Org-mode and Pandoc for Technical Writing

Tomáš Kruliš

16. 05. 2021

Contents

List of Tables

List of Figures

List of Listings

Acronyms

Notation	Description
CVCS	cetralized version control system
DVCS	distributed version control system

1 Introduction

Technical writing is an art in its own. When writing a manual, technical writer has to be the middle ground between software developers and users. As a technical writer you have to bring the subject matter, the software you are writing about, closer to its users. Very often you will have to cooperate directly with software developers and using version control tools such as `git` [1] . I found it beneficial to use efficient tools that help you get the task done, and are also extensible to accommodate any need should arise from writing process.

That is why I have decided to make this basic blog, suitable for beginners trying to find good workflow setup for their writing.

1.1 Tools I recommend and rationale why to use them

As noted above, I prefer to use tools that are extensible and allow a lot of options to meet any requirements that I may be facing during writing. The con is, as you might have guessed from the blog title, that these tools are often very complex. Also, I prefer to use open source tools whenever possible.

What does a technical writer need to do his job? For starters, it would be an *text editor*. That is a program that allows you to write text. Simple as that.

I would advise against writing in binary formats like `docx` [3] and `odt` [4] (which are in fact archives of multiple files). The reason being is that it is actually more challenging to maintain consistent document structure and look with writing in those formats. From my experience, it is much more efficient to focus on content as a writer, and leave upholding the formatting and overall consistency on another application.

For choosing a text editor, well, the name of this blog is "Org-mode and Pandoc for Technical Writing," and `org-mode` [5] is a markup format native to program `GNU/Emacs` [6] (further referred to simply as `emacs`). Because `org-mode` is just a markup format, you can actually use any text editor to write text in `org-mode` markup. However, `emacs` has the best support for it. If you insist on using different text editor than `emacs`, some alternatives would be `vim` [7] or `neovim` [8] , `atom` [9] or `notepad++` [10] . In that case, you can skip chapters chapter references to chapters about `emacs` and `doom emacs` command shortcuts, which are focused on specifics of working with `emacs`. But for the full span of these blog posts, I will assume that you are using `emacs` as a text editor.

I have chosen `org-mode` as format for technical writing because it allows great workflow management and is tightly integrated into `emacs`. But it is a markup format that is best used for documentation source, not its "presentation" form.

Nowadays most documentation has to be accessible from web, or must be compatible with web-based applications, which means we need our documentation in html format. Often is very handy having access to offline documentation, for which is best choice the pdf format. I personally have been facing requirements to provide documentation also in `.docx` or `.odt` formats.

2 Basics of Version Control with Git

When you start a technical writing project, you can easily go by with an folder in your PC and thats it. But as the project grows, or you meet with more requirements from the client (maybe like to add another document format), you might wish to be able to test things out, or to comfortably revert from one state of things to another. That can be managed within folders, but sooner or later it will become clumsy and error prone. Other thing is, that client might straight from the beginning want you to contribute to some versioning system, to keep the manual in sync with current version of the software.

For those cases, you will have to use a version control system. Nowadays, the absolutely most used one is `git`. `git` is by definition "distributed version control system (DVCS) " [2] . How does it work? Software acting as version control is checking your files each time it is invoked, storing changes that have been made to those files in its own database. That way, you can revert to any previously recorded state of those files.

Now the "distributed" part means that the whole system does not have a central server (like it is with cetralized version control system (CVCS)), but everything from the repository is mirrored to client local PC, even file history. This decentralized model brings more reliability, since if the server would went out of operation, the full repository can be revived from another clone. Also, these systems cope very well with multiple repository instances, so you can collaborate on your project easily with more people.

One of those DVCS is `git`. It is one of the most used DVCS today, maybe even THE most used DVCS . Especially in open source development.

2.1 `git` in a nutshell

Little contrary to previous description of the version control systems, `git` does not store changes to a file in time. Instead, `git` is taking snapshots of the whole repository filesystem in each commit. If there were no changes to an particular file compared to previous commit, `git` just stores reference to that file. `git` have uses hashes. Everything in `git` is checksummed with SHA-1 hash. SHA-1 hash is 40 character string consisting of hexadecimal characters (0 – 9 and a – f) and calculated based on the file contents. All of `git` actions only add data. It takes an effort to make `git` erase any data.

