Getting Started With Pandoc Markdown

Denice Deatrich

Created: July 2023 Last update: September 4, 2023

Table of Contents

Overview

Documenting with the Markdown language is wonderfully simple and clean. When combined with the capabilities of the Pandoc command set then many options exist for generating elegant documents from a Markdown document.

Though many output formats are possible with Pandoc, the tools provided with this guide generate *html*, *pdf* and *docx* output.

There are some issues with 'docx' documents (!! I need to check some of these points with MS Word once I borrow a laptop from the family):

- A *table of contents* is not generated for Libreoffice/Openoffice. You must open the document in your word processor and ask it to generate a table of contents.
- If you wish to modify the default document style then you can either modify the style of the reference docx file using your word processor, or you can unpack the reference document, modify the *styles.xml* file, and the repack the document. I describe how to unpack the reference file further in this document.
- Images are not elegantly scaled in the docx presentation. Therefore you need to adjust images to fit the required placement.
- You will need to enable a footer in the page style and add the page number for page numbering.

These tools focus on writing computer technical documentation; however they can easily be adapted for other purposes, like writing presentation slides, or scientific latex documents. By modifying style and template parameters you can adapt the tools to your needs.

The platform used for this project is Ubuntu LTS 22.04 on a Raspberry Pi, as documented at:
https://github.com/deatrich/linux-home-server

The version of Pandoc used is 2.9. Any other Linux distribution or UNIX-based system with a recent installation of *Pandoc* should work as well.

If you discover issues with instructions in this document, or have other comments or suggestions then you can contact me via my email address shown on my github project repository page, or you can file specific issues under the 'Issues' tab on the project page.

About this Document

In this guide command-line sessions appear in a pale-yellow box, using a customized *.console* syntax highlighting convention described in my *console syntax highlighting* project.

Simplified command-line prompts have syntax highlighting. As well, explanatory comments starting with two slashes are italicized and are coloured blue:

```
// Usually your prompt would be more complex, something like this:
//    myname@ubuntu:~/.vim/syntax$
// or like this:
//    [desktop /tmp]$
// But I simplify its appearance when illustrating command-line sessions.
// Normal Users' command-line prompt, coloured 'green':
$ some-command
```

```
// The root superuser's prompt, simplifed also, and coloured 'red':
# some-other-command
```

A few other command-line programs with a shell-like interface – *mysql*, *mariadb* and *virsh* – also have syntax highlighting:

```
// mariadb session:
$ mysql -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
...

MariaDB> select concat(user, '@', host) from mysql.global_priv;
+-------------------------+
| concat(user, '@', host) |
+-------------------------+
| ro@192.168.1.%          |
| rw@192.168.1.%          |
| mariadb.sys@localhost   |
| mysql@localhost         |
| root@localhost          |
+-------------------------+
5 rows in set (0.001 sec)
MariaDB> exit;
Bye

// virsh session:
$ sudo virsh
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit

virsh # list --all
 Id   Name          State
------------------------------
 -    rocky8        shut off
 -    rocky8.2      shut off
 -    ubuntu22.04   shut off

virsh # exit
```

Sometimes command output is long and/or uninteresting in the context of this guide. I might show such segments with a ellipsis (**...**)

Sometimes a double-exclamation (**!!**) mark might appear somewhere – this is only a reminder for myself to fix an issue at that point in the documentation. These reminders will eventually disappear.

You can of course use syntax highlighting for various languages in Pandoc's Markdown. Here is an example of a bash script:

```
#!/bin/bash -e

unalias -a
PATH=/usr/bin:/bin

## get the timestamp from the file argument and apply it
update_file_timestamp() {
  file_time=$(git log -1 --pretty=format:%ai "$1")
  touch -d "$file_time" "$1"
}

OLD_IFS=$IFS
IFS=$'\n'

for file in `git ls-files`
do
  if [ -f "$file" ] ; then  ## check for file existence
    update_file_timestamp "$file"
  fi
done

IFS=$OLD_IFS

git update-index --refresh
```

Guide outline

In this guide I cover the following topics. It should be enough to get you started writing documentation in Markdown.

We start by setting up your workspace. We install Pandoc, as well as my favourite editor (or your favourite editor) and a few PDF viewers. Other viewers like Firefox and LibreOffice are typically already installed.

Though spell checking is primitive, we take a look at *aspell*, and I also show how to install the spell-checking Lua filter and how to use it.

Then we pull the files from this project and from the 'console syntax' project on *github.com* and install them into your home directory. We also test the installation by running 'make' on the initial generated project.

I give a quick overview of the purpose of the provided tools and files.

The next section of this guide is an outline of Markdown elements so that you have a quick start on common Markdown elements.

