New Blog

Meta Post (not really related to Donald Knuth)

Yann Esposito

[2019-08-17 Sat 16:00]

tl;dr: The first blog post of a blog should certainly be about the blog itself to provide a feeling of self-reference.

Peaceful and Respectful Website

There is a trend about website being quite less accessible, using more and more resources, adding trackers, popups, videos, animations, big js frameworks, etc...

I wanted a more peaceful and respectful website.

That website was created with the following constraints in mind by order of priority:

1. **Respect Privacy**; no tracker of any sort (no ads, no google analytics, no referrer for all external links, etc...)
2. **nearly no javascript**; no js at all except for a single exception, pages containing Math formula are displayed using mathjax. That means that event the CSS theme switcher does not use javascript.
3. **Accessible**; should be easy to read on a text browser so people with disabilities could easily consume it
4. **nerdy**; should feel mostly like markdown text in a terminal and source code should be syntax highlighted.
5. **theme switchable**; support your preferred light/dark theme by default but you can change it if you want.
6. **rss**; you should be able to get informed when I add a new blog post.
7. **frugal**; try to minimize the resources needed to visit my website; no javascript, no web-font, not too much CSS magic, not much images or really compressed one.

Some of the constraints are straightforward to get, some not.

You can also check that not using more resources one of the theme of my website looks quite more classical and modern that my preferred ones.

Respect Privacy

The one should be easy, simply not put any 3rd party inclusion in my website. So, no external CSS in my headers, no link to any image I do not host myself. No 3rd party javascript.

Javascript Free

I do not really see why a content oriented website should need to execute javascript.

Accessible

A good way to check that a website is friendly to disabled people is by looking at it with a text browser. If you open most website today you see that at the top of the page is crippled with a numerous number of links/metas info used for javascript tricks, login/logout buttons, etc... The website should only contain, a pretty minimal menu to navigate, and the content.

Nerdy

The feel of the website should be nerdy, it should look like reading a terminal or emacs. It should almost feel the same as if you were using a text-browser. For sensible people, I added a "modern" theme that should better suit modern eye, still the first design should always be the terminal looking one.

Theme switchable

Even if you are not used to disability friendly browser. The website should try to guess your preferred way to consume my website. Recently we dark/light themes were integrated as a new CSS feature. This website should propose your apparently preferred theme. But you could also change it manually.

RSS

This is another layer that help you consume my website as you prefer. You should at least be informed a new article has been published.

Frugal

This one is a bit tricky. It would mean, that visiting my website should not consume much resources. Mainly, this would prevent using heavy medias as much as possible. So, no video, no animated gif, no image if possible or very compressed small one. So I have a script that convert all images to maximize site to '800x800' and use at max 16 colors. On my current example image the size goes from 3.1MB to 88KB.

How

CSS

Regarding CSS, I always found that the default text display by navigator is terrible. So just to ”fix” a minimal CSS to have something bearable it takes me about 120 lines of CSS.

By fixing I mean things like using a fixed line width for text (there is an optimal range to improve legibility). Also having correct list indentation, line-height and font-size. Table displaying correctly.

Then I have about 90 lines of CSS to make my HTML look like text source of a markdown.

Then I set a few CSS rules to handle the ids and classes added by org-export as instead of using the ubiquitous Markdown, I prefer greatly to use org mode files. I need 60 lines of CSS for them.

In order to handle color themes (5 at the time of writing those lines) I use almost 350 line to handle those.

CSS Theme selection

One thing that wasn't straightforward while writing the CSS was to provide an interactive theme selector without any javascript involved. That theme switcher is really the limit I can concede to modern standards because it is CSS only.

The trick is to provide one top-level element per theme at the beginning of the body of the HTML. Then hide those elements (I chose inputs). Finally provide a set of anchor links.

```
  ...
<body>
   <input id="light"/>
   <input id="dark"/>
   <div id="labels">
     Change theme:
     <a href="#light">Light</a>
     <a href="#dark">Dark</a>
   </div>
   <div class="main">
   ALL YOUR CONTENT HERE
   </div>
 </body>
```

Then use the *sibling* CSS selector ~. Then put all your content in a div of class `.main` for example. Finally in the CSS you can write things like:

```
/* hide all radio button that are not inside another div of body */
```

```
body > input {
  display: none;
}
:root {
  --light-color: #fff;
  --dark-color: #000;
}
input#light:checked ~ .main {
  background-color: var(--light-color);
  color: var(--dark-color);
}
input#dark:checked ~ .main {
  background-color: var(--dark-color);
  color: var(--light-color);
}
```

I previously used checkbox inputs but using URL fragment feels better.