`git` recognizes three states of files: *modified*, *staged* and *committed*.

- modified means that you have changed the file but have not committed it to your database yet.
- staged means that you have marked a modified file in its current version to go into your next commit snapshot
- committed means that the data is safely stored in your local database

`git` representation of the project has three main parts:

- working directory (project files currently on local PC)
- staging area (file that stores information about staged files)

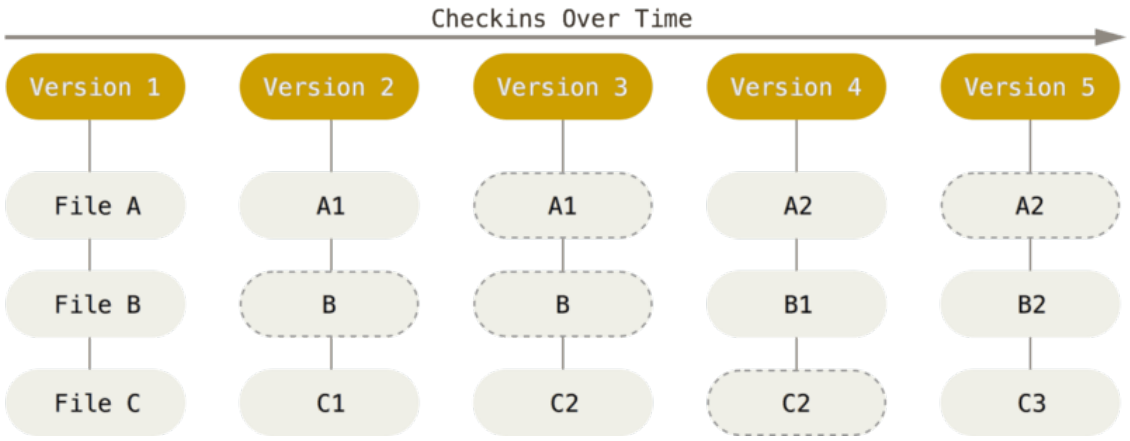

Figure 2.1: `git` storing data as snapshots of the project over time, source: Pro Git Book, [2]

- `.git` directory (stores database of all the objects in your project)

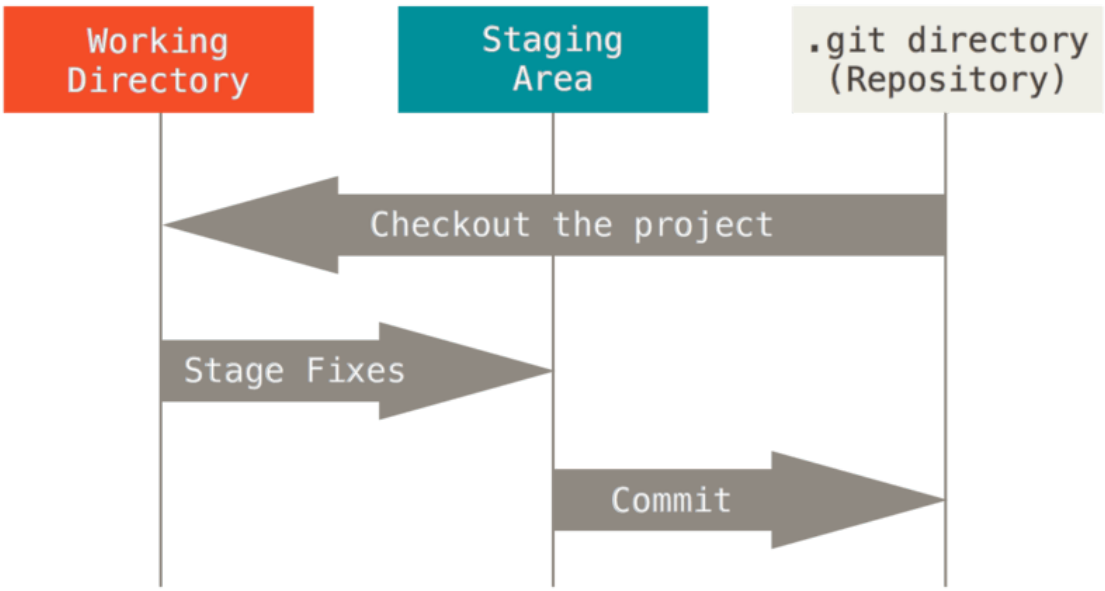

Figure 2.2: Main parts of `git` project repository, source: Pro Git Book, [2]