An extra final chapter was added to illustrate that a complete Markdown-based document can be used as a data source to generate yet another form of information in the final document. In this case, a few scripts find all *console* command-line fenced code blocks and extracts a list of all common commands used in the document. The output is inserted into the last chapter. Of course, the one-line definitions of these commands was added manually. It might have been possible to pipe man-page information for each command and fill that in as well:

```
$ for i in cp ls git ... ; do
> man $i | head -4 | tail -1
> done
       cp - copy files and directories
       ls - list directory contents
       git - the stupid content tracker
```

Where to find this guide

HTML, DOCX and PDF versions of this guide can be found at:

HTML: https://deatrich.github.io/doc-with-pandoc-markdown/current/doc-with-pandoc-markdown.html
PDF: https://deatrich.github.io/doc-with-pandoc-markdown/current/doc-with-pandoc-markdown.pdf
DOCX: https://deatrich.github.io/doc-with-pandoc-markdown/current/doc-with-pandoc-markdown.docx

Creating a Pandoc workspace

An installation of Pandoc along with the requisite latex support files is needed. In this document I install Pandoc on Ubuntu LTS 22.04. Other Linux distributions will have differing package name and differing package dependencies.

Installing the packages

Pandoc

```
// Let's see what would be installed, without actually installing it:
$ sudo apt install pandoc
The following additional packages will be installed:
  libcmark-gfm-extensions0.29.0.gfm.3 libcmark-gfm0.29.0.gfm.3 pandoc-data
Suggested packages:
  texlive-latex-recommended texlive-xetex texlive-luatex pandoc-citeproc
  texlive-latex-extra context wkhtmltopdf librsvg2-bin groff ghc nodejs
  php python ruby r-base-core libjs-katex citation-style-language-styles
 ...
Do you want to continue? [Y/n] n
Abort.
```

I have installed pandoc a few times now, and I know how important some of the suggested packages are. Since we will be generating PDF formats then TeX and its various flavours and support files will also be needed.

```
// So I install the suggested packages ... over 100 packages will be pulled in:
$ sudo apt install texlive-latex-recommended texlive-xetex texlive-luatex \
 pandoc-citeproc texlive-latex-extra context texlive-fonts-recommended wkhtmltopdf
...
The following NEW packages will be installed:
  context context-modules dvisvgm feynmf fonts-gfs-artemisia
  fonts-gfs-baskerville fonts-gfs-bodoni-classic fonts-gfs-didot
  fonts-gfs-didot-classic fonts-gfs-gazis fonts-gfs-neohellenic fonts-gfs-olga
  ...
  tex-gyre texlive-base texlive-binaries texlive-extra-utils
  texlive-font-utils texlive-fonts-recommended texlive-latex-base
  texlive-latex-extra texlive-latex-recommended texlive-luatex
  texlive-metapost texlive-pictures texlive-plain-generic texlive-xetex tipa
  tk tk8.6 wkhtmltopdf
 ...
Running updmap-sys. This may take some time... done.
Running mktexlsr /var/lib/texmf ... done.
Building format(s) --all.
        This may take some time... done.
Running mktexlsr /var/lib/texmf ... done.
Building format(s) --all.
        This may take some time... done.
```

Now we install pandoc, which pulls in 'pandoc-data'

```
$ sudo apt install pandoc
 ...
The following additional packages will be installed:
  pandoc-data
 ...
```

Editors

My editor of choice is *gvim* – that is – *graphical vim*. You will need to use a decent editor. It would be painful I think to write Markdown documentation with a minimalist text editor (like *nano* or *le*).

Perhaps you already have a favourite editor on the Linux platform. Some other text editors in my version of Ubuntu LTS are:

- pluma (`$ sudo apt install pluma`)
- gedit (if not installed then: `$ sudo apt install gedit`)
- kate (`$ sudo apt install kate`)
- emacs (`$ sudo apt install emacs-gtk`)

I provide help syntax highlighting the *console input/output fenced code* with both 'vim/gvim' and 'kate'. If you are not documenting console input/output then it will not matter which editor you use. There are after all more than 130 computer 'languages' which Pandoc's Markdown is able to highlight.

```
// To install graphical vim:
$ sudo apt install vim-gtk3
```

PDF viewers

I like the *evince* PDF viewer on my Ubuntu desktop. The 'table of contents' outline shows up in the left pane on 'evince', and you can toggle it on/off by clicking the 'Side pane' icon in the upper left.

Another PDF viewer is *qpdfview*. In order to see the 'outline', select:

View -> Docks -> Outline

```
// Install the PDF viewers:
$ apt install evince qpdfview
```

An excellent, but non-free PDF viewer is Master PDF Editor.

> *The unregistered version can be used only in personal, noncommercial purposes to view documents, fill PDF forms, comment and print documents. In order to use the application for commercial purposes, and with its full functionality you are required to purchase a license.*

For Ubuntu, Debian and OpenSuse / Red Hat / CentOS you can download a free version. If you are interested in using it for more complex reasons like form-filling, PDF document editing and digital signing, then buy a license.

