Hacking Emacs

Jethro Kuan

July 21, 2016

Contents

1 Preface

> Emacs is a nice operating system, but what it lacks, in order to compete with Linux, is a good text editor. – Thomer M. Gil

Today, you begin your journey with Emacs, *the extensible, customizable, self-documenting real-time display editor*.

Because of its complexity, Emacs will seem difficult to grok. Rest assured that learning a select few features and packages more than suffices.

The tinkerer will marvel at the masterpiece which is Emacs. We hack on Emacs because we can; every facet of Emacs is extensible and customizable. Imagine managing email, web browsing, IRC, running shell commands, and compiling code, all just a few keystrokes away.

Emacs is not for the faint of heart, so buckle up, embrace the challenge and enjoy the ride. I hope to impart a level of Emacs-fu that's sufficient for further self-exploration, and hope you'll enjoy using Emacs as much as I do.

Atom users, go grab yourselves another cup of coffee while you wait for it to load up :P.

– Jethro Kuan

2 Introduction

2.1 Installing Emacs

If you're using a Linux distribution, obtaining Emacs should be easily achievable through your respective package managers.

For OSX users, I recommend Emacs for OSX.

This workshop assumes that a recent Emacs version (of version >24.0) has been installed. You can check your Emacs version from the command line with `emacs --version`.

2.2 Terminology

2.2.1 Windows, Frames and Buffers

The text you are editing in Emacs resides in an object called the **buffer**.

A **window** is a container for a buffer. A window can contain one and only one buffer.

A **frame** is a container for windows. While inaccurate, one can think of it as the window configuration for Emacs.

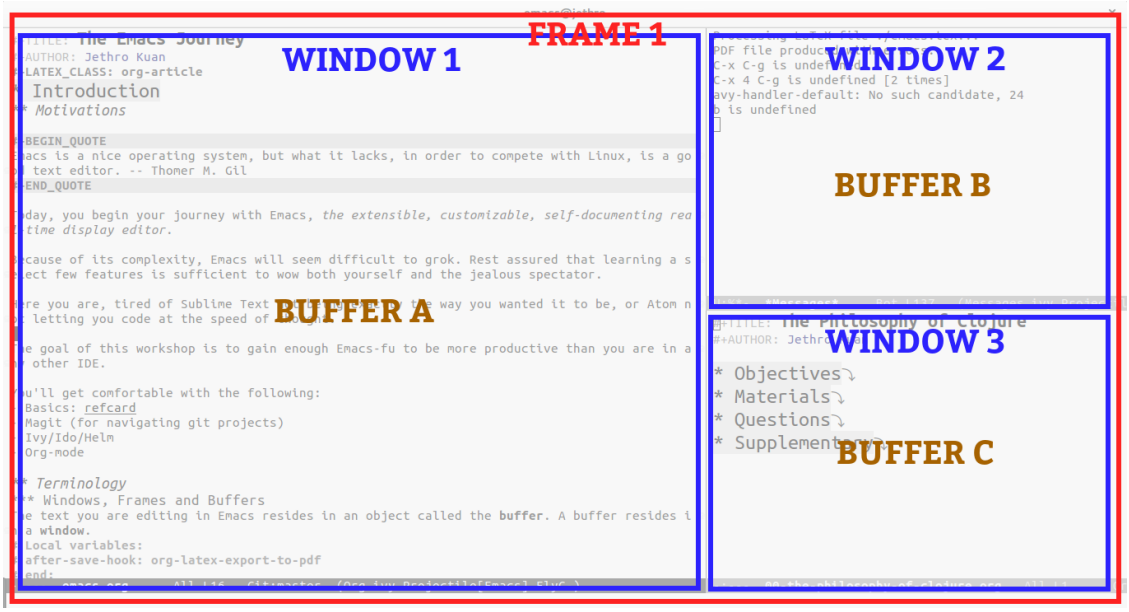

Figure 1: A pictorial representation of windows, frames and buffers. In this picture, there are 1 frame, 3 windows and 3 buffers.

