Design Document for *docitool* (working name)

Revision `b2ef1ea` (October 25, 2023)

R Breil Soenoto

Table of Contents

Table of Contents **2**

Revision History **3**

Introduction **4**

- Background 4
- Purpose 4
- Definitions 5

Architectural Overview **6**

- Summary 6
- Command Parsing 7
- Heading Detection 8

Implementation Overview **9**

- Commands 9
  - Core commands 9
  - Optional commands 10
    - Citeproc 10
    - PlantUML 13
    - KaTeX 14
    - Matplotlib 14
- Usage 15

Revision History

The following is automatically generated from the `git log` command.

DATE	REVISION ID	AUTHOR	CHANGES
2023-10-25	`b2ef1ea`	Rodriguez Breil Soenoto	move documentation to doc/

Introduction

Background

There is a need to create technical or academic documents using tooling with just the right amount of control. That is, the tooling does not perform unwanted formatting for the user (e.g. Microsoft Word, LibreOffice and similar products), yet is not too complicated or requiring much boilerplate to obtain a pleasing result (e.g. LaTeX). The tooling should be as unobstrusive as possible, yet allows for a predictable result that does not require obscure tricks.

The author believes that HTML and CSS allows for precisely this level of control. HTML's standardized ***SEMANTIC ELEMENTS*** as well as the advanced capabilities of modern CSS allows for adequate representation of a document in both text and print.

This is an HTML preprocessor which aims to provide the user with a means of generating such documents by focusing on the content and style as code, resulting in predictable and more consistent documents. The result can act as both a web page in its own right, or as printable documentation via the use of HTML-to-PDF tooling such as ***WEASYPRINT***.

Purpose

This document aims to outline the specific system in which this document itself is generated with, and serves as a live example with which to demonstrate the system.

Definitions

CWD
Current working directory; the directory the user is currently in when the program is run.

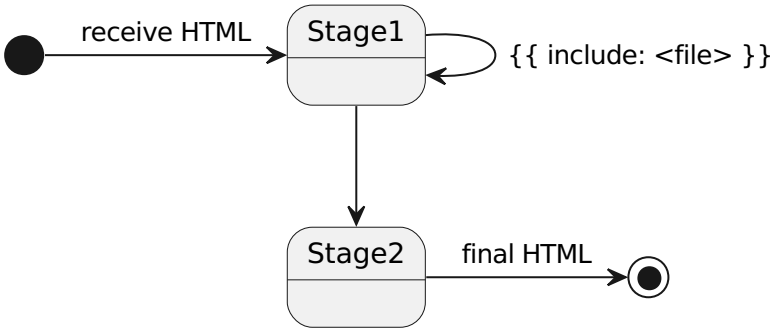
PATH
A list of directories used by the shell or operating system to run binaries from. See details for ***Windows*** and ***Linux***.

TOML
Tom's Obvious Minimal Language. A data description language as an alternative to JavaScript Object Notation (JSON) and YAML Ain't Markup Language (YAML). ***See here*** for information about the file format.

stdout
The default output stream for a terminal. It is reserved for the main output of a program, not for logs or errors.

Architectural Overview

Summary



Application state diagram

The preprocessor receives as its input an HTML file, and outputs another HTML file on stdout. It operates in two stages:

