

Doom Emacs Configuration

Shaurya Singh

`fatal: not a git repository (or any of the parent
directories): .git`

2021-09-25

13:54 *[110]EDT

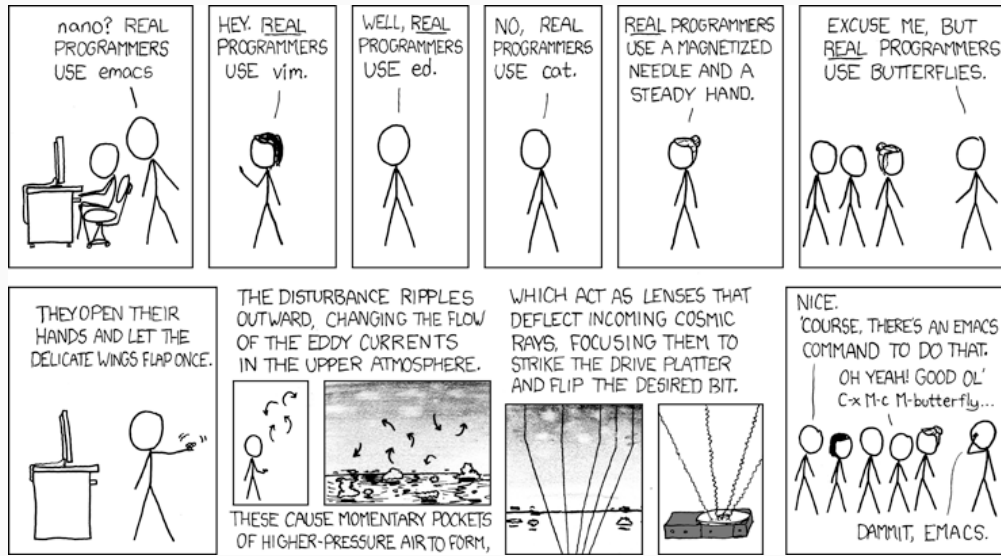
Contents

1	Intro	3
2	Basic Configuration	4
2.1	Personal information	4
2.2	Authinfo	5
2.3	Shell	5
2.3.1	Vterm	5
2.4	Fonts	6
2.5	Themes	8
2.6	Very large files	8
2.7	Company	8
2.8	LSP	12
2.9	Better Defaults	13
2.10	Selectric mode	15
3	Visual configuration	15
3.1	Modeline	15
3.2	Centaur tabs	17
3.3	Vertico	18
3.4	Treemacs	18
3.5	Emojis	19
3.6	Splash screen	19
3.7	Writeroom	24
3.8	Font Display	25
3.8.1	Fontifying inline src blocks	27

3.9	Symbols	29
3.10	Keycast	31
3.11	Transparency	32
3.12	Screenshots	32
3.13	RSS	32
4	Org	37
4.1	Org-Mode	37
4.1.1	HTML	39
4.2	Org-Roam	52
4.3	Org-Agenda	53
4.4	Org-Capture	55
4.4.1	Prettify	55
4.4.2	Templates	59
4.5	ORG Plot	59
4.6	View Exported File	61
4.7	Dictionaries	62
5	Latex	63
5.1	Basic configuration	63
5.2	PDF-Tools	64
5.3	Export	65
5.3.1	Conditional features	65
5.3.2	Tectonic	70
5.3.3	Classes	71
5.3.4	Packages	72
5.3.5	Pretty code blocks	73
5.3.6	ox-chameleon	76
5.3.7	Async	77
5.3.8	(sub super)script characters	77
6	Mu4e	77
7	Browsing	78
7.1	Webkit	78
7.2	IRC	79

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. — Donald Knuth

1 Intro



Customizing an editor can be very rewarding ... until you have to leave it. For years I have been looking for ways to avoid this pain. Then I discovered [vim-anywhere](#). The issue is

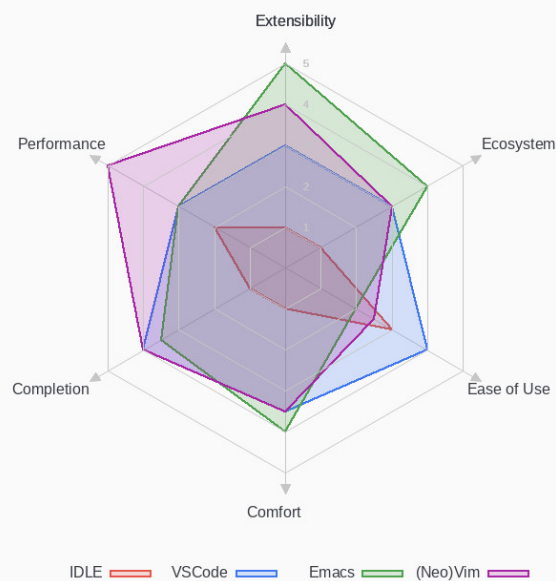
1. I use neovim (and neovide), not vim (and gvim)
2. Firenvim is only for browsers
3. Even if I found a neovim alternative, you can't do everything.

I wanted everything, in one place. Hence why I (mostly) switched to Emacs.

This was enough for me to install Emacs, but there are [many other reasons](#) to keep using it.

That's not to say using Emacs doesn't have its pitfalls. The performance leaves something to be desired, but the benefits far outweigh the drawbacks. It's unrivaled in extensibility.

Editor	Extensibility	Ecosystem	Ease of Use	Comfort	Completion	Performance
IDLE	1	1	3	1	1	2
VSCode	3	3	4	3.5	4	3
Emacs	5	4	2	4	3.5	3
(Neo)Vim	4	3	2.5	3.5	4	5



2 Basic Configuration

Make this file run (slightly) faster with lexical binding

```
;;; config.el -*- lexical-binding: t; -*-
```

I want to run emacs28's new native-compiler with both -O3 and processor specific optimizations, if possible

```
(when 'native-comp-compiler-options
  (setq native-comp-compiler-options '("-O3"
    ↪ "-march=native" "-mtune=native")))
```

2.1 Personal information

Of course we need to tell emacs who I am

```
(setq user-full-name "Shaurya Singh"
      user-mail-address "shaunsingh0207@gmail.com")
```

2.2 Authinfo

I frequently delete my `~/ .emacs.d` for fun, so having authinfo in a separate file sounds like a good idea

```
(setq auth-sources '("~/ .authinfo.gpg")
      auth-source-cache-expiry nil) ; default is 7200 (2h)
```

2.3 Shell

I use the fish shell. If you use zsh/bash, be sure to change this

```
(setq explicit-shell-file-name (executable-find "fish"))
```

2.3.1 Vterm

Vterm is my terminal emulator of choice. We can tell it to use ligatures, and also tell it to compile automatically Vterm clearly wins the terminal war. Also doesn't need much configuration out of the box, although the shell integration does. You can find that in `~/ .config/fish/config.fish`

1. Always compile Fixes a weird bug with native-comp

```
(setq vterm-always-compile-module t)
```

2. Kill buffer If the process exits, kill the `vterm` buffer

```
(setq vterm-kill-buffer-on-exit t)
```

3. Functions Useful functions for the shell-side integration provided by vterm.

```
(after! vterm
  (setf (alist-get "magit-status" vterm-eval-cmds nil nil #'equal)
        '((lambda (path)
              (magit-status path)))))
```

4. Ligatures Use ligatures from within vterm (and eshell), we do this by redefining the variable where *not* to show ligatures

```
(setq +ligatures-in-modes t)
```

2.4 Fonts

I like the apple fonts for programming, so I'll go with Liga SFMono Nerd Font. I prefer a rounder font for plain text, so I'll go with Overpass for that. I have a retina display as well, so let's keep the fonts light.

```
;; fonts
(setq doom-font (font-spec :family "Liga SFMono Nerd Font" :size 14)
      doom-big-font (font-spec :family "Liga SFMono Nerd Font" :size 20)
      doom-variable-pitch-font (font-spec :family "Overpass" :size 16)
      doom-unicode-font (font-spec :family "Liga SFMono Nerd Font")
      doom-serif-font (font-spec :family "Liga SFMono Nerd Font" :weight
      ↪ 'light))
```

For mixed pitch, I would go with something comfier. I like Alegreya, so let's go with that

```
;; mixed pitch modes
(defvar mixed-pitch-modes '(org-mode LaTeX-mode markdown-mode gfm-mode
↪ Info-mode)
  "Modes that `mixed-pitch-mode' should be enabled in, but only after UI
  ↪ initialisation.")
(defun init-mixed-pitch-h ()
  "Hook `mixed-pitch-mode' into each mode in `mixed-pitch-modes'.
  Also immediately enables `mixed-pitch-modes' if currently in one of
  ↪ the modes."
  (when (memq major-mode mixed-pitch-modes)
    (mixed-pitch-mode 1))
  (dolist (hook mixed-pitch-modes)
    (add-hook (intern (concat (symbol-name hook) "-hook"))
      ↪ #'mixed-pitch-mode)))
(add-hook 'doom-init-ui-hook #'init-mixed-pitch-h)
(add-hook! 'org-mode-hook #'org-pretty-mode) ;enter mixed pitch mode in
↪ org mode

;; set mixed pitch font
(after! mixed-pitch
  (defface variable-pitch-serif
    '((t (:family "serif")))
    "A variable-pitch face with serifs."
    :group 'basic-faces)
  (setq mixed-pitch-set-height t)
  (setq variable-pitch-serif-font (font-spec :family "Alegreya" :size
  ↪ 16))
  (set-face-attribute 'variable-pitch-serif nil :font
  ↪ variable-pitch-serif-font))
```

```
(defun mixed-pitch-serif-mode (&optional arg)
  "Change the default face of the current buffer to a serified variable
  ↪ pitch, while keeping some faces fixed pitch."
  (interactive)
  (let ((mixed-pitch-face 'variable-pitch-serif))
    (mixed-pitch-mode (or arg 'toggle)))))
```

Harfbuzz is missing the beautiful ff ffi ffj ffi fti fi fj ft Th ligatures, lets add those back in with the help of composition-function-table

```
(set-char-table-range composition-function-table ?f
  ↪ '(["\\(?:ff?[fijlt]\\)" 0 font-shape-gstring]))
(set-char-table-range composition-function-table ?T '(["\\(?:Th\\)" 0
  ↪ font-shape-gstring]))
```

Just in case the fonts aren't there, lets add a fallback

```
(defvar required-fonts '("Overpass" "Liga SFMono Nerd Font" "Alegreya" ))
(defvar available-fonts
  (delete-dups (or (font-family-list)
    (split-string (shell-command-to-string "fc-list :
    ↪ family" )
      "[,\\n"]))))
(defvar missing-fonts
  (delq nil (mapcar
    (lambda (font)
      (unless (delq nil (mapcar (lambda (f)
        (string-match-p (format "%s$" font) f))
          available-fonts)
        font))
        required-fonts)))
  (if missing-fonts
    (pp-to-string
      `(unless noninteractive
        (add-hook! 'doom-init-ui-hook
          (run-at-time nil nil
            (lambda ()
              (message "%s missing the following fonts: %s"
                (property "Warning!" 'face '(bold
                ↪ warning))
                (mapconcat (lambda (font)
                  (property font 'face
                    ↪ 'font-lock-variable-name-face))
                    ',missing-fonts
                    ", ")))
              (sleep-for 0.5))))))
    ";; No missing fonts detected"))
```

```
<<detect-missing-fonts(>>
```

2.5 Themes

Right now I'm using nord, but I use doom-one-light sometimes

```
;; (setq doom-theme 'doom-one-light)
(setq doom-one-light-padded-modeline t)
(setq doom-theme 'doom-nord)
(setq doom-nord-padded-modeline t)
```

2.6 Very large files

Emacs gets super slow with large files, this helps with that

```
;; (use-package! vlf-setup
;; :defer-incrementally vlf-tune vlf-base vlf-write vlf-search vlf-occur
;↪ vlf-follow vlf-ediff vlf)
```

2.7 Company

I think company is a bit too quick to recommend some stuff

```
(after! company
  (setq company-idle-delay 0.1
        company-minimum-prefix-length 1
        company-selection-wrap-around t
        company-require-match 'never
        company-dabbrev-downcase nil
        company-dabbrev-ignore-case t
        company-dabbrev-other-buffers nil
        company-tooltip-limit 5
        company-tooltip-minimum-width 50))
(set-company-backend!
  '(text-mode
    markdown-mode
    gfm-mode)
  '(:seperate
    company-yasnippet
    company-ispell
    company-files))

;; nested snippets
(setq yas-triggers-in-field t)
```


Lets add some snippets for latex

```
(use-package! aas
  :commands aas-mode)

(use-package! laas
  :hook (LaTeX-mode . laas-mode)
  :config
  (defun laas-tex-fold-maybe ()
    (unless (equal "/" aas-transient-snippet-key)
      (+latex-fold-last-macro-a)))
  (add-hook 'org-mode #'laas-mode)
  (add-hook 'aas-post-snippet-expand-hook #'laas-tex-fold-maybe))
```

And with a little help from henrik, lets use those snippets in org mode

```
(defadvice! fixed-org-yas-expand-maybe-h ()
  "Expand a yasnippet snippet, if trigger exists at point or region is
  active.
  Made for `org-tab-first-hook'."
  :override #'org-yas-expand-maybe-h
  (when (and (featurep! :editor snippets)
    (require 'yasnipet nil t)
    (bound-and-true-p yas-minor-mode))
    (and (let ((major-mode (cond ((org-in-src-block-p t)
      (org-src-get-lang-mode
        ↪ (org-eldoc-get-src-lang)))
      ((org-inside-LaTeX-fragment-p)
        'latex-mode)
      (major-mode))))
      (org-src-tab-acts-natively nil) ; causes breakages
      ;; Smart indentation doesn't work with yasnippet, and
      ↪ painfully slow
      ;; in the few cases where it does.
      (yas-indent-line 'fixed))
      (cond ((and (or (not (bound-and-true-p evil-local-mode))
        (evil-insert-state-p)
        (evil-emacs-state-p))
        (or (and (bound-and-true-p yas--tables)
          (gethash major-mode yas--tables))
          (progn (yas-reload-all) t))
          (yas--templates-for-key-at-point)))
        (yas-expand)
        t)
        ((use-region-p)
          (yas-insert-snippet)
          t)))
      ;; HACK Yasnippet breaks org-superstar-mode because yasnippets
      ↪ is
      ;; overzealous about cleaning up overlays.
```

```
(when (bound-and-true-p org-superstar-mode)
      (org-superstar-restart))))
```

Source code blocks are a pain in org-mode, so lets make a few functions to help with our snippets

```
(defun +yas/org-src-header-p ()
  "Determine whether `point' is within a src-block header or
  ↪ header-args."
  (pcase (org-element-type (org-element-context))
    ('src-block (< (point) ; before code part of the src-block
                  (save-excursion (goto-char (org-element-property
                  ↪ :begin (org-element-context)))
                                (forward-line 1)
                                (point))))
    ('inline-src-block (< (point) ; before code part of the
    ↪ inline-src-block
                      (save-excursion (goto-char
    ↪ (org-element-property :begin
    ↪ (org-element-context)))
                                    (search-forward "]{" )
                                    (point))))
    ('keyword (string-match-p "^header-args" (org-element-property :value
    ↪ (org-element-context))))))
```

Now let's write a function we can reference in yasnippets to produce a nice interactive way to specify header args.

```
(defun +yas/org-prompt-header-arg (arg question values)
  "Prompt the user to set ARG header property to one of VALUES with
  QUESTION.
  The default value is identified and indicated. If either default is
  selected,
  or no selection is made: nil is returned."
  (let* ((src-block-p (not (looking-back "^#\\"+property:[
  ↪ \t]+header-args:.*" (line-beginning-position))))
        (default
         (or
          (cdr (assoc arg
                     (if src-block-p
                         (nth 2 (org-babel-get-src-block-info t))
                         (org-babel-merge-params
                          org-babel-default-header-args
                          (let ((lang-headers
                               (intern (concat
    ↪ "org-babel-default-header-args:"
    ↪ (+yas/org-src-lang))))
                          (when (boundp lang-headers) (eval
    ↪ lang-headers t))))))
          default-value)))
        ""))
  (interactive)))
```

```

(setq values (mapcar
  (lambda (value)
    (if (string-match-p (regexp-quote value) default)
      (setq default-value
        (concat value " "
          (propertize "(default)" 'face
            ↪ 'font-lock-doc-face)))
      value))
  values))
(let ((selection (consult--read question values :default
  ↪ default-value)))
  (unless (or (string-match-p "(default)$" selection)
    (string= "" selection))
    selection)))

```

Finally, we fetch the language information for new source blocks.

