My Doom Emacs configuration, with commentary

POST

My Doom Emacs configuration, with commentary

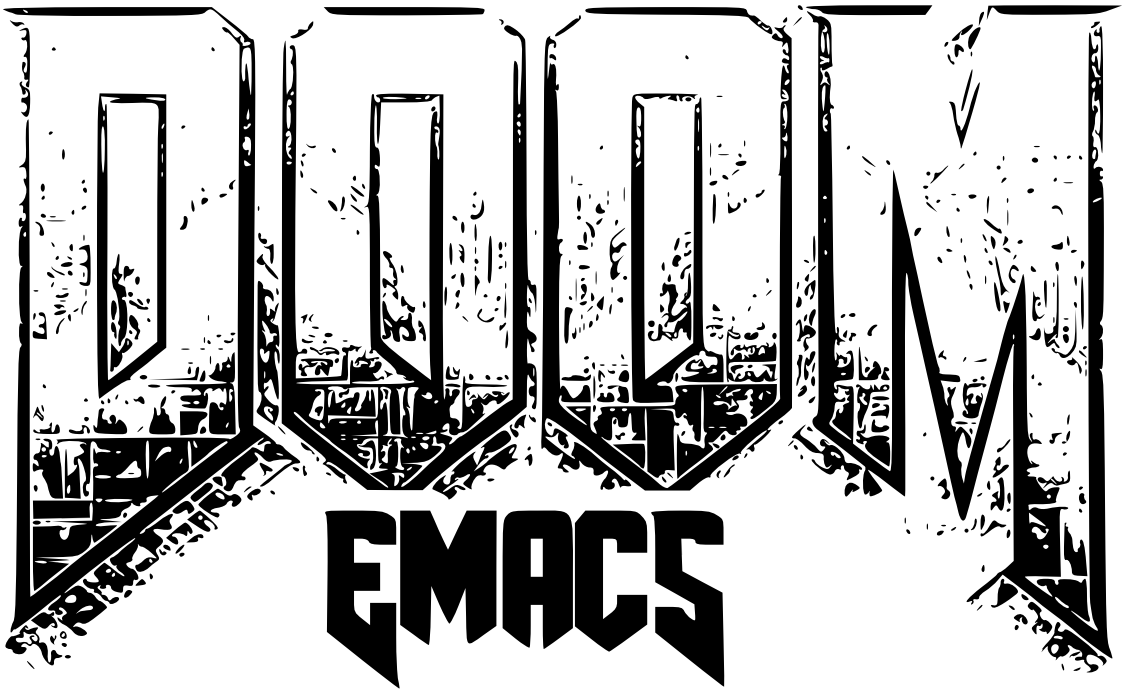
October 19, 2020

Last update: **January 8, 2021**

In my ongoing series of literate config files, I am now posting my Doom Emacs config. I switched to Doom from my hand-crafted Emacs config some time ago, and I have been really enjoying it. Hope you find it useful!

As usual, the post below is included directly from my live doom.org file.

If you are interested in writing your own Literate Config files, check out my book Literate Config on Leanpub!



This is my Doom Emacs configuration. From this org file, all the necessary Doom Emacs config files are generated.

This file is written in literate programming style using org-mode. See init.el, packages.el and config.el for the generated files. You can see this in a nicer format on my blog post My Doom Emacs configuration, with commentary.

References

Emacs config is an art, and I have learned a lot by reading through other people's config files, and from many other resources. These are some of the best ones (several are also written in org mode). You will find snippets from all of these (and possibly others) throughout my config.

- Sacha Chua's Emacs Configuration
- Uncle Dave's Emacs config
- PythonNut's Emacs config
- Mastering Emacs
- Tecosaur's Emacs config

Note: a lot of manual configuration has been rendered moot by using Emacs Doom, which aggregates a well-maintained and organized collection of common configuration settings for performance optimization, package management, commonly used packages (e.g. Org) and much more.

Doom config file overview

Doom Emacs uses three config files:

- `init.el` defines which of the existing Doom modules are loaded. A Doom module is a bundle of packages, configuration and commands, organized into a unit that can be toggled easily from this file.
- `packages.el` defines which packages should be installed, beyond those that are installed and loaded as part of the enabled modules.
- `config.el` contains all custom configuration and code.

There are other files that can be loaded, but theses are the main ones. The load order of different files is defined depending on the type of session being started.

All the config files are generated from this Org file, to try and make its meaning as clear as possible. All `package!` declarations are written to `packages.el`, all other LISP code is written to `config.el`.

Config file headers

We start by simply defining the standard headers used by the three files. These headers come from the initial files generated by `doom install`, and contain either some Emacs-LISP relevant indicators like `lexical-binding`, or instructions about the contents of the file.

- init.el
- packages.el
- config.el

Customized variables

Doom [does not recommend the Emacs `customize` mechanism]:

> *Note: do not use M-x customize or the customize API in general. Doom is designed to be configured programmatically from your config.el, which can conflict with Customize's way of modifying variables.*

All necessary settings are therefore set by hand as part of this configuration file. The only exceptions are "safe variable" and "safe theme" settings, which are automatically saved by Emacs in `custom.el`, but this is OK as they don't conflict with anything else from the config.