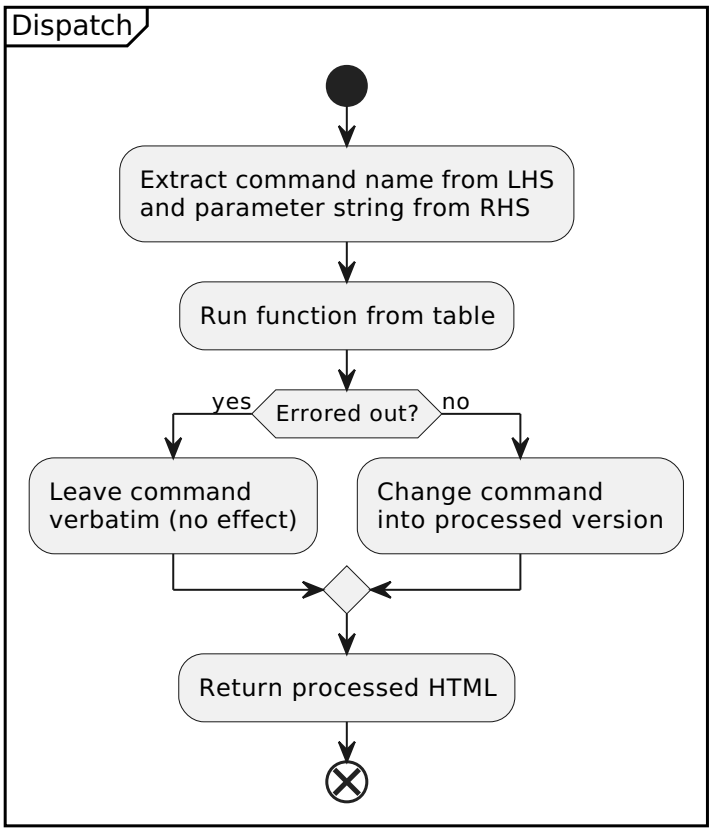
- **Stage 1**: the main processing stage. Commands that only apply to one specific file—such as include and rendering commands—may run here. It should be considered safe to run the commands multiple times in a single conversion run.
- **Stage 2**: post processing stage. Commands that only apply to the resulting document may run here. They will run just before the final file is returned. For example, table of contents, references, and other such commands that require the main content to be "fully baked".

Each stage has its own set of commands which take some state and replaces the command invocation with the result of the command. Dispatchers are used to map the correct command with the correct function in the program.

Command Parsing

In an input HTML file, commands take the form of `{{ command: arguments }}`. If there are no arguments, it becomes simply `{{ command: }}`. A simple split is performed on the `:` character, such that the command name ends up on the left hand side, and the arguments on the right hand side.

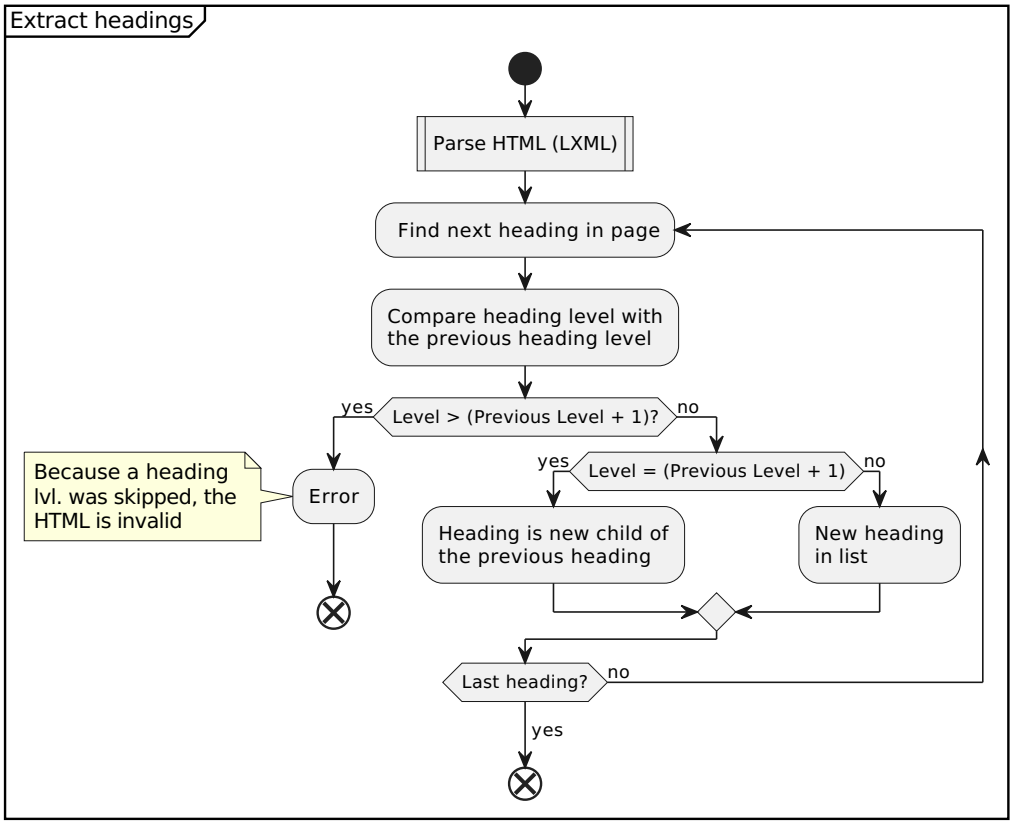
The dispatcher takes the command name and uses a look up table to determine the correct function. If the dispatcher is unable to find such a function for the invoked command, or the invocation itself is malformed, no replacement is made and the original invocation remains in the resulting HTML document. The process is repeated for all detected command invocations.