Basic `git` workflow can be something like this:

1. You modify files in your working directory (working tree)
2. you selectively stage those changes that you want to be part of your next commit. Only those changes are added to the staging area
3. you commit those changes

Committing clears the staging area and uses its content to create new snapshot of your working tree. This snapshot is stored in your `.git` directory.

2.2 How to get git

For getting git on GNU/Linux I would recommend using your distribution package manager. The package will be most probably named "git," so for example with apt package manager on Ubuntu or other debian based GNU/Linux distribution the command would be:

Bash shell listing 1 bash

```
1 sudo apt-get install git
```

The other installation option is compiling git from source code, but that could be for a little bit advanced users.

After the installation you can check whether git is working or not. As with most programs, you can use following command:

Bash shell listing 2 bash

```
1 git --version
```

If you get git version info on the terminal without any error, you are good to go.

2.3 Basic setup

git has its own tool for configuring. You can execute it with command git config. git configuration is divided in three layers: *system*, *global* and *local*.

1. *system*: located in /etc/gitconfig file, requires admin rights to allow any modification. It is accessed via command:

   Bash shell listing 3 bash

   ```
   1 git config --system
   ```

2. *global*: located in ~/.gitconfig or in ~/.config/git/config files is configuration for current user. It is accessed with command:

   Bash shell listing 4 bash

   ```
   1 git config --global
   ```

3. *local*: located in .git/config file in project repository. It is accessed via command:

   Bash shell listing 5 bash

   ```
   1 git config --local
   ```

You can view your current settings with this command:

Bash shell listing 6 bash

```
1 git config --list --show-origin
```

or you can view value of specific configuration variable with command:

Bash shell listing 7 bash

```
1 git config <variable.name>
```

If you didn't set any `git` configuration variables before, it might return nothing. You can set `git` configuration variables with those commands:

Bash shell listing 8 bash

```
1 git config --global user.name "John Doe"
2 git config --global user.email my.email@my.org
3 git config --global core.editor emacs
```

Setting `emacs` as your core editor for dealing with `git` will prove handy later on when we will discover `magit emacs` package. Also, this blog is about `emacs` ... List of all `git` configuration variables can be accessed via `--help` flag:

Bash shell listing 9 bash

```
1 git config --help
```

2.4 First `git` repository

You can get a `git` repository in two ways:

- *clone* an already *existing* repository using its web url address, for example like so:
  přidat git clone odkaz na repositář projektu
  which is actually the repository of this blog and full book releases of this blog.
- you can *create one* yourself on your local PC. To do that, go into your project folder and type command

  Bash shell listing 10 bash

  ```
  1 git init
  ```

Using any of these options you get `.git` folder added to your project folder. This folder contains all of the `git` files necessary to maintain version control over your project. But only by initializing `git` repository doesn't make anything tracked by `git`. There are more steps to achieve that.

First, you have to add files, that you want to have tracked, to the initialized repository. For example:

Bash shell listing 11 bash

```
1 git add *.org
```

which adds to initialized `git` repository all `.org` files for version control. By issuing `git add` command, those files get only into staging area (as described in previous section). After adding the files, you have to commit them:

Bash shell listing 12 bash

```
1 git commit -m "Initial commit."
```

That also means, that you can have in the project folder also files that are not tracked by `git`. `git` doesnt mind. Until you *add* them to `git` tracking that is.

