Nix-Darwin-Config + Doom Emacs Configuration

Nix-powered declarative macOS configuration

Shaurya Singh

November 11, 2021

Contents

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 Introduction

Once upon a time I was a wee little lad playing around with vim. After that, my "ricing" addiction grew, and soon it turned into a dotfiles repo. Since I moved machines often, I wanted a simple way to install all dependencies for my system. What started off as a simple `install.sh` script turned into a dotfiles repo managed via YADM. However this raised a few issues:

1. It was slow and clunky. Apps like Discord and Firefox started to clutter up my `~/.config`

directory, and my `.gitignore` kept growing. With nix, my config is stored in one folder, and symlinked into place

2. Applications were all configured using different languages. With home-manager for the most part I can stick to using nix,

3. Building apps was a pain, and switching laptops was getting annoying.

1.1 Note On Installing

If you like the look of this, that's marvellous, and I'm really happy that I've made something which you may find interesting, however:

✤ Warning

This config is *insidious*. Copying the whole thing blindly can easily lead to undesired effects. I recommend copying chunks instead.

Oh, did I mention that I started this config when I didn't know any `nix` or `lisp`, and this whole thing is a hack job? If you can suggest any improvements, please do so, no matter how much criticism you include I'll appreciate it :)

1.2 Why Nix?

Nix consists of two parts: a package manager and a language. The language is a rather simple lazy (almost) pure functional language with dynamic typing that specializes in building packages. The package manager, on the other hand, is interesting and pretty unique. It all starts with one idea.

Nix stems from the idea that FHS is fundamentally incompatible with reproducibility. Every time you see a path like `/bin/python` or `/lib/libudev.so`, there are a lot of things that you don't know about the file that's located there.

What's the version of the package it came from? What are the libraries it uses? What configure flags were enabled during the build? Answers to these questions can (and most likely will) change the behaviour of an application that uses those files. There are ways to get around this in FHS – for example, link directly to `/lib/libudev.so.1.6.3` or use `/bin/python3.7` in your shebang. However, there are still a lot of unknowns.

This means that if we want to get any reproducibility and consistency, FHS does not work since there is no way to infer a lot of properties of a given file.

One solution is tools like Docker, Snap, and Flatpak that create isolated FHS environments containing fixed versions of all the dependencies of a given application, and distribute those environments. However, this solution has a host of problems.

What if we want to apply different configure flags to our application or change one of the dependencies? There is no guarantee that you would be able to get the build artifact from build instructions, since putting all the build artifacts in an isolated container guarantees consistency, not reproducibility, because during build-time, tools from host's FHS are often used, and besides the dependencies that come from other isolated environments might change.

For example, two people using the same docker image will always get the same results, but two people building the same Dockerfile can (and often do) end up with two different images.

1.3 Drawbacks of Nix (on macOS)

The biggest issue with Nix on darwin is that NixOS (and Nix on linux) takes priority. This means:

1. Apps aren't guaranteed to build on macOS
2. External dependencies and overlays (e.g. `home-manager`) aren't guaranteed to work perfectly on darwin
3. GUI application support is almost nonexistent

MacOS is also quite locked down compared to linux, which limits the customization you can do. You also need `nix-darwin` to manage flake configurations and macOS settings. Be prepared for nix (and other package managers) to break in a future macOS update. On top of this, `aarch64-darwin` is a Tier 4 platform, if packages that are failing the test aren't critical, they get merged. You will run into packages that don't run on m1 at all, and will likely have to PR or open an issue to get them fixed. Lastly, remember that `aarch64-darwin` is fairly new. Especially if you use the stable channel, expect to have to build the majority of packages from source. Even if you use the unstable/master channels, you will likely end up building some packages from source

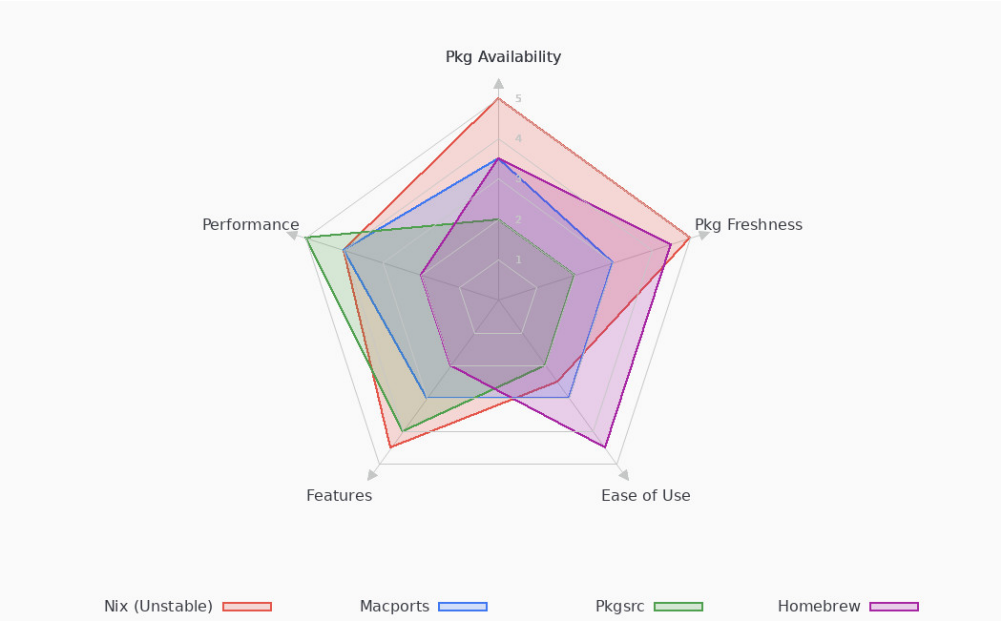
1.4 Nix vs Homebrew, Pkgsrc, and Macports

The main package managers on macOS are:

1. Nix

2. Macports
3. Pkgsrc
4. Homebrew

Pkg Manager	Pkg Availability	Pkg Freshness	Ease of Use	Features	Performance
Nix (Unstable)	5	5	2.5	4.5	4
Macports	3.5	3	3	3	4
Pkgsrc	2	2	2	4	5
Homebrew	3.5	4.5	4.5	2	2



Package management on macOS has a somewhat complex history, mostly owing to the fact that unlike most Linux distributions, macOS does not ship with a default package manager out of the box. It's not surprising that one of the first projects to solve the problem of package management, Fink, was created very early, with its initial releases predating that of Mac OS X 10.0 by several months. Using Debian's `dpkg` and `apt` as its backend, Fink is still actively maintained, though I haven't looked at it very closely.

MacPorts, on the other hand, was released in 2002 as part of OpenDarwin, while Homebrew was released seven years later as a "solution" to many of the shortcomings that the author saw in MacPorts. Pkgsrc is an older package manager for UNIX-like systems, and supports several

BSD's, as well as Linux and MacOS. Nix is a cross-platform package manager that utilizes a purely functional deployment model where software is installed into unique directories generated through cryptographic hashes. It is also the name of the tool's programming language. A package's hash takes into account the dependencies. This package management model advertises more reliable, reproducible, and portable packages.

Homebrew makes several questionable design decisions, but one of these deserves its own section: the choice to explicitly eschew root (in fact, it will refuse to work at all if run this way). This fundamentally is a very bad idea: package managers that install software for all users of your computer, as Homebrew does by default, should always require elevated privileges to function correctly. This decision has important consequences for both security and usability, especially with the advent of System Integrity Protection in OS X El Capitan.

For quite a while, Homebrew essentially considered itself the owner of `/usr/local` (both metaphorically and literally, as it would change the permissions of the directory), to the point where it would do things like plop its README down directly into this folder. After rootless was introduced, it moved most of its files to subdirectories; however, to maintain the charade of "sudoless" installation, Homebrew will still trash the permissions of folders inside `/usr/local`. Homebrew's troubleshooting guide lists these out, because reinstalling macOS sets the permissions back to what they're supposed to be and breaks Homebrew in the process.

> *If commands fail with permissions errors, check the permissions of /usr/local's subdirectories. If you're unsure what to do, you can run cd /usr/local && sudo chown -R $(whoami) bin etc include lib sbin share var opt Cellar Caskroom Frameworks.*

MacPorts, on the other hand, swings so far in the other direction that it's actually borderline inconvenient to use in some sense. Philosophically, MacPorts has a very different perspective of how it should work: it tries to prevent conflicts with the system as much as possible. To achieve this, it sets up a hierarchy under `/opt` (which is the annoying bit, because this directory is not on `$PATH` by default, nor is picked up by compilers without some prodding).

Of course, this design means that there is a single shared installation is among users, so running `port` requires elevated privileges whenever performing an operation that affects all users (which, admittedly, is most of the time). MacPorts is smart about this, though: it will shed permissions and run as the `macports` user whenever possible.

In line with their stated philosophy to prevent conflicts with macOS, MacPorts will set up its own tools in isolation from those provided by the system (in fact, builds run in "sandboxes" under the `macports` user, where attempts to access files outside of the build directory–which includes system tools–are intercepted and blocked). This means MacPorts needs to install some "duplicate" tools (whereas Homebrew will try to use the ones that come with your system where possible), the downside of which is that there is an one-time "up-front" cost as it installs base packages. The upside is that this approach is significantly more contained, which makes it eas-

ier to manage and more likely to continue working as macOS changes under it.

Finally, MacPorts just seems to have a lot of thought put into it with regards to certain aspects: for example, the MacPorts Registry database is backed by SQLite by default, which makes easily introspectable in case something goes wrong. Another useful feature is built-in "livechecks" for most ports, which codify upstream version checks and make it easy to see when MacPorts's package index need to be updated.

I won't delve too much into why I choose nix in the end (as I've covered it before), but I feel like nix takes the best of both worlds and more. You have the ease of use that homebrew provides, the sandboxing and though that was put into MacPorts, while having excellent sandboxing and the seperate `nixbld` user.

2 Installing and notes

NOTE: These are available as an executable script ./extra/install.sh

Install Nix. I have it setup for multi-user, but you can remove the `--daemon` if you want a single user install

```
sh <(curl -L https://nixos.org/nix/install) --daemon
```

Launch an ephemeral shell with git, nixUnstable, and Emacs

```
nix-shell -p nixUnstable git emacs
```

Tangle the `.org` files (not needed, but recommend in case I forgot to update tangled files)

```
git clone --depth 1 https://github.com/shaunsingh/nix-darwin-dotfiles.git
↪ ~/nix-darwin-dotfiles/ && cd nix-darwin-dotfiles
emacs --batch --eval "(progn (require 'org) (setq org-confirm-babel-evaluate nil)
↪  (org-babel-tangle-file \"~/nix-darwin-dotfiles/nix-config.org\"))"
emacs --batch --eval "(progn (require 'org) (setq org-confirm-babel-evaluate nil)
↪  (org-babel-tangle-file \"~/nix-darwin-dotfiles/configs/doom/config.org\"))"
```

Build, and switch to the dotfiles

```
nix build ~/nix-darwin-dotfiles\#darwinConfigurations.shaunsingh-laptop.system
↪ --extra-experimental-features nix-command --extra-experimental-features flakes
./result/sw/bin/darwin-rebuild switch --flake .#shaunsingh-laptop
```

(note, `--extra-experimental-features` is only needed the first time around. After that the configuration will edit `/etc/nix/nix.conf` to enable flakes and nix-command by default) Symlinking with nix (and managing doom with `nix-doom-emacs`) is very finicky, so for now we need to manually symlink them

```
ln -s ~/nix-darwin-dotfiles/configs/doom/ ~/.config/doom
ln -s ~/nix-darwin-dotfiles/configs/nvim/ ~/.config/nvim
```

Install doom emacs

```
git clone --depth 1 https://github.com/hlissner/doom-emacs ~/.config/emacs
~/.config/emacs/bin/doom install
```

2.1 Additional Configuration

2.1.1 Emacs

If you want to use Emacs-NG, use the following build options

```
git clone --depth 1 https://github.com/emacs-ng/emacs-ng.git
cd emacs-ng
./autogen.sh
./configure CFLAGS="-Wl,-rpath,shared,--disable-new-dtags -g -O3 -mtune=native
↪  -march=native -fomit-frame-pointer" \
            --prefix=/usr/local/ \
            --with-json --with-modules --with-compress-install \
            --with-threads --with-included-regex --with-zlib --with-libsystemd \
            --with-rsvg --with-native-compilation --with-webrender
            ↪  --without-javascript \
            --without-sound --without-imagemagick --without-makeinfo --without-gpm
            ↪  --without-dbus \
            --without-pop --without-toolkit-scroll-bars --without-mailutils
            ↪  --without-gsettings \
            --with-all
make -j$(($(nproc) * 2)) NATIVE_FULL_AOT=1
make install-strip
```

If you want to update the doom configuration, you can run

```
doom upgrade
```

If you modify your shell configuration, please do run `doom env` to regenerate env vars

1. Mu4e and Gmail Email will have a few issues, since its hardcoded to my account. Replace instances of my name and email in `~/.doom.d/config.org` Indexed mail will go under `~/.mbsync/`, you can either manually run mbsync or use emacs to update mail.

2. Org Mode My org mode config includes two additional plugins, org-agenda and org-roam. Both these plugins need a set directory. All org files can go under the created `~/org` dir. Roam files go under `~/org/roam`

2.1.2 Fonts

SFMono must be installed seperately due to liscensing issues, all other fonts are managed via nix.

2.1.3 Neovim

Run `:PackerSync` to install packer and plugins. Run `:checkhealth` to check for possible issues. If you want to take advantage of the LSP and/or treesitter, you can install language servers and parsers using the following command: `:LspInstall (language)` `:TSInstall (language)` **NOTE:** If you want to use neorg's treesitter parser on macOS, you need to link GCC to CC. Instructions here. I also recommend installing Neovide

3 Flakes

3.1 Why Flakes

Once upon a time, Nix pioneered reproducible builds: it tries hard to ensure that two builds of the same derivation graph produce an identical result. Unfortunately, the evaluation of Nix files into such a derivation graph isn't nearly as reproducible, despite the language being nominally purely functional.

For example, Nix files can access arbitrary files (such as `~/.config/nixpkgs/config.nix`), environment variables, Git repositories, files in the Nix search path (`$NIX_PATH`), command-line arguments (`--arg`) and the system type (`builtins.currentSystem`). In other words, evaluation isn't as hermetic as it could be. In practice, ensuring reproducible evaluation of things like NixOS system configurations requires special care.

Furthermore, there is no standard way to compose Nix-based projects. It's rare that everything you need is in Nixpkgs; consider for instance projects that use Nix as a build tool, or NixOS system configurations. Typical ways to compose Nix files are to rely on the Nix search path (e.g. `import <nixpkgs>`) or to use `fetchGit` or `fetchTarball`. The former has poor reproducibility, while the latter provides a bad user experience because of the need to manually update Git hashes to update dependencies.

There is also no easy way to deliver Nix-based projects to users. Nix has a "channel" mechanism (essentially a tarball containing Nix files), but it's not easy to create channels and they are not composable. Finally, Nix-based projects lack a standardized structure. There are some con-

ventions (e.g. `shell.nix` or `release.nix`) but they don't cover many common use cases; for instance, there is no way to discover the NixOS modules provided by a repository.

Flakes are a solution to these problems. A flake is simply a source tree (such as a Git repository) containing a file named `flake.nix` that provides a standardized interface to Nix artifacts such as packages or NixOS modules. Flakes can have dependencies on other flakes, with a "lock file" pinning those dependencies to exact revisions to ensure reproducible evaluation.

When you clone this flake and install it, your system should theoretically be the *exactly* the same as mine, down to the commit of nixpkgs. There are also other benefits, such as that nix evaluations are cached.

3.2 Notes on using the flake

When you install this config, there are 3 useful commands you need to know

- Updating the flake. This will update the `flake.lock` lockfile to the latest commit of nixpkgs, emacs-overlay, etc

```
nix flake update
```

- Building and Installing the flake. This will first build and download everything you need, then `rebuild` your machine, so it "installs"

```
nix build ~/nix-darwin-dotfiles\#darwinConfigurations.shaunsingh-laptop.system
↪  --extra-experimental-features nix-command --extra-experimental-features flakes
   ./result/sw/bin/darwin-rebuild switch --flake .#shaunsingh-laptop
```

- Testing the flake. If you have any errors when you play around with this config, then this will let you know what went wrong.

```
nix flake check
```

The `flake.nix` below does the following:

1. Add a binary cache for `nix-community` overlays
2. Add inputs (`nixpkgs-master`, `nix-darwin`, `home-manager`, and `spacebar`)
3. Add overlays to get the latest versions of `neovim` (nightly) and `emacs` (emacs29)
4. Create a nix-darwin configuration for my hostname

5. Source the mac, home, and pam modules
6. Configure home-manager and the nix-daemon
7. Enable the use of touch-id for sudo authentication
8. Configure `nixpkgs` to use the overlays above, and allow unfree packages
9. Configure `nix` to enable `flakes` and `nix-command` by default, and add `x86-64-darwin` as a platform (to install packages through rosetta)
10. Install my packages and config dependencies
11. Install the required fonts

```
{
  description = "Shaurya's Nix Environment";

  nixConfig = {
    # Add binary cache for neovim-nightly/emacsGcc
    extra-substituters =
      [ "https://cachix.cachix.org" "https://nix-community.cachix.org" ];
    extra-trusted-public-keys = [
      "cachix.cachix.org-1:eWNHQldwUO7G2VkjpnjDbWwy4KQ/HNxht7H4SSoMckM="
      "nix-community.cachix.org-1:mB9FSh9qf2dCimDSUo8Zy7bkq5CX+/rkCWyvRCYg3Fs="
    ];
  };

  inputs = {
    # All packages should follow latest nixpkgs
    unstable.url = "github:nixos/nixpkgs/master";
    darwin = {
      url = "github:LnL7/nix-darwin/master";
      inputs.nixpkgs.follows = "unstable";
    };
    home-manager = {
      url = "github:nix-community/home-manager/master";
      inputs.nixpkgs.follows = "unstable";
    };
    # Bar
    spacebar = {
      url = "github:shaunsingh/spacebar/master";
      inputs.nixpkgs.follows = "unstable";
    };
    # Editors
    neovim = {
      url = "github:nix-community/neovim-nightly-overlay";
      inputs.nixpkgs.follows = "unstable";
    };
    emacs = {
```

```
    url = "github:nix-community/emacs-overlay";
    inputs.nixpkgs.follows = "unstable";
  };
  # overlays
  rust-overlay = {
    url = "github:oxalica/rust-overlay";
    inputs.nixpkgs.follows = "unstable";
  };
  nixpkgs-overlays = {
    url = "path:./overlays/";
    inputs.nixpkgs.follows = "unstable";
  };
};

outputs = { self, nixpkgs, nixpkgs-overlays, spacebar, neovim, emacs, darwin
  , home-manager, ... }@inputs: {
    darwinConfigurations."shaunsingh-laptop" = darwin.lib.darwinSystem {
      system = "aarch64-darwin";
      modules = [
        { nixpkgs.overlays = [ nixpkgs-overlays.overlay ]; }
        ./modules/mac.nix
        ./modules/home.nix
        ./modules/pam.nix
        home-manager.darwinModule
        {
          home-manager = {
            useGlobalPkgs = true;
            useUserPackages = true;
          };
        }
        ({ pkgs, lib, ... }: {
          services.nix-daemon.enable = true;
          security.pam.enableSudoTouchIdAuth = true;
          nixpkgs = {
            overlays = with inputs; [
              spacebar.overlay
              neovim.overlay
              emacs.overlay
              rust-overlay.overlay
            ];
            config.allowUnfree = true;
          };
          nix = {
            package = pkgs.nixUnstable;
            extraOptions = ''
                                system = aarch64-darwin
                                extra-platforms = aarch64-darwin x86_64-darwin
                                experimental-features = nix-command flakes
                                build-users-group = nixbld
                              '';
          };
          environment.systemPackages = with pkgs; [
            # Emacs deps
```

```
    ((emacsPackagesNgGen emacsGcc).emacsWithPackages
      (epkgs: [ epkgs.vterm epkgs.pdf-tools ]))
    ## make sure ripgrep supports pcre2 (for vertico)
    (ripgrep.override { withPCRE2 = true; })
    binutils
    gnuplot
    sqlite
    zstd
    sdcv
    (aspellWithDicts (ds: with ds; [ en en-computers en-science ]))
    (texlive.combine {
      inherit (texlive)
        scheme-small dvipng dvisvgm l3packages xcolor soul adjustbox
        collectbox amsmath siunitx cancel mathalpha capt-of chemfig
        wrapfig mhchem fvextra cleveref latexmk tcolorbox environ arev
        amsfonts simplekv alegreya sourcecodepro newpx svg catchfile
        transparent hanging;
    })

    # Jetbrains deps
    jdk

    # Neovim deps
    neovim-nightly
    # neovide-git
    nodejs
    tree-sitter

    # Language deps
    python39Packages.grip
    python39Packages.pyflakes
    python39Packages.isort
    python39Packages.pytest
    nodePackages.pyright
    pipenv
    nixfmt
    black
    rust-analyzer
    rust-bin.nightly.latest.default
    shellcheck

    # Terminal utils and rust alternatives :tm:
    exa
    procs
    tree
    fd
    zoxide
    bottom
    discocss
  ];
  fonts = {
    enableFontDir = true;
    fonts = with pkgs; [
```

```
                overpass
                alegreya
                alegreya-sans
                emacs-all-the-icons-fonts
              ];
            };
          })
        ];
      };
    };
}
```

3.3 Overlays

Sometimes there are packages that I want from git, or that aren't available from `nixpkgs`. This overlay adds the following:

- Yabai (mac) from https://github.com/donaldguy/yabai
- Neovide (mac) from https://github.com/neovide/neovide

```
{
  description = "Shaunsingh's stash of fresh packages";

  inputs = {
    nixpkgs.url = "github:nixos/nixpkgs/master";

    flake-utils = {
      url = "github:numtide/flake-utils";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    rust-nightly = {
      url = "github:oxalica/rust-overlay";
      inputs.nixpkgs.follows = "nixpkgs";
    };

    yabai-src = {
      url = "github:donaldguy/yabai/canon";
      flake = false;
    };

    neovide-src = {
      url = "github:neovide/neovide";
      flake = false;
    };

  };
```

```
  outputs = args@{ self, flake-utils, nixpkgs, rust-nightly, ... }:
    {
      overlay = final: prev: {
        inherit (self.packages.${final.system}) yabai-git neovide-git;
      };
    } // flake-utils.lib.eachSystem [ "aarch64-darwin" ] (system:
      let
        pkgs = import nixpkgs {
          inherit system;
          overlays = [ rust-nightly.overlay ];
          # allowBroken = true;
          # allowUnsupportedSystem = true;
        };
        version = "999-unstable";
      in {

        defaultPackage = self.packages.${system}.neovide-git;

        packages = rec {

          yabai-git = (pkgs.yabai.overrideAttrs (old: {
            inherit version;
            src = args.yabai-src;
            # buildInputs = [ Carbon Cocoa ScriptingBridge xxd ];
            buildInputs = (old.buildInputs or [ ])
              ++ (with pkgs; [ xcodebuild ]);
          }));

          neovide-git = (pkgs.neovide.overrideAttrs (old: {
            inherit version;
            src = args.neovide-src;
            buildInputs = (old.buildInputs or [ ])
              ++ (with pkgs; [ rust-bin.nightly.latest.default ]);
          }));
        };
      });
}
```

4 Modules

4.1 Home.nix

Home Manager allows you to use Nix's declarative approach to manage your user-level configuration and packages. It works on any *nix system supported by Nix, including MacOS.

```
{ pkgs, lib, config, home-manager, nix-darwin, ... }: {
```

4.1.1 Doom-emacs

Nix via doom-emacs is very, *very* annoying. Initially I was using Nix-doom-emacs. However, this has a few drawbacks

1. It doesn't support straight `:recipe`, so all packages must be from melpa or elpa
2. It pins the version of doom, so you need to update doom and its dependencies painstakingly manually
3. It just ends up breaking anyways.

