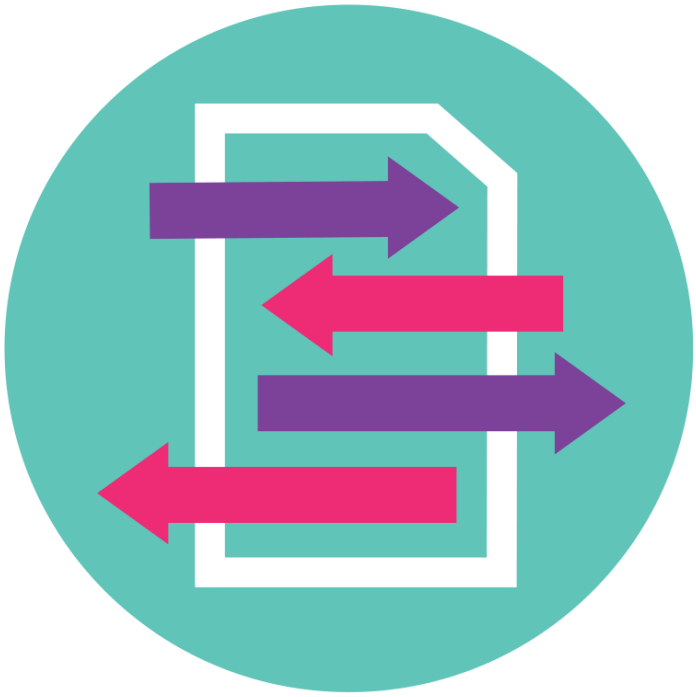
Coko

HTML Typescript - redistributing labor

A thought piece on how Single Source Publishing redistrubutes labour.

Published: May 6, 2018
Author(s): Adam Hyde , Wendel Piez
Illustrations: Henrik van Leeuwen
PDF Design: Harshna Haswani
Rendered from HTML using Paged.js

note: written in 2018 and to be updated soon!



Apparently small changes in writing technologies can radicalise publishing workflows. Matthew Kirschenbaum makes a good case for this in his book “Track Changes: A Literary History of Word Processing“.

There are a number of cases in the book that illustrate this. I’m about 1/3 of the way through and the example that astonished me was related to Issac Asimov.

The setting is mid 1981 and Asimov has just encountered his first word processor at age 61 (Kirschenbaum places 1981 as, pretty much, the start of the word processing revolution). Previous to this, Asimov had written with a typewriter. He was extremely fast and hence many errors were made that could not be corrected easily with a typewriter. Errors, in those days, were the problem of the copy editor who marked the manuscript up with corrections.

As it happens, Asimov finds typing *faster* with the new electronic word processor and so he makes *more* mistakes. The copy editor then gets frustrated and in turn frustrates Asimov with many more clarifying queries about the errors made. Dr Asimov then starts correcting his own text using find-and-replace to correct common errors...

> “cold” suddenly becomes “could” and no sign exists that it was ever anything else.

The result is cleaner copy which elicits fewer inline queries from the copy editor. Right there, publishing workflows changed and labor was redistributed from

the copy editor to the author.

A migration from a typewriter to a word processor is *not* a small change. However, on a higher, more abstract, level the ability to correct content 'as you go' is not a huge cognitive leap. When the technology enabled this, however, the change it effected was great.

Small design changes can have huge consequences for publishing workflows to the point of redefining roles. With the introduction of the word processor came the ability to change copy easily and consequently the author assumed more of this role and the copy editor's role was redefined.

I think we might have something similar in the systems we are building at the moment – particularly Editoria, the platform for scholarly monograph production. I'm not claiming the system will have an effect on the same scale of the introduction of word processors, rather that a few, apparently small, design decisions will have a large impact on the workflow of those using the system. This aiming to achieve, but *how things are done* ie. a similar redistribution of labor.

In the current workflow of the University of California Press (UCP), with whom we are designing the system, the following steps occur:

1. The acquisition dept acquires the book from the author in form of a collection of Microsoft Word (MS Word) files in docx format. One file per chapter. These are sent to the production dept.
2. Production editors (in the production dept) open each file on the premises of UCP where MS Word is installed with custom macros. These macros enable the editors to select a part of the text and apply custom styles (see image below).
3. The newly styled MS Word files are then pushed through the review cycle featuring the copy editor and author(s).

Step 2, styling, is a long manual process. Also, it can *only occur* on computers that have these macros installed. Further, the subsequent edits by the copy editor and author(s) might revert some styling changes. So some of the work will need to be done again. Lastly, when the version of MS Word is updated then the Macros must be checked, potentially rewritten to keep pace with the new version, and re-installed on each machine.