Doom modules

This code is written to the `init.el` to select which modules to load. Written here as-is for now, as it is quite well structured and clear.

```
(doom!
 :input
 ;;chinese
 ;;japanese
 ;;layout              ; auie,ctsrnm is the superior home row

 :completion
 (company +childframe) ; the ultimate code completion backend
 ;;helm                ; the *other* search engine for love and life
 ;;ido                 ; the other *other* search engine...
 (ivy +prescient -childframe
      -fuzzy +icons)   ; a search engine for love and life
```

```
       :ui
       ;;deft               ; notational velocity for Emacs
       doom                ; what makes DOOM look the way it does
       doom-dashboard      ; a nifty splash screen for Emacs
       ;;doom-quit         ; DOOM quit-message prompts when you quit Emacs
       ;;fill-column       ; a `fill-column' indicator
       hl-todo             ; highlight TODO/FIXME/NOTE/DEPRECATED/HACK/REVIEW
       ;;hydra
       ;;indent-guides     ; highlighted indent columns
       (ligatures +extra)  ; ligatures or substitute text with pretty symbols
       ;;minimap           ; show a map of the code on the side
       modeline            ; snazzy, Atom-inspired modeline, plus API
       nav-flash           ; blink cursor line after big motions
       ;;neotree          ; a project drawer, like NERDTree for vim
       ophints             ; highlight the region an operation acts on
       (popup +defaults)   ; tame sudden yet inevitable temporary windows
       ;;tabs              ; a tab bar for Emacs
       ;;treemacs          ; a project drawer, like neotree but cooler
       ;;unicode           ; extended unicode support for various languages
       ;;vc-gutter         ; vcs diff in the fringe
       vi-tilde-fringe     ; fringe tildes to mark beyond EOB
       window-select       ; visually switch windows
       workspaces          ; tab emulation, persistence & separate workspaces
       zen                 ; distraction-free coding or writing

       :editor
       ;;(evil +everywhere)  ; come to the dark side, we have cookies
       file-templates      ; auto-snippets for empty files
       ;;fold              ; (nigh) universal code folding
       ;;(format +onsave)  ; automated prettiness
       ;;god               ; run Emacs commands without modifier keys
       ;;lispy             ; vim for lisp, for people who don't like vim
       ;;multiple-cursors  ; editing in many places at once
       ;;objed             ; text object editing for the innocent
       ;;parinfer          ; turn lisp into python, sort of
       ;;rotate-text       ; cycle region at point between text candidates
       snippets            ; my elves. They type so I don't have to
       ;;word-wrap         ; soft wrapping with language-aware indent

       :emacs
       dired               ; making dired pretty [functional]
       electric            ; smarter, keyword-based electric-indent
       ;;ibuffer           ; interactive buffer management
       undo                ; persistent, smarter undo for your inevitable mistake
       vc                  ; version-control and Emacs, sitting in a tree

       :term
       ;;eshell            ; the elisp shell that works everywhere
```

```
       ;;shell              ; simple shell REPL for Emacs
       ;;term               ; basic terminal emulator for Emacs
       vterm               ; the best terminal emulation in Emacs

       :checkers
       (syntax +childframe)  ; tasing you for every semicolon you forget
       spell               ; tasing you for misspelling mispelling
       ;;grammar           ; tasing grammar mistake every you make

       :tools
       ;;ansible
       debugger            ; FIXME stepping through code, to help you add bugs
       ;;direnv
       ;;docker
       ;;editorconfig      ; let someone else argue about tabs vs spaces
       ;;ein               ; tame Jupyter notebooks with emacs
       (eval +overlay)     ; run code, run (also, repls)
       gist                ; interacting with github gists
       lookup              ; navigate your code and its documentation
       lsp
       (magit +forge)      ; a git porcelain for Emacs
       ;;make              ; run make tasks from Emacs
       pass                ; password manager for nerds
       ;;pdf               ; pdf enhancements
       ;;prodigy           ; FIXME managing external services & code builders
       ;;rgb               ; creating color strings
       ;;taskrunner        ; taskrunner for all your projects
       ;;terraform         ; infrastructure as code
       ;;tmux              ; an API for interacting with tmux
       ;;upload            ; map local to remote projects via ssh/ftp

       :os
       (:if IS-MAC macos)  ; improve compatibility with macOS
       ;;tty               ; improve the terminal Emacs experience

       :lang
       ;;agda              ; types of types of types of types...
       ;;cc                ; C/C++/Obj-C madness
       ;;clojure           ; java with a lisp
       common-lisp         ; if you've seen one lisp, you've seen them all
       ;;coq               ; proofs-as-programs
       ;;crystal           ; ruby at the speed of c
       ;;csharp            ; unity, .NET, and mono shenanigans
       ;;data              ; config/data formats
       ;;(dart +flutter)   ; paint ui and not much else
       ;;elixir            ; erlang done right
       ;;elm               ; care for a cup of TEA?
       emacs-lisp          ; drown in parentheses
       ;;erlang            ; an elegant language for a more civilized age
```

```
      (ess +lsp)            ; emacs speaks statistics
      ;;faust               ; dsp, but you get to keep your soul
      ;;fsharp              ; ML stands for Microsoft's Language
      ;;fstar               ; (dependent) types and (monadic) effects and Z3
      ;;gdscript            ; the language you waited for
      (go +lsp)             ; the hipster dialect
      ;;(haskell +dante)    ; a language that's lazier than I am
      ;;hy                  ; readability of scheme w/ speed of python
      ;;idris               ; a language you can depend on
      json                  ; At least it ain't XML
      ;;(java +meghanada)   ; the poster child for carpal tunnel syndrome
      ;;javascript          ; all(hope(abandon(ye(who(enter(here))))))
      ;;julia               ; a better, faster MATLAB
      ;;kotlin              ; a better, slicker Java(Script)
      (latex +latexmk)      ; writing papers in Emacs has never been so fun
      ;;lean
      ;;factor
      ;;ledger              ; an accounting system in Emacs
      lua                   ; one-based indices? one-based indices
      markdown              ; writing docs for people to ignore
      ;;nim                 ; python + lisp at the speed of c
      ;;nix                 ; I hereby declare "nix geht mehr!"
      ;;ocaml               ; an objective camel
      (org +pretty +journal -dragndrop
           +hugo +roam +pandoc
           +present)        ; organize your plain life in plain text
      ;;php                 ; perl's insecure younger brother
      plantuml              ; diagrams for confusing people more
      ;;purescript          ; javascript, but functional
      python                ; beautiful is better than ugly
      ;;qt                  ; the 'cutest' gui framework ever
      racket                ; a DSL for DSLs
      ;;raku                ; the artist formerly known as perl6
      ;;rest                ; Emacs as a REST client
      rst                   ; ReST in peace
      ;;(ruby +rails)       ; 1.step {|i| p "Ruby is #{i.even? ? 'love' : 'life'}"
      ;;rust                ; Fe2O3.unwrap().unwrap().unwrap().unwrap()
      ;;scala               ; java, but good
      ;;scheme              ; a fully conniving family of lisps
      (sh +lsp)             ; she sells {ba,z,fi}sh shells on the C xor
      ;;sml
      ;;solidity            ; do you need a blockchain? No.
      ;;swift               ; who asked for emoji variables?
      ;;terra               ; Earth and Moon in alignment for performance.
      ;;web                 ; the tubes
      (yaml +lsp)           ; JSON, but readable

      :email
      ;;(mu4e +gmail)
```

```
    ;;notmuch
    ;;(wanderlust +gmail)

    :app
    ;;calendar
    irc                 ; how neckbeards socialize
    ;;(rss +org)        ; emacs as an RSS reader
    ;;twitter           ; twitter client https://twitter.com/vnought

    :config
    ;;literate
    (default +bindings +smartparens))
```

General configuration

My user information.

```
(setq user-full-name "Diego Zamboni"
      user-mail-address "diego@zzamboni.org")
```

Change the Mac modifiers to my liking

```
(cond (IS-MAC
       (setq mac-command-modifier      'meta
             mac-option-modifier       'alt
             mac-right-option-modifier 'alt)))
```

When at the beginning of the line, make `Ctrl-K` remove the whole line, instead of just emptying it.

```
(setq kill-whole-line t)
```

Disable line numbers.

```
;; This determines the style of line numbers in effect. If set to `nil', line
;; numbers are disabled. For relative line numbers, set this to `relative'.
(setq display-line-numbers-type nil)
```

For some reason Doom disables auto-save and backup files by default. Let's reenable them.

```
(setq auto-save-default t
      make-backup-files t)
```

Disable exit confirmation.

```
(setq confirm-kill-emacs nil)
```

Visual, session and window settings

I made a super simple set of Doom-Emacs custom splash screens by combining a Doom logo with the word "Emacs" rendered in the Doom Font. You can see them at https://gitlab.com/zzamboni/dot-doom/-/tree/master/splash (you can also see one of them at the top of this file). I configure it to be used instead of the default splash screen. It took me all of 5 minutes to make, so improvements are welcome!

