Building a Website with Make4ht: References and Footnotes

David Friant

February 1st, 2026

Abstract

References and footnotes are required for any sufficently complex body of text. While the implementation of these in LaTeX is straightforward, one must contend with the fundamental differrences between paginated documents and continuous ones for the translation to HTML. This article introduces substantial post-processing to the current system as it is, if not necessary, the easier way to resolve this conflict. Indeed, this allows one begin leveraging the power of a dynamic display by implementing tooltip-like behavior for both references and footnotes. This should create a smoother experience as readers will not be required to either jump to the bottom of the document or otherwise break their stride.

Key Words: LaTeX, Make4ht, References, Footnotes, IndieWeb

Design Goals

Following the successful implementation of basic typesetting and internal and external links in the previous article, it is now time to implement footnotes and references. To start, one should note the fundamental difference in the implementation of footnotes between PDF and HTML documents: PDFs are comprised of multiple pages, and thus can sport footnotes in their natural position at the foot of the page, while HTML documents have only one end point. This necessitates a change in the manner in which footnotes are presented between the two.

There are many ways in which one could go about this, putting the footnotes at the end of the section they're defined in, for example. This might be appropriate for extremely long HTML documents, but these articles are designed to be reasonably short. Thus, the desired behavior here is actually quite simple: put the footnotes at the end of the document. However, it is also desirable to leverage the ability for HTML documents to dynamically display content in a context-dependent manner. As such, the design also calls for the ability to display a tooltip-like pop-up when hovering over the marker for a footnote.

References (citations) are similar to footnotes in that they are, in essence, parenthetical insertions to the text to provide supporting or tangential information. Indeed, some writing styles treat references as footnotes and typeset them at the bottom of the page, however most have a dedicated section for references at the end of the document. This is the style that will be pursued in both document types. A key difference between the two is where the actual text is defined. For footnotes, it is defined inline with the text itself. References, on the other hand, are typically defined in a BibTeX file and then generated with the appropriate command. Thus, a different approach will be required for typesetting the references in the HTML documents than for the footnotes. The tooltip-like behavior will also be implemented for references as it provides a natural way for readers to easily check sources without losing their place in the document.

Thus, there are two similar design goals for references and footnotes, summarized as:

- In the HTML document, ensure that footnotes and references are printed at the bottom of the document.
- In the HTML document, present a tool-tip-like pop-up when hovering over either a footnote or reference.
- Use BibTeX to store references for ease of use and good book-keeping.
- Preserve the function of the `\cite` and `\footnote` commands without alteration when compiling to a PDF.

[A]The desired behavior is likely entirely possible in LaTeX via Make4ht, but it was deemed easier to simply use a post-

These are all quite reasonably achievable, though it will require[A] a bit of post-processing.

Additional Configurations

To begin the implementation, one must make some small additions to the configuration file. The first of these, seen in Listing 1, ensures that a Footnote section is always generated, regardless of whether any footnotes are actually defined. This is OK as it will be removed in the post-processing step if no footnotes are found.

Listing 1: Ensure that a Footnotes section is created in the HTML document by inserting this code into the `BODY` generator.

```
% ...
%\Configure{@BODY}{\HCode{<!--Main--><main>}}
\Configure{@/BODY}{
        \EndP
        \HCode{<!--Footnotes--><section class="section">}
        \HCode{<h2 id="Footnotes" class="linkable">Footnotes}
        \HCode{</h2>}
        \HCode{<ol class="footnotes"></ol>}
        \HCode{</section>}
}
%\Configure{@/BODY}{\EndP\HCode{</main>}}
% ...
```

Listing 2: Append this to the configuration file in order to overwrite the behavior of the commands and set them up for post-processing.

```
%Configure Footnotes
\renewcommand{\footnote}[1]{
        \refstepcounter{footnote}
        \HCode{<span class="tooltipcontainer">}
        \HCode{<sup class="tooltipmark">}
        \HCode{<a class="linkicon" href="\#Footnote_\Alph{footnote}">
                \Alph{footnote}</a>
        }
        \HCode{</sup><span class="tooltip footnote"
                onmouseenter="keepOnScreen(this)"
                onmouseleave="resetPositions(this)">
        }
        \HCode{#1}
        \HCode{</span></span>}
}

%Configure Citations and Bibliography
\renewcommand{\cite}[1]{
        \HCode{<stub class="citation">#1</stub>}
}
\renewcommand{\bibliographystyle}[1]{}
\renewcommand{\bibliography}[1]{
        \HCode{<ol class="references"></ol>}
}
```

