

Resource Saving Minimalism



Tools
Scientifically Approved

pjs64



Blog-roller ·
pjs64.wordpress.com

No-Kite-Surfer ·

Org-mode Rookie
2022 Somewhere

Contents

I	Org to Website	5
1.	Intro	5
2.	Test, One, Two	7
3.	Org Publish	8
4.	Example Selection	10
4.1.	Pages	12
5.	Publishing Experiments	13
5.1.	Producing the Laboratory Files	13
5.2.	Analyze It	15
5.3.	Resulting Configuration	17
6.	Website Template	18
7.	Org HTML Export	20
7.1.	Aside	22
7.2.	Headline Levels	23
8.	Org CSS Construction	25
9.	Glue HTML and Extract Images	28
10.	Netlify Drop	29
11.	Test, One, two, three	30
11.1.	Expanded Configuration	31
11.2.	Bibliography and PDF	34
11.2.1.	Compiling Latex	35
11.2.2.	BibTeX HTML	36
11.2.3.	SymLink Publishing Adaption	37
11.3.	Glue HTML	39

11.3.1. Paths and Files	39
11.3.2. Select Updated Pages	40
11.3.3. Page Assembly	40
11.4. Image Handling	41
11.4.1. Change Relative Image Links	41
11.4.2. Collect Media	43
11.4.3. Hidden Agenda for Images	44
12. What's more	46
12.1. Publishing Hooks	47
12.2. Index Toc Tag Category Bibentry Link-List	48
12.2.1. BibTeX	49
12.2.2. Links and Indices	50
12.2.3. XML Tools for Attribution Retrieval	52
12.3. Select Files and Content	58
12.4. Template Development	62
12.5. Netlify Alternatives	64
12.6. Weaving Amaya	65
13. Appendix	69
13.1. Ogbe's Website Construction	69
13.1.1. Publishing Action	71
13.1.2. Head	72
13.1.3. Mathjax	73
13.1.4. Pre- and Postamble → Header and Footer	74
13.1.5. Home Up	75
13.1.6. Footnote	75
13.1.7. Pre- and Post-Processors	76
13.1.8. Options, Drawer	77
13.1.9. Sitemap	77
13.1.10. RSS	78

Part I

Org to Website

1 Intro

Just another ORG to HUGO JEKYLL whatever guide? Guess again. But you also may read first. Exe-cute-if some-hurry: How to utilize ORG publish for a modern Website layout, including template and integrated CSS production; with a helping hand from R.

The motivation for this work was to reproduce and exercise the publishing **features** of ORG MODE. The objective turned from information architecture into taming Web technology for public interests.



This is a volatile expansion of [20]. You may post comments there and refer to the volatile expansion of 2022-08-02. I tried to avoid repetition, unless I turned inline links to bibliography citations.

The tangible result of this expansion is still a static web site without JAVASCRIPT,¹ based on HTML5 and something that might be called CSS3.² The deployment is aimed at content delivery network, CDN, hosting of static sites. And now it is accessible by an URL with the restriction of volatility; see Section 11 for the volatile connotation.

It is still hard to sort out all the detours and excursions; and to justify temporary simplifications. In the long version I rather report on some of the many aspects of Web technology approaches instead of a clean how-to with a straight line of conduct. I consider the result as my own lookup source for further work on free software solutions,

¹At first I only aim at switching off all JAVASCRIPT which is rendered superfluous with new CCS mechanics. Afterwards, but still not here, JAVASCRIPT and HTML API's are considered to be reinforced again.

²Due to *CSS: The Definite Guide* [11], p.2, "it's hard to speak of a single 'CSS3 specification.' There isn't any such thing, nor can there be."

beginning with a free publishing suite. And I publish it because sharing is the new having.

The research started with Sebastian Rose’s [WORG entry](#) *Publishing Org-Mode Files to HTML* at [orgmode.org](#) [22] and a comparison to Dennis Ogbe’s approach discussed in the [blog entry](#) *Blogging using exclusively org-mode* at [ogbe.net](#) [16]. A lot of additional insight is from the doc strings of the publishing functions, which are mainly part of `⊃ ox-publish.el` but also spread all over the individual export libraries, especially `⊃ ox-html.el`. There are

- new hints from *Migrating from Jekyll to org-mode and GitHub Actions* [18] at [duncan.codes](#) and
- reasons to revisit two blogs of Yuan Fu about exclusively using ORG MODE for blogs [6], [5] at [casouri.github.io](#).

The exercise quickly raised a lot of questions about content management and information architecture. These questions were the basis of the hypothesis that ORG MODE has everything to combine into the tasks of a CMS. Collaboration and system management, IMHO, should be outsourced to special software like TRELLO and GIT.

For me even the “[simple](#)” project in the ORG Manual offers too many variations to act as an introductory showcase for ORG publishing. Nevertheless it’s the only point of entry, so be prepared to get confused. As an anti confusion agent I offer a sketch of my “proceedings.”

- Section 2 is about what I expected from ORG publish and how I constructed some rookie Web pages

The next bulk of sections explains and expands the corresponding sections in the carpenter’s Web site assimilation [20], while Section 9 is a commented link to the untouched section of the short version at [pjs64.wordpress.com](#). In contrast to [20] the loop mode for Sections 3 to 7 is extended to and includes the deployment. And the order of the items is not an issue anymore; it’s just kept for reference to the condensed WORDPRESS preview.

- Section 3 introduces to a coarse understanding of the ORG publish concept.
- Section 4 shows how I adapted my rookie expectations by choosing another example set.

- Section 5 introduces an experimental set of ORG publish results in order to categorize the publishing configurations.
- Section 6 extracts the “lowest common denominator” for Web *site* design derived from the amazing number of four Web *pages*.
- Section 7 brings up some general and HTML ORG export options and their defaults.
- Section 8 shows the potential of ORG MODE to offer automatic Web site construction demonstrated by CSS organization.
- Section 9 contains some comments on [20]’s [gluing section](#).
- Section 10 relates a teaser for using NETLIFY deployments to my ideas of a publication model.

Then I’m returning to my test-1-2 rookie Web site and apply the new skills of a static template constructor to generate a differently structured Web site. It was built to get the article you’re looking at into a the next step on the way to a proper Web site and provide another workflow framework.

- Section 11 rearranges the steps in [20]’s [gluing section](#) to put some of the *What’s more* topics of [20] into action: CMS basics and a bibliography workflow.
- Section 12 expands the *What’s more* topics of [20] – ELISP, document information, structure, templates – and concludes with similarities to the Amaya project.
- The Appendix, Section 13, analyses [Ogbe’s Website construction](#).

2 Test, One, Two

I produced my first *basic* set of test files from the attempt to develop plain ORG HTML exports which look very sober; see Figure 1.

The contents are from the [capture chapter](#) of the ORG MODE Manual. The pages show some structural elements of Web pages: content, header, navigation, topics. Obviously they don’t look like a WORDPRESS blog. They lack Web site structure, layout and real design: a site header, navigation slots, some additional information in optional right or left columns or boxes, a footer.

Usually that’s one of the tasks of a well developed content management system. My hypothesis from the Intro: ORG MODE offers every

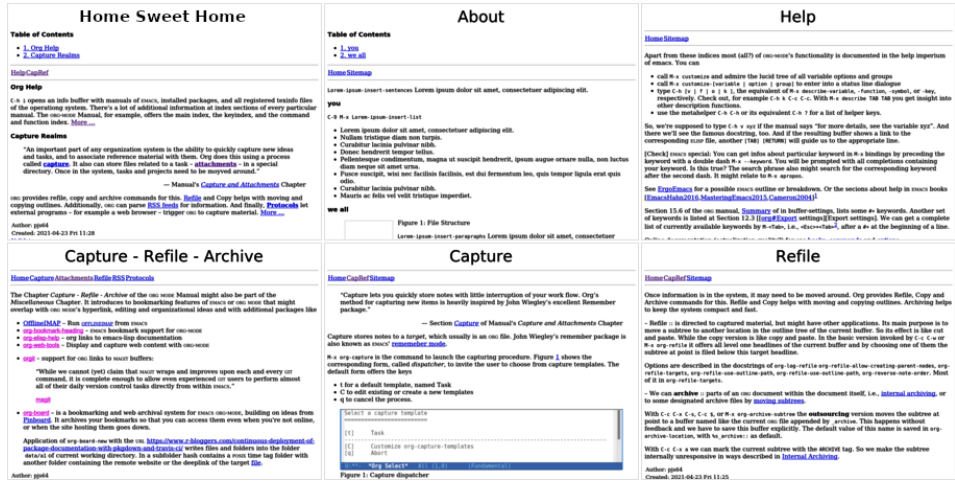


Fig. 1 – Some ORG HTML exports, decorated with links between horizontal lines or a table of contents in order to look roughly like a Website.

feature to act as a CMS. And I thought if ORG’s publishing procedure should do anything useful it should at least be able to convert files into a complete Web site; see the commentary in [ox-publish.e1](#), cited below in Section 3. But Web sites reflect a site structure not a page collection.

By employing some of ORG’s publishing and export features I began to differentiate expectation from coded methods. A coarse introduction to ORG publish should set the scene for the first building brick.

3 Org Publish

The commentary of [ox-publish.e1](#) tells us that “this program allows configurable publishing of related sets of ORG MODE files as a complete website.” It can

- publish all one’s ORG files to a given export back-end
- upload HTML, images, attachments and other files to a web server
- exclude selected private pages from publishing
- publish a clickable sitemap of pages
- manage local timestamps for publishing only changed files
- accept plugin functions to extend the range of publishable content

The statement “publishing is configured almost entirely through setting the value of one variable, called `org-publish-project-alist`” leaves the user with the uncertainty of “almost.” First of all *publishing* means ORG MODE publishing. And the corresponding *configuration* implies an `alist` of this restricted publishing model, only. But it handles dependencies to the whole ORG MODE empire. The `org-publish-project-alist` is not a value, it’s an association list, and it can be a pretty large one; see Ogbe’s blog [entry](#) cited in the Intro. The manual Section *Project Alist* offers two syntactic patterns: the pattern of the lines one and two, and the pattern of line three:

```
("postings" :property1 value1 :property2 value2)
("static-files" :property3 value3 :property4 value4)
("web-site" :components ("postings" "static-files"))
```

The third line defines `web-site` as a set of the sub-projects `postings` and `static-files`. The sub-projects are called components; they combine files requiring the same publishing procedures. When we publish such a *meta-project*, all the components are also published, in the given order. One of the publishing commands for the code above is

```
M-x org-publish-project RET web-site RET
```

A project consisting only of the first two lines’ syntax is shown in the Manual’s Section *Example: simple publishing configuration*. This example publishes a set of ORG files to the \exists `~/public_html` directory on the local machine. But it uses a pain-in-the-ass attribute: *simple*. There’s nothing *simple* about this example, it’s just restricted to the first syntax model and it’s kind of short. Even in the *simple* project we have to be aware of *all* export and publish switches. IMHO that’s *very* far from *simple*. The *simple* example is

```
(setq org-publish-project-alist
  '("org"
    :base-directory "~/org/"
    :publishing-function org-html-publish-to-html
    :publishing-directory "~/public_html"
    :section-numbers nil
    :with-toc nil
    :html-head "<link rel=\"stylesheet\"
              href=\"../other/mystyle.css\"")
```

```
type="text/css"/>"))))
```

The mandatory properties are

- `:base-directory` – the folder containing publishing sources. All files meeting the criteria defined by the properties described in the Manual’s Section *Selecting Files* are *published* according to Section *Publishing Action*.
- `:publishing-directory` – target folder or **TRAMP syntax** for published files.

Another necessary property is the `:publishing-function`, but it has a default, so it might not be called mandatory³

- `:publishing-function` is, by default, set to `org-html-publish-to-html`

For the other three properties I offer a very short insight into configurations of the default HTML export:

- `:section-numbers nil` – switches off the numbers for the headings
- `:with-toc nil` – switches off the table of contents, which would be put beneath the title; title is the first entry of the export `<body>` if `:with-title` is switched on. Without title the table of contents is first.
- `:html-head` – the usage of `:html-head` is restricted to a *part* of the export `<head>`. The whole `<head>` consists of a date comment, one constructor of `<meta>` elements, another big one for head elements like `<style>`, and of MATHJAX info. The *big one* is addressed by `:html-head`; see the `org-html-template` in `ox-html.el` for the whole composition, sketched in Section 5.2.

4 Example Selection

That was too much information. But at least I can figure out that for my purpose the HTML export collection about ORG MODE capture, Figure 1, will miss the point. With these exports I don’t

³Another mandatory one is `:base-extension` which defaults to ORG files; see *Selecting Files* in the Manual. Expansion on this issue would lead directly to a wide field of selective opportunities in the ORG file construction or access to file parts.

get a clue about Web *sites*. In the *Example Selection* of the short version at pjs64.wordpress.com I begin with the German **HTML Intro** (→ [google Translation](#)), accompanied by a [showcase](#) (→ [gooTrl](#)), at [wiki.selfhtml.org](#).⁴ The Section 12.2.3 discusses the statement about the [wiki.selfhtml.org](#) supplements being declared as public domain, CC0.



Fig. 2 – From left to right: \boxtimes [index.html](#), \boxtimes [products.html](#), and a cut version of \boxtimes [contact.html](#). Note the smaller font size of *Impressum* headline in the right page.

All the additional layout material at the **WIKI entry** (→ [gT](#)) *CSS - fertige Layouts* differs in design, only; though the name “CSS complete layouts”, not “complete Web site architectures.” So for the *structural* approach I envisioned in Section 2 I won’t get a satisfying solution, but it turned out to be the entry for my CMS hypothesis. The original file structure of the **HTML Intro** (→ [google Translation](#)), is

- \boxtimes ./ : [index.html](#) • [inhalt.html](#) • [kontakt.html](#) • [preise.html](#)
- \boxtimes css/ : [formate.css](#) • [hobel.svg](#)
- \boxtimes img/ : some SVG and JPG images
- \boxtimes video/ : [carpentry.mp4](#)

Now I take a step back and revisit the knowledge I earned from JOOMLA about template design [23] and my TYPO3 research about the

⁴My motivation for turning to selfhtml is to reconstruct the information architecture of the deprecated web technology *site* initiated by Stephan Münz; see the [mirror](#) of version 8.0 at [www2.informatik.hu-berlin.de](#) for example or look for the latest version 8.1.2.

TYPOSCRIPT language dedicated to template scripting and its current template focus called **fluid template**. Combined with a three month seminar about management systems this previous knowledge puts me in a position to dissect the test site without destroying the workflow of the scientific workplace⁵ I planned to set up for Web technology experiments beginning with ...

4.1 Pages

In my working environment the transferred and renamed original HTML files get the sub-folder \ni `shtm00`, i.e., 00 for nil-original, in my publishing root \ni `~/www`. It contains symbolic \ni `css/` and \ni `img/` folders and the four files

- \ni `shtm00/:` `index.html` • `products.html` • `contact.html` • `pricing.html`

Then I PANDOC these HTML files to ORG, e.g.,

```
pandoc -f html -t org -o index.org index.html
```

and copy the ORG files into a sub-folder of my ORG file root

- \ni `myOrgRoot/pub/shtm0/` : `index.org` • `products.org` • `contact.org` • `pricing.org`

I prepare another sub-folder \ni `shtm0/` in my publishing root \ni `~/www`. Here I add real \ni `css/` and \ni `img/` folders which are supposed to be filled with

Images I want to have all my pictures at one location only, my picture root; this is something like \ni `~/Pictures`. I'm referring to it with symbolic links. So I can access it from LATEX files in custom folders, from ORG folders, from HTML files, whatever. I want to keep this structure of the picture root sub-folders for the uploaded online files, too. For the example I create the sub-folder \ni `shtm0/`. In the case of the small carpentry web site example I

⁵As a matter of fact, part of the motivation for scientific publication stems from mimicking an open source **Scientific WorkPlace**, beginning with the LATEX combination, abandoning the RSTUDIO branded MARKDOWN temptation, and finally switching to the all inclusive ORG MODE offer connected to GNU EMACS and GNU LINUX.

departed from this principle and put *all* files into `Ⓛ shtm0/`, no subsubfolders.