For x86/x86_64 architectures there are software packages available. However the version of Master PDF Editor for the *arm64* architecture must be installed from the supplied tarball:

Here is an example of installing it on Ubuntu LTS on a Raspberry Pi:

```
$ mkdir ~/temp/
$ cd ~/temp/
$ tar zxf /path/to/master-pdf-editor-5.9.50-qt5.arm64.tar.gz
$ ls -CpF
master-pdf-editor-5/
$ mkdir -p ~/bin/mpdf
$ cp -a master-pdf-editor-5/ ~/bin/mpdf/
// Remove the temporary installation directory
$ rm -rf ~/temp
```

```
// A few extra libraries are needed to run this program:
$ sudo apt install libqt5xml5 libqt5concurrent5

$ cd ~/bin/mpdf/
$ ls -CpF
fonts/  license_en.txt     masterpdfeditor5.desktop  stamps/
lang/   masterpdfeditor5*  masterpdfeditor5.png      templates
$ file masterpdfeditor5
masterpdfeditor5: ELF 64-bit LSB executable, ARM aarch64,
version 1 (GNU/Linux), dynamically linked, interpreter
/lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0,
BuildID[sha1]=94ca9680e3f0ae39e622aa4e2ab9cbc9c189e63e, stripped

$ cd
$ ~/bin/mpdf/masterpdfeditor5 -h

Master PDF Editor 5
Build 5.9.50, arm 64 bit
 ...
```

Spell checking software

The larger the document, the more likely it is that you will appreciate some form of spell checking.

Spell checking in editors

A good editor will offer spell checking as an option; it is typically found in a *Tools* menu of your favourite editor. If you decide that your editor's spell checking is useful then proceed with it. Embedded spell checking is available on gvim, pluma, emacs, kate and gedit. It is even better to have spell checking which is *markdown-aware*; by that I mean it does not try to spell check inside Markdown syntax and elements. You will need to experiment with your editor's spell checking.

Spell checking with native programs

Two widely used open source spell checkers are *aspell* and *hunspell*. Both programs have dictionaries in dozens of languages:

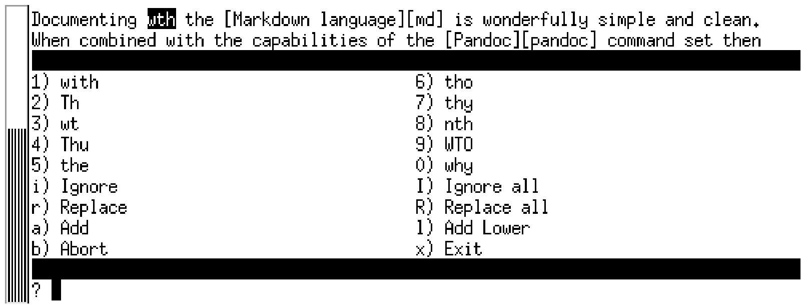
```
$ apt list | egrep '^aspell-' | wc -l
69

$ apt list | egrep '^hunspell-' | grep -v tools | wc -l
75
```

For markdown-based documentation 'aspell' has a markdown-aware option. I gave it a try and at least for me I found it somewhat useful. Here I invoke 'aspell' using the *Markdown* mode on a file named *overview.md*:

```
$  aspell check --dont-backup --mode=markdown overview.md
```

Its spell checking is fairly easy to navigate – the image below shows a terminal session with 'aspell':



```
Documenting wth the [Markdown language][md] is wonderfully simple and clean.
When combined with the capabilities of the [Pandoc][pandoc] command set then

1) with                                6) tho
2) Th                                  7) thy
3) wt                                  8) nth
4) Thu                                 9) WTO
5) the                                 0) why
i) Ignore                              I) Ignore all
r) Replace                             R) Replace all
a) Add                                 l) Add Lower
b) Abort                               x) Exit

?
```

Figure 1: *aspell* in Markdown mode in a terminal window

Spell checking with the Lua filter

Another option is to use the Lua filtering capability of Pandoc. It is primitive, but it has a few useful options:

- you can quickly generate a list of technical words that you can ask *aspell* to ignore
- by eliminating technical words you can quickly find the odd spelling mistake even though it is simply a list of possibly misspelled words.

Software packages for these filters do not exist (or at least I did not find them), so you would need to pull them from github:

```
// get the files
$ mkdir ~/git
$ cd ~/git
$ git clone --depth=1 'https://github.com/pandoc/lua-filters'
$ cd lua-filters
```

You can install only the spell checking file, or you can install all of the filters:

```
// The filters need to be installed in your pandoc configuration directory:
$ mkdir -p ~/.pandoc/filters

// copy only the spelling filter to your work area
$ cp -up spellcheck/spellcheck.lua  ~/pandoc/filters/

// or copy all the filters to your work area using 'find' and looping over them:
$ find . -type f -a  -name \*.lua | \
  while read f ; do \
    echo $f; \
    cp -up $f ~/.pandoc/filters/; \
  done
$ ls -l ~/.pandoc/filters/*spell*
/home/myname/.pandoc/filters/spellcheck.lua
```

Invoke the filter and get a list of possibly misspelled words. You can add them to your personal dictionary:

```
// As an example, use the filter on 'overview.md'
$ pandoc --lua-filter spellcheck.lua overview.md
docx
wth

// So I add docx to my personal dictionary (and fix the spelling of 'with'):
$ nano ~/.aspell.en.pws
$ head ~/.aspell.en.pws
personal_ws-1.1 en 37
backticks
CentOS
```

```
css
dev
docx
fsck
 ...
```