An Emacs beginner might find the distinction betweeen a buffer and a window unnecessary. This distinction is made because a different windows can display the same buffer, and this may come in useful when juxtaposing text in different positions of the file.

2.2.2 Killing, Yanking and the CUA

The biggest deviation from modern standards in Emacs would most likely be the Emacs clipboard system. The table below shows the analogous terminologies for the clipboard system:

Modern	Emacs
Cut	Kill
Paste	Yank
Copy	Save To Kill Ring

The Emacs terminology was set in stone decades ago, long before the terms 'cut', 'copy' and 'paste' (derived from the CUA, or Common User Access) were formed.

I will demonstrate in a later chapter why the Emacs way is better for text-editing.

2.2.3 Modes, Major Modes and Minor Modes

Major modes control how buffers behave. Each buffer will have *one and only one* major mode. Most major modes tend to be language-specific. For example, when opening a Python file `foo.py`, the central Emacs register will figure out that this file is a Python file and load the Python major mode.

Major modes often offer the following functionality:

1. Font Locking (aka syntax highlighting outside of Emacs)
2. Indentation engines
3. Language-specific keybindings (for refactoring etc.)

In the scenario where Emacs central registry for file extensions should fail to associate the file with a major mode, Emacs will scan the first portion of the file and infer one.

Minor modes can be thought of as plugins, meant to add functionality to a buffer. These are optional, and can be added locally per buffer, or globally.

An example of a popular minor mode is the *aggressive-indent mode*, which keeps the text in the buffer properly indented at all times.

2.2.4 Modeline

The bottom bar in Emacs is called the **modeline**. The modeline presents useful information, such as what modes are active. The first item in the brackets is typically the major mode, and the others are all minor modes. Note that some minor modes can be configured to not appear on the modeline (a.k.a diminished). To see the full list of modes activated, run `M-x describe-mode`, or `C-h m`.

Figure 2: A typical modeline for Emacs

2.2.5 Keybindings and Elisp Functions

Keybindings are combinations of keys that, when pressed, invoke functions. These functions could come out-of-the-box, be provided by a package, or be self-written. They are defined in a language called **Emacs Lisp**, also referred to as **Elisp**.

All these functions are invokable with `M-x`.

3 Taming the Beast

Vanilla Emacs works in a variety of unfortunate ways. These defaults have grown a resistance to change over the decades.

In this chapter, we perform tweaking on Emacs minutiae to improve usability, and simultaneously learn how Emacs is customized.

3.1 Customizing Emacs

During initialization, Emacs attempts to load an *init* file. The *init* file is a Lisp program which is processed top-down. The scripting language for Emacs is aptly named **Emacs Lisp**. Emacs Lisp files typically have the file extension `.el`.

> PROTIP: To load vanilla Emacs, run `emacs` with the command-line switch `-q`. This will come in useful when your configuration file breaks.

Emacs searches for *init* files in several locations:

1. `~/.emacs`
2. `~/.emacs.el`
3. `~/.emacs.d/init.el`

The 3rd option `~/.emacs.d/init.el` is recommended. Having a dedicated folder for Emacs-related configuration simplifies versioning.

If you haven't done so, create a blank file `init.el` in `~/.emacs.d`.

The tweaks are listed in order of importance. To enable them, copy them into the `init.el` file.

3.1.1 Enabling Package Archives

MELPA is the de-facto package archive for Emacs. Because it is not enabled by default, we add it to the list of `package-archives`.

In similar effect, we enable the Org-mode repository, which contains the most up-to-date version of `org-with-contrib`.

```
(when (>= emacs-major-version 24)
  (require 'package)
  (add-to-list 'package-archives '("melpa" . "http://melpa.org/packages/") t)
  (add-to-list 'package-archives '("org" . "http://orgmode.org/elpa/") t)
  (package-initialize))
```

The more security conscious will note that packages are fetched using HTTP, instead of HTTPS. HTTPS, however, did not work for me. Refer here for the reasons why you might want to do so, and how to do it.