After exporting the ORG page main content and embedding the resulting HTML files into the template the media will be collected into the `Ⓛ ./img/` directory of the web site. An XML crawler will have looked up the all the images which are actually used in the HTML files of the `Ⓛ ~/www/shtm0/` folder.

CSS The final CSS file from the procedure below, in Section 8, will be *tangled* into `Ⓛ ~/www/shtm0/css/` and end up in the `Ⓛ ./css/` folder of the Web site.

5 Publishing Experiments

The goal of this section is to investigate the publishing process and to identify cornerstones of possible ORG publish configurations. Well, and to get ideas about constructing the real Web sites to come. And then come up with the most basic configuration, which some people tend to call *simple*.



This is kind of a fake order. I don't `org-publish` the PANDOC'ed ORG files from above but the files I already modified to get the right input for the templates, developed in Section *Website Template*.

5.1 Producing the Laboratory Files

The experimental setup of the elisp code for discovering `org-publish-project-alist` begins with

```
(setq org-publish-project-alist
      '(("shtm0"
         :base-directory "~/myOrgRoot/pub/shtm0/"
         :publishing-directory "~/www/shtm0/"
         :publishing-function org-html-publish-to-html )))
```

For the short version of *org2shtm* I kept this code in the file `⊃ shtm0.e1` for maintenance.⁶ Then I produced a couple of publication scenarios described in the listing below and picked the `⊃ index.html` files for comparison.

When evaluated the code creates a publishing project called `shtm0` which can be invoked by `M-x org-publish-project [RET] shtm0` or `M-x org-publish [RET] shtm0`. The `:publishing` function exports all `ORG` files in the `:base-directory` to `HTML` files in the `:publishing-directory`. The first four files are the result of producing two `doc-types` `HTML5` (`H5`) and `xHTML` (which is the default, so I call it `Hd`) with and without `<head>`. The `:body-only` files are marked with a trailing `y`.

- After the first invocation of the publishing process I rename the `⊃ index.html` export to `⊃ indexHd.html`
- `⊃ indexH5.html` is the renamed result of adding

```
:html-doctype "html5"
:html-container "section"
```

- the headless `HTML` of `⊃ indexH5y.html` is made with an additional

```
:body-only t
```

- `⊃ indexHdy.html` is the same without the `:html-doctype` and the `:html-container`.

The `ORG` Manual for versions above 9.2 reports about seven switches to get a *bare* `HTML`, i.e., a minimal `HTML` file, with no `CSS`, no `JAVASCRIPT`, no preamble or postamble, but with a complete `doctype-html-head-body` structure; see Section *Exporting to minimal HTML* in the `ORG` Manual.

⁶It's available in the `BITBUCKET` repository's folder `⊃ supp`. In the long version it pretty soon moved into the `-` initially `-` `CSS` producing `ORG` file, which currently might be called the `CMS` `ORG` file.

```
(setq org-html-head ""
      org-html-head-extra ""
      org-html-head-include-default-style nil
      org-html-head-include-scripts nil
      org-html-preamble nil
      org-html-postamble nil
      org-html-use-infojs nil)
```

The corresponding `org-publish-project-alist` notation of the publishing properties looks similar;

```
:html-head ""
:html-head-extra ""
:html-head-include-default-style nil
:html-head-include-scripts nil
:html-preamble nil
:html-postamble nil
:html-use-infojs nil
```

According to this info I add two *bare* index file versions marked by a trailing r: \ni `indexHdr.html` and \ni `indexH5r.html`, respectively.

Table 1 summarizes the set of the six experimental files \ni `indexH5.html`, \ni `indexH5r.html`, \ni `indexH5y.html`, \ni `indexHd.html`, \ni `indexHdr.html`, and \ni `indexHdy.html`.

Table 1 – Set of base name trailers for experimental ORG HTML exports.

format	with-head	bare	body-only
xhtml	Hd	Hdr	Hdy
html5	H5	H5r	H5y

5.2 Analyze It

In the first run the most interesting results were the `:body-only` versions because I considered them to deliver the main content.

Comparing \ni `indexH5y.html` and \ni `indexH5.html` I found that `:body-only t` removes the

- `<!DOCTYPE html>` element,

- environment of `<html lang="de">`,
- `<head>` and
- `<h1>` brace for the title and its surrounding `<div id="content">` brace *in* the `<body>`
- preamble and the postamble.

Comparing `indexHd.html` and `indexH5.html` I observed that `<h1>` was embraced in a `<header>` element for the `html5` version. The abbreviated version of `indexH5.html` shows these elements.

```
<!DOCTYPE html>
<html lang="de">
  <head> ... </head>
  <body>
    <div id="content">
      <header><h1 class="title"> ... </h1></header>
      <!-- left out content beginning with a section element -->
    </div>
  </body>
</html>
```

For *all* building blocks here's a coarse summary of the `org-html-template` from `ox-html.e1`. The comments contain references to corresponding variables or publishing properties.

```
<!-- xml- or php-declaration -->
<!DOCTYPE html> <!-- choose from org-html-doctype-alist -->
<html lang="de"> <!-- build <html> element language info -->
  <head> ... </head> <!-- date-comment, meta, head, mathjax info -->
  <body>
    <!-- :html-link-up :html-link-home :html-home/up-format -->
    <!-- org-html--build-preamble -->
    <div id="content"> <!-- read =org-html-divs= -->
      <header> <!-- a html5 feature -->
      <h1 class="title"> ... </h1> <!-- depends on :with-title -->
      </header> <!-- still a html5 feature -->
      <!-- left out content beginning with a section element -->
    </div>
    <!-- org-html--build-postamble -->
    <!-- insert html-klipsify-src -->
  </body>
```



```
</html>
```

The title which is injected by the `org-html--build-meta-info` into the `<head>` brace doesn't depend on the `:with-title` switch. By the way, another difference of the `:body-only` export is that the `id` attributes of the additional outline container `<div>`'s have different hash tags. For example, the added outline `<div>` of a `<h3>` header looks like

```
<div id="outline-container-orgab9da8b" class="outline-3">
<h3 id="unsere-leistungen">Unsere Leistungen:</h3>
<div class="outline-text-3" id="text-unsere-leistungen">
```

In this code block I call the sequence `ab9da8b` the *hash tag* of the `<div>`'s `id="outline-container-orgab9da8b"` attribute. This side effect might come in handy for customized CSS without analyzing the whole export procedure for the HTML body. Especially the fact that the hash tags get un-hashed by defining a `custom_id` property for the corresponding section; see the ORG Manual's [Section Properties and Columns](#).

5.3 Resulting Configuration

The most intriguing result of the experiments is that `:body-only` in an HTML export is not only a `<body>` issue. But I'm fairly happy with my minimal set of publishing options which are shown in the code below, which I put into `⊃ shtm0.e1` available in the bitbucket repository, and revisited in [Section 7](#).

```
(setq org-publish-project-alist
  '(("shtm0"
    :base-directory "~/myOrgRoot/pub/shtm0/"
    :publishing-directory "~/www/shtm0/"
    :publishing-function org-html-publish-to-html
    :body-only t
    :html-doctype "html5"
    :html-container "section" )))
```



The detailed description of the experimental setup in this section is for further investigations of all switches which

affect the `<body>`. For all exports, general or HTML specific. In the short version I only need to notice that the HTML5 decision is a source of wide spread effects. E.g., both the preamble and postamble switches produce a `<div>` element in the `<body>`, but not in the `:body-only` version. For inspiration I will get into (1) `:options-alist` of `org-export-define-backend` in `⊃ ox-html.el` and (2) `org-export-options-alist` in `⊃ ox.el`.

6 Website Template

Back to the transferred and renamed semi-original HTML files in `⊃ ~/sthtm00`. They are the source for identifying locations in the HTML code where I might place the main content, the navigational elements, the footer, or a menu.

The pages of the example Web site are very similar, so the header, the navigation and the footer might be considered static. And I won't make a big mistake if I decide to inject only the main content. Figure 2 shows three pages of the exercise layout `showcase` (\rightarrow `gooTrl`) at the German `wiki.selfhtml.org`. In the caption the file names are my customizations of the structure I declared above, but they are linked to the original files.

“Considered to be static” means that the `<header>` *is* really static: it consists of an icon, the company name, and a rotated commercial message. The navigation is a plain⁷ list of links. The footer of all but the contact pages contains links to the contact page and its imprint section.

Another minor irregularity regards the first line of what I will use as the main content. All main contents begin with an `<h1>` element. But in the contact page the `<h1>` element is embedded in an `<article>` element and its font size is smaller. To figure out the reason for this behavior I sketched the coarse structure of the main content in the

⁷The `` markup for the current page deviates from “plain”. It is discussed in the (German) `quick-start manual` referring to a blog `entry` [4] at `html5doctor.com`.

<body> of all pages – neglecting the <header>, <nav>, and <footer> elements; see Table 2.

Table 2 – Sketch of the web site page elements. Every bullet introduces a lower element level.

index	contact	products	pricing
<h1>	<article>	<h1>	<h1>
<p>	• <h1>	<p>	<table>
<section>	• <p>	<p>	
• <h2>	• 2 <dl>'s	• 7 's	
• 	• <h3>		
• <aside>	• 9 <p>'s		
• • <h3>	<aside>		
• • <p>	• <h2>		
<section>	• <dl>		
• <h2>			
• 2 <p>'s			

The contact appears kind of rough. <h1> and <h3> are on the same level. The <aside> element isn't used. The font-size for the <h1> element isn't specified explicitly in the corresponding CSS file `formate.css`. Degrading to a lower level by being embedded in another element seems to cause an implicit font-size loss. The [Design 01](#) (→ [gT](#)) layout at [selfhtml.org](#) offers another version of this contacts page.

Well, to be complete, I also considered the page *title* to be static; this is the title that shows up in the browser's tab register. In the layout original the titles of the pages are different. Browser tab titles are defined in the HTML <head>.

For the home page I want to produce the HTML code below with a headerless HTML export from the file `index.org` in the publication folder `pub/` of my ORG file root

```
<h1>Willkommen ...</h1>
<p>Wir sind seit ....</p>
<section id="service">
<h2>Unsere Leistungen:</h2>
...
</section>
```

The template for every page of the example web site is the rest derived from the index file with the considerations above.

```
<!doctype html>
<html lang="de">
  <head> ... </head>
  <body>
    <header> ... </header>
    <nav> ... </nav>
    <!-- here's where the main content should be injected -->
    <footer> ... </footer>
  </body>
</html>
```

I declare this to be the template; I'll add it as `⊃ tmp1t.html` to the BITBUCKET supplements and put the two parts above, below, and without the main-content-comment as `⊃ tmp1t1.html` and `⊃ tmp1t2.html` into the publishing folder `⊃ pub/shtm0` of `⊃ ~/myOrgRoot/`. Before proceeding to the CSS I'll have to know how the export fills in.

7 Org HTML Export

Exporting an ORG file to HTML offers a lot of features, which are reflected in a lot of publishing switches for HTML. And the publishing empire in turn produces many additional opportunities for help, inspiration, and distraction. First I'll investigate on the file based options for the header of my `⊃ index.org` file which is supposed to substitute the main content placeholder in the template.

The lines beginning with `#+`⁸ in an ORG file escape the text mode to some kind of control mode which translates to `in-buffer-settings` or `export settings`, for example. Typing `#+` in a new line and asking for completion with `M-[TAB]`⁹ shows all available setting keywords.

⁸I began to call these escapes *directives* for myself to separate them from `#+begin` container constructs and to see them as parallel to C's preprocessor directives; see the WIKIPEDIA entry [directive \(programming\)](#). But I'm not sure if directive is a good choice.

⁹Many desktops intercept `M-[TAB]` to switch windows. Use `C-M-i` or `[ESC][TAB]` instead.

The following `#+` settings in the ORG file header are for the test suite. Later some of them are transferred into the project's `org-publish-project-alist` entry. But that depends on the nature of each option. For example, `#+Language: de` is related to the individual file, so it should stay in the file. While `#+Options: num:nil toc:nil` depends on the web site architecture, so they might end up in the publishing `alist` as `:section-numbers nil` or `:with-toc nil`; see the end of Section 3.

The ORG code of the in-buffer `#+HTML_CONTAINER: section` property transfers to the ELISP code publishing property `:html-container "section"`. We can derive this connection by looking up the in-buffer property at the Manual's [Section *HTML Specific export settings*](#) and inspect the doc-string of the corresponding `M-x h org-html-container-element`. Or look up the whole collection of property relations in the backend definition of \exists `ox-html.e1`, i.e., the `:options-alist` of `org-export-define-backend`.

Apart from entries like `#+Title:`, `#+Subtitle:`, `#+Author:`, or `#+Email:` the settings for the carpenter's site are

```
#+Language: de
#+Options: num:nil toc:nil
#+HTML_CONTAINER: section
#+HTML_DOCTYPE: html5
```

I assume the `#+Language:` to affect the body, so I'll keep it. I'll set the `num` and `toc` attributes of `#+Options:` to abandon header numbering and the table of contents, respectively; see the general [Export Settings](#). The `#+HTML_DOCTYPE` setting seems useless for the body-only export. While the corresponding element isn't inserted, it affects the usage of HTML5 elements in the body. The `#+HTML_CONTAINER:` determines the next level beyond the `<body>` element; this level adds an element bracket for the whole content; see [HTML Specific Export Settings](#) in the Manual.



The Manual's [Section *HTML preamble and postamble*](#) can be another source of inspiration for producing the main content. See the docstring of `C-h o`

org-html-postamble-format. Unfortunately they are skipped for body-only exports.

7.1 Aside

The `#+HTML_DOCTYPE:` (see the [Manual](#)) triggers different conversions of special elements like `<aside>` which is used in the index page,¹⁰ see the *Angebot* panel in the left part of Figure 2 and the `<aside>` placement in Table 2. The ORG code to produce this element is

```
#+Attr_html: :id offer
#+begin_aside
#+Html: <h3>Angebot</h3>
Nächste Woche 10% auf alles!
#+end_aside
```

The first line of the block below shows the enclosing elements for regular conversion, the second line is the `html5` version

```
<div id="offer" class="aside"> .. </div>
<aside id="offer"> .. </aside>
```

The `#+HTML_CONTAINER: section` (see the Manual’s Section *HTML doctypes*) has two relevant effects. The usage in the ORG header puts the whole document body into a `<section>` brace. Here the difference of the `xHTML` default and `HTML5` is

```
<div id="outline-container-org7bdb0cc" class="outline-2">
<section id="outline-container-org79ac04b" class="outline-2">
```

Another possible usage is related to the last paragraph of the Manual’s Section *HTML doctypes*: “Special blocks cannot have headlines. For the HTML exporter to wrap the headline and its contents in `<section>` or `<article>` tags, set the `HTML_CONTAINER` property for the headline.” The offer `<aside>` could be ORG coded like

```
*** Angebot
:PROPERTIES:
:HTML_CONTAINER: aside
```

¹⁰The `<aside>` is also used in the contacts page, but it’s shown like a section. Fixing this might be a proper exercise after finishing this blog entry.

```
:CUSTOM_ID: offer
:END:
Nächste Woche 10% auf alles!
```

And it would be exported as the HTML snippet below; another advantage: for the LATEX, and any other export it would produce a regular subsection, not an `aside` environment which I would have to define otherwise. But it gets more difficult to address the CSS background property for this `<aside>` element.

```
<aside id="outline-container-org70aac58" class="outline-4">
  <h4 id="offer">Angebot</h4>
  <div class="outline-text-4" id="text-offer">
    <p>Nächste Woche 10% auf alles!</p>
  </div>
</aside>
```

7.2 Headline Levels

In the preceding sections I produced an `index.org` file for the main content of my home page. Now I can compare the differences between export and intended main content; I'll call them *ex-main* and *in-main*.

