Building a Website with Make4ht: Building a Build System

David Friant

June 20th, 2026

Abstract

While all previous articles have been about the satisfactory compilation of individual documents, this article is dedicated to the description of the greater build process which allows for the entire website to be built. This includes discussion on the input file describing the website's directory tree, the creation of the navigation menu, and the compilation of each type of page on the website: home, directory and subdirectory, article, contact, and search. Much detail is included in the discussion of the latter two page types as they are wholly unique in the scheme of the website. This is tentatively the last article in this project, depending on future needs.

Key Words: LaTeX, Make4ht, Build System, Search, Contact, IndieWeb

Introduction

As indicated at the end of the previous article, this article discusses the tooling developed around the Make4ht system such that a full website can be generated from a series of fully-independent LaTeX source documents (and associated images, code snippets, etc). Prior to that, it should be noted that Make4ht possesses the ability to produce multiple HTML pages from a single source document[1]. However, that would entail including all content in a singular[A] LaTeX document. The goal for this website is instead to separate each article into an individually compiled document. As will be seen, this allows the website to be updated piecemeal instead of requiring a full recompilation of every document and, thus, webpage.

The overall build system is written in Python[6] just as with the post-processing steps discussed in the previous articles. This was chosen largely out of convenience as Python is already being heavily used in the project, and mixing in another language would only further complicate things. With that stated, Python is in many ways a convenient choice for the build system due to its well-established and documented libraries for the likes of walking a directory tree, calling sub-processes, hashing files, text search and replacement, etc.

The next section describes the general overall process of the build system. The section after describes the handling for some specific page types: the home page, directories, articles, and the contact and search pages.

Generalities

The primary input for the build system is a JSON file which describes the directory tree of the website. A reduced example of this file is provided by Listing 1. This provides all the information required for walking the real directory tree, building the navigation menu, temporally ordering the pages, and more. While some of the information may seem redundant (e.g. the path, file, title, ID, and download fields), it was a conscience choice to separate these for flexibility in naming things as the website grows larger over months and years of additions. The meaning and use of each field of the JSON entries are provided:

- **Type:** Defines the processing to apply to the webpage. This can vary widely from simply building a list of articles and/or subdirectories to fully compiling and processing a LaTeX document into an HTML document.
- **Path:** Recursively appended to from the root directory to uniquely address a webpage's source directory. These are defined manually such that work in progress articles can exist within the normal website directory without being compiled.
- **File:** Provides the name of the LaTeX and/or HTML source document to process for the build process.

[A] Of course, the LaTeX document itself could be broken up into many sub-documents.

This is not guaranteed to to be the same as the Path field, but they are often very similar.

- **Download:** Provides the name of the file to request when the user clicks the "Download" button in the navigation menu header. While this will often be the PDF file compiled from the LaTeX source, in some cases it may be a compressed file including code or even nothing at all. This provides flexibility for providing an arbitrary file as desired for download.
- **ID:** This is a unique identifier for each webpage. It is used in the build process but has no other meaning.
- **Date:** This is the canonical date of the webpage's publishing. Any and all temporal sorting will be based upon this value.
- **Children:** Webpages which represent directories that contain other directories and/or articles are required to have a "Children" field such that the entire tree structure of the website can be built recursively from the JSON file.

Listing 1: A snippet of the JSON file describing the layout of the website to be provided to the build script.

```
{
        "Type":"Main",
        "Path":".",
        "File":"Main.html",
        "Download": null,
        "ID":"main",
        "Date": "2026-01-15",
        "Children":[
        {
                "Type":"Directory",
                "Path":"Personal_Projects",
                "File":"Personal_Projects.html",
                "Download": null,
                "Title":"Personal Projects",
                "ID":"personal_projects",
                "Date": "2026-01-22",
                "Children":[
                {
                        "Type":"Subdirectory",
                        "Path":"This_Website",
                        "File":"This_Website.html",
                        "Download": null,
                        "Title":"Creating This Website",
                        "ID":"this_website",
                        "Date": "2026-01-22",
                        "Children":[
                        {
                                "Type":"Article",
                                "Path":"00_Overview",
                                "File":"Overview.html",
                                "Download": "Overview.pdf",
                                "Title":"Overview",
                                "ID":"overview",
                                "Date": "2026-01-22",
                        },
                        {
                                "Type":"Article",
                                "Path":"01_Basic_Typesetting",
                                "File":"Basic_Typesetting.html",
                                "Download": "Basic_Typesetting.pdf",
                                "Title":"Basic Typesetting",
                                "ID":"basic_typesetting",
                                "Date": "2026-01-24",
                        }
```

```
                        ]
                },
                {
                        "Type":"Subdirectory",
                        "Path":"Operator_Language",
                        "File":"Operator_Language.html",
                        "Download": null,
                        "Title":"Operator Language",
                        "ID":"operator_language",
                        "Date": "2026-05-29",
                        "Children":[
                                {
                                        "Type":"Article",
                                        "Path":"00_Design_Goals_and_Intent",
                                        "File":"Design_Goals_and_Intent.html",
                                        "Download": "Design_Goals_and_Intent.pdf",
                                        "Title":"Design Goals and Intent",
                                        "ID":"operator_design_goals",
                                        "Date":"2026-05-29",
                                }
                        ]
                }
                ]
        }
        ]
}
```

