Vim Plugin for Web Publishing

2026-03-09 Frank Siebert

I wrote lengthily how I moved away from wordpress to write articles in mediawiki and publish via git hook to my then new designed web site. Now its time to report the sunset of my mediawiki. I moved on to Vim and GitLab flavored Markdown.

> **Update 5 2026-03-18:** I used blockquote in the past for update notes like this as well. Since I decided to visualize quotes as it became common with a light-gray left border, I need now a different solution for the update notes.
>
> These are now maintained as:
>
> ```
> > [!note]
> > **Update yyyy-MM-dd:** Note text as it applies
> ```
>
> which creates `<div class="note"><div class="title">Note</div>...</div>` in the HTML. I updated the code to remove the superfluous `<div class="title">` tag. I considered this a style matter, see function `style()` .
>
> **Update 4 2026-03-18:** Markdown does not define any standard to control columnwidth at table columns. Applying a style update to an old article, I required to enable such a feature.
>
> ```
> | %25% Header1 | %25% Header2 | %50% Header3          |
> |:-------------|:-------------|:----------------------|
> | text 1       | text 2       | Much longer text here |
> ```
>
> Above formating directions lead to columnwidth entries at the `<th>` tags. It's in the first part of the new function `tables()`
>
> **Update 3 2026-03-18:** Markdown does not define any standard to create rowspan for table cells. For the update of an existing article I needed it.
>
> ```
> | Header1      | Header2      | Header3               |
> |:-------------|:-------------|:----------------------|
> | text 1       | text a       | belongs to 1.a        |
> |       ^      | text b       | belongs to 1.b        |
> | text 2       |       ^      | belongs to 2.b        |
> ```
>
> The char ^ is used to signal, that this cell gets joined with its predecessor. No further text in this cell must exist. The implementation is in the second part of the new function `tables()

> **Update 2 2026-03-18:** Weasy does automatic page breaks in tables, even repeating the table header on the new page. However, depending on the space requirement of the single cell, some cells of one row might be on one page and some other on the other.
>
> I enabled now to set a manual page-break in the first table-cell of the row to be moved to the next page via `<!-- page-break -->` . The implementation is in method `page_break()` .
>
> **Update 1 2026-03-18:** subprocess calls got the additional parameter `check=True` .

I'll not go into the details of the migration. It's enough to mention that I extracted all wiki content as mediawiki files and converted those to markdown via Pandoc.

Those mediawiki files, which where my article source files, where converted the same way.

No, this time I will restrict myself to describe the authoring and publishing plugin I created for Vim. Another article might follow to describe the implementation of the deployment via git hook.

The appearance of the web site stays the same, and my design goals for the site as well.

- No JavaScript, at least not for the portal, only for the content, if it does require it.
- No resources linked into the site, which are loaded from 3rd-party servers.

Sure, my content has links to other sites, as I'm used to provide references in my articles. But you have to click on these consciously to be taken to the referenced page. The visitor is in control.

Table of Content

Plugin Short Description

It's a bit more, but this is the core functionality:

- Render markdown into HTML, applying the semantic HTML structure, the style and, if applicable, media links, as they show up in the published result.
- Hand the rendered HTML and its assets over to the deployment staging area of my site.

Some assets are generated together with the HTML, the QR-Code which contains the link to the article and the PDF containing the same article as the HTML.

If an audio is recorded by me to provide the article as podcast, that is linked into the HTML as well.

Those steps where done before by software launched via git-hook during commit, which means that my deployment process became much leaner now. But that is part of the deployment to the web-site, and I plan to write a separate article about that.

Meta Data

There are two kinds of meta data, which, due to the lack of a better place, have to be inserted into the markdown file. To place this meta data information into the markdown file, I decided to use XML/HTML comments like:

```
<!-- pdf -->
```

The plugin provides a list of meta data comments to insert the selected one at the current cursor position.

Control Data

Shall a PDF be created? Do I want to have a table of content, and where would I like to have that? Where shall the references be placed? Where are additional page-breaks required in the PDF? Which language is the article in?

While writing mainly in German, there are sometimes exceptions to that. This article, for example, is in English, and will therefore not be shown as part of the "idee"-portal, but of the "concept"-portal. This makes the language information in my case control data.

It is, however, also article meta data.

Article Meta Data

Who is the author? Under which Licence is the content provided?

As long as I write all articles myself, always using the creative-commons zero licence, I could just hard code this information. And a lot things, which should be customizable, are currently not. But I decided to enable guest articles with different content licence. At least in principle that function is there, I would need to add some information about this other content licence the first time it is used.

Workflow and File System Structure

Lets first take a look at the target file system structure.

```
frank@Asimov:~/projects/idee/website$ tree -d -L 1
.
├── archive
├── article
├── audio
├── css
├── env
├── files
├── image
├── js
├── legal
├── MathJax -> SimpleMathJax/resources/MathJax/es5
├── pdf
├── portal
├── qrcode
├── SimpleMathJax
└── sitemap
```

Articles, audios, files, images, JavaScripts, PDFs and QR-Codes are placed in separate folders next to next of each other.

At one hand we might want to re-use an image or JavaScript already existing, at the other hand we want to have easy access to those files we are working with, and want to avoid searching for these in a long list of files day by day.

To facilitate both options, the authoring happens in a separate folder, one per article currently in creation, with the asset-folders beneath and a symbolic link to the idee folder, by which existing assets can be referenced in the markdown file.

That looks as follows:

File System Structure

```
frank@Asimov:~/projects/writing/1-vim-plugin-idee$ tree
.
├── audio
├── files
├── idee -> /home/frank/projects/idee
├── image
├── js
├── pdf
├── qrcode
└── vim-plugin-for-web-publishing.md
```

To create this setup for authoring again and again by hand would be too much work. It is done by the plugin command `IdeeFolders` .

Workflow

The workflow is quite lean and basic:

1. create article folder in your preferred location
2. `vim your-markdown-file.md`
3. write the article
4. call command `IdeeFolders`
5. create and link assets
6. add meta data via command `IdeeMeta` or

7. call command `IdeeDisplay` to render HTML (opens the page in Firefox)
8. re-iterate steps 5 to 7 until everything fits
9. call command `IdeePublish`

The command `IdeePublish` copies the markdown, the HTML and all assets to the idee folder structure and adjusts the relative links used during authoring to fit to the new location.

Step 9 is the handover to the deployment of the article.