Blog Engine - org-mode with org-publish

So publishing a website is something that could go from. Write your own HTML each time. But this is quite tedious, so we generally all use a website generator. The next thing with the minimal possible amount of work is using org-mode with org-publish. Because a website is mostly, export all of file in org-mode format (easier to write and manipulate than raw HTML) to HTML.

In fact, there are numerous details that make this task not this straightforward. You want:

1. from a tree of org-mode files, generate an equivalent file tree of HTML files generated from the org-mode files. This is the main purpose of org-publish.
2. We also want to set specific headers, a CSS file, a favicon, link to RSS file, mobile friendly directives.
3. Have common header/footer (preamble, postamble) if possible with a menu.
4. An archive page with a list of posts.
5. Generate an RSS file
6. Niceties:
   - obfuscate your email to prevent spam
   - link to your email with a link to the current page integrated in the body/subject
   - replace your external link to open in a new tab securely (`noopener / noreferrer`).
   - compress images during publishing

Also, a single detail make using org-publish a bit awkward compared to classical other classical static website generators; it is designed to be set in you full emacs configuration. But I wanted to be able to clone my git repository and be able to generate my website locally even if I clone it on different directories.

So I created a package just for that: Autoload eLISP file in projects.

Tree of files

There is a first pass that use `projectile` emacs package to detect the current root file of the project and provide a list of absolute paths.

Then you set the associative list `org-publish-project-alist` with many straightforward details. The source directory, the destination directory, but also, file to exclude, a function used to transform org files to HTML, etc...

```
(setq domainname "https://john.doe")
(setq base-dir (concat (projectile-project-root) "src"))
(setq publish-dir (concat (projectile-project-root) "_site"))
(setq assets-dir (concat base-dir "/"))
(setq publish-assets-dir (concat publish-dir "/"))
(setq rss-dir base-dir)
(setq rss-title "Subscribe to articles")
(setq publish-rss-dir publish-dir)
(setq publish-rss-img (concat domainname "/rss.png"))
(setq css-path "/css/min.css")
(setq author-name "John Doe")
(setq author-email "john@doe.com")

(require 'org)
(require 'ox-publish)
(require 'ox-html)
(require 'org-element)
(require 'ox-rss)

(setq org-link-file-path-type 'relative)
(setq org-publish-timestamp-directory
      (concat (projectile-project-root) "_cache/"))

(setq org-publish-project-alist
      `(("orgfiles"
         :base-directory ,base-dir
         :exclude ".*drafts/.*"
         :base-extension "org"
         :publishing-directory ,publish-dir
         :recursive t
         :publishing-function org-blog-publish-to-html

         :with-toc nil
```

```
 :with-title nil
 :with-date t
 :section-numbers nil
 :html-doctype "html5"
 :html-html5-fancy t
 :html-head-include-default-style nil
 :html-head-include-scripts nil
 :htmlized-source t
 :html-head-extra ,org-blog-head
 :html-preamble org-blog-preamble
 :html-postamble org-blog-postamble

 :auto-sitemap t
 :sitemap-filename "archive.org"
 :sitemap-title "Blog Posts"
 :sitemap-style list
 :sitemap-sort-files anti-chronologically
 :sitemap-format-entry org-blog-sitemap-format-entry
 :sitemap-function org-blog-sitemap-function)

("assets"
 :base-directory ,assets-dir
 :base-extension ".*"
 :exclude ".*\.org$"
 :publishing-directory ,publish-assets-dir
 :publishing-function org-blog-publish-attachment
 :recursive t)

("rss"
 :base-directory ,rss-dir
 :base-extension "org"
 :html-link-home ,domainname
 :html-link-use-abs-url t
 :rss-extension "xml"
 :rss-image-url ,publish-rss-img
 :publishing-directory ,publish-rss-dir
 :publishing-function (org-rss-publish-to-rss)
 :exclude ".*"
 :include ("archive.org")
 :section-numbers nil
 :table-of-contents nil)

("blog" :components ("orgfiles" "assets" "rss"))))
```