- the whole *ex-main* is enclosed in

```
<section id="outline-container-org60ba67b"
  class="outline-2"> .. </section>
```

- the heading tags include the `id` attribute of the PROPERTIES drawer's `CUSTOM_ID:` field. I didn't include the drawers myself; they were planted there by PANDOC'ing the example HTML files to ORG markup.

```
<h3 id="unsere-geschichte">Unsere Geschichte:</h3>
```

This also encourages the usage of custom ID's for proper linkage.

- every header is followed by another `<div>` enclosing the text of the section

```
<div class="outline-text-3" id="text-unsere-geschichte"> ..
↳</div>
```

- and all the ex-main heading levels are increased by one level, e.g., `<h1>` in in-main relates to `<h2>` in ex-main.

Headline Level Increase

The reason for the level-increasing effect led to the insight stated in Section 5.2. Using the “body-only” switch C-b in ORG’s HTML export not only leaves out the whole `<head>` element, it also refuses to put the `<header>` element into the `<body>`. Well, “body-only” is definitely shorter than “no head and header and som other effects” and the export dispatcher is used for other exports, too. So, this seems to be the ambiguity we have to deal with for HTML export. Probably I didn’t find the export switch for the `<header>` element, yet. For a “head-and-body” export the `<header>` element in the `<body>` would look like

```
<header>
<h1 class="title">Schreinerei Meier</h1>
</header>
```

That’s where the `<h1>` is lost. On the other hand the `<header>` part of the intended example homepage is already sourced out to the template:

```
<header>
  <a id="backlink" href="index.html"></a>
  <p>Schreinerei Meier</p>
  <p>ihre Werkstatt für gutes Wohnen!</p>
</header>
```

It doesn’t use a heading element, and the style file `formate.css` employs some very recent CSS magic to address the first and the last paragraph: `p:first-of-type` and `p:last-of-type`. This insight plus the field trip above about “sectionizing” the `<aside>` element might deliver ideas for structural ORG files

aimed at web site architecture. See also the doc-string of `C-h o org-html-toplevel-hlevel`.

Now I can think of three procedures to connect the main content to the template

- use the HTML extract and change the CSS
- change the HTML extract with the CSS untouched
- change both to support ORG export and publishing features.

For this to decide I'll introduce the ORG MODE CSS procedure borrowed from Fabrice Niessen in the next section.

8 Org CSS Construction

This section is about tangling designated parts of an ORG file into a style file. The short version in *Org CSS Construction* at pjs64.wordpress.com skips the details about Niessen's idea and the implications I derived from it.

Apart from the headlines the only descriptive text, Niessen puts in his CSS/JS construction file `⊕ readtheorg.org` is

“Get the lowdown on the key pieces of ReadTheOrg's infrastructure, including our approach to better HTML export. The setup file links to the web pages.”

This is a rather unobtrusive way of advertising a groundbreaking method of CSS organization. With `⊕ readtheorg.org` Niessen covers the construction of two CSS and one JAVASCRIPT file. IMHO the JAVASCRIPT parts can be covered by CSS3

Brainstorm

The HTML-*export* of Niessen's `⊕ readtheorg.org` doesn't explain much either, unless the fact that it is *possible* to export it. The potential that I see in this CSS – and JAVASCRIPT and template and Web site – constructing ORG file is not discussed in the repository. Probably I missed something. In my opinion the multipurpose constructor file delivers for Web sites what reproducible research is

supposed to do for science.

- It can be employed to act as documentation and showcase for the methods and definitions included.
- It can contain any information or source code blocks which can be used for manipulation or description of the tangled code or for producing additional parts of a proper documentation.
- It can be employed to map information architecture. A thorough combination of the ORG features of publish, export, noweb, tangle, babel, capture, hyperlinks, agenda, timestamps, or tables might deliver a sequential one-level solution.

The approach combined with ORG source code blocks is aimed at replacing the SASS and LESS precompilers for CSS combined with JavaScript magic. Or a DIY alternative or supplement to the [Jam-Stack](#) approach, recommended by NETLIFY.

In the exercise example of this blog the CSS construction is focused on ORG's tangling concept, only. Only? The implementation of the tangling concept also offers diversified commenting switches, which invite the user to a wide range of applications; see the `:comment` and the `:padline` header arguments in the Manual's [Section *Extracting Source Code*](#). The same section holds other goodies like `:link`, `:mkdirp`, `:shebang`, or `:tangle-mode`.

My utilization of ORG tangling begins with cutting the example's CSS file `format.css` into pieces, putting the pieces into CSS source code blocks and embedding them in an outline structure of an ORG file. The source code block has to employ a tangling mark, i.e., a header argument, like `:tangle yes` or `:tangle filename`. There are four levels of customizing the tangling target or any header entry

- put `:tangle path/2/style.css` in the header of the source code block; see the [Section *Extracting Source Code*](#) in the ORG Manual.
- put it into a heading's property drawer like `:header-args:css: :tangle path/2/style.css`; see the [Section *Using Header Arguments*](#).
- put `#+PROPERTY: header-args:css :tangle`

path/2/style.css anywhere into the document, probably in the file header

- setting `org-global-properties`; see the *Property Syntax Section*.

For the short carpenter’s site example it’s not necessary to use either of them, because for `:tangle yes` every CSS source block is filled into a file with the base name of the ORG file which contains the code blocks.

Nonetheless I’ll include an `:tangle ~/www/shtm0/css/shtm0.css` in the property drawers of the sections which contain CSS code for the `shtm0` example. In the example these are all code blocks, but the next step for producing different files is close at hand. I don’t know if this is a good idea; the next projects will show.



The header argument `:output-dir` doesn’t work for tangling, but for source code block output, which is addressed by the result *type* `:file`; see *Section Results of Evaluation* which also introduces `:file-desc`, `:file-mode`, `:file-ext`, or the result *format* `html` with `:wrap` fine tuning. The *Section Environment of a Code Block* offers a paragraph about `:dir` and `:mkdirp` corresponding to the working directory. The examples in these condensed manual sections might also serve a source of inspiration for publication-related aspects of ORG source code blocks. Anyway I’ll have to make sure that the `css/` folder exists; see *Section 11.3.1* for the corresponding routines applied in the *Test, one, two, three* case.

The adaption of the CSS to the HTML export and the design measures are described in the short version’s *Section Org CSS Construction* at `pjs64.wordpress.com`.



For the HTMLIZE¹¹ driven feature of inline CSS, especially its failure in batch mode, see the doc-string of

¹¹External info at the [emacs-htmlize](#) github page, [ELPA entry](#), or the [entry](#) at the EMACS WIKI.

`org-html-htmlize-output-type`. In contrast with other applications of HTMLIZE this reflects the situation for HTML *export* of a buffer.¹² A controlled construction of `htmlize` is facilitated by `org-html-htmlize-generate-css`.

Get more DIY ORG → HTML hints in the doc-strings of `htmlize-buffer`, `htmlize-file`, `htmlize-many-files`, `htmlize-many-files-dired`, `htmlize-region`.

9 Glue HTML and Extract Images

In the carpenter’s Web site assimilation [20] Section *Glue HTML and Extract Images* delivers the central content of the post. In the extended version the procedure is sketched in Section 11; it gets a structural introduction, a preparation for information retrieval and the glue and image part are split up.

[20]’s glue and image section concludes with the collection of media into the publication folder. I maintain an image root folder and determined its sub-directory `⊃ shtml0/` for keeping all the files which are from the ZIP repository of the example and the files I choose to add myself. In the ORG source for the Web pages links to the images were inserted with `file:./img/xxx.yyy` where `⊃ ./img/` is a symlink to my image root. In the HTML export this `⊃ img/` directory is a real folder which is supposed to contain real media.



My image concept usually needs a sub-folder in the `⊃ img/` directory of the upload folder. For this to work the `file.copy()` of the last line would need a directory check; see Section 11.4.2 for the routine. In the short version [20] I don’t use this subfolder structure; I just put all the media into `⊃ img/`.

For the extended 1-2-3 version, see Section 11, I might have been able to combine publishing properties like `:base-extension`,

¹²For LATEX export we have the choice between plain verbatim environments, like verbatim itself or its fancy version `fancyvrb` and code highlighters, like `listings` or the pygment wrapper `minted`.

:exclude, and :include to utilize the publishing function org-publish-attachment. For instance I could have used the method for selecting updated pages to construct a file list for the :include property and use my image root sub-folder as :base-directory; see *Selecting Files* and its preceding section in the ORG Manual. It might be far off in this case but a reasonable solution for other approaches. See Ogbé’s solutions, annotated in Section 13.1 and summarized in Table 5.

10 Netlify Drop

The result of the short version [20] was a Web site which is shown as a screenshot excerpt in Figure 3.

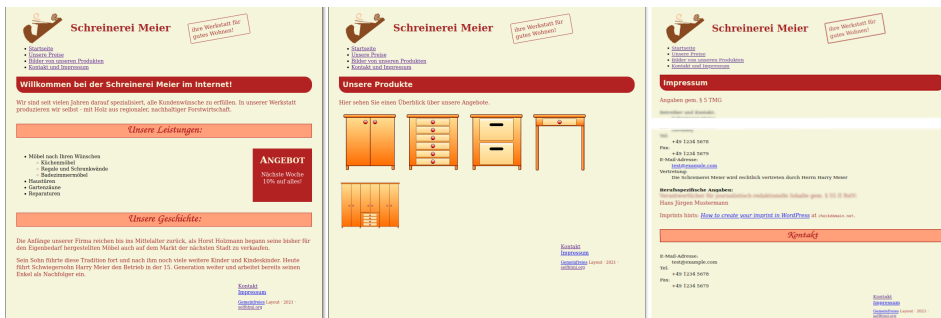


Fig. 3 – Org mode version of `index.html`, `products.html`, and a shrunked version of `contact.html`.

Netlify has a very fine grained billing model. I didn’t offer the URL for the resulting netlify.app as a show case, because – without further investigations – I can’t relate the billing for traffic to the bandwidth concept. For me this is another rookie issue like my ELISP illiteracy.

Netlify offers some insight into their models but the documentation has the same issues as many other usage manuals. It’s about standards not the many implications of a multitude of applications. There’s a dedicated *Billing* Section, and there are many topic related billing parts of every *metered* feature, like *monitoring*, *visitor access*, *forms*, *functions*, and *large media*. But for me even the terminology is so far from my imagination that I had to visit other resources to tackle the issues. The

Billing Section links to the second item below, and I don't remember what led me to the first one.

- *cloudflare + netlify* 2021-02-08 [entry](#) at blog.bytefaction.com. According to the URL it's about outsourcing bandwidth to CLOUDFLARE. But this might be a matter of jumping out of a frying pan into the fire.
- More Info at the support [guide](#) *How to reduce your site's bandwidth usage (without reducing traffic!)* at answers.netlify.com.

I tend to a more elaborate model which avoids the metered [large media](#) feature. For pixelized photos I'd prefer a thumbnail which points to some sources at WIKIPEDIA COMMONS or PIXELFED; the GIT hash of a thumbnail at least offers the necessary updating feedback. For info graphics and plots I'd suggest an SVG version, or even better the graph producing script with raw data access. That's more like reproducible research. For videos it's even more interesting to reduce the traffic or bandwidth. Videos that are the result of screen casting might be turned into a LATEX BEAMER presentation, or a slideshow. Audio for screen casts is usually not based on a script; in the worst case an unedited conglomeration of hmmmms and ehems. Same counts for presentations. So there are many stages in the production cycle to reduce the potential of data volume increments.

11 Test, One, two, three

The fact that `:body-only` is not only a `<body>` issue makes too much publishing features unavailable. On the long term I think I will go with the bare model and immensely use R aided XML editing based on explicit node treatment. Particularly the section *Glue HTML* about embedding the body-only exports into the template will probably move from gluing to node replacement.

But first I want to apply the the short version approach to my initial steps of a clueless preparation of a few ORG MODE stubs – see Section 2. This application is going to include

- [directory split](#) for structural files and postings
- [pdflatex](#) the *org2shtm expanded* or *long* version (this document)
- HTML'izing both the posting and its bibliography. This part

turned out to deliver a pretty tricky construction of symlink and `#+include:`. The problem is reported and solved in Section [11.2.3](#).

- adaption of image links and `collection` into the upload folder

The result will be a volatile publication. This time I'm going to provide the `netlify.app` URL in order to monitor all the un-billed statistics I can get hold of. And when something unusual happens I intend to pull the plug. That's the volatile part.



The whole procedure depends on unshared

- ORG macros in the setup file `config.org` included by `#+SetupFile: ../config.org`,
- custom ORG entities listed in `entity.org` included by an `org-entities-user` variable defined in my `~/ .emacs` file,
- LATEX macros invoked by the ORG macros in `config.org` and in the LATEX template; see Section [11.2.1](#).

The whole construction is a work in progress driven by the motivation to switch from LATEX SWEAVE to ORG MODE procedures. There are so many variables, methods and files involved in this process that I'm just happy to derive the necessary steps manually. A proof of concept. While I don't provide the source files at least I've illustrated the scaffold of my image, LATEX and ORG file roots in [Figure 4](#).

11.1 Expanded Configuration

My choice for this next level publishing is about wiring the hard disk to a publication folder by harnessing symlinks, ORG `#+include:`'s, and ORG `search options`. The ORG files themselves could provide additional selection features like tags; see Section [12.3](#). The symlink ORG files in the `:base-directory` will result in real HTML files at the publishing folder. A symlink `index.html` in the publishing root will point to the starting page file `index.html` in the `sys/` folder. Here's the

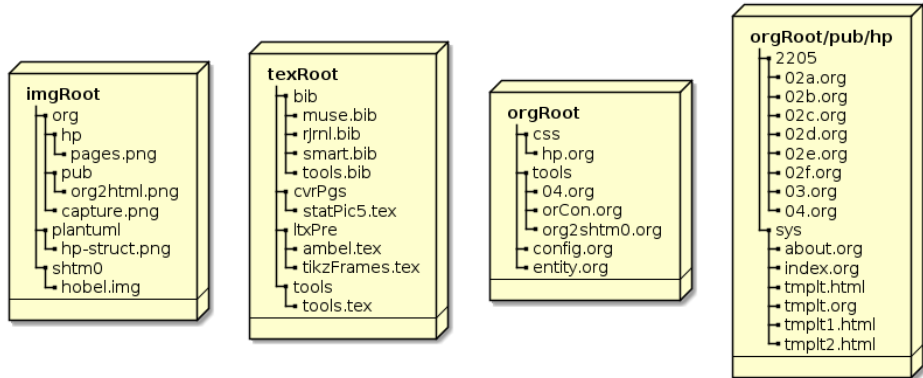


Fig. 4 – Three root folders of the expanded home page structure. Image produced with embedded PLANTUML source code.

:base-directory with its files, the settings within them, and the features they provide; see Figure 4 for the file structure.

- `sys/`, the folder with Web site structure files
 - `index.org` `#+Options:` `num:t` switch on headline numbers. The text has ORG file links which turn into HTML file links.
 - `about.org` has two subsections, an internal Web site link, and an inset picture.
 - `tmplt.org` is supposed to produce the template, but the best I could do by now is to prepare `tmplt0.html` manually and split it to `tmplt1.html` and `tmplt2.html`.
- `2205/`, a monthly folder
 - `02a.org` contains an `#+include:` of `rel/path/2/orCon.org::#cap-ref-arch` with a `:lines "1-47"` option; it uses the headline of the referenced section in `orCon.org` and add the specified number of lines. The line specification cuts off the following subsection structure; this post qualifies for testing the `:minlevel` argument. The file also contains table of links to the sub-02-posts b–f. Note that the sub-02-posts d–f are accessible through this table only, not by the menu.