Feature List

As shown in the workflow description, the plugin features 4 commands:

- `IdeeFolders` creates the asset folder and the symbolic link to the idee website project.
- `IdeeMeta` provides a list of meta data comments
- `IdeeDisplay` creates the HTML and opens it in the web-browser
- `IdeePublish` hands the article and its assets over to the deployment

Plugin Folder Structure

This, I confess, took some time to figure out. Where do I need to place what and how to I make sure that it is loaded when a markdown file is edited?

As soon as it works, it seems to be simple.

```
frank@Asimov:~/projects/writing/1-vim-plugin-idee$ tree -L 5 ~/.vim/pack/idee/
/home/frank/.vim/pack/idee/
└── start
    └── plugin
        ├── ftplugin
        │   └── markdown_idee.vim
        └── python3
            └── idee
                ├── display.py
                ├── folders.py
                ├── __init__.py
                ├── publish.py
                └── __pycache__
```

The folder `pack` probably stands for package, I do not know. If so, then I have a plugin package named `idee`, with a folder `start` which is, as I understand, searched during the start of vim, whether applicable plugins are found below.

Below folder `plugin` there is the `ftplugin` folder, where the code resides, which makes the python functions available via new vim commands.

For whatever reason the python code needs to be placed below a python3 folder or it is not found by vim.

File Type Plugin markdown_idee.vim

There is a binding naming convention for file type plugins, and if you do not adhere to it, it is not loaded when a file of that type is edited.

The name has to start with the file type, as it is known to vim. That can be followed by an underline and whatever you like to write afterwards. I stuck with the plugin name, which is also my web-site project name.

markdown_idee.vim

```
py3 import idee

function! markdown_idee#meta_comments()
  let l:comments = [
      \ '<!-- author: -->  ',
      \ '<!-- article-licence: cc0 -->  ',
      \ '<!-- en-US -->  ',
      \ '<!-- page-break -->  ',
      \ '<!-- pdf -->  ',
      \ '<!-- references -->  ',
      \ '<!-- toc -->  '
      \ ]

  function! s:meta_insert(id, result) closure
    if a:result > 0 && a:result <= len(l:comments)
      execute "normal! i" . l:comments[a:result - 1]
    endif
  endfunction

  let winid = popup_menu(l:comments, #{
        \ callback: 's:meta_insert'
        \ })
endfunction

command IdeeFolders :py3 idee.folders.folders(vim.eval("expand('%')"))
command IdeeDisplay :py3 idee.display.display(vim.eval("expand('%')"))
command IdeeMeta call markdown_idee#meta_comments()
command IdeePublish :py3 idee.publish.publish(vim.eval("expand('%')"))
```

The documentation of vim still refers to the command `python` , but that doesn't work, it needs to be `py3` . The file type plugin imports the python package idee with its modules.

The `meta_comments()` function is also defined in vim-script, defining a nested callback function to insert the selected comment into the current editor buffer at the current cursor position.

Oh, yes, the function shows a popup menu with the comments to choose from.

After loading the python package and defining the function `meta_comments()` the file type plugin binds the python functions and the vim-script function to new vim commands.

The current markdown file, the file the author is working in when using the commands, is represented by `%` . Python, called via `py3` doesn't know that and wouldn't know what to do with that percent sign. Therefore this placeholder for the current file name needs to be expanded into the file name using the python function `vim.eval()` provided by the vim developers.

Module initialization

The python modules are imported via the `__init__.py` of the package.

__init__.py

```
"""
The package 'idee' provides functions to be used via command in vim during the
editing of markdown files, which are meant to be published as articles on the
idee-website or the concept-website.

Functions provided are:
    idee.folders():
        - Creates the folder structure for assets
        - Creates a link to the idee/concept website project
    idee.display()
        - Creates idee/concept website style HTML from the markdown file.
        - Creates a corresponding PDF file, if requested.
        - Shows the result in Firefox
    idee.publish()
        - Does the handover to the idee/concept website project for deployment
"""
import os
from pathlib import Path
import pkg_resources

import idee.folders
import idee.display
import idee.publish
```

IdeeFolders

The module `folders` creates subfolders below the folder the current markdown file is in.

Function folders()

folders.py - folders()

```
"""
folders(filepath) creates folders for idee website assets. It creates also a
symbolic link to the idee project to make existing assets available.

These folders are:
    - files - arbitrary linked files
    - image - image files
    - js - javascript files for interactive content
    - pdf - article pdf
    - qrcode - article qrcode
    - audio - recording of the article

For the target project idee a symbolic link is created
    - ln -s idee

@date: 2026-02-19
"""

import os
import string
from pathlib import Path

def folders(filepath: string):
    """
    Make the folder of the given markdown file a workfolder.
    This is done by creating the subfolders for possible assets.

    Returns
    -------
        Feedback Message via stdout
    """

    dlist = ["files", "image", "js", "pdf", "qrcode", "audio"]

    inpath = Path(filepath)
    workdirpath = inpath.parents[0]
    workdirpath = workdirpath.resolve()

    # create symbolic link to idee project
    idee = workdirpath / "idee"
    if not idee.exists():
        os.symlink(Path("/home/frank/projects/idee").resolve(), idee)

    for d in dlist:
        d = workdirpath / d
        if not d.exists():
            d.mkdir()
```

IdeeDisplay

The module `display` creates HTML, on request via meta data comment also PDF, and opens the created page in Firefox. It makes sense to look at this module part by part.

display.py - imports and debugging

```
#!/user/bin/python3
"""
Generate and Display HTML in idee-website style from github flavored
markdown.

    To debug the function display(), start this executable python file with
    the markdown file as parameter.

    TODO: Check alternatives to soft deprecated module getopt.

@date: 2026-02-05
"""

import sys
import os
import subprocess
import string
from pathlib import Path
from datetime import datetime
import re
import copy
import getopt

from selenium import webdriver
from selenium.webdriver.chrome.options import Options

from bs4 import BeautifulSoup
from bs4 import Comment
from bs4.builder._htmlparser import HTMLParserTreeBuilder
from weasyprint import HTML
from weasyprint import CSS
import qrcode
...
if __name__ == "__main__":

    MARKDOWNFILE = None

    try:
        opts, args = getopt.getopt(sys.argv[1:], ["o"])

    except getopt.GetoptError:
        print("No Parameter given")
        sys.exit(2)

    if len(args) == 0:
        print("No Parameter given")
        sys.exit(2)

    MARKDOWNFILE = args[0]
    if not MARKDOWNFILE:
        print("No Parameter given")
        sys.exit(2)

    display(Path(MARKDOWNFILE))

    sys.exit(0)
```

The code shows the imports and the `__main__` part used to call the display function from terminal for e.g. debug purposes. The ellipsis shown in the middle of this code snipped stands for all the function definitions in between, which in the following will be shown one by one.

