a few other tags and attributes that can be used in a map (see Using keys on page 12 for some more of those). The 4 tags listed above are the ones that are used on a day-to-day basis when you work with maps.

Types of map

The basic type of map shown here is usually called a *root* map. It is used to define the structure of the document. However, there are other types of map. To make full use of MDITA's content reuse

functionality, you can also employ key defintion maps and external link maps. These maps are usually sub-maps within the root maps and are used to define variables and store external links, respectively. Again see Using keys on page 12 for more information on key definition and external links maps.

New functionality for Markdown

MDITA extends what markdown can do. We've already seen how you can add videos to your Markdown content in MDITA topic syntax on page 5, but the real power of MDITA lies in the DITA content reuse features that it brings to Markdown content.

- Text variables using key definitions to create variable text that allows you update keywords in your document in one place and have the changes reflected throughout the document immediately. For more detail on this, see Using keys on page 12.
- Key references that allow you assign a new name to a file or external resource that you can use when referencing that resource. You can happily change the file name, location, or URL in a ditamap without breaking any links in the document. For more detail on this, see Using keys on page 12.
- Content references that you can use to refer to content in multiple places in the document without having multiple copies of the content. If the content ever needs updated, it can be done so in one place and the changes automatically reflected everywhere else in the document.For more detail on this, see Using content references with MDITA on page 15.
- Text filtering that you can deploy to exclude elements of text from a build to produce different versions of a document for diffrent purposes (perhaps different audiences, releases, software platforms etc.).For more detail on this, see Using filters with MDITA on page 16.

Using keys

The use of keys in root maps is not mandatory but it is best practice. Keys can play an important role in:

- Managing internal links so that path and names of topics in a map can be changed without breaking cross references.
- Managing external links so that links to external resources that are used more than once in a document can be more easily updated if the URL changes.
- Creating text variables for things like company and product names so that these can be easily and safely updated everywhere in the document if they change, without resorting to risky global find and replace commands, or painful manual updating.

Using keys for internal links

By adding a *keys* attribute value to a topic's <topicref> in your map, you can refer to the key value within square brackets in any topic and the output generates a link to the topic with the topic title. For example, you link to this topic in the following way:

```
For more information, see [using-maps].
```

The key *using-maps* is the value of the *keys* attribute for the topic reference for this topic in the current document ditamap.

By using this method, you can change the name or the path to the markdown file referenced in the *href* attribute in your <topicref> without breaking any references to the file in the document itself.

Using keys for external links

As you have no control over the URL of an external link, and you might use the same external link multiple times in a document, it is best practice to keep your external links in their own ditamap and refer to them using keys (just as you would for internal links). Here is a sample external links ditamap:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="extlinks">
    <title>External links</title>
    <topicref format="html" href="https://www.commonmark.org" keys="commonmark"
 scope="external" >
        <topicmeta>
            <navtitle>CommonMark specification</navtitle>
        </topicmeta>
    </topicref>
    <topicref format="html" href="https://books.google.co.uk/books/about/
Docs_Like_Code.html)" keys="docs-like-code" scope="external" >
        <topicmeta>
            <navtitle>Docs like code</navtitle>
        </topicmeta>
    </topicref>
    <topicref format="html" href="https://www.dita-ot.org/dev/topics/markdown-dita-
syntax-reference.html" keys="MDITA-syntax-ref" scope="external">
        <topicmeta>
            <navtitle>MDITA syntax reference</navtitle>
        </topicmeta>
    </topicref>
</map>
```

A few things of note:

- The *format* attribute uses *html* as a value because the reference is to a web page.
- Unlike internal links, you have to explicitly define the link text for external links via the <topicmeta> and <navtitle> tags. If you omit the <topicmeta> and <navtitle> tags, the link text is replaced by the URL.
- The <topicmeta> tag is a container for metadata (in this case title of the web page that appears in the link).
- The <navtitle> tag contains the link text that the reader sees when the document is built.

To refer to the last item in the map, use the following:

```
For more information see [MDITA-syntax-ref].
```

When built, the key value is replaced by link entitled "MDITA syntax reference".

Although you can use the standard markdown linking format to place external links in the body of a topic (and initially that might seem easier), the use of external links maps makes updating external URLs much easier.

An external links map is added as a submap to the root map using a <mapref> tag with the *processing-role* set to *resource-only* (see Organizing topics with maps on page 10 for an example).

Using MDITA keys for text variables

MDITA keys also allow you to create text variables. This is functionality that is simply not possible with common-or-garden Markdown.

Keys in DITA provide an means of indirect referencing. Keys are defined in a ditamap. They can be added to a root map or - if there are many - contained in a map of their own which is referenced in the root map.

The recommended format for creating a key definition in an XDITA map is:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map id="Keydefs">
    <title>Key Definitions</title>

    <keydef keys="company_name">
      <topicmeta>
        <keywords>
            <keyword>Grunt Industries</keyword>
       </keywords>
      </topicmeta>
    </keydef>
    <keydef keys="product_name">
      <topicmeta>
        <keywords>
            <keyword>Grunt Master 6000</keyword>
       </keywords>
      </topicmeta>
    </keydef>

</map>
```

As you can see, this ditamap does not reference any topics. It is used solely to define keys, and added to the root map using a <mapref> tag with the *processing-role* set to *resource-only* (see Organizing topics with maps on page 10 for an example). Here are the tags that are used in the above example in more detail:

- **keydef** - You use this tag to define the key itself by setting a value in the *keys* attribute.
- **topicmeta** - This tag is a container for metadata (in this case the variable string that is used by the key). The <topicmeta> tag can container a number of tags but for our purposes generally, it only contains those you see in the above example.
- **keywords** - The <keywords> tag holds the keyword that it is the value of the key.
- **keyword** - The <keyword> tag contains the text string itself.

When you are defining keys always use the format of tags shown above.

Within an MDITA topic the key is referenced by putting it within square brackets, for example:

```
The [product_name] represents a leap forward in home exercise technology by
 [company_name]!
```

Once built, *[product_name]* will be replaced by "Grunt Master 6000", and *[company_name]* by "Grunt Industries".

Referencing keys in HTML5 content

Because some content in MDITA files, such as notes and context references (for more information on context references see Using content references with MDITA on page 15), is in HTML and not Markdown. The usual methods for referencing keys do not work.

So to add a text variable in HTML text in an MDITA, use ** in stead of *[company_name]*. For example:

```
<p id="disclaimer"><note><span data-keyref="company_name"></span> accepts no liability
 for any mental illness, blindness, infertility, or death caused by even casual use of
 the <span data-keyref="product_name"/></span>.</note></p>
```

The next example includes a key reference for an external link defined in the external links dita map:

```
<p id="install-step1">First, download the appropriate installer for your operating
 system from the <span data-keyref="company_name"></span> website <span data-
keyref="grunt-downloads"></span> page. </p>
```

The example below is for an internal link, referencing a key from the root map:

```
<p><note type="tip">For more information on using filters in MDITA documents, see <span
 data-keyref="MDITA-filters"></span>.</note></p>
```

Using content references with MDITA

For content that appears in several locations in document, DITA writers use the content referencing (*conref*) mechanism. Again, this is functionality that heretofore was simply not possible with Markdown on its own. Some static site generators such as Jekyll permitted the use of *includes* but each include had to be a separate HTML file. *Conrefs* allow to put all referenced content in one place.

Best practice for using conrefs involves placing the content to be reused in a separate "warehouse" file which does not itself appear in the final output.

A MDITA Markdown topic can be used for a warehouse file but the content to be referenced must be in HDITA tags. The Markdown warehouse topic must include a YAML header with an idea (as in the example below.

