Org-a-nice Your Life In Emacs

https://orgmode.org/

Ellis Kenyő

October 11, 2023

Welcome everyone to my org-mode talk. I've noticed recently that there's still a surprising amount of Emacs users who haven't yet tried out or even heard of what I consider to be one of Emacs' big so-called "killer apps" that truly differentiate it from other similar systems. I'm also undersold on asciidoc both to read in-editor, read on github and write, of which I think org-mode is a much better fit.
In this talk I hope to go over why it's so powerful and what exactly it can be used for.

What is Org-Mode?

ORGanizing and tracking everything in your life

- Written by Carsten Dominik
- Used in various domains by various people
- Note taking, agenda, document export; to name a few

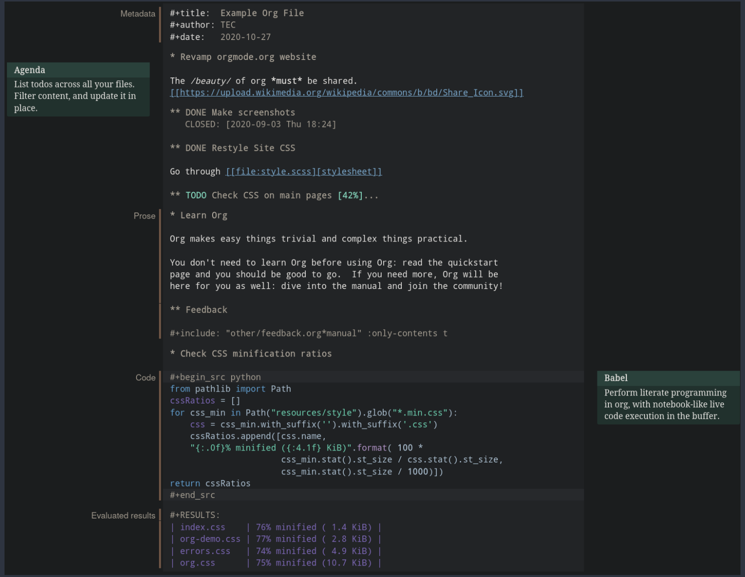
A scientist's take on planning, note-taking, documentation and publishing using plain-text files
– Prof. Carsten Dominik

So as the name suggests; org-mode is about ORGanizing and tracking everything in your life. And as all good things are; this time it's from the comfort of Emacs. Written by a Professor called Carsten Dominik as an attempt to group together all his notes and improve his ability to capture, catalogue and index those notes; it has since evolved to have many fingers in many pies in terms of the breadth of domains and users.
Some of the functionality now encompassed by org-mode that I intend to touch on in this talk include:

- Note-taking
- Agenda/time-tracking
- Code blocks/execution
- Document export

I also intend to touch on the wider ecosystem of org-mode and finally some example, more complex use cases; as time permits.
Included on the slide is a link to a small snippet of users, including a farmer, a mountaineer and even someone who is completely blind.

What *is* an org document?