Function display()

The display function uses Pandoc to create the initial HTML from the markdown file. Afterwards the module `BeautifulSoup` is used to adjust the HTML to the style used for the idee/concept portal.

display.py - display() - part 1

```
def display(filepath: string):
    """
    Creates an html web page frm the markdown file.

    The html web page is fully flavored for that site,
    including the optional linked in pdf and audio versions
    and content licence information.

    Returns
    -------
    None.
    """

    mdpath = Path(filepath)
    workdirpath = mdpath.parents[0]

    # check that this a designated workdirectory
    idee = workdirpath / "idee"
    if not idee.exists():
        print("Use first IdeeFolders to designate this location as"
                " workdirectory, or use the PreVim-Plugin instead."
              )
        return

    htmlpath = workdirpath / mdpath.stem
    htmlpath = htmlpath.with_suffix(".html")

    # To enable --toc, the parameter -s (standalone) needs to be set.
    # This parameter leads to the generation of an html header with
    # some meta tags.

    # The TOC is created as <nav id="TOC"> tag,

    # Own meta data lines need be injected and
    # the toc needs to be moved to the correct location if specified,
    # or removed, if specified.
    htmltext = subprocess.run( [ "pandoc", "-s",
                                # create table of content
                                "--toc", "--toc-depth=5",
                                # use MathJax
                                "--mathjax" +
                                "=./idee/website/MathJax/tex-chtml.js",
                                # github flavor markdown
                                "-f", "gfm",
                                # html as output format
                                "-t", "html",
                                # input file
                                "-i", mdpath
                                # don't use stdout, return the result
                                ],
                                capture_output=True,
                                check=True)
```

The function first checks the existence of the symbolic link to the idee website project. If that link does not exist, then obviously the current directory had not been prepared to serve as authoring directory for an idee/concept article.

Pandoc is called to convert from GitLab flavored markdown to HTML and is told to process also MathTex statements like:

From "The Cosmological Constant as Event Horizon" [1]

$$\nabla_\mu g^\mu = \frac{d\Theta}{d\tau} + \frac{1}{3}\Theta^2 = R_{\mu\nu}u^\mu u^\nu = \Lambda - 4\pi G(\rho + 3p)$$

The HTML-comments in the markdown are preserved during conversion to HTML. With this meta information and with the target HTML structure in mind, the returned HTML is processed further.

display.py - display() - part 2

```
    html_doc = htmltext.stdout.decode("utf-8")

    # do some things the soup does not want to do
    html_doc = html_doc.replace("<body>",
                                "<body><main><article><header></header>")
    html_doc = html_doc.replace("</body>", "</article></main></body>")

    builder = HTMLParserTreeBuilder()
    soup = BeautifulSoup(html_doc, builder=builder)

    # remove script added by pandoc
    scripts = soup.find_all("script")
    for script in scripts:
        script.decompose()

    style(soup)
    urnmeta(soup, mdpath.stem)
    images(soup, workdirpath)
    language(soup)
    title(soup)
    dates(soup, workdirpath, mdpath.stem)
    tables(soup)
    article_author(soup)
    article_qrcode(soup, workdirpath, mdpath.stem)
    article_licence(soup)
    article_audio(soup, workdirpath, mdpath.stem)
    movetoc(soup)
    footnotes(soup)
    # keep this as last step
    article_pdf(soup, workdirpath, mdpath.stem, htmlpath)

    html_doc = soup.prettify()

    with open(htmlpath, 'w', encoding='utf-8') as outfile:
        print(html_doc, file=outfile)
        outfile.flush()
        outfile.close()

    print(f'wrote file {htmlpath}')

    subprocess.run(["firefox", htmlpath], capture_output=False, check=True)
```

That's it. First I sneak in the target HTML structure, having a main tag containing the article tag, which contains a header as first element to place some header information about the article.

Afterwards I create a BeautifulSoup and then remove the scripts placed by Pandoc, since I want to have HTML free of scripts as long as its not absolutely required.

Then I have a number of functions running on that soup to add and modify some details based on the meta data and the assets found, into which we will look next in detail.

Function style()

The function style() adds everything to the soup, which I consider to be part of the portals style.

display.py - style()

```
def style(soup):
    """
    Show some personal style.
    """
    # inject the generator meta information.
    # one exists already
    tag = soup.find("meta", attrs={"name": "generator"})
    tag.attrs.update({"name": "generator", "content": "vim, pandoc, idee"})

    head = soup.find("head")
    # inject ghost message from Terry Prachett
    # http://www.gnuterrypratchett.com/
    meta = soup.new_tag("meta")
    meta.attrs.update({"http-equiv": "X-Clacks-Overhead",
                       "content": "GNU Terry Pratchett"})
    head.insert(6, meta)

    # pandoc creates a style tag. if --mathml option is set
    tag = soup.find("style")
    if tag:
        tag.decompose()
    # my own style
    # inject stylesheet link <link rel="stylesheet"
    # href="../website/css/fs.css"/>
    link = soup.new_tag("link")
    link.attrs.update({"rel": "stylesheet", "href": "./idee/website/css/fs.css"})
    head.insert(7, link)

    # pandoc creates <div class="note"><div class="title">...</div></div>
    # for > [!nete] and similar for > [!important]. The titles have to go.
    tags = soup.find_all("div", class_="title")
    for tag in tags:
        tag.decompose()

    # favicon
    # href="../image/favicon.ico" rel="icon" type="image/x-icon"
    link = soup.new_tag("link")
    link.attrs.update({"rel": "icon", "href": "./idee/website/image/favicon.ico",
                       "type": "image/x-icon"
                       })
    head.insert(8, link)

    math = soup.find("span", class_="math")
    if math:
        # behind title tag, before MathJax loading
        script = soup.new_tag("script")
        script.attrs.update({"type": "text/javascript"})
        script.append("window.MathJax={tex: {tags: 'span'}};")  # instead of all
        head.insert(9, script)

        script = soup.new_tag("script")
        script.attrs.update({"type": "text/javascript",
                             "src": "./idee/website/MathJax/tex-chtml.js"})
        head.insert(9, script)
```

Part of style is the honoring the software used to create the web-page in the generator meta data. I used vim, Pandoc and now my own software, named like my web-site and vim-plugin simply 'idee'.