A sample MDITA conref warehouse topic:

```
---
id: conref-content
---

# Conref content

<p id="install-step1">First, download the appropriate installer for your operating
 system from the Grunt Master website [Downloads](https://gruntmaster.com/downloads)
 page.</p>

<p id="disclaimer">Grunt Master Industries accepts no liability for any injury,
 insanity, infertility, or death caused by even casual use of the Grunt Master 6000.</
p>

...
```

To reference the shared content within an MDITA markdown file, an HDITA tag must be used along with the *data-conref* attribute. The value for the *data-conref* attribute uses the following format <TOPIC FILE NAME>#<TOPIC ID>/<TAG ID>. For example:

```
## Installing the Grunt Master 6000 companion app

Follow the instructions below to install the Grunt Master 6000 companion app:

1. <p data-conref="conref.md#conref-content/install-step1"></p>
<--! Reviewers go to conref.md and look at the paragraph with the install-step1 ID if
 you want to review this step. -->
1. Double-click on the installer, and follower the installation wizard instructions.
1. When installation is complete, restart your device.
<p data-conref="conref.md#conref-content/disclaimer"></p>
```

Once built the HDITA tag will be replaced by the referenced text.

As in the above example, it's a good idea to add a comment that points reviewers towards the conref source in case they might want to review that content too.

Keys in conref text

Because conref content is HTML not Markdown you cannot use references to any keys you have defined in the ditamap. For example is you used *[company_name]* in a the conref snippet, it would be rendered literally as "*[company_name]*" and not "Grunt Industries". Therefore, as

general rule, you should try to avoid where possible using content that you have also made into text variables in conref content.

Using filters with MDITA

MDITA filters are used to apply build conditions. In other words, to exclude certain text elements that are not relevant, for example, to certain readers, or for certain software versions, or releases. By applying different filters to an MDITA document you can produce different versions of the document to suit different purposes.

To incorporate text that can be filtered out at build time, MDITA topics must include HDITA tags that use the *props* attribute. Note in the example below that the entire unordered list must be in HTML markup not simply the list items that are to be filtered.

```
## Supported Operating Systems

The following operating systems are supported by the [product_name] companion app:

<ul>
  <li props="mac">Mac</li>
  <li>Windows</li>
  <li props="linux">Linux</li>
  <li>BSD</li>
</ul>
```

You use a special *ditaval* filter file is used to filter content. To filter out the first list item, the ditaval file used by the build instruction is:

```
<?xml version="1.0" encoding="UTF-8"?>
<val>
    <prop action="exclude" att="props" val="mac"/>
</val>
```

The *props* attribute can also added at the map level to filter out whole files from a build:

```
<topicref href="somefile.md" format="markdown" props="mac"/>
```

CAUTION: If the *props* attribute is added to a <topicref> element that has child topics, the children will also be filtered out.

Getting out of MDITA

I firmly believe that all source format content is always in a state of transition. Whatever format your source files are in, you need be sure you can get them into a different source format, should that be required.

MDITA may be the new flavour of the month, but like all tech writing technologies it'll be old hat some day. Something newer and shinier may come along that answers your needs even better. Or simply, events (such as a corporate takeover) may require you to migrate your MDITA content to another format to fit in with your new parent company's documentation system.

So how do you get out of MDITA?

1. The first step would be to transform it in a way that merges in all your referenced and filtered content to an intermediate format like HTML or a standard Markdown. Converting to Markdown, owing to its simplicity, gives you lots of options for conversion.
2. Once you have the content in Markdown, you have several options:

1. Use a tool like Oxygen XML Editor's Batch Convertor tool to switch from Markdown to DITA/Docbook/XHTML. The Batch Convertor tool in Oxygen is probably the cleanest way to get Markdown into DITA proper and provides excellent results in my experience.
2. Use Pandoc the universal converter which convert Markdown to a range of formats (including other docs-as-code formats like asciidoc).
3. If you want to convert to DITA and don't use Oxygen XML Editor, you can use the DITA-OT's *Normalized DITA* transform. This works pretty well but requires a bit more clean-up than the Batch Convertor tool. The converted files still have a *.md* extension which needs to be changed to *.dita*, for example, and the transform adds some unnecessary attributes and other stuff which you need to remove.

Simplifying DITA

At the time of writing (Spring 2023), there is no news on when exactly the new Lightweight DITA 1.0 (LwDITA) standard will be released. From what I've seen it will be a great step forward in simplifying DITA and easing the fairly steep learning curve encountered by many writers when taking on DITA for the first time.

However, in the meantime, what do we do?

Even though there is growing editor support for the new LwDITA DTDs and functionality, it's not quite there yet.

And even when it comes online, LwDITA might be too lightweight. I have worked with document sets that contained many complex tables and which were published to PDF. LWDITA only supports simple (as opposed to CALS) tables and does not support bookmaps at all. In a case like this, LwDITA would not be powerful enough.

I think it is possible to simplify standard DITA 1.3 use for ordinary - especially new - writers, without having to write whole new DTDs.

Here's how:

- **Stop worrying about Information Typing**

  I have never worked anywhere where the information type made any difference in how the content was managed. Forget about whether something is a task, a reference, or a concept topic. Just use the *topic* type for all topics. This is essentially the approach in LwDITA.

  This might have some DITA purists clutching their pearls over the very idea of using an ordered list for procedures instead of steps but, in the end, you can't tell in the output, and you'll save someone from learning all those task elements.
- **Stick to the `<map>` ditamap type wherever possible**

  Again, this is the approach taken by the team creating LwDITA. Admittedly, jettisoning bookmaps may not be feasible if you are required to publish PDF outputs.
- **Radically reduce available elements you use**

  Full-fat DITA 1.3 has something like 193 elements in its base implementation alone. The vast majority of which most writers don't use. I suggest auditing your content and looking at the type of elements and attributes you use and make a point of restricting access for writers to all the rest.
- **Simplify the Oxygen XML Author UI**

  Oxygen XML Editor/Author is my favorite software application. It is unbelievably powerful, very well-designed, and no other editor comes even close when working with DITA. And it can be a bit daunting for first-time users. Luckily, its UI is very extensible, and there are many things you can do to hide a lot of the more complex functionality that most writers, especially relatively inexperience ones, simply don't use on a regular basis.

  Here are somethings you can do to make the Oxygen Editor more user friendly:

  - Edit the `cc_config.xml` file so that only the elements (and attributes) you actually use are available in the **Elements** and **Attributes** windows. Editing this file also hides the unwanted elements and attributes in the right-click content completion menus.

    You can also use the `cc_config.xml` file to enforce element patterns (for example always having a `<p>` element inside a `<li>` element).
  - Create a shareable custom frameworks for topics and maps that novice OxygenXML users can install allow them work with the reduced list of elements and attributes.

- Define a simple editor layout with only the windows that are necessary, and share this with new team members. This is a big help for new users.

Creating a `cc_config.xml` file

The Oxygen `cc_config.xml` file allows you define which DITA elements and attributes are displayed in the Elements and Attributes windows in the Oxygen UI. By configuring element proposals in the `cc_config.xml` file, you can control which elements and attributes your users can add to topics and maps.

> **Note:** You are not writing a new DTD here, so a writer could technically manually type in any legal DITA element or attribute and the topic would still validate.

To create a new `cc_config.xml` file, you could copy and edit the default file that lives in the `[Oxygen install folder]/frameworks/dita/resources` folder. Alternatively you can create one from scratch using a conveniently-provided template that you can open from **File**>**New**>**Framework templates**>**Oxygen extensions**>**Content Completion Configuration**.

The template contains some sample code (which you can adopt or delete). The Oxygen online help provides fairly comprehensive information on how work with the `cc_config.xml` file.

For more information, see OxygenXML Help: Customize Content Completion.

A sample cc_config.xml file

I have created a sample `cc_config.xml` file based largely on the Lightweight DITA 1.0 XDITA proposal for a reduced list of DITA elements. Of course, your `cc_config.xml` file could contain any elements and attributes you deem necessary.

