

My Emacs Initialisation File, Written in Org-mode

Musa Al-hassy

2018-07-25

Contents

1	Introduction	3
2	What’s in, or at the top of, my ~/.emacs	3
3	Version Control	4
4	Managing Local Variables	4
5	Loads	4
5.1	Programming Language Supports	4
5.2	package-initialize: Melpa, gnu, and org	5
6	Cosmetics	5
6.1	Flashing when something goes wrong	5
6.2	My todo list: The initial buffer when Emacs opens up	5
6.3	Showing date, time, and battery life	5
6.4	Minibuffer should display line and column numbers	5
6.5	Highlight parenthesis pair when cursor is near ;-)	5
6.6	Increase/decrease text size	5
6.7	Delete Selection mode	6
6.8	Ido Mode	6
7	Helpful Functions & Shortcuts	6
7.1	Bind recompile to C-c C-m – “m” for “m”ake	6
7.2	Reload buffer with f5	6
7.3	Kill to start of line	6
7.4	file-as-list and file-as-string	6
7.5	kill-other-buffers	7
7.6	create-scratch-buffer	7
7.7	Switching from 2 horizontal windows to 2 vertical windows	7
7.8	Making then opening html’s from org’s	7
7.9	re-replace-in-file	8
7.10	Obtaining Values of #+KEYWORD Annotations	8
8	Spelling	8
9	Org-mode related things	9
9.1	ox-extra: Using :ignore: to ignore headings but use the bodies	9
9.2	Executing code from src blocks	9
9.3	<X Completion	10
9.3.1	Demoing Dot Graphs	10
9.3.2	<E for emacs-lisp source blocks	10

9.4	org-mode header generation	10
9.5	Org-mode cosmetics	11
10	Other fun things	11
11	Summary of Utilities Provided	12

Abstract

Herein I document the configurations I utilise with Emacs. Of note are:

This is a literate programming setup.

I have a variety of cosmetics such as showing battery life and flashing upon errors.

Production of org-mode ready-to-go skeletons.

Utilities for working with org-mode files, namely `#+KEYWORD: VALUE` pairs.

This is a literate program file.

```
;;  
;; This header file is used to create the articles for: https://alhassy.github.io/blog/  
;; This file is generated from the literate file 'AlBasmala.org' in the same repo.  
;; Musa Al-hassy, 2018  
;;
```

1 Introduction

Why not keep Emacs's configurations in the `~/.emacs` file? This is because the Emacs system may explicitly add, or alter, code in it.

For example, execute the following

1. M-x `customize-variable` RET `line-number-mode` RET
2. Then press: `toggle`, `state`, then 1.
3. Now take a look: `(find-file "~/.emacs")`

Notice how additions to the file have been created by 'custom'.

As such, I've chosen to write my Emacs' initialisation configurations in a file named `~/.emacs.d/init.org`: I have a literate configuration which is then loaded using org-mode's tangling feature.

2 What's in, or at the top of, my `~/.emacs`

We evaluate every piece of emacs-lisp code available here when Emacs starts up by placing the following at the top of our `.emacs` file:

```
(org-babel-load-file "~/.emacs.d/init.org")  
;;  
;; My Emacs settings: (find-file "~/.emacs.d/init.org")
```

(I do not generate my `.emacs` file from this source code in-fear of overriding functionality inserted by `custom`.)

Our `.emacs` should be byte-compiled so that when we start Emacs it will automatically determine if the `init.org` file has changed and if so it would tangle it producing the `init.el` file which will then be loaded immediately.

```
;; In-case I forget to byte-compile!  
(byte-compile-file "~/.emacs")
```

```
;; Change this silly counter to visually notice a change.  
;; (progn (message "Init.org contents loaded! Counter: 7") (sleep-for 3))
```

3 Version Control

Soft links are pointers to other filenames, whereas hardlinks are pointers to memory location of a given filename!

Soft links are preferably since they defer to the original filename and can work across servers.

