Tabulate

Table of Contents

Introduction

Exporting data to external file formats can be tedious and cumbersome - especially when business wants to have reports covering vast majority of system functionalities. Writing every exporting method using imperative API directly will soon make code verbose, error prone, hard to read and maintain. In such cases You want to hide implementation details using abstractions, but this is additional effort which is not desirable.

`Tabulate` tries to mitigate above problems with the help of `Kotlin`, its `type-safe DSL builders` and `extension functions`.

Key concepts

Table model with attributes.

Table model describes how table will look like after data exporting. Its building blocks are:

- `column` - defines a single column in table,
- `row` - may be user defined custom row or row that carries attributes for enriching existing record,
- `row cell` - defines cell within row. Cell is bound to a column via column id,
- `attribute` - introduces extensions to basic presentation capabilities.

Table model is internal concept and is not exposed to API consumers (Only attribute model can be exposed as it is fully customisable). Table is always constructed using table builders as follows:

```
productList.tabulate("file.xlsx") {
    name = "Table id" ①
    columns {   ②
        column("nr")
    }
    rows { ③
        row {  // first row when no index provided.
            cell("nr") { value = "Nr.:" } ④
        }
    }
}
```

① Firstly we give the table name. It can be used by exporter e.g. to add metadata like sheet name.

② Secondly we can provide column definitions. Column definition can be used to aggregate `ColumnAttributes` as well as `CellAttributes`. All attributes associated with particular column will apply to each cell in that column. Specifying column can also help to make table layout more readable.

③ Next step is to define table rows. Here we can create additional custom rows (like header or footer) or enhance table look and feel with attributes associated with particular row.

④ Each row can contain as many cells as many columns exist. Similarily to `row`, `cell` may be used to associate cell attributes with selected cell within row. You can also create cell with custom predefined or computed value.

Above, we have created table definition with single column and one row with single cell. Cell binds to column by column identifier which in our case is represented by simple text id "nr".

This is very basic example. In order to gain more powers You will need to start using `attributes`.

`Attributes` are plain objects with inner properties that extends base model. Attributes can be mounted on multiple levels: *table*, *column*, *row* and single *cell* levels.

Example with attributes included:

```
productList.tabulate("file.xlsx") {
    name = "Table id"
    attributes {
      filterAndSort {} ①
    }
    columns {
        column("nr") {
            attributes { width { px = 40 }} ②
        }
        column(Product::code) {
            attributes { width { auto = true}}
            attributes {
                text {
                    weight = DefaultWeightStyle.BOLD ③
                }
            }
        }
    }
    rows {
        row {  // first row when no explicit index provided.
              cell("nr") {
                value = "Nr.:"
                attributes {
                  text { ④
                    fontFamily = "Times New Roman"
                    fontColor = Colors.BLACK
                    fontSize = 12
                  }
                  background { color = Colors.BLUE }
                }
              }
        }
    }
}
```

① Top level table attribute `TableAttribute` applicable only for excel file format

② Column level `ColumnAttribute` that defines width of entire column

③ Column level `CellAttribute` - an attribute applicable for every cell in particular column.

④ Cell level attribute. This is the lowest level. Only `CellAttribute` can be used on that level.

Table DSL type-safe builders API.

Kotlin type-safe DSL builders API helps a lot with describing table structure. It makes source code look more concise and readable and makes maintenance tasks much easier due to DSL type-safety. At coding time, your IDE will make use of it by offering completion hints - this elevates developer experience, as almost zero documentation is required to start.

DSL functions take `lambda with receivers` as arguments which abstracts away internal API instantiation details from consumers. Within lambda, you can call API methods which can take downstream builders as arguments. We can end up having multi-level DSL API structure, where each level is extensible via Kotlin extension functions. On each DSL level You are allowed to invoke scope accessible methods and access variables which can lead to interesting results:

```
val additionalProducts = ... ①
tabulate {
      name = "Products table"
      attributes {
        template { fileName = "src/test/resources/template.xlsx" } ②
      }
      rows {
          header("Code", "Name", "Description", "Manufacturer") ③
          additionalProducts.forEach { ④
              row {
                  cell { value = it.code }
                  cell { value = it.name }
                  cell { value = it.description }
                  cell { value = it.manufacturer }
              }
          }
      }
}.export("products.xlsx")
```

① Here we are using `additionalProducts` val which is collection of elements to be exported.

② Then we are specifying a template file on which interpolation should take place.

③ Define header as long as we know that our template doesn't mention it.