In addition to restricting the available elements and attributes for users, this `cc_config.xml` file also defines child elements or mandatory attributes and attribute values to be included (or omitted) for some elements. To give one example, if you add a `<section>` element, Oxygen will also add a `<title>` and `<p>` child elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Allows contributing to the values presented on content completion for element and
 attribute values.
    You can append to the values obtained from the schema or replace them all together.
    These values can be given as literal values or they can be obtained by executing an
 XSLT script.

    IMPORTANT: This file must be saved as cc_config.xml in a folder that is present in
 the Classpath
    of the Document Type (or framework).
-->
<?xml-model href="http://www.oxygenxml.com/ns/ccfilter/config/ccConfigSchemaFilter.sch"
 type="application/xml" schematypens="http://purl.oclc.org/dsdl/schematron"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.oxygenxml.com/ns/ccfilter/config http://
www.oxygenxml.com/ns/ccfilter/config/ccConfigSchemaFilter.xsd"
    xmlns="http://www.oxygenxml.com/ns/ccfilter/config">

    <elementProposals
        possibleElements="

        topic title shortdesc prolog body section p
        ph ul ol li dl
        dlentry dt dd pre
        simpletable sthead strow stentry
        fig image object desc alt note tt xref

        map topicref topicmeta navtitle data mapref keydef keywords keyword
        "
```

```
        possibleAttributes="
        DITAArchVersion
        domains class outputclass
        id href conref keyref keys
        processing-role scope format type
        props others dir xml:lang translate"/>

    <elementProposals path="ol/li" insertElements="p"/>
    <elementProposals path="ul/li" insertElements="p"/>
    <elementProposals path="image" rejectAttributes="alt"/>
    <elementProposals path="image" insertElements="alt"/>
    <elementProposals path="fig" insertElements="title image"/>
    <elementProposals path="section" insertElements="title p"/>
    <elementProposals path="table" insertElements="title"/>
    <elementProposals path="topicref" rejectAttributes="navtitle"/>

    <elementProposals path="pre">
 <insertAttribute name="translate" value="no"/>
</elementProposals>

<elementProposals path="tt">
 <insertAttribute name="translate" value="no"/>
</elementProposals>

    <match attributeName="translate">
        <items action="addIfEmpty">
            <item value="yes"/>
            <item value="no"/>
        </items>
    </match>

    <match attributeName="dir">
        <items action="addIfEmpty">
            <item value="ltr" annotation="right-to-left text"/>
            <item value="rtl" annotation="left-to-right text"/>
            <item value="lro" annotation="left-to-right override"/>
            <item value="rto" annotation="right-to-left override"/>
        </items>
    </match>

    <!-- Contributes values for xml:lang attribute -->
    <match attributeName="lang">
        <items action="addIfEmpty">
            <item value="de" annotation="German"/>
            <item value="de-DE" annotation="German"/>
            <item value="en" annotation="English"/>
            <item value="en-US" annotation="English (US)"/>
            <item value="es" annotation="Spanish"/>
            <item value="fi" annotation="Finnish"/>
            <item value="fr" annotation="French"/>
            <item value="fr-FR" annotation="French"/>
            <item value="he" annotation="Hebrew"/>
            <item value="it" annotation="Italian"/>
            <item value="it-IT" annotation="Italian"/>
            <item value="ja" annotation="Japanese"/>
            <item value="ja-JP" annotation="Japanese"/>
            <item value="nl" annotation="Dutch"/>
            <item value="ro" annotation="Romanian"/>
            <item value="ru" annotation="Russian"/>
            <item value="sl" annotation="Slovenian"/>
            <item value="sv" annotation="Swedish"/>
            <item value="zh" annotation="Chinese"/>
            <item value="zh-CN" annotation="Chinese (simplified)"/>
        </items>
    </match>

    <match elementName="note" attributeName="type" editable="onlyAllowedItems">
        <items action="replace">
            <item value="note"/>
            <item value="caution"/>
            <item value="tip"/>
            <item value="important"/>
            <item value="remember"/>
            <item value="restriction"/>
            <item value="other"/>
        </items>
    </match>

    <match attributeName="format">
        <items action="addIfEmpty">
            <item value="dita"/>
            <item value="ditamap"/>
            <item value="html"/>
            <item value="pdf"/>
```

```
            <item value="markdown"/>
            <item value="mdita"/>
            <item value="png"/>
            <item value="svg"/>
            <item value="gif"/>
            <item value="jpg"/>
            <item value="zip"/>
        </items>
    </match>

    <match elementName="object" attributeName="type">
        <items action="addIfEmpty">
            <item value="video/mp4"/>
            <item value="video/ogg"/>
            <item value="audio/mpeg"/>
            <item value="audio/ogg"/>
            <item value="audio/wav"/>
        </items>
    </match>

    <match elementName="pre" attributeName="outputclass">
        <items action="addIfEmpty">
            <item value="language-xml"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for XML."/>
            <item value="language-java"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for Java."/>
            <item value="language-css"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for CSS."/>
            <item value="language-javascript"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for JavaScript."/>
            <item value="language-json"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for JSON."/>
            <item value="language-sql"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for SQL."/>
            <item value="language-c"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for C."/>
            <item value="language-cpp"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for C++."/>
            <item value="language-csharp"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for C#."/>
            <item value="language-ini"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for ini scripts."/>
            <item value="language-python"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for Python."/>
            <item value="language-ruby"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for Ruby."/>
            <item value="language-perl"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for Perl."/>
            <item value="language-bourne"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for Bourne Shell."/>
            <item value="language-php"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for PHP."/>
            <item value="language-lua"
                annotation="Provide content highlight in the DITA-OT XHTML and PDF
outputs for Lua."
            />
        </items>
    </match>

</config>
```

Setting up a custom framework

Custom frameworks are a way to customize the Oxygen XML editor to work more easily with different XML vocabularies. We are not creating a new vocabulary here but rather restricting existing ones. We will need to create to custom frameworks: one for DITA topics, and one for DITA maps. These custom frameworks can then be shared with colleagues and imported into their Oxygen instances.

The first step in defining the new frameworks is to set up the location from which they will be shared. I created a GitHub repository call `tools` and added the following directory structure:

```
custom_frameworks
            - simple_dita
                - resources
                - templates
            - simple_ditamap
                - resources
                - templates
```

- **custom_frameworks** - this folder is the root folder for all custom frameworks.
- **simple_dita** - is the container for the `simple_ditamap.framework` file.
- **simple_ditamap** - is the container for the `simple_ditamap.framework` file.
- **/resources** - holds the `cc_config.xml` file. There are many other files that could live here such as CSS or Schematron files if required.
- **/templates** - contains the topic or map templates.

When complete, colleagues can clone this repository, and use some Oxygen preferences settings to access its files.

Template and resource files

Now that you have the folder structure in place, it's time to add the template and resource files:

1. First move a copy of the `cc_config.xml` file you created in to the resources folder in within both the `simple_dita` and `simple_ditamap` folders.

   > **Note:** In the example I cited in Creating a `cc_config.xml` file on page 20, both topic and map elements and attributes were configured. You could also create separate `cc_config.xml` files defining only the relevant elements and attributes for each framework.

2. Next, copy the `Topic.dita` and `Topic.properties` files from `frameworks/dita/templates/topic` folder within your Oxygen install folder to the `simple_dita/templates` folder.
3. Finally, copy the `Map.ditamap` and `Map.properties` files from `frameworks/dita/templates/map` folder within your Oxygen install folder to the `simple_dita/templates` folder.

Configuring the custom framework location

Having set up a location for the custom frameworks, the next step is to tell Oxygen where to find it:

1. In the Oxygen UI, go to **Options**>**Preferences**>**Document Type Association**>**Locations**, and click **Add**.

2. In the **Choose frameworks directory** dialog that appears, navigate to the `custom_ frameworks` folder, and click **OK**.

   The path to your `custom_ frameworks` folder appears in the **Additional frameworks directories** list.
3. Click **OK** to save.

> **Note:** If you plan to continue on to the next part, you can skip the final step here and continue working the **Preferences** dialog.

Creating the `simple_dita` framework

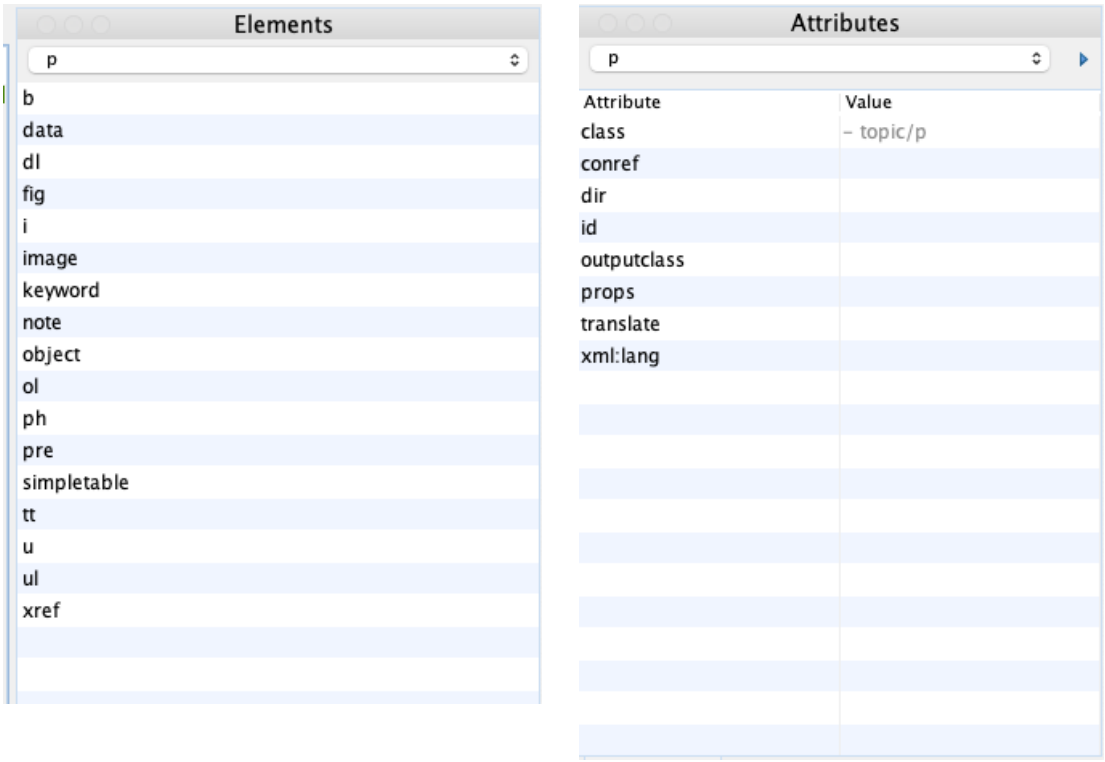
Having told Oxygen where to find the frameworks and their associated files, you can proceed to create the first framework. Because this is a simplified version of DITA 1.3, I've called my first framework `simple_dita`.

For more information on creating custom frameworks in Oxygen, see OxygenXML Help: Create Custom Frameworks.

1. If the **Preferences** dialog is not already open, go to **Options**>**Preferences**>**Document Type Association**.
2. Click **DITA** and then **Extend**.
3. In the **Document type** dialog that opens:
   1. Give the new framework a **Name**. I used *SIMPLE DITA*.
   2. Set **Priority** to **Highest**.
   3. Set **Storage** to **External**, and click the folder icon to navigate to the `simple_dita` folder within your `customer_frameworks` folder.
   4. On the **Classpath** tab, select the `${baseFramework}/resources` row and click the wrench icon.

      Change the content in the **Directory** field to `${frameworkDir}/resources/` and click **OK**. In this case, the `${frameworkDir}` variable contains the path to the `custom_frameworks/simple_dita` folder.
   5. On the **Templates** tab, select all the rows and click the X icon to delete them all. Click the + icon, and enter `${frameworkDir}/templates/` in the **Directory** field and click **OK**.
   6. On the Author tab, edit the **Menu**, **Contextual Menu**, **Toolbar**, and **Content Completion** sub-tabs, to add or remove any icons not relevant to your framework.
   7. Click **OK**, and then **OK** again close the **Preferences** dialog.

Once completed, your Elements and Attributes windows in Oxygen look like the examples below:



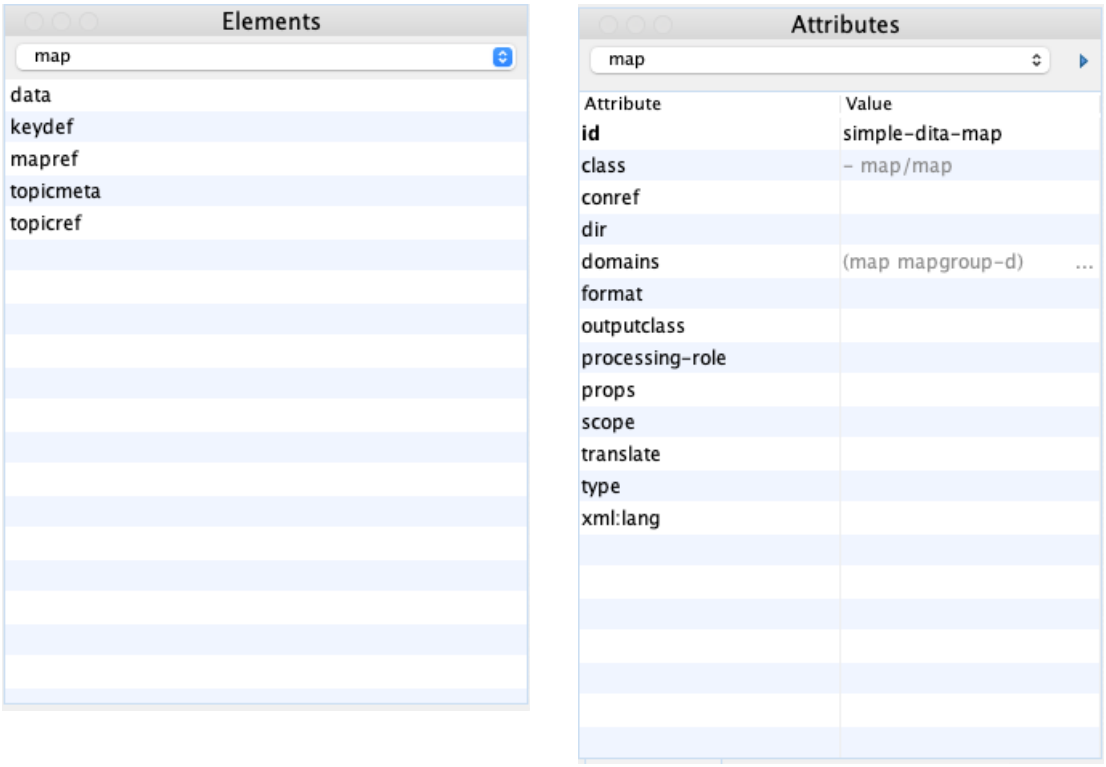
Creating the `simple_ditamap` framework

Having created the topic framework `simple_dita`, the next task is to create a map framework. For simplicity's sake, let's call it `simple_ditamap`.

1. If the **Preferences** dialog is not already open, go to **Options**>**Preferences**>**Document Type Association**.
2. Click **DITA Map** and then **Extend**.
3. In the **Document type** dialog that opens:
    1. Give the new framework a **Name**. I used *SIMPLE DITAMAP*.
    2. Set **Priority** to **Highest**.
    3. Set **Storage** to **External**, and click the folder icon to navigate to the `simple_ditamap` folder within your `customer_frameworks` folder.
    4. On the **Classpath** tab, select the `${baseFramework}/resources` row and click the wrench icon.

        Change the content in the **Directory** field to `${frameworkDir}/resources/` and click **OK**. In this case, the `${frameworkDir}` variable contains the path to the `custom_frameworks/simple_ditamap` folder.
    5. On the **Templates** tab, select all the rows and click the X icon to delete them all. Click the + icon, and enter `${frameworkDir}/templates/` in the **Directory** field and click **OK**.
    6. On the Author tab, edit the **Menu**, **Contextual Menu**, **Toolbar**, and **Content Completion** sub-tabs, to add or remove any icons not relevant to your framework.
    7. Click **OK**, and then **OK** again close the **Preferences** dialog.

Once completed, your Elements and Attributes windows in Oxygen look like the examples below:



A simple editor layout

The final thing that I think can contribute to making the life of DITA newbie easier is a custom editor layout that only shows the Oxygen windows that a new user is most likely to use. In my experience, in addition to the main editor pane of course, these are:

- DITA Maps manager
- Elements
- Attributes
- DITA Reusable Components

You may have your own preferences, but sharing your favorite editor layout could not be easier. Click **Window**>**Export Layout**, give the layout a name and save it to a location such as GitHub repository from where it can be shared as a `.layout` file.

Sharing the frameworks and layout

If you have saved the layout and custom framework files to GitHub repository, or shared location on your network, installing them is very straightforward:

1. First, clone the repository (or open the network location), where the files have been saved.
2. To install custom editor layout in Oxygen, click **Window**>**Load Layout**>**Custom**, locate and select the `.layout` file. Then, click **Open**.

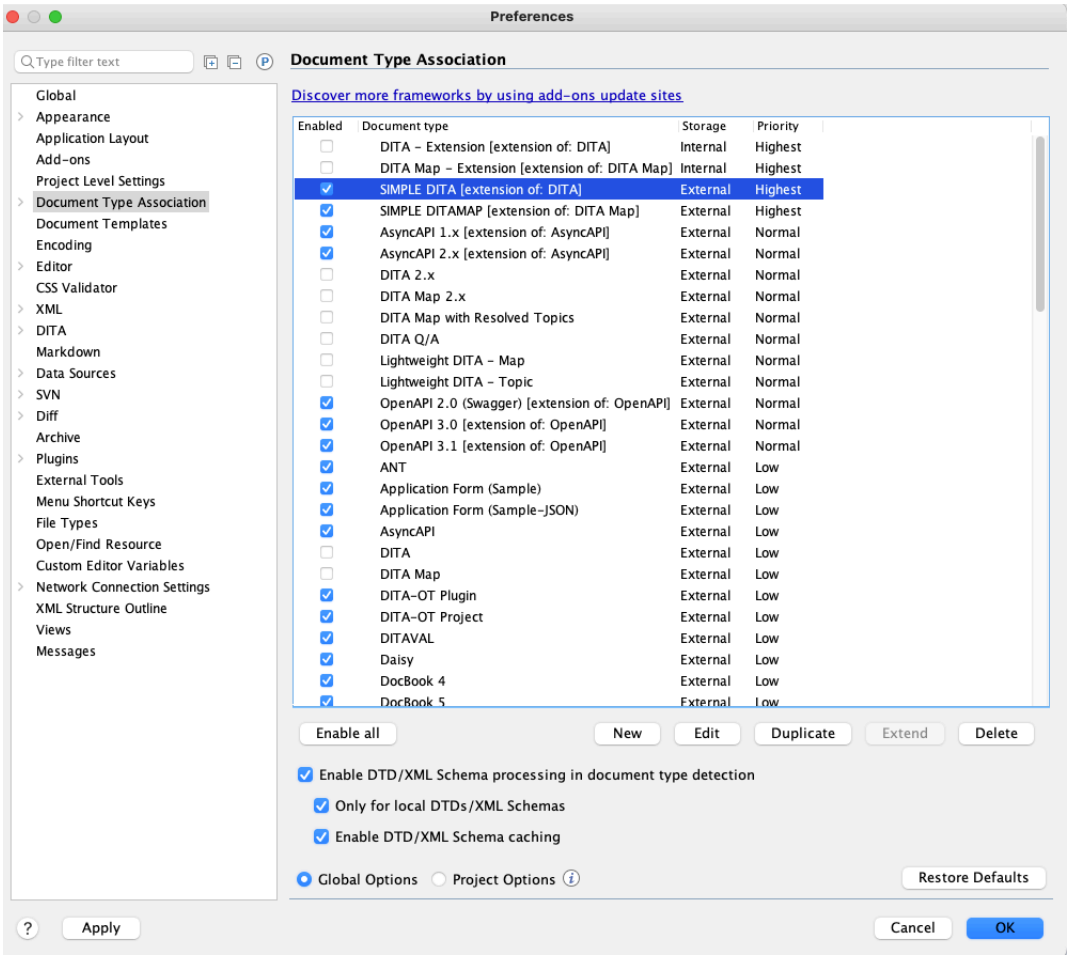
   Your editor layout updates automatically.
3. To install the frameworks, first set the location of the new framework files:

   1. In the Oxygen UI, go to **Options**>**Preferences**>**Document Type Association**>**Locations**, and click **Add**.

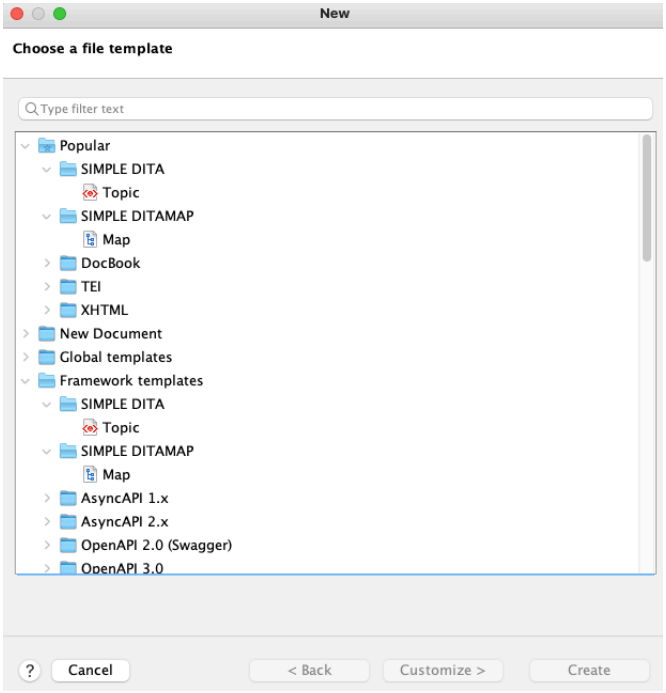
2. In the **Choose frameworks directory** dialog that appears, navigate to the `custom_ frameworks` folder, and click **OK**.

   The path to your `custom_ frameworks` folder appears in the **Additional frameworks directories** list.
3. Click **OK** to save.

4. Next got to **Options**>**Preferences**>**Document Type Association** where you should the new frameworks installed. Deselect any other frameworks that reference DITA and click **OK**. The example below is from my machine:



   The Elements and Attributes windows should reflect the restricted choices created in the `cc_config.xml` file. If they don't, close and reopen Oxygen.
5. To check for the new templates, click **File**>**New**. The **Choose a file template** dialog shows the new Topic and Map templates.



How do I undo all this?

Undoing the changes imported above is very simple.

1. To return to the default Oxygen layout, click **Window**>**Load Layout**>**Default**.
2. To return to the default DITA elements, attributes and templates, go to **Options**>**Preferences**>**Document Type Association** and deselect the new templates your you imported. Select all the other frameworks that mention DITA and click **OK**. You may need to restart Oxygen to changes. That's it!

PDF2 theme files

For many years, DITA Information Architects have done battle with the mighty DITA Open Toolkit (DITA-OT) PDF2 plugin. Armed with Leigh White's indispensable tome DITA for Print, they have struggled to create custom PDF plugins that produce good looking PDFs from DITA source files. And while for truly bespoke PDF output, customizing the PDF2 plugin is the only way, release 4.0 of the DITA-OT offers an somewhat easier (but by no means simple) way to work with the PDF2 plugin - *theme* files.

Prior to this, creating a custom PDF plugin involved creating a whole set of customization folders and override files (the system was based largely on the methods used in the Docbook Style Sheets). It involved hacking around in XSLT templates and attributes files. Generally this was done by consultants, usually when the team first converted to DITA. And generally the expertise was never passed on, or soon lost as team personnel changed. Something as simple as a logo change or an update to brand colors could prove an expensive nightmare for teams that did not have XSLT expertise on staff.

Theme files are YAML files whose key/value pairs represent CSS properties and values (or actually XSL-FO properties which are *nearly* always the same). Once configured, the theme file can be applied to the build instruction. For example:

```
dita -i some.ditamap -o out --format=pdf2 --theme=mytheme.yaml
```

The theme YAML file feeds the theme file parameters to the build-in PDF2 plugin at build time with no requirement to actually go in and mess with the XSLT attributes files within the PDF2 plugin itself. And although it may not solve all the problems of PDF creation from DITA, it goes a long way to help documentation teams work with PDF output in a manageable and maintainable way.

Working with theme files

The `libexec/docsrc/samples/themes` folder within the 4.0 (and later) release of the DITA-OT contains several sample theme files. The most useful of which is the dita-ot-docs-theme.yaml file. It's a long file which I won't reproduce here but click on the link to go to DITA-OT docs GitHub repository to see it in full.

The easiest way to work with theme files to copy and edit your own version of the `dita-ot-docs-theme.yaml`.

What can theme files do?

Even though theme files are not as powerful as custom plugins, there is still a wide range of things you can do with them:

- Define page, size, orientation and margins.
- Define headers and footers.
- Create a cover page and add an image and title to it.
- Style a range of block level and inline elements.
- Create style variables that you can use elsewhere in the theme file.
- Extend existing theme files to produce variants (e.g. versions of the document in A4 and US Letter formats).

The DITA-OT website contains a reasonably good quickstart guide which, coupled with the sample theme file, are enough to get you started. A knowledge of XSL-FO properties (or at least

CSS) would really help but a little trial and error, and practice styling some basic PDF output will prove fruitful.

Limitations

As theme files are a relatively recent innovation from the team behind the DITA-OT, there are limitations to what you can do with them. With new releases of the toolkit, there may well be further functionality added but theme files will likely not replace all the functionality you can achieve by building a custom PDF plugin. There are several reasons for this:

- Theme files only interact with the XSL attributes files in the PDF2 plugin. They do not change anything in the XSLT templates. This means advanced configuration of PDFs (for example adding bookmap variables to a cover page) is not possible.
- There are also not insignificant financial reasons for not developing theme files beyond a certain point. Many supporters of the DITA-OT are DITA consultants who make a living out of writing custom PDF and other DITA-OT plugins, and they would not be happy if there was tool that allowed customers to do it for themselves.

  Similarly, some of the DITA-OT's sponsors are software houses that sell tools for producing PDF output from DITA (such as Syncrosoft's Oxygen Styles Basket, or Antenna House's PDF5-ML plugin). Any open source tool that seriously rivalled their proprietary offerings could ultimately affect the DITA-OT project's funding. Of course that does not mean that an independent developer could not develop such an open source tool based on theme files.

So what can't you do with theme files?

Having created several PDF templates now using theme files, here are some limitations that I found:

- **Front covers** - Other than `<title>`, `<booktitle>` and `<booktitlealt>` (if you are using a bookmap) there are no other variables you put on the front page. I also was not able to find a way to style the `<booktitlealt>` to provide subtitle content - which means I had to omit it.
- **Back covers** - There is currently no way to add a back cover to your PDF document.
- **Footers** - Configuration of footer content is very limited. You can add page numbers and even some text, but all footer content must be styled the same way. So if your page number is right-aligned and bold, so will any text be that you put in there. Basically, footers in theme files are good for page numbers, and anything like copyright text would better be placed elsewhere in your document.
- **Tables** - I could find no way to style table cell borders or content.
- **Heading levels** - You can only style headings as far as H4. H5 and H6 headings are styled in the output but there is no way change this currently. Although you can style `<example>` and `<section>` titles.
- **Lists** - List item markers (numbers or bullets) cannot be styled.

  Embedded lists (lists within lists) are not styled differently from their parent. An embedded numbered list will be styled 1, 2, 3, etc. rather than a, b, c, or i, ii, iii for example. Embedded unordered lists use the same bullet style as their parent list.

Despite all of the above, the theme file feature is in active development, and the DITA-OT developer Jarno Elovirta has said he would continue to add new features in upcoming releases. Hopefully some of the issues outlined above will be addressed in release 4.1.

Are theme files worth using?

Despite their limitation, theme files are definitely worth using. Yes, you will still need some knowledge of XSL-FO properties but anyone with some knowledge of CSS would be able to pick that up pretty quickly. True, your docs will not look quite as pretty as those made with a custom PDF plugin, but you can still make pretty good looking and totally functional documents using theme files.

The big benefit here is ease of use and maintenance. Currently if a request to change some styling on our PDF output comes in while I'm away on holiday, it has to wait until I come back because no-one else in the team dares touch the custom plugins with any confidence. But a little training is all it takes for anyone on the team to take a stab at replacing a cover image or changing the corporate colors in a theme file.

And when it comes to developing PDFs from DITA content, using theme files comes a whole lot cheaper than paying a consultant to write a plugin.

I work for a company that has a lot of very long, complex documents that are outputted to PDF. I am actively looking at ways we can simplify the styling and structure of our documents so that they can be build using a theme file and the standard PDF2 plugin rather than the current custom PDF plugin I wrote. This will enable the team to continue to style and maintain the PDF output when I am not around.

Automating DITA builds

Back in the day I used to automate my DITA-OT builds by writing shell scripts that chained DITA build commands along with their various build parameters one after the other.

Then `.properties` files came along and it became a lot easier to do builds because you could put most of a document's build parameters in a single file and call it from the DITA build command, as in the example below:

```
dita -i my_map.ditamap -o out/pdfs --format=pdf2 --filter=my_filter.ditaval --
propertyfile=my_build.properties
```

This made for more concise shell scripts because the build parameters for most documents were the same and you could call the same `.properties` file for most of your build commands. However, if you required different parameters for a document, you needed a different `.properties` file. Each `.properties` file could only list the parameters for one output format, and you still needed to use a different DITA build command in your script for every document you wanted to build.

Project files

Finally with DITA-OT release 3.4 came project files. Project files allow you to put all the build parameters (including input files, output folders, output formats, and filters) all in the same file for multiple documents. You can even define multiple outputs with different parameters for the same document.

Your DITA build command now looks something like this:

```
dita --project=myproject.xml -v
```

> **Note:** The `-v` switch simply tells the DITA Open Toolkit to use verbose mode and run the log file in the command prompt window.

The above example uses a project file in XML format. You can also write them in YAML and JSON.

DITA-OT project file structure

A DITA project file is an XML file containing build parameters for one or more documents that you can pass to the `dita` build command.

The following is a sample project file in XML format:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://www.dita-ot.org/project">

  <deliverable name="Grunt Master 6000 Guide" id="ug_gruntmaster6000_html5">
    <context name="Grunt Master 6000 Guide" id="ug_gruntmaster6000">
        <input href="ug_gruntmaster6000.ditamap"/>
        <profile>
            <ditaval href="GM6000.ditaval"/>
        </profile>
    </context>
    <output href="out"/>
    <publication transtype="html5">
        <param name="args.css" href="../css/custom.css"/>
        <param name="args.copycss" value="yes"/>
    </publication>
  </deliverable>

</project>
```