The other addition can be found in Listing 2. This simply redefines the commands for creating footnotes, citations, and the bibliography. The redefinition of the `\footnote` command warrants some explanation. It first creates a `<span>` tag to contain the entire footnote, followed by another to contain the mark to identify the footnote. The mark is defined to be a letter of the alphabet and links to the relevant entry in the

processing step which will be greatly expanded in function and scope in later articles.

footnote section at the bottom of the document when clicked. After the mark, the actual tooltip text is contained in another `<span>` tag, though this one has event listeners for the mouse entering and leaving the footnote such that the content may be displayed dynamically on mouse hover.

The other things to note in Listing 2 are the definition of a `<stub>` tag and the definition of the bibliography as an ordered list. The tag is not valid HTML, but instead a marker for the post-processing to identify where things should go. This will be a structure that will be used a great deal in the coming articles.

Post-Processing

One could choose nearly any programming or scripting language to perform the post-processing. However, as mentioned in the first article of this project, Python is already required so as to use the Pygments module. Hence, the choice was made to use Python for the post-processing. Regardless of the reason, this is a fairly natural choice as Python is well-known, well-documented, and has a powerful suite of ready-to-use modules. Readers should be able to translate the presented code into any other preferred language, if they should choose to do so.

Listing 3 lays the foundation of the post-processing script. Starting from the `main()` function, the script reads in the HTML file provided as the argument[B] to the script, parses the document, performs any post-processing functions, and overwrites the old HTML file with the post-processed data. The rest of the listing is dedicated to setting up the parser and node tree to store the parsed data in.

The `GeneralHTMLParser` class extends the base HTML parser that comes as part of Python's module library. Here, one really must only define the functions for handling each possible type of HTML content:

- Start Tag
- Start/End Tag
- End Tag
- Data
- Entity Reference
- Character Reference
- Comment
- Declaration
- Processing Instruction

The start and end tags represent a normal `<tag> </tag>` pair. This code creates a new node in the tree (to be discussed below) when encountering a start tag and stepping back up the tree to the parent node when encountering an end tag. Start/end tags are tags like `<hr/>` which may **not** have child elements. Data represents most of the things which actually end up displayed in the web browser, e.g. the text between `<p> </p>` tags. These also may not have children, but their values are stored in the tree. Entity and character references represent special character sequences which allow the presentation of Unicode characters that might not be on the keyboard. These are treated like data. Comments are exactly that; they are discarded so as to remove unnecessary data from the file. The declaration is the header at the top of the HTML file declaring it as such. It is always the first node. Processing instructions are invalid HTML[2] and are discarded. Anything else that might be encountered throws an error.

That's really it for setting up the file parsing. The default module does all the hard work, and all that is left is to do something with the data that it produces. On that topic, the discussion of the tree is next.

Listing 3: The ground work for performing any post-processing. The workhorse of all this is the HTML parser which lexes the HTML file and fills the node tree with the appropriate information.

```
0  import sys
1  from enum import Enum
2  from html.parser import HTMLParser
3
4  # Return code (0 => success, 1 => error)
5  programOut = 0
6
7  # List of nodes representing the HTML document
8  nodes = []
9
10 #An enum class to easily keep track of the type of the HTMLTreeNodes' Type
11 class HTMLContentType(Enum):
12     UNDEFINED        = 0
13     DECLARATION      = 1
```