④ Finally, we are iterating over collection elements to build static table model.

Although it is possible to build row definitions by iterating on collection directly, you should always prefer to use Column bound cell value extractors.. They are much faster and consume much less memory than approach shown in point number 4.

Table DSL type-safe builders extension functions.

As already said, it is possible to extend each DSL level by using extension functions on DSL API builder classes.

Take the example from previous section:

```
    tabulate {
          rows {
              header("Code", "Name", "Description", "Manufacturer")
          }
    }.export("products.xlsx")
```

Function `.header` is implemented as follows:

```
fun <T> RowsBuilderApi<T>.header(vararg names: String) =
    newRow(0) { ①
        cells {
            names.forEach {
                cell { value = it }
            }
        }
    }
```

① Calling `.newRow(0)` `RowsBuilderApi` method internally ensures that `.header` extension function always defines custom row at index `0`.

This way you can create various shortcuts and templates, making DSL vocabulary richer and more expressive. It is worth mentioning that by using extension functions on DSL builders - scope becomes restricted by DslMarker annotation, so it is not possible to break table definition by calling methods from upstream builders.

Column bound cell value extractors.

Column API makes it possible to pass property getter reference as a column key. This creates object to column binding that is applied later at run time for cell value evaluation.

```
productsRepository.loadProductsByDate(now()).tabulate("file/path/products.xlsx") {
            name = "Products table"
            columns {
                column(Product::code)
                column(Product::name)
                column(Product::description)
            }
        }
```

Property getter as column key kills two birds with one stone:

- It allows to reference column later in cell builder,
- it allows to extract collection element property value when row context is built for rendering.

Row predicates.

You have already seen how `.header` extension function is implemented. Internally it invokes `.newRow(0)` which requests rendering of a row at index `0`. What if You want to apply entire row definition for several indices ? You may repeat `.newRow()` invokation as many times as required, but there is better option. You can use row index predicate as follows:

```
atIndex { gt(0) and lt(100) } newRow { ①
    cell { expression = RowCellExpression { "index : ${it.rowIndex.getIndex()}" } } ②
}
```

① On first line we have a construct built mainly from infix functions. `atIndex { ⋯ }` takes row index predicate `gt(0) and lt(100)` which literally says: 'Apply this row definition to all indices between index 0 and index 100'. Last 'keyword' sounds: `newRow` and delivers row definition from within curly braces.

② This line represents definition of a row which is about to be created for each matching index. It contains single cell with runtime expression evaluated at context rendering time.

There is also alternative notation used to achieve the same result:

```
newRow({ gt(0) and lt(100) }) {
    cell { expression = RowCellExpression { "index : ${it.rowIndex.getIndex()}" } }
}
```

Extension points.

I have put lots of effort to make **Tabulate** extensible. Currently, it is possible to:

- add user defined attributes,
- add custom renderers for already defined attribute,
- implement table export operations from scratch (e.g. html table, cli table, mock renderer for testing),
- extend DSL type-safe builder APIS on all possible level.

Implementing new table export operations.

In order to support new tabular file format you have to extend `ExportOperationsConfiguringFactory<C, T, O>` where:

- `C` stands for rendering context - which is usually wrapper around 3rd party api like Apache POI,
- `T` stands for object class representing element of exported collection,
- `O` stands for type of result of operation (e.g. `OutputStream` for Apache POI)

As long as tabulate uses java ServiceLoader infrastructure, You need to create file `resource/META-INF/io.github.voytech.tabulate.template.spi.ExportOperationsProvider`, and put fully qualified class name of your custom factory in the first line. **This step is required by a template in order to resolve your extension at run-time**.

Basic CSV implementation looks like this:

```
①
open class CsvRenderingContext: RenderingContext {
    private lateinit var bufferedWriter: BufferedWriter
    private val line = StringBuilder()

    fun doBind(output: OutputStream) {
        bufferedWriter = output.bufferedWriter()
    }

    fun startRow() {
        line.clear()
    }

    private fun AttributedCell.getSeparatorCharacter(): String =
        attributes?.get(CellSeparatorCharacterAttribute::class.java)?.separator ?: ","

    fun <T> endRow(context: AttributedRowWithCells<T>) {
        val lastIndex = context.rowCellValues.size - 1
        context.rowCellValues.values.forEachIndexed { index, cell ->
            line.append(cell.value.value.toString())
            if (index < lastIndex) line.append(cell.getSeparatorCharacter())
        }
        bufferedWriter.write(line.toString())
        bufferedWriter.newLine()
    }

    fun finish() {
        bufferedWriter.close()
    }
}
```