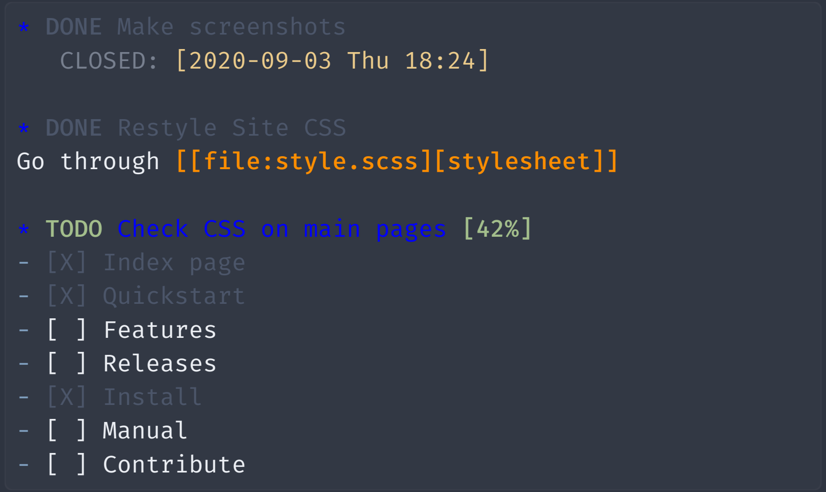

Metadata
Agenda
List todos across all your files. Filter content, and update it in place.
Prose
Code
Babel
Perform literate programming in org, with notebook-like live code execution in the buffer.
Evaluated results

```
#+title:  Example Org File
#+author: TEC
#+date:   2020-10-27

* Revamp orgmode.org website

The /beauty/ of org *must* be shared.
[[https://upload.wikimedia.org/wikipedia/commons/b/bd/Share_Icon.svg]]

** DONE Make screenshots
   CLOSED: [2020-09-03 Thu 18:24]

** DONE Restyle Site CSS

Go through [[file:style.scss][stylesheet]]

** TODO Check CSS on main pages [42%]...
* Learn Org

Org makes easy things trivial and complex things practical.

You don't need to learn Org before using Org: read the quickstart
page and you should be good to go.  If you need more, Org will be
here for you as well: dive into the manual and join the community!

** Feedback

#+include: "other/feedback.org*manual" :only-contents t

* Check CSS minification ratios

#+begin_src python
from pathlib import Path
cssRatios = []
for css_min in Path("resources/style").glob("*.min.css"):
    css = css_min.with_suffix('').with_suffix('.css')
    cssRatios.append([css.name,
    "{:.0f}% minified ({:4.1f} KiB)".format( 100 *
                      css_min.stat().st_size / css.stat().st_size,
                      css_min.stat().st_size / 1000)])
return cssRatios
#+end_src

#+RESULTS:
| index.css    | 76% minified ( 1.4 KiB) |
| org-demo.css | 77% minified ( 2.8 KiB) |
| errors.css   | 74% minified ( 4.9 KiB) |
| org.css      | 75% minified (10.7 KiB) |
```

The image here is a demo from the org-mode website showcasing a number of org features. We'll go over what everything here means shortly in more detail, but at the end of the day it's just a plain text format.
Other packages have been written for neovim and vscode, so the majority of things here should be applicable to them too (maybe not babel and export, which we'll touch on later).
The following explanations are intended to be terse so we can get to the larger features of org, so as with everything here I encourage you to review the manual pages or ask questions at the end if something isn't clear.

Time to go deeper

Headings



```
* DONE Make screenshots
   CLOSED: [2020-09-03 Thu 18:24]

* DONE Restyle Site CSS
Go through [[file:style.scss][stylesheet]]

* TODO Check CSS on main pages [42%]
- [X] Index page
- [X] Quickstart
- [ ] Features
- [ ] Releases
- [X] Install
- [ ] Manual
- [ ] Contribute
```

Here we have a header; denoted by `*`. This can be as deep as needed, Emacs has 8 faces by default but anything past 8 it just cycles through the faces. Some of these headings also exhibit agenda properties; with the `TODO` and `DONE` keywords. These are defined in a variable called `org-todo-keywords` and denote various stages of task completion, used for the agenda we'll talk about later.

You'll also notice on the green `TODO` entry there's a percentage at the end, that denotes how many child tasks are marked as `DONE`. As items are marked off, this progress updates automatically.

Headings are the "main" kind of entity in a document, as the whole document is a hierarchy. If you specify no headers, everything is at the same level. But as you add them, all children then belong to a particular heading.

Useful shortcuts are included for moving headings around (including all children), promoting/demoting headings (by that I mean increasing or decreasing the nesting level) and even refiling headings (by that I mean moving them to headings in other documents)

Markup

- _underline_ *

```
- /italics/
- *bold*
- +strike-through+
- =code= and ~verbatim~
- [[https://orgmode.org][links]].
```

*due to a bug in org-mode, underline doesn't export correctly

Here we have examples of all the various kinds of markup shown both as their in-editor variants and the "exported" equivalents (the exact medium of which depends what you're exporting to, more on that later)
All of these can technically be combined, though the outcomes can be quite undesirable.
We'll touch more more on export & what org-babel is doing later; but in short there's a small bug here that's stopping underline from showing correctly, so I have to manually use LaTeX.