We can declare them as follows,

```
ln -s source_file myfile
```

If `repo` refers to a directory under version control –or Dropbox– we move our `init` file and `emacs` directory to it, then make soft links to these locations so that whenever `~/.emacs` is accessed it will refer to `repo/.emacs` and likewise for `.emacs.d` :-)

On a new machine, copy-paste any existing emacs configs we want to the `repo` folder then `rm -rf ~/.emacs*` and then make the soft links only.

```
repo=~/.Dropbox      ## or my git repository: ~/dotfiles
```

```
cd ~
```

```
mv .emacs $repo/
```

```
ln -s $repo/.emacs .emacs
```

```
mv .emacs.elc $repo/
```

```
ln -s $repo/.emacs.elc .emacs.elc
```

```
mv .emacs.d/ $repo/
```

```
ln -s $repo/.emacs.d/ .emacs.d
```

Note the extra / after `.emacs.d`!

You may need to unlink softlinks if you already have them; e.g., `unlink .emacs.d`.

To make another softlink to a file, say in a blogging directory, we `cd` to the location of interest then execute, say: `ln -s $repo/.emacs.d/init.org init.org`

4 Managing Local Variables

It is dangerous to load a file with local variables; instead we should load files without evaluating locals, read the locals to ensure they are safe –e.g., there’s nothing malicious like `eval: (delete-file your-important-file.txt)`– then revert the buffer to load the locals.

However, when preprocessing my own files I sometimes wish to accept all locals without being queried and so have these two combinators.

```
;; Accept all local variables versus query for possibly non-safe locals.
```

```
(defun DANGER-all-locals () (setq enable-local-variables :all))
```

```
(defun SAFE-query-locals () (setq enable-local-variables t))
```

5 Loads

5.1 Programming Language Supports

```
(load (shell-command-to-string "agda-mode locate"))
```

```
;;
```

```
;; Seeing: One way to avoid seeing this warning is to make sure that agda2-include-dirs is not bound.
```

```
; (makunbound 'agda2-include-dirs)
```

```
;; Open .v files with Proof General’s Coq mode
```

```
(load "~/emacs.d/lisp/PG/generic/proof-site")
```

5.2 package-initialize: Melpa, gnu, and org

- **M-x list-packages** to see all melpa packages that can install
 - Not in alphabetical order, so maybe search with **C-s**.
- For example to download the haskell mode: **M-x package-install RET haskell-mode RET**.
 - Or maybe to install **unicode-fonts** ;-)
- Read more at http://ergoemacs.org/emacs/emacs_package_system.html or at <https://github.com/milkypostm/melpa>

```
(require 'package)
(setq package-archives
  '(("melpa" . "https://melpa.org/packages/")
    ("gnu" . "https://elpa.gnu.org/packages/")
    ("org" . "http://orgmode.org/elpa/")))
(package-initialize)
```

6 Cosmetics

6.1 Flashing when something goes wrong

Make top and bottom of screen flash when something unexpected happens thereby observing a warning message in the minibuffer. E.g., **C-g**, or calling an unbound key sequence, or misspelling a word.

```
(setq visible-bell 1)
;; Enable flashing mode-line on errors
```

6.2 My todo list: The initial buffer when Emacs opens up

```
(setq initial-buffer-choice "~/Dropbox/todo.org")
```

6.3 Showing date, time, and battery life

```
(setq display-time-day-and-date t)
(display-time)
(display-battery-mode 1)
```

6.4 Minibuffer should display line and column numbers

```
(line-number-mode 1)
(column-number-mode 1)
```

6.5 Highlight parenthesis pair when cursor is near ;-)

```
(load-library "paren")
(show-paren-mode 1)
(transient-mark-mode t)
(require 'paren)
```

6.6 Increase/decrease text size

```
(global-set-key (kbd "C-+") 'text-scale-increase)
(global-set-key (kbd "C--") 'text-scale-decrease)
;; C-x C-0 restores the default font size
```

6.7 Delete Selection mode

Delete Selection mode lets you treat an Emacs region much like a typical text selection outside of Emacs: You can replace the active region. We can delete selected text just by hitting the backspace key.

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

6.8 Ido Mode

Ido, “interactively do things”, mode is used for most commands that require you to select something from a list: It provides possible completions.