① `CsvRenderingContext` implements `RenderingContext` marker interface and provides logic and state responsible for generating table in selected format. It is a common denominator used as argument of all export operation methods in order to share rendering state and allow interaction with it.

```
class CsvOutputStreamOutputBinding : OutputStreamOutputBinding<CsvRenderingContext>()
{

    override fun onBind(renderingContext: CsvRenderingContext, output: OutputStream) {
①
        renderingContext.doBind(output)
    }

    override fun flush(output: OutputStream) { ②
        renderingContext.finish()
        output.close()
    }
}
```

① The `.onBind` method wires particular rendering context instance with actual output representation (in this particular case it is an OutputStream). Method will be invoked internally by `TabulationTemplate` as soon as both: output and rendering context instances are available.

② The `.flush` dumps in-memory rendering context representation into output.

```
class CsvExportOperationsFactory: ExportOperationsProvider<CsvRenderingContext> {

    override fun getContextClass(): Class<CsvRenderingContext> =
CsvRenderingContext::class.java ①

    override fun createRenderingContext() = CsvRenderingContext()  ②

    override fun supportsFormat(): TabulationFormat = format("csv") ③

    ④
    override fun createExportOperations():
AttributedContextExportOperations<CsvRenderingContext> = object :
AttributedContextExportOperations<CsvRenderingContext> {

        override fun beginRow(renderingContext: CsvRenderingContext, context:
AttributedRow) {
            renderingContext.startRow()
        }

        override fun renderRowCell(renderingContext: CsvRenderingContext, context:
AttributedCell) { }

        override fun <T> endRow(renderingContext: CsvRenderingContext, context:
AttributedRowWithCells<T>) {
            renderingContext.endRow(context)
        }
    }

    ⑤
    override fun createOutputBindings(): List<OutputBinding<CsvRenderingContext, *>> =
listOf(CsvOutputStreamOutputBinding())

}
```

① 1

② 2

③ 3

④ 4

⑤ 5

If target tabular format supports styles, You may add support for rendering built-in attributes as follow:

```
class ExampleExportOperationsConfiguringFactory<T> :
ExportOperationsConfiguringFactory<T, SomeRenderingContext>() {

  ..
  override fun getAttributeOperationsFactory(renderingContext: SomeRenderingContext):
AttributeRenderOperationsFactory<T> =
      StandardAttributeRenderOperationsFactory(renderingContext, object:
StandardAttributeRenderOperationsProvider<ApachePoiExcelFacade,T>{
          override fun createTemplateFileRenderer(renderingContext:
ApachePoiExcelFacade): TableAttributeRenderOperation<TemplateFileAttribute> =
            TemplateFileAttributeRenderOperation(renderingContext)

          override fun createColumnWidthRenderer(renderingContext:
ApachePoiExcelFacade): ColumnAttributeRenderOperation<ColumnWidthAttribute> =
            ColumnWidthAttributeRenderOperation(renderingContext)

          override fun createRowHeightRenderer(renderingContext:
ApachePoiExcelFacade): RowAttributeRenderOperation<T, RowHeightAttribute> =
            RowHeightAttributeRenderOperation(renderingContext)

          override fun createCellTextStyleRenderer(renderingContext:
ApachePoiExcelFacade): CellAttributeRenderOperation<CellTextStylesAttribute> =
            CellTextStylesAttributeRenderOperation(renderingContext)

          override fun createCellBordersRenderer(renderingContext:
ApachePoiExcelFacade): CellAttributeRenderOperation<CellBordersAttribute> =
            CellBordersAttributeRenderOperation(renderingContext)

          override fun createCellAlignmentRenderer(renderingContext:
ApachePoiExcelFacade): CellAttributeRenderOperation<CellAlignmentAttribute> =
            CellAlignmentAttributeRenderOperation(renderingContext)

          override fun createCellBackgroundRenderer(renderingContext:
ApachePoiExcelFacade): CellAttributeRenderOperation<CellBackgroundAttribute> =
            CellBackgroundAttributeRenderOperation(renderingContext)
      })
}
```