Blocks

Quote

```
#+begin_quote
Don't trust everything you read on the internet
-- Abraham Lincoln
#+end_quote
```

> *Don't trust everything you read on the internet*
>
> *– Abraham Lincoln*

Code

```
#+begin_src emacs-lisp
(message "This is the most useful one!")
#+end_src
```

```
(message "This is the most useful one!")
```

Next we have blocks; which are logical groupings of some kind of contained environment. We use `#+begin` and `#+end` to wrap these, and they can be defined to mean anything.
Above we have quote and code, which for example when exported to LaTeXwould produce a fancy quote box for the quote.
If we look at the last example, we have source code. We've touched on this a few times, but we will go into detail on what we can do with this later.

Metadata

Document metadata

```
#+title: My cool document
#+date: \today
```

Properties “drawer”

```
:PROPERTIES:
:BEAMER_ENV: note
:END:
```

State, priority & tags

```
* TODO [#A] Heading :with:some:tags:
SCHEDULED: <2023-10-02 Mon>
```

The global document and each heading can have metadata associated with it, declared with simple key-value pairs as in the first example. These can be used for a variety of uses far too wide for me to go over here, but you can set document-local options for exporting (eg latex compiler options, extra HTML tags when exporting to HTML etc) and document settings like the title, author & creation date.
You can also set these per-heading under what’s referred to as the “properties drawer”, named because you typically keep it collapsed, expand it to add properties, then close it again (like a physical desk drawer). The other drawer of note is the `LOGBOOK` drawer.
Lastly here we have state, a scheduled date, priority & tags. The state refers to the completion state of a given task (here `TODO` representing the task is yet to be completed), in square brackets we have the priority by default defined in order of importance as `A`, `B` and `C`; and at the end in the colons we have 3 tags. Tags are used as you’d expect, for grouping tasks logically and allowing you to create more granular agenda views (which will be touched on further).
As expected, the `SCHEDULED` date refers to the task having some date in which the task should be completed, and a similar `DEADLINE` property.

Tables

```
| Weight |  Salt |
|--------+-------|
|    255 | 3.825 |
#+TBLFM: $2=$1*1.5/100
```

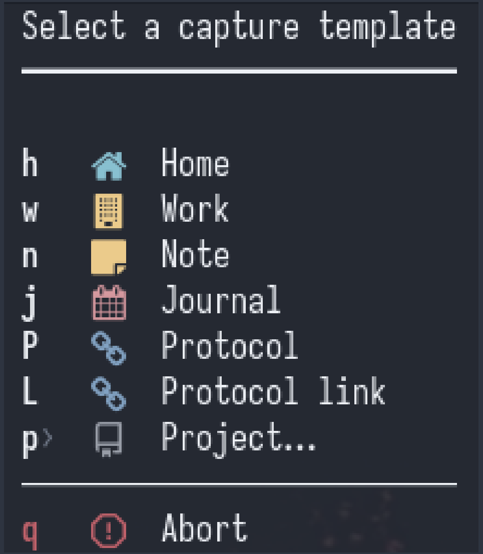
```
| Weight |  Salt |
|--------+-------|
|    255 | 3.825 |
|    233 |       |
#+TBLFM: $2=$1*1.5/100
```

```
| Weight |  Salt |
|--------+-------|
|    255 | 3.825 |
|    233 | 3.495 |
#+TBLFM: $2=$1*1.5/100
```

Like a lot of things in org-mode, tables are deserving of their own Safari entirely. The simple demo here shows an Excel-like capability of tables letting you define formulae to be executed. Here I have a simple table I use when cooking steak to calculate the correct amount of salt to use, all I have to do is `TAB` across to create a new row, add in the weight and do `C-c C-c`. The salt value is then computed.
There's far too much to go into here, so you encouraged to view the manual.

Note taking

Org-capture



```
Select a capture template

h    Home
w    Work
n    Note
j    Journal
P    Protocol
L    Protocol link
p›   Project...

q    Abort
```

One of the more powerful use-cases for org-mode is (shockingly) organisation. The first stages of planning are capturing ideas. As you're coding or planning, you have a brainwave and you need to quickly jot something down to revisit later.

This is where org-capture comes in, and lets you define extensible templates (my complicated example borrowed from tecosaur) to let you quickly pop open a note to be saved to a pre-defined org file, or even “refile” to another file.