Element	Description
project	The wrapper element for deliverables in the project file. It contains only one attribute `xmlns` which defines the XML namespace for the elements in the file.
deliverable	The wrapper element for all the deliverable parameters. Its attributes define the deliverable name and a unique ID for the deliverable.
context	This element wraps the elements that hold the input information (the source ditamap) and filters that are used by the build. **Note:** Context elements can be used independently of the deliverable elements and so can employ their own name and unique ID attributes.
input	This element contains the relative path to the input ditamap you want to use for the build within a `href` attribute.
profile	The <profile> element wraps one or more <ditamap> elements.
ditaval	This element contains the relative path to the any filter file you want to use for the build within a `href` attribute. There can multiple <ditaval> tags for each context.
output	This element contains the relative path to the output folder where you want the outputted content to be deposited within a `href` attribute.
publication	The <publication> element wraps parameter elements which content the build parameters to be fed to the DITA-OT. Importantly, it also must contain a `transtype` attribute that tells the DITA-OT which output format to transform the content into.
param	This element contains the actual build parameters that the DITA-OT will use when creating the output. Build generally differ according to the output you have defined in the <publication> element `transtype` attribute. For a full list of parameters by output format, see the DITA-OT website

Visit the DITA-OT website see examples of project files in JSON and YAML formats, and for other information on using project files.

Building multiple documents

The example shown in DITA-OT project file structure on page 33 was for one document deliverable in one output format. But with project files you can include many different deliverables and even several different deliverables for the same ditamap. See the following sample project file, `gm_deliverables.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://www.dita-ot.org/project">

<--! Grunt Master 6000 Guide deliverables -->

<deliverable name="Grunt Master 6000 Guide" id="ug_gruntmaster6000_html5">
    <context name="Grunt Master 6000 Guide" id="ug_gruntmaster6000">
        <input href="ug_gruntmaster6000.ditamap"/>
        <profile>
            <ditaval href="GM6000.ditaval"/>
        </profile>
    </context>
    <output href="./html5-output"/>
```

```
    <publication transtype="html5">
        <param name="args.css" href="../css/custom.css"/>
        <param name="args.copycss" value="yes"/>
    </publication>
  </deliverable>

<--! Grunt Master 3000 Guide deliverables -->

  <context name="Grunt Master 3000 Guide" id="ug_gruntmaster3000">
    <input href="ug_gruntmaster3000.ditamap"/>
    <profile>
      <ditaval href="GM3000.ditaval"/>
    </profile>
  </context>

  <deliverable name="Grunt Master 3000 Guide" id="ug_gruntmaster3000_html5">
    <context idref="ug_gruntmaster3000"/>
    <output href="./html5-output"/>
    <publication transtype="html5">
      <param name="args.css" href="../css/custom.css"/>
      <param name="args.copycss" value="yes"/>
    </publication>
  </deliverable>

<deliverable name="Grunt Master 3000 Guide" id="ug_gruntmaster3000_pdf">
    <context idref="ug_gruntmaster3000"/>
    <output href="./pdf-output"/>
    <publication transtype="pdf2">
      <param name="args.chapter.layout" href="BASIC"/>
      <param name="outputFile.base" value="Grunt Master 3000 Guide"/>
    </publication>
  </deliverable>

</project>
```

The above example builds 3 deliverables from 2 source ditamaps. All 3 deliverables can be built with just one command:

```
dita --project=gm_deliverables.xml
```

The first part builds HTML5 output for the *Grunt Master 6000 Guide*.

The second part builds HTML5 and PDF outputs for the *Grunt Master 3000 Guide* using two different filter files.

Notice in this section of the project file that the `context` is independent of the `deliverable` sections, and that there are two `deliverable` sections that both have their own <context> elements which reference the same `context` ID via the `idref` attribute.

Other things you can do with project files

There are a couple of other things you can do with project files. You can publish deliverables from several different project files at once, or just one deliverable from a project file containing several. You can also use the DITA `deliverables` command as a quick and easy way to see what deliverables a project file contains.

Using multiple project files together

The project file syntax also incorporates an <include> element which you can use to chain several project files together and produce all their deliverables in one go. See the following example, `all_deliverables.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="https://www.dita-ot.org/rng/project.rnc" type="application/relax-ng-
compact-syntax"?>

    <project xmlns="https://www.dita-ot.org/project">

        <include href="product1_deilverables.xml"/>
        <include href="product2_deilverables.xml"/>
        <include href="product3_deilverables.xml"/>
```

```
    </project>
```

The command below therefore would build all the deliverables contained in the `product1_deilverables.xml`, `product2_deilverables.xml`, and `product3_deilverables.xml` project files:

```
dita --project=all_deliverables.xml
```

Listing all the deliverables in a project file

If a project file contains many deliverables, and especially if it is written in XML, it can be no easy task to find exactly which deliverables it builds. Luckily, there is a command for just for that. For example:

```
dita deliverables gm_deliverables.xml
```

The DITA `deliverables` command if used on the `gm_deliverables.xml` described in Building multiple documents on page 34 returns each deliverable's ID and name:

```
ug_gruntmaster6000_html5 Grunt Master 6000 Guide
ug_gruntmaster3000_html5 Grunt Master 3000 Guide
ug_gruntmaster3000_pdf   Grunt Master 3000 Guide
```

Building a single deliverable from a project file

If you only want to build a single deliverable and you know that its build parameters are contained in a certain project file you can pass the deliverable's unique ID to the `--deliverable` switch with the DITA build command. The following example builds the PDF version of the *Grunt Master 3000 Guide* listed in the `gm_deliverables.xml` project file:

```
dita --project=gm_deliverables.xml --deliverable=ug_gruntmaster3000_pdf
```

Note: You can also build all (or just one) of the deliverables in a project file you have open in Oxygen by going to **Document**>**Transformation**>**Configure Transformation Scenario(s)** and selecting either **Publish DITA-OT Project (all deliverables)** or **Publish DITA-OT Project (select deliverable).**

DITA CICD with GitHub Actions

GitHub Actions are a quick and easy way to automate build outputs form DITA content stored in a GitHub repository.

The following code is a simple GitHub Actions workflow builds HTML5 files from MDITA content and deposits them in a branch called *published*. This build is kicked off manually from GitHub but, as you can see in the code snippet below, you can also set it start builds on push or on pull_requests to a certain branch. It is also possible to configure a cron job to run the build at a specific time.

```
name: site-build

# Controls when the action will run.
on:

  # Triggers the workflow on push events but only for the main branch
  push:
    branches: [ main ]

jobs:
  build-dita:
    name: Build DITA
    runs-on: ubuntu-latest
    steps:
      - name: Git checkout
        uses: actions/checkout@v2
      - name: Build HTML5
        id: DITA-build
        uses: dita-ot/dita-build-action@master
        with:
          install: |
            dita install fox.jason.extend.css
            dita install net.infotexture.dita-bootstrap
            dita install fox.jason.prismjs
          build:  |
            dita --project=myproject.xml
      - name: Upload DITA
        id: upload
        uses: actions/upload-artifact@v2
        with:
          name: dita-artifact
          path: 'out'
      - name: Deploy
        id: deploy
        uses: JamesIves/github-pages-deploy-action@3.7.1
        with:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          BRANCH: published # The branch the action should deploy to.
          FOLDER: out # The folder the action should deploy.
          TARGET-FOLDER: docs
```

The `jobs` section of the workflow file spins up an Ubuntu Linux instance and then runs a series of steps:

1. The first step, *Git checkout*, is GitHub-provided action that checks out the current branch.
2. The real work is done by the next step, *Build HTML5*. It uses the *dita-build-action* GitHub Action created by Jason Fox which downloads the latest version of the DITA Open Toolkit, installs 3 required plugins from the DITA-OT plugin registry, and runs the build instruction using the *myproject.xml* project file stored in the branch. The *myproject.xml* project file contains all the information on the source ditamap, output transtype, output folder, filter files, and any build parameters. For more information on configuring project files, see .
3. The third step, *Upload DITA*, uploads the files outputted by the build to a folder called *out*.
4. The last step, *Deploy* uses a GitHub action written by James Ives originally meant to deploy content to a GitHub Pages branch but used here to move to the *docs* folder in the *published* branch. From there web hooks could be employed to the content to a service like Netlify.

