Smart Docs Cheat Sheet

Templates

We maintain a set of templates for writing Smart Docs on https://github.com/SUSE/doc-modular/tree/main/templates. To start your own Smart Docs project, grab these and start from there.

Definitions

Assemblies XML files that determine an article's structure, topic order and hierarchy, and metadata.
Topics Stand-alone logical units that are used to build an output document.
Common files Licenses and entity declarations.
Snippets Small, reusable text snippets such as warnings etc.

Smart Docs directory structure

```
| -- DC-<article>
| -- articles/
|    | -- <article>.asm.xml
| -- common/
|    | -- *.ent
|    | -- <license>.xml
| -- snippets/
|    | -- <snippet>.xml
| -- concepts/
|    | -- <concept>.xml
| -- glues/
|    | -- <glue>.xml
| -- references/
|    | -- <reference>.xml
| -- tasks/
|    | -- <task>.xml
| -- images/
|    | -- *.(dia|ditaa|svg|png|jpg|odg)
```

Controlling the article's processing

The article's processing is still controlled by the DC file. The DC file resides on the topmost level in the project's file system. The Smart Doc-related settings in a DC file:

```
MAIN="<article>.asm.xml"
SRC_DIR="articles"
IMG_SRC_DIR="images"
```

There is one DC file per article. To build and validate, topics must be part of an assembly that is referenced by the **`MAIN`** variable set in the DC file.
The DC file name determines the article's URL and is thus subject to SEO rules (human-readable, hyphenated). In short:
`<TOPIC>[-SUBTOPIC]-<TITLE>`
An example DC file name:
`DC-systemd-working-with`
The DC file name must match the article/assembly file name. Once a DC file is named, never change the name again!
The DC file also determines the directory where the article specification (assembly) is kept (**`SRC_DIR`**). Image files reside under the directory specified by **`IMG_SRC_DIR`**. Once we implement profiling, profiling instructions will also be specified here.

Shaping the article

The assembly file must reside in the **`articles/`** directory and match the name of the DC file. In our example: **`systemd-working-with.asm.xml.`**
Append **`.asm.xml`** to distinguish between ordinary XML files and assembly files.

Creating and maintaining topics

All the information in an article is maintained in topic files. Each topic file must have a title or info element as the first element after topic. Depending on their type, they reside in different directories (**`concepts/`**, **`glues/`**, **`references/`** and **`tasks/`**).

Naming conventions:

- Start with the overarching topic. In our example: **`systemd-`**
- Add the topic title next. Use gerunds for task topics and nouns for concept or reference topics. For glue files, naming depends on whether it is a descriptive (noun) or a task-based glue topic (gerund). Examples:
  `tasks/systemd-creating-timers`
  `concepts/systemd-timers`
  `references/systemd-configuration-options`
- IDs are derived from the file name and should be human-readable as they will be visible to the world.

Maintaining image files

Images of all supported formats reside directly under **`images/.`**
There is no distinction between formats. Supported formats: dia, ditaa, svg, png, jpg, odg.
Naming for images follows the same conventions as for topics:.
`images/systemd-configuring-resources-start.<file format>`

Creating a Smart Docs article from scratch

1. Check out the template directory structure and files from https://github.com/SUSE/doc-modular/tree/main/templates.
2. Name your files:
   - assembly
   - topics
   - images
3. Open the **`assembly`** file and specify all the resources (files) needed to build your article:
   - topics
   - license files

   The **`resource`** elements in the assembly hold all files that are pulled in to build the article. The **`href`** attribute points to the relative path of the referenced file, the **`xml:id`** attribute provides an internal ID (_ prefix) to the resource under which it can be pulled in. To improve the readability of your assembly file, add an optional (internal) **`description`** to each resource.

   ```
   <resources>
     <resource xml:id="_concept-example"
               href="../concepts/concept.xml">
     <description>Concept example</description>
   </resources>
   ```
4. Enter the basic information and metadata of your article:
   a. If your article refers to a piece of existing SUSE documentation, add an XML comment that provides a link to that resource and vice versa. If this also needs to be done at topic level, add it to the topic files as well.

b. Set the article's title (SEO-compliant, 29 to 55 chars) and optional subtitle.
c. Provide an xml:id.
d. Add the revision history:
   i. Latest revision on top
   ii. Short and meaningful messages that tell readers the purpose of this revision. This might be sent out to subscribers once we set this up.

```
…
<revhistory>
  <title>Changelog</title>
  <revision>
    <revnumer>1</revnumber>
    <date>2023-11-11</date>
    <revdescription>
      <para>Purpose of this revision</para>
    </revdescription>
  </revision>
  …
</revhistory>
…
```

e. Set a **`maintainer`** by adding your e-mail address.
f. Specify a date in **`updated`**. Use the **`YYYY-MM-DD`** format.
g. Provide the architecture information:
```
<meta name="architecture" content="x86;power"/>
```
h. Provide product name and version information:
```
<meta name="productname">
  <productname
version="15-SP4,15-SP5">&sles;</productname>
</meta>
```
i. Provide an SEO-compliant meta **`title`** and **`description`** to be displayed on the search engine result pages. The title can have up to 55 characters, the description is limited to 150.
j. Provide an ultrashort description (**`social-descr`**) that is used for social media shares. Limit this to 55 characters.
k. Determine the **`category`** the article should be listed under. If there is more than one possibility, enter a comma-separated list of them. Categories are predefined.
l. Until the metadata implementation for translation tagging and bug tracker is working, use the **`dm:`** tags provided with the template.
m. Enter a short descriptive **`abstract`** that states the purpose of your article. Refer to https://documentation.suse.com/smart/systems-management/html/systemd-working-with-timers/index.html for reference.
5. Specify all the resources (files) needed to build your article:
   - topics
   - license files

   The **`resource`** elements in the assembly hold all files that are pulled in to build the article. The **`href`** attribute points to the relative path of the referenced file, the **`xml:id`** attribute provides an internal ID (**_** prefix) to the resource under which it can be pulled in. To improve the readability of your assembly file, add an optional (internal) **`description`** to each resource.
6. Build the article's structure by pulling all the **`modules`** in the order and hierarchy they should appear in the final article.