Since we're getting this info, we might as well go a step further and also provide the ability to determine the most popular language in the buffer that doesn't have any `header-args` set for it (with `#+properties`).

```

(defun +yas/org-src-lang ()
  "Try to find the current language of the src/header at `point'.
  Return nil otherwise."
  (let ((context (org-element-context)))
    (pcase (org-element-type context)
      ('src-block (org-element-property :language context))
      ('inline-src-block (org-element-property :language context))
      ('keyword (when (string-match "^header-args:\\\([^ ]+\)"
        ↪ (org-element-property :value context))
        (match-string 1 (org-element-property :value
        ↪ context)))))))

(defun +yas/org-last-src-lang ()
  "Return the language of the last src-block, if it exists."
  (save-excursion
    (beginning-of-line)
    (when (re-search-backward "[ \t]*#\+begin_src" nil t)
      (org-element-property :language (org-element-context))))

(defun +yas/org-most-common-no-property-lang ()
  "Find the lang with the most source blocks that has no global
  ↪ header-args, else nil."
  (let (src-langs header-langs)
    (save-excursion
      (goto-char (point-min))
      (while (re-search-forward "[ \t]*#\+begin_src" nil t)
        (push (+yas/org-src-lang) src-langs))
      (goto-char (point-min))
      (while (re-search-forward "[ \t]*#\+property: +header-args" nil
        ↪ t)

```

```

(push (+yas/org-src-lang) header-langs)))

(setq src-langs
  (mapcar #'car
    ;; sort alist by frequency (desc.)
    (sort
      ;; generate alist with form (value . frequency)
      (cl-loop for (n . m) in (seq-group-by #'identity
        ↪ src-langs)
        collect (cons n (length m)))
      (lambda (a b) (> (cdr a) (cdr b))))))

(car (cl-set-difference src-langs header-langs :test #'string=)))

```

Lets also include « to autocomplete, as with () and {}

```

(sp-local-pair
 '(org-mode)
 "<<" ">>"
 :actions '(insert))

```

And lastly lets add some helpful snippets for org-mode, and add a better template

```

(set-file-template! "\\..org$" :trigger "__" :mode 'org-mode)

```

2.8 LSP

I think the LSP is a bit intrusive (especially with inline suggestions), so lets make it behave a bit more

```

(use-package! lsp-ui
  :hook (lsp-mode . lsp-ui-mode)
  :config
  (setq lsp-ui-sideline-enable nil; not anymore useful than flycheck
        lsp-lens-enable t
        lsp-ui-doc-enable t
        lsp-tex-server 'digestif
        lsp-headerline-breadcrumb-enable nil
        lsp-ui-peek-enable t
        lsp-ui-peek-fontify 'on-demand
        lsp-enable-symbol-highlighting nil))

```

2.9 Better Defaults

The defaults for emacs aren't so good nowadays. Lets fix that up a bit

```
(setq undo-limit 80000000 ;I mess up too much
evil-want-fine-undo t ;By default while in
  ↳ insert all changes are one big blob. Be more granular
scroll-margin 2 ;having a little
  ↳ margin is nice
auto-save-default t ;I dont like to lose
  ↳ work
display-line-numbers-type nil ;I dislike line
  ↳ numbers
history-length 25 ;Slight speedup
delete-by-moving-to-trash t ;delete to system
  ↳ trash instead
browse-url-browser-function 'xwidget-webkit-browse-url
truncate-string-ellipsis "...") ;default ellipses suck

(fringe-mode 0) ;;disable fringe
(global-subword-mode 1) ;;navigate through Camel Case words
```

There's issues with emacs flickering on mac (and sometimes wayland). This should fix it

```
(add-to-list 'default-frame-alist '(inhibit-double-buffering . t))
```

Instead of fundamental mode, lisp-interaction-mode seems much more useful

```
(setq doom-scratch-initial-major-mode 'lisp-interaction-mode)
```

Ask where to open splits

```
(setq evil-vsplits-window-right t
      evil-split-window-below t)
```

...and open a buffer for it

```
(defadvice! prompt-for-buffer (&rest _)
  :after '(evil-window-split evil-window-vsplits)
  (consult-buffer))
```

The default bindings of doom are pretty good. I'm not so good with motions though, so lets make life easier with avy

```
(map! :leader
      :desc "hop to word" "w w" #'avy-goto-word-0)
(map! :leader
      :desc "hop to line"
      "l" #'avy-goto-line)
```

I also find ; more intuitive than : for entering command mode

```
(after! evil
  (map! :nmv ";" #'evil-ex))
```

When I'm doing regexes, it's usually with /g anyways, let's make that the default

```
(after! evil
  (setq evil-ex-substitute-global t      ; I like my s/./.. to be global
        ↪ by default
        evil-move-cursor-back nil      ; Don't move the block cursor
        ↪ when toggling insert mode
        evil-kill-on-visual-paste nil)) ; Don't put overwritten text in
        ↪ the kill ring
```

Doom looks much cleaner with the dividers removed. Not sure why it isn't the default honestly

```
(custom-set-faces!
  `(vertical-border :background ,(doom-color 'bg) :foreground
    ↪ ,(doom-color 'bg)))

(when (boundp 'window-divider-mode)
  (setq window-divider-default-places nil
        window-divider-default-bottom-width 0
        window-divider-default-right-width 0)
  (window-divider-mode -1))
```

I don't like seeing the cursorline, especially while writing. Let's disable that

```
(remove-hook 'doom-first-buffer-hook #'global-hl-line-mode)
```

Doom has a weird bug with emacs-plus where the cursor will just turn white on a light theme. Let's fix that.

```
(defadvice! fix-evil-default-cursor-fn ()
  :override #'evil-default-cursor-fn
  (evil-set-cursor-color (face-background 'cursor)))
(defadvice! fix-evil-emacs-cursor-fn ()
  :override #'evil-emacs-cursor-fn
```

```
(evil-set-cursor-color (face-foreground 'warning)))
```

I like using the minimap, even if its slow. Looks cool in my opinion, lets make it a little cooler by removing the scroll highlighting

```
(setq minimap-highlight-line nil)
(custom-set-faces!
  `(minimap-active-region-background :background unspecified))
```

I like a bit of padding on the left hand side, and lets make the line spacing comfier

```
(set-frame-parameter nil 'internal-border-width 24)
(setq-default line-spacing 0.35)
```

2.10 Selectric mode

Typewriter go br

```
(use-package! selectric-mode
  :commands selectric-mode)
```

3 Visual configuration

3.1 Modeline

Tecosaurus PDF improvements:

```
(after! doom-modeline
  (doom-modeline-def-segment buffer-name
    "Display the current buffer's name, without any other information."
    (concat
      (doom-modeline-spc)
      (doom-modeline--buffer-name))))

(doom-modeline-def-segment pdf-icon
  "PDF icon from all-the-icons."
  (concat
    (doom-modeline-spc)
    (doom-modeline-icon 'octicon "file-pdf" nil nil)))
```

```

:face (if (doom-modeline--active)
          'all-the-icons-red
          'mode-line-inactive)
:v-adjust 0.02)))

(defun doom-modeline-update-pdf-pages ()
  "Update PDF pages."
  (setq doom-modeline--pdf-pages
    (let ((current-page-str (number-to-string (eval
      ↪ (pdf-view-current-page))))
          (total-page-str (number-to-string
      ↪ (pdf-cache-number-of-pages))))
      (concat
        (propertize
          (concat (make-string (- (length total-page-str) (length
      ↪ current-page-str)) ? )
          " P" current-page-str)
          'face 'mode-line)
        (propertize (concat "/" total-page-str) 'face
      ↪ 'doom-modeline-buffer-minor-mode))))))

(doom-modeline-def-segment pdf-pages
  "Display PDF pages."
  (if (doom-modeline--active) doom-modeline--pdf-pages
      (propertize doom-modeline--pdf-pages 'face 'mode-line-inactive)))

(doom-modeline-def-modeline 'pdf
  '(bar window-number pdf-pages pdf-icon buffer-name)
  '(misc-info matches major-mode process vcs)))

```

Doom modeline already looks good, but it can be better. Lets add some icons, the battery status, and make sure we don't lose track of time

```

(after! doom-modeline
  (display-time-mode 1) ;Enable time in the
  ↪ mode-line
  (display-battery-mode 1) ;display the battery
  (setq doom-modeline-major-mode-icon t ;Show major mode
  ↪ name
    doom-modeline-enable-word-count t ;Show word count
    doom-modeline-modal-icon t ;Show vim mode icon
    inhibit-compacting-font-caches t) ;Don't compact font
  ↪ caches in gc

```

The encoding is always UTF-8, so its a bit redundant. Lets take that out

```

(defun doom-modeline-conditional-buffer-encoding ()
  "We expect the encoding to be LF UTF-8, so only show the modeline when
  ↪ this is not the case"
  (setq-local doom-modeline-buffer-encoding

```



```

(unless (and (memq (plist-get (coding-system-plist
  ↪ buffer-file-coding-system) :category)
              '(coding-category-undecided
                ↪ coding-category-utf-8))
          (not (memq (coding-system-eol-type
                    ↪ buffer-file-coding-system) '(1 2))))
  t)))
(add-hook 'after-change-major-mode-hook
  ↪ #'doom-modeline-conditional-buffer-encoding) ;; remove encoding

```

3.2 Centaur tabs

There isn't much of a point having tabs when you only have one buffer open. This checks the number of tabs, and hides them if there's only one left

```

(defun centaur-tabs-get-total-tab-length ()
  (length (centaur-tabs-tabs (centaur-tabs-current-tabset))))

(defun centaur-tabs-hide-on-window-change ()
  (run-at-time nil nil
    (lambda ()
      (centaur-tabs-hide-check
        ↪ (centaur-tabs-get-total-tab-length)))))

(defun centaur-tabs-hide-check (len)
  (shut-up
    (cond
      ((and (= len 1) (not (centaur-tabs-local-mode))) (call-interactively
        ↪ #'centaur-tabs-local-mode))
      ((and (≥ len 2) (centaur-tabs-local-mode)) (call-interactively
        ↪ #'centaur-tabs-local-mode))))))

```

I also like to have icons with my tabs.

```

(after! centaur-tabs
  (centaur-tabs-mode -1)
  (setq centaur-tabs-height 20
    centaur-tabs-set-icons t
    centaur-tabs-gray-out-icons 'buffer)
  (add-hook 'window-configuration-change-hook
    ↪ 'centaur-tabs-hide-on-window-change)
  (centaur-tabs-change-fonts "Liga SFMono Nerd Font" 105))

```

3.3 Vertico

For marginalia (vertico), lets use relative time, along with some other things

```
(after! marginalia
  (setq marginalia-censor-variables nil)

  (defadvice! +marginalia--anotate-local-file-colorful (cand)
    "Just a more colourful version of `marginalia--anotate-local-file'."
    :override #'marginalia--annotate-local-file
    (when-let (attrs (file-attributes (substitute-in-file-name
                                       (marginalia--full-candidate cand))
                                       'integer))

      (marginalia--fields
        ((marginalia--file-owner attrs)
         :width 12 :face 'marginalia-file-owner)
        (marginalia--file-modes attrs))
        ((+marginalia-file-size-colorful (file-attribute-size attrs))
         :width 7)
        ((+marginalia--time-colorful (file-attribute-modification-time
                                     ↪ attrs))
         :width 12))))))

(defun +marginalia--time-colorful (time)
  (let* ((seconds (float-time (time-subtract (current-time) time)))
        (color (doom-blend
                  (face-attribute 'marginalia-date :foreground nil t)
                  (face-attribute 'marginalia-documentation :foreground
                                  ↪ nil t)
                  (/ 1.0 (log (+ 3 (/ (+ 1 seconds) 345600.0)))))))
    ;; 1 - log(3 + 1/(days + 1)) % grey
    (propertize (marginalia--time time) 'face (list :foreground
                                                      ↪ color))))))

(defun +marginalia-file-size-colorful (size)
  (let* ((size-index (/ (log10 (+ 1 size)) 7.0))
        (color (if (< size-index 10000000) ; 10m
                    (doom-blend 'orange 'green size-index)
                    (doom-blend 'red 'orange (- size-index 1)))))
    (propertize (file-size-human-readable size) 'face (list :foreground
                                                             ↪ color))))))
```

3.4 Treemacs

Lets theme treemacs while we're at it

```
(setq treemacs-width 25)
(setq doom-themes-treemacs-theme "doom-colors")
```

3.5 Emojis

Disable some annoying emojis

[illegible]

3.6 Splash screen

Emacs can render an image as the splash screen, and the emacs logo looks pretty cool. Now we just make it theme-appropriate, and resize with the frame.

```
(defvar fancy-splash-image-template
  (expand-file-name "misc/splash-images/emacs-e-template.svg"
    ↪ doom-private-dir)
  "Default template svg used for the splash image, with substitutions
  ↪ from ")

(defvar fancy-splash-sizes
  `((:height 300 :min-height 50 :padding (0 . 2))
    (:height 250 :min-height 42 :padding (2 . 4))
    (:height 200 :min-height 35 :padding (3 . 3))
    (:height 150 :min-height 28 :padding (3 . 3))
    (:height 100 :min-height 20 :padding (2 . 2))
    (:height 75 :min-height 15 :padding (2 . 1))
    (:height 50 :min-height 10 :padding (1 . 0))
    (:height 1 :min-height 0 :padding (0 . 0))))
```

```

"list of plists with the following properties
:height the height of the image
:min-height minimum `frame-height' for image
:padding `+doom-dashboard-banner-padding' (top . bottom) to apply
:template non-default template file
:file file to use instead of template")

(defvar fancy-splash-template-colours
  '("$colour1" . keywords) (" $colour2" . type) (" $colour3" . base5)
  ↪ (" $colour4" . base8))
"list of colour-replacement alists of the form (\"$placeholder\" .
  ↪ 'theme-colour) which applied the template")

(unless (file-exists-p (expand-file-name "theme-splashes"
  ↪ doom-cache-dir))
  (make-directory (expand-file-name "theme-splashes" doom-cache-dir) t))

(defun fancy-splash-filename (theme-name height)
  (expand-file-name (concat (file-name-as-directory "theme-splashes")
                             theme-name
                             "-" (number-to-string height) ".svg")
                    doom-cache-dir))

(defun fancy-splash-clear-cache ()
  "Delete all cached fancy splash images"
  (interactive)
  (delete-directory (expand-file-name "theme-splashes" doom-cache-dir) t)
  (message "Cache cleared!"))

(defun fancy-splash-generate-image (template height)
  "Read TEMPLATE and create an image if HEIGHT with colour substitutions
  as
  described by `fancy-splash-template-colours' for the current
  ↪ theme"
  (with-temp-buffer
    (insert-file-contents template)
    (re-search-forward "$height" nil t)
    (replace-match (number-to-string height) nil nil)
    (dolist (substitution fancy-splash-template-colours)
      (goto-char (point-min))
      (while (re-search-forward (car substitution) nil t)
        (replace-match (doom-color (cdr substitution)) nil nil)))
    (write-region nil nil
                  (fancy-splash-filename (symbol-name doom-theme) height)
                  ↪ nil nil)))

(defun fancy-splash-generate-images ()
  "Perform `fancy-splash-generate-image' in bulk"
  (dolist (size fancy-splash-sizes)
    (unless (plist-get size :file)
      (fancy-splash-generate-image (or (plist-get size :template)
                                       fancy-splash-image-template)
                                   (plist-get size :height))))))

```

```

(defun ensure-theme-splash-images-exist (&optional height)
  (unless (file-exists-p (fancy-splash-filename
                          (symbol-name doom-theme)
                          (or height
                              (plist-get (car fancy-splash-sizes)
                                          ↪ :height))))
    (fancy-splash-generate-images)))

(defun get-appropriate-splash ()
  (let ((height (frame-height)))
    (cl-some (lambda (size) (when (≥ height (plist-get size
                                                         ↪ :min-height)) size))
              fancy-splash-sizes)))

(setq fancy-splash-last-size nil)
(setq fancy-splash-last-theme nil)
(defun set-appropriate-splash (&rest _)
  (let ((appropriate-image (get-appropriate-splash)))
    (unless (and (equal appropriate-image fancy-splash-last-size)
                  (equal doom-theme fancy-splash-last-theme)))
      (unless (plist-get appropriate-image :file)
        (ensure-theme-splash-images-exist (plist-get appropriate-image
                                                         ↪ :height)))
      (setq fancy-splash-image
            (or (plist-get appropriate-image :file)
                (fancy-splash-filename (symbol-name doom-theme) (plist-get
                                                                    ↪ appropriate-image :height))))
      (setq +doom-dashboard-banner-padding (plist-get appropriate-image
                                                         ↪ :padding))
      (setq fancy-splash-last-size appropriate-image)
      (setq fancy-splash-last-theme doom-theme)
      (+doom-dashboard-reload)))

(add-hook 'window-size-change-functions #'set-appropriate-splash)
(add-hook 'doom-load-theme-hook #'set-appropriate-splash)

```

Lets add a little phrase in there as well

```

(defvar splash-phrase-source-folder
  (expand-file-name "misc/splash-phrases" doom-private-dir)
  "A folder of text files with a fun phrase on each line.")

(defvar splash-phrase-sources
  (let* ((files (directory-files splash-phrase-source-folder nil
                                  ↪ "\\*.txt\\'"))
         (sets (delete-dups (mapcar
                              (lambda (file)
                                (replace-regexp-in-string
                                   ↪ "\\(?:-[0-9]+-\\w+\\)?\\.txt" ""
                                   ↪ file))
                              files)))
    sets)

```

```

                                files))))
  (mapcar (lambda (sset)
            (cons sset
                  (delq nil (mapcar
                             (lambda (file)
                               (when (string-match-p (regexp-quote
                                                         ↪ sset) file)
                                 file))
                             files))))
          sets))
  "A list of cons giving the phrase set name, and a list of files which
  ↪ contain phrase components.")

(defun splash-phrase-set
  (nth (random (length splash-phrase-sources)) (mapcar #'car
                                                         ↪ splash-phrase-sources))
  "The default phrase set. See `splash-phrase-sources'.")

(defun splase-phrase-set-random-set ()
  "Set a new random splash phrase set."
  (interactive)
  (setq splash-phrase-set
        (nth (random (1- (length splash-phrase-sources)))
              (cl-set-difference (mapcar #'car splash-phrase-sources)
                                ↪ (list splash-phrase-set))))
  (+doom-dashboard-reload t))

(defun splase-phrase--cache nil)

(defun splash-phrase-get-from-file (file)
  "Fetch a random line from FILE."
  (let ((lines (or (cdr (assoc file splase-phrase--cache))
                   (cdar (push (cons file
                                     (with-temp-buffer
                                       (insert-file-contents
                                         ↪ (expand-file-name file
                                         ↪ splash-phrase-source-folder))
                                       (split-string (string-trim
                                                         ↪ (buffer-string)) "\n"))
                                     splase-phrase--cache))))))
    (nth (random (length lines)) lines)))

(defun splash-phrase (&optional set)
  "Construct a splash phrase from SET. See `splash-phrase-sources'."
  (mapconcat
   #'splash-phrase-get-from-file
   (cdr (assoc (or set splash-phrase-set) splash-phrase-sources))
   " "))

(defun doom-dashboard-phrase ()
  "Get a splash phrase, flow it over multiple lines as needed, and make
  ↪ fontify it."
  (mapconcat