Dispatcher

Heading Detection

If table of contents are to be made, the preprocessor enforces correct HTML structuring, as it is necessary to ensure a correctly-formatted table. The preprocessor does this by disallowing skipping heading levels when going down levels (e.g. h2 then h4 or h5). Each heading **must** be followed by one of: content, a heading one or more levels above it, a heading on the same level, or a heading one level below it.



Heading detection

Implementation Overview

The preprocessor requires LXML to be installed.

The preprocessor assumes that heading tags (h2, h3, etc.) *really are* used as headings, and not e.g. as a shorthand for "bigger text". It *does not* allow for skipping headings e.g. h4 after an h2 without an intervening h3. Other than what was just described, it does not validate markup any further than LXML does.

Commands

Every command in a HTML follows this format: `{{ (command): (arguments) }}` If there is no arguments, the format is simply `{{ (command): }}`.

Core commands

The list of core commands is as follows:

`table of contents`
Places a table of contents in the page from every heading tag **except** h1, because an h1 is assumed to be used for the title of the document. It takes no arguments.

> **Arguments**: None

> **Example**: `{{ table of contents: }}`

Each affected heading *must* have an HTML ID defined, because it will be used to link to it in the table of contents.

> **Correct**: `<h2 id="heading">Heading</h2>`

> **Wrong**: `<h2>Heading</h2>`

`include`
Copies the contents of an external file and pastes it right where the command is. Takes one argument: the file itself, which is relative to CWD.

> **Arguments**:
> - File path.

Example: `{{ include: chapters/001.html }}`

`verbatim`

Pastes a piece of text into the document as-is. Useful for escaping preprocessor commands, as is shown here in this document.

> **Arguments**:
> - The raw string to paste in.

Example: `{{ verbatim: {{ include: chapters/999.html }} }}`

`shell`

Pastes the output of an external comamnd. This command is parsed by the user's default shell, operating on CWD.

> **Arguments**:
> - The shell command to execute.

Example: `{{ shell: echo "Hello world" }}`

Optional commands

There are also optional commands, meaning you need to have something else installed to be able to use them.

Citeproc

The following commands require ***CITEPROC-PY*** to be installed in the current Python environment. Additionally, the ***TOML*** library must also be installed to load Citeproc sources.

`add sources from`

Includes all Citeproc entries from a folder. These entries should be in TOML format. Example entries:

Conference paper

```
id = "ordonez16"
type = "paper-conference"

title = "Fault attack on FPGA implementations of Trivium stream cipher"
container-title = "2016 IEEE International Symposium on Circuits and Systems"
event-place = "Montreal, QC, Canada"

DOI = "10.1109/ISCAS.2016.7527302"

[[author]]
given = "F. Eugenio"
family = "Potestad-Ordóñez"

[[author]]
given = "Carlos Jesús"
family = "Jiménez-Fernández"

[[author]]
given = "Manuel"
family = "Valencia-Barrero"

[issued]
date-parts = [[2016,5,25]]
```