Terry Prattchet gets honored in the X-Clacks-Overhead [2], which is not very effectful, but my web-server honors him as well in the HTTP Header, which makes the X-Clacks Browser-Extension signal his presence in the overhead.

Not less important is the link to the CSS-stylesheet, the web-sites favicon icon, and, only in case span tags of type 'math' are found, the script for MathJax.

Function urnmeta()

In my previous implementation the title used in mediawiki became the file name of the mediawiki file, and I had to compute an URN, an unique resource name, to be used as HTML file name.

That meant very long file names and, well, the mediawiki file names even had spaces. That mess is over now, at least for new articles.

display.py - urnmeta()

```
def urnmeta(soup, urn):
    """
    The name of the markdown file becomes urn for the web.
    Filenames are all lowercase with '-' instead of spaces,
    and roman letters only (no german umlaute).

    I got tired of mixed case filenames with spaces anyhow during
    my use of my mediawiki extract tool.
    """
    # meta data
    head = soup.find("head")
    meta = soup.new_tag("meta")
    meta.attrs.update({
        "property": "article:urn",
        "content": urn})
    head.insert(6, meta)
```

As URN parameter `mdpath.stem` is used, that forces me to take care while I name my markdown file, that I adhere to the self set rules.

Yes, I should probably add a check for that, and probably I will. And probably I should retire that meta data entry, since the information exists first in the file name and later in the URL.

Function images()

This function looks at all img tags and, if no relative path is given to the image, looks in which of the two possible image folders the image can be found.

That means, that I can use the image file name in the markdown without any path.

Images used in the header, especially the qrcode image, are in different folders. Therefore this function has to be called early and not be moved to later process stages.

display.py - images()

```
def images(soup, workpath):
    """
    Run early, before we place pictures in the header.

    Inspect image source path. Keep realtive paths alone,
    lookup image location if only the image filename is given.

    PARAMETER
    soup:
        The html soup
    workpath:
        The location of the md-file
    """
    imgs = soup.find_all("img")
    for image in imgs:
        src = image.attrs.get("src")
        if src and "./" not in src:
            imagepath = None
            imgp = None
            for imgp in ["image", "idee/website/image"]:
                imgp = workpath / imgp / src
                if imgp.exists():
                    imagepath = imgp
                    break

            if imagepath:
                src = f"./{os.path.relpath(imagepath, workpath)}"
                image.attrs.update({"src": src})

        if src:
            # make image anchors with target _blank to open in separate tab
            anchor = soup.new_tag("a")
            anchor.attrs.update({"href": src, "target": "_blank"})
            image.insert_after(anchor)
            anchor.append(image)
```

Function iscomment()

Time to introduce a very small helper function to find all comments we placed in the markdown file.

display.py - iscomment()

```
def iscomment(elem):
    """
    Helper function to search for comments

    source:
    https://www.tutorialspoint.com/beautiful_soup/beautiful_soup_find_all_comments.htm
    """
    return isinstance(elem, Comment)
```

Function language()

The language function maintains all language specific aspects. That's for one the `xml-lang` information in the HTML tag and the language meta tag, but it's also the HTML comment with the information, whether the idee web-site portal or the concept web-site portal has to be included for the SSI header injection done be the web-server.

If no language information was provided in the markdown file, it defaults to German.

display.py - language()

```
def language(soup):
    """
    Check for <!-- en-US --> language information comment.
    If not, assume de-DE.

    Populate the language attributes at the html tag and insert
    the include comment for the portal.
    """

    # look for a language comment
    comments = soup.find_all(string=iscomment)
    c = None
    for comment in comments:
        if comment in ' en-US ':
            c = comment
            break
    if c:
        lang = str(c).strip()
        c.decompose()
    else:
        lang = "de-DE"

    # SSI header injection is a function of the language
    body = soup.find("body")
    if lang.startswith("de"):
        c = Comment('# include file="/portal/idee-header.html" ')
    else:
        c = Comment('# include file="/portal/concept-header.html" ')

    body.insert(0, c)

    # inject language informationi
    html = soup.find("html")
    html.attrs.update({"lang": lang})
    html.attrs.update({"xml:lang": lang})

    # and 2 lines meta information
    # <meta content="de-DE" property="og:locale"/>
    # <meta content="Idee" property="og:site_name"/>
    head = soup.find("head")

    meta = soup.new_tag("meta")
    meta.attrs.update({
        "property": "og:locale",
        "content": lang})
    head.insert(6, meta)

    meta = soup.new_tag("meta")
    meta.attrs.update({
        "property": "og:site_name",
        "content": "idee" if lang.startswith("de") else "concept"})
    head.insert(6, meta)
```

Function title()

The title is also inserted in multiple locations, in a title tag in the header and in a meta tag in the HTML head.

display.py - title()

```
def title(soup):
    """
    Move the title to the location we want it to have and fill also the
    corresponding meta data tag with the title information.
    """
    h1 = soup.find("h1")
    t = None
    if h1:
        t = h1.text
        h1.decompose()
    else:
        t = "No title found"

    # target header structure: check div exists, otherwise create
    # <header><h1></h1><div><time></time><address></address></div></header>
    header = soup.find("header")
    h1 = soup.new_tag("h1")
    h1.string = t
    header.insert(0, h1)

    # meta data
    head = soup.find("head")
    meta = soup.new_tag("meta")
    meta.attrs.update({
        "property": "og:title",
        "content": t})
    head.insert(6, meta)

    titletag = soup.find("title")
    titletag.string = t
```

Since Pandoc already created an `h1` tag, it needs just to be moved into the desired location.

Function dates()

Only two dates are saved in the HTML, the published date, which is put into the meta data and visibly into the article header, and the last modified date.