[B] Calling the script should look something like: `python postprocessing.py myfile.html`

```
    NORMAL_TAG       = 2
    SELF_CLOSING_TAG = 3
    CONTENT          = 4
    ENTITY_REF       = 5
    CHAR_REF         = 6
#

# Basic tree node class for storing HTML DOM
class TreeNode:
    global programOut
    htmltype = HTMLContentType.UNDEFINED
    value    = None
    parent   = None
    children = None

    def __init__(self, htmltype, value, parent, children):
        self.htmltype = htmltype
        self.value = value
        self.parent = parent
        self.children = children
    #

    def __str__(self):
        match self.htmltype:
            case HTMLContentType.UNDEFINED:
                print("[ERROR] Tree node of undefined type!")
                programOut = 1
                return ""
            #
            case HTMLContentType.DECLARATION:
                out = ""
                for child in self.children:
                    out += str(nodes[child])
                #
                return f"<!{self.value}>" + out
            #
            case HTMLContentType.NORMAL_TAG:
                out = ""
                for child in self.children:
                    out += str(nodes[child])
                #
                attrStr = ""
                for attrib in self.value[1]:
                    attrStr += f" {attrib[0]}=\"{attrib[1]}\""
                #
                out = f"<{self.value[0]}{attrStr}>" + out
                return out + f"</{self.value[0]}>"
            #
            case HTMLContentType.SELF_CLOSING_TAG:
                attrStr = ""
                for attrib in self.value[1]:
                    attrStr += f" {attrib[0]}=\"{attrib[1]}\""
                #
                return f"<{self.value[0]}{attrStr}/>"
            #
            case HTMLContentType.CONTENT:
                return self.value
            #
            case HTMLContentType.ENTITY_REF:
```

```
                return "&" + self.value + ";"
            #
            case HTMLContentType.CHAR_REF:
                return "&#" + self.value + ";"
            #
            case _:
                print("[ERROR] Tree node of unhandled type!")
                programOut = 1
                return ""
            #
        #
    #
#

# Parser for the HTML file
class GeneralHTMLParser(HTMLParser):
    global programOut
    index = 0

    def handle_starttag(self, tag, attrs):
        nodeType = HTMLContentType.NORMAL_TAG
        nodes[self.index].children.append(len(nodes))
        nodes.append(TreeNode(nodeType, (tag, attrs), self.index, []))
        self.index = len(nodes) - 1
    #
    def handle_startendtag(self, tag, attrs):
        nodeType = HTMLContentType.SELF_CLOSING_TAG
        nodes[self.index].children.append(len(nodes))
        nodes.append(TreeNode(nodeType, (tag, attrs), self.index, []))
    #
    def handle_endtag(self, tag):
        self.index = nodes[self.index].parent
    #
    def handle_data(self, data):
        nodeType = HTMLContentType.CONTENT
        nodes[self.index].children.append(len(nodes))
        nodes.append(TreeNode(nodeType, data, self.index, []))
    #
    def handle_entityref(self, name):
        nodeType = HTMLContentType.ENTITY_REF
        nodes[self.index].children.append(len(nodes))
        nodes.append(TreeNode(nodeType, name, self.index, []))
    #
    def handle_charref(self, name):
        nodeType = HTMLContentType.CHAR_REF
        nodes[self.index].children.append(len(nodes))
        nodes.append(TreeNode(nodeType, name, self.index, []))
    #
    def handle_comment(self, data):
        pass
    #
    def handle_decl(self, decl):
        nodeType = HTMLContentType.DECLARATION
        nodes.append(TreeNode(nodeType, decl, 0, []))
        self.index = 0
    #
    def handle_pi(self, data):
        pass
    #
```

```
132         def unknown_decl(self, data):
133             print(f"[ERROR] Unknown HTML declaration: {data}")
134             programOut = 1
135             return
136         #
137     #
138
139     def main():
140         global programOut
141
142         # Read in the html file to process
143         htmlFile = open(sys.argv[1], "rt")
144         htmlText = htmlFile.read()
145         htmlFile.close()
146
147         # Parse the htmlText into a tree representing the DOM
148         parser = GeneralHTMLParser(convert_charrefs = False)
149         parser.feed(htmlText)
150         parser.close()
151
152         # Postprocessing functions will go here
153         # ...
154
155         # Write the modified DOM back to the original file
156         htmlFile = open(sys.argv[1], "wt")
157         htmlFile.write(str(nodes[0]))
158         indexFile.close()
159
160         sys.exit(programOut)
161     #
162
163     # Call the main() function when script is run
164     if __name__ == "__main__":
165         main()
166     #
```

Figure 1: An example simplified tree structure of the HTML DOM. Note the use of semantic HTML[3] to ensure a clean experience for screen readers and the like.