3.1.2 Setting User Details

These variables are used in some packages:

```
(setq user-full-name "John Appleseed"
      user-mail-address "john@me.com")
```

3.1.3 UI Cruft

All these UI cruft take up precious screen estate, and should be removed.

```
(tooltip-mode -1)
(tool-bar-mode -1)
(menu-bar-mode -1)
(scroll-bar-mode -1)
```

3.1.4 Startup Screen

More useless stuff to be removed.

```
(setq inhibit-splash-screen t)
(setq inhibit-startup-message t)
```

3.1.5 Use-package

use-package is a macro which allows you to isolate package configuration in a organized and performant fashion. It was created by John Wiegley, the current Emacs maintainer.

We'll be using it to install packages, so let's go ahead and add that in.

```
(unless (package-installed-p 'use-package)
  (package-refresh-contents)
  (package-install 'use-package))

(eval-and-compile
  (defvar use-package-verbose t)
  (require 'cl)
  (require 'use-package)
  (require 'bind-key)
  (require 'diminish)
  (setq use-package-always-ensure t))
```

Here, we set `use-package-always-ensure` to `true`, so if a package is found missing, it will be installed automatically.

3.1.6 y/n

It is easier to type y/n than to type yes/no.

```
(defalias 'yes-or-no-p 'y-or-n-p)
```

3.1.7 Custom Files

Emacs comes with a built-in interface to customize variable values such as font-faces. However, these get added to the bottom of your `init.el` file, which ends up looking like a mess. I like to keep these things saved in a separate file, and load them in.

```
(setq custom-file "~/.emacs.d/custom.el")
(load custom-file)
```

3.1.8 Backup Files

Backup files are important, but they tend to litter your directories. The following snippet moves the temp files to the system temp directory.

```
(setq backup-directory-alist
      `((".*" . ,temporary-file-directory)))
(setq auto-save-file-name-transforms
      `((".*" ,temporary-file-directory t)))
```

When added to your `init.el`, the code will make Emacs scan the temp directory and purge old backup files on startup.

```
(let ((week (* 60 60 24 7))
      (current (float-time (current-time))))
  (dolist (file (directory-files temporary-file-directory t))
    (when (and (backup-file-name-p file)
               (> (- current (float-time (fifth (file-attributes file))))
                  week))
      (message "%s" file)
      (delete-file file))))
```

3.1.9 Overwriting Text (Optional)

I'm used to having text being overwritten when highlighted. These always saves me a few keystrokes.

```
(delete-selection-mode +1)
```

3.1.10 Default Font (Optional)

Because I love mononoki.

```
(defvar emacs-english-font "mononoki Regular 14"
  "the font name of English.")

(defun font-exist-p (fontname)
  "Test if this font is exist or not.
  This function only work on GUI mode, on terminal it just
  return nil since you can't set font for emacs on it."
  (if (or (not fontname) (string= fontname "") (not (display-graphic-p)))
      nil
    (if (not (x-list-fonts fontname))
        nil t)))

(if (font-exist-p emacs-english-font)
    (setq default-frame-alist '((emacs-english-font))))
```

3.1.11 Tabs vs Spaces (Optional)

I'm a fan of the 2 spaces rule.

```
(setq-default tab-width 2)
(setq-default indent-tabs-mode nil)
```

3.2 Theming

There are a myraid of themes available for your picking. Here I list the better ones:

1. Zenburn
2. Solarized
3. Leuven (has an impressive org-mode theme)
4. Monokai
5. Tomorrow by Sanityinc
6. Darkorai

I'm currently using tao, a monochrome theme, with personal customizations for org-mode.

To enable a theme, find the relevant name of the theme on MELPA and add in the following snippet of code:

```
(use-package tao-theme
  :init
  (load-theme 'tao-yang t))
```

At this point you should have quite a hefty amount of modification done. Remember to save your configuration directory into version control.

