Emacs Usage

abdul

September 7, 2025

Contents

1 Emacs Usage **1**
1.1 Keyboard Shortcuts 1
1.2 Tasks 2
1.2.1 Org agenda 2
1.2.2 Tags 3
1.3 Org-babel 3
1.3.1 Source Blocks 3
1.3.2 Org-babel tangle 4
1.4 Lsp-mode 5
1.5 Term mode 5
1.5.1 Some history: 5
1.5.2 Code 6
1.6 Dired 6
1.6.1 Keyboard Shortcuts 6
1.7 Theory 7
1.7.1 Hooks and Modes 7
1.7.2 Universal-argument 8
1.8 Projectile and Magit 8
1.8.1 How to use github with magit? 8

1 Emacs Usage

1.1 Keyboard Shortcuts

Name	Action	Further details
Creating a buffer	C-x 3	Split vertically, side by side
Switching between buffers	C-x o	Place the cursor in the other buffer
Creating a buffer	C-x 1	Split horizontally, one up and one dow
Evaluate buffer	M-x eval-buffer	or alternatively, use **C-c C-e**
Accessing Messages buffer	C-x b **Messages**	
Yank	C-y	or copy from menu bar
Set link	C-c C-l	Example: orgmode manual can be fou
Open link	C-c C-o	
Highlight text	C-spc	Essentially just sets a mark that follo
Tables	Just add headings + TAB	
Bulleted Lists	Shift + left	to cycle between the different modes
Checklists	Just add square brackets + SPC	To check it, put capital X or **C-x C-**
Heading states	C-c C-t / Shift+left	(Todo, done, next)
Bold text	**	Two stars
Italic text	//	Two forward slashes
Insert separator after row	C-e RET	
Code text in line	~~	Two squiggly lines
Source code block (C)	<c + TAB	
Source code block (e-lisp)	<el + TAB	

1.2 Tasks

Commands and shortcuts that help organise daily life

1.2.1 Org agenda

1. a view that will aggregate all tasks in one place so that you may look at all of them at once.

2. Use **org-schedule** command to schedule certain dates (or use **C-c C-s)**

3. Run **org-agenda** command to see the tasks scheduled for certain days.

4. Run **org-refile** command to refile the tasks from the main Tasks.org file to the other headings / subheadings.

5. Run **org-habit** command to schedule a task that repeats.

1.2.2 Tags

- Use **Counsel-org-tag** and then add tags
- Use **C-c C-q** to quickly access tags
- Use :PROPERTIES: and :END: to assign features like
  - **:Effort:** - how long you reck the task would take
- **Shift**+^ - to set the priority

1.3 Org-babel

Configurations and tweaks available when compiling and writing code or config

1.3.1 Source Blocks

- To compile code, just run **C-c C-c**, but first remember to include in init.el

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '((C . t))
 '(Python . t))
```

- To quickly get the source block, just add the < and the shortcut that you added.

  e.g. in the following source block just add the <**code** + tab for the source code block for c to show up:

```
(add-to-list 'org-structure-template-alist '("code" . "src C"))
(add-to-list 'org-structure-template-alist '("el" . "src emacs-lisp"))
```

1.3.2 Org-babel tangle

- This is a way to put all code blocks in one file, e.g let's say you have a bunch of code blocks on one file with normal text surrounding them, like headings, subheadings, etc. This would only take all code blocks and then write them to another file, where you won't see the text. Adding the following text to the top of the org file would tell emacs to tangle all code blocks and write them to example.el.
  - Helpful for when trying to understand a *C programme*, since then you can just write the explanation and separate the code into blocks, but when running the full programme, you can write it to *example.c* (for example) and then just compile the programme separately.
  - All the following does is add the properties to the headers of the code blocks with the following arguments:
    - emacs-lisp
    - :tangle ./example.el

```
#+PROPERTY: header-args:emacs-lisp :tangle ./example.el
```

  - If you don't want all the code blocks to go to that file, only a certain number, then you can just add the following next the *begin*$_{src}$ for the code blocks that you want to have that property:

```
:tangle ./example.el
```

- If you want to save it to a file that is in a folder that does not exist, then you can just set the following property:

```
:mkdirp yes
```

  - Everytime you change the source blocks, just refresh **org-mode** and then run **org-babel-tangle** to tangle all blocks to their respective file.

- To then see the changes in the new file, just **revert-buffer**.
  - Enabling **auto-revert-mode** makes this much easier.

1. Auto-tangling

   - Adding the following code to *init.el* would allow you to auto tangle your file:

```
;; Automatically tangle our Emacs.org config file when we save it
(defun efs/org-babel-tangle-config ()
  (when (string-equal (buffer-file-name)
                      (expand-file-name "~/example.org"))

    ;; Dynamic scoping to the rescue
    (let ((org-confirm-babel-evaluate nil))
      (org-babel-tangle))))