A simpler solution is just to have nix clone `doom-emacs` to `~/.config/emacs`, and the user can handle doom manually

```
home-manager.users.shauryasingh.home.file = {
  "~/.config/doom" = {
    recursive = true;
    source = ../configs/doom;
  };
};
#  system.activationScripts.postUserActivation.text = ''
#    if [ -d $HOME/.config/emacs ]; then
#      git clone --depth 1 https://github.com/hlissner/doom-emacs $HOME/.config/emacs
#    fi
#  '';
```

4.1.2 Git

As opposed to what the xcode CLT provides, I want the *full* package of git, and use `delta` instead of the default diff tool (rust alternatives go brr). MacOS is also quite annoying with its `.DS_Store`'s everywhere, so lets ignore that

```
home-manager.users.shauryasingh.programs.git = {
  package = pkgs.gitFull;
  enable = true;
  userName = "shaunsingh";
  userEmail = "shaunsingh0207@gmail.com";
  lfs.enable = true;
  delta = {
    enable = true;
    options = { syntax-theme = "Nord"; };
  };
  ignores = [ ".dir-locals.el" ".envrc" ".DS_Store" ];
};
```

4.1.3 IdeaVim

Intellij Idea ships with a very nice Vim emulation plugin. This is configured via a vimrc-like file (`~/.ideavimrc`). Since it doesn't have proper support in home-manger, we can just generate a file and symlink it into place

```
home-manager.users.shauryasingh.home.file = {
  ".ideavimrc".text = ''
            " settings
            set ignorecase
            set smartcase
            set scrolloff=3 " 3 lines above/below cursor when scrolling
            set nonumber
            set clipboard+=unnamed
            set multiple-cursors
            set numberwidth=2
            set expandtab=true
            set shiftwidth=4
            " plugins
            set easymotion
            set NERDTree
            set surround
            set highlightedyank
            " bindings
            let mapleader = " "
            nmap <leader>. :action GotoFile<cr>
            nmap <leader>fr :action RecentFiles<cr>
            nmap <leader>ww <Plug>(easymotion-w)
            nmap <leader>tz :action Enter Zen Mode<cr>
            nmap <leader>op :NERDTreeToggle<cr>
            nmap <leader>ot :Terminal<cr>
            nmap <leader>: :action SearchEverywhere<cr>
            nmap <leader>/ :action Find<cr>
            " use ; to enter command
            nmap ; :
            " use jk for escaping
            inoremap jk <Esc>
            cnoremap jk <Esc>
            " move by visual lines"
            nmap j gj
            nmap k gk
            " use C-hjkl to navigate splits
            nmap <C-h> <c-w>h
            nmap <C-l> <c-w>l
            nmap <C-k> <c-w>k
            nmap <C-j> <c-w>j
            nmap <leader>E :action Tool_External Tools_emacsclient<cr>
          '';
```

4.1.4 Discocss

Discocss is a way to inject custom CSS into discord. Similar to ideavim, it doesn't have proper support but we can generate a file for `~/.config/discocss/custom.css`

```
    ".config/discocss/custom.css".text = ''
          /*
              Discord Nord
              https://github.com/joinemm/discord-css
          */
          /* define colors */
          :root {
              --background-primary: #2E3440;
              --background-secondary: #3B4252;
              --background-secondary-alt: #3B4252;
              --background-tertiary: #2E3440;
              --background-accent: #2E3440;
              --channeltextarea-background: #2E3440;
              --text-normal: #D8DEE9;
              --text-muted: #616E88;
              --channels-default: #616E88;
              --interactive-normal: #616E88;
              --interactive-hover: #8FBCBB;
              --interactive-active: #D8DEE9;
              --interactive-muted: #81A1C1;
              --header-primary: #D8DEE9;
              --header-secondary: #8FBCBB;
              --background-floating: #2E3440;
              --scrollbar-auto-thumb: #3B4252;
              --scrollbar-auto-track: #2E3440;
              --text-link: #88C0D0;
              --selection: #3B4252;
          }
          * {
              font-family: Liga SFMono Nerd Font;
              font-size: 11px;
          }
          /* main backgrounds */
          .scrollerThemed-2oenus.themeDark-2cjlUp.scrollerTrack-1ZIpsv>.scroller-
    2FKFPG::-webkit-scrollbar-track,
          .scrollerThemed-2oenus.themeDark-2cjlUp.scrollerTrack-1ZIpsv>.scrollerStore-
    390omS::-webkit-scrollbar-track,
          .theme-dark .scrollerWrap-2lJEkd.scrollerTrack-1ZIpsv>.scroller-2FKFPG::-
    webkit-scrollbar-track,
          .theme-dark .scrollerWrap-2lJEkd.scrollerTrack-1ZIpsv>.scrollerStore-
    390omS::-webkit-scrollbar-track,
          .theme-dark .da-messageGroupWrapper,
          .theme-dark .bodyInner-245q0L,
          .theme-dark .bottomDivider-1K9Gao,
          .theme-dark .headerNormal-T_seeN,
          .theme-dark .root-1gCeng,
          .tabBody-3YRQ8W,
          .theme-dark .container-1D34oG
          .theme-dark .uploadModal-2ifh8j,
          .theme-dark .modal-yWgWj-,
          .uploadModal-2ifh8j,
          .theme-dark .emojiAliasInput-1y-NBz .emojiInput-1aLNse,
          .theme-dark .selected-1Tbx07,
          .theme-dark .option-96V44q.selected-rZcOL- {
              background-color: var(--background-primary) !important;
          }
          .da-popouts .da-container,
          .da-friendsTableHeader,
          .da-friendsTable,
          .da-autocomplete,
          .da-themedPopout,
          .da-footer,
          .da-userPopout>*,
          .da-autocompleteHeaderBackground,
          .theme-dark .bar-2Qqk5Z,
          .theme-dark .payment-xT17Mq,
          .theme-dark .paymentPane-3bwJ6A,
          .theme-dark .paginator-166-09,
          .theme-dark .codeRedemptionRedirect-1wVR4b,
```

4.1.5 Firefox

Although safari is my main browser, firefox looks very appealing with its excellent privacy and speed

```
home-manager.users.shauryasingh.programs.firefox.enable = true;
```

GUI apps are very finicky with nix, and so I create a fake package so that we can still use the configuration from `home-manager` without having to install it via nix. The user can then install firefox manually to `~/Applications`

```
home-manager.users.shauryasingh.programs.firefox.package =
  pkgs.runCommand "firefox-0.0.0" { } "mkdir $out";
```

Now for the configuration. We want firefox to use the css at ./configs/userChrome.css, and we want to configure the UI. Lets also enable the (rust powered ftw) webrender/servo renderer.

```
home-manager.users.shauryasingh.programs.firefox.profiles = let
  userChrome = builtins.readFile ../configs/userChrome.css;
  settings = {
    "app.update.auto" = true;
    "browser.startup.homepage" = "https://shaunsingh.github.io/startpage/";
    "browser.search.region" = "US";
    "browser.search.countryCode" = "US";
    "browser.search.isUS" = false;
    "browser.ctrlTab.recentlyUsedOrder" = false;
    "browser.newtabpage.enabled" = false;
    "browser.bookmarks.showMobileBookmarks" = true;
    "browser.uidensity" = 1;
    "browser.urlbar.placeholderName" = "DuckDuckGo";
    "browser.urlbar.update1" = true;
    "distribution.searchplugins.defaultLocale" = "en-GB";
    "general.useragent.locale" = "en-GB";
    "identity.fxaccounts.account.device.name" = config.networking.hostName;
    "privacy.trackingprotection.enabled" = true;
    "privacy.trackingprotection.socialtracking.enabled" = true;
    "privacy.trackingprotection.socialtracking.annotate.enabled" = true;
    "reader.color_scheme" = "sepia";
    "services.sync.declinedEngines" = "addons,passwords,prefs";
    "services.sync.engine.addons" = false;
    "services.sync.engineStatusChanged.addons" = true;
    "services.sync.engine.passwords" = false;
    "services.sync.engine.prefs" = false;
    "services.sync.engineStatusChanged.prefs" = true;
    "signon.rememberSignons" = false;
    "toolkit.legacyUserProfileCustomizations.stylesheets" = true;
  };
in {
  home = {
    inherit settings;
    inherit userChrome;
```

```
      id = 0;
    };

    work = {
      inherit userChrome;
      id = 1;
      settings = settings // {
        "browser.startup.homepage" = "about:blank";
        "browser.urlbar.placeholderName" = "Google";
      };
    };
  };
```

4.1.6 Alacritty

Alacritty is my terminal emulator of choice. Similar to firefox, we want to create a fake package, and then configure it as normal

```
  home-manager.users.shauryasingh.programs.alacritty = {
    enable = true;
    # We need to give it a dummy package
    package = pkgs.runCommand "alacritty-0.0.0" { } "mkdir $out";
    settings = {
      window.padding.x = 45;
      window.padding.y = 45;
      window.decorations = "buttonless";
      window.dynamic_title = true;
      live_config_reload = true;
      mouse.hide_when_typing = true;
      use_thin_strokes = true;
      cursor.style = "Beam";

      font = {
        size = 14;
        normal.family = "Liga SFMono Nerd Font";
        normal.style = "Light";
        bold.family = "Liga SFMono Nerd Font";
        bold.style = "Bold";
        italic.family = "Liga SFMono Nerd Font";
        italic.style = "Italic";
      };

      colors = {
        cursor.cursor = "#81a1c1";
        primary.background = "#2e3440";
        primary.foreground = "#d8dee9";
        normal = {
          black = "#3B4252";
          red = "#BF616A";
          green = "#A3BE8C";
```

```
        yellow = "#EBCB8B";
        blue = "#81A1C1";
        magenta = "#B48EAD";
        cyan = "#88C0D0";
        white = "#E5E9F0";
      };
      bright = {
        black = "#4c566a";
        red = "#bf616a";
        green = "#a3be8c";
        yellow = "#ebcb8b";
        blue = "#81a1c1";
        magenta = "#b48ead";
        cyan = "#8fbcbb";
        white = "#eceff4";
      };
    };
  };
};
```

4.1.7 Kitty

I no longer use kitty (its quite slow to start and has too many features I don't need), but I keep the config around just in case

```
# home-manager.users.shauryasingh.programs.kitty = {
#   enable = true;
#   package = builtins.path {
#     path = /Applications/kitty.app/Contents/MacOS;
#     filter = (path: type: type == "directory" || builtins.baseNameOf path ==
↪  "kitty");
#   };
#   # enable = true;
#   settings = {
#     font_family = "Liga SFMono Nerd Font";
#     font_size = "14.0";
#     adjust_line_height = "120%";
#     disable_ligatures = "cursor";
#     hide_window_decorations = "yes";
#     scrollback_lines = "50000";
#     cursor_blink_interval = "0.5";
#     cursor_stop_blinking_after = "10.0";
#     window_border_width = "0.7pt";
#     draw_minimal_borders = "yes";
#     macos_option_as_alt = "no";
#     cursor_shape = "beam";

#     foreground           =   "#D8DEE9";
#     background           =   "#2E3440";
#     selection_foreground =   "#000000";
```

```
#     selection_background =   "#FFFACD";
#     url_color            =   "#0087BD";
#     cursor               =   "#81A1C1";
#     color0               =   "#3B4252";
#     color8               =   "#4C566A";
#     color1               =   "#BF616A";
#     color9               =   "#BF616A";
#     color2               =   "#A3BE8C";
#     color10              =   "#A3BE8C";
#     color3               =   "#EBCB8B";
#     color11              =   "#EBCB8B";
#     color4               =   "#81A1C1";
#     color12              =   "#81A1C1";
#     color5               =   "#B48EAD";
#     color13              =   "#B48EAD";
#     color6               =   "#88C0D0";
#     color14              =   "#8FBCBB";
#     color7               =   "#E5E9F0";
#     color15              =   "#B48EAD";
#   };
# };
```

4.1.8 Fish

I like to use the fish shell. Although it isn't POSIX, it has the best autocompletion and highlighting I've seen.

```
programs.fish.enable = true;
environment.shells = with pkgs; [ fish ];
users.users.shauryasingh = {
  home = "/Users/shauryasingh";
  shell = pkgs.fish;
};
```

1. Settings fish as default On macOS nix doesn't set the fish shell to the main shell by default (like it does on NixOS), so lets do that manually

   ```
   system.activationScripts.postActivation.text = ''
         # Set the default shell as fish for the user
         sudo chsh -s ${lib.getBin pkgs.fish}/bin/fish shauryasingh
       '';
   ```

2. Aliases I also like to alias common commands with other, better rust alternatives :tm:

   ```
   programs.fish.shellAliases = with pkgs; {
     ":q" = "exit";
     vi = "emacsclient -c";
     git-rebsae = "git rebase -i HEAD~2";
   ```

```
    ll =
      "exa -lF --color-scale --no-user --no-time --no-permissions
      ↪  --group-directories-first --icons -a";
    ls = "exa -lF --group-directories-first --icons -a";
    ps = "ps";
    tree = "tree -a -C";
    cat = "bat";
    top = "btm";
    find = "fd";
    calc = "emacs -f full-calc";
    neovide =
      "/Applications/Neovide.app/Contents/MacOS/neovide --frameless --multigrid";
    nix-fish = "nix-shell --command fish";
  };
```

3. Prompt I like to make my prompt look pretty (along with some `nix-shell` and `git` integration)

```
programs.fish.promptInit = ''
      set -g fish_greeting ""
      set -U fish_color_autosuggestion      brblack
      set -U fish_color_cancel              -r
      set -U fish_color_command             green
      set -U fish_color_comment             magenta
      set -U fish_color_cwd                 green
      set -U fish_color_cwd_root            red
      set -U fish_color_end                 magenta
      set -U fish_color_error               red
      set -U fish_color_escape              cyan
      set -U fish_color_history_current     --bold
      set -U fish_color_host                normal
      set -U fish_color_normal              normal
      set -U fish_color_operator            cyan
      set -U fish_color_param               blue
      set -U fish_color_quote               yellow
      set -U fish_color_redirection         yellow
      set -U fish_color_search_match       'yellow' '--background=brightblack'
      set -U fish_color_selection           'white' '--bold'
  '--background=brightblack'
      set -U fish_color_status              red
      set -U fish_color_user                green
      set -U fish_color_valid_path          --underline
      set -U fish_pager_color_completion    normal
      set -U fish_pager_color_description   yellow
      set -U fish_pager_color_prefix        'white' '--bold' '--underline'
      set -U fish_pager_color_progress      'white' '--background=cyan'
      # prompt
      set fish_prompt_pwd_dir_length 1
      set __fish_git_prompt_show_informative_status 1
      set fish_color_command green
      set fish_color_param $fish_color_normal
      set __fish_git_prompt_showdirtystate 'yes'
      set __fish_git_prompt_showupstream 'yes'
      set __fish_git_prompt_color_branch brown
      set __fish_git_prompt_color_dirtystate FCBC47
      set __fish_git_prompt_color_stagedstate yellow
      set __fish_git_prompt_color_upstream cyan
      set __fish_git_prompt_color_cleanstate green
      set __fish_git_prompt_color_invalidstate red
      set __fish_git_prompt_char_dirtystate '~~'
```

4. Init I also want to disable the default greeting, and use tmux with fish. Lets also set `nvim` as the default editor, and add emacs to my path

```
programs.fish.interactiveShellInit = ''
      set -g fish_greeting ""
      if not set -q TMUX
        tmux new-session -A -s main
      end
      zoxide init fish --cmd cd | source
      set -x EDITOR "nvim"
      set -x PATH ~/.config/emacs/bin $PATH
    '';
```

4.1.9 Neovim

Lastly, I didn't feel like nix-ifying my neovim lua config. Lets cheat a bit and just symlink it instead

```
home-manager.users.shauryasingh.home.file = {
  "~/.config/nvim" = {
    recursive = true;
    source = ../configs/nvim;
  };
};
```

4.1.10 Bat

Bat is another rust alternative :tm: to cat, and provides syntax highlighting. Lets theme it to match nord

```
home-manager.users.shauryasingh.programs.bat = {
  enable = true;
  config = { theme = "Nord"; };
};
```

4.1.11 Tmux

Lastly, lets make tmux look just as pretty as our prompt, and enable truecolor support.

```
  programs.tmux.enable = true;
  programs.tmux.extraConfig = ''
          # make sure fish works in tmux
          set -g  default-terminal   "xterm-256color"
          set -sa terminal-overrides ',xterm-256color:RGB'
          # so that escapes register immidiately in vim
          set -sg escape-time 1
          # mouse support
          set -g mouse on
          # change prefix to C-a
          set -g prefix C-a
          bind C-a send-prefix
          unbind C-b
          # extend scrollback
          set-option -g history-limit 5000
          # vim-like pane resizing
          bind -r C-k resize-pane -U
          bind -r C-j resize-pane -D
          bind -r C-h resize-pane -L
          bind -r C-l resize-pane -R
          # vim-like pane switching
          bind -r k select-pane -U
          bind -r j select-pane -D
          bind -r h select-pane -L
          bind -r l select-pane -R
          # styling
          set -g status-style fg=white,bg=default
          set -g status-left ""
          set -g status-right ""
          set -g status-justify centre
          set -g status-position bottom
          set -g pane-active-border-style bg=default,fg=default
          set -g pane-border-style fg=default
          set -g window-status-current-format "#[fg=cyan]#[fg=black]#[bg=cyan]#I
      #[bg=brightblack]#[fg=white] #W#[fg=brightblack]#[bg=default] #[bg=default]
      #[fg=magenta]#[fg=black]#[bg=magenta]λ #[fg=white]#[bg=brightblack] %a %d %b
      #[fg=magenta]%R#[fg=brightblack]#[bg=default]"
          set -g window-status-format "#[fg=magenta]#[fg=black]#[bg=magenta]#I
      #[bg=brightblack]#[fg=white] #W#[fg=brightblack]#[bg=default] "
        '';
}
```

4.2 Mac.nix

There are mac-specific tweaks I need to do. In the future if I switch to nixOS full-time, then I wuold likely need to remove the mac-specific packages. An easy way to do this is just keep them in a seperate file:

```
{ pkgs, lib, spacebar, ... }: {
```

4.2.1 Yabai

Yabai is my tiling WM of choice. As this is an m1 (`aarch64-darwin`) laptop, I use the `the-future` branch, which enables the SA addon on m1 machines and monterey support

Now to configure the package via nix

```
services.yabai = {
  enable = false;
  enableScriptingAddition = false;
  package = pkgs.yabai-git;
  config = {
    window_border = "off";
    # window_border_width = 5;
    # active_window_border_color = "0xff3B4252";
    # normal_window_border_color = "0xff2E3440";
    focus_follows_mouse = "autoraise";
    mouse_follows_focus = "off";
    mouse_drop_action = "stack";
    window_placement = "second_child";
    window_opacity = "off";
    window_topmost = "on";
    window_shadow = "on";
    active_window_opacity = "1.0";
    normal_window_opacity = "1.0";
    split_ratio = "0.50";
    auto_balance = "on";
    mouse_modifier = "fn";
    mouse_action1 = "move";
    mouse_action2 = "resize";
    layout = "bsp";
    top_padding = 18;
    bottom_padding = 46;
    left_padding = 18;
    right_padding = 18;
    window_gap = 18;
  };
};
```