If you want to choose at random among a few different splash images, you can list them in `alternatives`.

```
(let ((alternatives '("doom-emacs-bw-light.svg")))
  ;;((alternatives '("doom-emacs-color.png" "doom-emacs-bw-light.svg")))
  (setq fancy-splash-image
        (concat doom-private-dir "splash/"
                (nth (random (length alternatives)) alternatives))))
```

Set base and variable-pitch fonts. I currently like Fira Code and ET Book.

```
(setq doom-font (font-spec :family "Fira Code" :size 18)
      doom-variable-pitch-font (font-spec :family "ETBembo" :size 18))
```

Allow mixed fonts in a buffer. This is particularly useful for Org mode, so I can mix source and prose blocks in the same document.

```
(add-hook! 'org-mode-hook #'mixed-pitch-mode)
(setq mixed-pitch-variable-pitch-cursor nil)
```

Set the theme to use. I like the Spacemacs-Light, which does not come with Doom, so we need to install it from `package.el`:

```
(package! spacemacs-theme)
```

And then from `config.el` we specify the theme to use.

```
(setq doom-theme 'spacemacs-light)
;;(setq doom-theme 'doom-nord-light)
;;(setq doom-theme 'doom-solarized-light)
```

In my previous configuration, I used to automatically restore the previous session upon startup. Doom Emacs starts up so fast that it does not feel right to do it automatically. In any case, from the Doom dashboard I can simply press Enter to invoke the first item, which is “Reload Last Session”. So this code is commented out now.

```
;;(add-hook 'window-setup-hook #'doom/quickload-session)
```

Maximize the window upon startup. The `(fullscreen . maximized)` value suggested in the Doom FAQ works, but results in a window that cannot be resized.

For now I just manually set it to a large-enough window size by hand.

```
;;(add-to-list 'initial-frame-alist '(fullscreen . maximized))
(setq initial-frame-alist '((top . 1) (left . 1) (width . 129) (height . 37)))
```

Truncate lines in `ivy` childframes. Thanks Henrik!

```
(setq posframe-arghandler
      (lambda (buffer-or-name key value)
        (or (and (eq key :lines-truncate)
                 (equal ivy-posframe-buffer
                        (if (stringp buffer-or-name)
                            buffer-or-name
                          (buffer-name buffer-or-name)))
                 t)
            value)))
```

Key bindings

Doom Emacs has an extensive keybinding system, and most module functions are already bound. I modify some keybindings for simplicity of to match the muscle memory I have from my previous Emacs configuration.

Note: I do not use VI-style keybindings (which are the default for Doom) because I have decades of muscle memory with Emacs-style keybindings. You may need to adjust these if you want to use them.

Miscellaneous keybindings

Use `counsel-buffer-or-recentf` for C-x b. I like being able to see all recently opened files, instead of just the current ones. This makes it possible to use C-x b almost as a replacement for C-c C-f, for files that I edit often. Similarly, for switching between non-file buffers I use `counsel-switch-buffer`, mapped to C-x C-b.

```
(map! "C-x b"   #'counsel-buffer-or-recentf
      "C-x C-b" #'counsel-switch-buffer)
```

The `counsel-buffer-or-recentf` function by default shows duplicated entries because it does not abbreviate the paths of the open buffers. The function below fixes this, I have submitted this change to the `counsel` library (https://github.com/abo-abo/swiper/pull/2687), in the meantime I define it here and integrate it via `advice-add`.

```
(defun zz/counsel-buffer-or-recentf-candidates ()
  "Return candidates for `counsel-buffer-or-recentf'."
  (require 'recentf)
  (recentf-mode)
  (let ((buffers
         (delq nil
               (mapcar (lambda (b)
                         (when (buffer-file-name b)
                           (abbreviate-file-name (buffer-file-name b))))
                       (delq (current-buffer) (buffer-list))))))
    (append
     buffers
     (cl-remove-if (lambda (f) (member f buffers))
                   (counsel-recentf-candidates)))))

(advice-add #'counsel-buffer-or-recentf-candidates
            :override #'zz/counsel-buffer-or-recentf-candidates)
```

The `switch-buffer-functions` package allows us to update the `recentf` buffer list as we switch between them, so that the list produced by `counsel-buffer-or-recentf` is shown in the order the buffers have been visited, rather than in the order they were opened. Thanks to @tau3000 for the tip.

```
(package! switch-buffer-functions)
```

```
(use-package! switch-buffer-functions
  :after recentf
```

```
  :preface
  (defun my-recentf-track-visited-file (_prev _curr)
    (and buffer-file-name
         (recentf-add-file buffer-file-name)))
  :init
  (add-hook 'switch-buffer-functions #'my-recentf-track-visited-file))
```

Use `+default/search-buffer` for searching by default, I like the Swiper interface.

```
;;(map! "C-s" #'counsel-grep-or-swiper)
(map! "C-s" #'+default/search-buffer)
```

Map C-c C-g to `magit-status` - I have too ingrained muscle memory for this keybinding.

```
(map! :after magit "C-c C-g" #'magit-status)
```

Interactive search key bindings - visual-regexp-steroids provides sane regular expressions and visual incremental search. I use the `pcre2el` package to support PCRE-style regular expressions.

```
(package! pcre2el)
(package! visual-regexp-steroids)
```

```
(use-package! visual-regexp-steroids
  :defer 3
  :config
  (require 'pcre2el)
  (setq vr/engine 'pcre2el)
  (map! "C-c s r" #'vr/replace)
  (map! "C-c s q" #'vr/query-replace))
```

The Doom `undo` package introduces the use of `undo-fu`, which makes undo/redo more "lineal". I normally use C-/ for undo and Emacs doesn't have a separate "redo" action, so I map C-? (in my keyboard, the same combination + Shift) for redo.

```
(after! undo-fu
  (map! :map undo-fu-mode-map "C-?" #'undo-fu-only-redo))
```

Replace the default `goto-line` keybindings with `avy-goto-line`, which is more flexible and also falls back to `goto-line` if a number is typed.

```
(map! "M-g g" #'avy-goto-line)
(map! "M-g M-g" #'avy-goto-line)
```

Map a keybindings for `counsel-outline`, which allows easily navigating documents (it works best with Org documents, but it also tries to extract navigation information from other file types).

```
(map! "M-g o" #'counsel-outline)
```