As I canceled my central meta data storage, I have to look whether an older version of the article exists and read the correct publishing date from that article.

display.py - dates()

```
def dates(soup, workpath, urn):
    """
    Create the meta and also the visual date tags.
    - article:modified_time: current local time
    - article_published_time: current local time if article is published first
      time, otherwise the time is taken from the published article.
    """
    head = soup.find("head")
    d = datetime.now().isoformat()

    meta = soup.new_tag("meta")
    meta.attrs.update({
        "property": "article:modified_time",
        "content": d[:19]})
    head.insert(6, meta)

    htmlpath = workpath /Path(f"idee/website/article/{urn}").with_suffix(".html")
    if htmlpath.exists():
        builder = HTMLParserTreeBuilder()
        with open(htmlpath, 'r', encoding='utf-8') as pf:
            published_html = pf.read()
            pf.close()
        published_soup = BeautifulSoup(published_html, builder=builder)
        timetag = published_soup.find("time", attrs={"pubdate": "true"})
        if timetag:
            d = timetag["datetime"]
        else:
            print("Warning: Published article has no pubdate.")

    meta = soup.new_tag("meta")
    meta.attrs.update({
        "property": "article:published_time",
        "content": d[:19]})
    head.insert(6, meta)

    # target header structure: check div exists, otherwise create
    # <header><h1></h1><div><time></time><address></address></div></header>

    header = soup.find("header")
    div = header.find("div")

    if not div:
        div = soup.new_tag("div")
        header.append(div)

    t = soup.new_tag("time")
    div.insert(0, t)
    t.string = d[:10]
    t.attrs.update({"datetime": d[:19]})
    # probably deprecated by itemprop alternative
    t.attrs.update({"pubdate": "true"})
```

Function tables()

Enables table column width control via %dd% entry in front of the respective column title. Numbers are always interpretated as percentage.

Enables the creation of rowspan for individual cells, by marking the cells to join via ^ .

display.py - tables()

```
def tables(soup):
    """
    1. Look in <th> tags for %dd% entries at the start of the text.
       Use dd for style: with:dd%
    2. Look in <td> tags for lone ^ entries.
       Update the rowspan in the respective <td> tag of the predecessing row.
       Dispose the ^ tag.
    3. Look in <td> tags for lone < entries.
       Update the columnspan in the predecessing <td> tag.
       Dispose the < tags. (implementation pending)
    """

    # 1
    tags = soup.find_all("th")
    for tag in tags:
        text = tag.string
        match = re.match(r'^%(\d{2}%)', text)
        width = match.group(1) if match else None
        if width:
            s = tag.get("style")
            if s:
                s = f"{s} width:{width};"
            else:
                s = f"width:{width}"
            tag.attrs.update({"style": s})
            text = text[4:].strip()  # fits for one and two digits
            tag.string = text

    # 2
    # to prevent index becoming a moving target, iterate backwards
    while True:
        tags = soup.find_all("td", string='^')
        if not tags:
            break
        tag = tags[-1]
        if tag.parent:  # we decompose tags as we go, therefore we have to check
            sibl = tag.parent.find_all("td")
            ## obviously search for index does compare strings, not identity
            index = len(sibl) - 1 - sibl[::-1].index(tag)  # backward search
            pstag = tag.parent.find_previous_sibling("tr")  # parent search tag
            rowspan = 2
            tag.decompose()
            while pstag and pstag.find_all("td")[index].string in '^':
                rowspan += 1
                pstag.find_all("td")[index].decompose()
                pstag = pstag.find_previous_sibling("tr")
            if pstag:
                pstag.find_all("td")[index].attrs.update({"rowspan": f"{rowspan}"})
```

Function article_author()

The author of the article is written into the meta data and visibly into the article header.

If a comment exists, naming an author, this information is used. Otherwise my name is assumed to be correct.

display.py - dates()

```
def article_author(soup):
    """
    Check for author information provided as comment.
    If not, assume Frank Siebert.

    Populate the author information in the soup.
    """
    comments = soup.find_all(string=iscomment)
    tag = None
    a = None
    for comment in comments:
        if 'author:' in comment.text:
            tag = comment
            break
    if tag:
        a = tag.text.split(':')[1].strip()
        tag.decompose()
    else:
        a = "Frank Siebert"

    head = soup.find("head")
    meta = soup.new_tag("meta")
    meta.attrs.update({
        "property": "article:author",
        "content": a})
    head.insert(6, meta)

    # target header structure: check div exists, otherwise create
    # <header><h1></h1><div><time></time><address></address></div></header>
    header = soup.find("header")
    div = header.find("div")

    if not div:
        div = soup.new_tag("div")
        header.append(div)

    address = soup.new_tag("address")
    address.string = a
    div.append(address)
```

Function article_qrcode()

Every article shows the QR-Code of its URL. This makes it easy to open the article also on a mobile, if you first see it on a computer screen or on paper.

The future article location is fully known by its URN.

display.py - article_qrcode()

```
def article_qrcode(soup, workpath, urn):
    """
    Run after author()
    PARAMETER
    soup:
    workpath: The location of the md-file
    urn: unique resource name of the article

    Based on thr urn we know the url the web-page will have on
    idee.frank-siebert.de, and url the qrcode image will have.

    For this url we create the qrcode image, if it does not already exist, and
    show it in the html header.
    """
    qrpath = None
    qrp = None
    for qrp in ["idee/website/qrcode", "qrcode"]:
        qrp = workpath / qrp / urn
        qrp = qrp.with_suffix(".png")
        if qrp.exists():
            qrpath = qrp
            break

    if not qrpath:  # create it
        qrpath = qrp
        docurl = "https://idee.frank-siebert.de/article/" + urn + ".html"
        image = qrcode.make(data=docurl)
        image.save(qrpath)

    # write into the soup, first figure in the second div of the header
    header = soup.find("header")
    div = header.find_all("div")
    if len(div) == 0:
        print("call article_qrcode() after author()")
    elif len(div) == 1:
        div = soup.new_tag("div")
        header.append(div)
    else:
        div = div[1]

    figure = soup.new_tag("figure")
    div.append(figure)

    caption = soup.new_tag("figcaption")
    figure.insert(0, caption)

    anchor = soup.new_tag("a")
    anchor.attrs.update({"href": f"./{os.path.relpath(qrpath, workpath)}"})
    figure.insert(0, anchor)

    img = soup.new_tag("img")
    img.attrs.update({"width": "150px", "height": "150px"})
    img.attrs.update({"src": f"./{os.path.relpath(qrpath, workpath)}"})
    img.attrs.update({"alt": "QR Code"})
    anchor.insert(0, img)
```

Function article_licence()

This function looks, whether a comment with licence information exists, and includes then the respective licence information into the article header.

A dictionary with a respective configuration exists currently only in the python module and contains only two entries for the "creative commons zero" licence. This is also the licence used, if no other licence information is found.