4.2.2 Spacebar

Spacebar is my bar of choice on macOS. Its lighter than any web-based ubersicht bar, and looks nice

```
services.spacebar = {
  enable = true;
  package = pkgs.spacebar;
  config = {
```

```
      position = "bottom";
      height = 28;
      title = "on";
      spaces = "on";
      power = "on";
      clock = "off";
      right_shell = "off";
      padding_left = 20;
      padding_right = 20;
      spacing_left = 25;
      spacing_right = 25;
      text_font = ''"Menlo:16.0"'';
      icon_font = ''"Menlo:16.0"'';
      background_color = "0xff2E3440";
      foreground_color = "0xffD8DEE9";
      space_icon_color = "0xff8fBcBB";
      power_icon_strip = " ";
      space_icon_strip = "I II III IV V VI VII VIII IX X";
      spaces_for_all_displays = "on";
      display_separator = "on";
      display_separator_icon = "|";
      clock_format = ''"%d/%m/%y %R"'';
      right_shell_icon = " ";
      right_shell_command = "whoami";
    };
  };
```

4.2.3 SKHD

Skhd is the hotkey daemon for yabai. As yabai is disabled, it makes sense to disable skhd too for the time being

```
  services.skhd = {
    enable = false;
    package = pkgs.skhd;
```

```
    skhdConfig = ''
              ctrl + alt - h : yabai -m window --focus west
              ctrl + alt - j : yabai -m window --focus south
              ctrl + alt - k : yabai -m window --focus north
              ctrl + alt - l : yabai -m window --focus east
              # Fill space with window
              ctrl + alt - 0 : yabai -m window --grid 1:1:0:0:1:1
              # Move window
              ctrl + alt - e : yabai -m window --display 1; yabai -m display --focus 1
              ctrl + alt - d : yabai -m window --display 2; yabai -m display --focus 2
             ctrl + alt - f : yabai -m window --space next; yabai -m space --focus next
             ctrl + alt - s : yabai -m window --space prev; yabai -m space --focus prev
              # Close current window
              ctrl + alt - w : $(yabai -m window $(yabai -m query --windows --window |
        jq -re ".id") --close)
              # Rotate tree
              ctrl + alt - r : yabai -m space --rotate 90
              # Open application
              ctrl + alt - enter : alacritty
              ctrl + alt - e : emacs
              ctrl + alt - b : open -a Safari
              ctrl + alt - t : yabai -m window --toggle float;\
                yabai -m window --grid 4:4:1:1:2:2
              ctrl + alt - p : yabai -m window --toggle sticky;\
                yabai -m window --toggle topmost;\
                yabai -m window --toggle pip
            '';
};
```

4.2.4 MacOS Settings

I like my hostname to be the same as the flake's target

```
  networking.hostName = "shaunsingh-laptop";
  system.stateVersion = 4;
```

Along with that, lets

- Increase key repeat rate
- Remap Caps to Esc
- Save screenshots to `/tmp`
- Autohide the dock and menubar
- Show extensions in Finder (and allow it to "quit")

- Set macOS to use the dark theme
- Configure Trackpad and mouse behavior
- Enable subpixel antialiasing on internal/external displays

```
  system.keyboard = {
    enableKeyMapping = true;
    remapCapsLockToEscape = true;
  };
  system.defaults = {
    screencapture = { location = "/tmp"; };
    dock = {
      autohide = true;
      showhidden = true;
      mru-spaces = false;
    };
    finder = {
      AppleShowAllExtensions = true;
      QuitMenuItem = true;
      FXEnableExtensionChangeWarning = true;
    };
    NSGlobalDomain = {
      AppleInterfaceStyle = "Dark";
      AppleKeyboardUIMode = 3;
      ApplePressAndHoldEnabled = false;
      AppleFontSmoothing = 1;
      _HIHideMenuBar = true;
      InitialKeyRepeat = 10;
      KeyRepeat = 1;
      "com.apple.mouse.tapBehavior" = 1;
      "com.apple.swipescrolldirection" = true;
    };
  };
}
```

4.3 Pam.nix

Apple's touchid is an excellent way of authenticating anything quickly and securely. Sadly, `sudo` doesn't support it by default, but its an easy fix. T do this, we edit `/etc/pam.d/sudo` via `sed` to include the relevent code to enable touchid.

We don't use `environment.etc` because this would require that the user manually delete `/etc/pam.d/sudo` which seems unwise given that applying the nix-darwin configuration requires `sudo`. We also can't use `system.patches` since it only runs once, and so won't patch in the changes again after OS updates (which remove modifications to this file).

As such, we resort to line addition/deletion in place using `sed`. We add a comment to the added line that includes the name of the option, to make it easier to identify the line that should be deleted when the option is disabled.

```
{ config, lib, pkgs, ... }:

with lib;

let
  cfg = config.security.pam;
  mkSudoTouchIdAuthScript = isEnabled:
    let
      file = "/etc/pam.d/sudo";
      option = "security.pam.enableSudoTouchIdAuth";
    in ''
      ${if isEnabled then ''
                  # Enable sudo Touch ID authentication, if not already enabled
                  if ! grep 'pam_tid.so' ${file} > /dev/null; then
                    sed -i "" '2i\
                  auth       sufficient     pam_tid.so # nix-darwin: ${option}
                    ' ${file}
                  fi
                '' else ''
                  # Disable sudo Touch ID authentication, if added by nix-darwin
                  if grep '${option}' ${file} > /dev/null; then
                    sed -i "" '/${option}/d' ${file}
                  fi
                ''}
              '';

in {
  options = {
    security.pam.enableSudoTouchIdAuth = mkEnableOption ''
              Enable sudo authentication with Touch ID
              When enabled, this option adds the following line to /etc/pam.d/sudo:
                  auth       sufficient     pam_tid.so
              (Note that macOS resets this file when doing a system update. As such,
        sudo
              authentication with Touch ID won't work after a system update until the
        nix-darwin
              configuration is reapplied.)
            '';
  };

  config = {
    system.activationScripts.extraActivation.text = ''
              # PAM settings
              echo >&2 "setting up pam..."
      ${mkSudoTouchIdAuthScript cfg.enableSudoTouchIdAuth}
    '';
  };
}
```

5 Editors

5.1 Emacs

5.1.1 Note: If you want a proper Emacs Config, look here:

https://tecosaur.github.io/emacs-config/config.html, this is just a compilation of different parts of his (and other's) configs, as well as a few parts I wrote by my own. I'm slowly working on making my config "mine"

1. Credit:

    - Tecosaur - For all his help and the excellent config
    - Dr. Elken - For his EXWM Module and help on the DOOM Server
    - Henrik - For making Doom Emacs in the first place

    Includes (snippets) of other software related under the MIT license:

    - *Doom Emacs Config, 2021 Tecosaur. https://tecosaur.github.io/emacs-config/config.html*
    - *.doom.d, 2021 Elken. https://github.com/elken/.doom.d/blob/master/config.org*

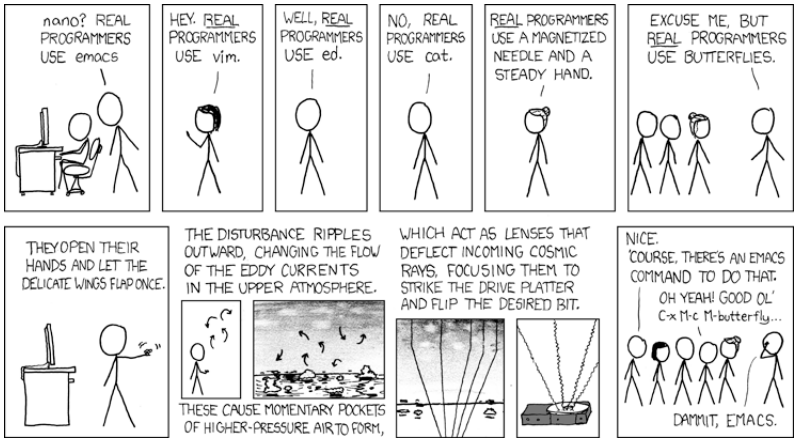
    Includes (snippets) of other software related under the GPLv3 license:

    - *.dotfiles, 2021 Daviwil. https://github.com/daviwil/dotfiles*

5.1.2 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 in neovim



Real Programmers Real programmers set the universal constants at the start such that the universe evolves to contain the disk with the data they want.

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

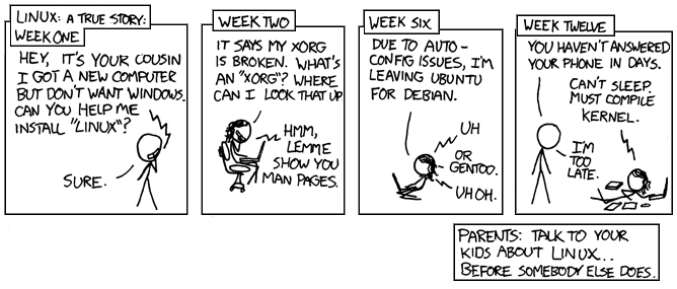
Separately, online I have seen the following statement enough times I think it's a catchphrase

> *Redditor 1: I just discovered this thing, isn't it cool.*
> *Redditor 2: Oh, there's an Emacs mode for that.*

This was enough for me to install Emacs, but there are many other reasons to keep using it.

I tried out the `spacemacs` distribution a bit, but it wasn't quite to my liking. Then I heard about `doom emacs` and thought I may as well give that a try.

With Org, I've discovered the wonders of literate programming, and with the help of others I've switched more and more to just using Emacs (just replace "Linux" with "Emacs" in the comic below).

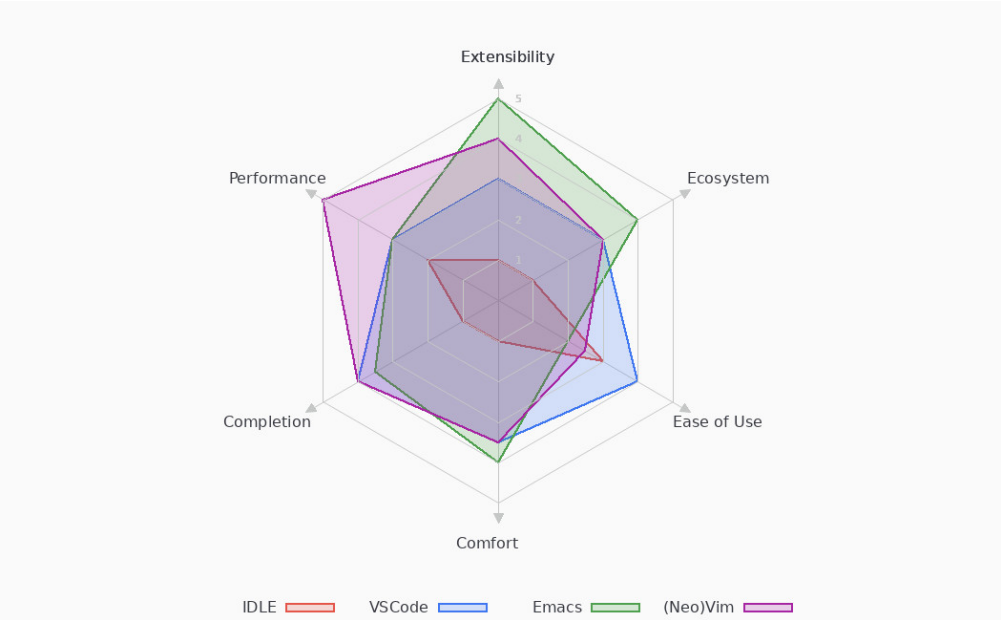


Cautionary This really is a true story, and she doesn't know I put it in my comic because her wifi hasn't worked for weeks.

Thats not to say using Emacs doesn't have its pitfalls. The performance leaves something to be

desired, but the benefits far outweigh the drawbacks. Its 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



1. Why Emacs? Emacs is not a text editor, this is a common misnomer. It is far more apt to describe Emacs as *a Lisp machine providing a generic user-centric text manipulation environment*. That's quite a mouthful. In simpler terms one can think of Emacs as a platform for text-related applications. It's a vague and generic definition because Emacs itself is generic.

   Good with text. How far does that go? A lot further than one initially thinks:

   - Task planning
   - File management
   - Terminal emulation

- Email client
- Remote server tool
- Git frontend
- Web client/server
- and more…

Ideally, one may use Emacs as *the* interface to perform `input → transform → output` cycles, i.e. form a bridge between the human mind and information manipulation.

(a) The enveloping editor Emacs allows one to do more in one place than any other application. Why is this good?

- Enables one to complete tasks with a consistent, standard set of keybindings, GUI and editing methods — learn once, use everywhere
- Reduced context-switching
- Compressing the stages of a project — a more centralised workflow can progress with greater ease
- Integration between tasks previously relegated to different applications, but with a common subject — e.g. linking to an email in a to-do list

Emacs can be thought of as a platform within which various elements of your workflow may settle, with the potential for rich integrations between them — a *life* IDE if you will.

Today, many aspects of daily computer usage are split between different applications which act like islands, but this often doesn't mirror how we *actually use* our computers. Emacs, if one goes down the rabbit hole, can give users the power to bridge this gap.

2. Notes for the unwary adventurer The lovely `doom doctor` is good at diagnosing most missing things, but here are a few extras.

- A LaTeX Compiler is required for the mathematics rendering performed in org, and that wonderful pdf/html export we have going. I recommend Tectonic.
- I use the Overpass font as a go-to sans serif. It's used as my `doom-variable-`

`pitch-font` I have chosen it because it possesses a few characteristics I consider desirable, namely:

  - A clean, and legible style. Highway-style fonts tend to be designed to be clear at a glance, and work well with a thicker weight, and this is inspired by *Highway Gothic*.
  - It's slightly quirky. Look at the diagonal cut on stems for example. Helvetica is a masterful design, but I like a bit more pizzazz now and then.
  - **Note:** Alegreya is used for my latex export and writeroom mode configurations

- I use my patched SFMono font as a go-to monospace. I have chosen it because it possesses a few characteristics I consider desirable, namely:
  - Elegent characters, and good ligatures/unicode support
  - It fits will with the rest of my system
- A few LSP servers. Take a look at init.el' to see which modules have the `+lsp` flag.
- Gnuplot, used for `org-plot`.
- A build of emacs with modules and xwidgets support. I also recommend the native-comp flag with emacs28.

5.1.3 Doom Configuration

1. Modules Doom has this lovely *modular configuration base* that takes a lot of work out of configuring Emacs. Each module (when enabled) can provide a list of packages to install (on `doom sync`) and configuration to be applied. The modules can also have flags applied to tweak their behaviour.

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

;; This file controls what Doom modules are enabled and what order they load in.
;; Press 'K' on a module to view its documentation, and 'gd' to browse its
↪  directory.

(doom! :completion
       <<doom-completion>>

       :ui
       <<doom-ui>>
```

```
       :editor
       <<doom-editor>>

       :emacs
       <<doom-emacs>>

       :term
       <<doom-term>>

       :checkers
       <<doom-checkers>>

       :tools
       <<doom-tools>>

       :os
       <<doom-os>>

       :lang
       <<doom-lang>>

       :email
       <<doom-email>>

       :app
       <<doom-app>>

       :config
       <<doom-config>>)
```

(a) Structure As you may have noticed by this point, this is a literate configuration. Doom has good support for this which we access though the `literate` module.

While we're in the `:config` section, we'll use Dooms nicer defaults, along with the bindings and smartparens behaviour (the flags aren't documented, but they exist).

```
literate
(default +bindings +smartparens)
```

(b) Interface There's a lot that can be done to enhance Emacs' capabilities. I reckon enabling half the modules Doom provides should do it.

```
(company                        ; the ultimate code completion backend
 +childframe)                   ; ... when your children are better than you
;;helm                         ; the *other* search engine for love and life
;;ido                          ; the other *other* search engine...
;;(ivy                         ; a search engine for love and life
;; +icons                      ; ... icons are nice
;; +prescient)                 ; ... I know what I want(ed)
```

```
(vertico +icons)             ; the search engine of the future
```

```
;;deft                       ; notational velocity for Emacs
doom                         ; what makes DOOM look the way it does
doom-dashboard               ; a nifty splash screen for Emacs
doom-quit                    ; DOOM quit-message prompts when you quit Emacs
(emoji +unicode)             ; 🙂
;;fill-column                ; a `fill-column' indicator
hl-todo                      ; highlight
↪  TODO/FIXME/NOTE/DEPRECATED/HACK/REVIEW
;;hydra                      ; quick documentation for related commands
;;indent-guides              ; highlighted indent columns, notoriously slow
(ligatures                   ; ligatures and symbols to make your code
↪  pretty again
 +extra)                     ; for those who dislike letters
minimap                      ; show a map of the code on the side
(modeline                    ; snazzy, Atom-inspired modeline, plus API
 +light)                     ; the doom modeline is a bit much, the default
 ↪  is a bit little
;;nav-flash                  ; blink the current line after jumping
;;neotree                    ; a project drawer, like NERDTree for vim
ophints                      ; highlight the region an operation acts on
(popup                       ; tame sudden yet inevitable temporary windows
 +all                        ; catch all popups that start with an asterix
 +defaults)                  ; default popup rules
;;(tabs                      ; an tab bar for Emacs
;;  +centaur-tabs)           ; ... with prettier tabs
treemacs                     ; a project drawer, like neotree but cooler
;;unicode                    ; extended unicode support for various languages
vc-gutter                    ; vcs diff in the fringe
;;vi-tilde-fringe            ; fringe tildes to mark beyond EOB
;;(window-select +numbers)   ; visually switch windows
workspaces                   ; tab emulation, persistence & separate
↪  workspaces
zen                          ; distraction-free coding or writing
```

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

```
(dired +icons)               ; making dired pretty [functional]
electric                     ; smarter, keyword-based electric-indent
(ibuffer +icons)             ; interactive buffer management
```

```
undo                        ; persistent, smarter undo for your inevitable
↪  mistakes
vc                          ; version-control and Emacs, sitting in a tree
```

```
;;eshell                   ; the elisp shell that works everywhere
;;shell                    ; simple shell REPL for Emacs
;;term                     ; basic terminal emulator for Emacs
vterm                       ; the best terminal emulation in Emacs
```

```
syntax                      ; tasing you for every semicolon you forget
(:if (executable-find "aspell") spell) ; tasing you for misspelling
↪  mispelling
;;grammar                  ; tasing grammar mistake every you make
```

```
;;ansible                  ; a crucible for infrastructure as code
;;biblio                   ; Writes a PhD for you (citation needed)
;;debugger                 ; FIXME stepping through code, to help you add
↪  bugs
;;direnv                   ; be direct about your environment
;;docker                   ; port everything to containers
editorconfig                ; let someone else argue about tabs vs spaces
;;ein                      ; tame Jupyter notebooks with emacs
(eval +overlay)             ; run code, run (also, repls)
;;gist                     ; interacting with github gists
(lookup                    ; helps you navigate your code and documentation
 +dictionary                ; dictionary/thesaurus is nice
 +docsets)                  ; ...or in Dash docsets locally
lsp                         ; Language Server Protocol
;;(:if IS-MAC macos)       ; MacOS-specific commands
magit                       ; a git porcelain for Emacs
 ;;+forge)                  ; interface with git forges
;;make                     ; run make tasks from Emacs
;;pass                     ; password manager for nerds
pdf                         ; pdf enhancements
;;prodigy                  ; FIXME managing external services & code
↪  builders
;;rgb                      ; creating color strings
;;taskrunner               ; taskrunner for all your projects
;;terraform                ; infrastructure as code
;;tmux                     ; an API for interacting with tmux
;;tree-sitter              ; ... sitting in a tree
;;upload                   ; map local to remote projects via ssh/ftp
```

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

(c) Language support We can be rather liberal with enabling support for languages as the associated packages/configuration are (usually) only loaded when first opening an associated file.