In broad strokes, the build process is as follows. First read in the `build.json` file as a singular object using Python's JSON library[8]. Next, recursively walk through the tree structure of the JSON file object and build all components of the navigation menu that are shared across all webpages, e.g. the most of the top row of navigation icons and all the nested drop-down menus. The JSON file object and the newly-constructed navigation menu object are then passed to the function which recursively builds each webpage. Some details for the building of each type of currently supported page type are discussed in the sections below.

Page Types

After the build and navigation objects are built, the proper build process begins by passing them to a function which first recursively calls itself on the children of the given build tree node. This ensures that data from child pages will be available to their parent, e.g. properly formatted abstracts. The function then simply selects the correct build function for the current node type. Listing 2 provides the full code for this.

Listing 2: The main build function which handles the correct build order (depth first) and type based on the switch statement.

```
#Build all webpages
def build(buildObj, navObj, path):
    log(trace, "build/build.py", "build(buildObj, navObj, path)")
    global returnCode

    for child in buildObj["Children"]:
        build(child, navObj, path + buildObj["Path"] + "/")
        if returnCode != 0:
            return
        #
    #
    match buildObj["Type"]:
        case "Main":
            log(info, sys.argv[0], "Build: Main")
```

```
                buildMain(buildObj, navObj, path)
            #
            case "Directory":
                log(info, sys.argv[0],\
                    f"Build: Directory: {path + buildObj["Path"]}")
                buildDirectory(buildObj, navObj, path)
            #
            case "Subdirectory":
                log(info, sys.argv[0],\
                    f"Build: Subdirectory: {path + buildObj["Path"]}")
                buildSubdirectory(buildObj, navObj, path)
            #
            case "Article":
                log(info, sys.argv[0],\
                    f"Build: Article: {path + buildObj["Path"]}")
                buildArticle(buildObj, navObj, path)
            #
            case "Contact":
                log(info, sys.argv[0],\
                    f"Build: Contact: {path + buildObj["Path"]}")
                buildContact(navObj)
            #
            case "Search":
                log(info, sys.argv[0],\
                    f"Build: Search: {path + buildObj["Path"]}")
                buildSearch(buildObj, navObj)
            #
            case _:
                log(error, sys.argv[0],\
                    f"Unhandled webpage type {path + buildObj["Path"]}")
                returnCode = 1
                return
            #
        #
    #
```

Home Page

With the primary build function defined as it is, the home (or main) page of the website is the last to be built. This is convenient as it is designed[B] to present of short list of just the most recent articles added to the website. This is done by simply sorting a list of all the articles and adding the links, abstracts, keywords, and publishing date of the most recent ones to a template HTML file which defines all of the content for the home page which is not subject to regular change, e.g. the "About" blurb.

