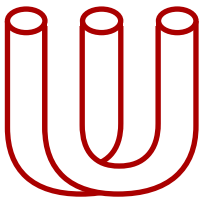

UniDoc Publisher

Table of contents

UniDoc Publisher is a highly customizable Kotlin library that provides a comprehensive set of tools to parse HTML files into an 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.

Table 1. Status

Maven Central	version 0.9.2

How to run

Uni-Doc Publisher is just a `.jar`, published to Maven Central. The most common way to use it is from Kotlin script or as a part of a gradle pipeline that is almost the same.

As an example you may download the Kotlin script file. It is used to build the current documentation (prerequisites: Java 11+ and Kotlin 19+). The script runs with the following CLI parameters.

```
Usage: cli-options [<options>]

Options:
  --adoc-file=<text>    File to process (required)
  --template=<text>     Template path (required)
  --fodt-output=<text>  Fodt output file (required)
  --yaml-output=<text>  Yaml output file
  --html-output=<text>  Html output file
  --check-spelling      Check spelling
  --logo=<text>         Path to logo
  -h, --help            Show this message and exit
```

The script uses FODT converter in the following way:

```
FodtConverter {
```

```
    adaptWith(AsciidoctorAdapter) // (1)
    template = File(CliOptions.template).readText() // (2)
    html = AsciidocHtmlFactory
        .getHtmlFromFile(File(CliOptions.adocFile)) // (3)
    odtStyleList.add(rougeStyles()) // (4)
    odtStyleList.add(indentPreamble()) // (5)
    parse() // (6)
    if (CliOptions.logo != null) addLogoToAst() // (7)
    ast2fodt() // (8)
    if (CliOptions.checkSpelling) checkSpelling() // (9)
    CliOptions.htmlOutput
        ?.let { File(it).writeText(html()) } // (10)
    CliOptions.yamlOutput
        ?.let { File(it).writeText(ast().toYamlString()) } // (11)
    File(CliOptions.fodtOutput).writeText(fodt()) // (12)
}
```

1. Applying Asciidoctor adapter
2. Loading template. You may take this template as a starting point
3. Converting source file from Asciidoc to HTML
4. Adding rouge styles for highlighting code in addition to basic styles from `AsciidoctorAdapter`
5. Indenting preamble
6. Parsing HTML into AST
7. Adding logo to AST if necessary
8. Converting AST to FODT
9. Checking spelling if necessary
10. Outputting HTML if necessary
11. Outputting AST as YAML if necessary
12. Outputting FODT

Warning
The Kotlin script file name should always end with `main.kts`.

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 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. For Asciidoc a basic adapter is already included in the library.

UniDoc Publisher has been designed with ease of use in mind, 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 it is now 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) and (2) can **uni**te several sources of documentation.

The main features of UniDoc Publisher

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.

In order to simplify using UniDoc Publisher with various markups, an adapter mechanism is implemented. Adapter is just a number of parsing rules, AST transformations, and styling rules that can be extended or changed in any part.

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 rules for numerating chapters and sections, 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.