```
;;agda                          ; types of types of types of types...
;;beancount                     ; mind the GAAP
;;(cc +lsp)                     ; C/C++/Obj-C madness
;;clojure                       ; java with a lisp
;;common-lisp                   ; if you've seen one lisp, you've seen them all
;;coq                           ; proofs-as-programs
;;crystal                       ; ruby at the speed of c
;;csharp                        ; unity, .NET, and mono shenanigans
;;data                          ; config/data formats
;;(dart +flutter)               ; paint ui and not much else
;;dhall                         ; JSON with FP sprinkles
;;elixir                        ; erlang done right
;;elm                           ; care for a cup of TEA?
emacs-lisp                      ; drown in parentheses
;;erlang                        ; an elegant language for a more civilized age
;;ess                           ; emacs speaks statistics
;;faust                         ; dsp, but you get to keep your soul
;;fsharp                        ; ML stands for Microsoft's Language
;;fstar                         ; (dependent) types and (monadic) effects and Z3
;;gdscript                      ; the language you waited for
;;(go +lsp)                     ; the hipster dialect
;;(haskell +lsp)                ; a language that's lazier than I am
;;hy                            ; readability of scheme w/ speed of python
;;idris                         ;
;;json                          ; At least it ain't XML
;;(java +lsp)                   ; the poster child for carpal tunnel syndrome
;;(javascript +lsp)             ; all(hope(abandon(ye(who(enter(here))))))
;;(julia +lsp)                  ; Python, R, and MATLAB in a blender
;;(kotlin +lsp)                 ; a better, slicker Java(Script)
(latex                          ; writing papers in Emacs has never been so fun
;; +fold                        ; fold the clutter away nicities
 +latexmk                       ; modern latex plz
 +cdlatex                       ; quick maths symbols
 +lsp)
;;lean                          ; proof that mathematicians need help
;;factor                        ; for when scripts are stacked against you
;;ledger                        ; an accounting system in Emacs
(lua                            ; one-based indices? one-based indices
 +lsp)
(markdown +grip)                ; writing docs for people to ignore
;;nim                           ; python + lisp at the speed of c
nix                             ; I hereby declare "nix geht mehr!"
;;ocaml                         ; an objective camel
(org                            ; organize your plain life in plain text
 +pretty                        ; yessss my pretties! (nice unicode symbols)
 +dragndrop                     ; drag & drop files/images into org buffers
 ;;+hugo                       ; use Emacs for hugo blogging
 ;;+noter                      ; enhanced PDF notetaking
 +jupyter                       ; ipython/jupyter support for babel
 +pandoc                        ; export-with-pandoc support
 +gnuplot                       ; who doesn't like pretty pictures
 +pomodoro                      ; be fruitful with the tomato technique
 +present                       ; using org-mode for presentations
```

```
 +roam2)                        ; wander around notes
;;php                           ; perl's insecure younger brother
;;plantuml                      ; diagrams for confusing people more
;;purescript                    ; javascript, but functional
(python +lsp +pyright)          ; beautiful is better than ugly
;;qt                            ; the 'cutest' gui framework ever
;;racket                        ; a DSL for DSLs
;;raku                          ; the artist formerly known as perl6
;;rest                          ; Emacs as a REST client
;;rst                           ; ReST in peace
;;(ruby +rails)                 ; 1.step {|i| p "Ruby is #{i.even? ? 'love' :
↪  'life'}"}
(rust +lsp)                     ; Fe2O3.unwrap().unwrap().unwrap().unwrap()
;;scala                         ; java, but good
;;scheme                        ; a fully conniving family of lisps
sh                              ; she sells {ba,z,fi}sh shells on the C xor
;;sml                           ; no, the /other/ ML
;;solidity                      ; do you need a blockchain? No.
;;swift                         ; who asked for emoji variables?
;;terra                         ; Earth and Moon in alignment for performance.
;;web                           ; the tubes
;;yaml                          ; JSON, but readable
;;zig                           ; C, but simpler
```

(d) Everything in Emacs It's just too convenient being able to have everything in Emacs. I couldn't resist the Email and Feed modules.

```
(:if (executable-find "mu") (mu4e +org +gmail))
;;notmuch
;;(wanderlust +gmail)
```

```
;;calendar                      ; A dated approach to timetabling
;;emms                          ; Multimedia in Emacs is music to my ears
;;everywhere                    ; *leave* Emacs!? You must be joking.
;;irc                           ; how neckbeards socialize
;;(rss +org)                    ; emacs as an RSS reader
;;twitter                       ; twitter client https://twitter.com/vnought
```

2. Packages Unlike most literate configurations I ~~am lazy~~ like to keep all my packages in one place

```
;; -*- no-byte-compile: t; -*-
;;; $DOOMDIR/packages.el

;;org
<<org>>

;;latex
<<latex>>

;;markdown and html
```

```
<<web>>

;;looks
<<looks>>

;;emacs additions
<<emacs>>

;;lsp
<<lsp>>

;;fun
<<fun>>
```

(a) Org: The majority of my work in emacs is done in org mode, even this configuration was written in org! It makes sense that the majority of my packages are for tweaking org then

```
(package! org-appear)
(package! doct)
(package! org-roam-ui ;https://github.com/melpa/melpa/pull/7784
  :recipe (:host github
           :repo "org-roam/org-roam-ui"
           :files ("*.el" "out")))
(package! org-pandoc-import ;https://github.com/melpa/melpa/pull/7326
  :recipe (:host github
           :repo "tecosaur/org-pandoc-import"
           :files ("*.el" "filters" "preprocessors")))
;; org cite additions
(package! citar)
(package! citeproc)
(package! org-cite-csl-activate :recipe (:host github :repo
↪  "andras-simonyi/org-cite-csl-activate"))
```

(b) LaTeX: When I'm not working in org, I'm probably exporting it to latex. Lets adjust that a bit too

```
(package! org-fragtog)
(package! aas)
(package! laas)
(package! engrave-faces)
```

(c) Web: Sometimes I need to use markdown too. **Note:** emacs-webkit is temporarily disabled because of its refusal to work without requiring org

```
(package! ox-gfm)
(package! websocket)
;;(package! webkit
;;          :recipe (:host github
;;                   :repo "akirakyle/emacs-webkit"
```

```
;;                   :branch "main"
;;                   :files (:defaults "*.js" "*.css" "*.so" "*.nix")
;;                  :pre-build (("nix-shell" "shell.nix" "--command make"))))
```

(d) Looks: Making emacs look good is first priority, actually working in it is second

```
(unpin! doom-themes)
(unpin! doom-modeline)
(package! modus-themes)
(package! solaire-mode :disable t)
(package! ox-chameleon :recipe (:host github :repo
↪  "tecosaur/ox-chameleon")) ;soon :tm:
```

(e) Emacs Tweaks: Emacs is missing just a few packages that I need to make it my OS. Specifically, screenshot capabilities are nice, and using the same dictionaries accross ~~operating systems~~ bootloaders would be nice too!

```
(package! lexic)
(package! magit-delta)
(package! pdf-tools)
(package! screenshot :recipe (:host github :repo "Jimmysit0/screenshot"))
↪  ;https://github.com/melpa/melpa/pull/7327
```

(f) LSP: I like to live life on the edge

```
(unpin! lsp-ui)
(unpin! lsp-mode)
```

(g) Fun: We do a little trolling (and reading)

```
(package! nov)
(package! xkcd)
(package! keycast)
(package! selectric-mode :recipe (:local-repo "lisp/selectric-mode"))
```

5.1.4 Basic Configuration

Make this file run (slightly) faster with lexical binding

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

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

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

3. Emacsclient mu4e is a bit finicky with emacsclient, and org takes forever to load. The solution? Use tecosaurs greedy daemon startup

```
(defun greedily-do-daemon-setup ()
  (require 'org)
  (require 'vertico)
  (require 'consult)
  (require 'marginalia)
  (when (require 'mu4e nil t)
    (setq mu4e-confirm-quit t)
    (setq +mu4e-lock-greedy t)
    (setq +mu4e-lock-relaxed t)
    (+mu4e-lock-add-watcher)
    (when (+mu4e-lock-available t)
      (mu4e~start))))

(when (daemonp)
  (add-hook 'emacs-startup-hook #'greedily-do-daemon-setup)
  (add-hook 'emacs-startup-hook #'init-mixed-pitch-h))
```

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

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

   (a) 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`

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

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

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

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

      iii. 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)))))
```

I also want to hook Delta into Magit

```
(after! magit
   (magit-delta-mode +1))
```

iv. Ligatures Use ligatures from within vterm (and eshell), we do this by redefining the variable where *not* to show ligatures. On the other hand, in select modes we want to use extra ligatures, so lets enable that.

```
(setq +ligatures-in-modes t)
(setq +ligatures-extras-in-modes '(org-mode emacs-lisp-mode))
```

5. Fonts

Papyrus I secretly, deep in my guilty heart, like Papyrus and don't care if it's overused. [Cue hate mail in beautifully-kerned Helvetica.]

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 lets 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 Sans for a minimalist feel, so lets 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 Sans" :size 16
  ↪  :weight 'Medium))
  (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 serifed 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 ffl fft 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]))
```

(a) Font collections Using the lovely conditional preamble, I'll define a number of font collections that can be used for LaTeX exports. Who knows, maybe I'll use it with other export formats too at some point.

To start with I'll create a default state variable and register `fontset` as part of `#+options`.

```
(after! ox-latex
(defvar org-latex-default-fontset 'alegreya
```

```
  "Fontset from `org-latex-fontsets' to use by default.
      As cm (computer modern) is TeX's default, that causes nothing
      to be added to the document.
      If \"nil\" no custom fonts will ever be used.")
(eval '(cl-pushnew '(:latex-font-set nil "fontset"
↪  org-latex-default-fontset)
                   (org-export-backend-options (org-export-get-backend
                   ↪  'latex)))))
```

Then a function is needed to generate a LaTeX snippet which applies the fontset. It would be nice if this could be done for individual styles and use different styles as the main document font. If the individual typefaces for a fontset are defined individually as `:serif`, `:sans`, `:mono`, and `:maths`. I can use those to generate LaTeX for subsets of the full fontset. Then, if I don't let any fontset names have `-` in them, I can use `-sans` and `-mono` as suffixes that specify the document font to use.

```
(after! ox-latex
(defun org-latex-fontset-entry ()
  "Get the fontset spec of the current file.
      Has format \"name\" or \"name-style\" where 'name' is one of
      the cars in `org-latex-fontsets'."
  (let ((fontset-spec
         (symbol-name
          (or (car (delq nil
                         (mapcar
                          (lambda (opt-line)
                            (plist-get (org-export--parse-option-keyword
                            ↪  opt-line 'latex)
                                       :latex-font-set))
                          (cdar (org-collect-keywords '("OPTIONS"))))))
              org-latex-default-fontset))))
    (cons (intern (car (split-string fontset-spec "-")))
          (when (cadr (split-string fontset-spec "-"))
            (intern (concat ":" (cadr (split-string fontset-spec "-"))))))))

(defun org-latex-fontset (&rest desired-styles)
  "Generate a LaTeX preamble snippet which applies the current fontset for
  ↪  DESIRED-STYLES."
  (let* ((fontset-spec (org-latex-fontset-entry))
         (fontset (alist-get (car fontset-spec) org-latex-fontsets)))
    (if fontset
        (concat
         (mapconcat
          (lambda (style)
            (when (plist-get fontset style)
              (concat (plist-get fontset style) "\n")))
          desired-styles
          "")
         (when (memq (cdr fontset-spec) desired-styles)
           (pcase (cdr fontset-spec)
             (:serif "\\renewcommand{\\familydefault}{\\rmdefault}\n")
             (:sans "\\renewcommand{\\familydefault}{\\sfdefault}\n")
```

```
            (:mono "\\renewcommand{\\familydefault}{\\ttdefault}\n"))))
      (error "Font-set %s is not provided in org-latex-fontsets" (car
      ↪  fontset-spec))))))
```

Now that all the functionality has been implemented, we should hook it into our preamble generation.

```
(after! ox-latex
(add-to-list 'org-latex-conditional-features '(org-latex-default-fontset .
↪  custom-font) t)
(add-to-list 'org-latex-feature-implementations '(custom-font :snippet
↪  (org-latex-fontset :serif :sans :mono) :order 0) t)
(add-to-list 'org-latex-feature-implementations '(.custom-maths-font :eager
↪  t :when (custom-font maths) :snippet (org-latex-fontset :maths) :order
↪  0.3) t))
```

Finally, we just need to add some fonts.

```
(after! ox-latex
(defvar org-latex-fontsets
  '((cm nil) ; computer modern
    (## nil) ; no font set
    (alegreya
     :serif "\\usepackage[osf]{Alegreya}"
     :sans "\\usepackage{AlegreyaSans}"
     :mono "\\usepackage[scale=0.88]{sourcecodepro}"
     :maths "\\usepackage[varbb]{newpxmath}")
    (biolinum
     :serif "\\usepackage[osf]{libertineRoman}"
     :sans "\\usepackage[sfdefault,osf]{biolinum}"
     :mono "\\usepackage[scale=0.88]{sourcecodepro}"
     :maths "\\usepackage[libertine,varvw]{newtxmath}")
    (fira
     :sans "\\usepackage[sfdefault,scale=0.85]{FiraSans}"
     :mono "\\usepackage[scale=0.80]{FiraMono}"
     :maths "\\usepackage{newtxsf} % change to firamath in future?")
    (kp
     :serif "\\usepackage{kpfonts}")
    (newpx
     :serif "\\usepackage{newpxtext}"
     :sans "\\usepackage{gillius}"
     :mono "\\usepackage[scale=0.9]{sourcecodepro}"
     :maths "\\usepackage[varbb]{newpxmath}")
    (noto
     :serif "\\usepackage[osf]{noto-serif}"
     :sans "\\usepackage[osf]{noto-sans}"
     :mono "\\usepackage[scale=0.96]{noto-mono}"
     :maths "\\usepackage{notomath}")
    (plex
     :serif "\\usepackage{plex-serif}"
     :sans "\\usepackage{plex-sans}"
     :mono "\\usepackage[scale=0.95]{plex-mono}"
     :maths "\\usepackage{newtxmath}") ; may be plex-based in future
```

```
    (source
     :serif "\\usepackage[osf]{sourceserifpro}"
     :sans "\\usepackage[osf]{sourcesanspro}"
     :mono "\\usepackage[scale=0.95]{sourcecodepro}"
     :maths "\\usepackage{newtxmath}") ; may be sourceserifpro-based in
     ↪ future
    (times
     :serif "\\usepackage{newtxtext}"
     :maths "\\usepackage{newtxmath}"))
  "Alist of fontset specifications.
      Each car is the name of the fontset (which cannot include \"-\").
      Each cdr is a plist with (optional) keys :serif, :sans, :mono, and
      :maths.
      A key's value is a LaTeX snippet which loads such a font."))
```

When we're using Alegreya we can apply a lovely little tweak to `tabular` which (locally) changes the figures used to lining fixed-width.

```
(after! ox-latex
(add-to-list 'org-latex-conditional-features '((string= (car
↪  (org-latex-fontset-entry)) "alegreya") . alegreya-typeface))
(add-to-list 'org-latex-feature-implementations '(alegreya-typeface) t)
(add-to-list 'org-latex-feature-implementations'(.alegreya-tabular-figures
    :eager t :when (alegreya-typeface table) :order 0.5 :snippet "
    \\makeatletter
    % tabular lining figures in tables
    \\renewcommand{\\tabular}{\\AlegreyaTLF\\let\\@halignto\\@empty\\@tabular}
    \\makeatother\n") t))
```

Due to the Alegreya's metrics, the `\LaTeX` symbol doesn't quite look right. We can correct for this by redefining it with subtlety shifted kerning.

```
(after! ox-latex
(add-to-list 'org-latex-conditional-features '("LaTeX" . latex-symbol))
(add-to-list 'org-latex-feature-implementations '(latex-symbol :when
    alegreya-typeface :order 0.5 :snippet "
    \\makeatletter
    % Kerning around the A needs adjusting
    \\DeclareRobustCommand{\\LaTeX}{L\\kern-.24em%
            {\\sbox\\z@ T%
             \\vbox to\\ht\\z@{\\hbox{\\check@mathfonts
                                    \\fontsize\\sf@size\\z@
                                    \\math@fontsfalse\\selectfont
                                    A}%
                               \\vss}%
            }%
            \\kern-.10em%
            \\TeX}
    \\makeatother\n") t))
```

Just in case the fonts aren't there, lets add check to notify the user of the issue. Seems like I forget ot install fonts every time I switch between ~~distros~~ emacs bootloaders

```
(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"
                                  (propertize "Warning!" 'face '(bold warning))
                                     (mapconcat (lambda (font)
                                             (propertize font 'face 'font-lock-
                                                   ↪  variable-name-face))
                                                ',missing-fonts
                                                ", "))
                         (sleep-for 0.5))))))
  ";; No missing fonts detected")
```

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

6. Themes Right now I'm using nord, but I use doom-vibrant sometimes

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

   (a) Modus Themes Generally I use doom-themes, but I also like the new Modus-themes bundled with emacs28/29

```
;; (use-package modus-themes
;;   :init
;;   ;; Add all your customizations prior to loading the themes
;;   (setq modus-themes-italic-constructs t
;;         modus-themes-completions 'opinionated
;;         modus-themes-variable-pitch-headings t
;;         modus-themes-scale-headings t
;;         modus-themes-variable-pitch-ui nil
;;         modus-themes-org-agenda
```

```
;;         '((header-block . (variable-pitch scale-title))
;;           (header-date . (grayscale bold-all)))
;;         modus-themes-org-blocks
;;         '(grayscale)
;;         modus-themes-mode-line
;;         '(borderless)
;;         modus-themes-region '(bg-only no-extend))

;;   ;; Load the theme files before enabling a theme
;;   (modus-themes-load-themes)
;;   :config
;;   (modus-themes-load-vivendi)
;;  :bind ("<f5>" . modus-themes-toggle))
```

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-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 'yasnippet 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)
    (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 templete

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

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))
```

The rust language server also has some extra features I would like to enable

```
(after! lsp-rust
  (setq lsp-rust-server 'rust-analyzer
  lsp-rust-analyzer-display-chaining-hints t
  lsp-rust-analyzer-display-parameter-hints t
  lsp-rust-analyzer-server-display-inlay-hints t
  lsp-rust-analyzer-cargo-watch-command "clippy"
  rustic-format-on-save t))
```

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
;; (tool-bar-mode +1) ;;re-enable the toolbar
;; (setq tool-bar-style 'text)
```

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-vsplit-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-vsplit)
  (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 fine ; more intuitive than : for entering command mode

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

When im doing regexes, its usually with /g anyways, lets make that the default

```
(after! evil
  (setq evil-ex-substitute-global t     ; I like my s/../.. to by 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. Lets 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. Lets 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, both inside and outside, and lets make the line spacing comfier

```
;; Make a clean & minimalist frame
(use-package frame
  :config
  (setq-default default-frame-alist
                (append (list
                '(internal-border-width . 24)
                '(left-fringe    . 0)
                '(right-fringe   . 0)
                '(tool-bar-lines . 0)
                '(menu-bar-lines . 0)
                '(line-spacing . 0.35)
                '(vertical-scroll-bars . nil))))
  (setq-default window-resize-pixelwise t)
  (setq-default frame-resize-pixelwise t)
  :custom
  (window-divider-default-right-width 24)
  (window-divider-default-bottom-width 12)
  (window-divider-default-places 'right-only)
  (window-divider-mode t))

;; Make sure new frames use window-divider
(add-hook 'before-make-frame-hook 'window-divider-mode)
```

10. ~~Selectric~~ NK-Creams mode Instead of using the regular selectric-mode, I modified it with a few notable tweaks, mainly:

    (a) Support for EVIL mode

    (b) It uses NK Cream sounds instead of the typewritter ones

    The samples used here are taken from monketype, but heres a similar board youtube

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

5.1.5 Visual configuration

1. Treesitter Nvim-treesitter is based on three interlocking features: language parsers, queries, and modules, where modules provide features – e.g., highlighting – based on queries for syntax objects extracted from a given buffer by language parsers. Allowing this to work in doom will reduce the lag introduced by fontlock as well as improve textobjects.

Since I use an apple silicon mac, I prefer if nix handles compiling the parsers for me

```
;; (use-package! tree-sitter
;;   :config
;;   (cl-pushnew (expand-file-name "~/.config/tree-sitter")
↪  tree-sitter-load-path)
;;   (require 'tree-sitter-langs)
;;   (global-tree-sitter-mode)
;;   (add-hook 'tree-sitter-after-on-hook #'tree-sitter-hl-mode))
```

2. Modeline 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-enable-word-count t))             ;Show word count
```

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. 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 theres 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))
```

4. 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)))))
```

5. Treemacs Lets theme treemacs while we're at it

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

6. Emojis Disable some annoying emojis

```
(defvar emojify-disabled-emojis
  '(;; Org
    "□" "☑" "□" "□" "□" "□" "□" "□" "□"
    ;; Terminal powerline
    "□"
    ;; Box drawing
    "▶" "◀")
  "Characters that should never be affected by `emojify-mode'.")

(defadvice! emojify-delete-from-data ()
  "Ensure `emojify-disabled-emojis' don't appear in `emojify-emojis'."
  :after #'emojify-set-emoji-data
  (dolist (emoji emojify-disabled-emojis)
    (remhash emoji emojify-emojis)))

(add-hook! '(mu4e-compose-mode org-msg-edit-mode) (emoticon-to-emoji 1))
```

7. 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))))
    (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.")

(defvar 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))

(defvar 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))
```

8. 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
- 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))))))
```

9. 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 customizations 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)
```

I dislike that end/begin statements in org mode are the same color as the background. I've changed them to use a darker color

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

(a) 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[{]+\\)[{[]?" 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)))
        (add-text-properties start end `(prettify-symbols-start ,start
        ↪  prettify-symbols-end ,end))))))

(defun org-fontify-inline-src-blocks-enable ()
  "Add inline src fontification to font-lock in Org.
     Must be run as part of `org-font-lock-set-keywords-hook'."
  (setq org-font-lock-extra-keywords
        (append org-font-lock-extra-keywords
        ↪  '((org-fontify-inline-src-blocks)))))
```

```
(add-hook 'org-font-lock-set-keywords-hook
↪  #'org-fontify-inline-src-blocks-enable)
```

10. Symbols Firstly, I dislike the default stars for org-mode, so lets improve that

```
;;make bullets look better
(after! org-superstar
  (setq org-superstar-headline-bullets-list '("◉" "○" "□" "□" "□" "□" "◆" "▶")
        org-superstar-prettify-item-bullets t ))
```

I also want to hide leading stars, since they feel redundant

```
(setq org-ellipsis " ▾ "
      org-hide-leading-stars t
      org-priority-highest ?A
      org-priority-lowest ?E
      org-priority-faces
      '((?A . 'all-the-icons-red)
        (?B . 'all-the-icons-orange)
        (?C . 'all-the-icons-yellow)
        (?D . 'all-the-icons-green)
        (?E . 'all-the-icons-blue)))
```

Lastly, lets add some ligatures for some org mode stuff

```
(appendq! +ligatures-extra-symbols
          `(:checkbox      "☐"
            :pending       "□"
            :checkedbox    "☑"
            :list_property "::"
            :em_dash       "—"
            :ellipses      "…"
            :arrow_right   "→"
            :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   ,(propertize "□" 'face 'all-the-icons-red)
            :priority_b   ,(propertize "□" 'face 'all-the-icons-orange)
            :priority_c   ,(propertize "■" 'face 'all-the-icons-yellow)
            :priority_d   ,(propertize "□" 'face 'all-the-icons-green)
            :priority_e   ,(propertize "□" '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))))
```

11. 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)))
```

12. 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)
```

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 (elfeed-clamp (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)
                        :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)))))))
```

14. Ebooks

To actually read the ebooks we use nov.

