date: 2023.12.07

title: Blogging with Word

I [wrote a book](https://augustl.com/blog/2023/pro_kotlin_web_apps_from_scratch/) in Word. That’s 72 000 words in Word.

But I’ve stopped blogging, partly because writing markdown in my code editor just doesn’t vibe with me anymore. These days, I ask myself the question: how can I just get a Word document online as fast as possible?

I like Word! It has nice tooling for grammar and spell checking, and it’s a nice environment for editing prose.

So, I wrote a parser for Word files that slurps them into my blog engine and emits clean HTML – like the post you’re seeing now! Yes, my friends, I now have support for .docx files in my personal blog.

Tldr: [Here’s the Word document](https://github.com/augustl/augustl.com/tree/master/posts/blog/2023) (called blogging\_with\_word.docx), and [here’s the parser code](https://github.com/augustl/augustl.com/blob/b6a3382e6ad49a4bd710332618cb58140c2cf4bd/src/augustl_com/word_post_parser.clj).

## A quick word on tech

Programmers should build their own ~~lightsaber~~ blog.

This blog is a JAM stack style Clojure app (it used to be a Ruby/Middleman app, and before that, Rails) that reads a bunch of files and outputs static HTML, that I host on CloudFront in AWS. [Source code here](https://github.com/augustl/augustl.com).

This setup turns out to be the best of both worlds. I get to write some fun code, once, and hopefully when it’s all up and running, I can just paste another docx file into the correct folder and have a fresh blog post online in no time.

Back in the old days I enjoyed hand-typing markdown in my code editor. In fact, the first years of this blog was hand-typing HTML. Clearly, I have changed.

## Parsing Word docs

I’m glad I use [a robust technology like the JVM](https://news.ycombinator.com/item?id=4310723), which is the platform Clojure runs on. Clojure has great Java interop, so I’m using the [good old Apache POI library](https://poi.apache.org/components/document/quick-guide-xwpf.html). It has full support for everything Microsoft Office, and convenient APIs for extracting text from a Word document.

Let’s get into it!

(defn get-body-from-docx [^File file]

(with-open [doc (XWPFDocument. (io/input-stream file))]

(->> (iterator-seq (.getParagraphsIterator doc))

(drop-while #(re-find header-line-re (.getText %)))

(map (fn [para]

(if-let [runs (seq (.getRuns para))]

{::runs runs

::style (.getStyleID para)

::para para

::doc doc}

::blank)))

(word-paragraphs-hiccup-seq))))

iterator-seq and the getParagraphsIterator() method are the juicy bits. It yields the Word document paragraphs as a plain lazy Clojure sequence, which lets me pretend I’m writing code against a neat Clojure library, instead of a noun heavy Java library.

The drop-while gets rid of the initial header lines. My word document starts with some metadata directly in the text, such as date: 2023-12-07. It also has title of the post, and so on.

Then I map each paragraph to ::blank for empty lines, and create a map with all the data I need for further processing for lines with contents.

## Chunking paragraphs

Then, the fun process of converting Word paragraphs to HTML begins. Paragraphs in Word documents are not like paragraphs on the web. A paragraph in Word is more like a sequence of <div>s, or lines of text separated by a <br> tag.

Let’s say the word document looks like this:

Here is my line

Here is my other line

with no space between

Here is my last line

That should result in HTML that looks roughly like this:

<p>Here is my line</p>

<p>Here is my other line<br>with no space between</p>

<p>Here is my last line</p>

In other words, the list of paragraphs from Word needs to be chunked properly.

This is what word-paragraphs-hiccup-seq does. That function is just some looping mechanics, the meat of that implementation is a function, get-next-hiccup-tag. This function returns the new HTML for the paragraphs it “consumed”, and the remaining paragraphs to be processed.

(defn get-next-hiccup-tag [paragraphs]

(let [paragraphs (->> paragraphs

(drop-while #(= % ::blank))

{style ::style runs ::runs doc ::doc} (first paragraphs)]

(case style

"Code" (let [[code-paras rest] (->> paragraphs

(consume-paragraph-chunk #(= "Code" (::style %))))]

[(get-code-block-tag code-paras) rest])

"Heading2" [(into [:h2] (runs-to-hiccup-seq runs doc)) (rest paragraphs)]

"Heading3" [(into [:h3] (runs-to-hiccup-seq runs doc)) (rest paragraphs)]

"Heading4" [(into [:h4] (runs-to-hiccup-seq runs doc)) (rest paragraphs)]

(get-next-hiccup-paragraph paragraphs))))

The first step is to remove any preceding blank paragraphs. If I insert 10 blank spaces in a row, I don’t want word to insert to paragraphs separated by 9 (?) <br> tags. This makes my Word documents behave similar to Markdown, where you also have to do some work to get a bunch of whitespace between paragraphs.

The ::style property of the first paragraph determines what to do. If it’s a code block or a header, I immediately create an element and then return the remaining paragraphs. There’s some logic in code blocks I’ll return to later. But that’s about it.

If the paragraph does not have a style associated with it (meaning it’s just a normal block of text), I invoke the main bulk of the chunking logic.

(defn get-next-hiccup-paragraph [paragraphs]

(let [[text-paragraphs rest] (->> paragraphs

(split-with

#(and (not= % ::blank)

(nil? (::style %)))))]

[(into [:p]

(->> text-paragraphs

(map #(runs-to-hiccup-seq (::runs %) (::doc %)))

(interpose [:br])))

rest]))

First, I take all the blocks of non-styled paragraphs that also aren’t blank lines. Then, I build a <p> tag that contains each paragraph converted into HTML (it reads out bold, italics, links, etc from the text runs), and I insert a <br> between each line in the paragraph.

The rest is what is returned to the looping code, meaning that for the next iteration of the processing loop, the remaining characters will be processed.

## Processing code blocks

I invoke a function called consume-paragraph-chunk to build the code blocks. Here’s that function.

(defn consume-paragraph-chunk [pred xs]

(let [[chunk rest] (split-with #(or (pred %) (= ::blank %)) xs)

whitespace-tail (->> chunk (reverse) (take-while #(= ::blank %)))]

[(->> chunk

(drop-last (count whitespace-tail)))

(concat whitespace-tail rest)]))

Maybe there’s a smarter way to do this, but at this point I just wanted to get this blog post published. The function takes all lines that are blank or matches the predicate (in this case, ::style is "Code"). I don’t want it to include any tails of whitespace after the last chunk of code, so I yank those out and add them to the rest for further processing in the loop.

## Great success

So that’s about it! I wrote this very blog post in Word. This has increased the likelihood of increased blog output by at least 1%.

My number one fear is that this will turn out to be some kind of churnfest due to incompatibilities with my old blog posts in new versions of Word, or something like that. The docx format has been stable for a while now though, so I sure hope not. At any rate, it was a fun little project to get up and running.

## Test area

Because I’m lazy, I don’t want to set up a separate test environment. So here’s a blurb that tests some random stuff.

Here, we should see a continuous block of code, with newlines inside it but not after.

(defn here-is-some-code []

;; Test newline stuff

;; Ok here we go

(prn "Hello, World!"))

Here, we should see a set of words separated by <br> tags. And, we should not see an actual line break here, but a single paragraph, and some in-line escaped HTML.

Can we have in-line code? (+ 1 2). This should use a monospace font.

Here

We

Go

We’ve already tested links. But let’s [test it again](https://crud.business/).