Emulating vi's % key

One of the few things I missed in Emacs from vi was the % key, which jumps to the parenthesis, bracket or brace which matches the one below the cursor. This function implements this functionality, bound to the same key. Inspired by NavigatingParentheses, but modified to use `smartparens` instead of the default commands, and to work on brackets and braces.

```
(after! smartparens
  (defun zz/goto-match-paren (arg)
    "Go to the matching paren/bracket, otherwise (or if ARG is not
    nil) insert %.  vi style of % jumping to matching brace."
    (interactive "p")
```

```
      (if (not (memq last-command '(set-mark
                                    cua-set-mark
                                    zz/goto-match-paren
                                    down-list
                                    up-list
                                    end-of-defun
                                    beginning-of-defun
                                    backward-sexp
                                    forward-sexp
                                    backward-up-list
                                    forward-paragraph
                                    backward-paragraph
                                    end-of-buffer
                                    beginning-of-buffer
                                    backward-word
                                    forward-word
                                    mwheel-scroll
                                    backward-word
                                    forward-word
                                    mouse-start-secondary
                                    mouse-yank-secondary
                                    mouse-secondary-save-then-kill
                                    move-end-of-line
                                    move-beginning-of-line
                                    backward-char
                                    forward-char
                                    scroll-up
                                    scroll-down
                                    scroll-left
                                    scroll-right
                                    mouse-set-point
                                    next-buffer
                                    previous-buffer
                                    previous-line
                                    next-line
                                    back-to-indentation
                                    doom/backward-to-bol-or-indent
                                    doom/forward-to-last-non-comment-or-eol
                                    )))
          (self-insert-command (or arg 1))
        (cond ((looking-at "\\s\(") (sp-forward-sexp) (backward-char 1))
              ((looking-at "\\s\)") (forward-char 1) (sp-backward-sexp))
              (t (self-insert-command (or arg 1))))))
    (map! "%" 'zz/goto-match-paren))
```

Org mode

Org mode has become my primary tool for writing, blogging, coding, presentations and more. I am duly impressed. I have been a fan of the idea of literate programming for many years, and I have tried other tools before (most notably noweb, which I used during grad school for homeworks and projects), but Org is the first tool I have encountered which makes it practical. Here are some of the resources I have found useful in learning it:

- Howard Abrams' Introduction to Literate Programming, which got me jumpstarted into writing code documented with org-mode.
- Nick Anderson's Level up your notes with Org, which contains many useful tips and configuration tricks. Nick's recommendation also got me to start looking into Org-mode in the first place!
- Sacha Chua's Some tips for learning Org Mode for Emacs, her Emacs configuration and many of her other articles.
- Rainer König's OrgMode Tutorial video series.

Doom's Org module provides a lot of sane configuration settings, so I don't have to configure so much as in my previous hand-crafted config.

General Org Configuration

Default directory for Org files.

```
(setq org-directory "~/org/")
```

Hide Org markup indicators.

```
(after! org (setq org-hide-emphasis-markers t))
```

Insert Org headings at point, not after the current subtree (this is enabled by default by Doom).

```
(after! org (setq org-insert-heading-respect-content nil))
```

Enable logging of done tasks, and log stuff into the LOGBOOK drawer by default

```
(after! org
  (setq org-log-done t)
  (setq org-log-into-drawer t))
```

Use the special C-a, C-e and C-k definitions for Org, which enable some special behavior in headings.

```
(after! org
  (setq org-special-ctrl-a/e t)
  (setq org-special-ctrl-k t))
```

Enable Speed Keys, which allows quick single-key commands when the cursor is placed on a heading. Usually the cursor needs to be at the beginning of a headline line, but defining it with this function makes them active on any of the asterisks at the beginning of the line.

```
(after! org
  (setq org-use-speed-commands
        (lambda ()
          (and (looking-at org-outline-regexp)
               (looking-back "^\**")))))
```

Disable electric-mode, which is now respected by Org and which creates some confusing indentation sometimes.

```
(add-hook! org-mode (electric-indent-local-mode -1))
```

Org visual settings

Enable variable and visual line mode in Org mode by default.

```
(add-hook! org-mode :append
           #'visual-line-mode
           #'variable-pitch-mode)
```

Capturing and note taking

First, I define where all my Org-captured things can be found.

```
(after! org
  (setq org-agenda-files
        '("~/gtd" "~/Work/work.org.gpg" "~/org/")))
```

I define some global keybindings to open my frequently-used org files (original tip from [Learn how to take notes more efficiently in Org Mode]).

First, I define a helper function to define keybindings that open files. Note that this requires lexical binding to be enabled, so that the `lambda` creates a closure, otherwise the keybindings don't work.

```
(defun zz/add-file-keybinding (key file &optional desc)
  (let ((key key)
        (file file)
        (desc desc))
    (map! :desc (or desc file)
          key
          (lambda () (interactive) (find-file file)))))
```

Now I define keybindings to access my commonly-used org files.

```
(zz/add-file-keybinding "C-c z w" "~/Work/work.org.gpg" "work.org")
(zz/add-file-keybinding "C-c z i" "~/org/ideas.org" "ideas.org")
(zz/add-file-keybinding "C-c z p" "~/org/projects.org" "projects.org")
(zz/add-file-keybinding "C-c z d" "~/org/diary.org" "diary.org")
```

I'm still trying out `org-roam`, although I have not figured out very well how it works for my setup. For now I configure it to include my whole Org directory.

```
(setq org-roam-directory org-directory)
(setq +org-roam-open-buffer-on-find-file nil)
```

Capturing images

Using `org-download` to make it easier to insert images into my org notes. I don't like the configuration provided by Doom as part of the `(org +dragndrop)` module, so I install the package by hand and configure it to my liking. I also define a new keybinding to paste an image from the clipboard, asking for the filename first.

```
(package! org-download)
```

```
(defun zz/org-download-paste-clipboard (&optional use-default-filename)
  (interactive "P")
  (require 'org-download)
  (let ((file
         (if (not use-default-filename)
             (read-string (format "Filename [%s]: "
                                  org-download-screenshot-basename)
                          nil nil org-download-screenshot-basename)
           nil)))
    (org-download-clipboard file)))

(after! org
  (setq org-download-method 'directory)
  (setq org-download-image-dir "images")
  (setq org-download-heading-lvl nil)
  (setq org-download-timestamp "%Y%m%d-%H%M%S_")
  (setq org-image-actual-width 300)
  (map! :map org-mode-map
        "C-c l a y" #'zz/org-download-paste-clipboard
        "C-M-y" #'zz/org-download-paste-clipboard))
```

Capturing links

Capturing and creating internal Org links

I normally use `counsel-org-link` for linking between headings in an Org document. It shows me a searchable list of all the headings in the current document, and allows selecting one, automatically creating a link to it. Since it doesn't have a keybinding by default, I give it one.

```
(map! :after counsel :map org-mode-map
      "C-c l l h" #'counsel-org-link)
```

I also configure `counsel-outline-display-style` so that only the headline title is inserted into the link, instead of its full path within the document.

```
(after! counsel
  (setq counsel-outline-display-style 'title))
```

`counsel-org-link` uses `org-id` as its backend which generates IDs using UUIDs, and it uses the `ID` property to store them. I prefer using human-readable IDs stored in the `CUSTOM_ID` property of each heading, so we need to make some changes.