Kindle I'm happy with my Kindle 2 so far, but if they cut off the free Wikipedia browsing, I plan to show up drunk on Jeff Bezos's lawn and refuse to leave.

```
(use-package! nov
  :mode ("\\.epub\\'" . nov-mode)
  :config
  (map! :map nov-mode-map
        :n "RET" #'nov-scroll-up)

  (advice-add 'nov-render-title :override #'ignore)
  (defun +nov-mode-setup ()
    (face-remap-add-relative 'variable-pitch
                             :family "Overpass"
                             :height 1.4
                             :width 'semi-expanded)
    (face-remap-add-relative 'default :height 1.3)
    (setq-local line-spacing 0.2
                next-screen-context-lines 4
                shr-use-colors nil)
    (require 'visual-fill-column nil t)
    (setq-local visual-fill-column-center-text t
                visual-fill-column-width 81
                nov-text-width 80)
    (visual-fill-column-mode 1)
    (add-to-list '+lookup-definition-functions #'+lookup/dictionary-definition)
    (add-hook 'nov-mode-hook #'+nov-mode-setup)))
```

15. Screenshot Testing

```
(use-package! screenshot
  :defer t)
```

5.1.6 Org

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

   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)))
```

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 accidently 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")))
```

I also want to change the order of bullets

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

The `[[yt:...]]` links preview nicely, but don't export nicely. Thankfully, we can fix that.

```
(after! ox
(org-link-set-parameters "yt" :export #'+org-export-yt)
(defun +org-export-yt (path desc backend _com)
  (cond ((org-export-derived-backend-p backend 'html)
         (format "<iframe width='440' \
             height='335' \
             src='https://www.youtube.com/embed/%s' \
             frameborder='0' \
             allowfullscreen>%s</iframe>" path (or "" desc)))
        ((org-export-derived-backend-p backend 'latex)
         (format "\\href{https://youtu.be/%s}{%s}" path (or desc "youtube")))
```

```
      (t (format "https://youtu.be/%s" path)))))
```

(a) 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! ox-html
  (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-collapsable'
       - `org-html-block-collapsable'
       - `org-html-table-wrapped'
       - `org-html--format-toc-headline-colapseable'
       - `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-html5-p info)) (org-html-xhtml-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-xhtml-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%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-klipse-selection-script)
               "</script><script src=\""
               org-html-klipse-js
               "\"></script><link rel=\"stylesheet\" type=\"text/css\"
               ↪  href=\""
               org-html-klipse-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
                    (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 (cadar (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-collapsable (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='copyPreToClipbord(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-colapseable (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;" . "&#8594;")
          '("&lt;-" . "&#8592;"))

(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
                                   (cdaddr
                                    (with-current-buffer (progn (message
                                    ↪  "Fetching metadata from %s" url)
                                                (url-retrieve-synchronously
                                                ↪  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\",
              multlineWidth: \"%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-collapsable (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='copyPreToClipbord(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-collapsable)
(advice-add 'org-html-fixed-width     :around #'org-html-block-collapsable)
(advice-add 'org-html-property-drawer :around #'org-html-block-collapsable)
```

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
  :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]\\{2\\}\\)[0-
       ↪  9]*-"
       "☒(\\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))
```

3. Org-Agenda Set the directory

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

4. Org-Capture Use doct

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

(a) 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 (propertize prefix 'face
                        ↪  'font-lock-comment-face) (propertize k 'face
                        ↪  'bold) (propertize "›" 'face
                        ↪  'font-lock-comment-face) "  " desc "…" "\n")))
                    ;; Usable entry.
                    (`(,(and key (pred (string-match re))) ,desc . ,_)
                     (let ((k (match-string 1 key)))
                       (insert (propertize prefix 'face
                       ↪  'font-lock-comment-face) (propertize 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" (propertize 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)
(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))
```

(b) 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)))
              )))
```

5. Org-Cite

```
(use-package! citar
  :when (featurep! :completion vertico))

(use-package! citeproc
  :defer t)

;;; Org-Cite configuration
(map! :after org
      :map org-mode-map
      :localleader
      :desc "Insert citation" "@" #'org-cite-insert)

(use-package! oc
  :after org citar
  :config
  (require 'ox)
  (setq org-cite-global-bibliography
        (let ((paths (or citar-bibliography
                         (bound-and-true-p bibtex-completion-bibliography))))
          ;; Always return bibliography paths as list for org-cite.
          (if (stringp paths) (list paths) paths)))
  ;; setup export processor; default csl/citeproc-el, with biblatex for latex
  (setq org-cite-export-processors
        '((t csl))))

;;; Org-cite processors
(use-package! oc-biblatex
  :after oc)

(use-package! oc-csl
```

```
  :after oc
  :config
  (setq org-cite-csl-styles-dir "~/.config/bib/styles"))

(use-package! oc-natbib
  :after oc)

;;;; Third-party
(use-package! citar-org
  :no-require
  :custom
  (org-cite-insert-processor 'citar)
  (org-cite-follow-processor 'citar)
  (org-cite-activate-processor 'citar)
  (org-support-shift-select t)
  (citar-bibliography '("~/org/references.bib"))
  (when (featurep! :lang org +roam2)
    ;; Include property drawer metadata for 'org-roam' v2.
    (citar-org-note-include '(org-id org-roam-ref)))
  ;; Personal extras
  (setq citar-symbols
        `((file ,(all-the-icons-faicon "file-o" :v-adjust -0.1) . " ")
          (note ,(all-the-icons-material "speaker_notes" :face
          ↪  'all-the-icons-silver :v-adjust -0.3) . " ")
          (link ,(all-the-icons-octicon "link" :face 'all-the-icons-dsilver
          ↪  :v-adjust 0.01) . " "))))

(use-package! oc-csl-activate
  :after oc
  :config
  (setq org-cite-csl-activate-use-document-style t)
  (defun +org-cite-csl-activate/enable ()
    (interactive)
    (setq org-cite-activate-processor 'csl-activate)
    (add-hook! 'org-mode-hook '((lambda () (cursor-sensor-mode 1))
    ↪  org-cite-csl-activate-render-all))
    (defadvice! +org-cite-csl-activate-render-all-silent (orig-fn)
      :around #'org-cite-csl-activate-render-all
      (with-silent-modifications (funcall orig-fn)))
    (when (eq major-mode 'org-mode)
      (with-silent-modifications
        (save-excursion
          (goto-char (point-min))
          (org-cite-activate (point-max)))
        (org-cite-csl-activate-render-all)))
    (fmakunbound #'+org-cite-csl-activate/enable)))
```

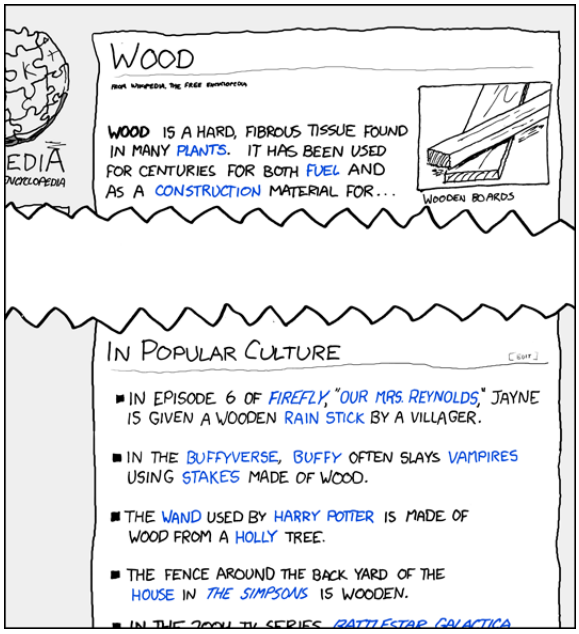
6. Org-Plot

   You can't ever have too many graphs! Lets make it look prettier, and 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
        fgal = \"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))
```

7. XKCD



In Popular Culture Someday the 'in popular culture' section will have its own article with an 'in popular culture' section. It will reference this title-text referencing it, and the blogosphere will implode.

Relevent XKCD:

I link to xkcd's so much that its better to just have a configuration for them We want to set this up so it loads nicely in org.

```
(use-package! xkcd
  :commands (xkcd-get-json
             xkcd-download xkcd-get
             ;; now for funcs from my extension of this pkg
             +xkcd-find-and-copy +xkcd-find-and-view
             +xkcd-fetch-info +xkcd-select)
  :config
  (setq xkcd-cache-dir (expand-file-name "xkcd/" doom-cache-dir)
        xkcd-cache-latest (concat xkcd-cache-dir "latest"))
  (unless (file-exists-p xkcd-cache-dir)
    (make-directory xkcd-cache-dir))
  (after! evil-snipe
    (add-to-list 'evil-snipe-disabled-modes 'xkcd-mode))
  :general (:states 'normal
            :keymaps 'xkcd-mode-map
            "<right>" #'xkcd-next
            "n"       #'xkcd-next ; evil-ish
            "<left>"  #'xkcd-prev
            "N"       #'xkcd-prev ; evil-ish
            "r"       #'xkcd-rand
            "a"       #'xkcd-rand ; because image-rotate can interfere
            "t"       #'xkcd-alt-text
            "q"       #'xkcd-kill-buffer
            "o"       #'xkcd-open-browser
            "e"       #'xkcd-open-explanation-browser
            ;; extras
            "s"       #'+xkcd-find-and-view
            "/"       #'+xkcd-find-and-view
            "y"       #'+xkcd-copy))
```

Let's also extend the functionality a whole bunch.

```
(after! xkcd
  (require 'emacsql-sqlite)

  (defun +xkcd-select ()
    "Prompt the user for an xkcd using `completing-read' and
    ↪  `+xkcd-select-format'. Return the xkcd number or nil"
    (let* (prompt-lines
           (-dummy (maphash (lambda (key xkcd-info)
                              (push (+xkcd-select-format xkcd-info) prompt-lines))
                            +xkcd-stored-info))
           (num (completing-read (format "xkcd (%s): " xkcd-latest)
           ↪  prompt-lines)))
      (if (equal "" num) xkcd-latest
        (string-to-number (replace-regexp-in-string "\\([0-9]+\\).*" "\\1"
        ↪  num)))))

  (defun +xkcd-select-format (xkcd-info)
    "Creates each completing-read line from an xkcd info plist. Must start with
    ↪  the xkcd number"
    (format "%-4s  %-30s %s"
            (propertize (number-to-string (plist-get xkcd-info :num))
```

```
                        'face 'counsel-key-binding)
          (plist-get xkcd-info :title)
          (propertize (plist-get xkcd-info :alt)
                      'face '(variable-pitch font-lock-comment-face))))

(defun +xkcd-fetch-info (&optional num)
  "Fetch the parsed json info for comic NUM. Fetches latest when omitted or 0"
  (require 'xkcd)
  (when (or (not num) (= num 0))
    (+xkcd-check-latest)
    (setq num xkcd-latest))
  (let ((res (or (gethash num +xkcd-stored-info)
                 (puthash num (+xkcd-db-read num) +xkcd-stored-info))))
    (unless res
      (+xkcd-db-write
       (let* ((url (format "https://xkcd.com/%d/info.0.json" num))
              (json-assoc
               (if (gethash num +xkcd-stored-info)
                   (gethash num +xkcd-stored-info)
                 (json-read-from-string (xkcd-get-json url num)))))
         json-assoc))
      (setq res (+xkcd-db-read num)))
    res))

;; since we've done this, we may as well go one little step further
(defun +xkcd-find-and-copy ()
  "Prompt for an xkcd using `+xkcd-select' and copy url to clipboard"
  (interactive)
  (+xkcd-copy (+xkcd-select)))

(defun +xkcd-copy (&optional num)
  "Copy a url to xkcd NUM to the clipboard"
  (interactive "i")
  (let ((num (or num xkcd-cur)))
    (gui-select-text (format "https://xkcd.com/%d" num))
    (message "xkcd.com/%d copied to clipboard" num)))

(defun +xkcd-find-and-view ()
  "Prompt for an xkcd using `+xkcd-select' and view it"
  (interactive)
  (xkcd-get (+xkcd-select))
  (switch-to-buffer "*xkcd*"))

(defvar +xkcd-latest-max-age (* 60 60) ; 1 hour
  "Time after which xkcd-latest should be refreshed, in seconds")

;; initialise `xkcd-latest' and `+xkcd-stored-info' with latest xkcd
(add-transient-hook! '+xkcd-select
  (require 'xkcd)
  (+xkcd-fetch-info xkcd-latest)
  (setq +xkcd-stored-info (+xkcd-db-read-all)))

(add-transient-hook! '+xkcd-fetch-info
```

```
  (xkcd-update-latest))

(defun +xkcd-check-latest ()
  "Use value in `xkcd-cache-latest' as long as it isn't older thabn
  ↪  `+xkcd-latest-max-age'"
  (unless (and (file-exists-p xkcd-cache-latest)
               (< (- (time-to-seconds (current-time))
                     (time-to-seconds (file-attribute-modification-time
                     ↪  (file-attributes xkcd-cache-latest))))
                  +xkcd-latest-max-age))
    (let* ((out (xkcd-get-json "http://xkcd.com/info.0.json" 0))
           (json-assoc (json-read-from-string out))
           (latest (cdr (assoc 'num json-assoc))))
      (when (/= xkcd-latest latest)
        (+xkcd-db-write json-assoc)
        (with-current-buffer (find-file xkcd-cache-latest)
          (setq xkcd-latest latest)
          (erase-buffer)
          (insert (number-to-string latest))
          (save-buffer)
          (kill-buffer (current-buffer)))))
    (shell-command (format "touch %s" xkcd-cache-latest))))

(defvar +xkcd-stored-info (make-hash-table :test 'eql)
  "Basic info on downloaded xkcds, in the form of a hashtable")

(defadvice! xkcd-get-json--and-cache (url &optional num)
  "Fetch the Json coming from URL.
      If the file NUM.json exists, use it instead.
      If NUM is 0, always download from URL.
      The return value is a string."
  :override #'xkcd-get-json
  (let* ((file (format "%s%d.json" xkcd-cache-dir num))
         (cached (and (file-exists-p file) (not (eq num 0))))
         (out (with-current-buffer (if cached
                                       (find-file file)
                                     (url-retrieve-synchronously url))
                (goto-char (point-min))
                (unless cached (re-search-forward "^$"))
                (prog1
                    (buffer-substring-no-properties (point) (point-max))
                  (kill-buffer (current-buffer))))))
    (unless (or cached (eq num 0))
      (xkcd-cache-json num out))
    out))

(defadvice! +xkcd-get (num)
  "Get the xkcd number NUM."
  :override 'xkcd-get
  (interactive "nEnter comic number: ")
  (xkcd-update-latest)
  (get-buffer-create "*xkcd*")
  (switch-to-buffer "*xkcd*")
```

```
  (xkcd-mode)
  (let (buffer-read-only)
    (erase-buffer)
    (setq xkcd-cur num)
    (let* ((xkcd-data (+xkcd-fetch-info num))
           (num (plist-get xkcd-data :num))
           (img (plist-get xkcd-data :img))
           (safe-title (plist-get xkcd-data :safe-title))
           (alt (plist-get xkcd-data :alt))
           title file)
      (message "Getting comic...")
      (setq file (xkcd-download img num))
      (setq title (format "%d: %s" num safe-title))
      (insert (propertize title
                          'face 'outline-1))
      (center-line)
      (insert "\n")
      (xkcd-insert-image file num)
      (if (eq xkcd-cur 0)
          (setq xkcd-cur num))
      (setq xkcd-alt alt)
      (message "%s" title))))

(defconst +xkcd-db--sqlite-available-p
  (with-demoted-errors "+org-xkcd initialization: %S"
    (emacsql-sqlite-ensure-binary)
    t))

(defvar +xkcd-db--connection (make-hash-table :test #'equal)
  "Database connection to +org-xkcd database.")

(defun +xkcd-db--get ()
  "Return the sqlite db file."
  (expand-file-name "xkcd.db" xkcd-cache-dir))

(defun +xkcd-db--get-connection ()
  "Return the database connection, if any."
  (gethash (file-truename xkcd-cache-dir)
           +xkcd-db--connection))

(defconst +xkcd-db--table-schema
  '((xkcds
     [(num integer :unique :primary-key)
      (year        :not-null)
      (month       :not-null)
      (link        :not-null)
      (news        :not-null)
      (safe_title  :not-null)
      (title       :not-null)
      (transcript  :not-null)
      (alt         :not-null)
      (img         :not-null)])))
```

```
(defun +xkcd-db--init (db)
  "Initialize database DB with the correct schema and user version."
  (emacsql-with-transaction db
    (pcase-dolist (`(,table . ,schema) +xkcd-db--table-schema)
      (emacsql db [:create-table $i1 $S2] table schema))))

(defun +xkcd-db ()
  "Entrypoint to the +org-xkcd sqlite database.
     Initializes and stores the database, and the database connection.
     Performs a database upgrade when required."
  (unless (and (+xkcd-db--get-connection)
               (emacsql-live-p (+xkcd-db--get-connection)))
    (let* ((db-file (+xkcd-db--get))
           (init-db (not (file-exists-p db-file))))
      (make-directory (file-name-directory db-file) t)
      (let ((conn (emacsql-sqlite db-file)))
        (set-process-query-on-exit-flag (emacsql-process conn) nil)
        (puthash (file-truename xkcd-cache-dir)
                 conn
                 +xkcd-db--connection)
        (when init-db
          (+xkcd-db--init conn)))))
  (+xkcd-db--get-connection))

(defun +xkcd-db-query (sql &rest args)
  "Run SQL query on +org-xkcd database with ARGS.
     SQL can be either the emacsql vector representation, or a string."
  (if  (stringp sql)
      (emacsql (+xkcd-db) (apply #'format sql args))
    (apply #'emacsql (+xkcd-db) sql args)))

(defun +xkcd-db-read (num)
  (when-let ((res
              (car (+xkcd-db-query [:select * :from xkcds
                                    :where (= num $s1)]
                                   num
                                   :limit 1))))
    (+xkcd-db-list-to-plist res)))

(defun +xkcd-db-read-all ()
  (let ((xkcd-table (make-hash-table :test 'eql :size 4000)))
    (mapcar (lambda (xkcd-info-list)
              (puthash (car xkcd-info-list) (+xkcd-db-list-to-plist
              ↪  xkcd-info-list) xkcd-table))
            (+xkcd-db-query [:select * :from xkcds]))
    xkcd-table))

(defun +xkcd-db-list-to-plist (xkcd-datalist)
  `(:num ,(nth 0 xkcd-datalist)
    :year ,(nth 1 xkcd-datalist)
    :month ,(nth 2 xkcd-datalist)
    :link ,(nth 3 xkcd-datalist)
    :news ,(nth 4 xkcd-datalist)
```

```
      :safe-title ,(nth 5 xkcd-datalist)
      :title ,(nth 6 xkcd-datalist)
      :transcript ,(nth 7 xkcd-datalist)
      :alt ,(nth 8 xkcd-datalist)
      :img ,(nth 9 xkcd-datalist)))

  (defun +xkcd-db-write (data)
    (+xkcd-db-query [:insert-into xkcds
                     :values $v1]
                    (list (vector
                           (cdr (assoc 'num        data))
                           (cdr (assoc 'year       data))
                           (cdr (assoc 'month      data))
                           (cdr (assoc 'link       data))
                           (cdr (assoc 'news       data))
                           (cdr (assoc 'safe_title data))
                           (cdr (assoc 'title      data))
                           (cdr (assoc 'transcript data))
                           (cdr (assoc 'alt        data))
                           (cdr (assoc 'img        data))
                          )))))
```

Now to just have this register with org

```
(after! org
  (org-link-set-parameters "xkcd"
                           :image-data-fun #'+org-xkcd-image-fn
                           :follow #'+org-xkcd-open-fn
                           :export #'+org-xkcd-export
                           :complete #'+org-xkcd-complete)

  (defun +org-xkcd-open-fn (link)
    (+org-xkcd-image-fn nil link nil))

  (defun +org-xkcd-image-fn (protocol link description)
    "Get image data for xkcd num LINK"
    (let* ((xkcd-info (+xkcd-fetch-info (string-to-number link)))
           (img (plist-get xkcd-info :img))
           (alt (plist-get xkcd-info :alt)))
      (message alt)
      (+org-image-file-data-fn protocol (xkcd-download img (string-to-number
      ↪  link)) description)))

  (defun +org-xkcd-export (num desc backend _com)
    "Convert xkcd to html/LaTeX form"
    (let* ((xkcd-info (+xkcd-fetch-info (string-to-number num)))
           (img (plist-get xkcd-info :img))
           (alt (plist-get xkcd-info :alt))
           (title (plist-get xkcd-info :title))
           (file (xkcd-download img (string-to-number num))))
      (cond ((org-export-derived-backend-p backend 'html)
             (format "<img class='invertible' src='%s' title=\"%s\" alt='%s'>"
             ↪  img (subst-char-in-string ?\" ?“ alt) title))
```

```
            ((org-export-derived-backend-p backend 'latex)
             (format "\\begin{figure}[!htb]
                     \\centering
                     \\includegraphics[scale=0.4]{%s}%s
                   \\end{figure}" file (if (equal desc (format "xkcd:%s" num)) ""
                        (format "\n  \\caption*{\\label{xkcd:%s} %s}"
                                num
                                (or desc
                                    (format "\\textbf{%s} %s" title alt))))))
            (t (format "https://xkcd.com/%s" num)))))

  (defun +org-xkcd-complete (&optional arg)
    "Complete xkcd using `+xkcd-stored-info'"
    (format "xkcd:%d" (+xkcd-select))))
```

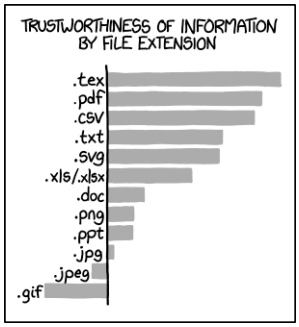
8. View Exported File I have to export files pretty often, lets setup some keybindings to make it easier

```
(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.")
```

9. 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))
```

5.1.7 Latex



File Extensions I have never been lied to by data in a .txt file which has been hand-aligned.