- An alternative is a third-party tool: Helm or ivy.

Extremely helpful for when switching between buffers, C-x C-b. Try and be grateful.

```
(ido-mode t)
```

7 Helpful Functions & Shortcuts

Here is a collection of Emacs-lisp functions that I have come to use in other files.

7.1 Bind recompile to C-c C-m – “m” for “m”ake

```
(defvar my-keys-minor-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "C-c C-m") 'recompile)
    map)
  "my-keys-minor-mode keymap.")

(define-minor-mode my-keys-minor-mode
  "A minor mode so that my key settings override annoying major modes."
  :init-value t
  :lighter " my-keys")
```

7.2 Reload buffer with f5

I do this so often it’s not even funny.

```
(global-set-key [f5] '(lambda () (interactive) (revert-buffer nil t nil)))
```

7.3 Kill to start of line

Dual to C-k,

```
;; M-k kills to the left
(global-set-key "\M-k" '(lambda () (interactive) (kill-line 0)) )
```

7.4 file-as-list and file-as-string

```
(defun file-as-list (filename)
  "Return the contents of FILENAME as a list of lines"
  (with-temp-buffer
    (insert-file-contents filename)
    (split-string (buffer-string)))))
```

```
(defun file-as-string (filename)
```

```
"Return the contents of FILENAME as a list of lines"
(with-temp-buffer
  (insert-file-contents filename)
  (buffer-string)))
```

7.5 kill-other-buffers

```
(defun kill-other-buffers ()
  "Kill all other buffers."
  (interactive)
  (mapc 'kill-buffer (delq (current-buffer) (buffer-list))))
```

7.6 create-scratch-buffer

```
;; A very simple function to recreate the scratch buffer:
;; ( http://emacswiki.org/emacs/RecreateScratchBuffer )
(defun create-scratch-buffer nil
  "create a scratch buffer"
  (interactive)
  (switch-to-buffer (get-buffer-create "*scratch*"))
  (lisp-interaction-mode))
```

7.7 Switching from 2 horizontal windows to 2 vertical windows

I often find myself switching from a horizontal view of two windows in Emacs to a vertical view. This requires a variation of C-x 1 RET C - x 3 RET C-x o X-x b RET. Instead I now only need to type C-| to make this switch.

```
(defun ensure-two-vertical-windows ()
  "hello"
  (interactive)
  (other-window 1) ;; C-x 0
  (let ((otherBuffer (buffer-name)))
    (delete-window) ;; C-x 0
    (split-window-right) ;; C-x 3
    (other-window 1) ;; C-x 0
    (switch-to-buffer otherBuffer) ;; C-x b RET
  )
  (other-window 1)
)
(global-set-key (kbd "C-|") 'ensure-two-vertical-windows)
```

7.8 Making then opening html's from org's

```
(defun my-org-html-export-to-html ()
  "Make an html from an org file then open it in my browser."
  (interactive)
  (org-html-export-to-html)
  (let ((it (concat (file-name-sans-extension buffer-file-name) ".html")))
    (browse-url it)
    (message (concat it " has been opened in Chromium."))
    'success ;; otherwise we obtain a "compiler error".
  )
)
```

7.9 re-replace-in-file

```
(defun re-replace-in-file (file regex whatDo) "Find and replace a regular expression in-place in a file"
  (find-file file)
  (goto-char 0)
  (let ((altered (replace-regexp-in-string regex whatDo (buffer-string))))
    (erase-buffer)
    (insert altered)
    (save-buffer)
    (kill-buffer)
  )
)
```

Example usage:

```
;; Within mysite.html we rewrite: <h1.*h1>          <h1.*h1>\n NICE
;; I.e., we add a line break after the first heading and a new word, 'NICE'.
(re-replace-in-file "mysite.html"
  "<h1.*h1>"
  (lambda (x) (concat x "\n NICE")))
```