These templates let you pre-populate things like the date if say you're creating a task, or the line in the file you're at when you invoke if you're recording a project note.

Agenda

Schedule tasks

```
** TODO Finish this Safari
SCHEDULED: <2023-10-11 Wed>
DEADLINE: <2023-10-11 Wed>
```

So we've touched on tasks briefly, but when we say task we very simply mean a "heading" that has a state. Once it has a state keyword, it is a valid task that can be used as part of the agenda. You can also link dates to that task, and this status will show up in the associated agenda views and also lets you ask questions of your tasks; say "Which tasks are scheduled to be done in the next 4 days?" or "What deadlines do I have next week?".

Repeating tasks/habits

```
** TODO Mention how useful org-mode is
DEADLINE: <2023-10-11 Wed +1h>
```

Similar to scheduling one-off tasks, you can also mark a task as “repeated” by adding a repeater at the end (in this case I mention how useful org-mode is every hour). When viewing these in an agenda view, they would show up differently and let you track and record each “instance” of completion, say if you were tracking a habit instead.
There’s a useful tutorial here on habit tracking.
See more on repeated tasks in the manual.

Task status/reason

```
* TODO Finish this section
* INPROG Present my Safari
* DONE Create jokes for the viewers
```

```
(cdar org-todo-keywords)
```

```
("TODO(t)" "INPROG(i)" "PROJ(p)" "STORY(s)"
↪  "WAIT(w@/!)" "|" "DONE(d@/!)"
 "KILL(k@/!)")
```

The status of a task comes from a defined set of keywords, the above showing what's defined for me. In short, everything before the `|` value defines tasks that have some kind of "action" applied to them, in order to advance it to the next state. The last two, `DONE` and `KILL` and "end points" for a task, no further action is needed here.
The other symbols denote things like whether or not you should be prompted to add an entry to the `LOGBOOK` drawer.
For further info, see the docstring for `org-todo-keywords`.

Agenda Demo

Showcase creating task from capture, logbook, date picker, cycling through states

Babel

So what's with all these code blocks?

```
#+begin_src emacs-lisp :exports both
(list (list "Header" "Value")
      (list "Date" (current-time-string))
      (list "It's just data"
       ↪  "$x=\\frac{-b\\pm\\sqrt{b^2-4ac}}{2a}$"))
#+end_src
```

```
(list (list "Header" "Value")
      (list "Date" (current-time-string))
      (list "It's just data"
       ↪  "$x=\\frac{-b\\pm\\sqrt{b^2-4ac}}{2a}$"))
```

Header	Value
Date	Wed Oct 11 13:54:40 2023
It's just data	$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$

Finally we come to these code blocks; these are part of what's called Org-Babel. Named after the biblical tale that told of a race of humans that spoke the same language trying to build a tower to the heavens. Yanweh, a deity, observed and punished the humans by making them all speak different languages; making constructing the tower impossible. He then scattered the humans, and that apparently is where the origin of everyone speaking different languages come from.

Unlike the story though, we can leverage multiple languages to our advantage. These code blocks can be executed, with many languages having an org-babel extension (shortened to ob) instructing babel how to run and interpret results from various languages.

Here we have an example of a simple code block, also showing how you pass options to the execution of said block. Here, we're instructing org-babel to export both the code and the results when the document is exported (more in the next section). And because the result is a cons list, babel by default renders the output as a table, which has been exported above in latex (including the quadratic formula)

Notebooks

First we define *the* most useful function we're ever likely to use

```
(defn super-cool-fn []
  42)
```

```
#'user/super-cool-fn
```

Then; we execute it. There might be a race condition here, I didn't verify if the channel was closed so the output might come in order of out.

```
(super-cool-fn)
```

```
42
```

A good use-case for this is the idea of a "literate notebook", with code sections interspersed with prose. Here we have a small sample of what could be one such notebook, these code blocks are run inside of the same "session" (what a session is differs from executor-to-executor, but here it's a cider REPL) meaning that each code block has access to the same code as any other block marked in that session.
If you're reading this after the fact, check the properties drawer under the heading.

