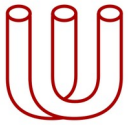

UniDoc Publisher

The idea

If you are tired of struggling with lengthy documentation transformation pipelines, juggling multiple technological stacks just to get decent ODT, DOCX or PDF output, try UniDoc Publisher, the simple and powerful document publishing solution for DocOps engineers!

UniDoc Publisher is a highly customizable Kotlin library that provides a comprehensive set of tools to parse HTML files into abstract syntax tree (AST), transform it in any desirable way and render the Open Document flat format (FODT). The latter you can convert to PDF or DOCX with LibreOffice.

UniDoc Publisher is heavily influenced by Pandoc. Unlike Pandoc, UniDoc Publisher allows you to perform almost all necessary steps within a single Kotlin script. Plus, UniDoc Publisher's heavy reliance on Kotlin DSL constructions means you'll have unlimited power to customize output on the parser, transformation, and writer sides.

True, UniDoc Publisher accepts only HTML as a source but all text markups produce HTML and some common markups, such as Asciidoc and Markdown have Java converters that enable you to get the source HTML within this same Kotlin script.

UniDoc Publisher has been designed with ease of use in mind, so you can focus on getting the job done without getting bogged down in complicated syntax or lengthy code.

UniDoc Publisher supersedes Open Document converter for Asciidoctor. Despite now it is designed to publish any text markup, Asciidoc IMHO is the best and most universal simple text markup with a very well-designed and extensible converter (Asciidoctor), large ecosystem and a great community.

The prefix [Uni] means UniDoc Publisher (1) is quite [uni]versal (almost all markups can output HTML) (2) can [uni]te several sources of documentation.

The goal of MVP

The goal of MVP is to test the UniPublisher underlying ideas and implementation approach.
I would be grateful for any critical comments.

Examples

1. example/mvp-doc/mvp-doc.main.kts

   Creates FODT file for this document from Asciidoc file /doc/pages/mvp.adoc.

2. example/ps-118/convert-to-pdf.main.kts

   Parses HTML-files and unites them in one book of A4 format (fodt, pdf), and electronic book format (fodt, pdf), adjusting formatting for each format.

The text contains Holy Father Interpretaions for Psalm 118, taken from https://bible.optina.ru/. The site is made with the DokuWiki engine.

3. example/builder/table.main.kts

   Builds the letter from JSON data and converts it to fodt, odt, pdf and docx formats. Adapted from Pandoc documentation.

4. example/customize-everything/customize-everything.main.kts

   Shows, how to customize all parts of the converter: reader (parser), model and writer. But the project has many local extension points, so the approach of customizing everything looks excessive in most cases.

To run examples you need the Kotlin compiler.

All examples in the current project are built with build-all.sh

```
./gradlew publishToMavenLocal
mvn install:install-file -Dfile=example/ps-118/JHyphenator-1.0.jar -DgroupId=mfietz \
  -DartifactId=jhyphenator -Dversion=1.0 -Dpackaging=jar
rougify style github > example/mvp-doc/syntax.css

rm -f example/ps-118/output/*
kotlin example/ps-118/convert-to-pdf.main.kts && \
  lo-kts-converter/lo-kts-converter.main.kts \
    -i example/ps-118/output/ps-118.fodt -f pdf && \
  lo-kts-converter/lo-kts-converter.main.kts \
    -i example/ps-118/output/ps-118-ebook.fodt -f pdf

rm -f example/builder/output/* && \
  kotlin example/builder/table.main.kts && \
  lo-kts-converter/lo-kts-converter.main.kts \
    -i example/builder/output/letter.fodt -f pdf,odt,docx

rm -f example/mvp-doc/output/* && \
  kotlin example/mvp-doc/mvp-doc.main.kts && \
  lo-kts-converter/lo-kts-converter.main.kts \
    -i example/mvp-doc/output/mvp-doc.fodt -f pdf
```

The main features of U-Pub

HTML as a source format

HTML format is produced by almost all native markup converters, and it is quite semantic in itself.

The result produced by those native converters is usually very similar, so we can create a universal reader with minimal customization for each markup language. Moreover, HTML is produced by exactly the native converter, so we are sure, that the source markup is correctly processed.

If HTML doesn't contain some necessary data, we can usually tune the native converter. For example, Asciidoc has an attribute `pdfwidth` that shouldn't necessarily be put into HTML. But we can force it to HTML extending the converter.

Styles for HTML and printing format may differ significantly, still we often need to reuse at least part of CSS styles in a printing document. For example, the code in this document is highlighted with Rouge. The styles are produced directly from the CSS, created with `rougify` command.