- `02b.org` and `02c.org` `#+include:` other sections with an `:only-contents t` option. `02b.org` substitutes the missing headline with a horizontal-line concept, while `02c.org` uses a headline and an abstract construction with additional information.
- `02d.org` to `02f` and `03.org` contain a plain `#+include:` with no options
- `04.org` is not included in the repository folder because it's a symlink to `orgRoot/tools/04.org`
 - `img/` a symlink to my `imgRoot/`
 - `css/` a symlink to a CSS folder in my `orgRoot/`

See the ORG Manual's *Include* Section for arguments like `:only-contents`, `:minlevel`, or `:lines`. The settings and features are a result of the procedures in the next Sections 11.2, 11.3, and 11.4. The publishing alist is

```
(setq org-publish-project-alist
  '(("hpSys"
    :base-directory "myOrgRoot/pub/hp/sys"
    :publishing-directory "~/www/hp/sys"
    :exclude "tmpl.org" )
    ("hpBlg"
    :base-directory "myOrgRoot/pub/hp/2205"
    :publishing-directory "~/www/hp/2205" )
    ("hp"
    :components ("hpSys" "hpBlg"))
  ))
```

with `hpSys` and `hpBlg` sharing the properties below. I removed the shared properties from the alist above to reduce the code block size and to emphasize the common settings.

```
:with-toc nil
:section-numbers nil
:publishing-function org-html-publish-to-html
:body-only t
:html-doctype "html5"
:html-container "section"
```

The main difference between the two publishing components `hpSys` and `hpBlg` is their timestamp behavior. As far as I observed in the

real `sys` files the publishing process can track the content, while the `#+include:` part of the files in the `blg` directory prevents from updating checks. And all files but the symlink `04.org` can be published by `org-publish-current-file`.

When I review the file structure I immediately got other ideas for a proper structure. For example in 2022-06 I'll run into the problem of having to define a new component. But right now the task is to define a structure at all, not the structure for every purpose. The most obvious choice would be `04.org/` instead of `2205/`. Or starting from the root recursively, excluding the `sys/` folder. Another idea is to put the CSS file into the `sys/` folder.

With an increasing number of files it's probably a good idea to put up a check list like Table 3. An ORG table can be used to control and document the blog entries. The short column names in Table 3 are designed for direct usage as variables. I could put this table in my `hp.org` file which started with the CSS constructor; now the file kind of naturally grows into the architecture file I was hoping for. First I added the ELISP code for publishing, then the R code for collecting the image files. But the file still sits in the `css/` folder; it probably will move into the `sys/` department which itself might grow into another structure with a corresponding table illustration.

11.2 Bibliography and PDF

In my first attempt I was working on the original `org2shtml.org` file in my `orgRoot/` then, according to Section 11.2.3 switched to the `#+include:` file `04.org` representing the ID of the publishing process. Apart from that detour the whole procedure depends on unshared ORG macros, custom ORG entities, LATEX macros; see the *What's More* Section 12.2.1 for their purpose and Figure 4 for the involved file structure.

In the PDF construction cycle I have to make sure that any `#+BIBLIOGRAPHY:` line is commented. This line is necessary for the HTML bibliography procedure handled by the inclusion of `ox-bibtex.el`; see Section 11.2.2.

Table 3 – Publishing Map. Legend: p .. the subfolder of \exists pub/hp/, f .. ORG file name base, t .. title, i .. flag for inclusion or symlink, s .. source, o .. include options and other text.

p	f	t	i	s	o
2205	02a	ORG Capture	i	orCon.org::<#cap- ref-arch	✓
2205	02b	Attachments	i	orCon.org::<#attachments	
2205	02c	Capture	i	orCon.org::<#capture- 2	✓
2205	02d	Protocols	i	orCon.org::<#protocols	
2205	02e	Refile	i	orCon.org::<#refile	
2205	02f	RSS Feeds	i	orCon.org::<#rss- feeds	
2205	03	ORG Help	i	orCon.org::<#sec- org-help	
2205	04	ORG Web Site	2	s 04.org	

11.2.1 Compiling Latex

I first export the LATEX body from the posting file \exists org2shtm0.org which delivers \exists org2shtm0.tex. Then I insert an inclusion macro \backslash PartIn[2]{ $\}$ into my LATEX template \exists tools.tex.

```
%-----
% Org 2 shtm0, publication
\PartIn{Org to Website}{rel/path/2/org2shtm0}
```

\exists tools.tex contains the LATEX configuration and all my victims for LATEX compilation in a structured ASCII manner.¹³ They are all commented unless they should be compiled.

The \backslash PartIn[2]{ $\}$ command shares its purpose of file inclusion with other constructs like \backslash LineFile[2]{ \dots }, \backslash FileIn[2]{ \dots }, \backslash FileInNN[2]{ \dots }, \backslash starsub[1]{ \dots }, \backslash FileLine[2]{ \dots }, or plain \backslash input{ $\}$. I'll hide the ingenuity of these macros from the curious

¹³This is the blueprint for something I might implement as an ORG MODE agenda, but I first turned all the material I'm working on into a project file. I just can't find a way to put my thoughts into a get-things-done pattern. Talking of ill-structured? Here we go.

reader. The top level scientific definition of `\PartIn[2]{}` is

```
\newcommand\PartIn[2]{\newpage\part{#1} \input{#2}}
```

I'll use two results of this procedure (1) the PDF is ready for upload (2) the AUX is the source for HTML creation.

11.2.2 BibTeX HTML

With the help of Filliâtre and Marché's `BIBTEX2HTML` tools I manually extract the BIBTEX entries to `org2shtm0.bib`, clean up the entry fields, and copy both the PDF and the BIB file to the `org2shtm0.org` folder. `BIBTEX2HTML` is also used by `ox-bibtex.el`; it offers the auxiliary programs `aux2bib` for extracting bibliography entries and `bib2bib` for manipulating the extract. The main program `bibtex2html` HTMLizes the BIBTEX source file and the `<table>` snippet.

```
cd my/docs/tex
cp ltxTplt.pdf rel/path/2/org2shtm0.pdf
aux2bib ltxTplt.aux > org2shtm0.bib
bib2bib --remove owner --remove timestamp --remove abstract
↳--remove journal-url --remove language --remove keywords
↳--no-comment -ob org2shtm0.bib org2shtm0.bib
cp org2shtm0.bib rel/path/2
```

Removing the abstract field might be a bad idea when you plan to use `BIBTEX2HTML`'s `feature` of printing commented bibliographies: “If a BIBTEX entry contains a field `abstract` then its contents are quoted right after the bibliography entry in a smaller font, like `this`.”

I may consider removing the comments in the BIB file manually. Furthermore I may like to change the sorting of the BIB file. That's a matter of `bib2bib` configuration; the flags `-s 'author'` `-s '$date'` for example sort by author, then by date.



Hint from the `bib2bib` man page: “When sorting, the resulting bibliography will always contain the comments first, then the preambles, then the abbreviations, and finally the regular entries. Be warned that such a sort may put cross-references before entries that refer to them, so be cautious.”

Now I test the *export* inside the `:base-directory` the first time; the *first time* reflects the situation with a symlink `04.org` in the `pub/hp/` to `org2shtm0.org` in the `tools/` folder of my `orgRoot`; see Figure 4. I uncomment, activate or insert

```
#+BIBLIOGRAPHY: org2shtm0 plain option:-a limit:t
```

in `org2shtm0.org` and *export* the file to HTML. With installed OX-BIBTEX.EL the existence of `#+BIBLIOGRAPHY:` results in two files `org2shtm0.html` and `org2shtm0_bib.html`. The first is the resulting HTML export with a bibliography table snippet called *citation list* and adapted citation links. The `_bib.html` is a HTMLized version of the collected bibliography entries. The citation link table is inserted at the `#+BIBLIOGRAPHY:` location.



I could have skipped copying `org2shtm0.bib` to the `:base-directory` and instead used `my/docs/tex/org2shtm0` as BIB file path in the `#+BIBLIOGRAPHY:` directive. Same results.



Bug? Citations like `\cite{fu2018bom,fu2018bor}` are put correctly in the text and in the citation table, i.e. the bibliography appendix, but the entry in the BIB file isn't transferred into the HTML version of the BIB file. For correct transfer they have to be separated into `\cite{fu2018bom}` and `\cite{fu2018bor}` by more than a space; a comma works. Not tested: or added with separate `\nocite{}` commands? I didn't look for the cause of this bug. [BIBTEX2HTML](#) or `ox-bibtex.el`?

11.2.3 Symlink Publishing Adaption

Remember that my intended approach was a `04.org` symlink in the `:base-directory` linked to my working file `org2shtm0.html`. And

manually commenting the `#+BIBLIOGRAPHY:` directive. In the previous section I called this configuration *the first time*.

So what happens when I *publish* the file from the symlink `⊃ 04.org`? It results in two files `⊃ org2shtm0.html` and `⊃ org2shtm0_bib.html` in the `:base-directory` and another `⊃ org2shtm0.html` in the `:publishing-directory`. But the base `⊃ org2shtm0.html` is the citation list `<table>` for the reference inclusion. Seems part of the OX-BIBTEX.EL feature to overwrite this snippet with the HTML export of `⊃ org2shtm0.org`. And the publishing process somehow enters into the BIBTEX process before overwriting or a final clean up.

So, I publish the symlink `⊃ 04.org` and get (two) `⊃ org2shtm0.html` files. Nice. But I want to keep the 04 signature. My first ideas for possible responses

1. forget about the date for bibliography blogs, just copy the BIB file in the publishing folder; this also might be part of a publishing project
2. rename the files and the internal references with a cumbersome R XML script
3. rename the original work file in the original folder to 04.org and copy the resulting BIB file like above; might lead to name clashes.

I began testing items 1–3 and then discovered item 4. When I was ready for publishing the first issue of my volatile Web site I debugged the error commented in the hint below which led to item number 5.

4. point the `⊃ 04.org` symlink in the `:base-directory` to a file `⊃ 04.org` in the same folder as `⊃ org2shtm0.org`, i.e., `⊃ orgRoot/tools`. This *real* `⊃ 04.org` `#+include:'s` `⊃ org2shtm0.org` and I can add publishing features, beginning with the `#+BIBLIOGRAPHY:` line.
5. if I use `⊃ org2shtm0.org` with the `#+BIBLIOGRAPHY:` line another way to put the internal references right is to include a bib file named `⊃ 04.bib`. This produces the citations in `⊃ 04.html` and the bibliography excerpt `⊃ 04_bib.html` *and* includes `⊃ 04.html` in the HTML export `⊃ org2shtm0.html`

Awesome. I found a way to overcome the manual commenting and uncommenting of the `#+BIBLIOGRAPHY:` line and made room for other editorial tasks. After publishing the symlink I find `⊃ 04.html` in the `:publishing-directory`, but `⊃ 04_bib.html` is still in the `⊃ orgRoot/`

folder, so I finish with

```
cd ~/orgRoot/Tools
cp 04_bib.html ~/www/hp/2205/
```



Hint: Don't hide the `#+BIBLIOGRAPHY:` line in a commented section. I just made this error by appending a commented appendix section as the last entry. Then my inclusion file, the *real* `04.org` in the `orgRoot/` folder, cut off all the content after the `#+include:` setting.

11.3 Glue HTML

In contrast to *Glue HTML and Extract Images* at `pjs64.wordpress.com` I now separate the HTML embodiment from the image treatment. The section *Paths and Files* is for both the template inclusion and the image comparison. It probably will be placed more properly at the beginning of the whole architecture file `hp.org`.

11.3.1 Paths and Files

Two differences to *Paths and Files* at `pjs64.wordpress.com`.

- The template files moved to `sys/`.
- Directory check for the `css/` folder.

The second item ensures an existing folder for the CSS tangling further down in an integrated ORG MODE file `hp.org`. The header arguments `:header-args:` `:tangle ~/www/hp/css/hp.css` in `hp.org` expect a `css/` folder in `~/www/hp/`.

```
pub <- "~/myOrgRoot/pub/hp";
www <- "~/www/hp"; setwd (www);
pic <- file.path(www, "img"); if(!dir.exists(pic)) dir.create(pic);
sty <- file.path(www, "css"); if(!dir.exists(sty)) dir.create(sty);
t <- c(file.path(pub,"sys","tmpl1.html"),
      file.path(pub,"sys","tmpl2.html"));
```

11.3.2 Select Updated Pages

For development mode I use `⊃ css/` and `⊃ img/` *symlinks*; for production mode I have to make sure to turn them into real folders. The symlink checker `Sys.readlink()` is for the `⊃ index.html` symlink in the root only; it avoids surrounding the `⊃ index.html` file in `⊃ sys/` twice with the template brace. According to the help file this is a POSIX feature not available at WINDOWS.

Looking for "`<!doctype`" should be expanded to find "`<!DOCTYPE`" too. Might be a matter of putting the `scan()` function into a `tolower()` function. The other functions are `list.files()`, `append()`, `length()`, and the `for(){}` control.

```
listHtml <- list.files(path=www, recursive=TRUE,
  pattern=".[hH][tT][mM][lL]?$");
del <- NULL;
# sort out
for (i in 1:length(listHtml)) {quest=FALSE;
# have to include "<!DOCTYPE" also
  if(scan(listHtml[i], character(), 1, quiet=TRUE)=="<!doctype") {
quest=TRUE } # doctype test
  if(Sys.readlink(listHtml[i])!="") {
quest=TRUE } # symlink test
  del <- append(del, quest)}
listHtml <- listHtml[!del]
```

11.3.3 Page Assembly

In the carpenter's version *Page Assembly and Media Check* included the collection of image `src` attributes. Now the page assembly is reduced to `file.append()`, `file.copy()` and `tempfile()`.

```
for (i in 1:length(listHtml)) {
  x <- tempfile(fileext=".html");
  file.append(x, c(t[1], listHtml[i], t[2]));
  file.copy(x, listHtml[i], overwrite=TRUE)
}
```


11.4 Image Handling

Relative links in `#+include:`'d ORG files are prepended by the relative inclusion path. If I include a file with

```
#+Include: "../../../tools/orCon.org::#attachments"
```

then a relative path like	↳ ../img/org/capture.png
is translated to	↳ ../../../../img/org/capture.png
A relative path like	↳ ./img/org/capture.png
would result in	↳ ../../../../tools/img/org/capture.png

In any of these cases I need to make sure that the image path begins with `↳ ../img`. This is a problem for file `↳ 02c.html` only, but I'm curiaous about a proper procedure for exchange.

11.4.1 Change Relative Image Links

Changing the relative image links exercises XPATH and XML node treatment in the external pointer concept of the R package `XML`. The two pages of Section 3.9 *Three Representations of the DOM Tree in R* in [15]'s Chapter 3 *Parsing XML Content* explain the concept. And they raise the awareness for the cloning feature. The description of `xmlClone()` in the function summary of [15]'s Chapter 6 *Generating XML* reveals a condensed version, which reflects the pointer-reference distinction in C-functions:

“Create a copy of the XML node or document provided. The element will be cloned to create a new C-level structure. The `recursive` argument indicates whether all the child nodes will be cloned as well, or only the top-level node. Cloning is not the same as assignment. When we clone, we make an explicit copy of the C data structure and return that copy (which may then be assigned to an R variable). Simply assigning an internal node to a variable does not make a copy of it, unlike most objects in R, but merely copies the `externalptr` object. As a result, a simple assignment means any subsequent changes to the node will

appear in all references to it. In contrast, when we clone a node or document, the original and the copy are independent copies and changes to one are not reflected in the other.”

