Why our HTML Docs don’t just `Print` and what to do about it

Nikolaj Potashnikov
PhD in Economics, Solution architect, Course-IT
consulting@yandex.ru @nmpotashnikoff

- As a solution architect I need to create targeted docs for each stakeholder. What is the easiest way to share them? Via email, messenger… as a PDF, a Word file. In short, in a 'print' format.

List of slides

1. Way in converting simple text markup to print formats....2
2. Print, Export to Word, Export to PDF are very often just a trap....2
3. Main formats for printing....3
3.1. Most widespread rendering approaches....3
4. Some brief conclusions....4
5. UniDoc Publisher approach suits best if at least one of....4
6. In search for flexibility: Asciidoctor open document....4
6.1. A simplified processing AST example....5
6.2. Great, but....5
6.3. And still....6
7. Thoughts before the second step....6
8. Native converter as a reader?....6
9. Let’s convert this presentation to LO Writer....7
10. Notes on this demo....7
11. Boilerplate....8
12. Processing AST....8
13. Rearranging title (Asciidoc source)....9
14. Rearranging title, extracting semantics....9
15. Rearranging title, constructing title....10
16. Let’s return to the result....10
17. And a little bit of styling....11
18. Let’s return to processing AST....11
19. Extending AST....12
19.1. Testing....12
20. Conclusion....13
21. Questions?....13
22.1. Comparison....13
23. Comparison #2....13
24. Comparison #3....14

1. Way in converting simple text markup to `print` formats

- https://github.com/CourseOrchestra/course-doc: XSL-FO templates for Asciidoctor → Docbook backend
- https://github.com/CourseOrchestra/asciidoctor-open-document: Open Document Converter for Asciidoc
- https://github.com/fiddlededee/unidoc-publisher: UniDoc Publisher — any markup to any printing rendering engine

---

- Today we are speaking about UniDoc Publisher approach, but we'll look at Asciidoctor Open Document. Why despite success I felt obliged to make an extra step.

2. `Print`, `Export to Word`, `Export to PDF` are very often just a trap

What to do with long line in listing?

- We may scale
- Or use landscape orientation
- Or both, but would it be enough?
- If not, we may fire error for long lines
- Or wrap them
- With linefeed and spaces? And how to copy?
- With indents? Still impossible to copy from PDF

Tip
And in web we can just add horizontal scroll bar.

- There is the core mismatch: semantic markup meets the rigid world of print.
- Take a simple listing… (go through very briefly just to give filling)
  - Stop on orientation — it is already not about semantic markup. Landscape for listing, for section, for higher level section? Should tech writer use media hints?
- My goal for today is to show my approach to for tackling such problems UniDoc Publisher

3. Main formats for printing

1. PDF
2. Text processing formats (Open XML — MS Office, Open Document — LibreOffice)
3. HTML?

Tip
CSS Paged Media — CSS extension, defining style specific for printing

---

- Despite we live with Paged Media CSS for almost 15 years, HTML — still poor browser support, usually used via PDF. HTML for printing is intermediary format. There is a great open source solutions to convert it to PDF — WeasyPrint and a number of a very good non opensource solutions. But still HTML is an intermediary format
- If somebody interested, probably most well-known are Antenna Publisher, Oxygen (the letter we used 20 years ago for authoring DocBook and converting to PDF via TeX)

3.1. Most widespread rendering approaches

- PDF ← native PDF-generating libraries
- PDF ← XSL-FO with FOP-processors
- PDF ← via TeX
- PDF ← HTML + Paged Media CSS
- DOCX/ODT, PDF ← +/- text processors (MS Word, LO Writer)

Warning
These technologies are not aligned in infinite details like: • Apache FOP has problems with Leader alignment (dots in a table of contents) • LO Writer doesn't support typography (like keep with next) within table cells • Microsoft doesn't recommend to run automation tasks (like saving PDF) on a server

---

- There are others, but I estimate these as 99% of all
- You can see Leader Problems in UniDoc Publisher documentation
- Native libraries examples: ReportLab in Python, Prawn in Ruby, PDFBox in Java
- (if somebody needs reference) https://support.microsoft.com/en-us/topic/considerations-for-server-side-automation-of-office-48bcfe93-8a89-47f1-0bce-017433ad79e2

4. Some brief conclusions

Warning
Feel like speleologist?