HTML Headers

I set the header to provide a link to the RSS file, the CSS, the favicon and viewport directive for mobile browsers.

```
(defvar org-blog-head
  (concat
   "<link rel=\"stylesheet\" type=\"text/css\" href=\"" css-path "\"/>"
   "<meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0\">"
       "<link  rel=\"alternative\"  type=\"application/rss+xml\"  title=\""  rss-
title "\" href=\"/archives.xml\" />"
   "<link rel=\"shortcut icon\" type=\"image/x-icon\" href=\"/favicon.ico\">"))
```

Preamble & Postamble

So I put a menu in both the preamble and postamble. The postamble contains a lot of details about the article, author, email, date, etc...

```
(defun menu (lst)
    "Blog menu"
    (concat
     "<nav>"
     (mapconcat 'identity
                (append
                 '("<a href=\"/index.html\">Home</a>"
                   "<a href=\"/archive.html\">Posts</a>"
                   "<a href=\"/about-me.html\">About</a>")
                 lst)
                " | ")
     "</nav>"))

  (defun get-from-info (info k)
    (let ((i (car (plist-get info k))))
      (when (and i (stringp i))
        i)))

  (defun org-blog-preamble (info)
    "Pre-amble for whole blog."
    (concat
     "<div class=\"content\">"
     (menu '("<a href=\"#postamble\">↓ bottom ↓</a>"))
     "<h1>"
     (format "%s" (car (plist-get info :title)))
     "</h1>"
     (when-let ((date (plist-get info :date)))
       (format "<span class=\"article-date\">%s</span>"
             (format-time-string "%Y-%m-%d"
                                 (org-timestamp-to-time
                                  (car date)))))
   (when-let ((subtitle (car (plist-get info :subtitle))))
     (format "<h2>%s</h2>" subtitle))
   "</div>"))

(defun org-blog-postamble (info)
  "Post-amble for whole blog."
  (concat
   "<div class=\"content\">"
   "<footer>"
   (when-let ((author (get-from-info info :author)))
     (if-let ((email (plist-get info :email)))
         (let* ((obfs-email (obfuscate-html email))
                (obfs-author (obfuscate-html author))
                (obfs-title (obfuscate-html (get-from-info info :title)))
                (full-email (format "%s &lt;%s&gt;" obfs-author obfs-email)))
        (format "<div class=\"author\">Author: <a href=\"%s%s%s%s\">%s</a></div>"
                   (obfuscate-html "mailto:")
                   full-email
                   (obfuscate-html "?subject=yblog: ")
                   obfs-title
                   full-email))
       (format "<div class=\"author\">Author: %s</div>" author)))
   (when-let ((date (plist-get info :date)))
     (format "<div class=\"date\">Created: %s</div>"
             (format-time-string "%Y-%m-%d"
                                 (org-timestamp-to-time
                                  (car date)))))
   (when-let ((keywords (plist-get info :keywords)))
    (format "<div class=\"keywords\">Keywords: <code>%s</code></div>" keywords))
   (format "<div class=\"date\">Generated: %s</div>"
           (format-time-string "%Y-%m-%d %H:%M:%S"))
   (format (concat "<div class=\"creator\"> Generated with "
            "<a href=\"https://www.gnu.org/software/emacs/\" target=\"_blank\" rel=\"noopener noreferrer\">Emacs %
            "<a href=\"http://spacemacs.org\" target=\"_blank\" rel=\"noopener noreferrer\">Spacemacs %s</a>, "
            "<a href=\"http://orgmode.org\" target=\"_blank\" rel=\"noopener noreferrer\">Org Mode %s</a>"
                   "</div>")
           emacs-version spacemacs-version org-version)
   "</footer>"
   (menu '("<a href=\"#preamble\">↑ Top ↑</a>"))
   "</div>"))
```

Obfuscate email

A simple function to obfuscate HTML by using hexadecimal and octal notation.

```
(defun rand-obfs (c)
  (let ((r (% (random) 20)))
    (cond ;; ((eq 0 r) (format "%c" c))
     ((<= 0 r 10) (format "&#%d;" c))
     (t (format "&#x%X;" c)))))

(defun obfuscate-html (txt)
  (apply 'concat
         (mapcar 'rand-obfs txt)))
```

Specific email subject

You can set a subject to an email when you click on it by writing a link that looks like:

```
mailto:john@doe.com?subject=the-subject
```