```

```

(lambda (line)
  (+doom-dashboard--center
   +doom-dashboard--width
   (with-temp-buffer
    (insert-text-button
     line
     'action
     (lambda (_) (+doom-dashboard-reload t))
     'face 'doom-dashboard-menu-title
     'mouse-face 'doom-dashboard-menu-title
     'help-echo "Random phrase"
     'follow-link t)
    (buffer-string))))
(split-string
 (with-temp-buffer
  (insert (splash-phrase))
  (setq fill-column (min 70 (/ (* 2 (window-width)) 3)))
  (fill-region (point-min) (point-max))
  (buffer-string))
 "\n"
 "\n"))

(defadvice! doom-dashboard-widget-loaded-with-phrase ()
:override #'doom-dashboard-widget-loaded
(setq line-spacing 0.2)
(insert
 "\n\n"
 (propertize
  (+doom-dashboard--center
   +doom-dashboard--width
   (doom-display-benchmark-h 'return))
  'face 'doom-dashboard-loaded)
 "\n"
 (doom-dashboard-phrase)
 "\n"))

```

Lastly, the doom dashboard “useful commands” are no longer useful to me. So, we’ll disable them and then for a particularly *clean* look disable the modeline, then also hide the cursor.

```

(remove-hook '+doom-dashboard-functions
 ↪ #'doom-dashboard-widget-shortmenu)
(add-hook! '+doom-dashboard-mode-hook (hide-mode-line-mode 1)
 ↪ (hl-line-mode -1))
(setq-hook! '+doom-dashboard-mode-hook evil-normal-state-cursor (list
 ↪ nil))

```

3.7 Writeroom

For starters, I think Doom is a bit over-zealous when zooming in

```
(setq +zen-text-scale 0.8)
```

Then, when using Org it would be nice to make a number of other aesthetic tweaks. Namely:

- Use a serif-ed variable-pitch font
- Hiding headline leading stars
- Using fleurons as headline bullets
- Hiding line numbers
- Removing outline indentation
- Centering the text
- Turning on `org-pretty-table-mode`
- Disabling `doom-modeline`

```
(defvar +zen-serif-p t
  "Whether to use a serifed font with `mixed-pitch-mode'.")
(after! writeroom-mode
  (defvar-local +zen--original-org-indent-mode-p nil)
  (defvar-local +zen--original-mixed-pitch-mode-p nil)
  (defun +zen-enable-mixed-pitch-mode-h ()
    "Enable `mixed-pitch-mode' when in `+zen-mixed-pitch-modes'."
    (when (apply #'derived-mode-p +zen-mixed-pitch-modes)
      (if writeroom-mode
        (progn
          (setq +zen--original-mixed-pitch-mode-p mixed-pitch-mode)
          (funcall (if +zen-serif-p #'mixed-pitch-serif-mode
                       ↪ #'mixed-pitch-mode) 1))
        (funcall #'mixed-pitch-mode (if +zen--original-mixed-pitch-mode-p
                                         ↪ 1 -1)))))
  (pushnew! writeroom--local-variables
    'display-line-numbers
    'visual-fill-column-width
    'org-adapt-indentation
    'org-superstar-headline-bullets-list
    'org-superstar-remove-leading-stars)
  (add-hook 'writeroom-mode-enable-hook
```



```
(defun +zen-prose-org-h ()
  "Reformat the current Org buffer appearance for prose."
  (when (eq major-mode 'org-mode)
    (setq display-line-numbers nil
          visual-fill-column-width 60
          org-adapt-indentation nil)
    (when (featurep 'org-superstar)
      (setq-local org-superstar-headline-bullets-list '(" "
        ↪ "○" " " " " " " " " " " " " " " "▶")
        org-superstar-remove-leading-stars t)
      (org-superstar-restart))
      (setq
        +zen--original-org-indent-mode-p org-indent-mode)
      (org-indent-mode -1))))

(add-hook! 'writeroom-mode-hook
  (if writeroom-mode
    (add-hook 'post-command-hook #'recenter nil t)
    (remove-hook 'post-command-hook #'recenter t)))

(add-hook 'writeroom-mode-enable-hook #'doom-disable-line-numbers-h)
(add-hook 'writeroom-mode-disable-hook #'doom-enable-line-numbers-h)
(add-hook 'writeroom-mode-disable-hook

  (defun +zen-nonprose-org-h ()
    "Reverse the effect of `+zen-prose-org'."
    (when (eq major-mode 'org-mode)
      (when (featurep 'org-superstar)
        (org-superstar-restart))
      (when +zen--original-org-indent-mode-p (org-indent-mode
        ↪ 1))))))
```

3.8 Font Display

Mixed pitch is great. As is `+org-pretty-mode`, let's use them.

```
(add-hook 'org-mode-hook #'org-pretty-mode)
```

However, the subscripts (and superscripts) are confusing with latex fragments, so lets turn those off

```
(setq org-pretty-entities-include-sub-superscripts nil)
```

Let's make headings a bit bigger

```
(custom-set-faces!
 '(org-document-title :height 1.2)
 '(outline-1 :weight extra-bold :height 1.25)
 '(outline-2 :weight bold :height 1.15)
 '(outline-3 :weight bold :height 1.12))
```

```
'(outline-4 :weight semi-bold :height 1.09)
'(outline-5 :weight semi-bold :height 1.06)
'(outline-6 :weight semi-bold :height 1.03)
'(outline-8 :weight semi-bold)
'(outline-9 :weight semi-bold))
```

It seems reasonable to have deadlines in the error face when they're passed.

```
(setq org-agenda-deadline-faces
  '(1.0 . error)
    (1.0 . org-warning)
    (0.5 . org-upcoming-deadline)
    (0.0 . org-upcoming-distant-deadline)))
```

We can then have quote blocks stand out a bit more by making them *italic*.

```
(setq org-fontify-quote-and-verse-blocks t)
```

```
(use-package! org-appear
  :hook (org-mode . org-appear-mode)
  :config
  (setq org-appear-autoemphasis t
        org-appear-autosubmarkers t
        org-appear-autolinks nil)
  (run-at-time nil nil #'org-appear--set-elements))
```

Org files can be rather nice to look at, particularly with some of the customisations here. This comes at a cost however, expensive font-lock. Feeling like you're typing through molasses in large files is no fun, but there is a way I can defer font-locking when typing to make the experience more responsive.

```
(defun locally-defer-font-lock ()
  "Set jit-lock defer and stealth, when buffer is over a certain size."
  (when (> (buffer-size) 50000)
    (setq-local jit-lock-defer-time 0.05
                jit-lock-stealth-time 1)))

(add-hook 'org-mode-hook #'locally-defer-font-lock)

(custom-set-faces!
  `(org-block-end-line :background ,(doom-color 'base2))
  `(org-block-begin-line :background ,(doom-color 'base2)))
```

3.8.1 Fontifying inline src blocks

Org does lovely things with `#+begin_src` blocks, like using font-lock for language's major-mode behind the scenes and pulling out the lovely colourful results. By contrast, inline `src_` blocks are somewhat neglected.

I am not the first person to feel this way, thankfully others have [taken to stackexchange](#) to voice their desire for inline src fontification. I was going to steal their work, but unfortunately they didn't perform *true* source code fontification, but simply applied the `org-code` face to the content.

We can do better than that, and we shall! Using `org-src-font-lock-fontify-block` we can apply language-appropriate syntax highlighting. Then, continuing on to `{{{results(...)}}}`, it can have the `org-block` face applied to match, and then the value-surrounding constructs hidden by mimicking the behaviour of `prettify-symbols-mode`.

```
(defvar org-prettify-inline-results t
  "Whether to use (ab)use prettify-symbols-mode on {{{results( ... )}}}.
   Either t or a cons cell of strings which are used as substitutions
   for the start and end of inline results, respectively.")

(defvar org-fontify-inline-src-blocks-max-length 200
  "Maximum content length of an inline src block that will be
  ↪ fontified.")

(defun org-fontify-inline-src-blocks (limit)
  "Try to apply `org-fontify-inline-src-blocks-1'."
  (condition-case nil
    (org-fontify-inline-src-blocks-1 limit)
    (error (message "Org mode fontification error in %S at %d"
                    (current-buffer)
                    (line-number-at-pos))))))

(defun org-fontify-inline-src-blocks-1 (limit)
  "Fontify inline src_LANG blocks, from `point' up to LIMIT."
  (let ((case-fold-search t)
        (initial-point (point)))
    (while (re-search-forward "\\_<src_\\([^\t\n[{}+\\)]{0,1}" limit t) ;
      ↪ stolen from `org-element-inline-src-block-parser'
      (let ((beg (match-beginning 0))
            pt
            (lang-beg (match-beginning 1))
            (lang-end (match-end 1)))
        (remove-text-properties beg lang-end '(face nil))
        (font-lock-append-text-property lang-beg lang-end 'face
          ↪ 'org-meta-line)
        (font-lock-append-text-property beg lang-beg 'face 'shadow)
        (font-lock-append-text-property beg lang-end 'face 'org-block)
        (setq pt (goto-char lang-end)))))
```

```

;; `org-element--parse-paired-brackets' doesn't take a limit, so
⇒ to
;; prevent it searching the entire rest of the buffer we
⇒ temporarily
;; narrow the active region.
(save-restriction
  (narrow-to-region beg (min (point-max) limit (+ lang-end
    ⇒ org-fontify-inline-src-blocks-max-length)))
  (when (ignore-errors (org-element--parse-paired-brackets ?\[))
    (remove-text-properties pt (point) '(face nil))
    (font-lock-append-text-property pt (point) 'face 'org-block)
    (setq pt (point)))
  (when (ignore-errors (org-element--parse-paired-brackets ?\{))
    (remove-text-properties pt (point) '(face nil))
    (font-lock-append-text-property pt (1+ pt) 'face '(org-block
    ⇒ shadow))
    (unless (= (1+ pt) (1- (point)))
      (if org-src-fontify-natively
        (org-src-font-lock-fontify-block
          ⇒ (buffer-substring-no-properties lang-beg lang-end)
          ⇒ (1+ pt) (1- (point)))
        (font-lock-append-text-property (1+ pt) (1- (point))
          ⇒ 'face 'org-block)))
      (font-lock-append-text-property (1- (point)) (point) 'face
        ⇒ '(org-block shadow))
      (setq pt (point))))
  (when (and org-prettify-inline-results (re-search-forward "\\="
    ⇒ {{{results(" limit t))
      (font-lock-append-text-property pt (1+ pt) 'face 'org-block)
      (goto-char pt))))
    (when org-prettify-inline-results
      (goto-char initial-point)
      (org-fontify-inline-src-results limit))))

(defun org-fontify-inline-src-results (limit)
  (while (re-search-forward "{{{results(\\(.+?\\))}}}" limit t)
    (remove-list-of-text-properties (match-beginning 0) (point)
      '(composition
        prettify-symbols-start
        prettify-symbols-end))
    (font-lock-append-text-property (match-beginning 0) (match-end 0)
      ⇒ 'face 'org-block)
    (let ((start (match-beginning 0)) (end (match-beginning 1)))
      (with-silent-modifications
        (compose-region start end (if (eq org-prettify-inline-results t)
          ⇒ " " (car org-prettify-inline-results)))
        (add-text-properties start end `(prettify-symbols-start ,start
          ⇒ prettify-symbols-end ,end))))
    (let ((start (match-end 1)) (end (point)))
      (with-silent-modifications
        (compose-region start end (if (eq org-prettify-inline-results t)
          ⇒ " " (cdr org-prettify-inline-results)))

```



```

:arrow_left      "←"
:property        " "
:options         "⌵"
:startup         "🔌"
:html_head       " "
:html            " "
:latex_class     " "
:latex_header    " "
:beamer_header   " "
:latex          " "
:attr_latex      " "
:attr_html       " "
:attr_org        " "
:begin_quote     "“"
:end_quote       "”"
:caption         " "
:header          " >"
:begin_export    " "
:end_export      " "
:properties      " "
:end             " "
:priority_a      ,(propertyize " " 'face 'all-the-icons-red)
:priority_b      ,(propertyize " " 'face 'all-the-icons-orange)
:priority_c      ,(propertyize "■" 'face 'all-the-icons-yellow)
:priority_d      ,(propertyize " " 'face 'all-the-icons-green)
:priority_e      ,(propertyize " " 'face 'all-the-icons-blue))
(set-ligatures! 'org-mode
:merge t
:checkbox          "[ ]"
:pending         "[-]"
:checkedbox      "[X]"
:list_property   " :: "
:em_dash         "___"
:ellipsis        "... "
:arrow_right     "->"
:arrow_left      "<-"
:title           "##+title:"
:subtitle        "##+subtitle:"
:author          "##+author:"
:date            "##+date:"
:property        "##+property:"
:options         "##+options:"
:startup         "##+startup:"
:macro           "##+macro:"
:html_head       "##+html_head:"
:html            "##+html:"
:latex_class     "##+latex_class:"
:latex_header    "##+latex_header:"
:beamer_header   "##+beamer_header:"
:latex           "##+latex:"
:attr_latex      "##+attr_latex:"
:attr_html       "##+attr_html:"
:attr_org        "##+attr_org:")

```

```

:begin_quote    "#+begin_quote"
:end_quote      "#+end_quote"
:caption        "#+caption:"
:header         "#+header:"
:begin_export   "#+begin_export"
:end_export     "#+end_export"
:results        "#+RESULTS:"
:property       ":PROPERTIES:"
:end            ":END:"
:priority_a     "[#A]"
:priority_b     "[#B]"
:priority_c     "[#C]"
:priority_d     "[#D]"
:priority_e     "[#E]"
(plist-put +ligatures-extra-symbols :name " ")

```

Lets also add a function that makes it easy to convert from upper to lowercase, since the ligatures don't work with Uppercase (I can make them work, but lowercase looks better anyways)

```

(defun org-syntax-convert-keyword-case-to-lower ()
  "Convert all #+KEYWORDS to #+keywords."
  (interactive)
  (save-excursion
    (goto-char (point-min))
    (let ((count 0)
          (case-fold-search nil))
      (while (re-search-forward "[ \t]*#[A-Z_]+" nil t)
        (unless (s-matches-p "RESULTS" (match-string 0))
          (replace-match (downcase (match-string 0)) t)
          (setq count (1+ count))))
      (message "Replaced %d occurances" count))))

```

3.10 Keycast

Its nice for demonstrations

```

(use-package! keycast
  :commands keycast-mode
  :config
  (define-minor-mode keycast-mode
    "Show current command and its key binding in the mode line."
    :global t
    (if keycast-mode
        (progn
          (add-hook 'pre-command-hook 'keycast--update t)
          (add-to-list 'global-mode-string '(" " mode-line-keycast " ")))
        (remove-hook 'pre-command-hook 'keycast--update)))

```

```
(setq global-mode-string (remove '(" " mode-line-keycast " ")
  ↳ global-mode-string)))
(custom-set-faces!
  '(keycast-command :inherit doom-modeline-debug
                    :height 1.0)
  '(keycast-key :inherit custom-modified
               :height 1.0
               :weight bold)))
```

3.11 Transparency

I'm not too big of a fan of transparency, but some people like it. You can use this little function to toggle it now. On **C-c t** inactive windows will dim (85% transparency) and focused windows remain opaque

```
(defun toggle-transparency ()
  (interactive)
  (let ((alpha (frame-parameter nil 'alpha)))
    (set-frame-parameter
      nil 'alpha
      (if (eql (cond ((numberp alpha) alpha)
                    ((numberp (cdr alpha)) (cdr alpha))
                    ;; Also handle undocumented (<active> <inactive>)
                    ↳ form.
                    ((numberp (cadr alpha)) (cadr alpha)))
          100)
          '(100 . 85) '(100 . 100))))
  (global-set-key (kbd "C-c t") 'toggle-transparency))
```

3.12 Screenshots

Make it easy to take nice screenshots. I need to figure out how to make clipboard work though.

```
(use-package! screenshot
  :defer t
  :config (setq screenshot-upload-fn "upload $s 2>/dev/null"))
```