- The world of printing is the world of constraints
- And that constraints differ for each technology, you often need to support several chains (exquisitely looking PDF with TeX and LibreOffice for coordination)
- With no universal solutions

Two speleologist meet in a narrow tunnel. I'm from a dead end. I'm too!

- Documentation of UniDoc Publisher is created with Asciidoctor is published with WeasyPrint, XSL-FO and Open Document toolchains. Looks pretty the same. But just as a proof of concept

5. UniDoc Publisher approach suits best if at least `one of`

- You don't prepare documentation especially for printing purposes
- You are automating documentation generation and hope it will look good, no matter what will be generated
- You output format is one of text processing format

- First point is usually true if inputs — Simple markup/Wiki, main means of publication — HTML or static site

6. In search for flexibility: Asciidoctor open document

Automation on the writer side

1. Asciidoctor parses markup into AST (Abstract Syntax Tree)
2. You may transform AST with Asciidoctor AST processing
3. Asciidoctor runs writer template for each AST node recursively
4. You may write you code in pure Ruby or with special Slim templates

---

- First idea was to follow Asciidoctor ways, and these proved to be working ways

6.1. A simplified processing AST example

```
- !<OrderedList>
  roles:
  - "arabic"
  id: "ol-1"
  captioned_title:
      children:
      - !<Text>
        text: "Automation"
  children:
  - !<ListItem>
    children:
    - !<Paragraph>
      children:
      - !<Text>
        text: "Asciidoctor..."
  - !<ListItem>
...
```

```
- list_style = "#{get_basic_style}"
- if captioned_title?
  text:p text:style-name="#{list_style}"
    text:bookmark text:name="#{id}"
    =captioned_title

text:list text:style-name="#{list_style}"
  - items.each_with_index do |item, index|
```

- dash means code in Ruby
- get_basic_style — is some external function, defined as a helper, so Ruby can be used withing template

6.2. Great, but

- You can’t override part of a template
- You should invent styling approach

Styling? But text processors do support styling!

Warning

- `<span class="bold green">bold, green</span>` — impossible to apply two styles to one element

Tip

- Asciidoctor Open Document introduces some extended Open Document format to preserve Asciidoctor AST contents
- Each function checks, if style should be applied, and if yes, applies it

---

- In text processors we can’t apply two styles, so we can’t leave styling to text processor template
- This extended format resembles AST itself

6.3. And still

- Unexpectedly transforming this extended Open Document format became one of the most used feature of Asciidoctor Open Document
- Styling as separate task of writing proved also to be useful
- The Gradle was magnificent in gluing all parts together

- But we have example of Pandoc that places heavy attention on AST transformation. There is even pandocfilter project, that simplifies AST transformation for Python. I think the world meta-converter quite fits Pandoc. But using own converters raises compatibility issues and no special styling approach.
- Although this circus is glued with docker, my experiments showed, that gluing with Gradle is much more controllable. Yes, finally it would be docker. but at the very end

7. Thoughts before the second step

- If creating universal converter impossible…
- We should create meta converter — platform for building converters