— Description of `xmlClone()` in Section 6.8 *Summary of Functions to Create and Modify XML* of [15]’s Chapter 6 *Generating XML*, p.224

After all this is just a motivation for harnessing node related treatment in XML or HTML files. Perhaps for using R, too. The image detour begins with excluding HTML symlinks by `append()`’ing `Sys.readlink()` positives from the HTML file list acquired by `list.files()` in a `length()` `for(){}` loop.

```
listHtml <- list.files(path=www, recursive=TRUE,
  pattern=".[.] [hH] [tT] [mM] [lL]?$");
# sort out symbolic links
del <- NULL;
for (i in 1:length(listHtml)) {
  del <- append(del, Sys.readlink(listHtml[i])!="") }
listHtml <- listHtml[!del]
```

In the reduced file list

- `htmlParse()` prepares for node extraction of `src` attributes by `getNodeSet()`. The XPATH checks for `` elements with `src` attributes which do not begin with `../img/` or `http`.
- `sub()` replaces the first occurrence of its search pattern with the relative path to the parallel `img/` folder. The assignment operator sets the value for the `src` attribute.
- `saveXML()` copies the modified external pointer collection to the current HTML file.

```
for (i in 1:length(listHtml)) {
  duc <- XML::htmlParse(listHtml[i]); # str(duc)
  imgNode <- XML::getNodeSet(duc, "//img[ not ( starts-with(@src,
  ↪'../img/') ) or not ( starts-with(@src, 'http') ) ]");
  li <- length(imgNode);
  if(li>0) { for (j in 1:li) {
```

```
XML::xmlAttrs(imgNode[[j]])["src"] <-
  sub("[A-Z/ a-z.-]+/img/", "../img/",
    XML::xmlAttrs(imgNode[[j]])["src"]) }
dummy <- XML::saveXML(duc,listHtml[i]) } }
```

11.4.2 Collect Media

- Same selection process as in the [previous section](#) for the HTML files with the objective to skip symbolic links.

```
listHtml <- list.files(path=www, recursive=TRUE,
  pattern=".[hH][tT][mM][lL]?$");
del <- NULL;
for (i in 1:length(listHtml)) {
  del <- append(del, Sys.readlink(listHtml[i])!="") }
listHtml <- listHtml[!del]
```

- Read the `src` and `data` attributes in `` and `<object>` elements, respectively, by applying `getHTMLExternalFiles()` and `append()`'ing them to the `img` collection. The attribute `xpQuery` of `getHTMLExternalFiles()` is an XPATH expression and defaults to `c("//img/@src", "//link/@href", "//a/@href", "//script/@href", "//embed/@src")`.

```
img <- NULL;
for (i in 1:length(listHtml)) {
  duc <- XML::htmlParse(listHtml[i]); # str(duc)
  xpq <- c("//img/@src", "//object/@data");
  img <- append(img, XML::getHTMLExternalFiles(doc=duc,
    ↪xpQuery=xpq));
}
```

Note that the image paths derived from the `orgRoot` variable `pub` contain a symlink

- `unique()` the image paths and remove the HTTP paths with a `grep()` selector

```
uImg <- unique(img);
pImg <- uImg[grep("^http", uImg, invert=TRUE)]
```

- construct the source and the target path for the images with a `sub()` selector and `file.path()`
- check for changed image files with `file.exists()` and `md5sum()` – assuming that (1) all file paths begin with `../img/` and (2) the Web site only has one level of folders – and overwrite them with `file.copy()` if necessary
- check for new image files and copy them with folder check employing `dirname()`, `dir.create()`, `dir.exists()`, and `file.copy()`.

```
for (j in 1:length(pImg)) {
  cImg <- sub("^.[.]/", "", pImg[j]);
  fPic <- file.path(www, cImg);
  fPub <- file.path(pub, cImg);
  if (file.exists(fPic)) {
    sPic <- tools::md5sum(fPic);
    sPub <- tools::md5sum(fPub);
    if (sPic != sPub) {
      file.copy(fPub, fPic, overwrite=TRUE) }
    } else {dPic <- dirname(fPic);
  if(!dir.exists(dPic)) {
    dir.create(dPic,recursive=TRUE,mode="0755"); }
  file.copy(fPub, fPic); } }
```

11.4.3 Hidden Agenda for Images

My method doesn't include SVG files; they would be found by looking for the `data` attribute in an `<object>` element.

```
objDat <- XML::getNodeSet(duc, "//object/@data"); # str(objDat)
```

That's because I didn't find a proper way to address backend specific image production. I'd prefer PDF for LATEX and SVG for HTML, but as long I can't figure it out, I'll stay with PNG for both.

I'm still working on inclusion of my favourite LATEX TIKZ candidate for infographics, for situations where R's graphics solutions `graphics`, `lattice` or `ggplot2` are too tedious to design. But this approach went into competition with ASYMPTOTE and GNU PLOT. Or PLANTUML and

pure GRAPHVIZ solutions. Except for LATEX TIKZ I've set up ORG BABEL blocks for these graphics tools to put them up for experimental trials.

The backend specificity also applies to tables, but the ORG solution in this case is much more advanced. It even expands to an amazing ASCII mode spreadsheet feature, which might be the sole reason to turn into an ORG MODE monk.

The change log of ORG MODE offers a new procedure of backend independent one for all solution, i.e., SVG.

“ORG BABEL now uses a two-stage process for converting LATEX source blocks to SVG image files (when the extension of the output file is `.svg`). The first stage in the process converts the LATEX block into a PDF file, which is then converted into an SVG file in the second stage. The TEX \rightarrow PDF part uses the existing infrastructure for `org-babel-latex-tex-to-pdf`. The PDF \rightarrow SVG part uses a command specified in a new customization, `org-babel-latex-pdf-svg-process`. By default, this uses `inkscape` for conversion, but since it is fully customizable, any other command can be used in its place. For instance, `DVISVGM` might be used here. This two-part processing replaces the previous use of HTLATEX to process LATEX directly to SVG (HTLATEX is still used for HTML conversion).

Conversion to SVG exposes a number of additional customizations that give the user full control over the contents of the latex source block. `org-babel-latex-preamble`, `org-babel-latex-begin-env` and `org-babel-latex-end-env` are new customization options added to allow the user to specify the preamble and code that precedes and proceeds the contents of the source block.”

— *New options and new behavior for babel L^AT_EX SVG image files* change log [entry](#) for version 9.5, accessed 2022-07-26.

12 What's more

For me one of the most intriguing aspects of Internet pages is their cloaking mechanism for links. They hide all their ingenuity behind clickable words. And the most the reader can expect from the design of a complete Web site is a cloud of tags, maybe even categorized, but in the worst case automatically generated. On the other hand the digital high performance machines could produce a table of contents, pre-matter, back-matter, footnotes, captions, indices, and bibliographies which are the main source of quickly getting an informed impression of the content and to acknowledge the external contribution of larger documents. For reintroducing these features in a CMS we would have to browse through a jungle of extensions and fill a multitude of database tables.

Another unpleasing feature of the blogging mainstream is the reduction to linear storylines. New CSS features and increasing integration of SVG are on their way but even with the old CSS the user drowns in a sea of configurations. For the creation of something in the range of o'Reilly's Head First series the author is urged to employ a whole industry of publication professionals. In this regard `ORG MODE` and its `EMACS` base can offer their embracing features, as Niesson's example shows.

For me it's important to use a tool which offers a starting point for perfect `LATEX` and `HTML` and `TEXINFO` exports, instead of reduced approaches like `RSTUDIO-MARKDOWN-HTML-RSHINY` or `LATEX-SWEAVE-PDF`. Of course, the configuration efforts will increase, because I'm dealing with different media or formats; with different concepts. IMHO that should be the real realm of responsive design discussions; or *barrier-free access*, to use a term that deals with people instead of devices. I can't produce a proper PDF from a `HTML MARKDOWN`, nor can I produce an animated `HTML` page from `DVI LATEX`. While it might be a proper idea to produce a page oriented `BEAMER` presentation from `LATEX`; or to transform a `TIKZ` portable graphics format¹⁴ to `SVG`.

¹⁴`TIKZ` is a parallel production of the `BEAMER` guy Tantau. `TIKZ` includes animation. A rapid slide show can be marked as the transition state to a motion picture. That's where I dare to compare `TIKZ` to `SVG`.

With regard to the *simple* example I'll sketch some expansions into a “what's next” agenda.

- Section 12.1 collects publishing functions, pre and post processors in ORG publish. And it introduces EMACS shell escapes, because I render them the first candidates for my experiments in ELISP.
- Section 12.2 is about extracting information from an ORG file and its LATEX or HTML exports; or RSS or TEXINFO. Introduction to XML node retrieval and manipulation. The outline covers bibliography, index, and list of figures.
- Section 12.3 discovers the chain: selection of files → Web *site* → site map, RSS → content management → information architecture.
- Section 12.4 reviews my ideas about HTML templates for Web *site* structure
- Section 12.5 offers my favourite deployment scenarios without naming many other failures
- Then in Section 12.6 I conclude the whole document with the spirit of my endeavor by comparing it to Berners-Lee's AMAYA vision.

12.1 Publishing Hooks

Apart from the `:publication` function which already can be a *list* of functions I identified three injection methods for publishing hooks.¹⁵

The main publishing hooks are called `:preparation-function` and `:completion-function`; The corresponding Manual [entry](#) tells us that “functions listed in these properties are to be called before starting [or after finishing] the publishing process, for example, to run `make` for updating files to be published [or change permissions of the resulting files]. Each function is called with a single argument: the project properties list.” I've no idea what the single argument *project properties list* means. For getting ideas I sketched Ogbe's [\[16\]](#) usage of pre- and postprocessors in Section [13.1.7](#).

¹⁵For now I can only guess their usage. My work-around is to use R for all the programmatical purposes in a semi-manual deployment scenario. ELISP intelligence postponed.

The function list `org-publish-after-publishing-hook` is another spot for getting information; it's called in `org-publish-file` which is part of `org-publish-projects` and `org-publish-current-file`. The default arguments are the source and the target file paths.

For me the syntax for an ELISP shell call on EMACS level might be the entry to ELISP programming. Such a shell call looks like (`shell-command "ls"`). In Table 4 I sorted the set of related definitions and commands into short and longer commands and removed their common prefix `shell-command-`.

Table 4 – EMACS shell escapes in ELISP without the preceding `shell-command-`.

<code>completion</code>	<code>saved-pos</code>	<code>-save-pos-or-erase</code>	<code>dont-erase-buffer</code>
<code>history</code>	<code>sentinel</code>	<code>-set-point-after-cmd</code>	<code>separator-regexp</code>
<code>on-region</code>	<code>switch</code>	<code>completion-function</code>	<code>with-editor-mode</code>
<code>regexp</code>	<code>to-string</code>	<code>default-error-buffer</code>	

For asynchronous calls – which don't block the whole EMACS instance – there are commands like `start-process` or `start-process-shell-command`; see Xah Lee's *Elisp: Call Shell Command* [10] at xahlee.info or the whole `subr.el` with the basic ELISP subroutines.

Ogbe's [16] usage of a shell command regards `CSSTidy`, sketched in Section 13.1.

12.2 Index Toc Tag Category Bibentry Link-List

Two of my next steps begin as a matter of the individual post. The first one is already realized on this volatile example site

- harness the BIBTEX run of PDFLATEX combined with `BIB-TEX2HTML` and the by-product of a supplemental PDF
- extract the HTML reference attributes `href` of the anchors `<a>`. Sort and unique them.

The full line of “self reference” approaches begins with the table of contents, the list of figures or tables, and indexes; next we can add appendices with all kind of specific supplemental information, usually arranged in a back matter part, or the content related information of

the pre-matter. What about mini tocs, footnotes, endnotes, examples, theorems, definitions, exhaustive info boxes or other boxes for warnings, hints, notes? The *L^AT_EX Companion* [12] dedicates the first Chapter *The Structure of a L^AT_EX Document* to most of these meta informations. The Subsection 3.3.3 *amsthm* – *Providing headed lists*, Section 3.3 *List structures* in [12]’s Chapter 3 *Basic Formatting Tools* provides for more. And o’Reilly’s *Head First* series shows the variety in non-math books.

Keeping track of all the references for the whole Web site is a major task of the CMS. Administrative tasks yield another field of applications. Both of them constructing their own display *modules*, just like the MYSQL PHP combination of the global players. In the case of BIBTEX my coordinated ORG MODE approach produced both a table for the posted file and an additional bibliography file; *and* a PDF which is not derived from the templated HTML export. I consider this a clean start. And this wouldn’t be a *What’s more* section if it didn’t offer more about BIBTEX, links, and “self scraping.”

12.2.1 BibTeX

To get started I only put up the resulting files for the bibliography and the PDFLATEX compilation; see Section 11.2 for elaboration. To step into “reproducible publication” I will have to provide

- the folder structure needed for the LATEX compilation
- a (reduced?) LATEX template with (reduced?) input files
- the ORG macros in `⊞ config.org`, my setup file, together with the corresponding LATEX macros and the ELISP code for the macro edits. I didn’t configure the LATEX compilation in ORG MODE, yet?, and the *listings* macros are still in the development cue, too. There might be other solutions like LATEX’s *minted* related to PYTHON’s *pygments*. For the HTML export ORG employs *htmlize*, which reflects the current buffer theme.
- my customized user entities as an ELISP entry for the `⊞ ~/.emacs` file.

Next step is to compare `org-bibtex`,¹⁶ `reftex`,¹⁷ `ox-bibtex`,¹⁸ the very new¹⁹ `org-citation`, and external libraries like `RefManageR` or services like ZOTERO, which probably includes expanding to LINKED DATA. The `org-citation` approach is done with contribution of Kitchin's `org-ref` which

“is an EMACS-LISP module to handle bibliographic citations, and references to figures, tables and sections [... and] was written first for use in ORG-MODE, and for reasonable export to LATEX. It does not work well for any other export (e.g. HTML) for now.”

— *Using org-ref for citations and references* a 2014-05-13 blog `entry` by Kitchin at `kitchingroup.cheme.cmu.edu`.

`org-ref` is an approach with `Citation Style Language`. Kitchin doesn't work with `ox-bibtex` or `BIBTEX2HTML`: “I don't plan to use `bibtex2html` in `org-ref`. It is not easy to install on Windows. The approach `org-ref` will take is described `here`.” `github org-ref issue 101`, 2016-01-10. The version `org-ref 2.0.0`, accessed 2021-10-01, available at `MELPA`, requires `dash-2.11.0`, `htmlize-1.51`, `helm-1.5.5`, `helm-bibtex-2.0.0`, `ivy-0.8.0`, `hydra-0.13.2`, `key-chord-0`, `s-1.10.0`, `f-0.18.0`, `pdf-tools-0.7`, `bibtex-completion-0 1.1.1`.

For the acknowledgment aspect I'm using hints from *Cite it Right* [8], *Who Did What? The Roles of R Package Authors and How to Refer to Them* [9], and *Free Software, Free Society* [7].

12.2.2 Links and Indices

To extract the HTML anchors to several indices and to insert back reference anchors could be another option. Here, my reference for insertion is [15]'s Chapter 6 *Generating XML*. And perhaps I would have to restructure the descriptive parts of my ORG links.

¹⁶According to a 2019-04-20 `discussion issue` at `bibeltex`, referring to the corresponding git `commit entry` at `code.orgmode.org` `⊃ org-bibtex.el` was renamed to `⊃ ol-bibtex.el`.

¹⁷Examine `reftex-create-bibtex-file`, `⊃ reftex-cite.el` or `⊃ reftex.el` or the `REFTEX Manual`.

¹⁸Containing many `org-bibtex-...` commands, but not part of GNU EMACS)

¹⁹ORG version 9.5

In my first long version I didn't put every citation into a bibliography link. I think the links to R functions or even Manual pages should be part of a different info system. XPATH mediated through [15]'s *Chapter 3 Parsing XML Content*, *4 XPath, XPointer, and XInclude*, and *Chapter 5 Strategies for Extracting Data from HTML and XML Content* sets the stage.