Directories and Subdirectories

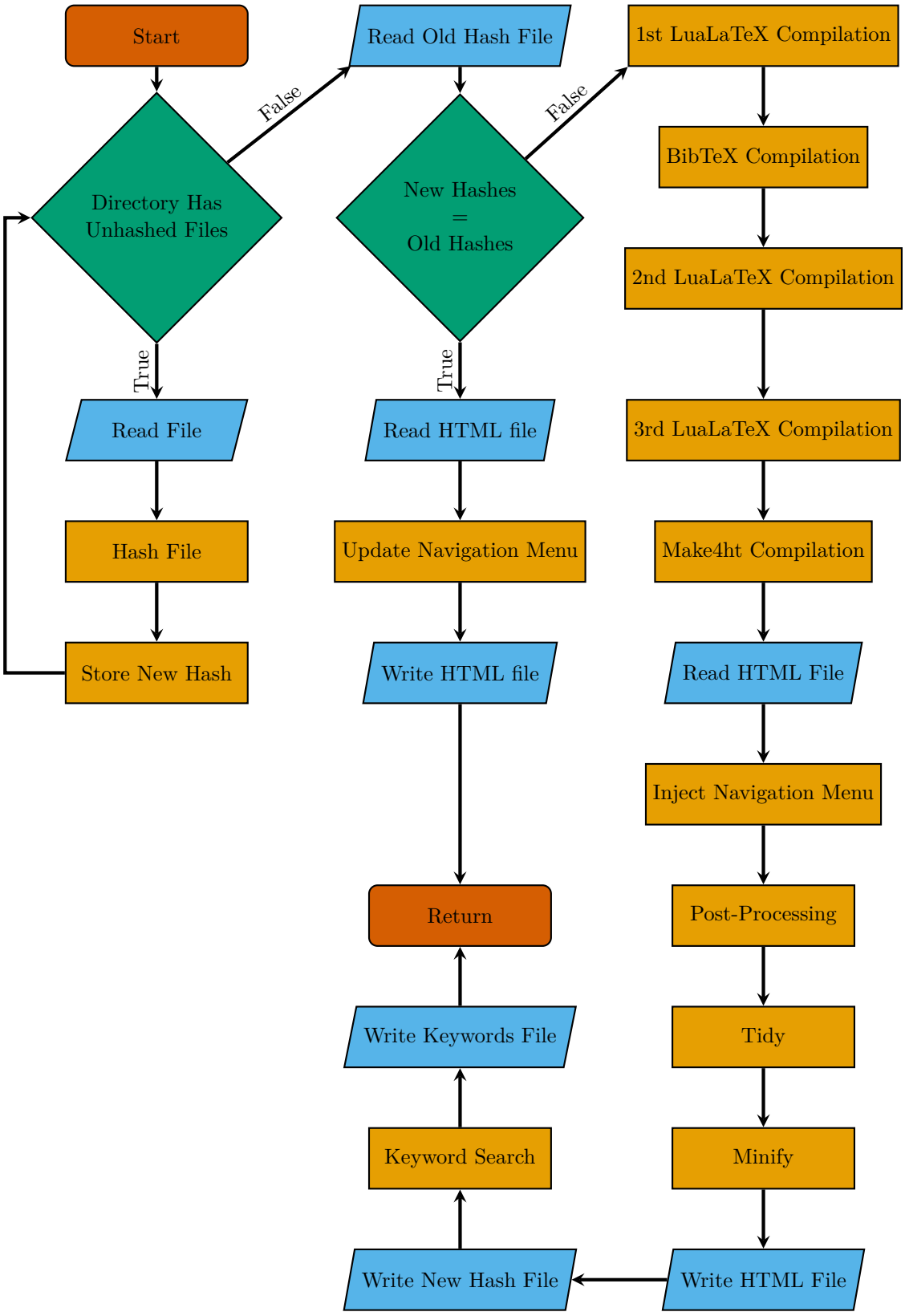
Directories and subdirectories are probably the easiest pages to build. Each directory contains a template HTML file with most of the webpage predefined. The function simply takes that template and adds a directory entry for each child page. This includes the title, abstract/about blurb, keywords, and date of publishing. In all, this is ultimately a simple exercise in copying the correct text from the child pages, formatting it a little, and pasting it into the template.

Articles

Articles compose the bulk of the website and are by far the most difficult to build due to the toolchain required for them as has been discussed in all the previous articles in this project. Figure 1 presents a flowchart of the build process for articles.

[B] As of the time of writing.

Figure 1: A flow chart describing the build process of articles for the website.



As can be observed from Figure 1, The toolchain for building the articles for this website is rather long, however each link in the chain exists for a specific and explicit reason. The initial hashing[10, 7] of each file in the article's directory allows for a comparison to the hash values from the previous compilation[C]. If the values are unchanged, then there is no need to recompile the document. If there has been a change in any of the files used to generate the PDF and HTML documents, then they are recompiled using the full toolchain that

[C] Assuming that a previous compilation has occurred, of course.

has been developed over the course of this project. This is achieved via Python's Subprocess library[9]. The final step, the keyword search will be expanded up in the section dedicated to the search page below.