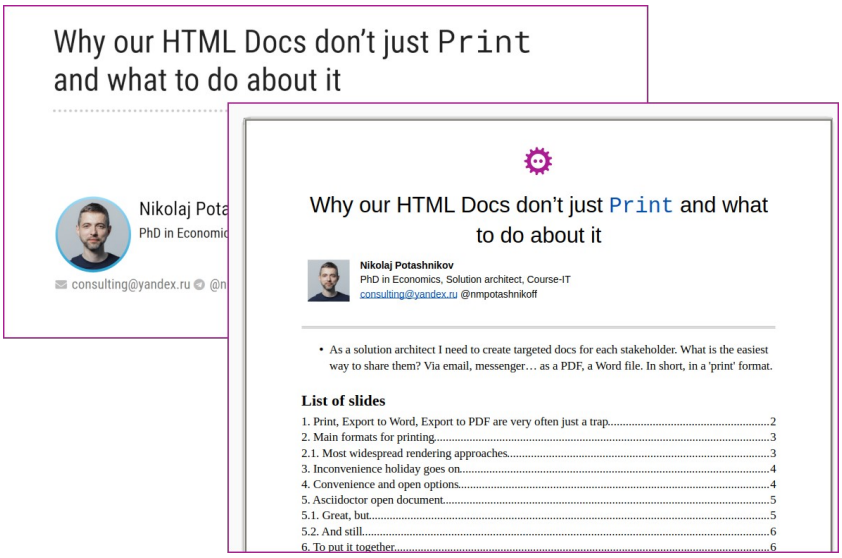
Estimated requirements

- Native converter as a reader
- Great ways of transforming AST
- A good approach for styling as a separate focus
- 99% generic writer
- Good integration with CI/CD

8. Native converter as a reader?

Tip
Each converter outputs HTML. HTML is quite semantic, why shouldn't we use it?

9. Let’s convert this presentation to LO Writer



This presentation as is shown in Ivan Ponomarev talk is created with Asciidoctor, so we can convert it to document with notes

10. Notes on this demo

- Everything is in a single `build.gradle.kts` file
- All Kotlin code examples are just includes from this `build.gradle.kts`

11. Boilerplate

```
FodtConverter {
    html = AsciidocHtmlFactory()
        .getHtmlFromFile(File("${project.projectDir}/$presentationFile.adoc"), true)
    template = File("${project.projectDir}/template-1.fodt").readText()
    adaptWith(AsciidoctorOdAdapter)
    unknownTagProcessingRule = unknownTagProcessingRuleRevealJs()
    parse()
        // Processing AST
    ast2fodt()
    File("${project.projectDir}/output/ast.yaml").writeText(ast().toYamlString())
    File("${project.projectDir}/output/$presentationFile-notes-
v$version.fodt").writeText(fodt())
}
```

Just stress

- That Asciidoctor is a part of a gradle build
- I started not from Asciidoctor HTML, but from Reveal.js Asciidoctor HTML to make notes more close to it and as a proof of constant, that technology have a safety margin against any HTML: Asciidoc HTML, Asciidoc Reveal.js HTML, MD, Wiki, ReST…
- And that it is quite minimalistic

12. Processing AST

```
ast().descendant { section ->
    section.sourceTagName == "section" &&
            section.descendant { it is Heading && it.level == 1 }
                .isNotEmpty()
}.first().also { it.insertBefore(makeTitle(it)) }.remove() (1)
```

Stress, that this is typical AST transforming code. Find nodes, and do smth with them.

Here we find section with H1 heading (title slide), and insert before it transformed version, then remove old version

13. Rearranging title (Asciidoc source)

```
|===
a|
[.title-photo]
image::images/nmp1.jpg[]
a|
[.full-name]
Nikolaj Potashnikov

[.bio]
PhD in Economics, Solution architect, Course-IT
.2+>.>a|{nbsp}
[.logo]
image::images/fosdem-logo.svg[]
2+a|
[.contact]
icon:envelope[] consulting@yandex.ru icon:telegram[]{nbsp}@nmpotashnikoff
|===
```

Stress that all elements has roles

14. Rearranging title, extracting semantics

```
val title = sourceNode.descendant { it is Heading && it.level == 1 }.first()
val notes = sourceNode.descendant { it.sourceTagName == "aside" }.first()
val (fullName, bio, photo, contact, logo) =
    arrayOf("full-name", "bio", "title-photo", "contact", "logo")
        .map { role -> sourceNode.descendant { it.roles.contains(role) }.first() }

logo.descendant { it is Image }.first().let { it as Image }
    .width = Length(1000F, LengthUnit.cmm)
photo.descendant { it is Image }.first().let { it as Image }
    .width = Length(1500F, LengthUnit.cmm)
```

