JETHRO KUAN

MODERN EMACS

Contents

Preface

LaTeX 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*.

Emacs is complex, and will seem difficult to grok. It takes a certain amount of courage, or an equal amount of frustration, to turn to Emacs.

This material was originally titled *Hacking Emacs*, because a great deal of customization can and will be made. The tinkerer will marvel at the masterpiece which is Emacs. We hack on it because we can; every facet of Emacs is extensible and customizable.

I have since renamed this piece to "Modern Emacs". Emacs has evolved over several decades, and it is important that our tools are kept up to date. I have handpicked several packages to introduce. These will greatly boost your productivity with the least amount of learning time.

For the novice, this will provide you with a sane environment for you to play around and do some self-exploration. For the first time you'll own your editor: Emacs should designed specifically for your own needs.

For the intermediate, this presents itself as a literate org-mode configuration. I hope you find something useful in here you could copy.

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

– Jethro Kuan LaTeX

Introduction

Installing Emacs

If you're on a Linux distribution, Emacs should be available through your respective package managers.

For OSX users, I recommend Emacs for OSX.

Refer here if you have further doubts.

This text assumes a recent Emacs version (of version >24.0). You can check your Emacs version with `M-x emacs-version`.

Terminology

Emacs is aged piece of software, so a lot of the terms it uses often confound first-timers. It is important to speak a common language, hence we begin by introducing the necessary terminology.

Windows, Frames and Buffers

The text you are editing in Emacs resides in an object called the **buffer**. It is important to note that a buffer is not a file. Buffers can contain more than just file contents: it could be an email draft, the compile error message, the list goes on. When you open a file, the file contents get loaded into a new buffer. When a buffer is saved, the buffer contents are written into the corresponding file.

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

A **frame** is a container for windows. One can think of it as the window configuration for Emacs.

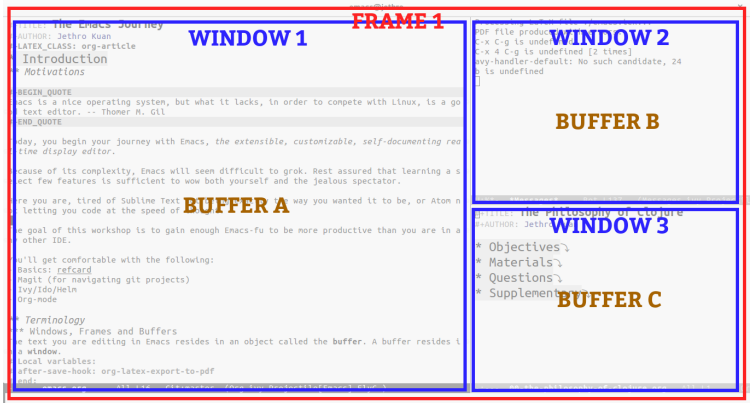

Figure 1: A pictorial representation of windows, frames and buffers. In this picture, there are 1 frame, 3 windows and 3 buffers.

Killing, Yanking and the CUA

The table below shows analogous terminologies between Emacs clipboard system and modern standards:

Modern	Emacs
Cut	Kill
Paste	Yank
Copy	Save To Kill Ring

The Emacs terminology had been 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.

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)
2. Indentation
3. Language-specific keybindings (for refactoring etc.)

In the scenario where the 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 extensions, adding functionality to a buffer. These are optional, and can be added locally on a per buffer basis, or globally affecting all buffers.

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

Modeline

The bottom bar in Emacs is called the **modeline**. Each window has a modeline, which presents useful information about the buffer it displays. The first item in brackets is 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

Emacs Lisp

Keybindings are combinations of keys that invoke functions when pressed. The origins of these functions can be one of three:

1. It could come out-of-the-box
2. It could be provided by an installed package
3. It could be self-written

These functions are defined in a language called **Emacs Lisp**, abbreviated as **Elisp**. Emacs Lisp files typically have the file extension `.el`.

The full list of functions are browsable and invokable with `M-x`.

Keybindings

What is `M-x`, you may ask? This is how keybindings are discussed in Emacs. In `M-x`, `M` stands for the Meta key. This key is mapped to `alt` in most systems and keyboards, but this could vary. The next important key is `Ctrl`, abbreviated as `C`. For example `C-w` on a highlighted region kills the region of text. That would be pressing `Ctrl-w`.

Because of the sheer number of functions Emacs has and could have, Many functions are invoked via more complex key combinations. Emacs does try to keep things mnemonic. For example, `C-h` is not a keybinding in itself, but opens a gateway to all the help functions. `C-h k` invokes `describe-key`, which brings up documentation for the function invoked by the next key combination. With `C-h k`, you press `Ctrl` and `h`, let go of `Ctrl` and press `k`. On the other hand, `C-h C-a` invokes `about-emacs`, which brings up the first screen that greets you on your new Emacs install. To invoke `C-h C-a`, hold `Ctrl` throughout, and press `h` then `a`.