The overall process is thus designed to minimize the amount of work that the CPU must do as the compilation of LaTeX documents (especially those containing TIKZ/PGFPlots content) may take a considerable amount of time. If a full recompilation of the website is required, all that is necessary is to delete the `md5.json` file in each directory which stores the most recent hash values then rebuild. This can be done with a simple script.

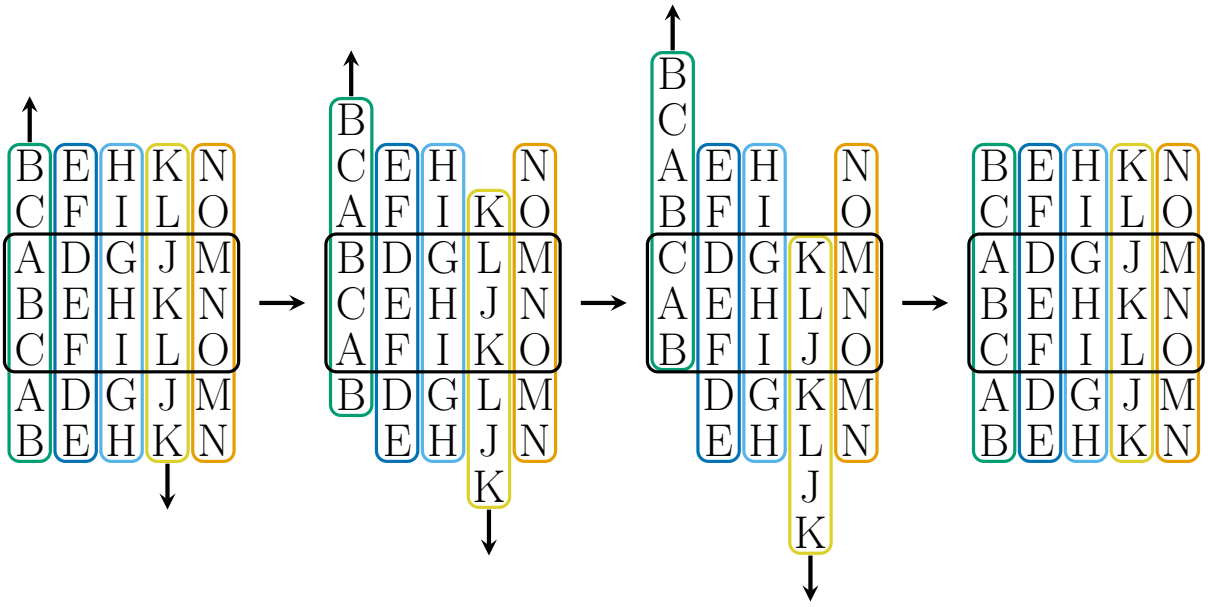
Contact Page

The contact page was designed with a singular goal in mind: allow a real human to read the contact address for this website while largely preventing webscrapers, bots, and AI agents from trivially doing the same. This naturally excluded simply writing the contact address in plain text, or even enciphering it within some text that gets deciphered when the page is opened. An alternative could be to include the address as an image such that it is plainly visible to a human but opaque to a simple bot. However, the rise of AI-driven text recognition renders this also potentially insufficient. As such, the choice was made to encode the contact address in a simple puzzle which should be sufficient to evade most automated extraction techniques.

This puzzle is a simple sliding-text problem where rings of text may be moved by clicking and dragging them to visually align a colored line and thus reveal the hidden contact address. The image itself is a hand-tooled[D] SVG[5] with each column of sliding characters organized into groups such that they may be transformed (translated vertically) uniformly by addressing the group.

The illusion of the rotating "ring" of characters is achieved by drawing duplicate sequences of characters in each ring above and below the midpoint of the character sequence and outside of viewport of the SVG. When a group of characters is moved too far up or down, it is transformed down or up, respectively, such that the visual appearance does not change. Figure 2 provides and illustration of the process.

Figure 2: A visual description of the workings of the Contact page puzzle. The uncolored rectangle represents the viewport of the SVG while the colored rectangles represent each group of characters.