I have a love-hate relationship with latex. Its extremely powerful, but at the same time its hard to write, hard to understand, and very slow. The solution: write everything in org and then export it to tex. Best of both worlds!

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")
```

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}
    ")
```

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))
```

3. Export

   (a) Conditional features

```
(defvar org-latex-italic-quotes t
  "Make \"quote\" environments italic.")
(defvar org-latex-par-sep t
  "Vertically seperate paragraphs, and remove indentation.")

(defvar org-latex-conditional-features

  '(("\\[\\[\\(?:file\\|https?\\):\\(?:[^]]\\|\\\\\\]\\)+?\\.\\(?:eps\\|pdf\\|png\\|jpeg\\|
  . image)
   ("\\[\\[\\(?:file\\|https?\\):\\(?:[^]]+?\\|\\\\\\]\\)\\.svg\\]\\]" .
   svg)
   ("^[ \t]*|" . table)
   ("cref:\\|\\cref{\\|\\[\\[[^\\]]+\\]\\]" . cleveref)
   ("[;\\\\]?\\b[A-Z][A-Z]+s?[^A-Za-z]" . acronym)
   ("\\+[^ ].*[^ ]\\+\\|_[^ ].*[^
   ]_\\|\\\\uu?line\\|\\\\uwave\\|\\\\sout\\|\\\\xout\\|\\\\dashuline\\|\\dotuline\\|\\mar
   . underline)
```

```
    (":float wrap" . float-wrap)
    (":float sideways" . rotate)
    ("^[ \t]*#\\+caption:\\|\\\\caption" . caption)
    ("\\[\\[xkcd:" . (image caption))
    ((and org-latex-italic-quotes "^[
    ↪  \t]*#\\+begin_quote\\|\\\\begin{quote}") . italic-quotes)
    (org-latex-par-sep . par-sep)
    ("^[ \t]*\\(?:[-+*]\\|[0-9]+[.)]\\|[A-Za-z]+[.)]\\) \\[[ -X]\\]" .
    ↪  checkbox)
    ("^[ \t]*#\\+begin_warning\\|\\\\begin{warning}" . box-warning)
    ("^[ \t]*#\\+begin_info\\|\\\\begin{info}"       . box-info)
    ("^[ \t]*#\\+begin_success\\|\\\\begin{success}" . box-success)
    ("^[ \t]*#\\+begin_error\\|\\\\begin{error}"     . box-error))
  "Org feature tests and associated LaTeX feature flags.
     Alist where the car is a test for the presense of the feature,
     and the cdr is either a single feature symbol or list of feature
     symbols.
     When a string, it is used as a regex search in the buffer.
     The feature is registered as present when there is a match.
     The car can also be a
     - symbol, the value of which is fetched
     - function, which is called with info as an argument
     - list, which is `eval'uated
     If the symbol, function, or list produces a string: that is used as a
     regex
     search in the buffer. Otherwise any non-nil return value will
     indicate the
     existance of the feature.")
```

```
(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\\textbf
    -}}$\\square$}
    \\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} \\text-
    color{#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 "\\renewcom-
    ↪  mand{\\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"))))
```

(b) Embed Externally Linked Images I don't like to keep images downloaded to my laptop, it clutters up everything. Org has a handy feature where you can pass a link instead, and org will display it inline as usual.

HTML export handles this use case just fine, if the image isn't named then it will display the image. However, latex doesn't have support for this. What we do is instead of linking the image, we can have emacs download the linked image and export that!

```
(defadvice! +org-latex-link (orig-fn link desc info)
  "Acts as `org-latex-link', but supports remote images."
  :around #'org-latex-link
  (setq o-link link
        o-desc desc
        o-info info)
  (if (and (member (plist-get (cadr link) :type) '("http" "https"))
           (member (file-name-extension (plist-get (cadr link) :path))
                   '("png" "jpg" "jpeg" "pdf" "svg")))
```

```
      (org-latex-link--remote link desc info)
    (funcall orig-fn link desc info)))

(defun org-latex-link--remote (link _desc info)
  (let* ((url (plist-get (cadr link) :raw-link))
         (ext (file-name-extension url))
         (target (format "%s%s.%s"
                         (temporary-file-directory)
                         (replace-regexp-in-string "[./]" "-"
                                                   (file-name-sans-extension
                                                    ↪  (substring
                                                    ↪  (plist-get (cadr
                                                    ↪  link) :path) 2)))
                         ext)))
    (unless (file-exists-p target)
      (url-copy-file url target))
    (setcdr link (--> (cadr link)
                      (plist-put it :type "file")
                      (plist-put it :path target)
                      (plist-put it :raw-link (concat "file:" target))
                      (list it)))
    (concat "% fetched from " url "\n"
            (org-latex--inline-image link info))))
```

(c) LatexMK 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

On the other hand, it still refuses to work with previews and just sucks with emacs overall. Back to LatexMK for me

```
(setq org-latex-pdf-process (list "latexmk -f -pdflatex='xelatex
↪  -shell-escape -interaction nonstopmode' -pdf -output-directory=%o %f"))

(setq xdvsvgm
        '(xdvsvgm
          :programs ("xelatex" "dvisvgm")
          :description "xdv > svg"
          :message "you need to install the programs: xelatex and dvisvgm."
          :use-xcolor t
          :image-input-type "xdv"
          :image-output-type "svg"
          :image-size-adjust (1.7 . 1.5)
          :latex-compiler ("xelatex -no-pdf -interaction nonstopmode
          ↪  -output-directory %o %f")
          :image-converter ("dvisvgm %f -n -b min -c %S -o %O")))

(after! org
    (add-to-list 'org-preview-latex-process-alist xdvsvgm)
    (setq org-format-latex-options
      (plist-put org-format-latex-options :scale 1.4))
```

```
    (setq org-preview-latex-default-process 'xdvsvgm))
```

Looks crisp!

$$f(x) = x^2$$
$$g(x) = \frac{1}{x}$$
$$F(x) = \int_b^a \frac{1}{3}x^3$$

i. Compilation

```
(setq TeX-save-query nil
      TeX-show-compilation t
      TeX-command-extra-options "-shell-escape")

(after! latex
  (add-to-list 'TeX-command-list '("XeLaTeX" "%`xelatex%(mode)%' %t"
  ↪  TeX-run-TeX nil t)))
```

(d) 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}"))
```

(e) Packages Add some packages. I'm trying to keep it basic for now, Alegreya for non-monospace and SFMono 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}"))
```

(f) 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
        engrave-faces-preset-styles (engrave-faces-generate-preset)))
```

```
(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,
      breaksym-
    bol=\\color{white!60!black}\\tiny\\ensuremath{\\hookrightarrow}}
    \\renewcommand\\theFancyVerbLine{\\footnotesize\\color{black!40!white}\\arabic{FancyVerbLin
    \\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
         "\\be-
         ↪  gin{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-scr-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)))
```

(g) ox-chameleon Nice little package to color stuff for us.

```
(use-package! ox-chameleon
  :after ox
  :config
  (setq ox-chameleon-snap-fgbg-to-bw nil))
```

(h) Async Run export processes in a background … process

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

(i) (sub|super)script characters Annoying having to gate these, so let's fix that

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

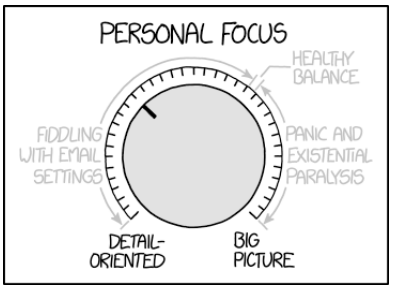
4. Calc Embedded calc is a lovely feature which let's us use calc to operate on LaTeX maths expressions. The standard keybinding is a bit janky however (`C-x * e`), so we'll add a localleader-based alternative.

```
(map! :map calc-mode-map
      :after calc
      :localleader
      :desc "Embedded calc (toggle)" "e" #'calc-embedded)
(map! :map org-mode-map
      :after org
      :localleader
      :desc "Embedded calc (toggle)" "E" #'calc-embedded)
(map! :map latex-mode-map
      :after latex
      :localleader
      :desc "Embedded calc (toggle)" "e" #'calc-embedded)
```

Unfortunately this operates without the (rather informative) calculator and trail buffers, but we can advice it that we would rather like those in a side panel.

```
(defvar calc-embedded-trail-window nil)
(defvar calc-embedded-calculator-window nil)

(defadvice! calc-embedded-with-side-pannel (&rest _)
  :after #'calc-do-embedded
  (when calc-embedded-trail-window
    (ignore-errors
      (delete-window calc-embedded-trail-window))
    (setq calc-embedded-trail-window nil))
  (when calc-embedded-calculator-window
    (ignore-errors
      (delete-window calc-embedded-calculator-window))
    (setq calc-embedded-calculator-window nil))
  (when (and calc-embedded-info
             (> (* (window-width) (window-height)) 1200))
    (let ((main-window (selected-window))
          (vertical-p (> (window-width) 80)))
      (select-window
       (setq calc-embedded-trail-window
             (if vertical-p
                 (split-window-horizontally (- (max 30 (/ (window-width) 3))))
               (split-window-vertically (- (max 8 (/ (window-height) 4)))))))
      (switch-to-buffer "*Calc Trail*")
      (select-window
       (setq calc-embedded-calculator-window
             (if vertical-p
                 (split-window-vertically -6)
               (split-window-horizontally (- (/ (window-width) 2))))))
      (switch-to-buffer "*Calculator*")
      (select-window main-window))))
```

5.1.8 Mu4e



Focus Knob Maybe if I spin it back and forth really fast I can do some kind of pulse-width modulation.

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 msmtp

```
(after! mu4e
  (setq sendmail-program "msmtp"
        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)
```

5.1.9 Browsing

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

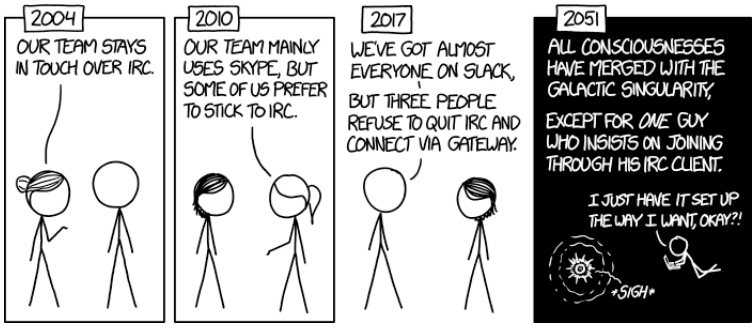
```
;;(use-package org
;;  :demand t)

;; (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))
```

2. IRC



Team Chat 2078: He announces that he's finally making the jump from screen+irssi to tmux+weechat.

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

(a) Have everything (serverinfo and passwords) in an authinfo.gpg file

(b) Tell circe to use it

(c) Use org syntax for formatting

(d) Add emoji support

(e) 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 empasis:
      *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 "\\( \\)?::?\\([^[: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"          . "😜")
    ("zanzy"                          . "🤪")
    ("raised_eyebrow"                 . "🤨")
    ("monocle"                        . "🧐")
    ("nerd"                           . "🤓")
    ("cool"                           . "😎")
    ("star_struck"                    . "🤩")
    ("party"                          . "🥳")
    ("smirk"                          . "😏")
    ("unamused"                       . "😒")
    ("disapointed"                    . "😞")
    ("pensive"                        . "😔")
    ("worried"                        . "😟")
    ("confused"                       . "😕")
    ("slight_frown"                   . "🙁")
    ("frown"                          . "☹")
    ("persevere"                      . "😣")
    ("confounded"                     . "😖")
    ("tired"                          . "😫")
    ("weary"                          . "😩")
    ("pleading"                       . "🥺")
    ("tear"                           . "😢")
    ("cry"                            . "😭")
    ("sob"                            . "😤")
```

```
    ("triumph"                  . "")
    ("angry"                    . "")
    ("rage"                     . "")
    ("exploding_head"           . "")
    ("flushed"                  . "")
    ("hot"                      . "")
    ("cold"                     . "")
    ("scream"                   . "")
    ("fearful"                  . "")
    ("disapointed"              . "")
    ("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"                       . "")
    ("thumbdown"                . "")
    ("-1"                       . "")
    ("ok"                       . "")
    ("pinch"                    . "")
    ("left"                     . "")
    ("right"                    . "")
    ("down"                     . "")
    ("wave"                     . "")
```

```
    ("pray"                          . "🙏")
    ("eyes"                          . "👀")
    ("brain"                         . "🧠")
    ("facepalm"                      . "🤦")
    ("tada"                          . "🎉")
    ("fire"                          . "🔥")
    ("flying_money"                  . "💸")
    ("lighbulb"                      . "💡")
    ("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")))
```

5.2 Neovim

There are many neovim configurations that exist (i.e. NvChad, Lunar Vim, etc.). However, many of these configurations suffer from a host of problems:

Some configurations (like NvChad), have very abstracted and complex codebases. Others rely on having as much overall functionality as possible (like LunarVim). While none of this is bad, there are some problems that can arise from these choices:

Complex codebases lead to less freedom for end-user extensiblity and configuration, as there is more reliance on the maintainer of said code. Users may not use half of what is made avalible to them simply because they don't need all of that functionality, so all of it may not be necessary. This config provides a solution to these problems by providing only the necessary code in order

to make a functioning configuration. The end goal of this personal neovim config is to be used as a base config for users to extend and add upon, leading to a more unique editing experience.

The configuration was originally based off of commit 29f04fc of NvChad, but this config has evolved to be much more than that.

5.2.1 Init

The init.lua first loads impatient.nvim if available (so we can cache the `.lua` files). It then disables builtin vim plugins, and loads the required modules for startup (`packer_compiled.lua`, `mappings.lua`, and `options.lua`)

```
--load impatient first
local impatient, impatient = pcall(require, "impatient")
if impatient then
   -- NOTE: currently broken, will fix soon
   --impatient.enable_profile()
end

--disable builtin plugins
local disabled_built_ins = {
   "2html_plugin",
   "getscript",
   "getscriptPlugin",
   "gzip",
   "logipat",
   "netrw",
   "netrwPlugin",
   "netrwSettings",
   "netrwFileHandlers",
   "matchit",
   "tar",
   "tarPlugin",
   "rrhelper",
   "spellfile_plugin",
   "vimball",
   "vimballPlugin",
   "zip",
   "zipPlugin",
}

for _, plugin in pairs(disabled_built_ins) do
   vim.g["loaded_" .. plugin] = 1
end

-- load options, mappings, and plugins
local doom_modules = {
   "options",
   "mappings",
```

```
   "packer_compiled",
}

for i = 1, #doom_modules, 1 do
   pcall(require, doom_modules[i])
end
```

5.2.2 Packer

My packer configuration is broken into two files: `packerInit` and `pluginList`. `packerInit` downloads packer if it isn't present, lazy loads it if it is, and configures packer. Notably:

- Put the `packer_compiled` file under `/nvim/lua` instead of `/nvim/plugin` so it can be chached by impatient.nvim
- Use packer in a floating window instead of a split, and remove the borders
- Increaes `clone_timeout`, just in case I'm on a more finicky network

`pluginList` contains the list of plugins, as well lazy loads and defines their configuration files.

```
vim.cmd "packadd packer.nvim"
local present, packer = pcall(require, "packer")

--clone packer if its missing
if not present then
   local packer_path = vim.fn.stdpath "data" .. "/site/pack/packer/opt/packer.nvim"

   print "Cloning packer.."
   -- remove the dir before cloning
   vim.fn.delete(packer_path, "rf")
   vim.fn.system {
      "git",
      "clone",
      "https://github.com/wbthomason/packer.nvim",
      "--depth",
      "20",
      packer_path,
   }

   vim.cmd "packadd packer.nvim"
   present, packer = pcall(require, "packer")

   if present then
      print "Packer cloned successfully."
   else
      error("Couldn't clone packer !\nPacker path: " .. packer_path)
   end
```

```
end

-- packer settings
return packer.init {
   -- tell packer to put packer_compiled under the /lua folder so its cached by
   ↪  impatient
   compile_path = vim.fn.stdpath "config" .. "/lua/packer_compiled.lua",
   display = {
      open_fn = function()
         return require("packer.util").float { border = "single" }
      end,
      prompt_border = "single",
   },
   git = {
      clone_timeout = 600, -- Timeout, in seconds, for git clones
   },
}
```

```
local present, packer = pcall(require, "packerInit")

if present then
   packer = require "packer"
else
   return false
end

local use = packer.use
return packer.startup(function()
   -- Have packer manage itself
   use {
      "wbthomason/packer.nvim",
      event = "VimEnter",
   }

   -- Required for telescope
   use {
      "nvim-lua/plenary.nvim",
      event = "VimEnter",
   }

   -- Startup optimizations

   use {
      "nathom/filetype.nvim",
   }

   use {
      "lewis6991/impatient.nvim",
   }

   use {
      "tweekmonster/startuptime.vim",
      cmd = "StartupTime",
```

```
}

-- Use fancy plugin for JK escape
use {
   "max397574/better-escape.nvim",
   event = "InsertEnter",
   config = function()
      require("better_escape").setup {
         mapping = { "jk", "jj" },
         clear_empty_lines = true,
         keys = "<Esc>",
      }
   end,
}

-- Theme <3 and UI
use {
   "shaunsingh/nord.nvim",
   after = "packer.nvim",
   config = function()
      require("nord").set()
   end,
}

use {
   "kyazdani42/nvim-web-devicons",
   after = "nord.nvim",
}

use {
   "NTBBloodbath/galaxyline.nvim",
   after = "nvim-web-devicons",
   config = function()
      require "plugins.statusline"
   end,
}

use {
   "akinsho/bufferline.nvim",
   after = "nvim-web-devicons",
   config = function()
      require "plugins.bufferline"
   end,
}

use {
   "lukas-reineke/indent-blankline.nvim",
   after = "nord.nvim",
   config = function()
      require("plugins.others").blankline()
   end,
}
```

```
use {
   "norcalli/nvim-colorizer.lua",
   cmd = "ColorizerToggle",
   config = function()
      require("plugins.others").colorizer()
   end,
}

use {
   "nvim-treesitter/nvim-treesitter",
   after = "nord.nvim",
   config = function()
      require "plugins.treesitter"
   end,
}

use {
   "nvim-treesitter/playground",
   cmd = "TSPlayground",
}

use {
   "p00f/nvim-ts-rainbow",
   after = "nvim-treesitter",
}

use {
   "lewis6991/gitsigns.nvim",
   after = "nord.nvim",
}

use {
   "kyazdani42/nvim-tree.lua",
   cmd = { "NvimTreeToggle", "NvimTreeFocus" },
   config = function()
      require "plugins.nvimtree"
   end,
}

-- LSP (and copilot
use {
   "github/copilot.vim",
   event = "InsertEnter",
}

use {
   "neovim/nvim-lspconfig",
   after = "nvim-lsp-installer",
   config = function()
      require "plugins.lspconfig"
   end,
}
```

```
use {
   "williamboman/nvim-lsp-installer",
   event = "InsertEnter",
}

use {
   "ray-x/lsp_signature.nvim",
   after = "nvim-lspconfig",
   config = function()
      require("plugins.others").signature()
   end,
}

use {
   "rafamadriz/friendly-snippets",
   event = "InsertEnter",
}

use {
   "hrsh7th/nvim-cmp",
   after = "friendly-snippets",
   config = function()
      require "plugins.cmp"
   end,
}

use {
   "L3MON4D3/LuaSnip",
   wants = "friendly-snippets",
   after = "nvim-cmp",
   config = function()
      require("plugins.others").luasnip()
   end,
}

use {
   "saadparwaiz1/cmp_luasnip",
   after = "LuaSnip",
}

use {
   "hrsh7th/cmp-nvim-lua",
   after = "cmp_luasnip",
}

use {
   "hrsh7th/cmp-nvim-lsp",
   after = "cmp-nvim-lua",
}

use {
   "lukas-reineke/cmp-rg",
   after = "cmp-nvim-lsp",
```

```
}

use {
   "ray-x/cmp-treesitter",
   after = "cmp-nvim-lsp",
}

use {
   "hrsh7th/cmp-path",
   after = "cmp-rg",
}

use {
   "nvim-telescope/telescope.nvim",
   cmd = "Telescope",
   requires = {
      {
         "nvim-telescope/telescope-fzf-native.nvim",
         run = "make",
      },
   },
   config = function()
      require "plugins.telescope"
   end,
}

use {
   "Pocco81/TrueZen.nvim",
   cmd = {
      "TZAtaraxis",
      "TZMinimalist",
      "TZFocus",
   },
   config = function()
      require "plugins.zenmode"
   end,
}

use {
   "folke/twilight.nvim",
   cmd = {
      "Twilight",
      "TwilightEnable",
   },
   config = function()
      require("twilight").setup {}
   end,
}

use {
   "phaazon/hop.nvim",
   cmd = {
      "HopWord",
```

```
            "HopLine",
            "HopChar1",
            "HopChar2",
            "HopPattern",
        },
        as = "hop",
        config = function()
            require("hop").setup()
        end,
    }

    use {
        "sindrets/diffview.nvim",
        after = "neogit",
    }

    use {
        "TimUntersberger/neogit",
        cmd = {
            "Neogit",
            "Neogit commit",
        },
        config = function()
            require "plugins.neogit"
        end,
    }

    use {
        "nvim-neorg/neorg",
        branch = "unstable",
        ft = "norg",
        requires = "nvim-lua/plenary.nvim",
        config = function()
            require "plugins.neorg"
        end,
    }

    use {
        "nvim-neorg/neorg-telescope",
        ft = "norg",
    }
end)
```

5.2.3 Settings

As I said earlier, there are 3 required modules for startup (`packer_compiled.lua`, `mappings.lua`, and `options.lua`). Of that, `packer_compiled.lua` is generated using `:PackerCompile`, so we will focus on the other two.

- `mappings.lua` contains all of my mappings. All of the mappings are the same as the

defaults for Doom Emacs (with a few exceptions). To list all of the keybinds, run `SPC h b b` in doom (or `SPC h b f` for major-mode specific binds). The file also contains some commands, which allow for the lazy loading of packer.

- `options.lua` contains all the basic options I want set before loading a buffer. Additionally, I want to disable the tilde fringe and `filetype.vim` (replaced with `filetype.nvim`).