3.13 RSS

RSS is a nice simple way of getting my news. Lets set that up


```

(map! :map elfeed-search-mode-map
      :after elfeed-search
      [remap kill-this-buffer] "q"
      [remap kill-buffer] "q"
      :n doom-leader-key nil
      :n "q" #' +rss/quit
      :n "e" #'elfeed-update
      :n "r" #'elfeed-search-untag-all-unread
      :n "u" #'elfeed-search-tag-all-unread
      :n "s" #'elfeed-search-live-filter
      :n "RET" #'elfeed-search-show-entry
      :n "p" #'elfeed-show-pdf
      :n "+" #'elfeed-search-tag-all
      :n "-" #'elfeed-search-untag-all
      :n "S" #'elfeed-search-set-filter
      :n "b" #'elfeed-search-browse-url
      :n "y" #'elfeed-search-yank)

(map! :map elfeed-show-mode-map
      :after elfeed-show
      [remap kill-this-buffer] "q"
      [remap kill-buffer] "q"
      :n doom-leader-key nil
      :nm "q" #' +rss/delete-pane
      :nm "o" #'ace-link-elfeed
      :nm "RET" #'org-ref-elfeed-add
      :nm "n" #'elfeed-show-next
      :nm "N" #'elfeed-show-prev
      :nm "p" #'elfeed-show-pdf
      :nm "+" #'elfeed-show-tag
      :nm "-" #'elfeed-show-untag
      :nm "s" #'elfeed-show-new-live-search
      :nm "y" #'elfeed-show-yank)

(after! elfeed-search
  (set-evil-initial-state! 'elfeed-search-mode 'normal))
(after! elfeed-show-mode
  (set-evil-initial-state! 'elfeed-show-mode 'normal))

(after! evil-snipe
  (push 'elfeed-show-mode evil-snipe-disabled-modes)
  (push 'elfeed-search-mode evil-snipe-disabled-modes))

(after! elfeed
  (elfeed-org)
  (use-package! elfeed-link)

  (setq elfeed-search-filter "@1-week-ago +unread"
        elfeed-search-print-entry-function
          ⇒ ' +rss/elfeed-search-print-entry
        elfeed-search-title-min-width 80
        elfeed-show-entry-switch #'pop-to-buffer
        elfeed-show-entry-delete #' +rss/delete-pane)

```

```

elfeed-show-refresh-function
↳ #' +rss/elfeed-show-refresh--better-style
shr-max-image-proportion 0.6)

(add-hook! 'elfeed-show-mode-hook (hide-mode-line-mode 1))
(add-hook! 'elfeed-search-update-hook #'hide-mode-line-mode)

(defface elfeed-show-title-face '((t (:weight ultrabold :slant italic
↳ :height 1.5)))
  "title face in elfeed show buffer"
  :group 'elfeed)
(defface elfeed-show-author-face `((t (:weight light)))
  "title face in elfeed show buffer"
  :group 'elfeed)
(set-face-attribute 'elfeed-search-title-face nil
  :foreground 'nil
  :weight 'light)

(defadvice! +rss-elfeed-wrap-h-nicer ()
  "Enhances an elfeed entry's readability by wrapping it to a width of
  `fill-column' and centering it with `visual-fill-column-mode'."
  :override #' +rss-elfeed-wrap-h
  (setq-local truncate-lines nil
    shr-width 120
    visual-fill-column-center-text t
    default-text-properties '(line-height 1.1))
  (let ((inhibit-read-only t)
        (inhibit-modification-hooks t))
    (visual-fill-column-mode)
    ;; (setq-local shr-current-font '(:family "Merriweather" :height
    ↳ 1.2))
    (set-buffer-modified-p nil)))

(defun +rss/elfeed-search-print-entry (entry)
  "Print ENTRY to the buffer."
  (let* ((elfeed-goodies/tag-column-width 40)
        (elfeed-goodies/feed-source-column-width 30)
        (title (or (elfeed-meta entry :title) (elfeed-entry-title
↳ entry) ""))
        (title-faces (elfeed-search--faces (elfeed-entry-tags entry)))
        (feed (elfeed-entry-feed entry))
        (feed-title
          (when feed
            (or (elfeed-meta feed :title) (elfeed-feed-title feed))))
        (tags (mapcar #'symbol-name (elfeed-entry-tags entry)))
        (tags-str (concat (mapconcat 'identity tags ",")))
        (title-width (- (window-width)
↳ elfeed-goodies/feed-source-column-width
        elfeed-goodies/tag-column-width 4))
        (tag-column (elfeed-format-column
        tags-str (length tags-str)

```

```

                                ↪ elfeed-goodies/tag-column-width
                                ↪ elfeed-goodies/tag-column-width)
:left))
(feed-column (elfeed-format-column
  feed-title (elfeed-clamp
    ↪ elfeed-goodies/feed-source-column-width
                                ↪ elfeed-goodies/feed-source-column-width
                                ↪ elfeed-goodies/feed-source-column-width
                                ↪ elfeed-goodies/feed-source-column-width
:left)))

(insert (propertize feed-column 'face 'elfeed-search-feed-face) "
  ↪ ")
(insert (propertize tag-column 'face 'elfeed-search-tag-face) " ")
(insert (propertize title 'face title-faces 'kbd-help title))
(setq-local line-spacing 0.2)))

(defun +rss/elfeed-show-refresh--better-style ()
  "Update the buffer to match the selected entry, using a mail-style."
  (interactive)
  (let* ((inhibit-read-only t)
    (title (elfeed-entry-title elfeed-show-entry))
    (date (seconds-to-time (elfeed-entry-date elfeed-show-entry)))
    (author (elfeed-meta elfeed-show-entry :author))
    (link (elfeed-entry-link elfeed-show-entry))
    (tags (elfeed-entry-tags elfeed-show-entry))
    (tagsstr (mapconcat #'symbol-name tags ", "))
    (nicedate (format-time-string "%a, %e %b %Y %T %Z" date))
    (content (elfeed-deref (elfeed-entry-content
      ↪ elfeed-show-entry)))
    (type (elfeed-entry-content-type elfeed-show-entry))
    (feed (elfeed-entry-feed elfeed-show-entry))
    (feed-title (elfeed-feed-title feed))
    (base (and feed (elfeed-compute-base (elfeed-feed-url
      ↪ feed)))))
    (erase-buffer)
    (insert "\n")
    (insert (format "%s\n\n" (propertize title 'face
      ↪ 'elfeed-show-title-face)))
    (insert (format "%s\t" (propertize feed-title 'face
      ↪ 'elfeed-search-feed-face)))
    (when (and author elfeed-show-entry-author)
      (insert (format "%s\n" (propertize author 'face
        ↪ 'elfeed-show-author-face))))
    (insert (format "%s\n\n" (propertize nicedate 'face
      ↪ 'elfeed-log-date-face)))
    (when tags
      (insert (format "%s\n"
        (propertize tagsstr 'face
          ↪ 'elfeed-search-tag-face)))))

```

```

;; (insert (propertize "Link: " 'face 'message-header-name))
;; (elfeed-insert-link link link)
;; (insert "\n")
(cl-loop for enclosure in (elfeed-entry-enclosures
  ⇒ elfeed-show-entry)
  do (insert (propertize "Enclosure: " 'face
    ⇒ 'message-header-name))
    do (elfeed-insert-link (car enclosure))
    do (insert "\n"))
(insert "\n")
(if content
  (if (eq type 'html)
    (elfeed-insert-html content base)
    (insert content))
  (insert (propertize "(empty)\n" 'face 'italic)))
(goto-char (point-min))))

(after! elfeed-show
  (require 'url))

(defvar elfeed-pdf-dir
  (expand-file-name "pdfs/"
    (file-name-directory (directory-file-name
      ⇒ elfeed-enclosure-default-dir))))

(defvar elfeed-link-pdfs

  ⇒ '("https://www.jstatsoft.org/index.php/jss/article/view/v0\[^\]+\\")
  ⇒ .
  ⇒ "https://www.jstatsoft.org/index.php/jss/article/view/v0\[^\]+\\1/v\[^\]+\\1.pdf")
  ⇒ ("http://arxiv.org/abs/\[^\]+\\") .
  ⇒ "https://arxiv.org/pdf/\[^\]+\\1.pdf")
  ⇒ "List of alists of the form (REGEX-FOR-LINK . FORM-FOR-PDF)")

(defun elfeed-show-pdf (entry)
  (interactive
    (list (or elfeed-show-entry (elfeed-search-selected
      ⇒ :ignore-region))))
  (let ((link (elfeed-entry-link entry))
    (feed-name (plist-get (elfeed-feed-meta (elfeed-entry-feed
      ⇒ entry)) :title))
    (title (elfeed-entry-title entry))
    (file-view-function
      (lambda (f)
        (when elfeed-show-entry
          (elfeed-kill-buffer))
        (pop-to-buffer (find-file-noselect f)))))
    pdf)

  (let ((file (expand-file-name
    (concat (subst-char-in-string ?/ ?, title) ".pdf")
    (expand-file-name (subst-char-in-string ?/ ?,
      ⇒ feed-name)

```

```

                                elfeed-pdf-dir))))
(if (file-exists-p file)
    (funcall file-view-function file)
    (dolist (link-pdf elfeed-link-pdfs)
      (when (and (string-match-p (car link-pdf) link)
                  (not pdf))
        (setq pdf (replace-regexp-in-string (car link-pdf) (cdr
↪ link-pdf) link))))
    (if (not pdf)
        (message "No associated PDF for entry")
        (message "Fetching %s" pdf)
        (unless (file-exists-p (file-name-directory file))
          (make-directory (file-name-directory file) t))
        (url-copy-file pdf file)
        (funcall file-view-function file))))))

```

4 Org

4.1 Org-Mode

Org mode is the best writing format, no contest. The defaults are more terminal-oriented, so lets make it look a little better

I like a little padding on my org blocks, just a millimeter or two on the top and bottom should do

```

(use-package! org-padding)
(add-hook 'org-mode-hook #'org-padding-mode)
(setq org-padding-block-begin-line-padding '(1.15 . 0.15))
(setq org-padding-block-end-line-padding '(1.15 . 0.15))

```

Some hooks are a bit annoying, so lets make them shut up

```

(defadvice! shut-up-org-problematic-hooks (orig-fn &rest args)
  :around #'org-fancy-priorities-mode
  :around #'org-superstar-mode
  (ignore-errors (apply orig-fn args)))

```

I prefer to preview my images

```

(setq org-startup-with-inline-images t)

```

Lets add org pretty table as well

```
(use-package! org-pretty-table
  :commands (org-pretty-table-mode global-org-pretty-table-mode))
```

Sadly I can't always work in org, but I can import stuff into it!

```
(use-package! org-pandoc-import
  :after org)
```

I prefer /org as my directory. Lets change some other defaults too

```
(setq org-directory "~/org" ; let's put files here
      org-use-property-inheritance t ; it's convenient to
      ↪ have properties inherited
      org-log-done 'time ; having the time a
      ↪ item is done sounds convenient
      org-list-allow-alphabetical t ; have a. A. a) A) list
      ↪ bullets
      org-export-in-background t ; run export processes
      ↪ in external emacs process
      org-catch-invisible-edits 'smart) ; try not to
      ↪ accidentally do weird stuff in invisible regions
```

I want to slightly change the default args for babel

```
(setq org-babel-default-header-args
  '(:session . "none")
    (:results . "replace")
    (:exports . "code")
    (:cache . "no")
    (:noweb . "no")
    (:hlines . "no")
    (:tangle . "no")
    (:comments . "link")))
```

Add auto-fill-mode too

```
(add-hook 'text-mode-hook #'auto-fill-mode)
```

Lastly, some nice maps for org-mode, using g + arrows to move up/down headings

```
(map! :map evil-org-mode-map
      :after evil-org
      :n "g <up>" #'org-backward-heading-same-level)
```

```
:n "g <down>" #'org-forward-heading-same-level
:n "g <left>" #'org-up-element
:n "g <right>" #'org-down-element)
```

I also want to change the order of bullets

```
(setq org-list-demote-modify-bullet '("(" . "-" ) ("-" . "+") ("*" . "+")
↳ ("1." . "a."))
```

Lets add some spellcheck

```
(add-hook 'org-mode-hook 'turn-on-flyspell)
```

org-ol-tree is nice for viewing the structure of an org file

```
(use-package! org-ol-tree
  :commands org-ol-tree)
(map! :map org-mode-map
  :after org
  :localleader
  :desc "Outline" "O" #'org-ol-tree)
```

4.1.1 HTML

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

:header-args:emacs-lisp: :noweb-ref ox-html-conf For some reason this only works if you have org first

```
(after! org
  (define-minor-mode org-fancy-html-export-mode
```

```

"Toggle my fabulous org export tweaks. While this mode itself does a
  little bit,
  the vast majority of the change in behaviour comes from switch
  statements in:
  - `org-html-template-fancier'
  - `org-html--build-meta-info-extended'
  - `org-html-src-block-collapsible'
  - `org-html-block-collapsible'
  - `org-html-table-wrapped'
  - `org-html--format-toc-headline-collapseable'
  - `org-html--toc-text-stripped-leaves'
  - `org-export-html-headline-anchor'"

:global t
:init-value t
(if org-fancy-html-export-mode
  (setq org-html-style-default org-html-style-fancy
        org-html-meta-tags #'org-html-meta-tags-fancy
        org-html-checkbox-type 'html-span)
  (setq org-html-style-default org-html-style-plain
        org-html-meta-tags #'org-html-meta-tags-default
        org-html-checkbox-type 'html)))

(defadvice! org-html-template-fancier (orig-fn contents info)
  "Return complete document string after HTML conversion.
  CONTENTS is the transcoded contents string. INFO is a plist
  holding export options. Adds a few extra things to the body
  compared to the default implementation."
  :around #'org-html-template
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
    ↪ org-msg-export-in-progress))
    (funcall orig-fn contents info)
    (concat
      (when (and (not (org-html-xml5-p info)) (org-html-xml-p info))
        (let* ((xml-declaration (plist-get info :html-xml-declaration))
              (decl (or (and (stringp xml-declaration) xml-declaration)
                        (cdr (assoc (plist-get info :html-extension)
                                    xml-declaration))
                        (cdr (assoc "html" xml-declaration))
                        "")))
          (when (not (or (not decl) (string= "" decl)))
            (format "%s\n"
              (format decl
                (or (and org-html-coding-system
                  (fboundp 'coding-system-get)
                  (coding-system-get
                    ↪ org-html-coding-system
                    ↪ 'mime-charset))
                  "iso-8859-1"))))))
        (org-html-doctype info)
        "\n"
        (concat "<html"
          (cond ((org-html-xml-p info)
            (format

```



```

        " xmlns=\"http://www.w3.org/1999/xhtml\" lang=\"%s\"
        ↪ xml:lang=\"%s\"
        (plist-get info :language) (plist-get info
        ↪ :language)))
        ((org-html-html5-p info)
        (format " lang=\"%s\" (plist-get info :language)))
    ">\n")
"<head>\n"
(org-html--build-meta-info info)
(org-html--build-head info)
(org-html--build-mathjax-config info)
"</head>\n"
"<body>\n<input type='checkbox' id='theme-switch'><div
↪ id='page'><label id='switch-label' for='theme-switch'></label>"
(let ((link-up (org-trim (plist-get info :html-link-up)))
      (link-home (org-trim (plist-get info :html-link-home))))
  (unless (and (string= link-up "") (string= link-home ""))
    (format (plist-get info :html-home/up-format)
            (or link-up link-home)
            (or link-home link-up))))
;; Preamble.
(org-html--build-pre/postamble 'preamble info)
;; Document contents.
(let ((div (assq 'content (plist-get info :html-divs))))
  (format "<%s id=\"%s\">\n" (nth 1 div) (nth 2 div)))
;; Document title.
(when (plist-get info :with-title)
  (let ((title (and (plist-get info :with-title)
                    (plist-get info :title)))
        (subtitle (plist-get info :subtitle))
        (html5-fancy (org-html--html5-fancy-p info)))
    (when title
      (format
        (if html5-fancy
          "<header class=\"page-header\">%s\n<h1
          ↪ class=\"title\">%s</h1>\n%s</header>"
          "<h1 class=\"title\">%s</h1>\n")
        (if (or (plist-get info :with-date)
                (plist-get info :with-author))
          (concat "<div class=\"page-meta\">"
                  (when (plist-get info :with-date)
                    (org-export-data (plist-get info :date) info))
                  (when (and (plist-get info :with-date) (plist-get
                    ↪ info :with-author)) ", ")
                  (when (plist-get info :with-author)
                    (org-export-data (plist-get info :author)
                    ↪ info))
                  "</div>\n")
          ""))
        (org-export-data title info)
        (if subtitle
          (format
            (if html5-fancy

```

```

        "<p class=\"subtitle\"
        ↪ role=\"doc-subtitle\">%s</p>\n"
        (concat "\n" (org-html-close-tag "br" nil info) "\n"
        " <span class=\"subtitle\">%s</span>\n"))
        (org-export-data subtitle info))
        "")))))

contents
(format "</%s>\n" (nth 1 (assq 'content (plist-get info
↪ :html-divs))))
;; Postamble.
(org-html--build-pre/postamble 'postamble info)
;; Possibly use the Klipse library live code blocks.
(when (plist-get info :html-klipsify-src)
  (concat "<script>" (plist-get info :html-clipse-selection-script)
  "</script><script src=\""
  org-html-clipse-js
  "\"></script><link rel=\"stylesheet\" type=\"text/css\"
  ↪ href=\""
  org-html-clipse-css "\"/>"))
;; Closing document.
"</div>\n</body>\n</html>"))))

(defadvice! org-html-toc-linked (depth info &optional scope)
  "Build a table of contents.
  Just like `org-html-toc`, except the header is a link to `\"#\"`.
  DEPTH is an integer specifying the depth of the table. INFO is
  a plist used as a communication channel. Optional argument SCOPE
  is an element defining the scope of the table. Return the table
  of contents as a string, or nil if it is empty."
  :override #'org-html-toc
  (let ((toc-entries
        (mapcar (lambda (headline)
                  (cons (org-html--format-toc-headline headline info)
                        (org-export-get-relative-level headline info)))
                (org-export-collect-headlines info depth scope))))
    (when toc-entries
      (let ((toc (concat "<div id=\"text-table-of-contents\">"
                        (org-html--toc-text toc-entries)
                        "</div>\n")))
        (if scope toc
            (let ((outer-tag (if (org-html--html5-fancy-p info)
                                "nav"
                                "div")))
              (concat (format "<%s id=\"table-of-contents\">\n" outer-tag)
                      (let ((top-level (plist-get info
↪ :html-toplevel-hlevel)))
                        (format "<h%d><a href=\"#\" style=\"color:inherit;
↪ text-decoration: none;\">%s</a></h%d>\n"
                        top-level
                        (org-html--translate "Table of Contents"
↪ info)
                        top-level)))
                      toc
                      toc