Installing the project files

Getting the files

The files provided with this project can be downloaded from these two required projects found on 'github':

- https://github.com/deatrich/doc-with-pandoc-markdown
- https://github.com/deatrich/console-syntax

You can either use *git* to clone the projects, or pick up *zip* files of the projects.

```
// Make a directory where you can temporarily put the files
$ mkdir ~/temp
$ cd ~/temp

// If you use git:
$ git clone https://github.com/deatrich/doc-with-pandoc-markdown.git
$ git clone https://github.com/deatrich/console-syntax.git

// If you use zip:
// For the zip files, browse to the project web sites, click on 'Code' and then
//  click on 'Download ZIP'
// Then move each downloaded file named 'main.zip' to your temporary directory
//  and extract, or unzip, its contents.

// You will have a directory named 'doc-with-pandoc-markdown' and another
// named 'console-syntax'
```

Generating your document working area

The provided shell script named *generate-new-guide.sh* will generate a new document area where it copies all of the needed files described in this section, and it creates the top-level Markdown file.

Once you have obtained the files as described above, then you can use this script to set up your work area:

```
$ cd ~/temp/doc-with-pandoc-markdown

$ ./generate-new-guide.sh
USAGE:  generate-new-guide.sh -d SOME_DIR -t FILE_NAME
WHERE:  SOME_DIR     specifies a new, non-existent directory
WHERE:  FILE_NAME    specifies the name of the top document without '.md' ending
Example: ./generate-new-guide.sh -d ~/docs/how-to-guide -t How-to-guide

$ ./generate-new-guide.sh -d ~/docs/pet-project -t pet-project
'../console-syntax/console.xml' -> './console.xml'
'../console-syntax/vim.ftdetect' -> './vim.ftdetect'
'../console-syntax/vim.syntax' -> './vim.syntax'
'../console-syntax/vimrc.example' -> './vimrc.example'
```

```
Done.  Do not forget to modify 'metadata.md' and possibly 'Makefile.project'
Please visit /home/myname/docs/pet-project/

// When you are done with the temporary files then delete them if you wish:
$ cd
$ rm -rf ~/temp
```

To test the setup, the generated work area contains an extra Markdown file (overview.md). Once you start writing your own markdown files then add them to the contents.txt file and remove overview.md from the list:

```
$ cd ~/docs/pet-project/
$ ls
console.xml             editorfiles/      metadata.md     style.css
contents.txt            Makefile          overview.md     template.htm
custom-highlight.theme  Makefile.project  pet-project.md  template.latex
custom-reference.docx   md_templates/     scriptfiles/

$ cat contents.txt
metadata.md
overview.md

// run the 'make' command to generate your documents
$ make
pandoc -s metadata.md overview.md ... -V lastupdate="`date +'%d %B %Y'`" -o pet-project.html
pandoc -s metadata.md overview.md --toc ... -o pet-project.docx
pandoc -s metadata.md overview.md --toc ... --pdf-engine=xelatex -o pet-project.pdf
```

Files provided with this project

Here is a list of the pertinent files provided with this project and their purpose.

- Shell scripts:
  - generate-new-guide.sh
  - generate-table.sh
- Build tools:
  - Makefile and Makefile.project
  - contents.txt
- Markdown files:
  - guide-template.md
  - chapter-template.md
  - metadata.md
- Pandoc templates affecting document content and presentation:
  - template.htm
  - template.latex
  - custom-reference.docx
- Style files:
  - custom-highlight.theme
  - style.css
- Editor files and syntax definition for console syntax highlighting:
  - console.xml
  - vim.syntax and vim.ftdetect

– vimrc.example

Shell scripts

generate-new-guide.sh

This is the shell script which generated a new document area for you.

generate-table.sh

If you find yourself creating tables in Markdown then sometimes it is useful to start with an empty table. The included script will print out an empty table, using the specified number of rows and columns; and the table width and the column height in characters. You should copy it into your PATH - normally this means you copy it into ~/bin/

```
$ cp -p /path/to/generate-table.sh ~/bin/
$ chmod 755 ~/bin/generate-table.sh
// We create a table 1 row by 3 columns, 60 chars wide. Row height is 3 chars
$ ./generate-table.sh -r 1 -c 3  -w 60 -rh 3 -header yes
+--------------------+--------------------+--------------------+
|                    |                    |                    |
+====================+====================+====================+
|                    |                    |                    |
|                    |                    |                    |
|                    |                    |                    |
+--------------------+--------------------+--------------------+
```

If you are using the Gvim editor, then you can map a control-key sequence to insert a table into the document by altering your .vimrc file:

```
$ grep generate-table ~/.vimrc
" call generate-table.sh when Control+Z is pressed:
map <C-z> <esc> !} generate-table.sh -r 4 -c 3  -w 70 -rh 3 -header yes<C-m><esc>
```

Build tools

Control document handling: *Makefile* and *Makefile.project*

To avoid entering in repetitive and complex commands it is useful to have a makefile which runs the commands on your behalf.