Summary

- Emacs has it's own terminology, some of which we have covered in detail: windows, frames, buffers, killing and cutting.
- Most functions in Emacs are written in Emacs Lisp, or Elisp. These functions are invokable with `M-x`.
- Key combinations in Emacs are bound to invokable functions, and their complexities may vary.

Armed with this new knowledge, you can begin exploring Emacs.

The Absolute Beginners Guide

It is important that I be careful to impart Emacs knowledge objectively. Each individual has differing personal development practices, and Emacs is to be customized to suit one's needs. It is therefore paramount that you absorb this material with a different mindset.

Before the deep dive begins, I'd like you to create a blank text file. This file will contain questions that arise while going through the material. For example, one might think: I have this feature, multiple-cursors in Sublime Text, how do I do the same (or better) in Emacs? Or how do I compile my C++ files directly in Emacs?

There many different paths to Emacs mastery, but most include the following steps:

1. Muscle memory. Removing the conscious effort in figuring out which key combinations to press, allows clarity of thought, and empowered by Emacs allows you to code/write at unimaginable speeds.
2. Aggressive Optimization. Be on the constant lookout for things you perform often in your daily work, and ensure these are easily accessible with key combinations. If functions are yet available for your purposes, write one.

The keybindings that you'll come across in this material are personal preference, and are only for your reference. Take note of keybindings that you dislike, and note them down to be replaced later.

Baby Steps

TBC

Taming the Beast

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

We perform tweaking on Emacs minutiae, and simultaneously learn how Emacs is customized.

Customizing Emacs

During initialization, Emacs attempts to load an *init* file. This *init* file is an Emacs Lisp file, and is processed top-down.

> TIP: To load Emacs without a configuration file, run `emacs` with the command-line switch `-q`. This is useful when your configuration file breaks.

Emacs searches for *init* files in several locations:

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

I recommend the third option, `~/.emacs.d/init.el`. Having a dedicated `.emacs.d` folder for all Emacs-related config simplifies versioning.

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

Next, I'll introduce your first few tweaks. To enable them, copy them into the `init.el` file.

Enabling Package Archives

MELPA is the de-facto community package archive for Emacs. Because it does not ship with Emacs, we add it to `package-archives`. Similarly, 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.

Setting User Details

These variables are used in several parts of Emacs, including email.

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

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

Use-package

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

```
(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 missing packages will be installed automatically.

y/n

This is mostly personal preference, but I find y/n easier to type than yes/no.

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

Custom Files

Emacs comes with a built-in interface to customize all parts of Emacs.

To persist the changes you've made, Emacs saves them (by default) into your `init.el` file, which can cause it to grow out of control.

I like to keep such customizations saved in a separate file.

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

Try to put this at the bottom of your `init.el` file, so it overrides any configuration you might have added in your `init.el` file.

Backup Files

Backup files are important, but they litter your directories with temporary files. I movs 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)))
```

Overwriting Text

I'm used to having text being overwritten when highlighted. This usually saves me a keystroke.

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

Default Font

Emacs looks for fonts installed in your file system. There are several ways to go about this, but I find the following the simplest and most reliable.

```
(add-to-list 'default-frame-alist
             '(font . "Fira Code-12"))
```

Tabs vs Spaces

I choose to set my tabs to 2 spaces, an increasingly common trend.

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

Theming

There are a myriad 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
7. Ample

Usually, themes are named `foo-theme` in MELPA.

```
(use-package ample-theme
  :init
  (load-theme 'ample-flat t))
```

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

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*

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.

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))
```

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.

Thought-speed Motion

With a more usable Emacs configuration, we'll begin navigating around Emacs, installing helper libraries where relevant.

I recommend printing this refcard, and refer to it when necessary.

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

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.

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`.

Moving Within the Buffer

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`.

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.

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.

Registers

> NOTE: Before I begin, note that while I introduce registers here, registers are not just for moving around the buffer.

Registers are compartments where text, rectangles, positions, window configurations and many more can be stored. Think of it as a temporal bookmarking system; these registers get wiped at the end of the Emacs process. Each register is denoted by a single character (eg. `?r` or `?1`). The register `?a` is different from the register `?A`.

Whatever you store inside a register persists until it is overwritten by something else, or until the Emacs process is killed.

Store a file in a register is simple:

```
1 (set-register r '(file . name))
```

Do this for all your bookmarks, and you can quickly jump to them with `C-x r reg`.

To make things simpler, bind `jump-to-register` to a more accessible key:

```
1 (bind-key* "C-o" 'jump-to-register)
```

Putting these in your `init.el` file ensures that they will always be available. I encourage you to play around with the other forms of registers.

Bookmarks

Bookmarks are similar to registers, but they are persisted in a file.

To create a bookmark, type `C-x r m bookmark-name`. Similarly, bind `bookmark-jump` to a more accessible key:

```
1 (bind-key* "C-o" 'bookmark-jump)
```

To change the file in which you store your bookmarks, invoke `M-x customize-variable bookmark-default-file`.

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" . counsel-locate))
  :config
  (progn
    (ivy-mode 1)
    (setq ivy-use-virtual-buffers t)
    (define-key read-expression-map (kbd "C-r") #'counsel-expression-history)
    (ivy-set-actions
     'counsel-find-file
     '(("d" (lambda (x) (delete-file (expand-file-name x)))
        "delete"
        )))
    (ivy-set-actions
     'ivy-switch-buffer
     '(("k"
        (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.

Thought-speed Editing

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.

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)`

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 abbrievi-ations (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.

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.

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))
```

Selecting Regions

The Mark, the Point and the Region

Many Emacs commands operate on an arbituary, contiguous part of the buffer, also known as the *region*. A region is delimited by two objects: the *mark* and the *point*.

The point is where your cursor (keyboard) is currently placed. `C-SPC` creates a mark at the current position of point, activating it, as well as activating the region. The region is the text between the point and the mark, regardless of which direction. To deactivate the mark, simply quit with `C-g`.

Some common region commands include `kill-region`.

Rectangular Region

Rectangle commands operate on retangular areas of text. This may seem rather esoteric, but it occasionally presents itself as the correct tool.

`C-x SPC` creates a rectangular mark. The following presents commonly used rectangular commands, all prefixed with `C-x r`, operate on rectangular regions:

(Region-rectangle is short-formed as RR)

Key	Action	Function
`C-x r k`	Kill text in RR, saving its contents into last-killed rectangle	`(kill-rectangle)`
`C-x r M-w`	Save text in RR into kill-ring	`(copy-rectangle-as-kill)`
`C-x r d`	Delete text in RR	`(delete-rectangle)`
`C-x r y`	Yank last killed rectangle	`(yank-rectangle)`
`C-x r c`	Clear RR, replacing all text with spaces	`(clear-rectangle)`
`C-x r t string RET`	Replace rectangle contents with string on each line	`(string-rectangle)`

Expand Region

expand-region is one of those packages that you can live without, but as you use it more often, you find yourself repeatedly going back to it. Here's a great overview of expand-region.

```
(use-package expand-region
  :bind (("C-=" . er/expand-region)))
```

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.

If you use avy, perhaps you'll find zzz-to-char to your liking. It uses the avy interface to select which letter to zap up till.

```
(use-package zzz-to-char
  :bind (("M-z" . zzz-up-to-char)))
```

Multiple-cursors

Multiple cursors would be familiar functionality to Sublime Text users. It's the perfect tool for many things, including editing variable names with visual feedback.

```
(use-package multiple-cursors
  :bind (("C->" . mc/mark-next-like-this)
         ("C-<" . mc/mark-previous-like-this)
         ("C-c C-<" . mc/mark-all-like-this)))
```

I use it in conjunction with `expand-region`: `expand-region` to select the keyword (variable names, for example), and use `C-c C-<` to select all instances of the variable, and simply type over it.

Templating

yasnippet is a templating system, allowing you to type an abbrieviation and automatically expand it into function templates with `<TAB>`. This feature is similar to the one offered by Textmate; in fact, the templating language is inherited from it.

```
(use-package yasnippet
  :diminish yas-global-mode yas-minor-mode
  :defer 5
  :init (add-hook 'after-init-hook 'yas-global-mode)
  :config (setq yas-snippet-dirs '("~/.emacs.d/snippets/")))
```

Andrea Crotti maintains an official repo for yasnippet templates. It supports many languages and major-modes. I recommend forking the repository – as I did – and cloning it as a git submodule under `~/.emacs.d`: this way you can add your own templates and version

control them. I had set the `yas-snippet-dirs` to `~/.emacs.d/snippets`, so following that configuration:

```
git submodule add git@github.com:foobar/snippets.git ~/.emacs.d/snippets
```

Autocompletion

Text completion in Emacs has Emacs users split between two major factions: autocomplete and company-mode. Both have similar feature sets, but it is generally argued that company-mode is more feature-rich.