```
-- helper function for clean mappings
local function map(mode, lhs, rhs, opts)
   local options = { noremap = true, silent = true }
   if opts then
      options = vim.tbl_extend("force", options, opts)
   end
   vim.api.nvim_set_keymap(mode, lhs, rhs, options)
end

vim.g.mapleader = " " --leader
map("n", ";", ":") --semicolon to enter command mode
map("n", "j", "gj") --move by visual line not actual line
map("n", "k", "gk")
map("n", "<leader>ww", "<cmd>HopWord<CR>") --easymotion/hop
map("n", "<leader>l", "<cmd>HopLine<CR>")
map("n", "<leader>/", "<cmd>HopPattern<CR>")
map("n", "<leader>fr", "<cmd>Telescope oldfiles<CR>") --fuzzy
map("n", "<leader>.", "<cmd>Telescope find_files<CR>")
map("n", "<leader>f", "<cmd>Telescope current_buffer_fuzzy_find<CR>")
map("n", "<leader>:", "<cmd>Telescope commands<CR>")
map("n", "<leader>bb", "<cmd>Telescope buffers<CR>")
map("n", "<leader>tz", "<cmd>TZAtaraxis<CR>") --ataraxis
map("n", "<leader>op", "<cmd>NvimTreeToggle<CR>") --nvimtree
map("n", "<leader>tw", "<cmd>set wrap!<CR>") --nvimtree
map("n", "<c-k>", "<cmd>wincmd k<CR>") --ctrlhjkl to navigate splits
map("n", "<c-j>", "<cmd>wincmd j<CR>")
map("n", "<c-h>", "<cmd>wincmd h<CR>")
map("n", "<c-l>", "<cmd>wincmd l<CR>")

-- since we lazy load packer.nvim, we need to load it when we run packer-related
↪  commands
vim.cmd "silent! command PackerCompile lua require 'pluginList'
↪  require('packer').compile()"
vim.cmd "silent! command PackerInstall lua require 'pluginList'
↪  require('packer').install()"
vim.cmd "silent! command PackerStatus lua require 'pluginList'
↪  require('packer').status()"
vim.cmd "silent! command PackerSync lua require 'pluginList' require('packer').sync()"
vim.cmd "silent! command PackerUpdate lua require 'pluginList'
↪  require('packer').update()"
```

```
-- Do not source the default filetype.vim
vim.g.did_load_filetypes = 1
vim.g.neovide_cursor_vfx_mode = "pixiedust" -- neovide trail
vim.opt.fillchars = { eob = " " } -- disable tilde fringe
```

```
vim.opt.undofile = true -- enable persistent undo
vim.opt.swapfile = false -- disable swap
vim.opt.ignorecase = true -- case insensitive search
vim.opt.splitbelow = true -- default to split below/right
vim.opt.splitright = true
vim.opt.cursorline = true -- enable cursorline
vim.opt.mouse = "a" -- enable mouse
vim.opt.signcolumn = "yes" -- enable signcolumn
vim.opt.clipboard = "unnamedplus" -- enable universal clipboard
vim.opt.scrolloff = 3 -- leave 3 lines up/down while scrolling
vim.opt.tabstop = 4 -- tabs should be 4 "space" wide
vim.opt.shiftwidth = 4 -- tabs should be 4 "space" wide
vim.opt.lazyredraw = true -- usefull for regexes with large files
vim.opt.linebreak = true -- clean linebreaks
vim.opt.number = false -- disable numbers
vim.opt.numberwidth = 2 -- two wide number column
vim.opt.shortmess:append "casI" -- disable intro
vim.opt.whichwrap:append "<>hl" -- clean aligned wraps
vim.opt.guifont = "Liga SFMono Nerd Font:h14" -- set guifont for neovide
```

5.2.4 Plugin Configuration

1. Bufferline

```
local present, bufferline = pcall(require, "bufferline")
if not present then
   return
end

-- function executed for top right close button in bufferline
vim.cmd "function! Doom_bufferline_quitvim(a,b,c,d) \n qa \n endfunction"

local colors = {
   bg = "NONE",
   black = "#2E3440",
   black2 = "#2a2e38",
   white = "#ECEFF4",
   fg = "#E5E9F0",
   yellow = "#EBCB8B",
   cyan = "#A3BE8C",
   darkblue = "#81A1C1",
   green = "#8FBCBB",
   orange = "#D08770",
   purple = "#B48EAD",
   magenta = "#BF616A",
   gray = "#616E88",
   blue = "#5E81AC",
   red = "#BF616A",
}
```

```
bufferline.setup {
   options = {
      offsets = { { filetype = "NvimTree", text = "", padding = 1 } },
      buffer_close_icon = "",
      modified_icon = "",
      close_icon = "",
      show_close_icon = true,
      left_trunc_marker = "",
      right_trunc_marker = "",
      max_name_length = 14,
      max_prefix_length = 13,
      tab_size = 20,
      show_tab_indicators = true,
      enforce_regular_tabs = false,
      view = "multiwindow",
      show_buffer_close_icons = true,
      separator_style = "thin",
      always_show_bufferline = false,
      diagnostics = false, -- "or nvim_lsp"
      custom_filter = function(buf_number)
         -- Func to filter out our managed/persistent split terms
         local present_type, type = pcall(function()
            return vim.api.nvim_buf_get_var(buf_number, "term_type")
         end)

         if present_type then
            if type == "vert" then
               return false
            elseif type == "hori" then
               return false
            else
               return true
            end
         else
            return true
         end
      end,
   },

   highlights = {
      background = {
         guifg = colors.fg,
         guibg = colors.black2,
      },

      -- buffers
      buffer_selected = {
         guifg = colors.white,
         guibg = colors.black,
         gui = "bold",
      },
      buffer_visible = {
         guifg = colors.gray,
```

```
      guibg = colors.black2,
    },

    -- for diagnostics = "nvim_lsp"
    error = {
      guifg = colors.gray,
      guibg = colors.black2,
    },
    error_diagnostic = {
      guifg = colors.gray,
      guibg = colors.black2,
    },

    -- close buttons
    close_button = {
      guifg = colors.gray,
      guibg = colors.black2,
    },
    close_button_visible = {
      guifg = colors.gray,
      guibg = colors.black2,
    },
    close_button_selected = {
      guifg = colors.red,
      guibg = colors.black,
    },
    fill = {
      guifg = colors.fg,
      guibg = colors.black2,
    },
    indicator_selected = {
      guifg = colors.black,
      guibg = colors.black,
    },

    -- modified
    modified = {
      guifg = colors.red,
      guibg = colors.black2,
    },
    modified_visible = {
      guifg = colors.red,
      guibg = colors.black2,
    },
    modified_selected = {
      guifg = colors.green,
      guibg = colors.black,
    },

    -- separators
    separator = {
      guifg = colors.black2,
      guibg = colors.black2,
```

```
      },
      separator_visible = {
         guifg = colors.black2,
         guibg = colors.black2,
      },
      separator_selected = {
         guifg = colors.black2,
         guibg = colors.black2,
      },
      -- tabs
      tab = {
         guifg = colors.gray,
         guibg = colors.black2,
      },
      tab_selected = {
         guifg = colors.black2,
         guibg = colors.darkblue,
      },
      tab_close = {
         guifg = colors.red,
         guibg = colors.black,
      },
   },
}
```

2. Nvim-cmp

```
local present, cmp = pcall(require, "cmp")

if not present then
   return
end

vim.opt.completeopt = "menuone,noselect"

-- nvim-cmp setup
cmp.setup {
   snippet = {
      expand = function(args)
         require("luasnip").lsp_expand(args.body)
      end,
   },
   formatting = {
      format = function(entry, vim_item)
         vim_item.menu = ({
            rg = "rg",
            nvim_lsp = "LSP",
            nvim_lua = "Lua",
            Path = "Path",
            luasnip = "LuaSnip",
            neorg = "Neorg",
            treesitter = "ts",
```

```
        })[entry.source.name]
        vim_item.kind = ({
           Text = "",
           Method = "",
           Function = "",
           Constructor = "",
           Field = "",
           Variable = "",
           Class = "",
           Interface = "",
           Module = "",
           Property = "",
           Unit = "",
           Value = "",
           Enum = "",
           Keyword = "",
           Snippet = "",
           Color = "",
           File = "",
           Reference = "",
           Folder = "",
           EnumMember = "",
           Constant = "",
           Struct = "",
           Event = "",
           Operator = "",
           TypeParameter = "",
        })[vim_item.kind]
        return vim_item
     end,
  },
  mapping = {
     ["<C-p>"] = cmp.mapping.select_prev_item(),
     ["<C-n>"] = cmp.mapping.select_next_item(),
     ["<C-d>"] = cmp.mapping.scroll_docs(-4),
     ["<C-f>"] = cmp.mapping.scroll_docs(4),
     ["<C-Space>"] = cmp.mapping.complete(),
     ["<C-e>"] = cmp.mapping.close(),
     ["<CR>"] = cmp.mapping.confirm {
        behavior = cmp.ConfirmBehavior.Replace,
        select = true,
     },
     ["<Tab>"] = function(fallback)
        if cmp.visible() then
           cmp.select_next_item()
        elseif require("luasnip").expand_or_jumpable() then
          vim.fn.feedkeys(vim.api.nvim_replace_termcodes("<Plug>luasnip-expand-
           ↪  or-jump", true, true, true),
           ↪  "")
        else
           fallback()
        end
     end,
```

```
      ["<S-Tab>"] = function(fallback)
         if cmp.visible() then
            cmp.select_prev_item()
         elseif require("luasnip").jumpable(-1) then
            vim.fn.feedkeys(vim.api.nvim_replace_termcodes("<Plug>luasnip-jump-
            ↪  prev", true, true, true),
            ↪  "")
         else
            fallback()
         end
      end,
   },
   sources = {
      { name = "nvim_lsp" },
      { name = "luasnip" },
      { name = "rg" },
      { name = "nvim_lua" },
      { name = "neorg" },
      { name = "treesitter" },
   },
}
```

3. Gitsigns

```
local present, gitsigns = pcall(require, "gitsigns")
if not present then
   return
end

gitsigns.setup {
   signs = {
      add = { hl = "GitSignsAdd", text = "▎", numhl = "GitSignsAddNr", linehl =
      ↪  "GitSignsAddLn" },
      change = { hl = "GitSignsChange", text = "▎", numhl = "GitSignsChangeNr",
      ↪  linehl = "GitSignsChangeLn" },
      delete = { hl = "GitSignsDelete", text = "_", numhl = "GitSignsDeleteNr",
      ↪  linehl = "GitSignsDeleteLn" },
      topdelete = { hl = "GitSignsDelete", text = "‾", numhl =
      ↪  "GitSignsDeleteNr", linehl = "GitSignsDeleteLn" },
      changedelete = { hl = "GitSignsChange", text = "~", numhl =
      ↪  "GitSignsChangeNr", linehl = "GitSignsChangeLn" },
   },
   signcolumn = true, -- Toggle with `:Gitsigns toggle_signs`
   numhl = false, -- Toggle with `:Gitsigns toggle_numhl`
   linehl = false, -- Toggle with `:Gitsigns toggle_linehl`
   keymaps = {
      -- Default keymap options
      noremap = true,

      ["n ]c"] = { expr = true, "&diff ? ']c' : '<cmd>lua
      ↪  require\"gitsigns.actions\".next_hunk()<CR>'" },
```

```
      ["n [c"] = { expr = true, "&diff ? '[c' : '<cmd>lua
      ↪  require\"gitsigns.actions\".prev_hunk()<CR>'" },

      ["n <leader>hs"] = '<cmd>lua require"gitsigns".stage_hunk()<CR>',
      ["v <leader>hs"] = '<cmd>lua
      ↪  require"gitsigns".stage_hunk({vim.fn.line("."),
      ↪  vim.fn.line("v")})<CR>',
      ["n <leader>hu"] = '<cmd>lua require"gitsigns".undo_stage_hunk()<CR>',
      ["n <leader>hr"] = '<cmd>lua require"gitsigns".reset_hunk()<CR>',
      ["v <leader>hr"] = '<cmd>lua
      ↪  require"gitsigns".reset_hunk({vim.fn.line("."),
      ↪  vim.fn.line("v")})<CR>',
      ["n <leader>hR"] = '<cmd>lua require"gitsigns".reset_buffer()<CR>',
      ["n <leader>hp"] = '<cmd>lua require"gitsigns".preview_hunk()<CR>',
      ["n <leader>hb"] = '<cmd>lua require"gitsigns".blame_line(true)<CR>',
      ["n <leader>hS"] = '<cmd>lua require"gitsigns".stage_buffer()<CR>',
      ["n <leader>hU"] = '<cmd>lua require"gitsigns".reset_buffer_index()<CR>',

      -- Text objects
      ["o ih"] = ':<C-U>lua require"gitsigns.actions".select_hunk()<CR>',
      ["x ih"] = ':<C-U>lua require"gitsigns.actions".select_hunk()<CR>',
   },
   update_debounce = 200,
}
```

4. Lspconfig

```
local present1, lspconfig = pcall(require, "lspconfig")
local present2, lsp_installer = pcall(require, "nvim-lsp-installer")

if not (present1 or present2) then
   return
end

local function on_attach(_, bufnr)
   local function buf_set_keymap(...)
      vim.api.nvim_buf_set_keymap(bufnr, ...)
   end
   local function buf_set_option(...)
      vim.api.nvim_buf_set_option(bufnr, ...)
   end

   -- Enable completion triggered by <c-x><c-o>
   buf_set_option("omnifunc", "v:lua.vim.lsp.omnifunc")

   -- Mappings.
   local opts = { noremap = true, silent = true }

   -- See `:help vim.lsp.*` for documentation on any of the below functions
   buf_set_keymap("n", "gD", "<cmd>lua vim.lsp.buf.declaration()<CR>", opts)
   buf_set_keymap("n", "gd", "<cmd>lua vim.lsp.buf.definition()<CR>", opts)
   buf_set_keymap("n", "K", "<cmd>lua vim.lsp.buf.hover()<CR>", opts)
```

```
   buf_set_keymap("n", "gi", "<cmd>lua vim.lsp.buf.implementation()<CR>", opts)
   buf_set_keymap("n", "gk", "<cmd>lua vim.lsp.buf.signature_help()<CR>", opts)
   buf_set_keymap("n", "<space>wa", "<cmd>lua
   ↪  vim.lsp.buf.add_workspace_folder()<CR>", opts)
   buf_set_keymap("n", "<space>wr", "<cmd>lua
   ↪  vim.lsp.buf.remove_workspace_folder()<CR>", opts)
   buf_set_keymap("n", "<space>wl", "<cmd>lua
   ↪  print(vim.inspect(vim.lsp.buf.list_workspace_folders()))<CR>", opts)
   buf_set_keymap("n", "<space>D", "<cmd>lua vim.lsp.buf.type_definition()<CR>",
   ↪  opts)
   buf_set_keymap("n", "<space>rn", "<cmd>lua vim.lsp.buf.rename()<CR>", opts)
   buf_set_keymap("n", "<space>ca", "<cmd>lua vim.lsp.buf.code_action()<CR>",
   ↪  opts)
   buf_set_keymap("n", "gr", "<cmd>lua vim.lsp.buf.references()<CR>", opts)
   buf_set_keymap("n", "<space>e", "<cmd>lua
   ↪  vim.lsp.diagnostic.show_line_diagnostics()<CR>", opts)
   buf_set_keymap("n", "[d", "<cmd>lua vim.lsp.diagnostic.goto_prev()<CR>", opts)
   buf_set_keymap("n", "]d", "<cmd>lua vim.lsp.diagnostic.goto_next()<CR>", opts)
   buf_set_keymap("n", "<space>q", "<cmd>lua
   ↪  vim.lsp.diagnostic.set_loclist()<CR>", opts)
   buf_set_keymap("n", "<space>f", "<cmd>lua vim.lsp.buf.formatting()<CR>", opts)
   buf_set_keymap("v", "<space>ca", "<cmd>lua
   ↪  vim.lsp.buf.range_code_action()<CR>", opts)
end

local capabilities = vim.lsp.protocol.make_client_capabilities()
capabilities.textDocument.completion.completionItem.documentationFormat = {
↪  "markdown", "plaintext" }
capabilities.textDocument.completion.completionItem.snippetSupport = true
capabilities.textDocument.completion.completionItem.preselectSupport = true
capabilities.textDocument.completion.completionItem.insertReplaceSupport = true
capabilities.textDocument.completion.completionItem.labelDetailsSupport = true
capabilities.textDocument.completion.completionItem.deprecatedSupport = true
capabilities.textDocument.completion.completionItem.commitCharactersSupport =
↪  true
capabilities.textDocument.completion.completionItem.tagSupport = { valueSet = {
↪  1 } }
capabilities.textDocument.completion.completionItem.resolveSupport = {
   properties = {
      "documentation",
      "detail",
      "additionalTextEdits",
   },
}
local capabilities = vim.lsp.protocol.make_client_capabilities()

lsp_installer.on_server_ready(function(server)
   local opts = {}
   server:setup(opts)
   vim.cmd [[ do User LspAttachBuffers ]]
end)

-- replace the default lsp diagnostic symbols
```

```
local function lspSymbol(name, icon)
   vim.fn.sign_define("LspDiagnosticsSign" .. name, { text = icon, numhl =
   ↪  "LspDiagnosticsDefaul" .. name })
end

lspSymbol("Error", "")
lspSymbol("Information", "")
lspSymbol("Hint", "")
lspSymbol("Warning", "")

vim.lsp.handlers["textDocument/publishDiagnostics"] =
↪  vim.lsp.with(vim.lsp.diagnostic.on_publish_diagnostics, {
   virtual_text = {
      prefix = "",
      spacing = 0,
   },
   signs = true,
   underline = true,
   update_in_insert = false, -- update diagnostics insert mode
})
vim.lsp.handlers["textDocument/hover"] = vim.lsp.with(vim.lsp.handlers.hover, {
   border = "single",
})
vim.lsp.handlers["textDocument/signatureHelp"] =
↪  vim.lsp.with(vim.lsp.handlers.signature_help, {
   border = "single",
})

-- suppress error messages from lang servers
vim.notify = function(msg, log_level, _opts)
   if msg:match "exit code" then
      return
   end
   if log_level == vim.log.levels.ERROR then
      vim.api.nvim_err_writeln(msg)
   else
      vim.api.nvim_echo({ { msg } }, true, {})
   end
end
```

5. Neogit

```
local present, neogit = pcall(require, "neogit")
if not present then
   return
end

neogit.setup {
   disable_signs = false,
   disable_context_highlighting = false,
   disable_commit_confirmation = false,
   -- customize displayed signs
```

```
   signs = {
      -- { CLOSED, OPENED }
      section = { "", "" },
      item = { "", "" },
      hunk = { "", "" },
   },
   integrations = {
      diffview = true,
   },
}
```

6. Neorg