Of course most of it is obfuscated.

```
(let* ((obfs-email (obfuscate-html email))
       (obfs-author (obfuscate-html author))
       (obfs-title (obfuscate-html (get-from-info info :title)))
       (full-email (format "%s &lt;%s&gt;" obfs-author obfs-email)))
  (format "<div class=\"author\">Author: <a href=\"%s%s%s%s\">%s</a></div>"
          (obfuscate-html "mailto:")
          full-email
          (obfuscate-html "?subject=yblog: ")
          obfs-title
          full-email))
```

Nice external links

Also, why not fix our external link (see this article as reference):

```
;; add target=_blank and rel="noopener noreferrer" to all links by default
(defun my-org-export-add-target-blank-to-http-links (text backend info)
  "Add target=\"_blank\" to external links."
  (when (and
         (org-export-derived-backend-p backend 'html)
         (string-match "href=\"http[^\"]+" text)
         (not (string-match "target=\"" text))
         (not (string-match (concat "href=\"" domainname "[^\"]*") text)))
    (string-match "<a " text)
   (replace-match "<a target=\"_blank\" rel=\"noopener noreferrer\" " nil nil text)))
```

```
(add-to-list 'org-export-filter-link-functions
             'my-org-export-add-target-blank-to-http-links)
```

Image compression

to compress images I use imagemagick like this:

```
convert src/a.png -resize 800x800\> +dither -colors 16 -depth 4 dest/a.png
```

For example:

I compress automatically images during publishing with:

```
(defun org-blog-publish-attachment (plist filename pub-dir)
  "Publish a file with no transformation of any kind.
FILENAME is the filename of the Org file to be published.  PLIST
is the property list for the given project.  PUB-DIR is the
publishing directory.
Take care of minimizing the pictures using imagemagick.
Return output file name."
  (unless (file-directory-p pub-dir)
    (make-directory pub-dir t))
  (or (equal (expand-file-name (file-name-directory filename))
             (file-name-as-directory (expand-file-name pub-dir)))
     (let ((dst-file (expand-file-name (file-name-nondirectory filename) pub-dir)))
        (if (string-match-p ".*\\.\\(png\\|jpg\\|gif\\)$" filename)
            (shell-command
             (format "convert %s -resize 800x800\\> +dither -colors 16 -depth 4 %s"
                     filename
                     dst-file))
          (copy-file filename dst-file t)))))
```