4 Managing the Workspace

It's common to want to create new windows in your Emacs frame to maximize screen estate and make editing easier.

Key	Window
`C-x 0`	Delete current window
`C-x 1`	Maximize current window
`C-x 2`	Split current window *horizontally*
`C-x 3`	Split current window *vertically*

4.1 Winner-mode

Winner-mode is a global minor mode. When activated, it allows you to "undo" and "redo" changes in the window configuration.

Key	Action
C-c left	`winner-undo`
C-c right	`winner-redo`

The keybinding for switching between windows is `C-x o`, which I find overly complex for such an essential key.

4.2 WindMove

WindMove is a library included in Emacs starting with version 21. It lets you switch between windows using Shift + arrow keys. To activate it on startup, add the following piece of code in your `init.el`.

```
(when (fboundp 'windmove-default-keybindings)
  (windmove-default-keybindings))
```

4.3 Ace-window

ace-window lets you quickly switch between windows. It's the one I'm currently using, and I'm very happy with it.

```
(use-package ace-window
  :bind (("M-q" . ace-window)))
```

I'd bind it to `M-q`, or anything else you find convenient.

5 Thought-speed Motion

With a more usable Emacs configuration, we'll begin exploring how to navigate around Emacs, installing helper libraries where relevant.

I recommend printing this refcard, to be referred to when needed.

The first rule to moving around quickly is to **never leave the keyboard**. This concept is pervalent across all pertinent text editors, be it Vim or Emacs. In Emacs, key combinations are the gateway to text-editing nirvana.

5.1 Moving Across Lines

The most common line-movement operations are listed below.

Key	Movement	Emacs Function
`C-e`	End of line	`(end-of-line)`
`C-a`	Start of line	`(beginning-of-line)`
`M-m`	first non-whitespace of line	`(back-to-indentation)`

PROTIP: To check what a key combination is bound to, press `C-h k kbd`. Alternatively, `M-x describe-keybindings` lists all defined keys and their definitions in order of precedence.

5.2 Moving Within Visible Text

avy is a package for jumping to visible text using a char-based decision tree. Within three keystrokes, you're able to get to any visible point in the buffer.

```
(use-package avy
  :bind* (("C-'" . avy-goto-char)
          ("C-," . avy-goto-char-2)))
```

PROTIP: To jump back to your previous location, use `C-u C-space`.

5.3 Moving Within the Buffer

5.3.1 isearch

isearch is short for incremental search. On several occasions you find yourself wanting to move to a different location of the document, knowing the textual content in the area. You can move to the location using the *isearch*, bound to `C-s`. To move to the next matching search result, press `C-s` again. The search can also be performed in the reverse direction, and this is bound to `C-r`.

5.3.2 moccur

moccur is short for multi-occur. Some find this useful, but I personally feel like Swiper (introduced below) is sufficient for my day to day operations. The key benefit of *moccur* is that a buffer for search result matches is created, and this can be used to move to the matched locations again.

5.3.3 imenu

imenu is short for interactive menu. Imenu offers a way to find the major definitions in a file by name. For example, in an Emacs Lisp (`.el`) file, you can navigate around with imenu to variables, and function definitions. In org-mode, you can navigate to title headers with imenu. Because of its utility, I bind it to `M-i`.

```
(bind-key* "M-i" imenu)
```

To use `bind-key`, you need `use-package` installed. Skip this step if you intend to install counsel, described below.

5.4 Ivy, Counsel and Swiper

Ivy is a generic completion mechanism for Emacs. It aims to be smaller, simpler and more highly customizable.

Counsel provides a collection of Ivy-enhanced versions of command Emacs commands, including `find-file`, `describe-function` and `M-x`.