With this preparation I can compare the efforts

- to setup a NLTK crawler [2] for tagging and LINKED DATA methods [24] for categorization; the latter may be combined with RSS export discussed in Section 13.1.1.
- of macro tagging, capturing, and the many org hyperlink versions, which include archive, agenda, the whole org-as-a-note-taking system
- to manually enter the keys for the elaborated index mechanism of the TEXINFO export,²⁰ and the special index property for the properties' drawer of a headline: either render exported INFO files, use PANDOC, or REGEX in the ORG file source
- to manually enter the keys for the ORG publishing *feature* of *Generating an index*

For the anchor's href attributes there's another challenge: bookmark management. Ever run into HTTP 404 of other pages? It hurts even more from the own pages. In 2008 I found JOOMLA BOOKMARKS to be a major inspiration for this management; complemented by LINK-AGOGO (still HTTP, no s) and MEMOTOO to collect FIREFOX bookmarks or IE favorites. Where MEMOTOO also offers ideas for address and eMail management, particularly the SYNCML background. They might be thought to be outdated by POCKET or REDDIT, but we all know that these startup goodies quickly turn into price tagged data collectors; see the delicio.us to PINBOARD transformation.

For ideas about numbered theorems or examples see the pre-defined macros in the ORG Manual's Section *Macro Replacement*. It contains

²⁰TEXINFO offers concept, function, key, #+PINDEX, #+TINDEX, and #+VINDEXT,...

the concept of custom counters in numbered theorems or examples or exercises that can be produced by the pre-defined macro `n(m,x)`.

In the forms `n`, `n(NAME)`, and `n(NAME,ACTION)` this macro implements custom counters by returning the number of times the macro has been expanded so far while exporting the buffer. You can create more than one counter using different `NAME` values. If `ACTION` is "-" the previous value of the counter is held, i.e. the specified counter is not incremented. If the value is a number, the specified counter is set to that value. If it is any other non-empty string, the specified counter is reset to 1. You may leave `NAME` empty to reset the default counter.

For the construction of theorem environments in `ORG` I collected four sources

- WORG [entry](#) *org-special-blocks.el — turn blocks into L^AT_EX envs and HTML divs*
- STACKOVERFLOW [entry](#) *org-mode special blocks latex attribute*
- GITHUB repository [latexcss](#), a class-less CSS file which can be attached to any HTML document to make it look like L^AT_EX. It combines *L^AT_EX Theorem-like Environments for the Web*, Zachary Harmany, 2013-01-17 at his octopress blog and a L^AT_EX style WIKIPEDIA CSS.

This might be useful on a single page. But for distributed pages there has to be a linking idea like the one concatenating the HTML version of a [large book series](#) by Julius O. Smith III. – presenting, for example, [Physical Audio Signal Processing](#). IMHO Smith's solution is not maintainable in that form. The discussion above could inspire other approaches.

12.2.3 XML Tools for Attribution Retrieval

As we can see from Section [11.4](#) the HTML image elements can be handled by XML node recognition. In this section I back up this practice with Temple Lang's statements about regEx not being a solid solution for handling XML. As an exercise I invite the reader to my expedition where I was looking for some background information of the media files in the borrowed exercise layout and reasoned about license aspects. Then I set the scene for another XML node based information retrieval of license information with regard to L^AT_EX's list of figures mechanism.

Remember, the exercise layout was borrowed from `selfhtml.org` (German) and stated as public domain; `ZIP` and `preview` (`gooTrans`) provided. It belongs to an HTML intro called *HTML Einstieg* (HTML Intro → `gT`). The images needed for the exercise are from the ZIP. But first some quotes about the XML node discussion.

“While we have not formally introduced the R functions for working with XML content, it is useful to note that the rich structure and formal grammar of XML makes it easy to work with XML documents. For example, we can find all `<email>` elements, or all `<r:func>` or `<r:package>` nodes. We can even locate the `<section>` node in a book which is, e.g.,

- within a chapter whose title contains the phrase *social network* and
- which has a paragraph with `<r:code>` that contains a call to load the `graph` package.

These are significantly harder to do robustly with markup languages such as LATEX or MARKDOWN since they do not have formal grammars. Typically, people use line-oriented regular expressions for querying such documents and so cannot use the hierarchical context to locate nodes. This also makes it much harder to programmatically update content.”

— Example 2-1 *A DocBook Document* of Section 2.3
Examples of XML Grammars in [15]’s Chapter 2 *An Introduction to XML*, p.30

That’s a statement in the introductory chapter about XML. But it really gets interesting when this memorandum is translated into action, which is the topic of [15]’s Chapter 6 *Generating XML*.

“Parsing and querying XML is more difficult using regular expressions than with an XML parser, and it is especially challenging for HTML due to its often irregular or malformed structure. With an XML parser, we can work with the tree and individual nodes. We can query and modify nodes of interest and adapt the tree. We can then serialize the result

back to a string if we want, e.g., to write to a file. In other words, creating XML content via string manipulation works adequately, but we typically want to operate at a higher level with nodes and trees. By parsing the string content, we can continue to think in terms of working with tree and node objects.”

— Section 6.1 *Introduction: A Few Ideas on Building XML Documents* in [15]’s Chapter 6 *Generating XML*, p.184

Also as part of this XML editing chapter Temple Lang concedes that direct text manipulation and `regEx` is sometimes useful. He enciphers this into *vectorized generation of XML*.

“We have indicated that creating node objects and combining them into trees using `newXMLNode()` and the other functions is a good and robust approach to create XML content, and that creating XML by pasting strings together is less robust and flexible. However, there are occasions when string manipulation to create XML/HTML content is useful. These are typically when we need to create many nodes that have the same structure but with different values in the content or attributes.”

```
?XML::newXMLNode
```

— Section 6.5 *Vectorized Generation of XML Using Text Manipulation* in [15]’s Chapter 6 *Generating XML*, p.206

Unfortunately there’s much more to know before applying this advance information. Fortunately the book [15] provides the tools and plenty of examples; see Section 12.2.2 for related chapters. Unfortunately some of the examples are not accessible any more. In the quoted section I also recommended to back up the studies by the usage of another book [14] developed by practitioners of political sociology. They provide a – still available – repository of their examples.






Or you can just follow my example in Section 11.4.1 and combine it with the challenge sketched in the rest of this section, which reflects on


the exercise layout from selfhtml.org (German) stated as public domain. The following block is related to the images in the corresponding ZIP file.

Image License Investigation

From image-search.org I'm introduced to the insight that the rocking chair, for example, needs the attribution

```
<a  
->href="https://www.vecteezy.com/free-vector/rocking-chair">Rocking  
Chair Vectors by Vecteezy</a>
```

The  [desk.svg](#) is borrowed from the WIKIMEDIA commons  [ryansdesk.svg](#). For the unused  [x-tree.jpg](#) I followed an image-search.org [link](#) to the BING engine which brings up the file  [tree55.jpg](#) at free-clip-art-images.net. For the  [hobel.svg](#) icon YANDEX – mediated by image-search.org again – found similar sources at cleanpng.com, dlpng.com, pngwing.com, and ya-webdesign.com, all of which claim personal usage only. I could assume that the red point version at SELFHTML is a SELFHTML product, but I wouldn't provide the file as public domain. So, I take three measures

- switch the logo to an ugly handmade version  [hobel.png](#)
- unlink the rocking chair and Ryan's desk – they don't go well with the style of the other images anyway
- include the WIKIMEDIA commons [attribution](#) for the rest of the media files.

As an additional measure I contacted Selfhtml e.V. for a statement about their public domain statement; the chairman Scharwies confirmed this status.

Mr. Scharwies, since 2016 chairman of the NGO Selfhtml e.V. told me that the exercise ZIP contains pictures, which are collected *mainly* from WIKIMEDIA commons.

I can find most of the images in the category [SVG Furniture](#) at commons.wikimedia.org. Based on the available information I'm constructing a check list to decide about the inclusion of the files.

- [filing-cabinet.svg](#) is a copy [Paul Robinson's cabinet.svg](#)

- cabinet.svg → Matthias Scharwies' Cabinet2.svg
- dresser.svg → Seahen's Dresser.svg
- table.svg → Seahen's Table.svg
- cupboard.svg → Matthias Scharwies' Cupboard.svg,
- hobel.png my public domain contribution

Wikimedia Common URL RegEx Behavior

When handling the corresponding link I observe an interesting ORG export effect. It occurs for links with a name tail that looks – for regEx – like a file extension. For example the URL of the SVG file entry at commons.wikimedia.org/wiki is `File:Nuvola_filing_cabinet.svg`. While the file itself resides at `https://upload.wikimedia.org/wikipedia/commons/b/bf`.

For ORG export both URI's in the plain form `[[URI]]` look like an SVG, which for the first link results in

```
<object type="image/svg+xml" class="org-svg" data="https:// \
commons.wikimedia.org/wiki/File:Nuvola_filing_cabinet.svg">
Sorry, your browser does not support SVG.</object>
```

(The backslash is a manually inserted linebreak.) For the second link this is the expected translation. But if I load a page containing the translation of the first link the browser shortly tries to embed the file as SVG object and then immediately switches to the `...cabinet.svg` page at commons.wikimedia.org. If a page contains multiple links like this the browser arrives at the last of these WIKI COMMONS pages. This doesn't happen when I include the URI in a named link like `[[URI] [name]]`; see the ORG Manual's [Section *Links in HTML export*](#).

If I want to state a by-sa copyright for my own work the [proposal](#) at creativecommons.org offers the code

```
<a rel="license"
↪href="http://creativecommons.org/licenses/by-sa/4.0/">
  
</a><br />This work is licensed under a <a rel="license"
href="http://creativecommons.org/licenses/by-sa/4.0/">Creative
↪Commons
```


Attribution 4.0 International License.

When quoting another person’s work I can do it in situ or at a central location, the list of figures; see Section 12.2 for more reference lists. LATEX’s list of figures depends on floating figure environments or manual `\addtocontents{}` and `\addcontentsline{}` entries. Usually I would think of an ORG MODE macro to translate the license information to LATEX’s figure list commands and to the HTML code above. The list of figure files of LATEX might then be combined with XML node retrieval of HTML media elements like ``, `<object>`, or `<video>`, which in turn can be pointed at the original site to parse license information.

An example for accepted in situ quotation infos is provided by the WIKIMEDIA commons entry *How to Give Attribution*. It recommends the caption in Figure ?? for the example of a FLICKR source.

https://creativecommons.org/wp-content/uploads/2019/10/38494602082_d135ee9c7c_k-768x432.jpg

How to attribute authors of the CC works will depend on whether you modify the content, if you create a derivative, if there are multiple sources, etc. But the caption in Figure ?? is supposed to be an ideal attribution note because it includes the:

- Title: “Furggelen afterglow”
- Creator: “Lukas Schlagenhaut” – with a link to their profile page
- Source: “Furggelen afterglow” – with a link to the original photo on FLICKR
- License: “CC BY-ND 2.0” – with a link to the license deed

The example above works for tedious in situ maintenance of a few external contributions. It will show proper results for LATEX and HTML export. The *L^AT_EX Companion* [12] offers background information in Section 2.3 *Table of Contents Structures* and expands thoroughly in Chapter 6 *Mastering Floats*.

My example in Section 11.4.1 shows the node treatment in action and [15]’s Chapter 6 *Generating XML* provides the tools for constructing an HTML snippet comparable to the BIBTEX2HTML processes in Section 11.2.2. Parsing the target URI’s for retrieving the information is an option, if it’s worth the effort; there’s plenty to discover.

I wonder if the study of [3]’s Chapter 11 *Emacs Lisp Program-*

ming would put me into a position to turn the procedure into a `:publishing-function` or begin with a hook; see Section 12.1. I'm sure that EMACS has some reasonable access to LIBCURL, too. Duck-Duck goes for Syohex's [emacs-curl](#) and Xu Chun Yang's [curl-to-elisp](#). But coming from the multiple options of PYTHON's Beautiful Soup or NLTK I'm happy to have arrived at a basic understanding of HTTP with the help of Temple Lang's [XML](#). Don't hurry, be happy.

12.3 Select Files and Content

The short version example of this document is a one level Web site. All HTML files are in the root. And for the example I imputed a template. In the long version postings and structural files get their own folder. But more content will raise the challenge of a further expanded folder structure. There many options because the physical location of the files doesn't have to reflect the information architecture. In this lookout section I'm collecting ideas for the implementation of both folder and information structure. I'd consider Section 11 as the generator for the second tangible product in this regard.

The FSFE Web site building [shell scripts](#) can help for identifying the main topics of the issue. The approach of the FSFE team is to use and painstakingly edit XML and XSL files for every purpose. Apart from the discipline this measure takes it's an absolutely transparent process. So, by experimenting with Morville's wireframe and blueprint design [13], Chapter 12 *Design and Documentation* I can assume to get some flesh on the bones of a structural skeleton.

That only regards the public elements of an information system. For structuring a private or single person's publication's efforts my major field of interest is to extract elements of my work to the publishing stage. I think about

- Publishing as a spotlight on an issue that is developed as the part of a semi-finished book, or a categorized collection. This can be mediated by ORG MODE's customizable select and exclude tags. The CSS for these pages can be based on Niessen's [readtheorg.org](#). The problem to explode the whole work into single chapters or sections, in order to reduce the resource us-

age, might be addressed by manual selection, XML aftermath, or TEXINFO procedures.

- The structure for reusable parts probably deviates from this pattern. It's more like linking or referencing using LINKED DATA or bibliographic methods.

For choosing sources of publication ORG publish first draws on contents of the `:base-directory` with an optional `:recursive t` expansion. The choice depends on the `:base-extension` which defaults to ORG files. See [20]'s Section *Glue HTML and Extract Images* for an expression to choose media files. The two properties `:exclude` and `:include` deselect and select on a regEx or file basis, respectively; see Section *Selecting Files* of the Manual's *Publishing* Chapter.

File Selection History

The 2012-02-26 answer of [cm2](#) to the 2011-03-07 STACKOVERFLOW entry *Customizing org-publish-project-alist* offers hints for (1) deleting an HTML file in the `:publishing-directory` if its counterpart ORG file in the `:base-directory` is missing and (2) inserting the CSS file link.

1. writing a `:completion-function` `cm2` links to Rose's WORG entry *Publishing Tree menus for Org-files* [21]. The function is derived from `org-publish-org-sitemap` and reads a menu structure from a dedicated syntax and somehow combines it with the auto-sitemap. According to `cm2` the function "shows how to get property values for the `org-publish-project-alist`."

Opposed to that statement for me the main information in this linked blog is to manually *provide* an intuitive syntax for the necessary infos of a menu structure:

- number of dots .. for the level
- text .. for the menu text
- link .. relative URL for the link target
- title .. either for a tab title or the article title, I guess

- target .. probably the target modul, in JOOMLA terms, of the page, i.e., where to include the menu as margin tree or top or bottom menu
 - expanded .. probably a switch for the initial viewing state of the menu: expanded or collapsed.
2. for the CSS file link the user cm2 suggests
- a file based `#+STYLE:` entry
 - a setup file inclusion like in the *Tired of Export Templates* Section of Rose's WORG entry *Publishing Org-mode files to HTML* [22]. cm2 calls this "generate generic template files for each level within the directory tree."
 - engaging an `:preparation-function` to call a shell script. For example, by invoking the stream editor `SED` with something like `href="@MYLOC@/stylesheet.css" />`.

The publishing scaffold below contains the idea of putting a symbolic link of a file into the publishing folder. In the corresponding link source the sections marked by `SELECT_TAGS` like the default `:export:`, or everything but `EXCLUDE_TAGS`, default `:noexport:`, are extracted for publishing; see the *Export Settings* Section in the Manual's *Exporting* Chapter and the doc-strings of `org-export-exclude-tags` or `org-export-select-tags`.

The `search options in file links` combined with `#+include:`²¹ are another source of inspiration. Or the concept of `attachments`.

For access to the absolute root of the finally deployed Website we can employ the org buffer entry `#+HTML_LINK_HOME:`, corresponding to `org-html-link-home`. And consult the doc-strings of `org-html-link-use-abs-url`, `org-html-home/up-format`, `org-html-link-up`.