2.5 Working with `git` repository

Basic workflow with `git` could be described as follows:

As you work with your project files, you modify them. If you have added them to `git` tracking, `git` sees them as *modified*, because you changed them from last commit. When you want to make another "version," or project snapshot, you can selectively add each modified file with `git add`, or you can use `--all` flag to add all modified files to `git` staging area. `git add` command is multipurpose command, so you will definitely use it a lot. Finally, you *commit* all the staged files to `git` with `git commit -m "Informative message."` to create new snapshot of files tracked by `git`.

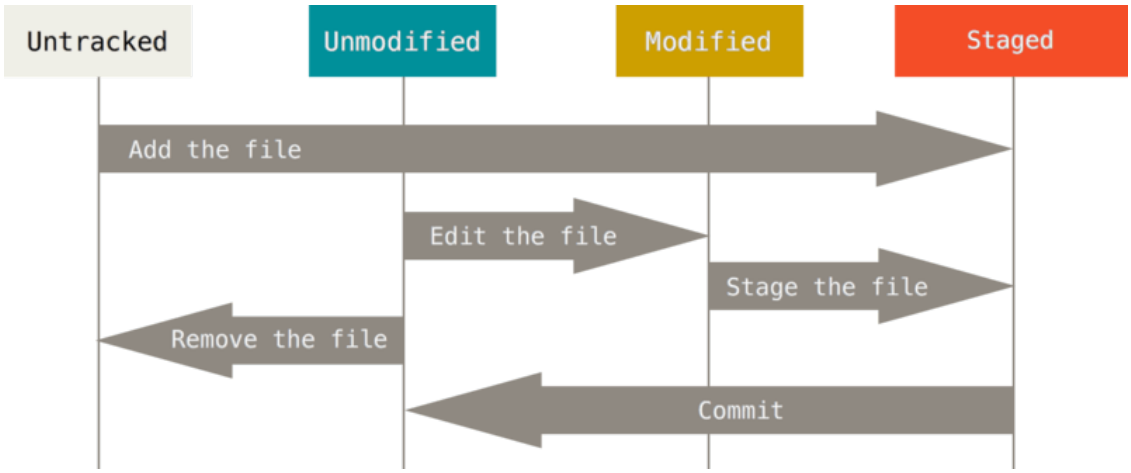

Figure 2.3: `git` workflow and file status changes as you work with them, source: Pro Git Book, [2]

2.6 Checking file status in `git` repository

The main command to find out status of all the files in your repository, that means whether they are *untracked*, *modified*, *staged*, or in-sync with git repository snapshot; as defined above.

Bash shell listing 13 bash

```
1 git status
```

Output message of this command should inform you which branch you are using, about its state compared to the *default branch* and about state of files in repository – whether there are any *staged*, or *modified* files that can be staged or *untracked* files that are not managed by `git`.

There is also short form of the `git` status report, that is invoked with command `git status --short` or `git status -s` that outputs abbreviated information about your `git` repository. In front of every file name is status identifier for staging area and current worktree. These identifiers have following meaning:

- `??` : signals that file is not tracked by `git`, therefore its state is unknown to `git`
- `A` : means tht file was added to the staging area
- `M` : file was modified

For example, identifier `MM` means that file was modified, staged and then modified again before committing previous changes.

To get more detailed view of changes in your repository you can use `git diff` command. The basic command

Bash shell listing 14 bash

```
1 git diff
```

compares state of your working tree with the current contents of staging area. This command shows what changes you have made, but not yet staged since last commit. The other information you might need is the difference between last commit and content of staging area. Next command does that:

Bash shell listing 15 bash

```
1 git diff --staged
```

If the default difference view from `git` in terminal is not suitable for you, or you want to use different tool for viewing repository changes, you can use command

Bash shell listing 16 bash

```
1 git difftool
```

This command uses application of your choosing to view changes. Command `git difftool --tool-help` lists all applications available to be used by `git difftool` command. You may also configure default `difftool` with `git config` (as `git` informs you when you run this command first time) like so:

Bash shell listing 17 bash

```
1 git config --global diff.tool vimdiff
```

that uses vim text editor as program to view changes. Again, when we get to magit package in this blog we will get through settings to use emacs for that. For now, setting up vim is faster.