So...the *small* change we are introducing we call *HTML Typescript*. It is nothing fancy, nothing new, but it is a fresh look at how we can bring offline word processing workflows into the browser. The critical problem being, how to do you get the MS word files provided by an author into a web-based production workflow? In clearer, slightly more technical, terms how do you transform a document from MS Word to HTML? Well... many people literally say "it can't be

done.” The criticism is not as literal as it sounds and really comes down to what you are trying to achieve. If you wish a one-to-one conversion of an author’s structural intent for the production a correctly marked up HTML document, then automated conversion through any process is not going to achieve this. Not even the very complex processes that go from docx to structured XML (of some type) and then to HTML, which is the approach file conversion specialists prefer, will achieve this.

This is where we think HTML Typescript solves some very interesting problems.

HTML Typescript offers a reframing of the problem that, consequently, enables processes that closely match the publisher’s current workflow. It also offers the potential to radically redefine that workflow.

Docx is a type of XML. It is, laughably, called OpenXML by Miscrosoft. Laughably, because while you can unzip a docx file (a docx files), you can open the *document.xml* and poke around with it (see video demo below), but it is incomprehensible. There are two critical problems that contribute to this:

1. The XML is pretty unreadable by humans. It is dense and verbose and very very messy.
2. Due to the editing environment (MS Word) and its inability to constrain authors from using ‘font size 12, bold’ instead of marking the text correctly as a Heading Level 2 (for example), the resulting styles applied by the author, and described in the XML, are erratic at best.

https://coko.foundation/images/uploads/demo-docx-extract.mp4

Number 2 is the big problem, and issue number 1 helps obfuscate it.

So, when you want to transfer from docx to HTML, it is very difficult to determine the author’s strucural intent without looking at the original display version of the MS Word file. Is this indented single line marked ‘font size 16.6, bold’ a heading 1,2, or 3...or is it meant to be a block quote? For example...

This is exactly why the macros mentioned above must be used, because named styles are not used ‘upstream’ by the author. So the production editors must work their way through each file and apply the correct designated styles using the custom macros.

With XSLT, the conversion process of choice for file conversion pros, it is very possible to convert from docx to HTML. Nothing impossible there. You don’t end up losing content (there are a few gotchas for this which I will cover in later posts) but you do end up with messy unstructured HTML. This is because the original docx is messy and unstructured. This is what people mean when they say ‘it can’t be done’ ie. you cannot infer all the structural intent of the author

from the docx file, because it is a mess, and by some magic subsequently convert it to lovely clean, structured, HTML.

But, that is actually ok. Converting to 'messy' HTML really puts you in about the same position as having a messy docx file. They are both equivalently unstructured.

File conversion specialists don't like this position. They want, by their very nature, documents to be nicely structured. HTML does not, in its raw form, stipulate much structure. Sure you can have headings 1,2,3 but also you can have, as MS Word does, arbitrary font sizes and styles that *look* like a heading, but the underlying markup doesn't explicitly state this is the case. In addition, HTML doesn't worry about document sectional structure too much. What if you wish to identify a section of the text as a 'method' section? How do you deal with that? Custom XML can deal with this as you can design a mark-up structure to suit. But HTML, as it is described by the standards, doesn't enable this (yet...web components will change this to some degree). *However,* with plain ole HTML you can use any sort of div or span you like to wrap around sections of a HTML document and call them what you like. For example, you can have 'div class="method"' and there you go...the section is defined, for *your* intents and purposes, as a method section. This does not have much currency in the outside world (the world outside your platform) since there is no standard way of describing a method section in HTML, but for the purposes of creating production systems, this is really ok. We can work with this. The important thing is that we can now apply document structure with HTML should we wish to. When we wish the document to leave the system we can make that conversion at export time (more on this below).

So... Where does that leave us? Ok... well, at the time of conversion we can move from docx to HTML and make some 'educated' guesses as to what the author's intent is. For example, if there is one single line text with 'font size 24, bold' and then 16 of 'font size 20, bold' and 6 of 'font size 14, bold' and then a bunch of sentences groups with font size 12 – then we can map these to heading 1,2,3 respectively, and the last being standard paragraphs. It is *not* perfect -it will not catch all structure, and the process will make incorrect inferences. However, it will get us part of the way there. So we can *already* start improving on the structure of the MS Word file automagically.

Arguably, we are in a better position with an initial rules-based clean up of the file structure as it passes from docx to HTML. The good thing is that we can improve these rules over time. In time, the automated conversion will produce better results.

After a conversion of this kind, we have a *partially* structured file. This is where file conversion specialists often leave the conversation. They don't like partially

structured anything. It is not in their DNA. That is because they are primarily concerned with conforming file A into structure X. Their metric is ' is it well structured?'. It is a pass/fail binary. If you are to look primarily at whether file A or B is well structured then you want well-defined schemas that describe the structure, and you want files that subscribe perfectly to that schema. Anything that falls out of this is a fail. Partially structured documents are a fail.