In an HTML export something like `[[file:./r.org:#graph-mod][R Graphic Models]]` is turned into the HTML anchor

²¹See the Manual's Section and the doc-string of `org-export--prepare-file-contents`.

```
<a href="./r.html#graph-mod">R Graphic Models</a>
```

An ID [search option](#) like `[[file:./rG.org:#ch1]]` would include the corresponding ID as `href="./rG.html#ch1"`; see [Links in HTML export](#) and [Publishing links](#). These references assume very disciplined organizational measures, which are very hard to maintain in something like blog entries. So I suppose the linking mechanism is designed for `index.html` files which reflect the Web site structure.

Here's a typical ill-structured tree of ideas about wiring the hard disk to a publication folder by harnessing symlinks, `ORG #+include:'s`, and `ORG search options`. The `ORG` files themselves can provide additional selection features like tags.

- `blog/`
 - `index.org`
 - `2020-04-25.org symLink → relative/path/2/foo.org`
 - `2021-05-29.org symLink → rel/pa/2/bar.org`
- `book/`
 - `index.org`
 - `lnx.org → /abs/path/2/00far.org`
 - `rGraphMod.org` contains


```
#+Include: "rel/pa/2/r.org::Graphical Models" :lines 2-
```
 - `xml.org` contains `[[file:.lnx.org][WebTech Cmd Line]]`
- `tuts/`
 - `index.org`
 - `sound.org` contains


```
* sec 1
#+Include: "rel/pa/2/hlmhltz::#fourier" :lines 2-
** sec 1.2
#+Include: "rel/pa/3/physics::#wave" :lines 20-58
* sec 2
#+Include: "rel/pa/4/math::#axiom6" :only-contents
```

- `xml.org` contains `[[file:blog/2020-04-25.org] [Insight]]` and `[[file:book/lnx.org:~#apx2] [Data]]`

- `index.org`

The real files in this tree, opposed to the symlinks, would contain the Web site structure. I can think of three already available mechanisms for automatic registration of the symlinks and population of the index files

- For every publishing project we can utilize the `:sitemap-..` properties triggered by `:auto-sitemap t`; see [Section *Generating a sitemap*](#). The default sitemap procedure generates a plain list of links to all files in the project, and can be customized by `:sitemap-function`.
- The un'manual'ed RSS feature of `org publish` works with the `:publishing-function (org-rss-publish-to-rss)`. Its docstring leads to `⊃ ox-rss.el`. So basically it seems to be designed as an export feature. Well, `ORG publish` itself is also integrated in the `ox-ELISP` family. For the RSS feature see *Blogging from GNU Emacs* a 2013-09-25 blog [entry](#) by Bastien Guerry at `bzg.fr`.
- The publishing index generator, already discussed in [Section 12.2.2](#).

Diving into theoretical aspects of information architecture can be a distracting mental training effort; see [Section 11](#) for practice. I'll just add one thought about long documents: if I happen to produce very long blogs I'd have to think about splitting the pages. There are CSS solutions for pagination, but I prefer structural splitting. The explosion to small chapters or sections might be solved by `TEXINFO HTML` exports. I'm talking about `TEXINFO` features, not `ORG MODE` exports; they might already be integrated into `ORG MODE` or `EMACS` like `HTLATEX` for interpreting `LATEX` snippets for `ORG MODE HTML` export.

12.4 Template Development

The combination of exports and the template is part of the image collection code in [\[20\]](#)'s [Section *Glue HTML and Extract Images*](#), but

there are other options for the expansion of the export files:

- produce templates by a combination of ORG MODE's NOWEB, tangling and export features. And add customized features with any favorite programming language.
- replace designated locations with corresponding modules, using R XML's node editing features, explained in [15]'s Chapter 6 *Generating XML*.
- design the template in HTML cut it in pieces and LINUX-CAT it around the ORG export to build a Web site page
- the utilization of a CDN feature like NETLIFY's *File-Based Configuration* is already mentioned in [20]'s Section *Glue HTML Expanded*.
- see Ogbe's blog for another template idea using `:html-preamble` `my-blog-header` and `:html-postamble` `,my-blog-footer` for inserting the header and the footer
- XTiger Language Specification; see *Templates in Amaya* at w3.org.
- *Building an Automatic Template System* in [3]'s Chapter 11 *Emacs Lisp Programming*
- employing Daniel Pfeiffer's EMACS *skeleton*. Interesting for usual edits – or for interactive template construction? See EMACS Manual Chapter *Commands for Human Languages* and the independent info file about *Autotyping*.

For the ORG MODE approach the first idea is the most attractive one. Just like Niessen's CSS tool it offers a self documenting environment for template development. This procedure is sketched in [20]'s Section *Glue HTML Expanded*. The template code with the NOWEB reference is not a valid HTML document, but this is solved in the short example by a previous tangling procedure. I still can't escape the short R code in Section 11.3.3. Perhaps I have to think it as ELISP solution? As a ELISP not-even-rookie?

Why worrying about tangling, nowebbing, whatever? I can just cut the template file into – in this *simple* case – two pieces, put the export in between, and save it in my local upload folder. With the LINUX `cat` command this process looks like

```
cat tmp1t1.html index.html tmp1t2.html > ~/www/shtm0/index.html
```

Maybe it's also helpful to know that LINUX commands can work with the standard input represented by a dash

```
cat index.html | cat tmp1t1.html - tmp1t2.html > ~
↪~/www/shtm0/index.html
```

With these commands at hand the fusion of the web pages would be part of a bash script. Which can be included in and triggered from an ORG file source code block. But my horizon didn't evolve to the bash script loop skills, yet.

For my skills it still is easier to stick to R. But it seems to be kind of a handicap, too. The code for the short version is part of [20]'s Section *Glue HTML and Extract Images*, the long version is offered in Section 11.3.

12.5 Netlify Alternatives

A cost-effective solution within NETLIFY could be the NETLIFY command line interface; see *Get started with Netlify CLI* [entry](#) at docs.netlify.com. The billing system of NETLIFY involves *build minutes* for automatic deployments via GITHUB or GITLAB repositories. As far as I deciphered the NETLIFY CLI process it employs manual deployments from a local GIT repository, which doesn't count for billing. This approach should even work for collaborative configurations.

WORDPRESS offers a XMLRPC interface that works with free accounts, while the REST API is for the paid plans only. The XMLRPC approach is sketched in [15].²² But it consumes much effort for serving serialization, tagging, media upload and the whole CMS structure, while resulting just in a blog entry with tags. The free version, for

²²Subsection 11.3.1 *Programmatically Accessing a Blog*, Section 11.3 *Writing R Functions to Use XML-RPC* in [15]'s Chapter 11 *Simple Web Services and Remote Method Calls with XML-RPC*.

example, doesn't provide MATHJAX support. Nonetheless the XML-RPC interface invites to examine related blogging giants like GOOGLE's BLOGGER sphere.

Then there are some media sites like `medium.com`; unfortunately the MATHJAX restriction also counts for `medium.com`, but at `medium.com` they offer a *free* REST API.

Anyway it's not an option to point out the currently 10 best hosting solutions, because they already changed when such an endeavor is finished.

12.6 Weaving Amaya

By working out the details of my next publications steps I began to see connections to the [Amaya project](#) documented at `w3.org`. The corresponding [mail archive](#) ended in 2010, if I dare to neglect four outliers until 2014. IMHO AMAYA was the most central project of the World Wide Web and it drowned in the sea of knowledge it was supposed to discover. I think `ORG` \rightarrow `EMACS` \rightarrow `GNU LINUX` has the tools for a revival, but there's no reason to focus on the Internet. The Web is just another medium. The main objective is information architecture. And to be reliable it has to be freely accessible, that's the realm of the Free Software Foundation, which in turn is deeply ingrained in the same `GNU LINUX` environment. The content of the short version of this blog describes one possible workflow for the first steps; with the following quotes I invite the reader to draw connections from the second long version to AMAYA.

For shifting the focus away from the Internet the corresponding toolbox won't be an all including software in a browser; it's not helpful to interpret any information-related topic in terms of Web technology. The real challenge is to make the Internet available as the democratic medium it was designed for. The AMAYA project was aimed at this target, so I find it helpful to sketch the motivation and environment for AMAYA presented in *Weaving the Web* [1].

[ANNOTEA](#), [XTIGER](#) and the [AMAYA documentation](#) deliver the specifications for a single Web site. The last chapter of [1], *Information Management: A Proposal*, delivers the specification for the World Wide Web presented in 1989 and 1990. And here are the book parts men-

tioning the AMAYA project. First it's about INRIA as a connecting link between GRIF, ARENA, and AMAYA:

“CERN’s resignation left the consortium without a European base, but the solution was at hand. I had already visited the Institut National de Recherche en Informatique et en Automatique (INRIA), France’s National Institute for Research in Computer Science and Control, at its site near Versailles. It had world-recognized expertise in communications: their Grenoble site had developed the hypertext browser/editor spun off as GRIF that I had been so enamored with. Furthermore, I found that Jean-François Abramatic and Gilles Kahn, two INRIA directors, understood perfectly well what I needed. INRIA became cohost of the consortium. Later, in early 1996, we would arrange that Vincent Quint and Irene Vatton, who had continued to develop GRIF would join the consortium staff. They would further develop the software, renamed AMAYA, replacing ARENA as the consortium’s flagship browser/editor.”

— *Weaving the Web* [1], Chapter 8 *Consortium* p.101f

Then some explanations about the tasks of AMAYA; to me they just sound like [bluefish](#) or [geany](#).

“In 1996 we negotiated the right to the GRIF code from INRIA and renamed it “Amaya.” It is designed completely around the idea of interactively editing and browsing hypertext, rather than simply processing raw incoming HTML so it can be displayed on the user’s screen. AMAYA can display a document, show a map of its structure, allow the viewer to edit it, and save it straight back to the Web server it came from. It is a great tool for developing new features, and for showing how features from various text-editing programs can be combined into one superior browser/editor, which will help people work together. I switched from AOLPRESS to AMAYA.”

— [1]’s Chapter 9 *Competition and Consensus*, p.119f

Towards the end of the book, it develops to a dense multitasking environment with immense integration needs. Even if it just touches the usual markup languages. And for collaborative work the book mentions JIGSAW.

“We are only in the early stages, but we now have an environment in which people who are collaborating with the consortium write and edit hypertext, and save the results back to our server. AMAYA, the browser/editor, handles HTML, XML, Cascading Stylesheets, Portable Network Graphics, and a prototype of Scalable Vector Graphics and Math ML. While we have always developed AMAYA on the LINUX operating system, the AMAYA team has adapted it for the WINDOWS NT platform common in business, too. I now road test the latest versions of these tools as soon as I can get them, sending back crash reports on a bad day and occasionally a bottle of champagne on a good one.

We are using our open source JAVA-based server, JIGSAW, for collaborative work. For example, JIGSAW allows direct editing, saves the various edited versions of a document, and keeps track of what has been changed from one version to the next. I can call up a list of all versions, with details about who made which changes when, and revert to an older version if necessary. This provides everyone with a feeling of safety, and they are more inclined to share the editing of a piece of work. JIGSAW and AMAYA allow our team space to come alive as our common room, internal library, and virtual coffee machine around which staff members who are in France, Massachusetts, Japan, or on an airplane can gather.

Making collaboration work is a challenge. It is also fun, because it involves the most grassroots and collegial side of the Web community. All Web code, since my first release in 1991, has been open source software: Anyone can scoop up the source code – the lines of programming – and edit and rebuild them, for free. The members of the original www-

talk mailing routinely picked up new versions of the original Web code library “libwww.” This software still exists on the consortium’s public server, www.w3.org, maintained for many years by Henrik Nielsen, the cheerful Dane who managed it at CERN and now MIT. LIBWWW is used as part of AMAYA, and the rest of AMAYA and JIGSAW are open source in the same way. There are a lot of people who may not be inclined to join working groups and edit specifications, but are happy to join in making a good bit of software better. Those who are inspired to try AMAYA or JIGSAW, want to help improve them, develop a product based on them, or pick apart the code and create an altogether better client or server can simply go to the w3.org site and take it from there, whether or not they are members of the consortium.”

— [1]’s Chapter 12 *Mind to Mind*, p.170f

The **ANNOTE**A specifications are easily contained within ORG MODE hyperlinks and necessarily separated from the RSS export. And I think inventions like *bookmarklets* or *annozilla* are outdated by linked data.

I can see many initiatives which are willing to invest the effort for generating informed decisions without outsourcing the reasoning to so called professionals linked to institutions of excellence. I might be on the way to offer proper tools; I’ll see if they deliver the intended freedom for my purpose chain thought → word → act.

“Who’s got the privilege to know, has the duty to act.”

— According to a 2021-09-08 **post** at thewisdomofpeople.com Jill S. Baron attached this quote, in a slightly different form, to Alberst Einstein in the Preface to *Rocky Mountain Futures: An Ecological Perspective* (Washington: Island Press, 2002)

No matter if Einstein said it, the duty to act depends on the words which communicate the thoughts into action. The idea of putting thoughts into words before acting on them is one essential step in human relations. It also *acts* as a buffer. So putting it into an EMACS buffer is definitely a good start.

13 Appendix

I just offer *Supplemental Material* for the carpenter’s Web site assimilation [20]. The files for the construction of the advanced rookie site are listed in Figure 4 and their purpose is illustrated in the *What’s More* Section 12.2.1. It’s because in the current state of transition the files contain too much outdated approaches.

Apart from the LINUX kernel the *System Information* of [20] is also the same in the long version. The carpenter supplements are still available at <https://bitbucket.org/StPjotr/shtm0>.

To get leads about my further steps I annotated Ogbe’s publishing process [17] with my interpretations.

13.1 Ogbe’s Website Construction

Ogbe uses XHTML for ORG export to HTML. In the *hidden* blog entry *My Emacs Configuration* [17] Ogbe offers structured insight into inspiring working space preparations; *hidden* because there’s only a link from an “official” blog entry available from his menu item `blog`; and it is linked from other hidden and `blog`-listed entries. This hidden entry spreads into numerous *Components*, i.e., also hidden entries, like *Org*, *Snippets*, *Completion*, or *References*. And it contains an *Exporting* Section which explains [17]’s publishing process. The Section *HTML Export* in the hidden *Org* component offers offline export tweaks for local ORG HTML exports.

So, the most central information for my purpose of reconstructing Ogbe’s construction work is in the *unhidden* blog post *Bloggng using org-mode* [16]. I figured some features to act as an ELISP crash course for me. And for myself I marked some which surpassed my rookie horizon. The folder structure for the publishing lists `blog-articles`, `blog-pages`, and `blog-rss` is summarized in Table 5. The different publishing actions are

- `org-html-publish-to-html` for articles and pages
- `org-rss-publish-to-rss` for RSS
- `org-publish-attachment` for resources, images, and downloads.

The site contents are articles, pages, and hidden entries; the resources are images, two CSS files, and plenty of download files. Most

of the download files are found in the *News Section* of the home page. The `alist` entry of whole blog's publishing components is

```
(setq org-publish-project-alist '(("blog" :components (
  "blog-articles", "blog-pages", "blog-rss",
  "blog-res", "blog-images", "blog-dl"))
```

Table 5 – The folders for articles and pages processed by `org-html-publish-to-html`; for RSS by `org-rss-publish-to-rss`. All base directories begin with $\ni \sim$ /repos/blog/. Yes, the blog articles' and RSS's source folder is $\ni \sim$ /repos/blog/blog/. All publishing directories begin with $\ni \sim$ /repos/blog/www/.

	articles	pages	rss
:base-directory	blog/	pages/	blog/
:publishing-directory	blog/	.	.
:base-extension	"org"	"org"	"org"
:exclude	–	–	"*"
:include	–	–	("blog.org")
:html-link-up	"	"	
:html-link-home	"/", ""	"/", ""	"https://ogbe.net"
:html-link-use-abs-url	–	–	t

Before getting into the article-page-sitemap production I shortly deal with the files which are copied only; their common `:publishing-function` is `org-publish-attachment`. The image and the download folder \ni `img/` and \ni `dl/` are copied recursively, the resources folder \ni `res/` employs `my-blog-minify-css`; see Table 6.