2.7 Adding files to git tracking

In order to begin tracking new file you have to *add* it to git tracking list via git add command. For example, if you want t start tracking file test.org, you would issue following command in terminal:

Bash shell listing 18 bash

```
1 git add test.org
```

If you run git status command again in the repository, you get information about new file being added to git tracking in output, section "Changes to be committed." If you specify directory name to git add command all the files in thee directory are added recursively to git tracking.

2.8 Staging and committing modified files in git

Modify a file that is already tracked by git and check output of git status command output:

Nejlepší by bylo zde dát příklad.

git informs you that there is a modified file that is not staged for commit. Lets remedy that by putting it into staging area with command git add. As was stated already, git add command has multiple use cases. All of them lead to adding file in its current state to git staging area. If the file is *untracked*, after commit is part of next and following project snapshots, until removed. If *added* file is already *tracked*, git puts its contents again into staging area for next commit.

Bash shell listing 19 bash

```
1 git add test.org
```

If you have more than one modified file and you want to add them all to next repository snapshot you can use command

Bash shell listing 20 bash

```
1 git add --all
```

that add all modified, new or deleted files to next snapshot commit.

After adding file to the next future repository snapshot you have to create that snapshot with command

Bash shell listing 21 bash

```
1 git commit
```

When you issue this command as is, `git` launches text editor specified as `core.editor` (in Basic setup we specified this editor to be `emacs`) to write commit message. The message usually works as description what has been changed with this commit. In the editor text you will also have output of `git status` commented out to remind you what changes are you committing to new repository snapshot. If you want to see even more information when writing your commit message you can use command `git commit -v`, which outputs to your editor also `git diff` output.

This is useful if you want to write long commit message. Otherwise, firing up your editor to write a commit message might be a little clunky. To write down commit message directly into `git commit` command use the `-m` flag like so:

Bash shell listing 22 bash

```
1 git commit -m "Commit message."
```

And finally, if you know that you want to commit to next repository snapshot all staged and modified unstaged files, and you dont want to hassle with `git add` command, you can use

Bash shell listing 23 bash

```
1 git commit -a -m "Commit message."
```

which does exactly that.

2.9 Setting ignored files to `git`

It might happen that you dont want to have some type of files tracked by `git`, for example some log files or files that are created during your build process and are not part of the final release. When we get to talking about LaTeX , you will notice that LaTeX generates pretty big number of build files that are needed to create resulting pdf, but other than that they have not use. For this occasion there is `.gitignore` file.

Basic LaTeX project example `.gitignore` file could look like this:

```
*.aux
```

`.gitignore` file understands basic shell syntax, so in this example makes `git` ignore all files with the `.aux` extension.

You can also ignore full directories. In that case you simply write name of the directory ended with slash `/`:

```
dirname/
```

2.10 File operations in `git` repository

2.10.1 Removing files from `git`

You most assuredly know how to delete files from your computer. In `git` terms, from your local working tree that is. But, if you delete a file in your working tree check `git status` you get information, that this change is recognized and not staged for commit. For staging file deletion also to `git` you have to use `git rm` command:

Bash shell listing 24 bash

```
1 git rm redundant.tmp
```

After next commit the file will be gone from your local working tree and also from `git` tracking. If you have the file already in staging area you have to use `-f` flag to force its removal.

Another remove operation you might need is to get rid of file, that is staged and already tracked, but you dont want to delete it from your local working tree. That could occur in case you forgot to put a file in `.gitignore` file and you are tracking its versions overthought you dont need it. In that case you need to use `--cached` flag to `git rm` command:

Bash shell listing 25 bash

```
1 git rm --cached redundant.tmp
```

If you try to remove directory that way `git` informs you that you have to add `-r` flag to the command, like so:

Bash shell listing 26 bash

```
1 git rm --cached -r folder/
```

These are just multiple safety measures to prevent from accidentally removing files or folders from `git` tracking. You have to be very specific with these commands, which also means that you have to be pretty much conscious about it.