But with the HTML Typescript approach we see partially structured documents as a strength, not a weakness. We know it is not possible to get to perfectly formed documents in one go, so rather than consider this a fail we accept we must get there *progressively*. That is the fundamental principle behind what we are calling *HTML Typescript.* It is the use of HTML as a document format that can be progressively improved to get to the structure you desire over time using both machine and manual processes.

HTML is the perfect format for this. HTML's lack of formal structure, along with its ability to define any kind of structure you want, enables us to progressively add any kind of structure we need to a document. One part of this process is the automated clean up at conversion time, and the next part of the process is where it starts getting interesting... this is the *manual* application of structure.

This is where the *apparent weakness* of HTML becomes a strength – we can manually add structure over time. We can progressively structure the document. For this process, we can build, and the Coko team are building, a suite of tools in the production environment so that a production editor (for example) can click on an element (eg a heading) and choose the style they wish to apply from a menu – similar to how they currently work with macros.

The advantages of adding the correct structure in the browser vs MS Word? Well, firstly, as mentioned above, we can computationally improve the structure before the manual process. This results in less work to do. Secondly, we have complete control over the tools available to the platform's distributed citizens (I don't like the term 'user' and so are experimenting with other terms. Let's try citizen for now). Hence we can make the tools available to everyone, not to just those citizens with the MS Word macros installed. That means:

1. There is no need to update the (equivalent of) macros against the underlying desktop software version across many machines.
2. If I wish to update the features that enable styling, then all users can leverage these updates immediately.

So, as a publisher, I'm not stuck in the harrowing, expensive, cycle of continual software upgrades and installs against random (or planned) updates of MS Word that maybe conducted by my org. That is already a saving. There is a second tier of savings here – we are building open source systems with the

intention that they are used by a large number of orgs. If we share a common platform and common toolkit, we can also share the costs of maintenance and development, and free from vendor upgrade cycles. Each publisher doesn't have to do their own software development to keep their own styling macros up to date. Yet you can still innovate and develop new features – while any new feature can potentially be available to everyone.

But...more importantly...just as with great power comes great responsibility, we could say that *with a shared toolset comes shared responsibility*. That doesn't reference the sharing of costs mentioned above (although it could) but rather it references the roles of the people in the organisation using the toolset. If the tools for styling are available to all citizens, then so might the responsibility for using those tools. Much like Asimov noticed he could now do some of the copy editing, so might a publisher re-distribute the work of styling a text. This is where a small design decision might have a large impact. Publishers that are forced to design workflows based, literally, on *where the tools are* (what machines the macros are installed on in this case), can now design workflows dependent on who they think would be best placed to do the work. That is pretty interesting. Such a small design decision might actually cause pretty radical changes to workflow.

There are a couple of things I want to comment on with regard to this. Firstly, achieving efficiencies like the above are only available to you if you design *systems* rather than software. Designing software is a technical endeavor, designing systems is not. Designing systems requires an understanding of what is trying to be achieved, by whom, and with what constraints. It is very difficult, for example, to imagine and design a progressive approach to structuring files if you are simply focused on building software that produces well-structured files. It sounds like a distinction without a difference, but it's not. The difference is profound.

Secondly, what I have described above requires, in this implementation, *HTML Typescript*. But HTML Typescript is not a format, it is merely a way of using an existing format – HTML – and seeing its apparent weakness (no strict formal structure) as a strength. So HTML Typescript is not really a technical solution so much as it is an approach — one which enables us to build interesting systems to solve the long-standing problem of 'getting out of MS Word' and into the web.

Lastly, there may be some concern that this kind of approach doesn't get us to structured formats like JATS (or whatever) that may be required by your content sector. Well, that's not a big deal. You can design your version of HTML with all the structure you need to make an easy conversion to whatever XML format you want. That's not so tricky. HTML can contain that structure and you can

build the production tools (the web-based equivalent of the MS Word macros) to suit. In other words, end-of-line formats (for distribution or syndication) should remain an end-of-line problem. When you press ‘export’ (or whatever), then that is the time, presumably, when the document is nicely structured, that you convert it to whatever you need in a relatively straightforward conversion.

In summary, I’ve been kicking around in this area for a long time and feel that there are some ideas that are surfacing that not only solve some tricky problems, but open the door to new possibilities. I think the HTML Typescript approach is one such case and I’m looking forward to finishing the implementation of the HTML Typescript approach with Editoria and trialing in the weeks to come. More thoughts to come.

HTML Typescript is an approach developed by myself and Wendell Piez.