Module variable LICENCES

```
LICENCES = {
        "cc0": [
            {
                "href":
                "./idee/website/legal/creative-commons-cc0-1-0-universal.html",
                "alt": "Creative Commons",
                "img": "./idee/website/image/CC-Icon.png"
            },
            {
                "href":
                "./idee/website/legal/creative-commons-cc0-1-0-universal.html",
                "alt": "Zero",
                "img": "./idee/website/image/CC0-Icon.png"
            }
            ]
        }
```

display.py - article_licence()

```
def article_licence(soup):
    """
    Insert the licence information, if requested by the author via
    <!-- article-licence: cc0 -->. Other licences might be added as required

    TODO: A variant of this method could search for <!-- licence: xy -->
    informations to replace them with licence information for 3d-party content.
    """
    comments = soup.find_all(string=iscomment)
    c = None
    for comment in comments:
        if 'article-licence:' in comment:
            c = comment
            break

    if not c:
        return

    c = c.split(":")[1].strip()

    licences = LICENCES.get(c)
    if not licences:
        print(f"No matching licence information found for {c}.")
        return

    header = soup.find("header")
    div = header.find_all("div")
    if len(div) < 2:
        print("Call article_licence() after article_qrcode()")
        return

    div = div[1]

    for licence in licences:
        anchor = soup.new_tag("a")
        div.append(anchor)
        anchor.attrs.update({"href": licence.get("href")})
        img = soup.new_tag("img")
```

```
        # The following scaling is for the PDF
        # In the browser the CSS overwrites this scaling:
        # newtag.attrs.update({"width": "28px", "height": "28px"})
        img.attrs.update({"src": licence.get("img")})
        img.attrs.update({"alt": licence.get("alt")})
        anchor.append(img)
```

Function article_audio()

This function looks whether a recording exists, sharing the same name as the HTML, just with extension `.mp3` . It looks into the two folders designated for audio files.

If the audio is found, it is added to the article header.

display.py - article_audio()

```
def article_audio(soup, workpath, urn):
    """
    Run after article_licence()
    PARAMETER
    soup:
    workpath: The location of the md-file
    urn: unique resource name of the article

    Based on thr urn we know the url the audio file has to have.

    """
    audiopath = None
    aup = None
    for aup in ["audio", "idee/website/audio"]:
        aup = workpath / aup / urn
        aup = aup.with_suffix(".mp3")
        if aup.exists():
            audiopath = aup
            break

    if not audiopath:
        return

    header = soup.find("header")
    div = header.find_all("div")
    if len(div) < 2:
        print("call article_audio() after article_licence()")
        return

    div = div[1]
    figure = soup.new_tag("figure")
    div.append(figure)
    audio = soup.new_tag("audio")
    figure.append(audio)
    audio.attrs.update({"accesskey": "a",
                        "type": "audio/mp3",
                        "preload": "none",
                        "controls": "true",
                        "src": f"./{os.path.relpath(audiopath, workpath)}"})
```

Function movetoc()

If a comment for the TOC exists, the method moves the TOC generated by Pandoc to the location of the comment, replacing it.

If no comment for the TOC exists, then no TOC was desired and the TOC is removed.

The highest level of the TOC, containing the article title, is removed. The highest level used for headlines inside the articles is <h2> .

display.py - movetoc()

```
def movetoc(soup):
    """
    Move the TOC to the correct location
    The author will set <!-- toc --> if he wants a TOC,
    and he will do it in the location, where it should be.
    """
    toc = soup.find("nav", id='TOC')
    # Eliminate the article title from the toc
    ul1 = toc.find("ul")
    ul2 = ul1.find("ul")
    ul1.replace_with(ul2)
    tag = None
    comments = soup.find_all(string=iscomment)
    for comment in comments:
        if comment in ' toc ':
            tag = comment
            break
    if tag:
        tag.replace_with(toc)
    else:
        toc.decompose()
```

Function footnotes()

The footnotes function moves the references section created by Pandoc to the location of the references comment.

More a style matter, but best done here, is the replacement of the default back-reference symbol with a symbol that indeed points backwards.

display.py - footnotes()

```
def footnotes(soup):
    """
    Move footnotes to the place of the authors choice
    given by <!-- references -->
    Change back reference symbol to reference back and <CR>
    """
    # use a better symbol for backreferences
    tags = soup.find_all("a", string='↩')
    for tag in tags:
        tag.clear()
        tag.append('↑')

    # Footnotes are generated as section  <section class="footnotes"
    # role="doc-endnotes"> - Search Section and use it to replace references comment.
    fnotes = soup.find("section", class_="footnotes")
    if fnotes:
        tag = soup.find(string=iscomment, text='references')
        if tag:
            tag.replace_with(fnotes)
```

article_pdf()

At last the question is, whether PDF-creation is requested. If this is the case, then the PDF and the link to it in the HTML page are created.

display.py - article_pdf()

```
def article_pdf(soup, workpath, urn, htmlpath):
    """
    Checks for <!-- pdf -->, which is the request to create PDF.

    Applies some changes to the html:
    - figure for the audio is senseless in PDF
    - relative URL's need to become absolute.
    - footnotes need written links to make sense on paper
    - creates the pdf
    - places the link to the PDF into the soup
    """
    comments = soup.find_all(string=iscomment)
    c = None
    for comment in comments:
        if 'pdf' in comment:
            c = comment
            break

    if not c:
        return

    csspath = Path(r"/home/frank/projects/idee/website/css/fspdf.css")

    workpath.resolve()
    pdfpath = workpath / "pdf" / urn
    pdfpath = pdfpath.with_suffix(".pdf")

    pdfsoup = pdf_soup(soup)
    page_break(pdfsoup)

    html_doc = pdfsoup.prettify()

    # weasy can't render MathTex, since it doesn't run the required JavaScript
    # let chromium hender MathTex to HTML and create PDF from that
    math = soup.find("span", class_="math")
    if math:
        html_doc = pdf_math(html_doc, workpath, htmlpath)

    weasy_html = HTML(string=html_doc, base_url=str(workpath))
    bpdf = weasy_html.write_pdf(stylesheets=[CSS(filename=str(csspath))])
    with open(pdfpath, 'wb') as outfile:
        outfile.write(bpdf)
        outfile.flush()
        os.fsync(outfile)
        outfile.close()

    # insert pdf symbol and link into the original soup
    link_pdf(soup, urn)
```

As can be seen in the code, pdf creation requires a number of steps, which will be detailed below.

pdf_soup()

The HTML needs a bit adjustment before the PDF is created. To avoid messing up the to be published HTML, a separate BeautifulSoup is created.