First, configure `org-id` to use `CUSTOM_ID` if it exists. This affects the links generated by the `org-store-link` function.

```
(after! org-id
  ;; Do not create ID if a CUSTOM_ID exists
  (setq org-id-link-to-org-use-id 'create-if-interactive-and-no-custom-id))
```

Second, I override `counsel-org-link-action`, which is the function that actually generates and inserts the link, with a custom function that computes and

inserts human-readable `CUSTOM_ID` links. This is supported by a few auxiliary functions for generating and storing the `CUSTOM_ID`.

```
(defun zz/make-id-for-title (title)
  "Return an ID based on TITLE."
  (let* ((new-id (replace-regexp-in-string "[^[:alnum:]]" "-" (downcase title
    new-id))

(defun zz/org-custom-id-create ()
  "Create and store CUSTOM_ID for current heading."
  (let* ((title (or (nth 4 (org-heading-components)) ""))
         (new-id (zz/make-id-for-title title)))
    (org-entry-put nil "CUSTOM_ID" new-id)
    (org-id-add-location new-id (buffer-file-name (buffer-base-buffer)))
    new-id))

(defun zz/org-custom-id-get-create (&optional where force)
  "Get or create CUSTOM_ID for heading at WHERE.

If FORCE is t, always recreate the property."
  (org-with-point-at where
    (let ((old-id (org-entry-get nil "CUSTOM_ID")))
      ;; If CUSTOM_ID exists and FORCE is false, return it
      (if (and (not force) old-id (stringp old-id))
          old-id
        ;; otherwise, create it
        (zz/org-custom-id-create)))))

;; Now override counsel-org-link-action
(after! counsel
  (defun counsel-org-link-action (x)
    "Insert a link to X.

X is expected to be a cons of the form (title . point), as passed
by `counsel-org-link'.

If X does not have a CUSTOM_ID, create it based on the headline
title."
    (let* ((id (zz/org-custom-id-get-create (cdr x))))
      (org-insert-link nil (concat "#" id) (car x)))))
```

Ta-da! Now using `counsel-org-link` inserts nice, human-readable links.

Capturing links to external applications

`org-mac-link` implements the ability to grab links from different Mac apps and insert them in the file. Bind `C-c g` to call `org-mac-grab-link` to choose an application and insert a link.

```
(when IS-MAC
  (use-package! org-mac-link
    :after org
    :config
    (setq org-mac-grab-Acrobat-app-p nil) ; Disable grabbing from Adobe Acrob
    (setq org-mac-grab-devonthink-app-p nil) ; Disable grabbinb from DevonThi
    (map! :map org-mode-map
          "C-c g"  #'org-mac-grab-link)))
```

Tasks and agenda

Customize the agenda display to indent todo items by level to show nesting, and enable showing holidays in the Org agenda display.

```
(after! org-agenda
  (setq org-agenda-prefix-format
        '((agenda . " %i %-12:c%?-12t% s")
          ;; Indent todo items by level to show nesting
          (todo . " %i %-12:c%l")
          (tags . " %i %-12:c")
          (search . " %i %-12:c")))
  (setq org-agenda-include-diary t))
```

Install and load some custom local holiday lists I'm interested in.

```
(package! mexican-holidays)
(package! swiss-holidays)
```

```
(use-package! holidays
  :after org-agenda
  :config
  (require 'mexican-holidays)
  (require 'swiss-holidays)
```

```
  (setq swiss-holidays-zh-city-holidays
        '((holiday-float 4 1 3 "Sechseläuten")
          (holiday-float 9 1 3 "Knabenschiessen")))
  (setq calendar-holidays
        (append '((holiday-fixed 1 1 "New Year's Day")
                  (holiday-fixed 2 14 "Valentine's Day")
                  (holiday-fixed 4 1 "April Fools' Day")
                  (holiday-fixed 10 31 "Halloween")
                  (holiday-easter-etc)
                  (holiday-fixed 12 25 "Christmas")
                  (solar-equinoxes-solstices))
                swiss-holidays
                swiss-holidays-labour-day
                swiss-holidays-catholic
                swiss-holidays-zh-city-holidays
                holiday-mexican-holidays)))
```

org-super-agenda provides great grouping and customization features to make agenda mode easier to use.

```
(package! org-super-agenda)
```

```
(use-package! org-super-agenda
  :after org-agenda
  :config
  (setq org-super-agenda-groups '((:auto-dir-name t)))
  (org-super-agenda-mode))
```

I configure `org-archive` to archive completed TODOs by default to the `archive.org` file in the same directory as the source file, under the "date tree" corresponding to the task's CLOSED date - this allows me to easily separate work from non-work stuff. Note that this can be overridden for specific files by specifying the desired value of `org-archive-location` in the `#+archive:` property at the top of the file.

```
(use-package! org-archive
  :after org
```

```
  :config
  (setq org-archive-location "archive.org::datetree/"))
```

I have started using `org-clock` to track time I spend on tasks. Often I restart Emacs for different reasons in the middle of a session, so I want to persist all the running clocks and their history.

```
(after! org-clock
  (setq org-clock-persist t)
  (org-clock-persistence-insinuate))
```

GTD

I am trying out Trevoke's org-gtd. I haven't figured out my perfect workflow for tracking GTD with Org yet, but this looks like a very promising approach.

```
(package! org-gtd)
```

```
(use-package! org-gtd
  :after org
  :config
  ;; where org-gtd will put its files. This value is also the default one.
  (setq org-gtd-directory "~/gtd/")
  ;; package: https://github.com/Malabarba/org-agenda-property
  ;; this is so you can see who an item was delegated to in the agenda
  (setq org-agenda-property-list '("DELEGATED_TO"))
  ;; I think this makes the agenda easier to read
  (setq org-agenda-property-position 'next-line)
  ;; package: https://www.nongnu.org/org-edna-el/
  ;; org-edna is used to make sure that when a project task gets DONE,
  ;; the next TODO is automatically changed to NEXT.
  (setq org-edna-use-inheritance t)
  (org-edna-load)
  :bind
  (("C-c d c" . org-gtd-capture) ;; add item to inbox
   ("C-c d a" . org-agenda-list) ;; see what's on your plate today
   ("C-c d p" . org-gtd-process-inbox) ;; process entire inbox
   ("C-c d n" . org-gtd-show-all-next) ;; see all NEXT items
```

```
    ;; see projects that don't have a NEXT item
    ("C-c d s" . org-gtd-show-stuck-projects)
    ;; the keybinding to hit when you're done editing an item in the
    ;; processing phase
    ("C-c d f" . org-gtd-clarify-finalize)))
```