Swiper, the ivy-enhanced version of isearch.

```
(use-package counsel)
(use-package swiper
  :bind*
  (("C-s" . swiper)
   ("C-c C-r" . ivy-resume)
   ("M-a" . counsel-M-x)
   ("C-x C-f" . counsel-find-file)
   ("C-c h f" . counsel-describe-function)
   ("C-c h v" . counsel-describe-variable)
   ("C-c i u" . counsel-unicode-char)
   ("M-i" . counsel-imenu)
   ("C-c g" . counsel-git)
   ("C-c j" . counsel-git-grep)
   ("C-c k" . counsel-ag)
   ("C-c l" . scounsel-locate))
  :config
  (progn
    (ivy-mode 1)
    (setq ivy-use-virtual-buffers t)
```

```
20    (define-key read-expression-map (kbd "C-r") #'counsel-expression-history)
21    (ivy-set-actions
22     'counsel-find-file
23     '(("d" (lambda (x) (delete-file (expand-file-name x)))
24        "delete"
25        )))
26    (ivy-set-actions
27     'ivy-switch-buffer
28     '(("k"
29        (lambda (x)
30          (kill-buffer x)
31          (ivy--reset-state ivy-last))
32        "kill")
33       ("j"
34        ivy--switch-buffer-other-window-action
35        "other window")))))
```

For a powerful preconfigured alternative, consider helm and its companion tutorial here. For something like Swiper, look at helm-swoop.

For a simpler in-built alternative, look at `ido-mode`, Mickey Petersen has a great write-up about it here.

6 Thought-speed Editing

6.1 Moving Text Around

Earlier, I introduced the terminology Emacs uses for its clipboard system. I missed one vital piece, because I felt it was more appropriate to introduce here to keep things fresh.

Text that gets killed is erased, and then stored inside the **kill ring**. This stored text is then retrievable by **yanking**. There is only one kill ring, global to Emacs.

A clear distinction has to be made between killing and deleting. Deleting text removes it from the buffer, but does not store it in the kill ring. Therefore extra caution has to be made when performing deletions.

6.1.1 Deleting Text

Here are the more useful text deletion commands:

Key	Action	Function
`M-\`	Delete spaces and tabs around point	`(delete-horizontal-space)`
`M-SPC`	Delete spaces and tabs around point, leaving one space	`(just-one-space)`
`C-x C-o`	Delete blank lines around current line	`(delete-blank-lines)`
`M-^`	Join two lines by deleting intervening newline, along with indentation	`(delete-indentation)`

6.1.2 Killing Text

Here are the more useful text killing commands:

Key	Action	Function
`C-k`	Kill line	`(kill-line)`
`C-w`	Kill region	`(kill-region)`
`C-x DEL`	Kill back to beginning of sentence	`(backward-kill-sentence)`
`M-k`	Kill to end of sentence	`(kill-sentence)`

NOTE: By default, a sentence is delimited by a period, followed by **two** spaces. This is so that Emacs can differentiate between abbrieviations (M. J. for example), and actual sentences. It is recommended that you follow the two space convention, but if you insist, `(setq sentence-end-double-space nil)` should do the trick.

6.1.3 Yanking Text

Here are the more useful text killing commands:

Key	Action	Function
`C-y`	Yank last killed text	`(yank)`
`M-y`	Replace last killed text with an earlier batch of killed text	`(yank-pop)`
`M-w`	Save region as last killed text without performing the kill	`(kill-ring-save)`
`C-M-w`	Append next kill to last batch of killed text	`(append-next-kill)`

You can think of the kill ring as a stack, so you could continuously pop the kill ring to obtain earlier batches of killed text.

6.1.4 browse-kill-ring

I often defer to browse-kill-ring to access my kill-ring history. I bind it to `M-y`, replacing `(yank-pop)`. Try it out, and see if it suits your workflow.

```
(use-package browse-kill-ring
  :bind ("M-y" . browse-kill-ring))
```

6.2 Zap-to-char

As an ex-vim user, I miss the `ct` and `dt` key dearly. Fret not, for what vim can do, emacs can do better.