Full code

Here is the full code:

```
  (setq domainname "https://john.doe")
  (setq base-dir (concat (projectile-project-root) "src"))
  (setq publish-dir (concat (projectile-project-root) "_site"))
  (setq assets-dir (concat base-dir "/"))
  (setq publish-assets-dir (concat publish-dir "/"))
  (setq rss-dir base-dir)
  (setq rss-title "Subscribe to articles")
  (setq publish-rss-dir publish-dir)
  (setq publish-rss-img (concat domainname "/rss.png"))
  (setq css-path "/css/min.css")
  (setq author-name "John Doe")
  (setq author-email "john@doe.com")
  (require 'org)
  (require 'ox-publish)
  (require 'ox-html)
  (require 'org-element)
  (require 'ox-rss)

  (setq org-link-file-path-type 'relative)
  (setq org-publish-timestamp-directory
        (concat (projectile-project-root) "_cache/"))

  (defvar org-blog-head
    (concat
     "<link rel=\"stylesheet\" type=\"text/css\" href=\"" css-path "\"/>"
     "<meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0\">"
         "<link  rel=\"alternative\"  type=\"application/rss+xml\"  title=\""  rss-
title "\" href=\"/archives.xml\" />"
     "<link rel=\"shortcut icon\" type=\"image/x-icon\" href=\"/favicon.ico\">"))

  (defun menu (lst)
    "Blog menu"
    (concat
     "<nav>"
     (mapconcat 'identity
                (append
                 '("<a href=\"/index.html\">Home</a>"
                   "<a href=\"/archive.html\">Posts</a>"
                   "<a href=\"/about-me.html\">About</a>")
                 lst)
                " | ")
     "</nav>"))

  (defun get-from-info (info k)
    (let ((i (car (plist-get info k))))
      (when (and i (stringp i))
        i)))

  (defun org-blog-preamble (info)
    "Pre-amble for whole blog."
    (concat
     "<div class=\"content\">"
     (menu '("<a href=\"#postamble\">↓ bottom ↓</a>"))
     "<h1>"
     (format "%s" (car (plist-get info :title)))
     "</h1>"
     (when-let ((date (plist-get info :date)))
       (format "<span class=\"article-date\">%s</span>"
                (format-time-string "%Y-%m-%d"
                                    (org-timestamp-to-time
                                     (car date)))))
      (when-let ((subtitle (car (plist-get info :subtitle))))
        (format "<h2>%s</h2>" subtitle))
      "</div>"))

   (defun rand-obfs (c)
     (let ((r (% (random) 20)))
       (cond ;; ((eq 0 r) (format "%c" c))
        ((<= 0 r 10) (format "&#%d;" c))
        (t (format "&#x%X;" c)))))

   (defun obfuscate-html (txt)
     (apply 'concat
            (mapcar 'rand-obfs txt)))

   (defun org-blog-postamble (info)
     "Post-amble for whole blog."
     (concat
      "<div class=\"content\">"
      ;; TODO install a comment system
        ;; (let ((url (format "%s%s" domainname (replace-regexp-in-string base-
 dir "" (plist-get info :input-file)))))
      ;;   (format "<a href=\"https://comments.esy.fun/slug/%s\">comment</a>"
      ;;           (url-hexify-string url)))
      "<footer>"
      (when-let ((author (get-from-info info :author)))
        (if-let ((email (plist-get info :email)))
            (let* ((obfs-email (obfuscate-html email))
                   (obfs-author (obfuscate-html author))
                   (obfs-title (obfuscate-html (get-from-info info :title)))
                   (full-email (format "%s &lt;%s&gt;" obfs-author obfs-email)))
             (format "<div class=\"author\">Author: <a href=\"%s%s%s%s\">%s</a></div>"
                      (obfuscate-html "mailto:")
                      full-email
                      (obfuscate-html "?subject=yblog: ")
                      obfs-title
                      full-email))
            (format "<div class=\"author\">Author: %s</div>" author)))
      (when-let ((date (plist-get info :date)))
        (format "<div class=\"date\">Created: %s</div>"
                (format-time-string "%Y-%m-%d"
                                    (org-timestamp-to-time
                                     (car date)))))
      (when-let ((keywords (plist-get info :keywords)))
      (format "<div class=\"keywords\">Keywords: <code>%s</code></div>" keywords))
     (format "<div class=\"date\">Generated: %s</div>"
             (format-time-string "%Y-%m-%d %H:%M:%S"))
     (format (concat "<div class=\"creator\"> Generated with "
              "<a href=\"https://www.gnu.org/software/emacs/\" target=\"_blank\" rel=\"noopener noreferrer\">Emacs %
              "<a href=\"http://spacemacs.org\" target=\"_blank\" rel=\"noopener noreferrer\">Spacemacs %s</a>, "
              "<a href=\"http://orgmode.org\" target=\"_blank\" rel=\"noopener noreferrer\">Org Mode %s</a>"
                     "</div>")
             emacs-version spacemacs-version org-version)
     "</footer>"
     (menu '("<a href=\"#preamble\">↑ Top ↑</a>"))
     "</div>"))

  (defun org-blog-sitemap-format-entry (entry _style project)
    "Return string for each ENTRY in PROJECT."
    (when (s-starts-with-p "posts/" entry)
      (format (concat "@@html:<span class=\"archive-item\">"
                      "<span class=\"archive-date\">@@ %s: @@html:</span>@@"
                      " [[file:%s][%s]]"
                      " @@html:</span>@@")