2.10.2 Moving files in repository

`git` doesn't have a notion about moving a file in a repository. `git` has a `git mv` command, that can be used like so:

Bash shell listing 27 bash

```
1 git mv previous-file.txt new-file.txt
```

But according to [2] it is mostly a convenience command for

Bash shell listing 28 bash

```
1 mv previous-file.txt new-file.txt
2 git rm previous-file.txt
3 git add new-file.txt
```

2.11 Viewing commit history

After some time working with `git` repository you might want to view repository commit history. For that there is a simple command:

Bash shell listing 29 bash

```
1 git log
```

After issuing this command you enter interface similar to `manpage`, `cat` or `less` command output – you can scroll the page with `<up>` or `<down>` arrows. When you see `END` written in the bottom of terminal window then you have reached the end of `git` repository history. You can end the pager anytime by pressing `q` which will return you to terminal window.

`git` repository history is displayed in reverse chronological order – newest commit is first, then goes previous, etc …

[2] mentions few usable options to `git log` command.

- `git log --patch` or `git log -p` : shows the difference between repository snapshots introduced by each commit
- `git log -1` or any other number : limits `git log` output to *n* commit entries. You can combine it with `-patch` option
- `git log --stat` : outputs abbreviated output compared to `--patch` option, giving just summaries of how many files were modified and how much rows were changed
- `git log --pretty=format:"<format>"` : powerful option to generate your own format of `git log` output. [2] gives good description of a lot of usable formats.
- `git log --graph` : displays ASCII graph of repository history, including branches
- `git log --after` : displays commits that happened *after* specified date in absolute format *YYYY-MM-DD* or relative
- `git log --grep "text"` : displays only commits with `text` in theyre commit message
- `git log -S myFunction` : searches for commits that introduced `myFunction` in theyre changes to repository
- `git log -- path/to/file` : searches for commits that introduced changed to specified `file`

2.12 Making undos and amends

Záložka: str. 46

2.13 Disclaimer

This chapter was written with heavy support of Pro Git book, see [2]

3 Export from org-mode to LaTeX – basics

– any text –

3.1 Customizing export from org-mode to LaTeX

– any introductive text –

I like to use KOMA-script classes, because they have a lot more customization features and very accessible documentation. However, exporting to KOMA-script classes from `org-mode` is not supported out of the box. To be able to use this document class you have to modify `org-latex-classes` defined in `ox-latex.el`. In standard Emacs, it would be done with this function napiš něco lepšího než tuhle větu:

Emacs lisp listing 1 Elisp

```
(with-eval-after-load 'ox-latex
   (add-to-list 'org-latex-classes
       '("scrbook
         "\\documentclass{scrbook}"
     [NO-DEFAULT-PACKAGES]
     [PACKAGES]
     [EXTRA]"
         ("\\chapter{%s}" . "\\chapter*{%s}")
         ("\\section{%s}" . "\\section*{%s}")
         ("\\subsection{%s}" . "\\subsection*{%s}")
         ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
         ("\\paragraph{%s}" . "\\paragraph*{%s}")
         ("\\subparagraph{%s}" . "\\subparagraph*{%s}"))))
```

Doom Emacs requires only slight modification to this by using `after!` ?macro? instead of `with-eval-after-load` function, so code in our personal `~/.doom.d/config.el` will be:

Emacs lisp listing 2 Elisp

```
(after! 'ox-latex
   (add-to-list 'org-latex-classes
       '("scrbook
         "\\documentclass{scrbook}"
     [NO-DEFAULT-PACKAGES]
     [PACKAGES]
     [EXTRA]"
         ("\\chapter{%s}" . "\\chapter*{%s}")
         ("\\section{%s}" . "\\section*{%s}")
         ("\\subsection{%s}" . "\\subsection*{%s}")
```

```
11          ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
12          ("\\paragraph{%s}" . "\\paragraph*{%s}")
13          ("\\subparagraph{%s}" . "\\subparagraph*{%s}"))))
```

– testing the function interactively –

Now, when we have working interactive export setup, we can move to incorporating that in our `makefile`. This will give us the opportunity to automate whole exporting process. Emacs has nice feature, that allows to run it in batch mode, that is non-interactively, just for evaluating some elisp:

Emacs lisp listing 3 **Elisp**

```
1 emacs --batch --eval="(require 'foo)"
```

This can be used to evaluate exporting function defined in `ox-latex.el` that governs export from `org-mode` to LaTeX – `org-latex-export-to-latex`:

–upravit název org-mode souboru–

Bash shell listing 30 **bash**

```
1  emacs test.org --batch -f org-latex-export-to-latex --kill
2
3  \end{elisplistingbox}
4
5  \%\%blíže vysvětlit \highLight{\texttt{-f} }a \highLight{\texttt{-{}-kill}
    ↪  }flags\%\%
6
7  However, with Doom Emacs there is additional hoop to be overtaken, that is, Doom
    ↪  Emacs make it a~little bit harder to run Emacs in batch mode, because in
    ↪  batch mode Doom Emacs doesnt load any user configuration. That means you
    ↪  have to load it manually:
8
9  \begin{elisplistingbox}
10
11 emacs --batch -l ~/.doom.d/config.el -f org-latex-export-to-latex --kill
12
13 \end{elisplistingbox}
14
15 If you try to do this with your current
    ↪  \highLight{\texttt{\textasciitilde{}/.doom.d/config.el}}, you probably end
    ↪  up with some kind of error about unknown function, in my case it was
    ↪  \highLight{\texttt{after!} }\highLight[red]{?macro?}. I~have found that the
    ↪  easiest option is actually to put this "batch mode configuration" into
    ↪  separate file. Lets name it \highLight{\texttt{latexExportConfig.el} }and
    ↪  put it to our other Doom Emacs configuration files into
    ↪  \highLight{\texttt{\textasciitilde{}/.doom.d} }folder.
16
17 Now we can simply load this file instead of our full Doom Emacs config:
18
19 \begin{bashlistingbox}
```

```
20
21 emacs --batch -l ~/.doom.d/latexExportConfig.el -f org-latex-export-to-latex
   ↪  --kill
```

and %%it **should** work (TM) :)%%

3.2 Resources

- Emacs Wiki: odkaz na Batch mode na Emacs Wiki
- various Stack Exchange questions and answers: Odkaz na general Stack Exchange a na Emacs Stack Exchange

Glossary

C

cetralized version control system

Centralized version control system is version control software that used one central server as main project repository. Clients commit to this one central copy through site connection (web or local).

D

distributed version control system

Distributed version control system is version control software, that mirrors complete project, including its full history, on every clients computer. This model gives good support of branching, allows to work offline and every clients mirror is also a backup of project, which increases reliability f the system.

Index

Bibliography

Versioning Software References

[1] Linus Torvalds. *Git official Webpage*. URL: https://git-scm.com/ (visited on 03/27/2021) (cit. on p. 13).

[2] Scott Chacon and Ben Straub. *Pro Git*. 2020 (cit. on pp. 15, 16, 19, 23, 24).

WSIWYG Office Tools

[3] Microsoft Corp. *Microsoft Word Homepage*. URL: https://www.microsoft.com/en-us/microsoft-365/word (visited on 03/27/2021) (cit. on p. 13).

[4] The Document Foundation. *Libre Office Project Homepage*. URL: https://www.libreoffice.org/ (visited on 03/27/2021) (cit. on p. 13).

Markup Formats suitable for Technical Writing

[5] Bastien Guerry. *Org-mode Official Webpage*. URL: https://orgmode.org/ (visited on 03/28/2021) (cit. on p. 13).

Text Editors

[6] Richard Stallman. *GNU/Emacs Official Webpage*. URL: https://www.gnu.org/software/emacs/ (visited on 03/28/2021) (cit. on p. 13).

[7] Bram Molenaar. *Vim Official Webpage*. URL: https://www.vim.org/ (visited on 03/31/2021) (cit. on p. 13).

[8] Justin M. Keyes. *Neovim Official Webpage*. URL: https://neovim.io/ (visited on 03/31/2021) (cit. on p. 13).

[9] Community Developed. *Atom Text Editor Official Webpage*. URL: https://atom.io/ (visited on 03/31/2021) (cit. on p. 13).

[10] Don Ho. *Notepad++ Official Webpage*. URL: https://notepad-plus-plus.org/ (visited on 03/31/2021) (cit. on p. 13).