Table 6 – The folders `org-publish-attachment` files. All base directories begin with $\ni \sim$ /repos/blog/, all publishing directories with $\ni \sim$ /repos/blog/www/.

	res	img	dl
:base-directory	res/	img/	dl/
:publishing-directory	res/	img/	dl/
:base-extension	"*"	"*"	"*"
:completion-function	my-blog-minify-css		
:recursive	–	t	t

my-blog-minify-css restructures all CSS files, i.e., `code.css` and `main.css` at the `:publishing` directory, with

```
csstidy --template=highest --silent=true
```

The ELISP code is shown at length at the *How to Blog..* [16] paragraph beginning with “The next preprocessor runs `CSSTidy...`”

13.1.1 Publishing Action

The central RSS constructor is `org-rss-publish-to-rss` from the non-GNU `ox-rss.el`, while the articles and the pages are made by

```
:publishing-function org-html-publish-to-html
:htmlized-source t ;; this enables htmlize, which means that I can
↪use
;; css for code!
```

`:htmlized-source` is a switch for exporting original ORG files. The Section *Publishing Action* in the ORG Manual says:

“If you want to publish the original ORG file with the archived, commented and tag-excluded trees removed, use the function `org-org-publish-to-org`. This will produce `file.org` and put it in the publishing directory. If you want a htmlized version of this file, set the parameter `:htmlized-source` to `t`; this will produce `file.org.html` in the publishing directory.”

On the other hand enabling “htmlize, which means that I can use CSS for code” is controlled by `org-html-htmlize-output-type` and an integral part of every ORG HTML export procedure, which is also used to HTMLize an original ORG file by `org-org-publish-to-org` with `:htmlized-source` switched on. The choices for `org-html-htmlize-output-type` are

- `css` to export the CSS selectors only
- `inline-css` to export the CSS attribute values inline in the HTML as `<style>` elements
- `nil` to export plain text.

Both depend on the current buffer theme, which makes them useless²³ for batch mode exports and sensitive to collaborate editing. `org-html-htmlize-generate-css` produces a buffer with all current rich font definitions which we can use as a source for fixed class definitions. But the intention of “I can use CSS for code” still depends on the choice for `org-html-htmlize-output-type`.

13.1.2 Head

Define `my-blog-extra-head` as a string

```
(setq my-blog-extra-head
      (concat
        "<link rel='stylesheet' href='../res/code.css' />\n"
        "<link rel='stylesheet' href='../res/main.css' />"))
```

Both the `blog-articles` and the `blog-pages` publishing list contain

```
:html-link-home "/"           :html-viewport nil
:html-head nil :html-head-extra ,my-blog-extra-head
:html-head-include-default-style nil
:html-head-include-scripts nil
```

The `:html-link-home` property is set twice in Ogbe’s script; further down both `link up` and `home` are set to an empty string which means that `:html-link-up/home-format` isn’t included. AFAIK the latter gets preference but I can’t remember where I found that statement.

The usage of the two properties `:html-head` and `:html-head-extra:` is IMHO equivalent to

```
:html-head ,my-blog-extra-head
```

The property `:html-head-include-scripts` only regards the script template `org-html-scripts` containing code highlighting functions, i.e., “basic JAVASCRIPT that is needed by HTML files produced by `ORG MODE`.”²⁴

²³This is my interpretation of the `org-html-htmlize-output-type` doc-string. Not experimentally verified.

²⁴Manual’s [Section](#) *JavaScript supported display of web pages*.

Whatever that means, it doesn't apply to neither MATHJAX nor Sebastian Rose's `org-info.js` enabled by `org-html-use-infojs`; see the Manual's [Section](#) *JavaScript supported display of web pages* or the WORG [entry](#) *Emacs org-info.js*.

13.1.3 Mathjax

MathJax usually recommends to use their CDN to load their JAVASCRIPT code; ORG supports this feature by setting up the configuration template `org-html-mathjax-template`. This template is fed by `org-html-mathjax-options` or an in buffer setting like

```
#+HTML_MATHJAX: align: left indent: 5em tagside: left font: Neo-Euler
```

See the docstring of `org-html-mathjax-options` for detailed explanations of the keywords and or the customization buffer for accepted values. Ogbe's choice is to

- use a local version and the configuration `TeX-AMS-MML_HTMLorMML` instead of the ORG default `TeX-AMS_HTML`; see [Combined Configurations](#) at docs.mathjax.org

```
(setq my-blog-local-mathjax
      '((path
         ↪"/res/mj/MathJax.js?config=TeX-AMS-MML_HTMLorMML")
        (scale "100") (align "center") (indent "2em") (tagside
         ↪"right")
        (mathml nil)))
```

- and to feed it into his own template defined literally in `blog-articles` and `blog-pages` publishing properties

```
:html-mathjax-options ,my-blog-local-mathjax
:html-mathjax-template "<script type=\"text/javascript\"
↪src=\"%PATH\"></script>"
```

13.1.4 Pre- and Postamble → Header and Footer

Set `my-blog-header-file` and define the function `my-blog-header` to insert this file into the current buffer

```
(setq my-blog-header-file "~/repos/blog/header.html")
(defun my-blog-header (arg)
  (with-temp-buffer
    (insert-file-contents my-blog-header-file)
    (buffer-string)))
```

According to the source code of the online files the content of `my-blog-header-file` ↪ `header.html` at ↪ `~/repos/blog/` is probably

```
<div id="preamble" class="status">
  <header><div class="banner">
    <a id="myname" href="/">Dennis Ogbe </a><hr>
    <nav><p><a href="/">Home</a>
    <a href="/about.html">About</a>
    <a href="/research.html">Research</a>
    <a href="/blog.html">Blog</a>
  </p></nav></div></header></div>
```

The footer contains the link to the RSS file produced with the `blog-rss` publishing list and to a (missing) contact page. It's a string set with

```
(setq my-blog-footer "<hr />\n<p><span style=\"float: left;\"><a
↪href= \" /blog.xml\">RSS</a></span>License: <a href=
↪\"https://creativecommons.org/licenses/by-sa/4.0/\">CC BY-SA
↪4.0</a></p>\n<p><a href= \" /contact.html\"> Contact</a></p>\n")
```

In the publishing projects `blog-articles` and `blog-pages` the function and the string are included with the project properties

```
:html-preamble my-blog-header
:html-postamble ,my-blog-footer
```

So the preamble, i.e., header, is named `my-blog-header` and gets a `<header>` element, while the postamble, i.e., footer, is named `my-blog-footer` and goes without the privilege of a `<footer>` element; see the `<header>` and the *ARIA: banner role* entries at developer.mozilla.org about the intentions of these elements.

13.1.5 Home Up

The `html-link-home` features are explained in the export script `ox-html.e1` only. Unless `:html-link-up` and `:html-link-home` are *both* "" the first entry after the `<body>` element is created from `:html-home/up-format`; see the corresponding doc-string.

```
<div id="org-div-home-and-up">
  <a accesskey="h" href="%s"> UP </a> |
  <a accesskey="H" href="%s"> HOME </a>
</div>
```

Both the `blog-articles` and the `blog-pages` publishing list contain the confusing settings – because the `link-home` is defined twice

```
:html-link-home "/" :html-home/up-format ""
:html-link-up "" :html-link-home ""
```

while the `blog-rss` list defines

```
:html-link-home "https://ogbe.net/" :html-link-use-abs-url t
```

with an activated `..-use-abs-url` setting which prepends relative links with the content of the file based `#+HTML_LINK_HOME:` or the publishing property `:html-link-home`.

13.1.6 Footnote

According to the doc-string of `org-html-footnotes-section` the footnote section defaults to

```
<div id="footnotes">
  <h2 class="footnotes">%s: </h2>
  <div id="text-footnotes">
    %s
  </div>
</div>
```

It should contain a two instances of `%s`.

- The first will be replaced with the language-specific word for "Footnotes",

- the second one will be replaced by the footnotes themselves.

Other settings are about

- the `org-html-footnote-format` which defaults to `^{%s}`, where `%s` will be replaced by the footnote reference itself
- and the `org-html-footnote-separator` set to `^{`, `}`

Ogbe removes the language dependent part by defining `:html-footnotes-section` as

```
<div id='footnotes'><!--%s-->%s</div>
```

He also marks his superscript setting as *important* without further explanation. This might be related to the footnotes.

```
:with-sub-superscript nil ;; important!!
```

But `nil` just means *not* to interpret the underscore `"_"` and the caret `"^"` for export; see the doc-string of `org-export-with-sub-superscripts`.

13.1.7 Pre- and Post-Processors

Ogbe says that his pre- and post-processors are used to move around some files before and after publishing.

Both `my-blog-pages-postprocessor` and `my-blog-articles-preprocessor` are message dummies. The code is shown at length at the *How to Blog..* [16] paragraph beginning with “Next I define some pre- and postprocessors...”

- `my-blog-pages-preprocessor` moves a fresh version of the `⊗ settings.org` file to the pages directory. As I understand the code it copies the current `⊗ settings.org` as `my-blog-emacs-config-name`, which is set to `⊗ emacsconfig.org`, to `destdir`, containing the value of `:base-directory`.
- `my-blog-articles-postprocessor`: “massage” the sitemap file and move it up one directory. The “massage”-part is a comment further down but not filled with a procedure: “massage the sitemap if wanted”.

So, the whole function block seems only being concerned about moving pages up. The inclusion at the `blog-articles` and `blog-pages` publishing properties are four lines syntactically summarized to

```
:preparation-function my-blog-[articles|pages]-preprocessor
:completion-function my-blog-[articles|pages]-postprocessor
```

13.1.8 Options, Drawer

The *options for exporters* Ogbe chooses to extract to his publishing configuration are discussed in Section 7. Here I focus on the two settings regarding ORG MODE *drawers*.

```
:with-author t           :with-creator nil
:with-date t             :headline-level 4
:section-numbers nil     :with-toc nil
:with-sub-superscript nil :with-drawers t
:html-format-drawer-function my-blog-org-export-format-drawer
```

`:with-drawers t` enables the export of ORG MODE *drawers* and customizes the `org-export-format-drawer-function` with the publishing property in the last line. Ogbe cites Panchekha [19] for this idea. According to Panchenka's *Drawer Section* Ogbe seems to dislike ORG MODE's drawer `export-to-code-block` default. The redefinition line at the end of Panchekha's code example is equivalent to the publishing property `:html-format-drawer-function` above. And the corresponding `<div>` section generator is given as

```
(defun my-blog-org-export-format-drawer (name content)
  (concat "<div class=\"drawer \" (downcase name) \">\n"
    "<h6>" (capitalize name) "</h6>\n"
    content
    "\n</div>"))
```

13.1.9 Sitemap

The customizations that Ogbe “bolted” on ORG's publishing features exceed my debugging skills. He built his own sitemap creator added as project property `:sitemap-function`

my-blog-sitemap in the publishing list `blog-articles` written into `:sitemap-filename "blog.org"`. The function is also a part of `my-blog-articles-postprocessor` which is employed as `:completion-function` for both the `blog-articles` and the `blog-pages` publishing list.²⁵ The whole sitemap producing part of the `blog-articles` publishing list is

```
;; sitemap - list of blog articles
:auto-sitemap t
:sitemap-filename "blog.org"
:sitemap-title "Blog"
;; custom sitemap generator function
:sitemap-function my-blog-sitemap
:sitemap-sort-files anti-chronologically
:sitemap-date-format "Published: %a %b %d %Y"
```

The purpose of the `my-blog-sitemap` specification is to include a part of a blog entry as teaser in the his sitemap construction. Ogbe encloses the “preview” part of the post in `#+BEGIN_PREVIEW...#+END_PREVIEW` tags, which his “(very simple) parser” then inserts into the sitemap page. The sitemap production is connected to the publishing list `blog-article`. According to this description the expanded sitemap production \ni `blog.html` is the central blog entry distributor for what I called *unhidden* files.

13.1.10 RSS

The RSS settings are addressed in a definition

```
(require 'ox-html)
(require 'ox-rss)
(setq org-export-html-coding-system 'utf-8-unix)
(setq org-html-viewport nil)
```

and a publishing section

```
("blog-rss"
 :base-directory "~/repos/blog/blog/"
 :base-extension "org")
```

²⁵In his Website’s *About*, Section Update 2021-10-14, he mentions that his function depends on ORG version 9.0 for his specific use of the signature of the `:sitemap-function`. So it should take some effort to reproduce the procedure.

```

:publishing-directory "~/repos/blog/www/"
:publishing-function org-rss-publish-to-rss

:html-link-home "https://ogbe.net/"
:html-link-use-abs-url t

:title "Dennis Ogbe"
:rss-image-url "https://ogbe.loc/img/feed-icon-28x28.png"
:section-numbers nil
:exclude ".*"
:include ("blog.org")
:table-of-contents nil)

```

They fill the `blog.org` file which is `org-rss-publish-to-rss` published to `blog.xml`. With these informations and the `ox-rss.el` script I'm sure to get some ideas about the RSS export features.

References

- [1] Tim Berners-Lee and Mark Fischetti. *Weaving the Web*. HarperCollins, 2000.
- [2] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media, 2009.
- [3] Debra Cameron, James Elliott, Marc Loy, Eric Raymond, and Bill Rosenblatt. *Learning GNU Emacs*. O'Reilly Media, 3rd edition, 2004.
- [4] Steve Faulkner. On html belts and aria braces. URL, 2015. updated 2020-08-03.
- [5] Yuan Fu. Blog in org mode revisited. URL, 2018.
- [6] Yuan Fu. Blog with only org mode. URL, 2018.
- [7] Joshua Gay, editor. *Free Software, Free Society*. GNU Press, Free Software Foundation, 2015.
- [8] Glucksman Library. *Cite it Right*. University of Limerick's Referencing Series. Glucksman Library, 2nd edition, 2008.
- [9] Kurt Hornik, Duncan Murdoch, and Achim Zeileis. Who did what? *The R Journal*, 4(1):64-69, June 2012.
- [10] Xah Lee. Emacs lisp: Call shell command. url, 2018. updated 2021-06-21.

- [11] Eric A. Meyer and Estelle Weyl. *CSS The Definitive Guide*. o'Reilly, 2017.
- [12] Frank Mittelbach, Michel Goossens, Johannes L. Braams, David P. Carlisle, and Chris A. Rowley. *The LaTeX Companion 2. Tools and Techniques for Computer Typesetting*. Addison-Wesley Professional, Reading, Massachusetts, second edition, April 2004.
- [13] Peter Morville and Louis Rosenfeld. *Information Architecture for the World Wide Web*. O'Reilly, Beijing u.a., 3. ed. edition, 2006.
- [14] Simon Munzert, Christian Rubba, Peter Meißner, and Dominic Nyhuis. *Automated Data Collection with R*. Wiley John + Sons, 2014.
- [15] Deborah Nolan and Duncan Temple Lang. *XML and Web Technologies for Data Sciences with R. Use R!* Springer New York, 2014.
- [16] Dennis Ogbe. Blogging using exclusively org-mode. URL, 2016.
- [17] Dennis Ogbe. My emacs configuration. URL, 2020.
- [18] Duncan Mac-Vicar P. Migrating from jekyll to org-mode and github actions. URL, 2019.
- [19] Pavel Panchekha. Using org-mode to publish a web site. URL, 2011.
- [20] pjs64. Org 2 shtml0 eazy. URL, 2022.
- [21] Sebastian Rose. Publishing treemenu for org-files. URL.
- [22] Sebastian Rose. Publishing org-mode files to html. URL, 2020.
- [23] Tessa Blakely Silver. *Joomla! Template Design*. packt publishing, Birmingham, 2007.
- [24] David Wood, Marsha Zaidman, Luke Ruth, and Michael Hausenblas. *Linked Data*. Manning, 2014.