Capture templates

We define the corresponding Org-GTD capture templates.

```
(after! (org-gtd org-capture)
  (add-to-list 'org-capture-templates
               '("i" "GTD item"
                 entry
                 (file (lambda () (org-gtd--path org-gtd-inbox-file-basename)
                 "* %?\n%U\n\n  %i"
                 :kill-buffer t))
  (add-to-list 'org-capture-templates
               '("l" "GTD item with link to where you are in emacs now"
                 entry
                 (file (lambda () (org-gtd--path org-gtd-inbox-file-basename)
                 "* %?\n%U\n\n  %i\n  %a"
                 :kill-buffer t))
  (add-to-list 'org-capture-templates
               '("m" "GTD item with link to current Outlook mail message"
                 entry
                 (file (lambda () (org-gtd--path org-gtd-inbox-file-basename)
                 "* %?\n%U\n\n  %i\n  %(org-mac-outlook-message-get-links)"
                 :kill-buffer t)))
```

I set up an advice before `org-capture` to make sure `org-gtd` and `org-capture` are loaded, which triggers the setup of the templates above.

```
(defadvice! +zz/load-org-gtd-before-capture (&optional goto keys)
    :before #'org-capture
    (require 'org-capture)
    (require 'org-gtd))
```

Exporting a Curriculum Vitae

I use `ox-awesomecv` from Org-CV, to export my Curriculum Vitæ.

My `ox-awesomecv` package is not yet merged into the main Org-CV distribution, so I install from my branch for now.

```
(package! org-cv
  :recipe (:host gitlab :repo "zzamboni/org-cv" :branch "awesomecv"))
```

```
(use-package! ox-awesomecv
  :after org)
(use-package! ox-moderncv
  :after org)
```

Publishing to LeanPub

I use LeanPub for self-publishing my books. Fortunately, it is possible to export from org-mode to both LeanPub-flavored Markdown and Markua, so I can use Org for writing the text and simply export it in the correct format and structure needed by Leanpub.

When I decided to use org-mode to write my books, I looked around for existing modules and code. Here are some of the resources I found:

- Description of ox-leanpub.el (GitHub repo) by Juan Reyero;
- Publishing a book using org-mode by Lakshmi Narasimhan;
- Publishing a Book with Leanpub and Org Mode by Jon Snader (from where I found the links to the above).

Building upon these, I developed a new `ox-leanpub` package which you can find in MELPA (source at https://github.com/zzamboni/ox-leanpub), and which I load and configure below.

The `ox-leanpub` module sets up Markua export automatically. I add the code for setting up the Markdown exporter too (I don't use it, but just to keep an eye on any breakage):

```
(package! ox-leanpub
  :recipe (:local-repo "~/Dropbox/Personal/devel/emacs/ox-leanpub"))
```

```
(use-package! ox-leanpub
  :after org
  :config
  (require 'ox-leanpub-markdown)
  (org-leanpub-book-setup-menu-markdown))
```

I highly recommend using Markua rather than Markdown, as it is the format that Leanpub is guaranteed to support in the future, and where most of the new features are being developed.

With this setup, I can write my book in org-mode (I usually keep a single `book.org` file at the top of my repository), and then call the corresponding "Book" export commands. The `manuscript` directory, as well as the corresponding `Book.txt` and other necessary files are created and populated automatically.

If you are interested in learning more about publishing to Leanpub with Org-mode, check out my book *Publishing with Emacs, Org-mode and Leanpub*.

Blogging with Hugo

ox-hugo is an awesome way to blog from org-mode. It makes it possible for posts in org-mode format to be kept separate, and it generates the Markdown files for Hugo. Hugo supports org files, but using ox-hugo has multiple advantages:

- Parsing is done by org-mode natively, not by an external library. Although goorgeous (used by Hugo) is very good, it still lacks in many areas, which leads to text being interpreted differently as by org-mode.
- Hugo is left to parse a native Markdown file, which means that many of its features such as shortcodes, TOC generation, etc., can still be used on the generated file.

Doom Emacs includes and configures `ox-hugo` as part of its `(:lang org +hugo)` module, so all that's left is to configure some parameters to my liking.

I set `org-hugo-use-code-for-kbd` so that I can apply a custom style to keyboard bindings in my blog.

```
(after! ox-hugo
  (setq org-hugo-use-code-for-kbd t))
```

Code for org-mode macros

Here I define functions which get used in some of my org-mode macros

The first is a support function which gets used in some of the following, to return a string (or an optional custom string) only if it is a non-zero, non-whitespace string, and `nil` otherwise.

```
(defun zz/org-if-str (str &optional desc)
  (when (org-string-nw-p str)
    (or (org-string-nw-p desc) str)))
```

This function receives three arguments, and returns the org-mode code for a link to the Hammerspoon API documentation for the `link` module, optionally to a specific `function`. If `desc` is passed, it is used as the display text, otherwise `section.function` is used.

```
(defun zz/org-macro-hsapi-code (module &optional func desc)
  (org-link-make-string
   (concat "https://www.hammerspoon.org/docs/"
           (concat module (zz/org-if-str func (concat "#" func))))
   (or (org-string-nw-p desc)
       (format "=%s="
               (concat module
                       (zz/org-if-str func (concat "." func)))))))
```

Split STR at spaces and wrap each element with the ~ char, separated by +. Zero-width spaces are inserted around the plus signs so that they get formatted correctly. Envisioned use is for formatting keybinding descriptions. There are two versions of this function: “outer” wraps each element in ~, the “inner” wraps the whole sequence in them.

```
(defun zz/org-macro-keys-code-outer (str)
  (mapconcat (lambda (s)
               (concat "~" s "~"))
             (split-string str)
             (concat (string ?\u200B) "+" (string ?\u200B))))
(defun zz/org-macro-keys-code-inner (str)
  (concat "~" (mapconcat (lambda (s)
                           (concat s))
                         (split-string str)
                         (concat (string ?\u200B) "-" (string ?\u200B)))
          "~"))
(defun zz/org-macro-keys-code (str)
  (zz/org-macro-keys-code-inner str))
```

Links to a specific section/function of the Lua manual.

```
(defun zz/org-macro-luadoc-code (func &optional section desc)
  (org-link-make-string
   (concat "https://www.lua.org/manual/5.3/manual.html#"
           (zz/org-if-str func section))
   (zz/org-if-str func desc)))
```

```
(defun zz/org-macro-luafun-code (func &optional desc)
  (org-link-make-string
   (concat "https://www.lua.org/manual/5.3/manual.html#"
           (concat "pdf-" func))
   (zz/org-if-str (concat "=" func "()=") desc)))
```

Reformatting an Org buffer

I picked up this little gem in the org mailing list. A function that reformats the current buffer by regenerating the text from its internal parsed representation. Quite amazing.

```
(defun zz/org-reformat-buffer ()
  (interactive)
  (when (y-or-n-p "Really format current buffer? ")
    (let ((document (org-element-interpret-data (org-element-parse-buffer))))
      (erase-buffer)
      (insert document)
      (goto-char (point-min)))))
```

Avoiding non-Org mode files

org-pandoc-import is a mode that automates conversions to/from Org mode as much as possible.

```
(package! org-pandoc-import
  :recipe (:host github
           :repo "tecosaur/org-pandoc-import"
           :files ("*.el" "filters" "preprocessors")))
```

```
(use-package org-pandoc-import)
```

Reveal.js presentations

I use `org-re-reveal` to make presentations. The functions below help me improve my workflow by automatically exporting the slides whenever I save the file, refreshing the presentation in my browser, and moving it to the slide where the cursor was when I saved the file. This helps keeping a “live” rendering of the presentation next to my Emacs window.