The following snippet installs company-mode.

```
(use-package company
  :defer 5
  :diminish company-mode
  :init (progn
          (add-hook 'after-init-hook 'global-company-mode)
          (setq company-dabbrev-ignore-case nil
                company-dabbrev-code-ignore-case nil
                company-dabbrev-downcase nil
                company-idle-delay 0
                company-begin-commands '(self-insert-command)
                company-transformers '(company-sort-by-occurrence))
          (use-package company-quickhelp
            :config (company-quickhelp-mode 1))))
```

One thing that people miss from autocomplete is documentation popups. We add that functionality with company-quickhelp. Another notable setting made was to set the delay for autocomplete to `0`. Play around with the numbers and see what you're comfortable with.

Note that company-mode is merely a framework for autocompletion. To enable autocompletion for various languages, you'd need to install various company backends.

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.

FFIP

find-file-in-project, or *ffip* in short, provides quick access to files in a directory managed by version-control (git/svn/mercurial). 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.

Projectile

Projectile is a different beast, leveraging a variety of tools to be a performant project interaction library. While ffip aims to be a minimal-

istic 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: `projectile-commander`. The default action upon switching projects is `find-file`, and that might not be desirable. Give yourself a choice between doing a `find-file`, a git-fetch, or even language specific things like starting a REPL.

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

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

Next, 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.

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 `.gitignore`.

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.

Icing on the Cake

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

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 function properly.

aggressive-indent

Keep your code nicely aligned while you hack away at 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))
```

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))
```

volatile-highlights

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

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

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))
```

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 "--")))
```

Honourable Mentions

1. hydra
2. electric-align

Miscellaneous Goodies

Minimizing Startup Time

Emacs Daemon

One way to avoid all perceived boot time, is to start emacs during the system boot as a daemon. This is the option I have gone with.

All you need to do add the Emacs daemon to auto-start:

```
emacs --daemon
```

I use *systemd* instead. For this option, create a user *systemd* service file with the following content:

To enable the service on startup, just do `systemd enable --user emacsd.service`.

Profiling with esup

Esup can perform a profile of your current configuration. Esup is obtainable over MELPA, so go ahead and use `use-package` like we normally do:

```
(use-package esup
  :defer t)
```

Next, just execute esup with `M-x esup`. A separate emacs process

will start, and the profiling results will be returned in a separate window, which would look like this:

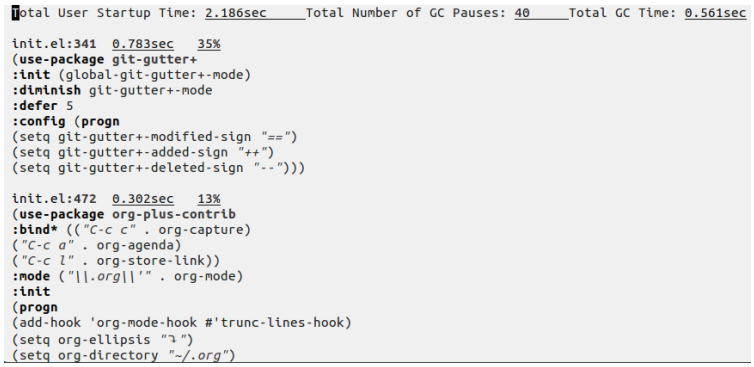

Figure 3: esup profiling results

You can take note of the packages that take the longest amount of time to initialize, and weigh their alternatives. For example, I might contemplate removing git-gutter+ or replacing it with a more lightweight alternative.

Micro-optimizations with keyfreq

Once you've determined your Emacs style, you begin to use the same keybindings again and again. Understanding which functions are invoked most often, allows you to optimize your workflow by binding them to shorter, more accessible key combinations.

keyfreq monitors your keypresses during daily Emacs usage.

```
(use-package keyfreq
  :config
  (keyfreq-mode 1)
  (keyfreq-autosave-mode 1))
```

After a period of Emacs usage, run `M-x keyfreq-show`. Perform micro-optimizations based on the results. After which, `M-x keyfreq-reset` to rinse and repeat.

Remapping Capslock

Without doubt, the capslock key is one of the most underused keys (that is if you're normal). Do yourself a favour and remap it to something more useful – I recommend the elusive Ctrl key. On the Mac, you can remap it in the keyboard preferences pane.

Ergonomic Keybindings

Anyone who spends enough time at the keyboard is at risk of repetitive strain injury, or RSI. Spend some time reflecting on how often you stretch out your pinky or move your thumb into awkward positions to press a key.

M-x

Let's face it. `M-x` is actually pretty damn hard to hit. *x* is highly inaccessible, not even on alternative keyboard layouts like Dvorak.

In prelude, `M-x` is bound to `C-x C-m`. I bind `M-x` to `M-a`, which is originally bound to `backward-sentence`, which I don't use much at all. Both are viable options.

ergoemacs

ergoemacs was developed to bring familiar keys to Emacs, as well as to reduce risk of RSI. While you could follow the instructions on the webpage to have it installed, I recommend studying their keybindings and adopting the ones you think you'll like.