```

```

(format "</%s>\n" outer-tag))))))

(defvar org-html-meta-tags-opengraph-image
  '(:image "https://tecosaur.com/resources/org/nib.png"
    :type "image/png"
    :width "200"
    :height "200"
    :alt "Green fountain pen nib")
  "Plist of og:image:PROP properties and their value, for use in
  ↪ `org-html-meta-tags-fancy'."

(defun org-html-meta-tags-fancy (info)
  "Use the INFO plist to construct the meta tags, as described in
  ↪ `org-html-meta-tags'."
  (let ((title (org-html-plain-text
                 (org-element-interpret-data (plist-get info :title))
                 ↪ info))
        (author (and (plist-get info :with-author)
                      (let ((auth (plist-get info :author)))
                        ;; Return raw Org syntax.
                        (and auth (org-html-plain-text
                              (org-element-interpret-data auth)
                              ↪ info))))))
    (append
     (list
      (when (org-string-nw-p author)
        (list "name" "author" author))
      (when (org-string-nw-p (plist-get info :description))
        (list "name" "description"
              (plist-get info :description)))
      ("name" "generator" "org mode")
      ("name" "theme-color" "#77aa99")
      ("property" "og:type" "article")
      (list "property" "og:title" title)
      (let ((subtitle (org-export-data (plist-get info :subtitle) info))
            (when (org-string-nw-p subtitle)
              (list "property" "og:description" subtitle))))
      (when org-html-meta-tags-opengraph-image
        (list (list "property" "og:image" (plist-get
                                           ↪ org-html-meta-tags-opengraph-image :image))
              (list "property" "og:image:type" (plist-get
                                                ↪ org-html-meta-tags-opengraph-image :type))
              (list "property" "og:image:width" (plist-get
                                                  ↪ org-html-meta-tags-opengraph-image :width))
              (list "property" "og:image:height" (plist-get
                                                  ↪ org-html-meta-tags-opengraph-image :height))
              (list "property" "og:image:alt" (plist-get
                                              ↪ org-html-meta-tags-opengraph-image :alt))))
      (list
       (when (org-string-nw-p author)
         (list "property" "og:article:author:first_name" (car
                                                           ↪ (s-split-up-to " " author 2))))
       (when (and (org-string-nw-p author) (s-contains-p " " author))

```

```

    (list "property" "og:article:author:last_name" (cadr
      ↪ (s-split-up-to " " author 2)))
  (list "property" "og:article:published_time"
    (format-time-string
      "%FT%T%z"
      (or
        (when-let ((date-str (cadr (org-collect-keywords
          ↪ ("DATE")))))
          (unless (string= date-str (format-time-string "%F"))
            (ignore-errors (encode-time (org-parse-time-string
              ↪ date-str))))))
        (if buffer-file-name
          (file-attribute-modification-time (file-attributes
            ↪ buffer-file-name))
          (current-time))))))
  (when buffer-file-name
    (list "property" "og:article:modified_time"
      (format-time-string "%FT%T%z"
        ↪ (file-attribute-modification-time (file-attributes
          ↪ buffer-file-name))))))

(unless (functionp #'org-html-meta-tags-default)
  (defalias 'org-html-meta-tags-default #'ignore))
(setq org-html-meta-tags #'org-html-meta-tags-fancy)

(setq org-html-style-plain org-html-style-default
  org-html-htmlize-output-type 'css
  org-html-doctype "html5"
  org-html-html5-fancy t)

(defun org-html-reload-fancy-style ()
  (interactive)
  (setq org-html-style-fancy
    (concat (f-read-text (expand-file-name
      ↪ "misc/org-export-header.html" doom-private-dir))
      "<script>\n"
      (f-read-text (expand-file-name "misc/org-css/main.js"
        ↪ doom-private-dir))
      "</script>\n<style>\n"
      (f-read-text (expand-file-name
        ↪ "misc/org-css/main.min.css" doom-private-dir))
      "</style>"))
  (when org-fancy-html-export-mode
    (setq org-html-style-default org-html-style-fancy)))
(org-html-reload-fancy-style)

(defvar org-html-export-collapsed nil)
(eval '(:cl-pushnew '(:collapsed "COLLAPSED" "collapsed"
  ↪ org-html-export-collapsed t)
  (org-export-backend-options (org-export-get-backend
    ↪ 'html))))
(add-to-list 'org-default-properties "EXPORT_COLLAPSED")

```

```

(defadvice! org-html-src-block-collapsible (orig-fn src-block contents
  ↪ info)
  "Wrap the usual <pre> block in a <details>"
  :around #'org-html-src-block
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪ org-msg-export-in-progress))
      (funcall orig-fn src-block contents info)
      (let* ((properties (cadr src-block))
              (lang (mode-name-to-lang-name
                    (plist-get properties :language)))
              (name (plist-get properties :name))
              (ref (org-export-get-reference src-block info))
              (collapsed-p (member (or (org-export-read-attribute :attr_html
  ↪ src-block :collapsed)
                                      (plist-get info :collapsed))
                                  '("y" "yes" "t" t "true" "all"))))
              (format
                "<details id='%s' class='code'%s><summary%s>%s</summary>
                <div class='gutter'>
                <a href='##%s'>#</a>
                <button title='Copy to clipboard'
                onclick='copyPreToClipboard(this)'> </button>\
                </div>
                %s
                </details>"
                ref
                (if collapsed-p "" " open")
                (if name " class='named'" "")
                (concat
                  (when name (concat "<span class=\"name\">" name "</span>"))
                  "<span class=\"lang\">" lang "</span>"))
                ref
                (if name
                  (replace-regexp-in-string (format "<pre\\(
  ↪ class=\"[^\"]+\"\\)? id=\"%s\">" ref) "<pre\\1>"
                    (funcall orig-fn src-block contents
  ↪ info))
                  (funcall orig-fn src-block contents info))))))

(defun mode-name-to-lang-name (mode)
  (or (cadr (assoc mode
    '("asymptote" "Asymptote")
      ("awk" "Awk")
      ("C" "C")
      ("clojure" "Clojure")
      ("css" "CSS")
      ("D" "D")
      ("ditaa" "ditaa")
      ("dot" "Graphviz")
      ("calc" "Emacs Calc")
      ("emacs-lisp" "Emacs Lisp")
      ("fortran" "Fortran")
      ("gnuplot" "gnuplot"))

```

```
("haskell" "Haskell")
("hledger" "hledger")
("java" "Java")
("js" "Javascript")
("latex" "LaTeX")
("ledger" "Ledger")
("lisp" "Lisp")
("lilypond" "Lilypond")
("lua" "Lua")
("matlab" "MATLAB")
("mscgen" "Mscgen")
("ocaml" "Objective Caml")
("octave" "Octave")
("org" "Org mode")
("oz" "OZ")
("plantuml" "Plantuml")
("processing" "Processing.js")
("python" "Python")
("R" "R")
("ruby" "Ruby")
("sass" "Sass")
("scheme" "Scheme")
("screen" "Gnu Screen")
("sed" "Sed")
("sh" "shell")
("sql" "SQL")
("sqlite" "SQLite")
("forth" "Forth")
("io" "IO")
("j" "J")
("makefile" "Makefile")
("maxima" "Maxima")
("perl" "Perl")
("picolisp" "Pico Lisp")
("scala" "Scala")
("shell" "Shell Script")
("ebnf2ps" "ebfn2ps")
("cpp" "C++")
("abc" "ABC")
("coq" "Coq")
("groovy" "Groovy")
("bash" "bash")
("csh" "csh")
("ash" "ash")
("dash" "dash")
("ksh" "ksh")
("mksh" "mksh")
("posh" "posh")
("ada" "Ada")
("asm" "Assembler")
("caml" "Caml")
("delphi" "Delphi")
("html" "HTML")
```

```

        ("idl" "IDL")
        ("mercury" "Mercury")
        ("metapost" "MetaPost")
        ("modula-2" "Modula-2")
        ("pascal" "Pascal")
        ("ps" "PostScript")
        ("prolog" "Prolog")
        ("simula" "Simula")
        ("tcl" "tcl")
        ("tex" "LaTeX")
        ("plain-tex" "TeX")
        ("verilog" "Verilog")
        ("vhdl" "VHDL")
        ("xml" "XML")
        ("nxml" "XML")
        ("conf" "Configuration File"))))

mode))

(defadvice! org-html-table-wrapped (orig-fn table contents info)
  "Wrap the usual <table> in a <div>"
  :around #'org-html-table
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
    ↪ org-msg-export-in-progress))
      (funcall orig-fn table contents info)
      (let* ((name (plist-get (cadr table) :name))
              (ref (org-export-get-reference table info)))
        (format "<div id='%s' class='table'>
          <div class='gutter'><a href='%s'>#</a></div>
          <div class='tabular'>
            %s
          </div>\
        </div>"
          ref ref
          (if name
            (replace-regexp-in-string (format "<table id=\"%s\""
              ↪ ref) "<table"
              (funcall orig-fn table
                ↪ contents info))
            (funcall orig-fn table contents info))))))

(defadvice! org-html--format-toc-headline-collapseable (orig-fn headline
  ↪ info)
  "Add a label and checkbox to `org-html--format-toc-headline's usual
  output,
  to allow the TOC to be a collapseable tree."
  :around #'org-html--format-toc-headline
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
    ↪ org-msg-export-in-progress))
      (funcall orig-fn headline info)
      (let ((id (or (org-element-property :CUSTOM_ID headline)
                    (org-export-get-reference headline info))))))

```

```

(format "<input type='checkbox' id='toc--%s'><label
  ↳ for='toc--%s'>%s</label>"
      id id (funcall orig-fn headline info))))

(defadvice! org-html--toc-text-stripped-leaves (orig-fn toc-entries)
  "Remove label"
  :around #'org-html--toc-text
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
    ↳ org-msg-export-in-progress))
      (funcall orig-fn toc-entries)
      (replace-regexp-in-string "<input [^>]+><label
    ↳ [^>]+>\\(\\.+?\\)\\</label></li>" "\\1</li>"
        (funcall orig-fn toc-entries))))

(setq org-html-text-markup-alist
  '( (bold . "<b>%s</b>")
    (code . "<code>%s</code>")
    (italic . "<i>%s</i>")
    (strike-through . "<del>%s</del>")
    (underline . "<span class='underline'>%s</span>")
    (verbatim . "<kbd>%s</kbd>")))

(appendq! org-html-checkbox-types
  '( (html-span
    (on . "<span class='checkbox'></span>")
    (off . "<span class='checkbox'></span>")
    (trans . "<span class='checkbox'></span>")))

(setq org-html-checkbox-type 'html-span)

(pushnew! org-html-special-string-regexps
  '("&gt;" . "&#594;")
  '("&lt;-" . "&#592;"))

(defun org-export-html-headline-anchor (text backend info)
  (when (and (org-export-derived-backend-p backend 'html)
    (not (org-export-derived-backend-p backend 're-reveal))
    org-fancy-html-export-mode)
    (unless (bound-and-true-p org-msg-export-in-progress)
      (replace-regexp-in-string
        "<h\\([0-9]\\) id='\\([a-z0-9-]+\\)'>\\([.*[^ ]\\)\\</h[0-9]>" ;
        ↳ this is quite restrictive, but due to
        ↳ `org-reference-contraction' I can do this
        "<h\\1 id='\\2\\'>\\3<a aria-hidden='true' href='\"#\\2\">#</a>
        ↳ </h\\1>"
        text))))

(add-to-list 'org-export-filter-headline-functions
  'org-export-html-headline-anchor)

(org-link-set-parameters "Https"
  :follow (lambda (url arg) (browse-url (concat
    ↳ "https:" url) arg))
  :export #'org-url-fancy-export)

```



```

(defun org-url-fancy-export (url _desc backend)
  (let ((metadata (org-url-unfurl-metadata (concat "https:" url))))
    (cond
      ((org-export-derived-backend-p backend 'html)
       (concat
        "<div class=\"link-preview\">"
        (format "<a href=\"%s\">" (concat "https:" url))
        (when (plist-get metadata :image)
          (format "<img src=\"%s\"/>" (plist-get metadata :image)))
        "<small>"
        (replace-regexp-in-string "//\\(?:www\\.\\|)?\\([\\/]+\\)/?.*" "\\1"
          url)
        "</small><p>"
        (when (plist-get metadata :title)
          (concat "<b>" (org-html-encode-plain-text (plist-get metadata
            ↪ :title)) "</b><br>"))
        (when (plist-get metadata :description)
          (org-html-encode-plain-text (plist-get metadata :description)))
        "</p></a></div>"))
      (t url))))

(setq org-url-unfurl-metadata--cache nil)
(defun org-url-unfurl-metadata (url)
  (cdr (or (assoc url org-url-unfurl-metadata--cache)
    (car (push
      (cons
        url
        (let* ((head-data
          (-filter #'listp
            (caddr
              (with-current-buffer (progn (message
                ↪ "Fetching metadata from %s" url)
                ↪ (url-retrieve-synchron
                ↪ url t
                ↪ t 5))
              (goto-char (point-min))
              (delete-region (point-min) (-
                ↪ (search-forward "<head") 6))
              (delete-region (search-forward
                ↪ "</head>") (point-max))
              (goto-char (point-min))
              (while (re-search-forward
                ↪ "<script[^\u2800]+?</script>"
                ↪ nil t)
                (replace-match ""))
              (goto-char (point-min))
              (while (re-search-forward
                ↪ "<style[^\u2800]+?</style>" nil
                ↪ t)
                (replace-match ""))

```

```

        (libxml-parse-html-region
         ↪ (point-min) (point-max))))))
(meta (delq nil
  (mapcar
    (lambda (tag)
      (when (eq 'meta (car tag))
        (cons (or (cdr (assoc 'name
          ↪ (cadr tag)))
              (cdr (assoc 'property
                ↪ (cadr tag)))
              (cdr (assoc 'content
                ↪ (cadr tag))))))
      head-data))))
(let ((title (or (cdr (assoc "og:title" meta))
  (cdr (assoc "twitter:title" meta))
  (nth 2 (assq 'title head-data))))
  (description (or (cdr (assoc "og:description"
    ↪ meta))
    (cdr (assoc
      ↪ "twitter:description"
      ↪ meta))
    (cdr (assoc "description"
      ↪ meta))))))
  (image (or (cdr (assoc "og:image" meta))
    (cdr (assoc "twitter:image"
      ↪ meta))))))
(when image
  (setq image (replace-regexp-in-string
    "^/" (concat "https://")
    ↪ (replace-regexp-in-string
    ↪ "//\\\[^\"]+\\\[\"/?.*\" \"\\1\" url)
    ↪ "/")
    (replace-regexp-in-string
      "^//" "https://"
      image))))
(list :title title :description description :image
  ↪ image)))
org-url-unfurl-metadata--cache))))

(setq org-html-mathjax-options
'((path "https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-svg.js" )
  (scale "1")
  (autonumber "ams")
  (multlinewidth "85%")
  (tagindent ".8em")
  (tagside "right")))

(setq org-html-mathjax-template

```

```

    "<script>
      MathJax = {
        chtml: {
          scale: %SCALE
        },
        svg: {
          scale: %SCALE,
          fontCache: \"global\"
        },
        tex: {
          tags: \"%AUTONUMBER\",
          multilineWidth: \"%MULTLINEWIDTH\",
          tagSide: \"%TAGSIDE\",
          tagIndent: \"%TAGINDENT\"
        }
      };
    </script>
    <script id=\"MathJax-script\" async
      src=\"%PATH\"></script>\" )
  )

```

There are quite a few instances where I want to modify variables defined in `ox-html`, so we'll wrap the contents of this section in a

```

(after! ox-html
  <<ox-html-conf>>
)

```

Tecosaur has a good collection of fonts, might as well take some

```

<link rel="icon" href="https://tecosaur.com/resources/org/nib.ico"
  ↪ type="image/ico" />
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
  ↪ href="https://tecosaur.com/resources/org/etbookot-roman-webfont.woff2">
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
  ↪ href="https://tecosaur.com/resources/org/etbookot-italic-webfont.woff2">
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
  ↪ href="https://tecosaur.com/resources/org/Merriweather-TextRegular.woff2">
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
  ↪ href="https://tecosaur.com/resources/org/Merriweather-TextItalic.woff2">
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
  ↪ href="https://tecosaur.com/resources/org/Merriweather-TextBold.woff2">

```

```

(defun org-html-block-collapsible (orig-fn block contents info)
  "Wrap the usual block in a <details>"
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
    ↪ org-msg-export-in-progress))
    (funcall orig-fn block contents info)

```

```

(let ((ref (org-export-get-reference block info))
      (type (pcase (car block)
                   ('property-drawer "Properties"))
      (collapsed-default (pcase (car block)
                                ('property-drawer t)
                                (_ nil)))
      (collapsed-value (org-export-read-attribute :attr_html block
                                                    ↪ :collapsed))
      (collapsed-p (or (member (org-export-read-attribute :attr_html
                                                    ↪ block :collapsed)
                              ('("y" "yes" "t" t "true"))
                      (member (plist-get info :collapsed)
                              ↪ ('("all")))))
      (format
       "<details id='%s' class='code'%s>
        <summary%s>%s</summary>
        <div class='gutter'>\
        <a href='#%s'>#</a>
        <button title='Copy to clipboard'
        onclick='copyPreToClipboard(this)'> </button>\
        </div>
        %s\n
        </details>"
       ref
       (if (or collapsed-p collapsed-default) "" " open")
       (if type " class='named'" "")
       (if type (format "<span class='type'%s</span>" type) ""))
       ref
       (funcall orig-fn block contents info))))))

(advice-add 'org-html-example-block :around
↪ #'org-html-block-collapsible)
(advice-add 'org-html-fixed-width :around
↪ #'org-html-block-collapsible)
(advice-add 'org-html-property-drawer :around
↪ #'org-html-block-collapsible)

```

4.2 Org-Roam

I would like to get into the habit of using org-roam for my notes, mainly because of that cool reddit post with the server.