```
local present, neorg = pcall(require, "neorg")
if not present then
   return
end

neorg.setup {
   -- Tell Neorg what modules to load
   load = {
      ["core.defaults"] = {}, -- Load all the default modules
      ["core.norg.concealer"] = {}, -- Allows for use of icons
      ["core.norg.dirman"] = { -- Manage your directories with Neorg
         config = {
            workspaces = {
               my_workspace = "~/org/neorg",
            },
         },
      },
      ["core.norg.completion"] = {
         config = {
            engine = "nvim-cmp", -- We current support nvim-compe and nvim-cmp
            ↪  only
         },
      },
      ["core.keybinds"] = { -- Configure core.keybinds
         config = {
            default_keybinds = true, -- Generate the default keybinds
            neorg_leader = "<Leader>o", -- This is the default if unspecified
         },
      },
      ["core.integrations.telescope"] = {}, -- Enable the telescope module
   },
}
```

7. Nvimtree

```
local present, nvimtree = pcall(require, "nvim-tree")

if not present then
   return
```

```
end

vim.o.termguicolors = true

vim.cmd [[highlight NvimTreeNormal guifg=#D8DEE9  guibg=#2a2e39]]

vim.g.nvim_tree_add_trailing = 0 -- append a trailing slash to folder names
vim.g.nvim_tree_highlight_opened_files = 0
vim.g.nvim_tree_indent_markers = 1
vim.g.nvim_tree_ignore = { ".git", "node_modules", ".cache" }
vim.g.nvim_tree_quit_on_open = 0 -- closes tree when file's opened
vim.g.nvim_tree_root_folder_modifier = table.concat { ":t:gs?$?/..",
↪  string.rep(" ", 1000), "?:gs?^??" }
--
vim.g.nvim_tree_show_icons = {
   folders = 1,
   files = 1,
}

vim.g.nvim_tree_icons = {
   default = "",
   symlink = "",
   git = {
      deleted = "",
      ignored = "◌",
      renamed = "",
      staged = "✓",
      unmered = "",
      unstaged = "",
      untracked = "",
   },
   folder = {
      -- disable indent_markers option to get arrows working or if you want both
      ↪  arrows and indent then just add the arrow icons in front
      ↪  ofthe default and opened folders below!
      -- arrow_open = "",
      -- arrow_closed = "",
      default = "",
      empty = "", -- 
      empty_open = "",
      open = "",
      symlink = "",
      symlink_open = "",
   },
}

nvimtree.setup {
   filters = {
      dotfiles = false,
   },
   disable_netrw = true,
   hijack_netrw = true,
   ignore_ft_on_setup = { "dashboard" },
```

```
   auto_close = false,
   open_on_tab = false,
   hijack_cursor = true,
   update_cwd = true,
   update_focused_file = {
      enable = true,
      update_cwd = true,
   },
   view = {
      allow_resize = true,
      side = "left",
      width = 30,
   },
}
```

8. Others

```
local M = {}

M.colorizer = function()
   local present, colorizer = pcall(require, "colorizer")
   if present then
      colorizer.setup({ "*" }, {
         RGB = true, -- #RGB hex codes
         RRGGBB = true, -- #RRGGBB hex codes
         names = true, -- "Name" codes like Blue
         RRGGBBAA = true, -- #RRGGBBAA hex codes
         rgb_fn = true, -- CSS rgb() and rgba() functions
         hsl_fn = true, -- CSS hsl() and hsla() functions
         css = true, -- Enable all CSS features: rgb_fn, hsl_fn, names, RGB,
         ↪  RRGGBB
         css_fn = true, -- Enable all CSS *functions*: rgb_fn, hsl_fn
         mode = "foreground", -- Set the display mode.
      })
      vim.cmd "ColorizerReloadAllBuffers"
   end
end

M.blankline = function()
   require("indent_blankline").setup {
      show_current_context = true,
      context_patterns = {
         "class",
         "return",
         "function",
         "method",
         "^if",
         "^while",
         "jsx_element",
         "^for",
         "^object",
         "^table",
```

```
            "block",
            "arguments",
            "if_statement",
            "else_clause",
            "jsx_element",
            "jsx_self_closing_element",
            "try_statement",
            "catch_clause",
            "import_statement",
            "operation_type",
         },
         filetype_exclude = {
            "help",
            "terminal",
            "dashboard",
            "packer",
            "lspinfo",
            "TelescopePrompt",
            "TelescopeResults",
         },
         buftype_exclude = { "terminal" },
         show_trailing_blankline_indent = false,
         show_first_indent_level = false,
      }
end

M.luasnip = function()
   local present, luasnip = pcall(require, "luasnip")
   if not present then
      return
   end

   luasnip.config.set_config {
      history = true,
      updateevents = "TextChanged,TextChangedI",
   }
   require("luasnip/loaders/from_vscode").load()
end

M.signature = function()
   local present, lspsignature = pcall(require, "lsp_signature")
   if present then
      lspsignature.setup {
         bind = true,
         doc_lines = 2,
         floating_window = true,
         fix_pos = true,
         hint_enable = true,
         hint_prefix = "  ",
         hint_scheme = "String",
         hi_parameter = "Search",
         max_height = 22,
```

```
         max_width = 120, -- max_width of signature floating_window, line will be
         ↪  wrapped if exceed max_width
         handler_opts = {
            border = "single", -- double, single, shadow, none
         },
         zindex = 200, -- by default it will be on top of all floating windows,
         ↪  set to 50 send it to bottom
         padding = "", -- character to pad on left and right of signature can be
         ↪  ' ', or '|'  etc
      }
   end
end

M.orgmode = function()
   local present, orgmode = pcall(require, "orgmode")
   if present then
      orgmode.setup({ "*" }, {
         org_highlight_latex_and_related = "entities",
         org_agenda_files = "~/org/*",
         org_default_notes_file = "~/org/notes.org",
         org_hide_leading_stars = true,
         org_hide_emphasis_markers = true,
         mappings = {
            global = {
               org_agenda = "<Leader>oa",
               org_capture = "<Leader>oc",
            },
            agenda = {
               org_agenda_later = "f",
               org_agenda_earlier = "b",
               org_agenda_goto_today = ".",
               org_agenda_day_view = "vd",
               org_agenda_week_view = "vw",
               org_agenda_month_view = "vm",
               org_agenda_year_view = "vy",
               org_agenda_quit = "q",
               org_agenda_switch_to = "<CR>",
               org_agenda_goto = { "<TAB>" },
               org_agenda_goto_date = "J",
               org_agenda_redo = "r",
               org_agenda_todo = "t",
               org_agenda_show_help = "?",
            },
            capture = {
               org_capture_finalize = "<C-c>",
               org_capture_refile = "<Leader>or",
               org_capture_kill = "<Leader>ok",
               org_capture_show_help = "?",
            },
            org = {
               org_increase_date = "<C-a>",
               org_decrease_date = "<C-x>",
               org_toggle_checkbox = "<C-Space>",
```

```
                org_open_at_point = "<Leader>oo",
                org_cycle = "<TAB>",
                org_global_cycle = "<S-TAB>",
                org_archive_subtree = "<Leader>o$",
                org_set_tags_command = "<Leader>ot",
                org_toggle_archive_tag = "<Leader>oA",
                org_do_promote = "<<",
                org_do_demote = ">>",
                org_promote_subtree = "<s",
                org_demote_subtree = ">s",
                org_meta_return = "<Leader><CR>", -- Add headling, item or row
                org_insert_heading_respect_content = "<Leader>oih", -- Add new
                ↪  headling after current heading block with same level
                org_insert_todo_heading = "<Leader>oiT", -- Add new todo headling
                ↪  right after current heading with same level
                org_insert_todo_heading_respect_content = "<Leader>oit", -- Add
                ↪  new todo headling after current heading block on same level
                org_move_subtree_up = "<Leader>oK",
                org_move_subtree_down = "<Leader>oJ",
                org_export = "<Leader>oe",
                org_next_visible_heading = "}",
                org_previous_visible_heading = "{",
                org_forward_heading_same_level = "]]",
                org_backward_heading_same_level = "[[",
            },
        },
    })
  end
end

return M
```

9. Statusline

```
local gl = require "galaxyline"
local condition = require "galaxyline.condition"

local gls = gl.section
gl.short_line_list = { "NvimTree", "packer" }

-- Colors
local colors = {
  bg = "#3B4252",
  fg = "#E5E9F0",
  section_bg = "#434C5E",
  section_bg2 = "#4C566A",
  yellow = "#EBCB8B",
  cyan = "#8FBCBB",
  darkblue = "#81A1C1",
  green = "#A3BE8C",
  orange = "#D08770",
  magenta = "#BF616A",
```

```
   blue = "#5E81AC",
   red = "#BF616A",
   none = "2E3440",
}

--hide inactive statusline
vim.cmd [[hi StatusLineNC gui=underline guibg=#2E3440  guifg=#2E3440]]

-- Local helper functions
local mode_color = function()
   local mode_colors = {
      n = colors.cyan,
      i = colors.green,
      c = colors.orange,
      V = colors.magenta,
      [""] = colors.magenta,
      v = colors.magenta,
      R = colors.red,
   }

   local color = mode_colors[vim.fn.mode()]

   if color == nil then
      color = colors.red
   end

   return color
end

-- Left side
gls.left[1] = {
   ViMode = {
      provider = function()
         local alias = {
            n = "λ",
            i = "",
            c = "",
            V = "",
            [""] = "",
            v = "",
            R = "",
         }
         vim.api.nvim_command("hi GalaxyViMode guibg=" .. mode_color())
         local alias_mode = alias[vim.fn.mode()]
         if alias_mode == nil then
            alias_mode = vim.fn.mode()
         end
         return "  " .. alias_mode .. " "
      end,
      highlight = { colors.bg, colors.bg },
      separator = " ",
      separator_highlight = { colors.section_bg2, colors.section_bg2 },
   },
```

```
}
gls.left[2] = {
   FileIcon = {
      provider = "FileIcon",
      highlight = {
         require("galaxyline.providers.fileinfo").get_file_icon_color,
         colors.section_bg2,
      },
   },
}
gls.left[3] = {
   FileName = {
      provider = "FileName",
      highlight = { colors.fg, colors.section_bg2 },
      separator = " ",
      separator_highlight = { colors.section_bg2, colors.section_bg },
   },
}
gls.left[4] = {
   GitIcon = {
      provider = function()
         return "  "
      end,
      condition = condition.check_git_workspace,
      highlight = { colors.red, colors.section_bg },
   },
}
gls.left[5] = {
   GitBranch = {
      provider = function()
         local vcs = require "galaxyline.providers.vcs"
         local branch_name = vcs.get_git_branch()
         if string.len(branch_name) > 28 then
            return string.sub(branch_name, 1, 25) .. "..."
         end
         return branch_name .. " "
      end,
      condition = condition.check_git_workspace,
      highlight = { colors.fg, colors.section_bg },
   },
}
gls.left[6] = {
   DiffAdd = {
      provider = "DiffAdd",
      condition = condition.check_git_workspace,
      icon = " ",
      highlight = { colors.green, colors.section_bg },
   },
}
gls.left[7] = {
   DiffModified = {
      provider = "DiffModified",
      condition = condition.check_git_workspace,
```

```
      icon = "  ",
      highlight = { colors.orange, colors.section_bg },
   },
}
gls.left[8] = {
   DiffRemove = {
      provider = "DiffRemove",
      condition = condition.check_git_workspace,
      icon = "  ",
      highlight = { colors.red, colors.section_bg },
   },
}
gls.left[9] = {
   LeftEnd = {
      provider = function()
         return "  "
      end,
      highlight = { colors.bg, colors.section_bg },
   },
}
gls.left[10] = {
   DiagnosticError = {
      provider = "DiagnosticError",
      icon = "   ",
      highlight = { colors.red, colors.bg },
   },
}
gls.left[11] = {
   Space = {
      provider = function()
         return " "
      end,
      highlight = { colors.section_bg, colors.bg },
   },
}
gls.left[12] = {
   DiagnosticWarn = {
      provider = "DiagnosticWarn",
      icon = "   ",
      highlight = { colors.orange, colors.bg },
   },
}
gls.left[13] = {
   DiagnosticHint = {
      provider = "DiagnosticHint",
      icon = "   ",
      highlight = { colors.fg, colors.bg },
   },
}
gls.left[14] = {
   Space = {
      provider = function()
         return " "
```

```
      end,
      highlight = { colors.bg, colors.bg },
   },
}
gls.left[15] = {
   DiagnosticInfo = {
      provider = "DiagnosticInfo",
      icon = "  ",
      highlight = { colors.blue, colors.bg },
      separator = " ",
      separator_highlight = { colors.bg, colors.bg },
   },
}

-- Right side
gls.right[1] = {
   LineInfo = {
      provider = "LineColumn",
      highlight = { colors.fg, colors.section_bg },
      separator = "  ",
      separator_highlight = { colors.section_bg, colors.bg },
   },
}
gls.right[2] = {
   Logo = {
      provider = function()
         return "  "
      end,
      highlight = { colors.red, colors.section_bg2 },
      separator = "  ",
      separator_highlight = { colors.section_bg2, colors.section_bg },
   },
}

-- Short status line
gls.short_line_left[1] = {
   ViModeShort = {
      provider = function()
         local alias = {
            n = "λ",
            i = "",
            c = "",
            V = "",
            [""] = "",
            v = "",
            R = "",
         }
         vim.api.nvim_command("hi GalaxyViMode guibg=" .. mode_color())
         local alias_mode = alias[vim.fn.mode()]
         if alias_mode == nil then
            alias_mode = vim.fn.mode()
         end
         return "   " .. alias_mode .. " "
```

```
      end,
      highlight = { colors.none, colors.bg },
      separator = " ",
      separator_highlight = { colors.none, colors.none },
   },
}
```

10. Telescope

```
local present, telescope = pcall(require, "telescope")
if not present then
   return
end

telescope.setup {
   defaults = {
      vimgrep_arguments = {
         "rg",
         "--color=never",
         "--no-heading",
         "--with-filename",
         "--line-number",
         "--column",
         "--smart-case",
      },
      prompt_prefix = " λ ",
      selection_caret = " > ",
      entry_prefix = "  ",
      initial_mode = "insert",
      selection_strategy = "reset",
      sorting_strategy = "ascending",
      layout_strategy = "horizontal",
      layout_config = {
         horizontal = {
            prompt_position = "top",
            preview_width = 0.55,
            results_width = 0.8,
         },
         vertical = {
            mirror = false,
         },
         width = 0.87,
         height = 0.80,
         preview_cutoff = 120,
      },
      file_sorter = require("telescope.sorters").get_fuzzy_file,
      file_ignore_patterns = {},
      generic_sorter = require("telescope.sorters").get_generic_fuzzy_sorter,
      path_display = { "absolute" },
      winblend = 0,
      border = {},
      borderchars = { "─", "│", "─", "│", "╭", "╮", "╯", "╰" },
```

```
      color_devicons = true,
      use_less = true,
      set_env = { ["COLORTERM"] = "truecolor" }, -- default = nil,
      file_previewer = require("telescope.previewers").vim_buffer_cat.new,
      grep_previewer = require("telescope.previewers").vim_buffer_vimgrep.new,
      qflist_previewer = require("telescope.previewers").vim_buffer_qflist.new,
      -- Developer configurations: Not meant for general override
      buffer_previewer_maker =
      ↪  require("telescope.previewers").buffer_previewer_maker,
   },
   extensions = {
      fzf = {
         fuzzy = true, -- false will only do exact matching
         override_generic_sorter = false, -- override the generic sorter
         override_file_sorter = true, -- override the file sorter
         case_mode = "smart_case", -- or "ignore_case" or "respect_case"
         -- the default case_mode is "smart_case"
      },
   },
}

local extensions = { "themes", "terms", "fzf" }
local packer_repos = [["extensions", "telescope-fzf-native.nvim"]]

pcall(function()
   for _, ext in ipairs(extensions) do
      telescope.load_extension(ext)
   end
end)
```

11. Treesitter

```
local present, ts_config = pcall(require, "nvim-treesitter.configs")
if not present then
   return
end

local parser_configs = require("nvim-treesitter.parsers").get_parser_configs()
parser_configs.norg = {
   install_info = {
      url = "https://github.com/nvim-neorg/tree-sitter-norg",
      files = { "src/parser.c", "src/scanner.cc" },
      branch = "main",
   },
}

ts_config.setup {
   ensure_installed = "maintained",
   indent = { enable = true },
   highlight = {
      enable = true,
      use_languagetree = true,
```

```
    },
    rainbow = {
        enable = true,
        extended_mode = true, -- Also highlight non-bracket delimiters like html
        ↪  tags, boolean or table: lang -> boolean
       max_file_lines = nil, -- Do not enable for files with more than n lines, int
    },
    playground = {
        enable = true,
        updatetime = 25, -- Debounced time for highlighting nodes in the playground
        ↪  from source code
        persist_queries = false, -- Whether the query persists across vim sessions
    },
}
```

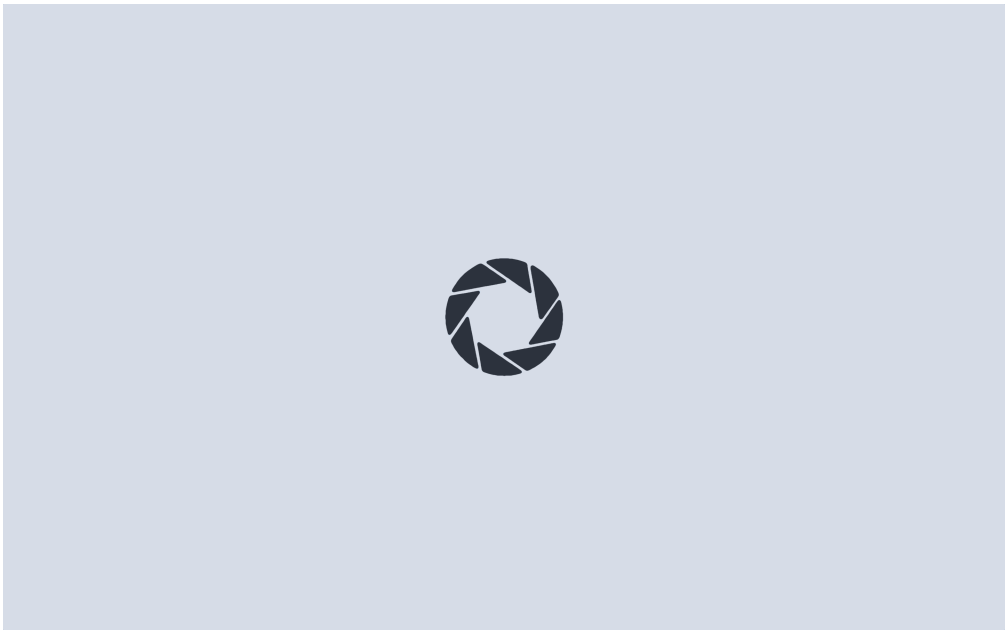
12. Zenmode

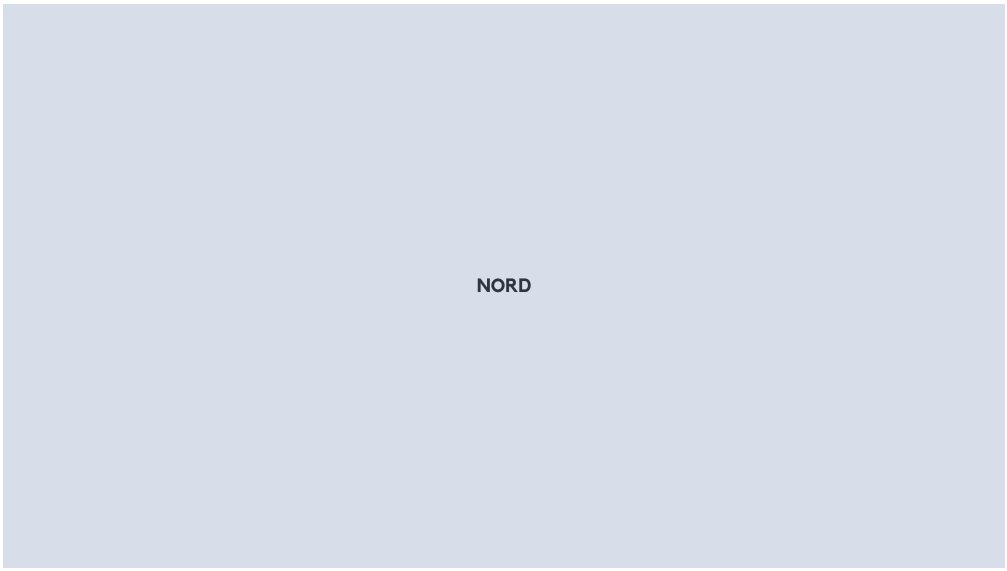
```
local present, true_zen = pcall(require, "true-zen")
if not present then
    return
end

true_zen.setup {
    ui = {
        bottom = {
            cmdheight = 1,
            laststatus = 0,
            ruler = true,
            showmode = true,
            showcmd = false,
        },
        top = {
            showtabline = 0,
        },
        left = {
            number = false,
            relativenumber = false,
            signcolumn = "no",
        },
    },
    modes = {
        ataraxis = {
            left_padding = 32,
            right_padding = 32,
            top_padding = 1,
            bottom_padding = 1,
            ideal_writing_area_width = { 0 },
            auto_padding = false,
            keep_default_fold_fillchars = false,
            bg_configuration = true,
        },
        focus = {
```

```
            margin_of_error = 5,
            focus_method = "experimental",
        },
    },
    integrations = {
        galaxyline = true,
        nvim_bufferline = true,
        twilight = true,
    },
}
```

6 Extra ATTACH

Wallpapers are in `./extra/wallpapers/` Some of my favorites:




NORD