Website

```
id = "trivium"
type = "article"

title = "Trivium Specifications"
publisher = "eSTREAM Submitted Papers"
URL = "https://www.ecrypt.eu.org/stream/ciphers/trivium/trivium.pdf"

[[author]]
family = "de Cannière"
given = "Cristophe"

[[author]]
given = "Bart"
```

```
family = "Preneel"

[issued]
date-parts = [[2005,4,29]]

[accessed]
date-parts = [[2023,3,23]]
```

> **Package requirements**:
> - toml
> - citeproc-py
>
> **Arguments**:
> - Folder to include

> **Example**: `{{ add sources from: bibliography/ }}`

`use citation styles from`

Sets the citation style from a file or a built-in e.g. "harvard1". *Can only be set once!*

> **Warning**: Must be used after the `add sources from` command!

> **Package requirements**:
> - citeproc-py
>
> **Arguments**:
> - File to load the CSL style from or a built-in

> **Example**: `{{ use citation styles from: citation_styles/ieee.csl }}`

`ref`

Print out one or more citations in the specified format into the document.

> **Warning**: Must be used after the `add sources from` and `use citation styles from` command!

> **Package requirements**:
> - citeproc-py

Arguments:

- Citation IDs, if multiple separated by commas

Example: `{{ ref: anon07, davie88 }}`

`table of references`

Places a table of references (also known as the bibliography) for the document.

Warning: Must be used after the `add sources from` and `use citation styles from` command! Additionally, at least one `ref` command must be used for the output to be meaningful.

Package requirements:

- citeproc-py

Arguments: None

Example: `{{ table of references: }}`

PlantUML

The preprocessor also supports embedding PlantUML files using the `uml` command. It requires the ***PlantUML*** binary to be installed on your system. It does not require any additional Python packages to be installed.

`uml`

Renders a PlantUML diagram file using an external PlantUML process. The file is relative to CWD. The output will be UML in the form of an *inlined* SVG image.

Warning: Ensure that the `plantuml` binary is available in PATH before using this command!

Arguments:

- The file to render as a UML diagram.

Example: `{{ uml: diagrams/activity.uml }}`

KaTeX

KaTeX can be used to embed mathematical formulas, through the use of the ***MARKDOWN-KATEX*** extension. This library needs to be installed in order to use the following commands:

`formula`

Renders a mathematical formula which KaTeX supports. Use of this command may cause a slight delay when initializing the KaTeX Markdown extension. It is highly recommended to use them with the ***KATEX CSS AND FONTS*** for best results. Additionally, when rendering the HTML as a PDF, include them in your project and then link the CSS in the main HTML file to increase performance.

> **Package requirements**:
> - markdown
> - markdown-katex
>
> **Arguments**:
> - The KaTeX formula file to render inline.

> **Example**: `{{ formula: math/matxmul.katex }}`

Matplotlib

MATPLOTLIB can be used to render out graphs.

`formula`

Renders a graph from a Python script. The program operates on a variable called `plt`. An example program is as follows:

Graph program

```
import numpy as np

def abc():
        return 1001

np.random.seed(abc())
plt.rcdefaults()
fig, ax = plt.subplots()

# Example data
```

```
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

ax.barh(y_pos, performance, xerr=error, align='center', color='blue', ecolor='black')
ax.set_yticks(y_pos)
ax.set_yticklabels(people)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Performance')
ax.set_title('How fast do you want to go today?',{'fontname':'serif'})
```

> **Package requirements**:
> • matplotlib
>
> **Arguments**:
> • The Python script to execute and render as an inline SVG graph.

> **Example**: `{{ graph: graphs/random.matplotlib.py }}`

Usage

The program can currently be run as a command line program, the following is an excerpt of the terminal output when run with no arguments:

```
usage: docitool [-h] [--verbose] [input] [output]

positional arguments:
  input          HTML file, use '-' to pipe from stdin instead
  output         Output HTML file, use '-' to output to stdout instead

optional arguments:
  -h, --help     show this help message and exit
  --verbose, -v  Write more diagnostic output
```

The script being run can be accessed in `docitool/__main__.py`. There are plans to split the code into separate files, and might also implement an add-on system.