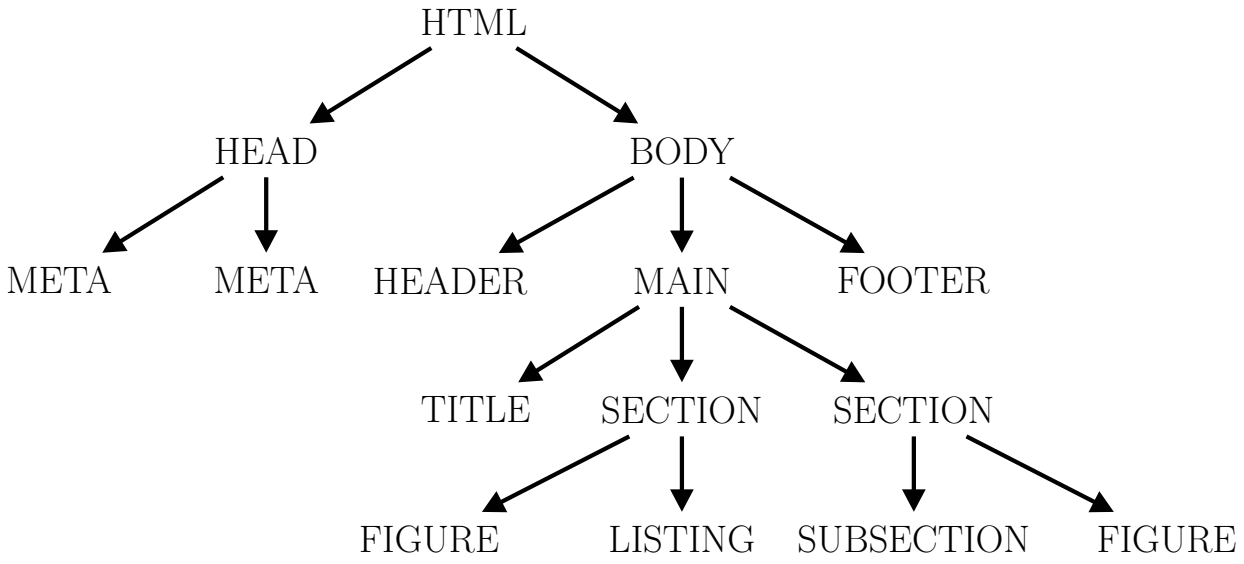

Figure 1 provides a simplified example of what the tree is meant to represent: the Document Object Model.[1] The definition of the `TreeNode` class in Listing 3 reflects this by having an enumerated type, a parent, children, and a value. The parent and children should be largely self-explanatory, but the value is not immediately obvious as it is dependent on the type.

If the type of the node is a normal or self-closing tag, then the value is a two-member tuple where the first member is the tag type (e.g. section, p, span…) and the second member is a list of attribute key/value pairs. If the node is content, an entity reference, character reference, or declaration, then the value is simply value returned by the parser.

The bulk of the `TreeNode` class is made up of the definition of the `__str__` method. This is called when the code requires that the object be turned into a string. For example, this could be when printing to the standard output, converting to a string for manipulation, or writing to a file. The definition provided recursively calls each child of the initial node such that the output string is well formed HTML. Thus, returning to the main function again, one must only write out `str(nodes[0])` to the file and all should function well.

Footnotes

Readers should note one peculiarity of this code as it is: the nodes themselves are stored in a list and the indices in the list of the parent and children are stored in the nodes. This is an unfortunate work around so that a node may be the child of more than one node. This will be immediately useful as Listing 4 “duplicates” the footnote text by simply pointing the children of the footnote at the bottom of the page to be the children of the inline footnote.

The helper function included here will be used a great deal in many of the other functions. So as not to overly lengthen this already too-long article, it simply traverses the tree structure looking for tagged nodes with the given set of attributes. It returns those it finds as a list.