`zap-up-to-char` does exactly what it says it does: it kills up to, but not including the ARGth occurrence of CHAR.

```
(autoload 'zap-up-to-char "misc"
  "Kill up to, but not including ARGth occurrence of CHAR."
  'interactive)

(bind-key* "M-z" 'zap-up-to-char)
```

Let's play with some examples:

> I think I love to eat pancakes and bananas.

We begin from the start of the sentence. Now let's say we want to kill up to "think", I'd do `M-z t RET`. If I wanted to kill up to "to", then I provide an argument value of 2 to `zap-up-to-char` by pressing `M-2 M-z t RET`.

Remember that the text is *killed*, which means it gets saved into the kill ring and can be retrieved at a later point in time through yanking.

7 Project Management

In most cases, your work is not limited to a single file. Instead, it's comprised of multiple files residing in a parent directory, or perhaps even version-controlled with Git or the likes.

While Emacs does not ship with project management tooling, there are a few quality libraries that help you with that.

7.1 FFIP

find-file-in-project, or *ffip* in short, provides quick access to files in a directory managed by version-control (git/svn/mecurial). It's intentionally kept simple. It uses GNU find under the hood, which makes it suitable even for large codebases. The default interface has been recently changed to *ivy* (introduced earlier). Look no further than ffip for a simple project-management tool.

```
(use-package find-file-in-project
  :bind (("s-f" . find-file-in-project)
         ("s-F". find-file-in-current-directory)
         ("M-s-f" . find-file-in-project-by-selected)))
```

The functions are so useful they deserve a short keybinding: `s-f` is what I'd go with.

7.2 Projectile

Projectile is a different beast, leveraging a variety of tools to be a performant project interaction library. While ffip aims to be a minimalistic and fast file-switcher for projects, projectile aims to be the all-encompassing project-management tool. It has certainly proved to be the only one you'll need.

Here are some handpicked features Projectile has to offer, as seen on the Github page:

- jump to a file in project
- jump to files at point in project
- jump to a project buffer
- jump to a test in project
- toggle between files with same names but different extensions (e.g. .h <-> .c/.cpp, Gemfile <-> Gemfile.lock)
- toggle between code and its test (e.g. main.service.js <-> main.service.spec.js)
- switch between projects you have worked on
- replace in project
- regenerate project etags or gtags (requires ggtags).
- run `make` in a project with a single key chord

I bind the projectile keymap to `C-x p`. If you use `ivy`, set the `projectile-completion-system` to `ivy`, and install counsel-projectile, which adds more ivy-friendly functions for projectile.

```
(use-package projectile
  :demand t
  :init (projectile-global-mode 1)
  :bind-keymap* ("C-x p" . projectile-command-map)
  :config
  (require 'projectile)
  (use-package counsel-projectile
    :bind (("s-p" . counsel-projectile)
           ("s-f" . counsel-projectile-find-file)
           ("s-b" . counsel-projectile-switch-to-buffer)))
  (setq projectile-use-git-grep t)
  (setq projectile-completion-system 'ivy))
```

Projectile also has a little known feature, called `projectile-commander`. The default action when switching projects is to perform a `find-file`, but that's not what you want most of the time. Give yourself a choice to choose between doing a `find-file`, a git-fetch, or even language specific things like starting a REPL.

First, set the command to `projectile-commander`:

```
(setq projectile-switch-project-action #'projectile-commander)
```

Then define the methods you want:

```
(def-projectile-commander-method ?s
  "Open a *eshell* buffer for the project."
  (projectile-run-eshell))
(def-projectile-commander-method ?c
  "Run 'compile' in the project."
  (projectile-compile-project nil))
(def-projectile-commander-method ?\C-?
  "Go back to project selection."
  (projectile-switch-project))
(def-projectile-commander-method ?d
  "Open project root in dired."
  (projectile-dired))
(def-projectile-commander-method ?F
  "Git fetch."
  (magit-status)
  (call-interactively #'magit-fetch-current))
(def-projectile-commander-method ?j
  "Jack-in."
  (let* ((opts (projectile-current-project-files))
         (file (ido-completing-read
                "Find file: "
                opts
                nil nil nil nil
                (car (cl-member-if
                      (lambda (f)
                        (string-match "core\\.clj\\'" f))
                      opts)))))
```

```
    (find-file (expand-file-name
                file (projectile-project-root)))
    (run-hooks 'projectile-find-file-hook)
    (cider-jack-in))))
```

Append all these code into `:config` for the `projectile` package.