display.py - pdf_soup()

```
def pdf_soup(soup):
    """
    Creates a separate soup for the PDF, because there some slight
    differences.

    First of all we need to honor additional software in the generator
    information. Then PDF is not suited for audio and we need to remove that.

    We also change all links into absolute links to make sure, that the
    generator finds all images and cascading style sheets.

    Returns:
        soup
    """
    # create an own soup for pdf
    html = soup.prettify()

    builder = HTMLParserTreeBuilder()
    pdfsoup = BeautifulSoup(html, builder=builder)

    # honor weasy as generator, and also chromium if we render math
    math = soup.find("span", class_="math")
    meta = pdfsoup.find("meta", attrs={"name": "generator"})
    generators = meta["content"]
    if math:
        meta.attrs.update({"name": "generator",
                           "content": f"{generators}, chromium, weasy"})
    else:
        meta.attrs.update({"name": "generator",
                           "content": f"{generators}, weasy"})

    # The article header
    article = pdfsoup.find("article")
    header = article.find("header")

    tag = header.find("figure", attrs={"class": "audio"})
    if tag:
        tag.decompose()

    # We need to change relative paths to own articles into absolute
    # paths.
    rhref = re.compile(r"^\.\.\/")
    anchors = pdfsoup.find_all("a", href=rhref)
    for anchor in anchors:
        url = rhref.sub("https://idee.frank-siebert.de/article/",
                        anchor["href"])
        anchor.attrs.update({"href": url})

    # On paper we need complete written URLs
    rhref = re.compile(r"^http.*")
    tag = pdfsoup.find("section", class_="footnotes")
    if tag:
        anchors = tag.find_all("a", href=rhref)
        for anchor in anchors:
            url = anchor["href"]
            anchor.parent.append(pdfsoup.new_tag("br"))
            anchor.parent.append(url)
    return pdfsoup
```

page_break()

Creating PDF means also, that automatic page breaks might simply look awful. It might e.g. let a headline stand alone at the bottom of one page, to start the chapter at the next page.

As mentioned, page-break comments can be included anywhere in the markdown file to take control of page-breaks in the PDF.

display.py - article_pdf()

```
def page_break(soup):
    """
    Find <!-- page-break --> comments set by the author and replace them
    with <div style="page-break-before: always;"> to control page break
    positions in the pdf
    """
    comments = soup.find_all(string=iscomment)
    for comment in comments:
        if 'page-break' in comment:
            pb = soup.new_tag("div")
            pb.attrs.update({"style": "page-break-before: always"})
            comment.replace_with(pb)
            # if the page-break is in the table, split table
            if pb.parent.name == 'td':
                tr = pb.parent.parent  # should be "tr"
                tbody = tr.parent
                table = tbody.parent
                thead = table.find("thead")
                newthead = copy.copy(thead)
                newtable = soup.new_tag("table")
                table.insert_after(pb)
                pb.insert_after(newtable)
                newtable.append(newthead)
                tbody = soup.new_tag("tbody")
                newtable.append(tbody)
                while (tr_next := tr.find_next_sibling("tr")) is not None:
                    tbody.append(tr_next)
                tbody.insert(0, tr)
```

pdf_math()

I found the resulting PDF most to my satisfaction using weasy to create it. I tried it first with Pandoc, and most probably the results could look the same with that tool, but for now my decision stands with weasy.

The only problem: Weasy does not support JavaScript, which is required to use MathJax to render MathTex into MathML.

Multiple solutions can be imagined to solve this issue, including running JavaScript directly from python, but I thought it a genius idea to use chromium to render MathML.

display.py - pdf_math()

```
def pdf_math(html_doc, workpath, htmlpath):
    """
    Use Chromium to render MathTex into MathML via MathJax, since weasy does
    not support javascript.

    TODO: Implement load and script control for webdriver to know exactly when
    MathJax scripting finished, to get rid of the sleep.

    Returns;
        string - html_doc with rendered MathML
    """
    # we use the same filename as for the final html
    # that way we do not need to delete the temporary file
    with open(htmlpath, 'w', encoding='utf-8') as outfile:
        print(html_doc, file=outfile)
        outfile.flush()
        os.fsync(outfile)
        outfile.close()

    # with webdriver
    chrome_options = Options()
    chrome_service = \
    webdriver.ChromeService(executable_path="/usr/bin/chromedriver")
    chrome_options.add_argument("--headless")
    chrome_options.add_argument("--enable-javascript")
    chrome_options.add_argument("--window.navigator.webdriver=False")
    chrome_options.add_argument("--useAutomationExtension=False")
    chrome_options.add_argument("--disable-blink-features=AutomationControlled")
    chrome_options.add_argument("--user-agent=Mozilla/5.0 (X11; Linux x86_64)" \
            " AppleWebKit/537.36 (KHTML, like Gecko) Chrome/138.0.0.0 " \
            "Safari/537.36")
    chrome_options.add_experimental_option("excludeSwitches",
                                           ["enable-automation"])
    browser = webdriver.Chrome(options=chrome_options, service=chrome_service)

    temp = workpath / htmlpath
    temp = temp.resolve()
    browser.get( f"file://{temp}" )
    # wait till the page finished changing
    time.sleep(3)

    test = ""
    while True:
        html_doc = browser.page_source
        if html_doc in test:
            break
        test = html_doc

    return html_doc
```

I consider the usage of chromium as a workaround, and I always tell everyone, that one workaround incubates the next. And indeed, you see the next workaround directly afterwards as `time.sleep(3)` .

As I'm not religious about workarounds, only cautious, I let this be as it is, for now. But I imagine future changes at this code location.

link_pdf()

At last, after the PDF has been written to the disk, the article HTML needs a link to the PDF in the article header.

display.py - link_pdf()

```
def link_pdf(soup, urn):
    """
    Adds the link to the PDF into the article header, represented by a
    pdfimage.
    """

    pdfimage = "3cd97bab8bb20288768b35fd72979ec3bbf4b2a8.png"

    header = soup.find("header")
    divs = header.find_all("div")
    if len(divs) < 1:
        print("Call article_pdf() last before printing html")
        return

    div = divs[1]

    figure = soup.new_tag("figure")
    div.insert(1, figure)
    anchor = soup.new_tag("a")
    figure.append(anchor)

    anchor.attrs.update({"accesskey": "p",
                         # "download": "",
                         "href": f"./pdf/{urn}.pdf",
                         "target": "_blank",
                         "type": "application/pdf"
                         })

    # Inject the PDF Icon
    img = soup.new_tag("img")
    anchor.append(img)
    img.attrs.update({"src": f"./idee/website/image/{pdfimage}"})
```

This is really the last thing done before the article HTML is written to the disk and shown in the Firefox web-browser.

If everything is copy-edited and all page-breaks are nicely set for the PDF, then the next to do is the handover to the deployment via git.