Listing 4: The function to process footnotes and a helper function to find tags with specific attributes.

```
0  # Find the indices of all nodes in the tree below the given index with the
1  # given tag and attributes
2  def findTags(index, tag, attr = []):
3      global programOut
4      out = []
5      if(nodes[index].htmltype == HTMLContentType.NORMAL_TAG or
6         nodes[index].htmltype == HTMLContentType.SELF_CLOSING_TAG):
7          if nodes[index].value[0] == tag:
8              if(attr == None or len(attr) == 0):
9                  out.append(index)
10             else:
11                 addIndex = True
12                 for toMatch in attr:
13                     match(toMatch):
14                         case tuple():
15                             if not (toMatch in nodes[index].value[1]):
16                                 addIndex = False
17                             #
18                         #
19                         case str():
20                             isMatched = False
21                             for matchTo in nodes[index].value[1]:
22                                 if toMatch == matchTo[0]:
23                                     isMatched = True
24                                     break
25                                 #
26                             #
27                             if not isMatched:
28                                 addIndex = False
29                                 break
30                             #
31                         case _:
32                             programOut = 1
33                             print(f"[ERROR] Unhandled type in findTags array.")
```

```
                            return
                        #
                    #
                if addIndex:
                    out.append(index)
                #
            #
        #
    #
    for child in nodes[index].children:
        for x in findTags(child, tag, attr):
            out.append(x)
        #
    #
    return out
#

# Find footnotes and create they're mirro at the bottom of the document.
def makeFootnotes():
    global programOut
    indices = findTags(0, "span", [ \
        ("class", "tooltip footnote"), \
        ("onmouseenter", "keepOnScreen(this)"), \
        ("onmouseleave", "resetPositions(this)")])
    fnSecIdx = nodes[findTags(0, "h2", [ \
        ("class", "linkable"), \
        ("id", "Footnotes")])[0]].parent
    fnListIdx = findTags(fnSecIdx, "ol", [("class", "footnotes")])[0]
    if len(indices) == 0:
        nodes[nodes[fnSecIdx].parent].children.remove(fnSecIdx)
        return
    #
    for i in range(len(indices)):
        nodes.append(TreeNode(
            HTMLContentType.NORMAL_TAG,
            ("li",[("id","Footnote_" + chr(65 + i))]),
            fnListIdx,
            nodes[indices[i]].children))
        nodes[fnListIdx].children.append(len(nodes) - 1)
    #
    return
#
```

With that, one must only call the `makeFootnotes()` function in the appropriate place of the `main()` function, and the footnotes will appear both inline and at the bottom of the document. They still need to be styled with CSS and a bit of scripting will need to be done to ensure that the tooltip stays on the screen, but that will wait until after references have been handled.

References

References must be handled in a slightly different fashion than the footnotes for all the reasons described prior but also for another: it is not uncommon to cite the same source multiple times in the same document. This requires one to ensure that the citations work inline as a tooltip, possibly many times for the same citation, and also ensure that the citation is listed only once in the Reference section.

Listing 5 provides the function that should be called to fulfill the stated goals. From the top, it first finds all of the citation stubs that were left by Make4ht and the References section. If the handful of logic checks are passed, the function then opens every `*.bib` file in the source directory and parses them. The parsing of the BibTeX files will not be discussed here, but could make for an interesting exercise for someone learning regular expressions as their format[C] is designed for easy pars-

[C] The BibTeX format is infuriatingly close to `JSON`, but not interchangeable. In fairness, BibTeX predates `JSON` by nearly twenty years.[4, 5]

ing.

Using the list of citation indices and the information extracted from the BibTeX files, the function then builds the inline citation tooltip structure. This can happen in two slightly different ways, depending on whether there are multiple citations in a single `\cite` command or not. Regardless, the structure is injected into the DOM using the HTML parser used earlier to parse the entire document.