Manual builds

The example above runs when a content is pushed to the `main` branch. When you are testing builds, it's easier to run builds manually. Adding the `workflow_dispatch` event trigger to your workflow file allows you to build manually by clicking **Run workflow** on the **Actions** tab in the GitHub UI.

```
name: site-build

on:
  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:

...
```

Timed builds with cron

In a mature CI/CD docs environment, documentation can be built and deployed on a regular, perhaps daily, basis. Workflows can integrate cron jobs that kick off the builds on specified days and times. The example below runs the build every weekday night at 2am:

```
name: site-build

on:
  schedule:
    # Runs "at 02:00 every Mon-Fri"
    - cron: '* 2 * * 1-5'

...
```

Note: For help with cron notation, https://crontab.guru/ is useful resource.

Configuring the dita-ot-action

There are several ways to configure the *dita-ot-action* GitHub Action. The DITA-OT website contains detailed instructions and several sample workflow files are provided in the `docsrc/samples/github-actions` folder within the DITA-OT install folder, and on the DITA-OT docs GitHub repository.

The use of Project files on page 33 is the simplest course of action when building using the *dita-ot-action* because a lot of the configuration is in the project file itself and does not have to be added to the workflow YAML file.

Installing community plugins

There are a limited number of very basic plugins included with the DITA-OT. However, other community developed plugins are available via the DITA-OT plugin registry. These can be installed most easily using the `dita install` command as in the following excerpt:

```
...

jobs:
  build-dita:
    name: Build DITA
    runs-on: ubuntu-latest
    steps:
      - name: Git checkout
        uses: actions/checkout@v2
      - name: Build HTML5
        id: DITA-build
        uses: dita-ot/dita-build-action@master
        with:
          install: |
            dita install fox.jason.extend.css
            dita install net.infotexture.dita-bootstrap
```

```
            dita install fox.jason.prismjs
          build:  |
            dita --project=myproject.xml
...
```

The above example also happens to install `nodejs` and `apt-utils` which are dependencies of the `net.infotexture.dita-bootstrap` plugin.

Installing custom plugins

Most companies that use DITA will have developed custom DITA-OT plugins at some point. There are two ways to add a custom plugin: via a custom registry or, the old-fashioned way, via a plugin ZIP archive. The former method involves forking the DITA-OT plugin registry, uploading your plugins to it, and creating a JSON registry file for each plugin. This is only worthwhile in my opinion if you have a lot of custom plugins, and a lot of folks installing and uninstalling them on a regular basis. As far as GitHub actions are concerned, simply installing a plugin as a ZIP archive from another repository is a lot easier.

To do this, upload you custom plugin files unzipped to a dedicated branch in a GitHub repository. GitHub zips the content in the background and you can install the ZIP archive by passing it as a URL to the `dita install` command.

The following example installs a plugin from a user (myCompany) repository (DITA-plugins) custom_pdf branch.

```
...

jobs:
  build-dita:
    name: Build DITA
    runs-on: ubuntu-latest
    steps:
      - name: Git checkout
        uses: actions/checkout@v2
      - name: Build Custom PDF
        id: DITA-build
        uses: dita-ot/dita-build-action@master
        with:
          install: |
            dita install https://github.com/myCompany/DITA-plugins/archive/custom_pdf.zip
          build:  |
            dita --project=myproject.xml
...
```

Building multiple deliverables in GitHub

The beauty of using Project files on page 33 in your GitHub builds is that a lot of the build information is already contained in the project file itself. You can use one `dita` command to build not just multiple documents from different ditamaps but also different output types for the same document, provided that:

1. Your GitHub workflow file downloads all the required plugins for the output types you want to build.
2. Your project file includes all the transtype information you need to define the output formats.

The following project file defines three deliverables for the *Grunt Master 3000 Guide*: a HTML5 website, a PDF version build with a custom PDF plugin, and another PDF version that uses the basic PDF2 plugin and which requires a theme file (for more information, see PDF2 theme files on page 29). Because of conflicting ditaval filter files, there are 2 different contexts for the *Grunt Master 3000 Guide* - one for each output format.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="https://www.dita-ot.org/project">

<--! Grunt Master 3000 Guide HTML 5 deliverable -->
```

```
  <context name="Grunt Master 3000 Guide" id="html5_ug_gruntmaster3000">
    <input href="ug_gruntmaster3000.ditamap"/>
    <profile>
      <ditaval href="exclude_pdfonly.ditaval"/>
    </profile>
  </context>

  <deliverable name="Grunt Master 3000 Guide" id="ug_gruntmaster3000_html5">
    <context idref="html5_ug_gruntmaster3000"/>
    <output href="./html5-output"/>
    <publication transtype="html5">
      <param name="args.css" href="../css/custom.css"/>
      <param name="args.copycss" value="yes"/>
    </publication>
  </deliverable>

<--! Grunt Master 3000 Guide PDF deliverables -->

  <context name="Grunt Master 3000 Guide" id="pdf_ug_gruntmaster3000">
    <input href="ug_gruntmaster3000.ditamap"/>
    <profile>
      <ditaval href="exclude_html5only.ditaval"/>
    </profile>
  </context>

 <deliverable name="Grunt Master 3000 Guide" id="ug_gruntmaster3000_custpdf">
    <context idref="pdf_ug_gruntmaster3000"/>
    <output href="./pdf-output"/>
    <publication transtype="custom_pdf">
      <param name="args.chapter.layout" href="BASIC"/>
      <param name="outputFile.base" value="Grunt Master 3000 Guide"/>
    </publication>
  </deliverable>

 <deliverable name="Grunt Master 3000 Guide" id="ug_gruntmaster3000_pdf">
    <context idref="pdf_ug_gruntmaster3000"/>
    <output href="./pdf-output"/>
    <publication transtype="pdf2">
      <param name="args.chapter.layout" href="BASIC"/>
      <param name="outputFile.base" value="Grunt Master 3000 Guide"/>
    </publication>
  </deliverable>

</project>
```

The GitHub Actions workflow file build section required to produce all three versions looks something like this:

```
...

jobs:
  build-dita:
    name: Build DITA
    runs-on: ubuntu-latest
    steps:
      - name: Git checkout
        uses: actions/checkout@v2
      - name: Build HTML5
        id: DITA-build
        uses: dita-ot/dita-build-action@master
        with:
          install: |
            dita install fox.jason.extend.css
            dita install net.infotexture.dita-bootstrap
            dita install fox.jason.prismjs
            dita install https://github.com/myCompany/DITA-plugins/archive/custom_pdf.zip
          build:  |
            dita --project=gm_3000.xml  --theme=mytheme.xml
...
```

Note: The first 3 plugins are used for the HTML5 build. The last for the custom PDF build. There is no need to install the PDF2 plugin as it is currently bundled with each DITA Open Toolkit release. The theme file is required for the PDF2 build. If not included, the output will revert to the default PDF2 styles. The theme file parameter is ignored by the other deliverable builds.