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

Lets set up the org-roam-ui as well

```
(use-package! websocket
  :after org-roam)

(use-package! org-roam-ui
  :after org-roam
  :commands org-roam-ui-open
  :hook (org-roam . org-roam-ui-mode)
  :config
    (setq org-roam-ui-sync-theme t
          org-roam-ui-follow t
          org-roam-ui-update-on-save t
          org-roam-ui-open-on-start t))
```

The doom-modeline is a bit messy with roam, lets adjust that

```
(defadvice! doom-modeline--buffer-file-name-roam-aware-a (orig-fun)
  :around #'doom-modeline-buffer-file-name ; takes no args
  (if (s-contains-p org-roam-directory (or buffer-file-name ""))
      (replace-regexp-in-string
        ↪ " \\(?:^\\|\\.*/\\|\\([0-9]\\{4\\}\\|\\([0-9]\\{2\\}\\|\\([0-9]\\{1\\}\\|\\1-\\2-\\3) "
        (subst-char-in-string ?_ ? buffer-file-name))
      (funcall orig-fun)))
```

Now, I want to replace the org-roam buffer with org-roam-ui, to do that, we need to disable the regular buffer

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

4.3 Org-Agenda

Set the directory

```
(setq org-agenda-files (list "~/org/school.org"
                              "~/org/todo.org"))
```

Org-super-agenda!

```
(use-package! org-super-agenda
  :commands (org-super-agenda-mode))

(after! org-agenda
```

```

(org-super-agenda-mode))

(setq org-agenda-skip-scheduled-if-done t
      org-agenda-skip-deadline-if-done t
      org-agenda-include-deadlines t
      org-agenda-block-separator nil
      org-agenda-tags-column 100 ;; from testing this seems to be a good
      ↪ value
      org-agenda-compact-blocks t)

(setq org-agenda-custom-commands
      '(("o" "Overview"
         ((agenda "" ((org-agenda-span 'day)
                       (org-super-agenda-groups
                        '(:name "Today"
                          :time-grid t
                          :date today
                          :todo "TODAY"
                          :scheduled today
                          :order 1))))))
         (alltodo "" ((org-agenda-overriding-header "")
                      (org-super-agenda-groups
                       '(:name "Next to do"
                         :todo "NEXT"
                         :order 1)
                       (:name "Important"
                          :tag "Important"
                          :priority "A"
                          :order 6)
                       (:name "Due Today"
                          :deadline today
                          :order 2)
                       (:name "Due Soon"
                          :deadline future
                          :order 8)
                       (:name "Overdue"
                          :deadline past
                          :face error
                          :order 7)
                       (:name "Assignments"
                          :tag "Assignment"
                          :order 10)
                       (:name "Issues"
                          :tag "Issue"
                          :order 12)
                       (:name "Emacs"
                          :tag "Emacs"
                          :order 13)
                       (:name "Projects"
                          :tag "Project"
                          :order 14)
                       (:name "Research"
                          :tag "Research"

```

```

      :order 15)
      (:name "To read"
       :tag "Read"
       :order 30)
      (:name "Waiting"
       :todo "WAITING"
       :order 20)
      (:name "University"
       :tag "uni"
       :order 32)
      (:name "Trivial"
       :priority ≤ "E"
       :tag ("Trivial" "Unimportant")
       :todo ("SOMEDAY" )
       :order 90)
      (:discard (:tag ("Chore" "Routine"
↪      "Daily")))))))

```

4.4 Org-Capture

Use doct

```

(use-package! doct
  :commands (doct))

```

4.4.1 Prettify

Improve the look of the capture dialog (idea borrowed from [tecosaur](#))

```

(defun org-capture-select-template-prettier (&optional keys)
  "Select a capture template, in a prettier way than default
  Lisp programs can force the template by setting KEYS to a string."
  (let ((org-capture-templates
        (or (org-contextualize-keys
             (org-capture-upgrade-templates org-capture-templates)
             org-capture-templates-contexts)
            '(("t" "Task" entry (file+headline "" "Tasks")
              "* TODO %?\n %u\n %a")))))
    (if keys
        (or (assoc keys org-capture-templates)
            (error "No capture template referred to by \"%s\" keys"
↪            keys))
        (org-mks org-capture-templates
                  "Select a capture template\n—————"
                  "Template key: ")

```

```

      `(("q" ,(concat (all-the-icons-octicon "stop" :face
        ↳ 'all-the-icons-red :v-adjust 0.01) "\tAbort")))))
(advice-add 'org-capture-select-template :override
  ↳ #'org-capture-select-template-prettier)

(defun org-mks-pretty (table title &optional prompt specials)
  "Select a member of an alist with multiple keys. Prettified.
  TABLE is the alist which should contain entries where the car is a
  string.
  There should be two types of entries.
  1. prefix descriptions like (\"a\" \"Description\")
     This indicates that 'a' is a prefix key for multi-letter
     selection, and
     that there are entries following with keys like \"ab\", \"ax\"...
  2. Select-able members must have more than two elements, with the
     first
     being the string of keys that lead to selecting it, and the
     second a
     short description string of the item.
  The command will then make a temporary buffer listing all entries
  that can be selected with a single key, and all the single key
  prefixes. When you press the key for a single-letter entry, it is
  selected.
  When you press a prefix key, the commands (and maybe further
  prefixes)
  under this key will be shown and offered for selection.
  TITLE will be placed over the selection in the temporary buffer,
  PROMPT will be used when prompting for a key. SPECIALS is an
  alist with (\"key\" \"description\") entries. When one of these
  is selected, only the bare key is returned."
  (save-window-excursion
    (let ((inhibit-quit t)
          (buffer (org-switch-to-buffer-other-window "*Org Select*"))
          (prompt (or prompt "Select: "))
          (case-fold-search
            current))
      (unwind-protect
        (catch 'exit
          (while t
            (setq-local evil-normal-state-cursor (list nil))
            (erase-buffer)
            (insert title "\n\n")
            (let ((des-keys nil)
                  (allowed-keys '("\C-g"))
                  (tab-alternatives '("\s" "\t" "\r"))
                  (cursor-type nil))
              ;; Populate allowed keys and descriptions keys
              ;; available with CURRENT selector.
              (let ((re (format "\\`%s\\(.\\)\\`"
                                (if current (regexp-quote current)
                                    "")))
                    (prefix (if current (concat current " ") "")))
                (dolist (entry table)

```



```

(pcase entry
  ;; Description.
  `((, (and key (pred (string-match re))) ,desc)
    (let ((k (match-string 1 key)))
      (push k des-keys)
      ;; Keys ending in tab, space or RET are
      ↪ equivalent.
      (if (member k tab-alternatives)
          (push "\t" allowed-keys)
          (push k allowed-keys))
      (insert (propertyize prefix 'face
        ↪ 'font-lock-comment-face) (propertyize k 'face
        ↪ 'bold) (propertyize ">" 'face
        ↪ 'font-lock-comment-face) " " desc "...")
        ↪ "\n"))))
  ;; Usable entry.
  `((, (and key (pred (string-match re))) ,desc . ,_)
    (let ((k (match-string 1 key)))
      (insert (propertyize prefix 'face
        ↪ 'font-lock-comment-face) (propertyize k 'face
        ↪ 'bold) " " desc "\n")
      (push k allowed-keys)))
    (_ nil)))
;; Insert special entries, if any.
(when specials
  (insert "—————\n")
  (pcase-dolist `((,key ,description) specials)
    (insert (format "%s %s\n" (propertyize key 'face
      ↪ '(bold all-the-icons-red)) description))
    (push key allowed-keys)))
;; Display UI and let user select an entry or
;; a sub-level prefix.
(goto-char (point-min))
(unless (pos-visible-in-window-p (point-max))
  (org-fit-window-to-buffer))
(let ((pressed (org--mks-read-key allowed-keys prompt
  ↪ nil)))
  (setq current (concat current pressed))
  (cond
    ((equal pressed "\C-g") (user-error "Abort"))
    ((equal pressed "ESC") (user-error "Abort"))
    ;; Selection is a prefix: open a new menu.
    ((member pressed des-keys))
    ;; Selection matches an association: return it.
    ((let ((entry (assoc current table)))
      (and entry (throw 'exit entry))))
    ;; Selection matches a special entry: return the
    ;; selection prefix.
    ((assoc current specials) (throw 'exit current))
    (t (error "No entry available")))))
(when buffer (kill-buffer buffer))))
(advice-add 'org-mks :override #'org-mks-pretty)

```

The `org-capture bin` is rather nice, but I'd be nicer with a smaller frame, and no modeline.

```
(setf (alist-get 'height +org-capture-frame-parameters) 15)
;; (alist-get 'name +org-capture-frame-parameters) " Capture" ;; ATM
↳ hardcoded in other places, so changing breaks stuff
(setq +org-capture-fn
      (lambda ()
        (interactive)
        (set-window-parameter nil 'mode-line-format 'none)
        (org-capture)))
```

Sprinkle in some `doct` utility functions

```
(defun +doct-icon-declaration-to-icon (declaration)
  "Convert :icon declaration to icon"
  (let ((name (pop declaration))
        (set (intern (concat "all-the-icons-" (plist-get declaration
↳ :set))))
        (face (intern (concat "all-the-icons-" (plist-get declaration
↳ :color))))
        (v-adjust (or (plist-get declaration :v-adjust) 0.01)))
    (apply set `(:name :face ,face :v-adjust ,v-adjust))))

(defun +doct-iconify-capture-templates (groups)
  "Add declaration's :icon to each template group in GROUPS."
  (let ((templates (doct-flatten-lists-in groups)))
    (setq doct-templates (mapcar (lambda (template)
      (when-let* ((props (nthcdr (if (=
↳ (length template) 4) 2 5)
↳ template))
        (spec (plist-get
↳ (plist-get props
↳ :doct) :icon)))
        (setf (nth 1 template) (concat
↳ (+doct-icon-declaration-to-icon
↳ spec)
↳ "\t"
↳ (nth
↳ 1
↳ template))))
      template)
    templates))))

(setq doct-after-conversion-functions '(+doct-iconify-capture-templates))
```

4.4.2 Templates

```
(setq org-capture-templates
  (doct `(("Home" :keys "h"
    :icon ("home" :set "octicon" :color "cyan")
    :file "Home.org"
    :prepend t
    :headline "Inbox"
    :template ("* TODO %"
      "%i %a"))
    ("Work" :keys "w"
    :icon ("business" :set "material" :color "yellow")
    :file "Work.org"
    :prepend t
    :headline "Inbox"
    :template ("* TODO %"
      "SCHEDULED: %^{Schedule:}t"
      "DEADLINE: %^{Deadline:}t"
      "%i %a"))
    ("Note" :keys "n"
    :icon ("sticky-note" :set "faicon" :color "yellow")
    :file "Notes.org"
    :template ("* *%"
      "%i %a"))
    ("Project" :keys "p"
    :icon ("repo" :set "octicon" :color "silver")
    :prepend t
    :type entry
    :headline "Inbox"
    :template ("* %{keyword} %"
      "%i"
      "%a")
    :file ""
    :custom (:keyword "")
    :children (("Task" :keys "t"
      :icon ("checklist" :set "octicon" :color
        ↪ "green")
      :keyword "TODO"
      :file +org-capture-project-todo-file)
      ("Note" :keys "n"
      :icon ("sticky-note" :set "faicon" :color
        ↪ "yellow")
      :keyword "%U"
      :file +org-capture-project-notes-file)))
  )))
```

4.5 ORG Plot

Tell it to use the doom theme colors

```

(after! org-plot
(defun org-plot/generate-theme (_type)
  "Use the current Doom theme colours to generate a GnuPlot preamble."
  (format "
    fgt = \"textcolor rgb '%s'\" # foreground text
    fgat = \"textcolor rgb '%s'\" # foreground alt text
    fgl = \"linecolor rgb '%s'\" # foreground line
    fgat = \"linecolor rgb '%s'\" # foreground alt line
    # foreground colors
    set border lc rgb '%s'
    # change text colors of tics
    set xtics @fgt
    set ytics @fgt
    # change text colors of labels
    set title @fgt
    set xlabel @fgt
    set ylabel @fgt
    # change a text color of key
    set key @fgt
    # line styles
    set linetype 1 lw 2 lc rgb '%s' # red
    set linetype 2 lw 2 lc rgb '%s' # blue
    set linetype 3 lw 2 lc rgb '%s' # green
    set linetype 4 lw 2 lc rgb '%s' # magenta
    set linetype 5 lw 2 lc rgb '%s' # orange
    set linetype 6 lw 2 lc rgb '%s' # yellow
    set linetype 7 lw 2 lc rgb '%s' # teal
    set linetype 8 lw 2 lc rgb '%s' # violet
    # border styles
    set tics out nomirror
    set border 3
    # palette
    set palette maxcolors 8
    set palette defined ( 0 '%s',\
1 '%s',\
2 '%s',\
3 '%s',\
4 '%s',\
5 '%s',\
6 '%s',\
7 '%s' )
    "

    (doom-color 'fg)
    (doom-color 'fg-alt)
    (doom-color 'fg)
    (doom-color 'fg-alt)
    (doom-color 'fg)
    ;; colours
    (doom-color 'red)
    (doom-color 'blue)
    (doom-color 'green)
    (doom-color 'magenta)
    (doom-color 'orange)

```

```

    (doom-color 'yellow)
    (doom-color 'teal)
    (doom-color 'violet)
    ;; duplicated
    (doom-color 'red)
    (doom-color 'blue)
    (doom-color 'green)
    (doom-color 'magenta)
    (doom-color 'orange)
    (doom-color 'yellow)
    (doom-color 'teal)
    (doom-color 'violet)
  ))
(defun org-plot/gnuplot-term-properties (_type)
  (format "background rgb '%s' size 1050,650"
    (doom-color 'bg)))
(setq org-plot/gnuplot-script-preamble #'org-plot/generate-theme)
(setq org-plot/gnuplot-term-extra #'org-plot/gnuplot-term-properties))

```

4.6 View Exported File

I have to export files pretty often, lets setup some keybindings to make it easier

```

;; spc+v = view exported file
(map! :map org-mode-map
  :localleader
  :desc "View exported file" "v" #'org-view-output-file)

(defun org-view-output-file (&optional org-file-path)
  "Visit buffer open on the first output file (if any) found, using
  ↪ `org-view-output-file-extensions'"
  (interactive)
  (let* ((org-file-path (or org-file-path (buffer-file-name) ""))
    (dir (file-name-directory org-file-path))
    (basename (file-name-base org-file-path))
    (output-file nil))
    (dolist (ext org-view-output-file-extensions)
      (unless output-file
        (when (file-exists-p
          (concat dir basename "." ext))
          (setq output-file (concat dir basename "." ext))))))
    (if output-file
      (if (member (file-name-extension output-file)
        ↪ org-view-external-file-extensions)
        (browse-url-xdg-open output-file)
        (pop-to-bufferpop-to-buffer (or (find-buffer-visiting
          ↪ output-file)
          (find-file-noselect output-file))))
      (message "No exported file found"))))

```

```
(defvar org-view-output-file-extensions '("pdf" "md" "rst" "txt" "tex"
↳ "html")
  "Search for output files with these extensions, in order, viewing the
  ↳ first that matches")
(defvar org-view-external-file-extensions '("html")
  "File formats that should be opened externally.")
```

4.7 Dictionaries

Lets use lexic instead of the default dictionary

```
(use-package! lexic
  :commands lexic-search lexic-list-dictionary
  :config
  (map! :map lexic-mode-map
    :n "q" #'lexic-return-from-lexic
    :nv "RET" #'lexic-search-word-at-point
    :n "a" #'outline-show-all
    :n "h" (cmd! (outline-hide-sublevels 3))
    :n "o" #'lexic-toggle-entry
    :n "n" #'lexic-next-entry
    :n "N" (cmd! (lexic-next-entry t))
    :n "p" #'lexic-previous-entry
    :n "P" (cmd! (lexic-previous-entry t))
    :n "E" (cmd! (lexic-return-from-lexic) ; expand
                (switch-to-buffer (lexic-get-buffer)))
    :n "M" (cmd! (lexic-return-from-lexic) ; minimise
                (lexic-goto-lexic))
    :n "C-p" #'lexic-search-history-backwards
    :n "C-n" #'lexic-search-history-forwards
    :n "/" (cmd! (call-interactively #'lexic-search))))

(defadvice! +lookup/dictionary-definition-lexic (identifier &optional
↳ arg)
  "Look up the definition of the word at point (or selection) using
  ↳ `lexic-search'."
  :override #' +lookup/dictionary-definition
  (interactive
    (list (or (doom-thing-at-point-or-region 'word)
              (read-string "Look up in dictionary: "))
          current-prefix-arg))
  (lexic-search identifier nil nil t))
```

5 Latex

5.1 Basic configuration

First of all, lets use pdf-tools to preview pdfs by defaults

```
(setq +latex-viewers '(pdf-tools evince zathura okular skim sumatrapdf))
```

I also want to adjust the look of those previews

```
(after! org
  (setq org-highlight-latex-and-related '(native script entities))
  (add-to-list 'org-src-block-faces '("latex" (:inherit default :extend
    ↪ t))))

(after! org
  (plist-put org-format-latex-options :background "Transparent"))
```