IdeePublish

The handover to the deployment is done with the command `IdeePublish` .

publish.py - imports and debugging

```
#!/user/bin/python3
"""
Publish generated HTML and attached assets to the idee website git.

@date: 2026-02-05
"""

import sys
import os
import string
import shutil
import getopt
from pathlib import Path
from bs4 import BeautifulSoup
from bs4.builder._htmlparser import HTMLParserTreeBuilder

...

if __name__ == "__main__":

    MARKDOWNFILE = None

    try:
        opts, args = getopt.getopt(sys.argv[1:], ["o"])

    except getopt.GetoptError:
        print("No Parameter given")
        sys.exit(2)

    if len(args) == 0:
        print("No Parameter given")
        sys.exit(2)

    MARKDOWNFILE = args[0]
    if not MARKDOWNFILE:
        print("No Parameter given")
        sys.exit(2)

    publish(Path(MARKDOWNFILE))

    sys.exit(0)
```

The code shows the imports and the `__main__` part used to call the publish function from terminal for e.g. debug purposes. The ellipsis shown in the middle of this code snipped stand for all the functions in between, which in the following will be shown one by one.

Function publish()

The publish function has not very much to do. It boils down to the following steps:

- adjust resource links to fit the target location
- move the assets to the target location
- copy the markdown file to the author directory of the idee website project.

The last step is important to have the source of the published article available for corrections, which might be required later.

Often something is found to correct directly after the publishing. Copy-editing sometimes fails, especially if the author is also the copy-editor, as in my case. I therefore decided not to cleanup the directory where authoring took place, but to delete that manually some days after publishing.

publish.py - publish()

```
def publish(filepath: string):
    """
    Creates an html web page frm the markdown file.

    The html web page is fully flavored for that site, including the optional
    linked in pdf and audio versions and content licence information.

    Returns
    -------
    None.
    """
    mdpath = Path(filepath)
    workdirpath = mdpath.parents[0]

    # check that this a designated workdirectory
    idee = workdirpath / "idee"
    if not idee.exists():
        print("Use first IdeeFolders to designate this location as"
                " workdirectory, and use IdeeDisplay to generate idee-website"
                " styled HTML"
              )
        sys.exit(0)

    htmlpath = workdirpath / mdpath.stem
    htmlpath = htmlpath.with_suffix(".html")
    # read html
    if htmlpath.exists():
        builder = HTMLParserTreeBuilder()
        with open(htmlpath, 'r', encoding='utf-8') as pf:
            published_html = pf.read()
            pf.close()
    else:
        print("Use IdeeDisplay to generate idee-website style HTML first")
        sys.exit(0)

    soup = BeautifulSoup(published_html, builder=builder)
    resource_paths(soup)
    html_doc = soup.prettify()

    outpath = workdirpath / "idee" / "website" / "article" / mdpath.stem
    outpath = outpath.with_suffix(".html")
    with open(outpath, "w", encoding='utf-8') as outfile:
        print(html_doc, file=outfile)
        outfile.flush()
        outfile.close()

    copy_assets(workdirpath)
    # copy the markdown file itself also
    shutil.copy(mdpath, idee / "author")

    print(f"Published to: {outpath}")
```

Function resource_paths()

Adjusts the relative paths to the articles assets as they are required by the web-sites directory structure.

publish.py - resource_paths()

```
def resource_paths(soup):
    """
    Change the resource paths to fit for the idee website
    """
    for i in ["src", "href"]:
        tags = soup.find_all(attrs={i: True})
        for tag in tags:
            p = tag[i]
            p = p.replace("./idee/website/", "./")
            p = p.replace("./", "../")
            tag.attrs.update({i: p})
```

Function copy_assets()

The function `copy_assets()` copies the assets in the authoring subfolders to their target location in the web-site folder structure.

publish.py - copy_assets()

```
def copy_assets(workpath):
    """
    Copy the assets located in subfolders to the respective subfolder in the
    idee-website
    """
    for d in ["audio", "files", "image", "js", "pdf", "qrcode"]:
        wd = workpath / d
        wtd = workpath / "idee" / "website" / d
        for f in os.listdir(wd):
            wdf = wd / f
            if os.path.isfile(wdf):
                shutil.copy(wdf, wtd)
```

Changes in vimrc

There is just one more thing to mention. My vimrc file, which by default is in the folder `~/.vim/` , got some additional lines defining a key mapping in normal- as well as in insert-mode to run the command `IdeeMeta` .

```
" Markdown autocmd for idee ftplugin
augroup md
  autocmd!
  autocmd FileType markdown imap <C-y> <C-O>:IdeeMeta<CR>
  autocmd FileType markdown nmap <C-y> :IdeeMeta<CR>
augroup END
```

This makes it very smooth to insert the comments used for meta data and to control aspects of the HTML/PDF creation.

Deployment

Deployment is a separate topic. I just teaser here, that I commit everything placed into the idee web-site project into git. During this commit RSS-feed, an archive page for the month, the index page of the web-site and the sitemap.xml are updated.

Those need then a second commit. It seems not possible to prevent that.

After pushing to the server-git, a client-git on the web-server is triggered automatically to fetch the latest changes, by which these become life in the web-site.

Two commits and one push to set a new or updated article life.

That's the topic of a future article, stay tuned.

Final Notes

Obviously this code is highly specialized for my own purposes. It is not meant to be copied, pasted and used as is in any other setting.

It shows, how I adjusted my own workspace to my own needs, which tools I used for this and how I utilized them.

I'm pretty sure it can be adjusted to serve the needs of someone else, and it could probably also be developed further to become customizable. At the present time I have no plans to do this.

However, I hope this to be a showcase, how to get rid of all the 3rd-party scripts loaded from 3rd-party servers doing not much more than exposing web-site visitors to private data sellers.

If you think it worth to derive a customizable version of this, usable by a broader audience, go ahead and do it. Its creative commons zero after all.

Or just ask. If someone asks for it, I have an incentive to go on with it.

Footnotes

1. **The Cosmological Constant as Event Horizon** ; Enrique Gaztanaga; Symmetry, volume 14; Multidisciplinary Digital Publishing Institute; DOI: **https://doi.org/10.3390/sym14020300** ; 2022-02-01 ↑
   https://www.mdpi.com/2073-8994/14/2/300
   https://doi.org/10.3390/sym14020300
2. **XClacksOverhead.org** ; XClacksOverhead.org; X-Clacks-Overhead ↑
   https://xclacksoverhead.org/home/about