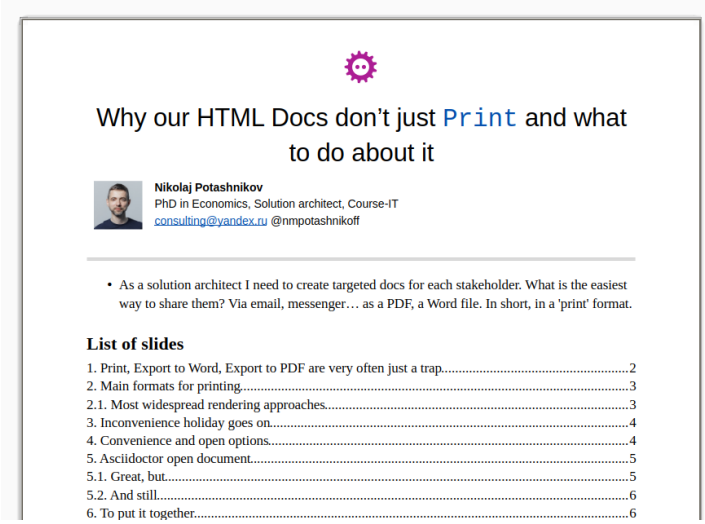
- stress that extracting looks like Xpath

15. Rearranging title, constructing title

```
appendChild(logo)
appendChild(title)
table {
    col(Length(18F)); col(Length(152F))
    roles("about-me")
    tableRowGroup(TRG.body) {
        tr {
            td { appendChild(photo) }
            td { arrayOf(fullName, bio, contact).forEach { appendChild(it) } }
        }
    }
}
appendChild(notes)
appendChild(Toc(2, "List of slides"))
normalizeImageDimensions()
```

Rearranging looks like slim (or XSLT) templates. Not so slim as Slim, but definitely slimmer than XSLT

16. Let’s return to the result



Why our HTML Docs don’t just `Print` and what to do about it

Nikolaj Potashnikov
PhD in Economics, Solution architect, Course-IT
consulting@yandex.ru @nmpotashnikoff

- As a solution architect I need to create targeted docs for each stakeholder. What is the easiest way to share them? Via email, messenger… as a PDF, a Word file. In short, in a 'print' format.

List of slides

1. Print, Export to Word, Export to PDF are very often just a trap....2
2. Main formats for printing....3
2.1. Most widespread rendering approaches....3
3. Inconvenience holiday goes on....4
4. Convenience and open options....4
5. Asciidoctor open document....5
5.1. Great, but....5
5.2. And still....6
6. To put it together....6

17. And a little bit of styling

```
OdtStyle { p ->
    if (p !is Paragraph) return@OdtStyle
    if (p.ancestor { it.roles.contains("logo") }.isEmpty()) return@OdtStyle
    attributes("style:master-page-name" to "First_20_Page")
},
OdtStyle { tableCell ->
    if (tableCell !is TableCell) return@OdtStyle
    if (tableCell.ancestor { it.roles.contains("about-me") }.isEmpty()) return@OdtStyle
    tableCellProperties {
        arrayOf("top", "right", "bottom", "left")
            .forEach { attributes("fo:border-$it" to "none") }
    }
},
```

Not a CSS, but the feeling resembles: select node, apply formatting

18. Let’s return to processing AST

```
ast().descendant { it.roles.contains("notes") } (1)
    .forEach { it.insertBefore(HorizontalLine()) }
ast().descendant { it is Heading && it.level > 1 }
    .forEach {
        it.insertBefore(
            Paragraph().apply { roles("slide-finish") }
        )
    } (2)
odtStyleList.add(odtStyles())
odtStyleList.add(rougeStyles()) (3)
```

1	Inserting horizontal lines before notes
2	Preventing slide from breaking
3	Applying custom style and rouge code highlighting styles