Piping results

```python
import hashlib, zlib, os
string_to_hash = 'Hello from Org-Mode'
header = "blob " + str(len(string_to_hash)) + "\0"
blob = (header + string_to_hash).encode('utf8')
sha = hashlib.sha1(blob).hexdigest()
git_object = f".git/objects/{sha[:2]}/{sha[2:]}"
os.makedirs(os.path.dirname(git_object), exist_ok=True)
with open(git_object, 'wb') as f:
    f.write(zlib.compress(blob))

return f"Wrote: {git_object}"
```

```
Wrote: .git/objects/fc/86439179b93382f976c577895ed72011f766c6
```

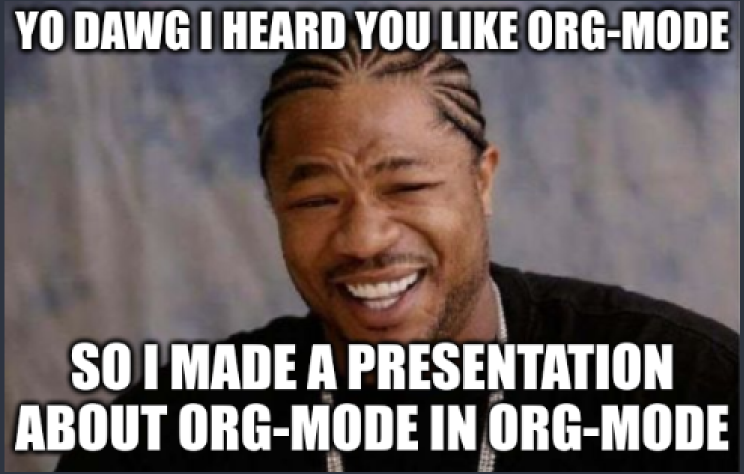
```
git cat-file -p 'fc86'
```

```
Hello from Org-Mode
```

Here we have what looks like a simple snippet of python (from another talk I've done on Git Fundamentals), but when we look at the code behind it shortly you'll see that there's couple of useful features here; piping results to other blocks and what's referred to as "noweb" syntax letting you refer to the results of other code blocks directly. The hash object being written here is *actually* being written and retrieved, and the value of "I am a file" is coming from another code block. By simply parametrizing these blocks, you can easily create a rich set of demos that are very easy to modify.

Export

How did I make this presentation?


YO DAWG I HEARD YOU LIKE ORG-MODE
SO I MADE A PRESENTATION ABOUT ORG-MODE IN ORG-MODE

And now we finally get to export. *Another* fantastic feature of org-mode that lets you take an org document and run it through an org-exporter (shorted to ox, like org-babel and ob) to produce some other format.
The most common of which are LaTeX(which is how this presentation has been created) and HTML, for example my Doom Emacs config.
This is done through a number of backends, one such being pandoc; instantly giving you access to a large variety of output formats.
As you've seen from previous slides, any raw LaTeXcode gets translated verbatim; the same applies to any HTML or any other format specs

Export Demo

Showcase the inner workings of the git slide, adjust some of the values and re-export, the many formats available ootb with Doom, run through the document behind the presentation

Advanced Use-Cases

Literate configuration

```
* My super cool Emacs config
#+begin_src emacs-lisp :tangle init.el
(message "Emacs Started!")
#+end_src
```

```
(let* ((default-directory user-emacs-directory)
       (changed-at (file-attribute-modification-time
       ↪  (file-attributes "config.org"))))
  (require 'org-macs)
  (unless (org-file-newer-than-p "init.el" changed-at)
    (require 'ob-tangle)
    (org-babel-tangle-file "config.org" "init.el"
    ↪  "emacs-lisp"))
  (load-file "init.el"))
```

Many Emacs users (myself included) use what's referred to a "literate configuration", which similar to the notebooks idea is a configuration file exported from code blocks interspersed with regular prose.
This allows you to have a very neat configuration that will be nicely readable in Emacs, in some code forge like Github or even better exported to a format like PDF (via LaTeX) or HTML.
It also leads nicely into our next use-case, which we've briefly touched on before

Compile-time dynamic content

```
#+NAME: keyboard-device-file
#+begin_src shell
find /dev/input/by-{path,id} -name '*-kbd' | head -n 1
↪  | tr -d "\n"
#+end_src

#+begin_src kbd :tangle kmonad.kbd :noweb yes
(defcfg
  input (device-file "<<keyboard-device-file()>>")
  output (uinput-sink "My KMonad output")
  fallthrough true)

...
#+end_src
```

Courtesy of ambirdsall.