Easily manipulated AST

AST transformation with Kotlin is very easy, and along with the styling mechanism, it is suggested as the main tool for customizing output. Don't define sections for numerating chapters, just numerate them. Don't define if the image caption should be above or below, just place it there. Don't define the way the title page should look, just create it the way you need.

In the script, that converts current document to PDF, HTML is produced with Asciidoctor. Asciidoctor in some cases wraps paragraphs into div. If the paragraph has an id it will be inserted into such a div. To correctly process such a situation we should either override the default paragraph writer or just transfer id from `<div>` to wrapped `<p>`.

```
ast().descendant { it is OpenBlock && it.roles.contains("paragraph") }.forEach {
    it.children()[0].id = it.id
    it.id = null
}
```

We can also build an AST from scratch:

```
val letterAst = Document().apply {
    p { +"Dear Boss:" }
    p { +"Here are the CNG stations that accept Voyager cards:" }
    table {
        repeat(3) { col(Width(1F)) }
        tableRowGroup(TRG.head) {
            tr {
                arrayOf("Station", "Address", "Cards Accepted")
                    .forEach { td { p { +it } } }
            }
        }
        stations.forEach { station ->
            tr {
                arrayOf(
                    station.stationName, station.streetAddress,
                    station.cardsAccepted
                ).forEach { td { p { +it } } }
            }
        }
    }
    p { roles("signature"); +"Your loyal servant" }
    p { +"John Hancock" }
    // TODO: img tag
}
```

Kotlin script as the main automation language

- Kotlin script is a standalone file, that contains dependency declarations. It can be run with the command `kotlin [kotlin script file]`.
- Kotlin is a statically typed language. Unlike DevOps scripts, DocOps scripts are more voluminous, and are more often modified as we change documentation structure rather often. Static typing contributes greatly to the maintainability of such scripts.
- Kotlin is a wrapper language for Java, so you may use the whole Java ecosystem.
- The project can be ported to Kotlin for JavaScript and probably to Kotlin Native.
- Kotlin has an excellent support for builder and templating constructions, that that allows us to tune the outlook of resulting document by transforming AST in a desirable way. It is a good deal simpler than customizing the outlook with a huge number of options that nobody remembers.

Using for publishing consideration

If you would like to use LibreOffice Writer for publishing consider the following issues.

1. LibreOffice ignores paragraph typography settings like window/orphan control or keeping together in table cells. With UniDoc Publisher you may style last row in a table to be unbreakable, but

Extension points

1. Unknown HTML tags can be parsed into a generic node with attributes, put in a `Map`. This generic node can be replaced with some specific node during the AST transformation.
2. Two types of styles are supported:
   - Output nodes style

     The following example stylizes a common table cell. The bottom padding is left with 0 value, because the indent after the in-cell last paragraph will create the necessary padding.

     ```
     OdtStyle { node ->
         if (node !is TableCell) return@OdtStyle
         tableCellProperties {
             arrayOf("top", "right", "bottom", "left").forEach {
                 attribute("fo:border-$it", "0.5pt solid #000000")
                 if (it != "bottom") attribute("fo:padding-$it", "1mm")
             }
         }
     },
     ```

   - Writers from scratch

     In the following example a writer for some custom PageRef node is defined. This writer outputs the page number of the referenced element.

     ```
     CustomWriter {
         if (it !is PageRef) return@CustomWriter
         preOdNode.apply {
             "text:bookmark-ref" {
                 attribute("text:reference-format", "page")
                 attribute("text:ref-name", it.href)
                 // The value to be shown if fields are not updated
                 -"-"
             }
         }
     },
     ```

3. Postprocessing. In the following example the page and footer parameters are adjusted for an electronic book view.

   ```
   fodtGenerator?.enrichedTemplate?.apply {
       xpath("//style:page-layout-properties").iterable()
           .map { it as Element }.forEach { el ->
               arrayOf(
                   "page-width" to "105mm", "page-height" to "148mm",
                   "margin-top" to "2mm", "margin-right" to "2mm",
                   "margin-bottom" to "3mm", "margin-left" to "2mm",
               ).forEach { el.setAttributeNS(foNS, it.first, it.second) }
           }
       xpath("//style:footer-style/style:header-footer-properties").iterable()
           .map { it as Element }
   ```

```
        .forEach { element -> element.setAttributeNS(foNS, "margin-top", "2mm") }
}
```

4. You can also customize everything as in this example.