Take a careful look at these files. The generator script edits Makefile.project and sets the correct name for the project's top file. Other variables that the makefile includes are the PDF, Office and the HTML viewers. Change them if you prefer other viewers.

The rules in this file are mentioned if you ask for help:

```
$ make help
make all          -- update all file types
make test         -- modify then run a test target
make html         -- update the html file
make pdf          -- update the pdf file
make docx         -- update the docx file
make showhtml     -- show the html file
make showpdf      -- show the pdf file
make showdocx     -- show the docx file
make copies       -- push html and pdf copies to generated area
make publish      -- push html and pdf copies to publishing area
make clean        -- clean up generated files

// because the very first rule is 'make all' then when you type
// 'make' it will make all generated files, for example:
$ make
pandoc -s metadata.md ... -c style.css --toc ... -o my-guide.html
pandoc -s metadata.md ... --pdf-engine=xelatex -o my-guide.pdf
pandoc -s metadata.md ... --reference-doc=custom-reference.docx -o my-guide.docx

// you can clean up the generated files at any time:
$ make clean
rm -i *.html *.pdf
rm: remove regular file 'my-guide.html'? y
rm: remove regular file 'my-guide.pdf'? y
rm: remove regular file 'my-guide.docx'? y
```

Ordered list of Markdown files: *contents.txt*

When you start a documentation project it might end up quite large, with many chapters or sections in it. It is annoying to edit one big file. Instead it is more manageable if you create, for example, one file per chapter.

The purpose of *contents.txt* is to simply list the chapters or sections by filename. They must be listed in the order you expect to find them in your final document. Note that *metadata.md* must be the first file.

The 'Makefile' will get the list of Markdown files from this file, and pass the list to the pandoc program.

Markdown files

There are two Markdown files which you can copy to other filenames. They are found in the *md_templates* subdirectory.

Template for the top-level document: *guide-template.md*

This file's sole purpose is the act as the place holder for your named top-level document. In the example above you specified that the document's name would be *pet-project*. The script which set up your work area simply copied *guide-template.md* to *pet-project.md*. Do not bother editing this file, since it is just used by the Makefile as the target name when generating output files.

Template for a new chapter or section: *chapter-template.md*

When creating new Markdown files which are part of your project you can copy *chapter-template.md* to your new file instead of starting from scratch. This file has skeleton Markdown lines in it.

Metadata needed by pandoc: *metadata.md*

The 'metadata' file is used by pandoc to fill in some variables used when generating documents. Most but not all of the metadata are used when generating PDF output. Some variables like 'lastupdate' are auto-generated, so you do not need to change that value.

A note at the top of the file reminds you to change the *title*, *author* and *date* (date is the initial creation date).

Files which affect document content and presentation

Pandoc provides template files which control metadata placement and content ordering for most output formats. You can get the default template by asking pandoc for it on the command-line:

```
// The latex template is used whenever PDF output is generated.
// There is no template for docx, but there is a reference data document.
$ pandoc -D latex > template.latex.default

$ pandoc -D html > template.html.default
$ pandoc --print-default-data-file reference.docx > reference.docx
```

I made a few changes for HTML and for PDF to add a 'Last update' field and I removed the class for the 'date' value. As well, for PDF I modified the latex template to get the table of contents to appear in the bookmarks of a PDF viewer.

template.htm

```
$ cp -p template.html.default template.htm
$ nano template.htm
$ diff template.html.default template.htm
41,42c41
< <p class="author">$author$</p>
< $endfor$
---
> <p class="author">$author$
44c43,46
< <p class="date">$date$</p>
---
>   <br />Created: $date$
> $endif$
> $if(lastupdate)$
>   <br />Last update: $lastupdate$
45a48,49
> </p>
> $endfor$
```

template.latex

```
$ cp -p template.latex.default template.latex
$ nano template.latex
$ diff template.latex.default template.latex
390c390,394
< \date{$date$}
---
> \date{Created: $date$
> $if(lastupdate)$
>   \normalsize{    Last update: \today}
> $endif$}
>
432a437
>   \pdfbookmark[0]{\contentsname}{toc}
440a446
```

custom-reference.docx

You can always generate your own reference.docx file using pandoc rather than using the file included with this project:

```
$ pandoc --print-default-data-file reference.docx > custom-reference.docx
```

Then you can use your word processor and modify the document styles. Note that only modifications that impact the document style are used; any other kind of modification will be ignored by Pandoc.