             (format-time-string "%Y-%m-%d" (org-publish-find-date entry project))
              entry
              (org-publish-find-title entry project))))

  (defun org-blog-sitemap-function (title list)
     "Return sitemap using TITLE and LIST returned by `org-blog-sitemap-format-
entry'."
    (concat "#+TITLE: " title "\n"
            "#+AUTHOR: " author-name "\n"
            "#+EMAIL: " author-email "\n"
            "\n#+begin_archive\n"
            (mapconcat (lambda (li)
                         (format "@@html:<li>@@ %s @@html:</li>@@" (car li)))
                       (seq-filter #'car (cdr list))
                       "\n")
            "\n#+end_archive\n"))

  (defun org-blog-publish-to-html (plist filename pub-dir)
    "Same as `org-html-publish-to-html' but modifies html before finishing."
    (let ((file-path (org-html-publish-to-html plist filename pub-dir)))
      (with-current-buffer (find-file-noselect file-path)
        (goto-char (point-min))
        (search-forward "<body>")
        (insert (mapconcat 'identity
                           '("<input type=\"radio\" id=\"light\" name=\"theme\"/>"
                             "<input type=\"radio\" id=\"dark\" name=\"theme\"/>"
```

```
                              "<input type=\"radio\" id=\"raw\" name=\"theme\"/>"
                            "<input type=\"radio\" id=\"darkraw\" name=\"theme\"/>"
                              "<div id=\"labels\">"
                              "<div class=\"content\">"
                              "Change theme: "
                              "<label for=\"light\">Light</label>"
                              "(<label for=\"raw\">raw</label>)"
                              " / "
                              "<label for=\"dark\">Dark</label>"
                              "(<label for=\"darkraw\">raw</label>)"
                              "</div>"
                              "</div>"
                              "<div class=\"main\">")
                            "\n"))
        (goto-char (point-max))
        (search-backward "</body>")
        (insert "\n</div>\n")
        (save-buffer)
        (kill-buffer))
      file-path))

(defun org-blog-publish-attachment (plist filename pub-dir)
    "Publish a file with no transformation of any kind.
  FILENAME is the filename of the Org file to be published.  PLIST
  is the property list for the given project.  PUB-DIR is the
  publishing directory.
  Take care of minimizing the pictures using imagemagick.
  Return output file name."
    (unless (file-directory-p pub-dir)
      (make-directory pub-dir t))
    (or (equal (expand-file-name (file-name-directory filename))
               (file-name-as-directory (expand-file-name pub-dir)))
         (let ((dst-file (expand-file-name (file-name-nondirectory filename) pub-
dir)))
          (if (string-match-p ".*\\.\\(png\\|jpg\\|gif\\)$" filename)
              (shell-command
             (format "convert %s -resize 800x800\\> +dither -colors 16 -depth 4 %s"
                       filename
                       dst-file))
            (copy-file filename dst-file t)))))

  (setq org-publish-project-alist
        `(("orgfiles"
           :base-directory ,base-dir
           :exclude ".*drafts/.*"
           :base-extension "org"
```

```
 :publishing-directory ,publish-dir

 :recursive t
 :publishing-function org-blog-publish-to-html

 :with-toc nil
 :with-title nil
 :with-date t
 :section-numbers nil
 :html-doctype "html5"
 :html-html5-fancy t
 :html-head-include-default-style nil
 :html-head-include-scripts nil
 :htmlized-source t
 :html-head-extra ,org-blog-head
 :html-preamble org-blog-preamble
 :html-postamble org-blog-postamble

 :auto-sitemap t
 :sitemap-filename "archive.org"
 :sitemap-title "Blog Posts"
 :sitemap-style list
 :sitemap-sort-files anti-chronologically
 :sitemap-format-entry org-blog-sitemap-format-entry
 :sitemap-function org-blog-sitemap-function)

("assets"
 :base-directory ,assets-dir
 :base-extension ".*"
 :exclude ".*\.org$"
 :publishing-directory ,publish-assets-dir
 :publishing-function org-blog-publish-attachment
 :recursive t)

("rss"
 :base-directory ,rss-dir
 :base-extension "org"
 :html-link-home ,domainname
 :html-link-use-abs-url t
 :rss-extension "xml"
 :rss-image-url ,publish-rss-img
 :publishing-directory ,publish-rss-dir
 :publishing-function (org-rss-publish-to-rss)
 :exclude ".*"
 :include ("archive.org")
 :section-numbers nil
```

```
         :table-of-contents nil)

        ("blog" :components ("orgfiles" "assets" "rss"))))

;; add target=_blank and rel="noopener noreferrer" to all links by default
(defun my-org-export-add-target-blank-to-http-links (text backend info)
  "Add target=\"_blank\" to external links."
  (when (and
         (org-export-derived-backend-p backend 'html)
         (string-match "href=\"http[^\"]+" text)
         (not (string-match "target=\"" text))
         (not (string-match (concat "href=\"" domainname "[^\"]*") text)))
    (string-match "<a " text)
  (replace-match "<a target=\"_blank\" rel=\"noopener noreferrer\" " nil nil text)))

(add-to-list 'org-export-filter-link-functions
             'my-org-export-add-target-blank-to-http-links)
```