As can be seen from Figure 2, the visual effect achieved is that of a continuous ring of characters by simply translating the groups back to the starting position if they are moved too far up or down. The implementation of this logic in JavaScript[4] can be found by using your browser's development tools on the Contact page. The actual implementation also includes a small "force" to each group such that they align themselves to the nearest common character baseline. This not strictly necessary, but it does improve the look and feel of the puzzle by ensuring some level of alignment in the characters.

Search Page

Before discussing the Search page itself, it is necessary to first describe the process of extracting keywords from the articles and directories. For articles, the keyword search occurs after all other build processes to ensure that only text that is actually in the document is included in the search and that nothing is unintention-

[D] As of the time of writing. This may change in the future if it becomes necessary to update the puzzle.

ally left out. For directories, the search is performed after the addition of the directory entries, resulting in the abstracts and keywords of their child documents being included in the search. This is not undesirable as it means the directory is more likely to come up during the search, though likely after the articles themselves.

The search algorithm is relatively simple. Using the same HTML Parser base class as was used in the post-processing, the class is extended as shown in Listing 3. As can be seen, this really only routes the processing of the text into three subfunctions: `processStartTag(tag, attrs)`, `processEndTag(tag)`, and `processData(data)`.

Listing 3: The definition for the HTML parser used in the extraction of keywords from the HTML files.

```
# Parser for the HTML file
class GeneralHTMLParser(HTMLParser):
    global programOut

    def handle_starttag(self, tag, attrs):
        log(trace, "KeyWordSearch.py","handle_starttag(self, tag, attrs)")
        processStartTag(tag, attrs)
        pass
    #
    def handle_startendtag(self, tag, attrs):
        log(trace, "KeyWordSearch.py","handle_startendtag(self, tag, attrs)")
        pass
    #
    def handle_endtag(self, tag):
        log(trace, "KeyWordSearch.py","handle_endtag(self, tag)")
        processEndTag(tag)
        pass
    #
    def handle_data(self, data):
        log(trace, "KeyWordSearch.py","handle_data(self, data)")
        processData(data)
        pass
    #
    def handle_entityref(self, name):
        log(trace, "KeyWordSearch.py","handle_entityref(self, name)")
        pass
    #
    def handle_charref(self, name):
        log(trace, "KeyWordSearch.py","handle_charref(self, name)")
        pass
    #
    def handle_comment(self, data):
        log(trace, "KeyWordSearch.py","handle_comment(self, data)")
        pass
    #
    def handle_decl(self, decl):
        log(trace, "KeyWordSearch.py","handle_decl(self, decl)")
        pass
    #
    def handle_pi(self, data):
        log(trace, "KeyWordSearch.py","handle_pi(self, data)")
        pass
    #
    def unknown_decl(self, data):
        log(trace, "KeyWordSearch.py","unknown_decl(self, data)")
        log(error, "KeywordSearch.py","Unknown HTML declaration: " + data)
        programOut = 0
        return
```

```
    #
#
```

The functions which handle the start and end tags are primarily responsible for ensuring that certain parts of the document are not included in the search. For example, text contained in <script>, <nav>, <table>, <pre>, <svg>,, <ol class="references">, and <ol class="footnotes"> tags are all skipped as of the time of writing. This is to avoid including things such that are common to all documents, such as the navigation menu, and things which are not useful to search in such as code blocks and data tables.

To track which mode the algorithm is running in (skip vs normal) a stack[12] is kept of the tags that are being skipped. Elements are added to the stack when they are encountered in the parsing and removed when a closing tag is found that matches the current top of the stack. Note that this does require strict adherence to the XML schema which isn't strictly necessary for the display of webpages.

When there are no elements on the stack, then the algorithm runs in "normal" mode wherein text is broken by whitespace and hyphens, transformed entirely to lower case, and stripped of all leading and trailing non-alphabetic characters. The resulting strings are added to a multiset[11]. Listing 4 provides the function definition.