7.10 Obtaining Values of #+KEYWORD Annotations

Org-mode settings are, for the most part, in the form `#+KEYWORD: VALUE`. Of notable interest are the `TITLE` and `NAME` keywords. We use the following `org-keywords` function to obtain the values of arbitrary `#+THIS : THAT` pairs, which may not necessarily be supported by native Org-mode –we do so for the case, for example, of the `CATEGORIES` and `IMAGE` tags associated with an article.

```
;; Src: http://kitchingroup.cheme.cmu.edu/blog/2013/05/05/Getting-keyword-options-in-org-files/
(defun org-keywords ()
  "Parse the buffer and return a cons list of (property . value) from lines like: #+PROPERTY: value"
  (org-element-map (org-element-parse-buffer 'element) 'keyword
    (lambda (keyword) (cons (org-element-property :key keyword)
                           (org-element-property :value keyword)))))

(defun org-keyword (KEYWORD)
  "Get the value of a KEYWORD in the form of #+KEYWORD: value"
  (cdr (assoc KEYWORD (org-keywords))))
```

Note that capitalisation in a `"#+KeyWord"` is irrelevant.

8 Spelling

I would like to check spelling by default.

M-\$ Check and correct spelling of the word at point

M-x ispell-change-dictionary RET TAB To see what dictionaries are available.

```
(define-globalized-minor-mode my-flyspell-global-mode flyspell-mode
  (lambda ()

    ;; spawns an ispell process
    (flyspell-mode 1)
```



```
))
(my-flyspell-global-mode 1)

(setq ispell-dictionary "british") ;; set the default dictionary

Colour incorrect works; default is an underline.

(global-font-lock-mode t)
(custom-set-faces '(flyspell-incorrect ((t (:inverse-video t)))))

Set up a thesaurus to avoid unwarranted repetition.

(load "~/.emacs.d/powerthesaurus.el")
(global-set-key (kbd "M-#") 'powerthesaurus-lookup-word-at-point)
```

Use this game to help you learn to spell words that you're having trouble with; see ~/Dropbox/spelling.txt.

```
(autoload 'typing-of-emacs "~/.emacs.d/typing.el" "The Typing Of Emacs, a game." t)
```

9 Org-mode related things

9.1 ox-extra: Using :ignore: to ignore headings but use the bodies

Use the :ignore: tag on headlines you'd like to have ignored, while not ignoring their content.

- See here: <https://emacs.stackexchange.com/a/17677/10352>

```
(load "~/.emacs.d/ox-extra.el")
(ox-extras-activate '(ignore-headlines))
```

9.2 Executing code from src blocks

For example, to execute a shell command in emacs, write a `src` with a shell command, then `C-c c-c` to see the results. Emacs will generally query you to ensure you're sure about executing the (possibly dangerous) code block; let's stop that:

```
; Seamless use of babel: No confirmation upon execution.
(setq org-confirm-babel-evaluate nil)
```

A worked out example can be obtained as follows: `<g TAB` then `C-c C-C` to make a nice simple graph –the code for this is in the next section.

Some initial languages we want org-babel to support:

```
(org-babel-do-load-languages
 'org-babel-load-languages
 '(
  (emacs-lisp . t)
  (shell . t)
  (python . t)
  (ruby . t)
  (ocaml . t)
  (dot . t)
  (latex . t)
  (org . t)
  (makefile . t)
))
```

```
(setq org-src-preserve-indentation t)
```

9.3 <X Completion

In org-mode we type <X TAB to obtain environment templates, such as <s for source blocks or <q for quote blocks.

9.3.1 Demoing Dot Graphs

We include one to demo the capabilities of the previous subsection.

```
;; Graphviz: Press <g-TAB to obtain a minimal editable example.
(add-to-list 'org-structure-template-alist
  '("g" "#+begin_src dot :results output graphics :file \"/tmp/graph.pdf\" :exports both
  digraph G {
    node [color=black,fillcolor=white,shape=rectangle,style=filled,fontname=\"Helvetica\"];
    A[label=\"A\"]
    B[label=\"B\"]
    A->B
  }\n#+end_src" "<src lang=\"dot\">\n\n</src>"))
```

Here's another example graph,

```
#+BEGIN_SRC dot :file simple_markov.png :cmdline -Kdot -Tpng
graph {
  rankdir="UD";
  A -- D;
  A -- B;
  D -- C;
  B -- C;
}
#+END_SRC
```

9.3.2 <E for emacs-lisp source blocks

<E to begin an emacs-lisp source block – <e is for an example block.

```
(add-to-list 'org-structure-template-alist
  '("E" "#+BEGIN_SRC emacs-lisp\n\n#+END_SRC" "<src lang=\"emacs-lisp\">\n\n</src>"))
```

9.4 org-mode header generation

Generate an untitled org-mode skeleton file C-x t –similar to C-x C-f for finding files.