Listing 5: The function to find the citation stubs and replace them with the properly formatted citation text. Copy that text into the References section at the bottom of the document.

```
import glob

# Inject the reference information inline and in the reference section
def makeReferences():
    global programOut

    # Find the citations. Pop error if can't find reference section
    indices = findTags(0, "stub", [("class", "citation")])
    hdrIdx = findTags(0, "h2", [("class", "linkable"), ("id", "References")])
    if len(hdrIdx) == 0:
        if len(indices) != 0:
            programOut = 1
            print("[ERROR] No Reference section found but citations exist!")
        #
        return
    #
    refSecIndex = nodes[hdrIdx[0]].parent

    # If no citations and reference section exists: remove reference section
    if len(indices) == 0 and refSecIndex != 0:
        nodes[nodes[refSecIndex].parent].children.remove(refSecIndex)
        return
    #

    # Find bib files. Pop error if none are found
    bibFileNames = glob.glob("*.bib")
    if len(bibFileNames) == 0:
        programOut = 1
        print("[ERROR] No bibliography file found despite citations.")
        return
    #

    # Get the info from the bibfiles, store in dict
    bibDict = {}
    for bibFileName in bibFileNames:
        res = parseBibFile(bibFileName)
        for x in res:
            if x in bibDict:
                programOut = 1
                print(f"[ERROR] Mutiply defined bib entry '{x}'")
                return
            #
            bibStr = ""
            match res[x]["type"]:
                case "article":
                    bibStr += res[x]["author"] + ". "
                    bibStr += res[x]["title"] + ". "
                    bibStr += res[x]["year"] + ". "
                case "book":
```

```
                    bibStr += res[x]["author"] + ". "
                    bibStr += res[x]["title"] + ". "
                    bibStr += res[x]["year"] + ". "
                    #bibStr += res[x]["url"] + ". "
                case "misc":
                    bibStr += res[x]["title"] + ". "
                    bibStr += res[x]["howpublished"] + ". "
                    bibStr += res[x]["note"] + ". "
                case _:
                    programOut = 1
                    print(f"[ERROR] Unhandled bibtex entry type " + \
                        "'{res[x]["type"]}'")
                    return
                #
            #
            bibDict[x] = bibStr
        #
    #

    # Inject the bib info into the document
    toList = []
    parser = GeneralHTMLParser(convert_charrefs = False)
    for index in indices:
        text = ""
        nodes[index].value = ("span", [("class", "tooltipcontainer")])
        id = nodes[nodes[index].children[0]].value
        if ',' in id:
            idList = (re.sub(r"\s+", "", id)).split(',')
            for item in idList:
                if item not in toList:
                    toList.append(item)
                #
            #
            for idx in range(len(idList)):
                if idx != 0:
                    text += "<sup>,</sup>"
                #
                refNum = toList.index(idList[idx]) + 1
                text += f"<sup class=\"tooltipmark\"><a class=\"linkicon\"" + \
                    "href=\"#Reference_{refNum}\">{refNum}</a></sub>"
                text += "<span class=\"tooltip reference\"" + \
                    "onmouseenter=\"keepOnScreen(this)\"" + \
                    "onMouseLeave=\"resetPositions(this)\">"
                text += bibDict[idList[idx]]
                text += "</span>"
            #

        else:
            if id not in toList:
                toList.append(id)
            #
            refNum = toList.index(id) + 1
            text  = f"<sup class=\"tooltipmark\"><a class=\"linkicon\"" + \
                "href=\"#Reference_{refNum}\">{refNum}</a></sup>"
            text += "<span class=\"tooltip reference\"" + \
                    "onmouseenter=\"keepOnScreen(this)\"" + \
                    "onMouseLeave=\"resetPositions(this)\">"
            text += bibDict[id]
            text += "</span>"
```

```
        #

        # Clear the children of the node and parse the bib info into it
        nodes[index].children.clear()
        parser.index = index
        parser.feed(text)
    #

    # Build the refernce list for the reference section and inject it
    text = "<ol class=\"references\">"
    for id in toList:
        refNum = toList.index(id) + 1
        text += f"<li id=\"Reference_{refNum}\">"
        text += bibDict[id]
        text += "</li>"
    #
    text += "</ol>"
    parser.index = refSecIndex
    parser.feed(text)
    return
#
```

Finally, the reference list is constructed and injected into the DOM in the same manner as the inline citations. Note that the function does not point the child of the list item to one of the inline citations, but instead recreates the text. This is due to the possibility for a citation to occur multiple times within the text. It is undoubtedly possible to find a matching citation and point towards it, but it was deemed easier to do this instead.

Wrapping Up

Footnotes are references are now fully implemented (including the ability to nest them). Admittedly, a few things were skipped, such as the parsing of the BibTeX files, but not every detail can be included here. Readers who are following along will likely note that the inline footnotes and citations are currently visible in the HTML document even when not hovering over the marker. Getting the tooltip behavior requires a bit of CSS magic and then a bit of JavaScript to ensure that they stay on the page. The JavaScript is left as an exercise for the reader, the CSS is covered in another article.

The next article in this series covers equations and tables. Equations are extremely straightforward; tables are somewhat less so. Indeed, they are probably the feature most in need of post-processing as the default Make4ht output is not strictly well-formed HTML.

References

[1] Mdn web docs: Document object model. `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model`. Accessed: 2026-01-31.

[2] Mdn web docs: Processing instructions. `https://developer.mozilla.org/en-US/docs/Web/API/ProcessingInstruction`. Accessed: 2026-01-31.

[3] Mdn web docs: Semantic html. `https://developer.mozilla.org/en-US/curriculum/core/semantic-html/`. Acessed: 2026-02-01.

[4] Wikipedia: Bibtex. `https://en.wikipedia.org/wiki/BibTeX`. Acessed: 2026-02-01.

[5] Wikipedia: Json. `https://en.wikipedia.org/wiki/JSON`. Acessed: 2026-02-01.