Listing 4: The code for extracting words from HTML data and adding them to the multiset for storage and future processing.

```
# Process HTML data and fill multiset of words
def processData(data):
    log(trace, "KeyWordSearch.py","processData(data)")
    global programOut
    global multiset
    global mode

    if mode == "skip":
        return
    #

    words = re.split(r"(\s|-)", data)
    for word in words:
        word = word.lower()
        matches = re.findall(r"[^a-z]*([a-z]+)[^a-z]*", word)
        for match in matches:
            if match in multiset:
                val = multiset[match]
                multiset[match] = val + 1
            else:
                multiset[match] = 1
            #
        #
    #
    return
#
```

While it would be acceptable to write out the multiset of words to a keywords file at this point, it would be suboptimal as the multiset is certainly both riddled with common words and full of near-duplicates where words are different only by a suffix. Listing 5 provides an imperfect but sufficient solution to this problem. The `processMultiset()` function should catch most regular variations of English words, remove the suffixed version, and increment the base version by the suffixed version's count.

Listing 5: The code for processing the extracted keywords so as to minimize the number of useless and/or irrelevant words included in the keywords file.

```
# Remove extraneous and meaningless words from the multiset
def processMultiset():
    log(trace, "KeywordSearch.py","processMultiset()")
    global programOut
    global multiset
    global toRemove

    remList = []
    for word in multiset:
        if word + "s" in multiset:
            val = multiset[word + "s"]
            multiset[word] = multiset[word] + val
            remList.append(word + "s")
        #
        if word + "es" in multiset:
            val = multiset[word + "es"]
            multiset[word] = multiset[word] + val
            remList.append(word + "es")
        #
        if word + "d" in multiset:
            val = multiset[word + "d"]
            multiset[word] = multiset[word] + val
            remList.append(word + "d")
        #
        if word + "ed" in multiset:
            val = multiset[word + "ed"]
            multiset[word] = multiset[word] + val
            remList.append(word + "ed")
        #
        if word + "ing" in multiset:
            val = multiset[word + "ing"]
            multiset[word] = multiset[word] + val
            remList.append(word + "ing")
        #
        if word + "ness" in multiset:
            val = multiset[word + "ness"]
            multiset[word] = multiset[word] + val
            remList.append(word + "ness")
        #
        modWord = word[0:-1]
        if modWord + "ing" in multiset:
            val = multiset[modWord + "ing"]
            multiset[word] = multiset[word] + val
            remList.append(modWord + "ing")
        #
        if modWord + "ion" in multiset:
            val = multiset[modWord + "ion"]
            multiset[word] = multiset[word] + val
            remList.append(modWord + "ion")
        #
    #
    for word in remList:
        if word in multiset:
            del multiset[word]
        #
    #
```

```
    for word in toRemove:
        if word in multiset:
            del multiset[word]
        #
    #
#
```

The `remList` used in Listing 5 is a simple list of many of the most commonly used words in the English language ('a', 'and' 'the', 'he', 'she', 'it', 'to', etc) slightly augmented with a few words which were noted to be common to most documents on the website.

Together, these processes reduce the amount of words in the multiset for a typical article on this website to contain of a few hundred words. This was deemed acceptable enough that the resulting multiset could be written to a JSON file for consideration when building the Search page.

The building of the Search page itself is fairly straightforward. In a fashion similar to that of the directory and subdirectory type pages, entries are added for each article and directory on the website, however, these are treated as flexbox[3] elements, allowing them to be reordered by altering their CSS `order` value dynamically. The manner in which to do so will be described after a brief digression.

While building the Search page, another document is produced: the master `keywords.json` file which associates each webpage with its keywords such that every word weighted by their rarity across the entire website. The process of doing so is actually quite simple: first create a multiset of all keywords across the entire website from the keywords file for each individual page, then simply divide the count associated with each word in each file by the sum of that word across the website. This gives words which are common across many pages a low weight while words that are rare or unique a higher weight. This weighting factor will become useful when computing each page's score.

The master keywords file is loaded when the page is initialized via JavaScript's Fetch API[2]. The keyword search is then performed on the text in the search bar when the user either hits enter or clicks the 'Search' button. Listing 6 provides the definition of the search algorithm.