   A simple example of a module declaration:
```
<module resourceref="_task-example"
renderas="section"/>
```
   The module is referenced by the internal ID. The root element of the topic file is changed to **`section`** by the **`renderas`** attribute.

   To include a module, but switch the existing title to a more appropriate one:
```
<module resourceref="_concept-example"
renderas="section">
  <merge>
    <title>Your new title</title>
  </merge>
</module>
```
   To include a module, strip it of its root and title elements to include the bare contents of this file into your article, pull in the module as follows:
```
<module resourceref="_concept-example"
renderas="section" contentonly="true"
omittitle="true"/>
```

Adjusting topic files using merge

With **`merge`** you can overwrite the title plus any element within **`info`** of the target topic. You can also add elements to the **`info`** section of the result document. Both, the **`info`** element from the topic and what you specify within **`merge`** will be merged (just like merging in git is done).

Examples

Topic document:

```
...
<info>
 <title>foo</title>
 <subtitle>bar</subtitle>
 <abstract><para>foobar</para></abstract>
</info>
…
```

Assembly:

To take the **`info`** element as is, do nothing

```
<module resourceref="_foo" renderas="section"/>
```

This results in:

```
<info>
 <title>foo</title>
 <subtitle>bar</subtitle>
 <abstract><para>foobar</para></abstract>
</info>
```

To add something:

```
<module resourceref="_foo" renderas="section"/>
 <merge>
   <date>2023-04-14</date>
</merge>
```

This results in:

```
<info>
 <title>foo</title>
 <subtitle>bar</subtitle>
 <abstract><para>foobar</para></abstract>
 <date>2023-04-14</date>
</info>
```

To overwrite something:

```
<module resourceref="_foo" renderas="section"/>
 <merge>
   <subtitle>FOOBAR</subtitle>
</merge>
```

This results in:

```
<info>
 <title>foo</title>
 <subtitle>FOOBAR</subtitle>
 <abstract><para>foobar</para></abstract>
</info>
```

You can combine overwriting and adding:

```
<module resourceref="_foo" renderas="section"/>
 <merge>
   <date>2023-04-14</date>
   <subtitle>FOOBAR</subtitle>
</merge>
```

This results in:

```
<info>
 <title>foo</title>
 <subtitle>FOOBAR</subtitle>
 <abstract><para>foobar</para></abstract>
 <date>2023-04-14</date>
</info>
```

As a consequence, if you want to delete something you need to replace it with an empty element:

```
<module resourceref="_foo" renderas="section"/>
 <merge>
   <subtitle/>
  <abstract><para/></abstract>
</merge>
```

This results in:

```
<info>
 <title>foo</title>
</info>
```

`<module resourceref="_foo" contentonly="true"/>` will only delete the root-element of the topic (in our case that would be the **`topic`** element. We probably have little use for this.
`<module resourceref="_foo" omittitles="true"/>` will delete the title element.You can combine **`contentonly="true"`** and **`omittitles="true"`** and use it for nesting topics. This only works if the nested topic does not include an **`info`** element. (Again, probably of little value to us.)

```
<module resourceref="_ref1" renderas="section">
  <merge>
    <title>title</title>
  </merge>
  <module resourceref="_ref2" contentonly="1" omittitles="1"/>
</module>
```

Validating Smart Docs

Before you push your Smart Docs document to the public repository, make sure that your source files are valid:

1. Set up daps to validate against GeekoDoc instead of standard DocBook:
   - If you already have a **`~/.config/daps/dapsrc`** file, add the following line at the end:

     ```
     DOCBOOK5_RNG_URI="urn:x-suse:rng:v2:geekodoc-flat
     ```

     Make sure to add a line break afterwards. If your **`dapsrc`** file contains another line setting **`DOCBOOK5_RNG_URI`**, comment or delete this/these line/s.

   - If you don't have **`~/.config/daps/dapsrc`**, yet, create it as follows (echo command as one continuous line):

     ```
     mkdir ~/.config/daps/
     echo -e 'DOCBOOK5_RNG_URI="urn:x-
     suse:rng:v2:geekodoc-flat"\n' >
     ~/.config/daps/dapsrc
     ```

2. To validate your article assembly file and all files pulled in by it, run:

   ```
   daps -v -d DC-file validate
   ```

3. If the validation is successful, you can proceed to build and/or commit your file(s).

4. If the validation was unsuccessful and you got a list of line numbers and error messages, be aware that these line numbers do not refer to the lines in the assembly file, but rather to those appearing in the bigfile (one big XML file containing the entire source, including all the resolved XIncludes). The bigfile is produced as part of every build and resides under **`build/.assembly/<assembly-name>.xml`**.
   Find the lines containing the errors in the bigfile and then grep for them in the unresolved source files and fix the errors. Repeat this process for all error messages you encounter.

Validating the assembly

daps validate currently cannot validate the assembly file itself. To do that, run:

```
jing -c /usr/share/xml/docbook/schema/rng/5.2/assemblyxi.rnc \
articles/ASSEMBLY.asm.xml
```

Building Smart Docs

To build your final article as HTML, run:

```
daps -v -d DC-file html
```

To build your final article as PDF, run:

```
daps -v -d DC-file pdf
```