The provided 'custom-reference.docx' file was modified directly. Here are the instructions for unpacking the docx file, modifying the styles, and re-zipping the reference file:

```
// Start by unzipping the file so that you can access 'styles.xml'
$ mkdir ~/styles
$ cd ~/styles
$ cp /path/to/custom-reference.docx .
$ unzip custom-reference.docx
$ rm custom-reference.docx
$ ls
'[Content_Types].xml'   docProps/   _rels/   word/

$ ls word/
comments.xml  fontTable.xml  numbering.xml  settings.xml  theme/
document.xml  footnotes.xml  _rels/         styles.xml    webSettings.xml

// Edit the styles.xml file:
$ gvim word/styles.xml

// In case you modified the reference docx file with your word processor
// then you might need to first reformat the styles file so that you are able
// to edit it:
$ xmllint --format styles.xml > styles.xml.new
$ mv styles.xml.new styles.xml

// When you are finished then you zip the file back up again.  Be sure to
// create the zip file outside of the your work directory.  In this example
// we used ~/styles/
$ pwd
/home/myname/styles/

// place the docx file above the current directory:
$ zip -r ../custom-reference-new.docx *
```

Here are the XML changes made to the supplied ‘custom-reference.docx’ file:

1. Make fenced code blocks use a better font as well as a smaller font size:

```
<!-- Locate the styleId 'VerbatimChar' -->
<!-- We change Consolas font to 'Courier New' and set the size to 18 -->
<w:style w:type="character" w:customStyle="1" w:styleId="VerbatimChar">
  <w:name w:val="Verbatim Char" />
  <w:basedOn w:val="BodyTextChar" />
  <w:rPr>
    <w:rFonts w:ascii="Courier New" w:hAnsi="Courier New" />
    <w:sz w:val="18" />
  </w:rPr>
</w:style>
```

2. Set the default paragraph alignment to *justified*:

```
<!-- Locate the styleId 'BodyText' -->
<!-- Set the default alignment to be justified, which is named 'both' -->
<w:style w:type="paragraph" w:styleId="BodyText">
  <w:name w:val="Body Text" />
  <w:basedOn w:val="Normal" />
  <w:link w:val="BodyTextChar" />
  <w:pPr>
    <w:spacing w:before="180" w:after="180" />
    <w:jc w:val="both"/>
  </w:pPr>
```

```
    <w:qFormat />
  </w:style>
```

3. Insert a page break before each ‘Heading1’:

```
  <!-- Locate the styleId 'Heading1' and add a pageBreakBefore -->
  <w:style w:type="paragraph" w:styleId="Heading1">
    <w:name w:val="Heading 1" />
    <w:basedOn w:val="Normal" />
    <w:next w:val="BodyText" />
    <w:uiPriority w:val="9" />
    <w:qFormat />
    <w:pPr>
      <w:pageBreakBefore/>
      <w:keepNext />
      <w:keepLines />
      <w:spacing w:before="480" w:after="0" />
```

Files which control style

style.css

You can provide your own CSS[1] files for HTML (and EPUB) output. Add as many style sheets as you want simply by adding multiple *-c* options:

-c thisfile.css -c SOME_URL -c anotherfile.css

Remember that later style sheets can override previous style sheet items when the same element’s style is defined; therefore order is important.

The included *makefile* uses the following CSS style file order. I rather like this latex-like CSS style file from *latex.now.sh*. You can obviously update the Makefile and use other style files which you prefer.

```
$ pandoc ... -c https://latex.now.sh/style.css -c style.css ...
```

Files which control syntax highlighting

Pandoc allows you to add syntax highlighting to enhance and differentiate the syntax in computer languages, scripting or other relevant digital output. Highlighted text visually clarifies the different functionality in the target text.

You can get a list of the languages which Pandoc can present with syntax highlighting:

```
// There are 134 'languages' which can be highlighted:
$ pandoc --list-highlight-languages | wc -l
134
$ pandoc --list-highlight-languages | tail
yaml
yacc
zsh
dot
noweb
rest
sci
sed
xorg
xslt
```

[1]Cascading Style Sheet

There are 8 different syntax highlighting styles provided by Pandoc. To see the effect of these styles, see Garrick's examples.

```
// The default syntax highlighting style is 'pygments'.
$ pandoc --list-highlight-styles
pygments
tango
espresso
zenburn
kate
monochrome
breezedark
haddock
```

But you can also add your own syntax highlighting rules and styles, which is what I have done. I introduced a new syntax highlighting mode for command-line sessions for Markdown documents which is named *console*.

Then *pandoc* consumes the following 2 files and generates the new highlighting in target documents:

```
$ pandoc ... --syntax-definition=console.xml --highlight-style=custom-highlight.theme ...
```

console.xml

This file describes how some console text strings are recognized, and assigns a pre-defined style to them.

This same file can be used to configure the Kate editor so that it is able to do this custom syntax highlighting. See the console syntax highlighting project for information on configuring the Kate editor.

custom-highlight.theme

Then this theme file provides the pre-defined text styles specified by the *console.xml* file.

Editor files for *vim/gvim*

While it is nice to have customized syntax highlighting in your final document, it is also extremely helpful to have your text editor know about the customized syntax highlighting and present similar highlighting.

I have created a syntax configuration file for vim and gvim, my editors of choice. At version 8.2 the editor recognizes over 600 computer languages, scripts and console input/output.

See the console syntax highlighting project for information on configuring vim editors.

Examples of Pandoc's Markdown usage

There are many resources about using Pandoc's Markdown flavour; here are some detailed links to look at:

- The official Pandoc manual
- Introduction to Pandoc
- A tutorial on Markdown and Pandoc
- A cheat sheet for generic Markdown
- A rundown on Markdown elements
- Pandoc Markdown

In this chapter I want to show a summary of common Pandoc's Markdown elements. Not all elements are shown here – I picked the ones that I tend to use. The web links above provide greater detail.