We've touched on something called "noweb" before, but here it is in with a great use-case.
`kmonad` is a tool for creating bespoke layouts using a lisp-like language. Here, it needs the keyboard device file; but that's never guaranteed to be the same across keyboards, systems, etc. So instead of updating the file constantly, here we can use the result of the now named block `keyboard-device-file` and add it in the below file in the chevrons with empty parens (since these blocks can also take arguments).
We also see the `tangle` execution option, which lets us redirect the output of the block to a file instead.
This can be useful for all manner of things, say you prefer a particular theme across all your devices. You can define a table with all the colours in it, and use that in every other place you need to "tangle" other config files. Update the table, and all your other files will be updated.
This is also a great option for handling secrets, you can have a code block to return a result from `pass` or equivalent password manager and have that secret be tangled into any files that need them.

Export snippets

```
:header-args:snippet: :mkdirp yes :tangle
↪  (expand-file-name (downcase (car (last
↪  (org-get-outline-path t)))) (expand-file-name
↪  (downcase (car (last (butlast (org-get-outline-path
↪  t))))) "snippets"))
```

```
tree ~/.config/doom/snippets
```

```
/home/lkn/.config/doom/snippets
├── clojure-ts-mode
│   └── __bb.edn
├── org-mode
│   └── __
└── slack-message-compose-buffer-mode
    └── standup

4 directories, 3 files
```

Here we have one from my personal literate configuration, using the structure of org-mode to create yasnippet snippets. Using `tangle`, we're simply telling org-mode the file should be the second-to-last header the code block is under, followed by the last header; which in my config is the mode-name followed by the file name (the format yasnippet expects)
There's other code that goes with it, I've made a small guide on setting this up here.

Static site generation

```
(setq org-publish-project-alist
      `(("files"
         :base-directory ,doom-user-dir
         :base-extension "org"
         :publishing-directory "out"
         :exclude "README.org"
         :publishing-function org-html-publish-to-html
         :completion-function +org-publish-rename
         :with-creator t
         :section-numbers nil)

        ("site" :components ("files"))))

(org-publish-project "site" t)
```

Implied by the idea of being able to export a file, it's also possible to create a "publish project" of various files and merge them together to create a published output.
Here is a simplified snippet from the script used to export my own literate config. You can define many components for various files, for example static content you have to run through a pre-processor or images you have to compress first.
These are then called by `org-publish-project` which produces the output you see here.

Closing

Wider ecosystem

- org-roam
- ox-chameleon
- ob-restclient
- org-caldav
- org-transclusion
- hugo and ox-hugo

Nearly there now, here's just a snippet of some of my favourite packages in the wider org ecosystem.
org-roam I'm sure many of you are at least somewhat familiar with, it's an implementation of Roam Research's tool for managing personal information. What's here is worthy of a Safari on its own, but it's worth looking into
ox-chameleon is the fantastic package being used here to have the presentation use my theme of choice, Nord, all over. The package also supports HTML exports, and there's a PR I've had ongoing to include the extra niceties found on my exported config. You'll likely have noticed some inconsistencies with how some code blocks are coloured, there's a bug in ox-chameleon that's not yet resolved causing it to default to the base colour scheme rather than mine..
ob-restclient is an org-babel extension for restclient.el allowing you to create a notebook for a REST API (there's one for GraphQL too). This is great for producing a self-annotating README for an API or service
org-caldav is one I've moved away from, but in essence it lets you manage calendar events through caldav using org-mode. Integrating with G suite, Office 365 etc you can manage all your events through Emacs, link to events easily, refer to them from emails, etc
org-transclusion is a simple package which lets you insert a copy of a file's content using a regular file link. Great for creating nested documents, worth checking out the README if you're interested
And lastly I've included Hugo here because it indirectly supports org-mode posts (with another package inbetween) and is commonly how people manage blogs using posts and pages written in org-mode
I've barely scratched the surface both on what's included as part of Emacs and what's available wider, so I encourage you to read through the manual and explore what's out there.

Any questions?