Lets add cdlatex org mode integration

```
(after! org
  (add-hook 'org-mode-hook 'turn-on-org-cdlatex))

(defadvice! org-edit-latex-emv-after-insert ()
  :after #'org-cdlatex-environment-indent
  (org-edit-latex-environment))
```

I like to preview images inline too

```
(setq org-display-inline-images t)
(setq org-redisplay-inline-images t)
(setq org-startup-with-inline-images "inlineimages")
```

I prefer svgs to pngs. It takes a little more time, but scales better on HiDPI displays

```
(setq-default org-html-with-latex `dvisvgm)
(setq org-preview-latex-default-process 'dvisvgm)
```

Obviously we can't edit a png though. Let use org-fragtog to toggle between previews and text mode

```
(use-package! org-fragtog
  :hook (org-mode . org-fragtog-mode))
```

Here's just my private \LaTeX config.

```
(setq org-format-latex-header "\\documentclass{article}
  \\usepackage[usenames]{xcolor}
  \\usepackage[T1]{fontenc}
  \\usepackage{booktabs}
  \\pagestyle{empty} % do not remove
  % The settings below are copied from fullpage.sty
  \\setlength{\\textwidth}{\\paperwidth}
  \\addtolength{\\textwidth}{-3cm}
  \\setlength{\\oddsidemargin}{1.5cm}
  \\addtolength{\\oddsidemargin}{-2.54cm}
  \\setlength{\\evensidemargin}{\\oddsidemargin}
  \\setlength{\\textheight}{\\paperheight}
  \\addtolength{\\textheight}{-\\headheight}
  \\addtolength{\\textheight}{-\\headsep}
  \\addtolength{\\textheight}{-\\footskip}
  \\addtolength{\\textheight}{-3cm}
  \\setlength{\\topmargin}{1.5cm}
  \\addtolength{\\topmargin}{-2.54cm}
")
```

5.2 PDF-Tools

DocView gives me a headache, but pdf-tools can be improved, lets configure it a little more

```
(use-package pdf-view
  :hook (pdf-tools-enabled . pdf-view-themed-minor-mode)
  :hook (pdf-tools-enabled . hide-mode-line-mode)
  :config
  (setq pdf-view-resize-factor 1.1)
  (setq-default pdf-view-display-size 'fit-page))
```


5.3.1 Conditional features

65

```

(defvar org-latex-caption-preamble "
  \\usepackage{subcaption}
  \\usepackage[hypcap=true]{caption}
  \\setkomafont{caption}{\\sffamily\\small}
  \\setkomafont{captionlabel}{\\upshape\\bfseries}
  \\captionsetup{justification=raggedright,singlelinecheck=true}
  \\usepackage{capt-of} % required by Org
"
  "Preamble that improves captions.")

(defvar org-latex-checkbox-preamble "
  \\newcommand{\\checkboxUnchecked}{$\\square$}
  \\newcommand{\\checkboxTransitive}{\\rlap{\\raisebox{-0.1ex}{\\hspace{0.35ex}\\Large\\t
-}}$\\square$}
  \\newcommand{\\checkboxChecked}{\\rlap{\\raisebox{0.2ex}{\\hspace{0.35ex}\\scriptsize
\\ding{52}}}$\\square$}
"
  "Preamble that improves checkboxes.")

(defvar org-latex-box-preamble "
  % args = #1 Name, #2 Colour, #3 Ding, #4 Label
  \\newcommand{\\defsimplebox}[4]{%
    \\definecolor{#1}{HTML}{#2}
    \\newenvironment{#1}[1][
    {%
      \\par\\vspace{-0.7\\baselineskip}%
      \\textcolor{#1}{#3}
    }%
    \\textcolor{#1}{\\textbf{\\def\\temp{##1}\\ifx\\temp\\empty#4\\else##1\\fi}}%
    \\vspace{-0.8\\baselineskip}
    \\begin{addmargin}[1em]{1em}
    }%
    \\end{addmargin}
    \\vspace{-0.5\\baselineskip}
  }%
  }
"
  "Preamble that provides a macro for custom boxes.")

```

```

(defvar org-latex-feature-implementations
  '(
    (image      :snippet "\\usepackage{graphicx}" :order 2)
    (svg        :snippet "\\usepackage{svg}" :order 2)
    (table      :snippet
      ⇒ "\\usepackage{longtable}\\n\\usepackage{booktabs}" :order 2)
    (cleveref   :snippet "\\usepackage[capitalize]{cleveref}" :order
      ⇒ 1)
    (underline  :snippet "\\usepackage[normalem]{ulem}" :order 0.5)
    (float-wrap :snippet "\\usepackage{wrapfig}" :order 2)
    (rotate     :snippet "\\usepackage{rotating}" :order 2)
    (caption    :snippet org-latex-caption-preamble :order 2.1)
  )

```

```

(acronym      :snippet
  ↪ "\newcommand{\acr}[1]{\protect\textls*[110]{\scshape
  ↪ #1}}\n\newcommand{\acrs}{\protect\scalebox{.91}[.84]{\hspace{0.15ex}s}"
  ↪ :order 0.4)
(italic-quotes :snippet
  ↪ "\renewcommand{\quote}{\list{}{\rightmargin\leftmargin}\item\relax\em}\n"
  ↪ :order 0.5)
(par-sep      :snippet
  ↪ "\setlength{\parskip}{\baselineskip}\n\setlength{\parindent}{0pt}\n"
  ↪ :order 0.5)
(.pifont      :snippet "\usepackage{pifont}")
(checkbox     :requires .pifont :order 3
  :snippet (concat (unless (memq 'maths features)
    "\usepackage{amssymb} % provides
    ↪ \square")
    org-latex-checkbox-preamble))
(.fancy-box   :requires .pifont :snippet org-latex-box-preamble
  ↪ :order 3.9)
(box-warning  :requires .fancy-box :snippet
  ↪ "\defsimplebox{warning}{e66100}{\ding{68}}{Warning}" :order 4)
(box-info     :requires .fancy-box :snippet
  ↪ "\defsimplebox{info}{3584e4}{\ding{68}}{Information}" :order 4)
(box-success  :requires .fancy-box :snippet
  ↪ "\defsimplebox{success}{26a269}{\ding{68}}{\vspace{-\baselineskip}}")
  ↪ :order 4)
(box-error    :requires .fancy-box :snippet
  ↪ "\defsimplebox{error}{c01c28}{\ding{68}}{Important}" :order 4))
"LaTeX features and details required to implement them.
  List where the car is the feature symbol, and the rest forms a
  plist with the
  following keys:
  - :snippet, which may be either
    - a string which should be included in the preamble
    - a symbol, the value of which is included in the preamble
    - a function, which is evaluated with the list of feature flags
  as its
    single argument. The result of which is included in the
  preamble
    - a list, which is passed to `eval', with a list of feature flags
  available
    as "\features\"
  - :requires, a feature or list of features that must be available
  - :when, a feature or list of features that when all available
  should cause this
    to be automatically enabled.
  - :prevents, a feature or list of features that should be masked
  - :order, for when ordering is important. Lower values appear
  first.
    The default is 0.
  Features that start with ! will be eagerly loaded, i.e. without
  ↪ being detected.")

```

```

(defun org-latex-detect-features (&optional buffer info)
  "List features from `org-latex-conditional-features' detected in
  ↪ BUFFER."
  (let ((case-fold-search nil))
    (with-current-buffer (or buffer (current-buffer))
      (delete-dups
        (mapcan (lambda (construct-feature)
                  (when (let ((out (pcase (car construct-feature)
                                         ((pred stringp) (car
                                         ↪ construct-feature))
                                         ((pred functionp) (funcall (car
                                         ↪ construct-feature) info))
                                         ((pred listp) (eval (car
                                         ↪ construct-feature)))
                                         ((pred symbolp) (symbol-value (car
                                         ↪ construct-feature)))
                                         (_ (user-error
                                         ↪ "org-latex-conditional-features
                                         ↪ key %s unable to be used" (car
                                         ↪ construct-feature))))))
                    (if (stringp out)
                        (save-excursion
                          (goto-char (point-min))
                          (re-search-forward out nil t))
                        out))
                  (if (listp (cdr construct-feature)) (cdr
                    ↪ construct-feature) (list (cdr
                    ↪ construct-feature))))))
        org-latex-conditional-features))))))

```

```

(defun org-latex-expand-features (features)
  "For each feature in FEATURES process :requires, :when, and :prevents
  ↪ keywords and sort according to :order."
  (dolist (feature features)
    (unless (assoc feature org-latex-feature-implementations)
      (error "Feature %s not provided in
      ↪ org-latex-feature-implementations" feature)))
    (setq current features)
    (while current
      (when-let ((requirements (plist-get (cdr (assq (car current)
        ↪ org-latex-feature-implementations)) :requires)))
        (setcdr current (if (listp requirements)
                            (append requirements (cdr current))
                            (cons requirements (cdr current)))))
        (setq current (cdr current)))
    (dolist (potential-feature
      (append features (delq nil (mapcar (lambda (feat)
        (when (plist-get (cdr
        ↪ feat) :eager)
        (car feat))))
        ↪ org-latex-feature-implementations))))))

```

```

(when-let ((prerequisites (plist-get (cdr (assoc potential-feature
  ↪ org-latex-feature-implementations)) :when)))
  (setf features (if (if (listp prerequisites)
    (cl-every (lambda (preq) (memq preq
  ↪ features)) prerequisites)
    (memq prerequisites features))
    (append (list potential-feature) features)
    (delq potential-feature features)))))
(dolist (feature features)
  (when-let ((prevents (plist-get (cdr (assoc feature
  ↪ org-latex-feature-implementations)) :prevents)))
    (setf features (cl-set-difference features (if (listp prevents)
  ↪ prevents (list prevents)))))
  (sort (delete-dups features)
    (lambda (feat1 feat2)
      (if (< (or (plist-get (cdr (assoc feat1
  ↪ org-latex-feature-implementations)) :order) 1)
        (or (plist-get (cdr (assoc feat2
  ↪ org-latex-feature-implementations)) :order) 1))
        t nil))))))

```

```

(defun org-latex-generate-features-preamble (features)
  "Generate the LaTeX preamble content required to provide FEATURES.
  This is done according to `org-latex-feature-implementations'"
  (let ((expanded-features (org-latex-expand-features features)))
    (concat
      (format "\n%% features: %s\n" expanded-features)
      (mapconcat (lambda (feature)
        (when-let ((snippet (plist-get (cdr (assoc feature
  ↪ org-latex-feature-implementations)) :snippet)))
          (concat
            (pcase snippet
              ((pred stringp) snippet)
              ((pred functionp) (funcall snippet features))
              ((pred listp) (eval `(let ((features ',features))
  ↪ (,@snippet))))
              ((pred symbolp) (symbol-value snippet))
              (_ (user-error "org-latex-feature-implementations
  ↪ :snippet value %s unable to be used"
  ↪ snippet)))
            "\n")))
        expanded-features
        ""))
      "% end features\n"))))

```

```

(defvar info--tmp nil)

(defadvice! org-latex-save-info (info &optional t_ s_)
  :before #'org-latex-make-preamble
  (setq info--tmp info))

```

```
(defadvice! org-splice-latex-header-and-generated-preamble-a (orig-fn tpl
↳ def-pkg pkg snippets-p &optional extra)
  "Dynamically insert preamble content based on
  ↳ `org-latex-conditional-preambles'."
  :around #'org-splice-latex-header
  (let ((header (funcall orig-fn tpl def-pkg pkg snippets-p extra)))
    (if snippets-p header
      (concat header
        (org-latex-generate-features-preamble
          ↳ (org-latex-detect-features nil info--tmp)
          "\n")))))
```

5.3.2 Tectonic

Tectonic is the hot new thing, which also means I can get rid of my tex installation. Dependencies are nice and auto-installed, and I don't need to bother with ascii stuff

```
(setq org-latex-pdf-process '("tectonic -X compile --print --outdir=%o -Z
↳ shell-escape %f"))
```

Now, previews won't work anymore. For that we need to set emacs to use Tectonic instead of Pdflatex

```
(setq org-preview-latex-process-alist
'((dvipng :programs
  ("tectonic" "dvipng")
  :description "dvi > png" :message "you need to install
  ↳ the programs: tectonic and dvipng."
  ↳ :image-input-type "dvi" :image-output-type "png"
  ↳ :image-size-adjust
  (1.0 . 1.0)
  :latex-compiler
  ;; tectonic doesn't have a non interactive mode
  ("tectonic --outdir %o %f")
  :image-converter
  ("dvipng -D %D -T tight -bg Transparent -o %O %f"))
  (dvisvgm :programs
  ("tectonic" "dvisvgm")
  :description "dvi > svg" :message "you need to install
  ↳ the programs: tectonic and dvisvgm."
  ↳ :image-input-type "dvi" :image-output-type "svg"
  ↳ :image-size-adjust
  (1.7 . 1.5)
  :latex-compiler
  ("tectonic --outdir %o %f")
  :image-converter
  ("dvisvgm %f -n -b min -c %S -o %O"))))
```

```
(imagemagick :programs
  ("latex" "convert")
  :description "pdf > png" :message "you need to
  ↳ install the programs: latex and imagemagick."
  ↳ :image-input-type "pdf" :image-output-type
  ↳ "png" :image-size-adjust
  (1.0 . 1.0)
  :latex-compiler
  ("tectonic --outdir %o %f")
  :image-converter
  ("convert -density %D -trim -antialias %f -quality
  ↳ 100 %O"))))
```

5.3.3 Classes

Now for some class setup

```
(after! ox-latex
  (add-to-list 'org-latex-classes
    '("cb-doc" "\\documentclass{scrartcl}"
      ("\\section{%s}" . "\\section*{%s}")
      ("\\subsection{%s}" . "\\subsection*{%s}")
      ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
      ("\\paragraph{%s}" . "\\paragraph*{%s}")
      ("\\subparagraph{%s}" . "\\subparagraph*{%s}")))))
```

And some saner defaults for them

```
(after! ox-latex
  (setq org-latex-default-class "cb-doc"
        org-latex-tables-booktabs t))
```

```
org-latex-hyperref-template
"\colorlet{greenyblue}{blue!70!green}
\colorlet{blueygreen}{blue!40!green}
\providecolor{link}{named}{greenyblue}
\providecolor{cite}{named}{blueygreen}
\hypersetup{
  pdfauthor={%a},
  pdftitle={%t},
  pdfkeywords={%k},
  pdfsubject={%d},
  pdfcreator={%c},
  pdflang={%L},
  breaklinks=true,
  colorlinks=true,
  linkcolor=,
  urlcolor=link,
  citecolor=cite\n}
\urlstyle{same}
"
org-latex-reference-command "\cref{%s}")
```

5.3.4 Packages

Add some packages. I'm trying to keep it basic for now, Alegreya for non-monospace and SF-Mono for code

```
(setq org-latex-default-packages-alist
  `(("AUTO" "inputenc" t
    ("pdflatex"))
    ("T1" "fontenc" t
    ("pdflatex"))
    (" " "fontspec" t)
    (" " "graphicx" t)
    (" " "grffile" t)
    (" " "longtable" nil)
    (" " "wrapfig" nil)
    (" " "rotating" nil)
    ("normalem" "ulem" t)
    (" " "amsmath" t)
    (" " "textcomp" t)
    (" " "amssymb" t)
    (" " "capt-of" nil)
    ("dvipsnames" "xcolor" nil)
    ("colorlinks=true, linkcolor=Blue, citecolor=BrickRed,
    ↪ urlcolor=PineGreen" "hyperref" nil)
    (" " "indentfirst" nil)
    "\\setmainfont[Ligatures=TeX]{Alegreya}"
    "\\setmonofont[Ligatures=TeX]{Liga SFMono Nerd Font}"))
```


5.3.5 Pretty code blocks

Teco is the goto for this, so basically just ripping off him. Engrave faces ftw

```
(use-package! engrave-faces-latex
  :after ox-latex
  :config
  (setq org-latex-listings 'engraved))
```

```
(defadvice! org-latex-src-block-engraved (orig-fn src-block contents
  ↪ info)
  "Like `org-latex-src-block', but supporting an engraved backend"
  :around #'org-latex-src-block
  (if (eq 'engraved (plist-get info :latex-listings))
      (org-latex-scr-block--engraved src-block contents info)
      (funcall orig-fn src-block contents info)))

(defadvice! org-latex-inline-src-block-engraved (orig-fn inline-src-block
  ↪ contents info)
  "Like `org-latex-inline-src-block', but supporting an engraved backend"
  :around #'org-latex-inline-src-block
  (if (eq 'engraved (plist-get info :latex-listings))
      (org-latex-inline-scr-block--engraved inline-src-block contents
  ↪ info)
      (funcall orig-fn src-block contents info)))

(defvar-local org-export-has-code-p nil)

(defadvice! org-export-expect-no-code (&rest _)
  :before #'org-export-as
  (setq org-export-has-code-p nil))

(defadvice! org-export-register-code (&rest _)
  :after #'org-latex-src-block-engraved
  :after #'org-latex-inline-src-block-engraved
  (setq org-export-has-code-p t))
```

```

(setq org-latex-engraved-code-preamble "
  \\usepackage{fvextra}
  \\fvset{
    commandchars=\\\\\\\\{\\},
    highlightcolor=white!95!black!80!blue,
    breaklines=true,

    breaksymbol=\\color{white!60!black}\\tiny\\ensuremath{\\hookrightarrow}
    \\renewcommand\\theFancyVerbLine{\\footnotesize\\color{black!40!white}\\arabic{FancyVer
    \\definecolor{codebackground}{HTML}{f7f7f7}
    \\definecolor{codeborder}{HTML}{f0f0f0}
    % TODO have code boxes keep line vertical alignment
    \\usepackage[breakable,xparse]{tcolorbox}
    \\DeclareTColorBox[]{}{Code}{o}%
    {colback=codebackground, colframe=codeborder,
      fontupper=\\footnotesize,
      colupper=EFD,
      IfNoValueTF={#1}%
      {boxsep=2pt, arc=2.5pt, outer arc=2.5pt,
        boxrule=0.5pt, left=2pt}%
      {boxsep=2.5pt, arc=0pt, outer arc=0pt,
        boxrule=0pt, leftrule=1.5pt, left=0.5pt},
      right=2pt, top=1pt, bottom=0.5pt,
      breakable}
  }

)

(add-to-list 'org-latex-conditional-features '((and org-export-has-code-p
↳ "\\t]*#\\+begin_src\\\\[^\\t]*#\\+BEGIN_SRC\\\\src_[A-Za-z]") .
↳ engraved-code) t)
(add-to-list 'org-latex-conditional-features '("\\t]*#\\+begin_example\\\\[^\\t]*#\\+BEGIN_EXAMPLE" .
↳ engraved-code-setup) t)
(add-to-list 'org-latex-feature-implementations '(engraved-code :requires
↳ engraved-code-setup :snippet (engrave-faces-latex-gen-preamble)
↳ :order 99) t)
(add-to-list 'org-latex-feature-implementations '(engraved-code-setup
↳ :snippet org-latex-engraved-code-preamble :order 98) t)

(defun org-latex-scr-block--engraved (src-block contents info)
  (let* ((lang (org-element-property :language src-block))
        (attributes (org-export-read-attribute :attr_latex src-block))
        (float (plist-get attributes :float))
        (num-start (org-export-get-loc src-block info))
        (retain-labels (org-element-property :retain-labels src-block))
        (caption (org-element-property :caption src-block))
        (caption-above-p (org-latex--caption-above-p src-block info))
        (caption-str (org-latex--caption/label-string src-block info))
        (placement (or (org-unbracket-string "[" "]") (plist-get
↳ attributes :placement)
↳ (plist-get info :latex-default-figure-position)))
        (float-env
        (cond
          ((string= "multicolumn" float)