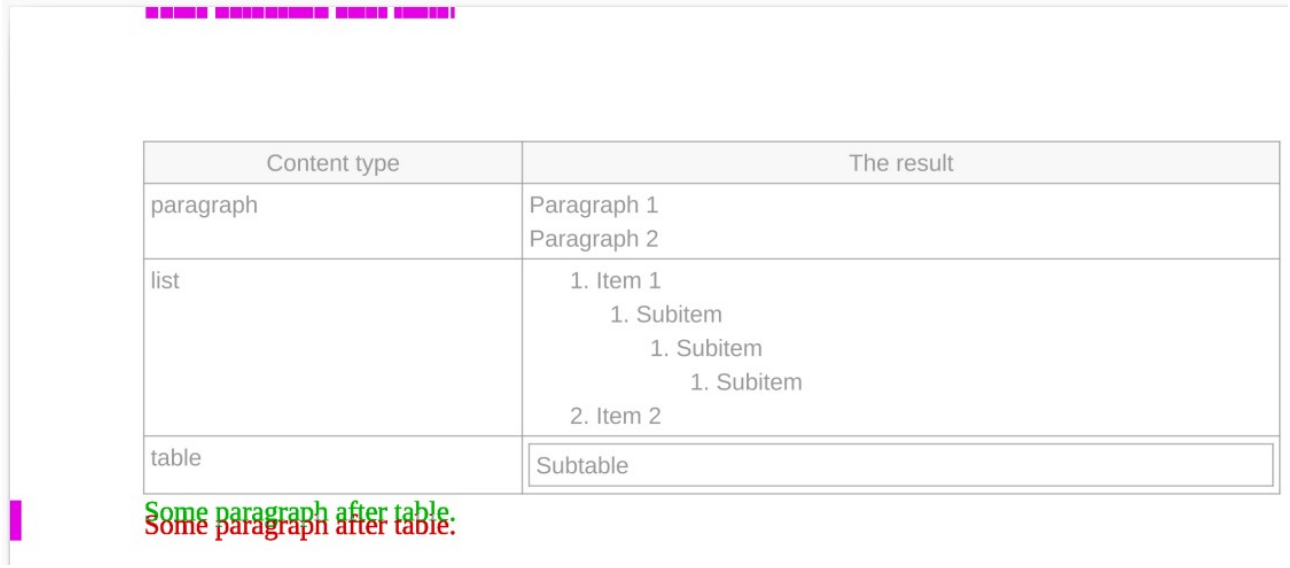
19. Extending AST

```
class HorizontalLine() : NoWriterNode() {
    override val isInline: Boolean get() = false
}
```

Extending AST is quite simple. But it is rarely used. This is artificial example, easier solution would be to add paragraph directly

```
OdtCustomWriter { horizontalLine ->
    if (horizontalLine !is HorizontalLine) return@OdtCustomWriter
    preOdNode.apply {
        "text:p" {
            attributes("text:style-name" to "Horizontal Line")
            process(horizontalLine)
        }
    }
},
```

19.1. Testing



If we have Asciidoctor with some Printing converter platform on top and Converter on top, we should be sure, that minor changes won’t break anything.

I use the system of snippets, render them into png and compare by pixels with reference. Almost 100% guaranty.

By the way these snippets can be put into documentation. These are real cases.

But it is already Ivan Ponomarev talk

20. Conclusion

- If you need to get doc in printing format, don’t pursue universality, limit, limit and limit
- Don’t search for solution with options, prepare to code
- Ecosystem is crucial. Gradle + Kotlin stack is just a coincidence, where syntax, wide ecosystem and CI accidentally met
- Still more coincidence, they met Asciidoctor
- Still more coincidence, they met LibreOffice API
- Too much coincidence for coincidence?

- There are slides with comparison, we can look at them personally after the talk

21. Questions?

22.1. Comparison

Technology	Strengths	Weaknesses
Native	• The technology is typically used by converters with a low entry barrier	• Most often has a unique configuration approach • High likelihood of encountering unexpected limitations
XSL-FO	• Time-tested technology, W3C standard	• Different processors yield different results (e.g., varying typography limitations) • Better to stick to FO; XSLT is outdated and hard to maintain • Slow build performance (at least with Apache FOP)

23. Comparison #2

Technology	Strengths	Weaknesses
TeX	• Best-in-class typography, suitable for mass printing • Mature, albeit complex, ecosystem	• …albeit complex ecosystem
Paged Media CSS	• W3C standard • Takes HTML as input, so it’s practically applicable everywhere	• Browser support remains very limited • Among free tools, I would recommend only WeasyPrint

24. Comparison #3

Technology	Strengths	Weaknesses
Word processors	• Format is designed for manual editing of the output document • Fastest way to produce output (even when PDF is required)	• Only LibreOffice aligns with the DocOps paradigm • Imposes numerous constraints, including on typography • Saving to a word processor format almost always requires extensive custom adjustments and limitations