First the template,

```
#+TITLE: ???
#+DATE: thedate
#+DESCRIPTION: A new radical entry of things I'm learning!
#+AUTHOR: Musa Al-hassy
#+EMAIL: alhassy@gmail.com
#+IMAGE: ../assets/img/rwh-200.jpg
#+CATEGORIES: ExampleTags Elisp Haskell Frama-C Specifications Krakatoa
#+OPTIONS: toc:nil html-postamble:nil
, # Other possible are num:nil todo:nil pri:nil tags:nil ^:Nil
#+STARTUP: indent

* Abstract :ignore:
#+BEGIN_CENTER
*Abstract*
```

This article serves to accomplish *??*.

Write your goal then attempt to realise it, otherwise there's no explicit direction!

#+END_CENTER

* Introduction

Let's recall concepts `~X~` needed to discuss notions `Y`.

* Middle

We're learnin'!

* Conclusion

Yeah! That was some fun stuff!

* COMMENT footer

Local Variables:

eval: (setq NAME (file-name-sans-extension (buffer-name)))

eval: (load-file "AlBasmala.el")

End:

Then the functionality,

```
(defun new-untitled-org-template ()
  "Produce an org-mode file template."
  (interactive)
  (switch-to-buffer (generate-new-buffer "*Untitled*"))
  (insert (file-as-string "~/emacs.d/template.org"))
  (org-mode)
)

(global-set-key (kbd "C-x t") 'new-untitled-org-template)
```

9.5 Org-mode cosmetics

```
;; org-mode math is now highlighted ;-)
```

```
(setq org-highlight-latex-and-related '(latex))
```

```
;; Hide the *,=,/ markers
```

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

```
;; (setq org-pretty-entities t)
```

```
;; to have \alpha, \to and others display as utf8 http://orgmode.org/manual/Special-symbols.html
```

10 Other fun things

- (nyan-mode) Use a cat on a rainbow to indicate the percentage of the buffer position. [Disabled]
- Coloured code delimiters.

```
(define-globalized-minor-mode my-rainbow-global-mode rainbow-delimiters-mode
  (lambda () (rainbow-delimiters-mode))
  (my-rainbow-global-mode 1))
```

- Googling words at point: M-x google-this-word

```
(require 'google-this)
```

11 Summary of Utilities Provided

<u>Command</u>	<u>Action</u>
C-c C-m	recompile file
<f5>	revert buffer
M-x k	kill to start of line
C-	toggle 2 windows from horizontal to vertical view
(file-as-list pathHere)	construe a file as a list of lines
(file-as-string pathHere)	construe a file as a string
(re-replace-in-file file regex whatDo)	perform an in-file regular expression rewrite
M-x create-scratch-buffer	—self evident—
M-x kill-other-buffers	—self evident—
M-\$	check spelling of word at point
M-#	thesaurus look-up word at point
(DANGER-all-locals)	accept, evaluate, all local variables
(SAFE-query-locals)	query whether local variables should be evaluated
C-+/-	increase/decrease text size
M-x my-org-html-export-to-html	make then open html from an org file
C-c C-c	execute code in an org src block
<E	produce an emacs-lisp src block
<g	produce a graph template src block
C-x t	open a new untitled org template file
(org-keywords)	get #+Property: Value pairs from an org file
(org-keyword property)	get the value of a given org #+property

Some possibly interesting reads:

- Arnaud Legrand’s article Emacs init file written in org-mode
- Stackexchange: Using org-mode to structure config files
- A tutorial on evaluating code within **src** blocks