```

```

(format "\\begin{listing*}[%s]\\n%s%%s\\n%s\\end{listing*}"
  placement
  (if caption-above-p caption-str "")
  (if caption-above-p "" caption-str)))
(caption
 (format "\\begin{listing}[%s]\\n%s%%s\\n%s\\end{listing}"
  placement
  (if caption-above-p caption-str "")
  (if caption-above-p "" caption-str)))
((string= "t" float)
 (concat (format "\\begin{listing}[%s]\\n"
  placement)
  "%s\\n\\end{listing}"))
(t "%s"))
(options (plist-get info :latex-minted-options))
(content-buffer
 (with-temp-buffer
  (insert
   (let* ((code-info (org-export-unravel-code src-block))
        (max-width
         (apply 'max
          (mapcar 'length
           (org-split-string (car code-info)
            "\\n")))))
    (org-export-format-code
     (car code-info)
     (lambda (loc _num ref)
      (concat
       loc
       (when ref
        ;; Ensure references are flushed to the right,
        ;; separated with 6 spaces from the widest line
        ;; of code.
        (concat (make-string (+ (- max-width (length loc))
          ↪ 6)
          ?\s)
         (format "(%s)" ref))))))
    nil (and retain-labels (cdr code-info))))
  (funcall (org-src-get-lang-mode lang))
  (engrave-faces-latex-buffer)))
(content
 (with-current-buffer content-buffer
  (buffer-string)))
(body
 (format
  ↪ "\\begin{Code}\\n\\begin{Verbatim}[%s]\\n%s\\end{Verbatim}\\n\\end{Code}"
  ;; Options.
  (concat
   (org-latex--make-option-string
    (if (or (not num-start) (assoc "linenos" options))
     options
     (append

```

```

        `(("linenos")
          ("firstnumber" ,(number-to-string (1+ num-start))))
        options)))
      (let ((local-options (plist-get attributes :options)))
        (and local-options (concat "," local-options))))
      content)))
(kill-buffer content-buffer)
;; Return value.
(format float-env body)))

(defun org-latex-inline-src-block--engraved (inline-src-block _contents
  ↪ info)
  (let ((options (org-latex--make-option-string
    (plist-get info :latex-minted-options)))
    code-buffer code)
    (setq code-buffer
      (with-temp-buffer
        (insert (org-element-property :value inline-src-block))
        (funcall (org-src-get-lang-mode
          (org-element-property :language inline-src-block)))
        (engrave-faces-latex-buffer))))
    (setq code (with-current-buffer code-buffer
      (buffer-string)))
    (kill-buffer code-buffer)
    (format "\\Verb%s{%s}"
      (if (string= options "") ""
        (format "[%s]" options))
      code)))

(defadvice! org-latex-example-block-engraved (orig-fn example-block
  ↪ contents info)
  "Like `org-latex-example-block', but supporting an engraved backend"
  :around #'org-latex-example-block
  (let ((output-block (funcall orig-fn example-block contents info)))
    (if (eq 'engraved (plist-get info :latex-listings))
      (format "\\begin{Code}[alt]\\n%s\\n\\end{Code}" output-block)
      output-block)))

```

5.3.6 ox-chameleon

Nice little package to color stuff for us.

```

(use-package! ox-chameleon
  :after ox)

```

5.3.7 Async

Run export processes in a background ... process

```
(setq org-export-in-background t)
```

5.3.8 (sub|super)script characters

Annoying having to gate these, so let's fix that

```
(setq org-export-with-sub-superscripts '{})
```

6 Mu4e

I'm trying out emails in emacs, should be nice. Related, check .mbsyncrc to setup your emails first

10 minutes is a reasonable update time

```
(setq mu4e-update-interval 300)
```

```
(set-email-account! "shaunsingh0207"
  '(mu4e-sent-folder      . "/Sent Mail")
    (mu4e-drafts-folder   . "/Drafts")
    (mu4e-trash-folder    . "/Trash")
    (mu4e-refile-folder   . "/All Mail")
    (smtpmail-smtp-user   . "shaunsingh0207@gmail.com")))

;; don't need to run cleanup after indexing for gmail
(setq mu4e-index-cleanup nil
      mu4e-index-lazy-check t)

(after! mu4e
  (setq mu4e-headers-fields
    '(:flags . 6)
      (:account-stripe . 2)
      (:from-or-to . 25)
      (:folder . 10)
      (:recipnum . 2)
      (:subject . 80)
      (:human-date . 8)))
```

```

+mu4e-min-header-frame-width 142
mu4e-headers-date-format "%d/%m/%y"
mu4e-headers-time-format " %H:%M"
mu4e-headers-results-limit 1000
mu4e-index-cleanup t)

(add-to-list 'mu4e-bookmarks
  '(:name "Yesterday's messages" :query "date:2d..1d" :key
    ↪ ?y) t)

(defvar +mu4e-header--folder-colors nil)
(appendq! mu4e-header-info-custom
  '(:folder .
    (:name "Folder" :shortname "Folder" :help "Lowest level
    ↪ folder" :function
      (lambda (msg)
        (+mu4e-colorize-str
          (replace-regexp-in-string "\\`.*/" " "
            ↪ (mu4e-message-field msg :maildir))
          '+mu4e-header--folder-colors))))))

```

We can also send messages using msmtplib

```

(after! mu4e
  (setq sendmail-program "~/.nix-profile/bin/msmtplib"
        send-mail-function #'smtpmail-send-it
        message-sendmail-f-is-evil t
        message-sendmail-extra-arguments '("--read-envelope-from")
        message-send-mail-function #'message-send-mail-with-sendmail))

```

Notifications are quite nifty, especially if I'm as lazy as I am

```
;;(setq alert-default-style 'osx-notifier)
```

7 Browsing

7.1 Webkit

Eventually I want to use emacs for everything. Instead of using xwidgets, which requires a custom (non-cached) build of emacs. Emacs-webkit is a good alternative, but is quite buggy right now. Once its stable, I'll fix this config

```

;; (use-package org
;;   :demand t)

(use-package webkit
  :defer t
  :commands webkit
  :init
  (setq webkit-search-prefix "https://google.com/search?q="
        webkit-history-file nil
        webkit-cookie-file nil
        browse-url-browser-function 'webkit-browse-url
        webkit-browse-url-force-new t
        webkit-download-action-alist '(("\\.pdf\\'" .
          ↪ webkit-download-open)
                                       ("\\.png\\'" .
          ↪ webkit-download-save)
                                       ("\\.*" . webkit-download-default)))

  (defun webkit--display-progress (progress)
    (setq webkit--progress-formatted
      (if (equal progress 100.0)
          ""
          (format "%s%.0f%%" " (all-the-icons-faicon "spinner")
            ↪ progress)))
    (force-mode-line-update)))

```

I also want to use evil bindings with this. It's not upstreamed yet, so I'll steal the ones from the repo

```

(use-package evil-collection-webkit
  :defer t
  :config
  (evil-collection-xwidget-setup))

```

7.2 IRC

I'm trying to move everything to emacs, and discord is the one electron app I need to ditch. With bitlbee and circe it should be possible

To make this easier, I

1. Have everything (serverinfo and passwords) in an authinfo.gpg file
2. Tell circe to use it
3. Use org syntax for formatting

4. Add emoji support
5. Set it up with discord

```
(defun auth-server-pass (server)
  (if-let ((secret (plist-get (car (auth-source-search :host server))
    ↪ :secret)))
    (if (functionp secret)
        (funcall secret) secret)
    (error "Could not fetch password for host %s" server)))

(defun register-irc-auths ()
  (require 'circe)
  (require 'dash)
  (let ((accounts (-filter (lambda (a) (string= "irc" (plist-get a
    ↪ :for))))
        (auth-source-search :require '(:for) :max
    ↪ 10))))
    (appendq! circe-network-options
      (mapcar (lambda (entry)
        (let* ((host (plist-get entry :host))
              (label (or (plist-get entry :label) host))
              (_ports (mapcar #'string-to-number
    ↪ (s-split "," (plist-get
    ↪ entry :port))))
              (port (if (= 1 (length _ports)) (car
    ↪ _ports) _ports))
              (user (plist-get entry :user))
              (nick (or (plist-get entry :nick) user))
              (channels (mapcar (lambda (c) (concat "#"
    ↪ c))
    ↪ (s-split "," (plist-get
    ↪ entry :channels)))))
          `(:label
            :host ,host :port ,port :nick ,nick
            :sasl-username ,user :sasl-password
    ↪ auth-server-pass
            :channels ,channels)))
        accounts))))
```

We'll just call `(register-irc-auths)` on a hook when we start Circe up.

Now we're ready to go, let's actually wire-up Circe, with one or two configuration tweaks.

```
(after! circe
  (setq-default circe-use-tls t)
  (setq circe-notifications-alert-icon
    ↪ "/usr/share/icons/breeze/actions/24/network-connect.svg"
        lui-logging-directory "~/.emacs.d/.local/etc/irc"
        lui-logging-file-format "{buffer}/%Y/%m-%d.txt"
```



```

      circe-format-self-say "{nick:+13s} | {body}"

(custom-set-faces!
 '(circe-my-message-face :weight unspecified))

(enable-lui-logging-globally)
(enable-circe-display-images)

<<org-emph-to-irc>>

<<circe-emojis>>
<<circe-emoji-alists>>

(defun named-circe-prompt ()
  (lui-set-prompt
   (concat (propertize (format "%13s > " (circe-nick))
                        'face 'circe-prompt-face)
            " ")))
(add-hook 'circe-chat-mode-hook #'named-circe-prompt)

(appendq! all-the-icons-mode-icon-alist
  '((circe-channel-mode all-the-icons-material "message" :face
    ↪ all-the-icons-lblue)
    (circe-server-mode all-the-icons-material
    ↪ "chat_bubble_outline" :face all-the-icons-purple))))

<<irc-authinfo-reader>>

(add-transient-hook! #'=irc (register-irc-auths))

```

Let's do our **bold**, *italic*, and underline in org-syntax, using IRC control characters.

```

(defun lui-org-to-irc ())
  "Examine a buffer with simple org-mode formatting, and converts the
  emphasis:
  *bold*, /italic/, and _underline_ to IRC semi-standard escape
  codes.
  =code= is converted to inverse (highlighted) text."
  (goto-char (point-min))
  (while (re-search-forward
    ↪ "\\_<\\(?:1:[*/_]=\\)\\(?:2:[^[:space:]]\\(?:[:space:]\\)\\)?\\1\\_>"
    ↪ nil t)
    (replace-match
     (concat (pcase (match-string 1)
                  ("*" "")
                  ("/" "")
                  ("_" "")
                  ("=" ""))
              (match-string 2)
              "") nil nil)))

```

```
(add-hook 'lui-pre-input-hook #'lui-org-to-irc)
```

Let's setup Circe to use some emojis

```
(defun lui-ascii-to-emoji ()
  (goto-char (point-min))
  (while (re-search-forward "\\( \\)?::?\\([^\t:space:]+\\):\\( \\)?"
    ↪ nil t)
    (replace-match
      (concat
        (match-string 1)
        (or (cdr (assoc (match-string 2) lui-emojis-alist))
            (concat ":" (match-string 2) ":"))
        (match-string 3))
      nil nil)))

(defun lui-emoticon-to-emoji ()
  (dolist (emoticon lui-emoticons-alist)
    (goto-char (point-min))
    (while (re-search-forward (concat " " (car emoticon) "\\( \\)?") nil
      ↪ t)
      (replace-match (concat " "
                            (cdr (assoc (cdr emoticon)
                                          ↪ lui-emojis-alist))
                            (match-string 1))))))

(define-minor-mode lui-emojify
  "Replace :emojis: and ;) emoticons with unicode emoji chars."
  :global t
  :init-value t
  (if lui-emojify
    (add-hook! lui-pre-input #'lui-ascii-to-emoji
      ↪ #'lui-emoticon-to-emoji)
    (remove-hook! lui-pre-input #'lui-ascii-to-emoji
      ↪ #'lui-emoticon-to-emoji)))
```

Now, some actual emojis to use.

```
(defvar lui-emojis-alist
  '(("grinning" . " 😄 ")
    ("smiley" . " 😊 ")
    ("smile" . " 😁 ")
    ("grin" . " 😏 ")
    ("laughing" . " 😂 ")
    ("sweat_smile" . " 😓 ")
    ("joy" . " 😆 ")
    ("rofl" . " 🤔 ")
    ("relaxed" . " 😌 ")
    ("blush" . " 😊 ")
    ("innocent" . " 😇 ")))
```

```

("slight_smile"      . " ")
("upside_down"       . " ")
("wink"              . " ")
("relieved"          . " ")
("heart_eyes"        . " ")
("yum"               . " ")
("stuck_out_tongue"   . " ")
("stuck_out_tongue_closed_eyes" . " ")
("stuck_out_tongue_wink" . " ")
("zany"              . " ")
("raised_eyebrow"     . " ")
("monocle"           . " ")
("nerd"              . " ")
("cool"              . " ")
("star_struck"        . " ")
("party"             . " ")
("smirk"             . " ")
("unamused"          . " ")
("disappointed"       . " ")
("pensive"           . " ")
("worried"           . " ")
("confused"          . " ")
("slight_frown"      . " ")
("frown"             . " ")
("persevere"         . " ")
("confounded"        . " ")
("tired"             . " ")
("weary"             . " ")
("pleading"          . " ")
("tear"              . " ")
("cry"               . " ")
("sob"               . " ")
("triumph"           . " ")
("angry"             . " ")
("rage"              . " ")
("exploding_head"     . " ")
("flushed"           . " ")
("hot"               . " ")
("cold"              . " ")
("scream"            . " ")
("fearful"           . " ")
("disappointed"       . " ")
("relieved"          . " ")
("sweat"             . " ")
("thinking"          . " ")
("shush"             . " ")
("liar"              . " ")
("blank_face"        . " ")
("neutral"           . " ")
("expressionless"     . " ")
("grimace"           . " ")
("rolling_eyes"      . " ")
("hushed"            . " ")

```

```

("frowning" . " ")
("anguished" . " ")
("wow" . " ")
("astonished" . " ")
("sleeping" . " ")
("drooling" . " ")
("sleepy" . " ")
("dizzy" . " ")
("zipper_mouth" . " ")
("woozy" . " ")
("sick" . " ")
("vomiting" . " ")
("sneeze" . " ")
("mask" . " ")
("bandaged_head" . " ")
("money_face" . " ")
("cowboy" . " ")
("imp" . " ")
("ghost" . " ")
("alien" . " ")
("robot" . " ")
("clap" . " ")
("thumpup" . " ")
("+1" . " ")
("thumbsdown" . " ")
("-1" . " ")
("ok" . " ")
("pinch" . " ")
("left" . " ")
("right" . " ")
("down" . " ")
("wave" . " ")
("pray" . " ")
("eyes" . " ")
("brain" . " ")
("facepalm" . " ")
("tada" . " ")
("fire" . " ")
("flying_money" . " ")
("lightbulb" . " ")
("heart" . " ")
("sparkling_heart" . " ")
("heartbreak" . " ")
("100" . " "))

(defvar lui-emoticons-alist
  '((":" . "slight_smile")
    (";" . "wink")
    (":D" . "smile")
    ("=D" . "grin")
    ("xD" . "laughing")
    (";(" . "joy")
    (":P" . "stuck_out_tongue")

```

```
(";D" . "stuck_out_tongue_wink")
("xP" . "stuck_out_tongue_closed_eyes")
(":( " . "slight_frown")
(";(" . "cry")
(";'(" . "sob")
(">:( " . "angry")
(">>:( " . "rage")
(":o" . "wow")
(":O" . "astonished")
(":/ " . "confused")
(":-/" . "thinking")
(":|" . "neutral")
(":¬" . "expressionless")))
```