Most elements are easy to remember since Markdown is all about writing simple text.

Except for the example *table* element, all other example elements are embedded in a table with the *element name* in the left column, the *markdown code* in the centre column, and the *generated output* in the right column.

Markdown elements are sometimes explained the way HTML elements are - generally speaking they are divided into two categories: *block* elements and *inline* elements.

Block elements are further divided into *container* blocks and *leaf* blocks. Container block elements may contain other block elements as well as *inline* elements. Leaf elements may only include *inline* elements – the exceptions in this group are code blocks and horizontal rules which cannot contain anything.

Block elements typically span at least an entire line. Block elements are very sensitive to having some white space around them. It is a good idea to put an empty line before and after them.

Inline elements are rather solitary, and typically occur in less than a line of text. The main defining feature though is that an inline element cannot contain other elements.

Leaf Block Elements

Paragraph structures and Headings

Paragraphs are simply text sequences of lines separated by at least one blank line.

Headings are the elements that provide a *table of contents* and give the document a hierarchical structure. I use the ATX-style headings, which use the pound sign – # – and consist of up to 6 levels of heading markers, one for each pound sign. The level with one pound sign is the top level.

Element	Markdown	Output

Headings	`#### Heading 4th level`
	`##### Heading 5th level`
		Heading 4th level
		Heading 5th level

Fenced code blocks

These code blocks are surrounded by a 'fence' in that they are surrounded by a sequence of 3 or more backticks or tildes. All text inside the fence is rendered as-is, without interpretation by the Markdown processor.

Moreover, the code block can be marked by an attribute that labels the kind of code inside the fence. In this guide I use the *console* attribute.

Element	Markdown	Output
Code Blocks	` ```console ` `// This is a comment` `$ echo I am a user` `# echo I am root` ` ``` `	`// This is a comment` `$ echo I am a user` `# echo I am root`

Horizontal rule

Horizontal rules are also known as *theme breaks*.

Element	Markdown	Output
Horizontal rule	`---`	———

Container Block Elements

Block quotations

Quoting something is done with with the greater-than character – > – in front of each line (or in lazy mode, only the first line).

Element	Markdown	Output
Block quote	Here is a quote: `> *this is a quote*`	Here is a quote: *this is a quote*
Block quote	This one is nested: `> this is a quote` `>` `> > a deeper quote`	This one is nested: this is a quote a deeper quote

Lists

Here are 3 kinds of lists; only the *definition list* is typically not implemented in many flavours of Markdown:

Element	Markdown	Output
Bullet List	`* item x` `* item y` `* item z`	• item x • item y • item z

Element	Markdown	Output

Ordered List	`#. item x`
	`#. item y`
	`#. item z`
		1. item x
		2. item y
		3. item z

Definition List	`Some Term:`
	
	`:    Definition 1`
	`:    More information`
	`:    And more info`
		Some Term: Definition 1
		More information
		And more info

Tables

There are a few kinds of tables, but the kind I am using in this document are *grid tables*. Note that tables might confuse your editor if it is trying to do syntax highlighting – that has been my experience with gvim at least.

Generally speaking, tables are a bit fussy to create in many document types. I wrote a shell script to generate empty grid tables that makes it a bit easier to get started with a table.

Tables in Pandoc Markdown act as container block elements, in that they may contain other elements. The exception I have found is other tables; you cannot yet embed a table inside another table.

Here is a modified table borrowed from the Pandoc manual. It also contains alignment hints. Alignment (left, centre, right) in Markdown is indicated by the *colon* character at the upper edge of the first row in the table. So in the following table the left column is left-aligned, the centre column is centred, and the right column is right aligned:

Table 6: This is a table caption.

Fruit	In Stock	Price
Bananas	Yes	$1.34
Oranges	No	$2.10

In Markdown language it looks like the following:

```
+---------------+------------+-------------+
| Fruit         | In Stock   | Price       |
+:==============+:==========:+============:+
| *Bananas*     | Yes        | $1.34       |
+---------------+------------+-------------+
| **Oranges**   | No         | $2.10       |
+---------------+------------+-------------+

Table: This is a table caption.
```

Footnotes and 'inline' notes

To generate a note or footnote, the caret character – ^ – is used:

Element	Markdown	Output
Inline note (no reference needed)	`Inline Eg^[my **note**]`	Inline Eg[2]
Footnote (a reference is needed)	`Footnote Eg[^note]`	Footnote Eg[3]

Element	Markdown	Output
(This is the footnote reference)	`[^note]: I am here..`	(See me at page bottom)

Inline elements

Formatting elements

These elements alter the appearance of enclosed words or phrases:

Element	Markdown	Output
Strikeout	`~~Example~~`	~~Example~~
Emphasis (italics)	`*Example*`	*Example*
Strong Emphasis (bold)	`**Example**`	**Example**
Superscript	`this^Example^`	this$^{\text{Example}}$
Subscript	`this~Example~`	this$_{\text{Example}}$
Underline (works in html output)	`[Example]{.underline}`	Example
Underline (works in pdf output)	`\underline{Example}`	$\underline{\text{Example}}$
Small Caps	`[Example]{.smallcaps}`	Example
Inline Code	`` `# echo Eg`{.console} ``	`# echo Eg`

Links

There are two kinds of links:

Links to external resources I like to mark a link with a label and list the marked link at the end of the section or at the end of the document.