The first function is a modified version of the `org-num--number-region` function of the `org-num` package, but modified to only return the numbering of the

innermost headline in which the cursor is currently placed.

```
(defun zz/org-current-headline-number ()
  "Get the numbering of the innermost headline which contains the
cursor. Returns nil if the cursor is above the first level-1
headline, or at the very end of the file. Does not count
headlines tagged with :noexport:"
  (require 'org-num)
  (let ((org-num--numbering nil)
        (original-point (point)))
    (save-mark-and-excursion
      (let ((new nil))
        (org-map-entries
         (lambda ()
           (when (org-at-heading-p)
             (let* ((level (nth 1 (org-heading-components)))
                    (numbering (org-num--current-numbering level nil)))
               (let* ((current-subtree (save-excursion (org-element-at-point)
                      (point-in-subtree
                       (<= (org-element-property :begin current-subtree)
                           original-point
                           (1- (org-element-property :end current-subtree))))
                 ;; Get numbering to current headline if the cursor is in it.
                 (when point-in-subtree (push numbering
                                              new))))))))
         "-noexport")
        ;; New contains all the trees that contain the cursor (i.e. the
        ;; innermost and all its parents), so we only return the innermost on
        ;; We reverse its order to make it more readable.
        (reverse (car new))))))
```

The `zz/refresh-reveal-prez` function makes use of the above to perform the presentation export, refresh and update. You can use it by adding an after-save hook like this (add at the end of the file):

```
* Local variables :ARCHIVE:noexport:
# Local variables:
# eval: (add-hook! after-save :append :local (zz/refresh-reveal-prez))
# end:
```

Note #1: This is specific to my OS (macOS) and the browser I use (Brave). I will make it more generic in the future, but for now feel free to change it to your needs.

Note #2: the presentation must be already open in the browser, so you must run "Export to reveal.js -> To file and browse" (`C-c C-e v b`) once by hand.

```
(defun zz/refresh-reveal-prez ()
  ;; Export the file
  (org-re-reveal-export-to-html)
  (let* ((slide-list (zz/org-current-headline-number))
         (slide-str (string-join (mapcar #'number-to-string slide-list) "-"))
         ;; Determine the filename to use
         (file (concat (file-name-directory (buffer-file-name))
                       (org-export-output-file-name ".html" nil)))
         ;; Final URL including the slide number
         (uri (concat "file://" file "#/slide-" slide-str))
         ;; Get the document title
         (title (cadar (org-collect-keywords '("TITLE"))))
         ;; Command to reload the browser and move to the correct slide
         (cmd (concat
"osascript -e \"tell application \\\"Brave\\\" to repeat with W in windows
set i to 0
repeat with T in (tabs in W)
set i to i + 1
if title of T is \\\"" title "\\\" then
  reload T
  delay 0.1
  set URL of T to \\\"" uri "\\\"
  set (active tab index of W) to i
end if
end repeat
end repeat\"")))
    ;; Short sleep seems necessary for the file changes to be noticed
    (sleep-for 0.2)
    (call-process-shell-command cmd)))
```

Other exporters

ox-jira to export in Jira markup format.

```
(package! ox-jira)
```

```
(use-package! ox-jira
  :after org)
```

Programming Org

Trying out org-ml for easier access to Org objects.

```
(package! org-ml)
```

```
(use-package! org-ml
  :after org)
```

Coding

Tangle-on-save has revolutionized my literate programming workflow. It automatically runs `org-babel-tangle` upon saving any org-mode buffer, which means the resulting files will be automatically kept up to date.

```
(add-hook! org-mode :append
  (add-hook! after-save :append :local #'org-babel-tangle))
```

Some useful settings for LISP coding - `smartparens-strict-mode` to enforce parenthesis to match. I map `M-(` to enclose the next expression as in `paredit` using a custom function. Prefix argument can be used to indicate how many expressions to enclose instead of just 1. E.g. `C-u 3 M-(` will enclose the next 3 sexps.

```
(defun zz/sp-enclose-next-sexp (num)
  (interactive "p")
  (insert-parentheses (or num 1)))
```

```
(after! smartparens
  (add-hook! (clojure-mode
              emacs-lisp-mode
              lisp-mode
              cider-repl-mode
              racket-mode
              racket-repl-mode) :append #'smartparens-strict-mode)
  (add-hook! smartparens-mode :append #'sp-use-paredit-bindings)
  (map! :map (smartparens-mode-map smartparens-strict-mode-map)
        "M-(" #'zz/sp-enclose-next-sexp))
```

Adding keybindings for some useful functions:

- `find-function-at-point` gets bound to `C-c l g p` (grouped together with other "go to" functions bound by Doom) and to `C-c C-f` (analog to the existing `C-c f`) for faster access.

```
(after! prog-mode
  (map! :map prog-mode-map "C-h C-f" #'find-function-at-point)
  (map! :map prog-mode-map
        :localleader
        :desc "Find function at point"
        "g p" #'find-function-at-point))
```

Some other languages I use.

- Elvish shell, with support for org-babel.

```
(package! elvish-mode)
(package! ob-elvish)
```

- CFEngine policy files. The `cfengine3-mode` package is included with Emacs, but I also install org-babel support.

```
(package! ob-cfengine3)
```

```
(use-package! cfengine
  :defer t
  :commands cfengine3-mode
  :mode ("\\.cf\\'" . cfengine3-mode))
```

- Graphviz for graph generation.

```
(package! graphviz-dot-mode)
```

```
(use-package! graphviz-dot-mode)
```

- I am learning Common LISP, which is well supported through the `common-lisp` Doom module, but I need to configure this in the `~/.slynkrc` file for I/O in the Sly REPL to work fine (source).

```
(setf slynk:*use-dedicated-output-stream* nil)
```