7.3 Using Ag or Grep

Projectile ships with functions that make use of grep and ag. Grep and Ag are both command-line tools used for searching code. You use `projectile-ag` (`C-x p s s`) or `projectile-grep` (`C-x p s g`) to perform a project-scoped search, and use the search results to navigate to the relevant locations. Ag is more performant, but does not come installed with most systems. In most cases, grep is sufficiently fast.

Alternatively, if you had installed counsel by following the instructions here, you'd have access to the function, `counsel-ag`, `counsel-git`, and `counsel-git-grep`. `counsel-git-grep` (`C-c j`) is especially great for projects, because it prunes out files captured by `.git-ignore`.

7.4 Magit

Magit is an interface for Git. It is an absolute joy to use, and is one of the main reasons I stuck with Emacs after a period with Vim.

```
(use-package magit
  :bind (("C-x g" . magit-status)
         ("C-x M-g" . magit-blame))
  :init (setq magit-auto-revert-mode nil)
  :config (add-hook 'magit-mode-hook 'hl-line-mode))
```

Surely you can figure out the basics like adding remotes, fetching, and committing with such a simplified interface. Here's a great tutorial on how to perform rebases, squashes and the like easily with Magit.

8 Icing on the Cake

Here I introduce packages I've installed that are not a must, but are definitely nice to have.

8.1 golden-ratio

Give the window you're working in more screen estate.

```
(use-package golden-ratio
  :diminish golden-ratio-mode
  :config (progn
            ;;(add-to-list 'golden-ratio-extra-commands 'ace-window)
            (golden-ratio-mode 1)))
```

If you're using ace-window, uncomment the line for golden-ratio to work properly.

8.2 aggressive-indent

Keep your code nicely aligned while you hack away at the more important stuff. Remember to disable this for languages that depend on indentation for syntax, like Python.

```
(use-package aggressive-indent
  :diminish aggressive-indent-mode
  :config (add-hook 'prog-mode-hook 'aggressive-indent-mode))
```

8.3 which-key

Which-key is a godsend when you're first starting out using Emacs. I still refer to the list of keybindings it shows from time to time.

```
(use-package which-key
  :diminish which-key-mode
  :config (add-hook 'after-init-hook 'which-key-mode))
```

8.4 volatile-highlights

Volatile-highlights provides visual feedback for operations such as yanking by highlighting the relevant regions.

```
(use-package volatile-highlights
  :defer 5
```

```
  :diminish volatile-highlights-mode
  :config (volatile-highlights-mode t))
```

8.5 firestarter

firestarter lets you execute commands (including shell commands) on save. Example use cases include compiling SASS files, and compiling a program.

```
(use-package firestarter
  :bind ("C-c m s" . firestarter-mode)
  :init (put 'firestarter 'safe-local-variable 'identity))
```

8.6 git-gutter+

I use git-gutter+ primarily for showing on the left side what parts of my files have changed. It also has additional features like staging hunks for commits, but I use Magit for that. You can take a look at the Github page for more details.

```
(use-package git-gutter+
  :init (global-git-gutter+-mode)
  :diminish git-gutter+-mode
  :defer 5
  :config (progn
            (setq git-gutter+-modified-sign "==")
            (setq git-gutter+-added-sign "++")
            (setq git-gutter+-deleted-sign "--")))
```

8.7 Honourable Mentions

1. hydra
2. electric-align