Links to internal document information Typically you want to link to a heading somewhere in the document. You can add a *label* to a heading so that you can link to it elsewhere. Pandoc automatically generates label names for headings in order to link a table of contents at the beginning of the document.

Element	Markdown	Output

External Link with a label	`Try out [this link][tiny]`
	
	`[tiny]: https://t.ly/`
		Try out this link

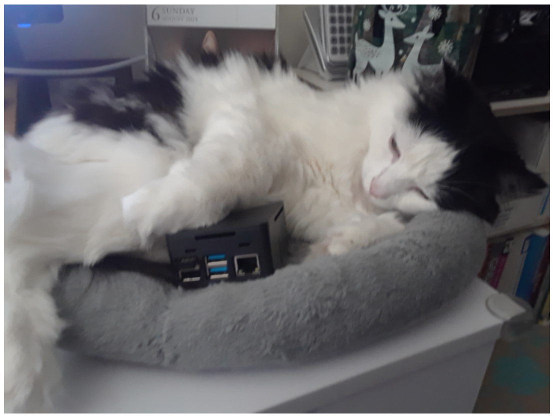
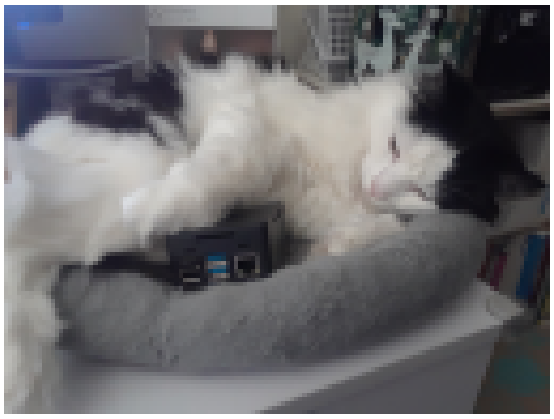
[2] my **note**
[3] I am here..

Element	Markdown	Output

Internal Link	`Look at [topic X](#top-x)`
	
	`(later in the document)`
	
	`#### Topic X {#top-x}`
		Look at topic X
		
		Topic X

Images

The *docx* output format does not handle scaling of images in the output presentation. The first image is 640x480 pixels; the second is 128x96 pixels. Both images are properly presented with *html* and *pdf* output formats.

Element	Markdown	Output
Image	`![Pi](dood.png){width=100%}`	
Image *scaled*	`![Pi](dood2.png){width=100%}`	

List of Common Linux Commands

Because Markdown text is predictable and relatively simple it is possible to use text processing command-line tools to generate other data that might be included in the final Markdown document.

Here we reap a list of command-line tools from *console* sessions and present them in a descriptive list. Each command is linked back to the chapter in which it was found. If a command can be found in more than one chapter then only the first chapter is linked. This requires a little discipline in chapter naming.

If you do not want the chapter links then alter *generate-commands-index.sh* and set 'links' to **no**.

The two included shell scripts used to generate this list are:

get-shell-segments.awk This small awk program looks for commands inside 'console' fenced code for a single Markdown file and prints out each command on a new line.

generate-commands-index.sh This bash script finds chapter links if wanted.

It calls the above awk script for all markdown files in *contents.txt*.

It then does a unique sort on the list and tries to remove duplicates.

Finally for each command it checks if they are in the *commands.md* file; if not it echoes out the command in definition list format.

It is up to you to add the commands to 'commands.md' yourself, as well as a definition for the command.

apt: command-line interface to Debian-based *dpkg* management system
aspell: text curses-based utility for spell checking
cat: concatenate files and print their contents to stdout
cd: change directories to the requested directory
built-in shell command
chmod: changes the mode of a file or directory
cp: copies file(s) and/or directories
diff: reports the difference of 2 target files or directories
file: reports the object type of a file system object
find: search for files in a directory hierarchy
git: utility for the Git revision control system
grep: prints lines matching the requested patterns found in file(s)
gvim: graphical VIM editor
head: print out the first part of files
ls: list file and/or directory contents
make: utility enabling repetitive command execution
mkdir: make one or more directories
mv: renames and/or moves file(s) or directories
mysql: shell interface to MySQL or MariaDB databases
nano: simple curses-based text editor
pandoc: document format converter
pwd: returns the *present working directory*
rm: removes file(s) or directories
sudo: execute a command as another user

tar: tape or file hierarchy archiving utility
unzip: extract compressed files from a ZIP archive
virsh: shell interface to virtual guest domains
xmllint: parses, checks and reformats XML files
zip: compress files into a ZIP archive