Listing 6: The JavaScript code for performing the keyword search on the website. Note that this is performed client-side.

```
//Perform the keyword search
function performSearch(){
        console.debug("[TRACE] performSearch()")

        //Get list of words to search for. Remove capitals, spaces, and punctuation.
        var searchText = document.getElementById("searchtext").value;
        if(searchText === ""){
                return;
        }
        searchText = searchText.toLowerCase();
        searchText = searchText.replace(/[^0-9a-z]/gi, ' ');
        var searchWords = searchText.split(" ");

        //Calculate score for each element and set flex order
        for(var i = 0; i < keywordsObj.length; i++){
                var score = 0;

                var id = keywordsObj[i][0];
                var words = keywordsObj[i][1];
                for(var j = 0; j < searchWords.length; j++){
                        var tempScore = 0;
                        for(var key in words){
                                if(searchWords[j] == key){
                                        tempScore = Number(words[key]);
                                        break
                                }
                                else{
```

```
                                        len = Math.min(key.length, searchWords[j].length);
                                        var n = 0
                                        for(var k = 0; k < len; k++){
                                                if(key[k] == searchWords[j][k]){
                                                        n++;
                                                }
                                                else{
                                                        break;
                                                }
                                        }
                                        var val = 0;
                                        if(n > Math.max(key.length, searchWords[j].length / 3)){
                                                val = Number(words[key]) * n /
                                                        Math.max(key.length, searchWords[j].length);
                                        }
                                        if(val > tempScore){
                                                tempScore = val;
                                        }
                                }
                        }
                        score = score + tempScore;
                }
                score = Math.round(10000 * score); //10000 is an arbitrary multiplier
                document.getElementById(id).style.order = score;
                if(score < 200){ // 200 is an arbitrary cutoff
                        document.getElementById(id).style.visibility = "collapse";
                }
                else{
                        document.getElementById(id).style.visibility = "visible";
                }
        }

        return;
}
```

The algorithm begins by transforming the text to all lower case, just as was done for the keywords in the documents themselves. The search text is then split by whitespace and the calculated score for each word summed for each document. This calculated value is then assigned as the CSS `order` value for the flexbox element corresponding to each page, and the browser's drawing engine handles the reordering automatically. Elements below a certain, arbitrary score are hidden so as to attempt to avoid false positives.

The scoring algorithm itself could use a bit of discussion. For each word, it is first checked if there is an exact match in the keywords set. If yes, the weight value for the word is added to the total score. If not, the algorithm attempts to calculate how much overlap there is between the search word and each word in the keywords set. The weight value for each keyword is further weighted by the amount of overlap between the search word and the keyword. These partial weights are all summed and then added to the total score.

Overall, the keyword search is certainly lacking in polish and refinement, but it does seem functional enough for the website as it stands right now. It is possible that performance issues may crop up in the future when there are many more webpages than at the time of writing, but that will be a problem if and when it becomes one.

The End (For Now)

This concludes the articles currently envisioned for this project. It has taken substantially longer than first imagined to get the website as a whole functioning, but that is due in no small part to it being an educational project as much as a functional one. The process of learning the necessary components of LaTeX, Python, HTML, CSS, and JavaScript has been fruitful in many ways as it has broadened my horizons well beyond my usual work in C++.

As for future articles, it was mentioned in the first article of this project that there are some features that would be nice to have, but aren't strictly critical to the "completion" of the website's build framework. Namely, it would be desirable to have GIFs, audio, and even perhaps video at some point. This will be explored if and when those media formats become relevant to a project.

References

[1] Make4ht: Tex4ht options. https://www.kodymirus.cz/tex4ht-doc/texfourhtOptions.html#list-of-options. Accessed: 2026-06-06.

[2] Mozilla webdocs: Fetch api. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch. Accessed: 2026-06-20.

[3] Mozilla webdocs: Flexbox. https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/CSS_layout/Flexbox. Accessed: 2026-06-20.

[4] Mozilla webdocs: Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 2026-06-20.

[5] Mozilla webdocs: Svg. https://developer.mozilla.org/en-US/docs/Web/SVG. Accessed: 2026-06-20.

[6] Python. https://www.python.org/. Accessed: 2026-06-09.

[7] Python docs: Hashlib. https://docs.python.org/3/library/hashlib.html. Accessed: 2026-06-18.

[8] Python docs: Json. https://docs.python.org/3/library/json.html. Accessed: 2026-06-15.

[9] Python: Subprocess. https://docs.python.org/3/library/subprocess.html. Accessed: 2026-06-20.

[10] Wikipedia: Md5. https://en.wikipedia.org/wiki/MD5. Accessed: 2026-06-18.

[11] Wikipedia: Multiset. https://en.wikipedia.org/wiki/Multiset. Accessed: 2026-06-20.

[12] Wikipedia: Stack. https://en.wikipedia.org/wiki/Stack_(abstract_data_type). Accessed: 2026-06-20.