(add-hook 'org-mode-hook (lambda () (add-hook 'after-save-hook #'efs/org-babel
```

1.4 Lsp-mode

Name	Command	Action	Notes
Basic completions	`completion-at-point`		for completions
Find definitions	`lsp-find-definition`	C-c l g r	to find any definitions in the fi
Find references	`lsp-find-reference`	C-c l g g	to find any references in the fil
Rename symbol	`lsp-rename`	C-c l r r	To rename all variables of the
Show diagnostics	`flymake-show-diagnostics-buffer`		To show the error messages be

1.5 Term mode

terminal emulator written in emacs-lisp

1.5.1 Some history:

- Terminals were originally devices that received input from the user and sent it to a remote computer, that then displayed the output.
- A terminal emulator replicates this behavior in software: it takes instructions from **you**, the user and sends them to the CPU or shell, while showing the results on the screen.

1.5.2 Code

```
(use-package eterm-256color
   :hook (term-mode . eterm-256color-mode))
```

1.6 Dired

*A way to effortlessly manage files, can also be accessed through **C-x d** .*

1.6.1 Keyboard Shortcuts

1. Emacs/ Evil

   - `n / j` - next line
   - `p / k` - previous line
   - `j / J` - jump to file in buffer
   - `RET` - select file or directory
   - `^` - go to parent directory
   - `S-RET / g O` - Open file in "other" window
   - `M-RET` - Show file in other window without focusing (previewing files)
   - `g o (dired-view-file) -` Open file but in a "preview" mode, close with q
   - `g / g r` Refresh the buffer with revert-buffer after changing configuration

2. Marking a file

   - `m` - Marks a file
   - `u` - Unmarks a file
   - `U` - Unmarks all files in buffer
   - `*t / t` - Inverts marked files in buffer
   - `% m` - Mark files in buffer using regular expression
   - `*-` Lots of other auto-marking functions

- `k / K` - “Kill” marked items (refresh buffer with g / g r to get them back)

3. Copying and renaming files
    - `C` - Copy marked files (or if no files are marked, the current file)
    - Copying single and multiple files
    - `U` - Unmark all files in buffer
    - `R` - Rename marked files, renaming multiple is a move!
    - `% R` - Rename based on regular expression: ˆtest , old-\
4. Deleting files
    - `D` - Delete marked file
    - `d` - Mark file for deletion
    - `x` - Execute deletion for marks
    - `delete-by-moving-to-trash` - Move to trash instead of deleting permanently
5. Creating and extracting archives
    - `Z` - Compress or uncompress a file or folder to (.tar.gz)
    - `c` - Compress selection to a specific file
    - `dired-compress-files-alist` - Bind compression commands to file extension
6. Other common operations
    - `T` - Touch (change timestamp)
    - `M` - Change file mode
    - `O`- Change file owner
    - `G` - Change file group
    - `S` - Create a symbolic link to this file
    - `L` - Load an Emacs Lisp file into Emacs

1.7 Theory

1.7.1 Hooks and Modes

- A hook function is something that is executed when that mode is active, e.g. term-mode-hook is when the terminal buffer is active.

1.7.2 Universal-argument

- A shortcut like `C-u` that you add before other shortcuts to change their behaviour; can be augmented multiple times to shift the behavioural change, but not commonly used.

1.8 Projectile and Magit

To help manage your github

Key	Action
`C-x g`	Open Magit status
~s	Stage file under cursor
`S`	Stage all files
`u`	Unstage file
`c c`	Commit changes
`C-c C-c`	Finalize commit message
`P p`	Push to GitHub
`F p`	Pull from GitHub
`b b`	Switch/create branch
`l l`	View commit log

1.8.1 How to use github with magit?

1. 1. Open your project in Emacs
    - C-x C-f ~/path/to/your-repo/
    - Navigate to the folder where your repo lives
2. 2. Edit or create files
    - Edit an existing file → save with C-x C-s
    - Create a new file → C-x C-f newfile.txt → write content → C-x C-s
    - Files are now changed locally but not yet tracked by Git
3. 3. Open Magit status
    - C-x g
    - Sections you'll see:
        - Unstaged changes → edited files

    - Untracked files → new files
    - Staged changes → files ready to commit

4. 4. Stage changes

   - Move cursor over a file → press s to stage
   - To stage all files at once → press S (capital S)
   - Staged files move to "Staged changes"

5. 5. Commit changes

   - Press c c → opens commit message buffer
   - Write a meaningful commit message
   - Press C-c C-c to finish commit
   - Changes are now saved in local Git history

6. 6. Push to GitHub

   - Press P → opens push popup
   - If upstream is set:
     - Press p → pushes current branch to GitHub
   - If upstream not set (first push):
     - Press u → select origin → main → sets upstream and pushes
   - After a rebase/conflict, force push if needed:
     - Press P → F (force push to upstream)

7. 7. Verify on GitHub

   - Open your repo page in a browser
   - You should see:
     - New or updated files
     - Commit history reflecting your latest commit

8. 8. Optional: Handle merge conflicts

   - Open conflicted file(s) → resolve manually
   - Stage resolved file → s
   - Commit → c c → C-c C-c
   - Push → P p

9. 9. Repeatable workflow summary

   - Edit/add files → save
   - Magit status → C-x g
   - Stage → s (or S)
   - Commit → c c → C-c C-c
   - Push → P p (or P u for upstream / P F for force push)
   - Refresh GitHub → changes appear