- package-lint for checking MELPA packages.

```
(package! package-lint)
```

Other tools

- Trying out Magit's multi-repository abilities. This stays in sync with the git repo list used by my chain:summary-status Elvish shell function by reading the file every time `magit-list-repositories` is called, using `defadvice!`. I also customize the display to add the `Status` column.

```
(after! magit
  (setq zz/repolist
        "~/.elvish/package-data/elvish-themes/chain-summary-repos.json")
  (defadvice! +zz/load-magit-repositories ()
    :before #'magit-list-repositories
    (setq magit-repository-directories
          (seq-map (lambda (e) (cons e 0)) (json-read-file zz/repolist))
  (setq magit-repolist-columns
        '(("Name" 25 magit-repolist-column-ident nil)
          ("Status" 7 magit-repolist-column-flag nil)
          ("B<U" 3 magit-repolist-column-unpulled-from-upstream
           ((:right-align t)
            (:help-echo "Upstream changes not in branch")))
          ("B>U" 3 magit-repolist-column-unpushed-to-upstream
           ((:right-align t)
            (:help-echo "Local changes not in upstream")))
          ("Path" 99 magit-repolist-column-path nil))))
```

- I prefer to use the GPG graphical PIN entry utility. This is achieved by setting `epg-pinentry-mode` (`epa-pinentry-mode` before Emacs 27) to `nil` instead of the default `'loopback`.

```
(after! epa
  (set (if EMACS27+
           'epg-pinentry-mode
         'epa-pinentry-mode) ; DEPRECATED `epa-pinentry-mode'
       nil))
```

- I find `iedit` absolutely indispensable when coding. In short: when you hit `Ctrl-;`, all occurrences of the symbol under the cursor (or the current selection) are highlighted, and any changes you make on one of them will be automatically applied to all others. It's great for renaming variables in code, but it needs to be used with care, as it has no idea of semantics, it's a plain string replacement, so it can inadvertently modify unintended parts of the code.

```
(package! iedit)
```

```
(use-package! iedit
  :defer
  :config
  (set-face-background 'iedit-occurrence "Magenta")
  :bind
  ("C-;" . iedit-mode))
```

- A useful macro (sometimes) for timing the execution of things. From StackOverflow.

```
(defmacro zz/measure-time (&rest body)
  "Measure the time it takes to evaluate BODY."
  `(let ((time (current-time)))
     ,@body
     (float-time (time-since time))))
```

- I'm still not fully convinced of running a terminal inside Emacs, but `vterm` is much nicer than any of the previous terminal emulators, so I'm giving it a try. I configure it so that it runs my favorite shell. Vterm runs Elvish flawlessly!

```
(setq vterm-shell "/usr/local/bin/elvish")
```

- Add "unfill" commands to parallel the "fill" ones, bind A-q to `unfill-paragraph` and rebind M-q to the `unfill-toggle` command, which fills/unfills paragraphs alternatively.

```
(package! unfill)
```

```
(use-package! unfill
  :defer t
  :bind
  ("M-q" . unfill-toggle)
  ("A-q" . unfill-paragraph))
```

- The annotate package is nice - allows adding annotations to files without modifying the file itself.

```
(package! annotate)
```

Experiments

Some experimental code to list functions which are not native-compiled. Sort of works but its very slow. This does not get tangled to my config.el, I just keep it here for reference.

```
(with-current-buffer (get-buffer-create "*Non-native functions*")
  (mapatoms
   (lambda (s)
     (when (and (functionp s)
                (not (helpful--native-compiled-p s))
                (not (helpful--primitive-p s t)))
       (insert (symbol-name s))
       (insert " --- ")
       (insert (or (cdr (find-function-library s)) "<no file>"))
       (insert "\n"))
     ))
  )
```

Make ox-md export src blocks with backticks and the language name.

```
(defun org-md-example-block (example-block _contents info)
  "Transcode EXAMPLE-BLOCK element into Markdown format.
CONTENTS is nil.  INFO is a plist used as a communication
channel."
  (let ((lang (or (org-element-property :language example-block) "")))
    (format "```%s\n%s```\n"
            lang
            (org-remove-indentation
             (org-export-format-code-default example-block info)))))
```

Tags: config howto literateprogramming literateconfig emacs doom

Related:

How to easily create and use human-readable IDs in Org mode and Doom Emacs
How to insert screenshots in Org documents on macOS
Using and writing completions in Elvish
Beautifying Org Mode in Emacs
My Hammerspoon Configuration, With Commentary
My Emacs Configuration, With Commentary
Bang-Bang (!!, !$) Shell Shortcuts in Elvish
My Elvish Configuration With Commentary

spacemacs and doom!

• Reply • Share ›

Gerardin • 2 months ago

Magnificent post! I got some ideas. Since I don't use DOOM, how shall I replace (map! :desc (or desc file) when using vanilla Emacs? Thanks!

• Reply • Share ›

Diego Zamboni Mod → Gerardin • 2 months ago • edited

Thanks for the nice words!

You can also refer to my old Emacs config, where a lot of the functionality is implemented as well, without any Doom-specific constructs. For example, instead of `map!` you can use `global-set-key`, as shown at https://github.com/zzamboni...

1 • Reply • Share ›

Gerardin → Diego Zamboni • 2 months ago

Thanks, and sorry for double asking :)

• Reply • Share ›

Gerardin → Gerardin • 2 months ago

Actually I am getting "Symbol's value as variable is void: file [2 times]".
I am using the classical method:

```
(defun zz/add-file-keybinding (key
file &optional desc)
(lexical-let ((key key)
(file file)
(desc desc))
(global-set-key (kbd key) (lambda ()
(interactive) (find-file file)))
(which-key-add-key-based-replacements
key (or desc file))))
```

• Reply • Share ›

Diego Zamboni Mod → Gerardin • 2 months ago • edited

That sound like lexical binding is not enabled. `lexical-let` disappeared in recent version of Emacs, in that case you need to change it to `let` but enable lexical binding by adding the following as the first line of the file:

```
;;; -*- lexical-binding: t; -*-
```

• Reply • Share ›

Gerardin → Diego Zamboni

Gerardin ↗ Diego Zamboni
• 2 months ago

Funny thing... my original function had let, not lexical-let! So I tried using lexical-let and it seems to work perfectly. Seems that it is still working on the last Emacs. Thanks again for all your help!

1 ▲ | ▼ • Reply • Share ›

Diego Zamboni Mod ↗ Gerardin
• 2 months ago

Glad it worked!

▲ | ▼ • Reply • Share ›

Gerardin ↗ Diego Zamboni
• 2 months ago

BTW, at the risk of being a pain... in the screenshot you sent, the menu to select the key associated to each orgmode file is vertical

```
a --> orgmode file 1
b --> orgmode file 2
c --> orgmode file 2
d --> orgmode file 2
```

However, in my Emacs this shows in one line in the minibuffer below, so it's really hard to go though them (especially having so many files).

```
a --> orgmode file 1 b --> orgmode
file 2 c --> orgmode file 2 d -->
orgmode file 2
```

Is there a way to control this behaviour? Thanks!

▲ | ▼ • Reply • Share ›

Diego Zamboni Mod ↗ Gerardin
• 2 months ago • edited

I assume there is, because it happened

What's in this Post