It is also possible to 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(Length(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" }
    p {
        id = "john-hancock-signature"
        img(src = "${__FILE__.parent}/JohnHancock.png") {
            width = Length(40F, LengthUnit.mm)
        }
    }
}
```

Kotlin script as the main automation language

- Kotlin script may be used as a standalone runnable 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 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.

Considerations for using UniDoc Publisher

As a report builder

UniDoc Publisher was created with the report building purpose in mind. That's why I tried to keep the number of dependencies as small as possible. But it was not optimized in any way, so before using it please check, whether its performance fits your requirements.

As a validation tool

Validation is a very considerable issue that was kept in mind while designing UniDoc Publisher. The examples show the approach for a document validation.

For now, UniDoc Publisher doesn't support source mapping. It will be implemented for Asciidoctor Adapter in the nearest future.

As a publishing tool

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

1. LO Writer ignores paragraph typography settings like window/orphan control or keep with next paragraph within table cells. With UniDoc Publisher you may style last row in a table to be unbreakable, but nothing more.
2. You can't make a table row keep together with the next row. You can't even prevent a cell from breaking if it spans several rows.
3. LibreOffice Writer makes a poor looking table of contents if the text heading can't be put in one line. Still with UniDoc Publisher you may automatically create your own table of contents which will look the proper way.
4. If you need to convert the document to MS Word, some tuning is needed. For example MS Word doesn't support margin after table. That's why for MS Word documents you should either define margin before the next paragraph (with UniDoc Publisher styling approach) or, alternatively, put after each table zero height paragraph with a margin after it.

5. Even minor changes in LibreOffice may lead to considerable changes in the way the document is rendered. For example the letter positioning in version 7.4 (marked with green) differs from the letter positioning in version 7.6 (marked with red).

Figure 1. Failed test after increasing LibreOffice version from 7.4 to 7.6

Extension points

1. FODT (Flat Open Document) templates allow including the whole converted content (inserted instead of variable `include` set to `all`) or content defined by special CSS classes like `tag--[entry-point-name]` (inserted instead of variable `include` set to `[entry-point-name]`). More examples are given in `TestContentPoints.kt`. All template content, placed after `process` variable set to `end` is ignored.
2. 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.
3. 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
            -"-"
        }
    }
},
```

4. Post-processing. 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") }
}
```

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

Examples

1. doc/build-doc.main.kts

   Creates this documentation from the Asciidoc file /doc/pages/unidoc-publisher.adoc.

2. example/ps-118/ps-118.main.kts

   Parses HTML-files and unites them into one book of A4 format (FODT, PDF, DOCX), and the electronic book format (FODT, PDF), adjusting the page layout for each format.

   The text contains Holy Father Interpretations for Psalm 118, taken from https://bible.optina.ru/ in the HTML format. 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/builder/writerside.main.kts

   Converts “Export to PDF” page from Writerside tutorial into FODT, ODT, PDF and DOCX formats.

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

   Shows, how to customize all parts of the converter: reader (parser), model and writer.

   The project has many extension points, so the approach of customizing everything looks excessive in most cases.

To run examples you need the Kotlin compiler. To convert FODT file to PDF, ODT, DOCX format a Kotlin script LibreOffice wrapper is used. But you may do the conversion manually or use any LibreOffice wrapper converter of your choice like JOD Converter.

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

```
echo build-doc
kotlin doc/build-doc.main.kts -h > doc/autopartials/example-cli-help.txt # (1)
 rm -rf doc/output && mkdir doc/output
kotlin doc/build-doc.main.kts \
  --adoc-file doc/pages/unidoc-publisher-doc.adoc \
  --template approved/asciidoc/template-1.fodt \
  --fodt-output doc/output/unidoc-publisher-doc.fodt \
  --yaml-output doc/output/unidoc-publisher-doc.yaml \
  --html-output doc/output/index.html \
  --logo doc/images/unidoc-processor-symbol.svg \
  --check-spelling
 cp doc/images -r doc-output
lo-kts-converter/lo-kts-converter.main.kts \
  -i doc/output/unidoc-publisher-doc.fodt -f pdf,odt,docx

echo ps-118
mvn install:install-file -Dfile=example/ps-118/JHyphenator-1.0.jar \
   -DgroupId=mfietz -DartifactId=jhyphenator -Dversion=1.0 -Dpackaging=jar # (2)
 rm -rf example/ps-118/output && mkdir example/ps-118/output
```

```
kotlin example/ps-118/ps-118.main.kts
lo-kts-converter/lo-kts-converter.main.kts \
  -i example/ps-118/output/ps-118.fodt -f pdf,odt,docx
lo-kts-converter/lo-kts-converter.main.kts \
  -i example/ps-118/output/ps-118-ebook.fodt -f pdf

echo builder
rm -rf example/builder/output && mkdir 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

echo writerside-tutorial
rm -rf example/writerside-tutorial/output && mkdir example/writerside-tutorial/output
kotlin example/writerside-tutorial/writerside.main.kts
lo-kts-converter/lo-kts-converter.main.kts \
  -i example/writerside-tutorial/output/export-to-pdf.fodt -f pdf,odt,docx
```

1. Generating this documentation builder script CLI help
2. Installing hyphenation `.jar` into a local Maven repository

License