Factory class `StandardAttributeRenderOperationsFactory` exposes API which assumes specific standard library attributes. If your file format allow additional attributes which are not present in standard library (tabulate-core), you may use `AttributeRenderOperationsFactory` interface directly, or fill additional constructor properties on `StandardAttributeRenderOperationsFactory` as below:

```
class ExampleExportOperationsConfiguringFactory<T> :
ExportOperationsConfiguringFactory<T,SomeRenderingContext>() {

  ...
  override fun getAttributeOperationsFactory(renderingContext: SomeRenderingContext):
AttributeRenderOperationsFactory<T> =
      StandardAttributeRenderOperationsFactory(renderingContext, object:
StandardAttributeRenderOperationsProvider<SomeRenderingContext,T>{
          override fun createTemplateFileRenderer(renderingContext:
SomeRenderingContext): TableAttributeRenderOperation<TemplateFileAttribute> =
TemplateFileAttributeRenderOperation(renderingContext)
      },
        additionalCellAttributeRenderers = setOf( .. )
        additionalTableAttributeRenderers = setOf( .. )
      )
}
```

Registering new attribute types for existing export operations.

It is possible that you have requirements which cannot be achieved with standard set of attributes, and your code is in different compilation unit than specific table export operation implementation. Assume You want to use existing Apache POI excel table exporter, but there is lack of certain attribute support. In such situation - You can still register attribute by implementing another service provider interface - `AttributeRenderOperationsProvider`:

```
class CustomAttributeRendersOperationsProvider<T> :
AttributeRenderOperationsProvider<T,ApachePoiExcelFacade> {

    override fun getContextClass() = ApachePoiExcelFacade::class.java

    override fun getAttributeOperationsFactory(creationContext: ApachePoiExcelFacade):
AttributeRenderOperationsFactory<T> {
        return object : AttributeRenderOperationsFactory<T> {
            override fun createCellAttributeRenderOperations():
Set<CellAttributeRenderOperation<out CellAttributeAlias>> =
                setOf(MarkerCellAttributeRenderOperation(creationContext))
        }
    }
}
```

After creating factory - You need to implement particular attribute together with DSL API extension function and attribute render operation to instruct 3rd party Apache Poi API on how to proceed.

```
data class MarkerCellAttribute(val text: String) :
CellAttribute<MarkerCellAttribute>() {

    class Builder(var text: String = "") : CellAttributeBuilder<MarkerCellAttribute> {
        override fun build(): MarkerCellAttribute = MarkerCellAttribute(text)
    }
}

class SimpleMarkerCellAttributeRenderOperation(poi: ApachePoiExcelFacade) :
    AdaptingCellAttributeRenderOperation<ApachePoiExcelFacade,
SimpleTestCellAttribute>(poi) {

    override fun attributeType(): Class<out MarkerCellAttribute> =
MarkerCellAttribute::class.java

    override fun renderAttribute(context: RowCellContext, attribute:
MarkerCellAttribute) {
        with(adaptee.assertCell(
            context.getTableId(),
            context.rowIndex,
            context.columnIndex
        )) {
            this.setCellValue("${this.stringCellValue} [ ${attribute.label} ]")
        }
    }

}

fun <T> CellLevelAttributesBuilderApi<T>.label(block: MarkerCellAttribute.Builder.()
-> Unit) =
    attribute(MarkerCellAttribute.Builder().apply(block))
```

Finally, You need to create file `resource/META-INF/io.github.voytech.tabulate.template.spi.AttributeRenderOperationsProvider`, and put fully qualified class name of our factory in it.

Extending Table DSL API

In the last section You saw how to define custom user attributes. The last step involves creating extension function on specific DSL attribute API. As DSL builder class name suggests (`CellLevelAttributesBuilderApi<T>`) this builder is part of a Cell DSL API only , which means that it won't be possible to add this attribute on row, column and table. You can leverage this behaviour for restricting say 'mounting points' of specific attributes. In order to enable cell attribute on all levels You will need to add more extension functions:

```
fun <T> ColumnLevelAttributesBuilderApi<T>.label(block: MarkerCellAttribute.Builder.()
-> Unit) =
    attribute(MarkerCellAttribute.Builder().apply(block).build())
fun <T> RowLevelAttributesBuilderApi<T>.label(block: MarkerCellAttribute.Builder.() ->
Unit) =
  attribute(MarkerCellAttribute.Builder().apply(block).build())
fun <T> TableLevelAttributesBuilderApi<T>.label(block: MarkerCellAttribute.Builder.()
-> Unit) =
  attribute(MarkerCellAttribute.Builder().apply(block).build())
```

Now You can call `label` on all DSL API levels in `attributes` scope like:

```
productList.tabulate("file.xlsx") {
    name = "Table id"
    attributes {
      label { text = "TABLE" }
    }
    columns {
        column("nr") {
            attributes { label { text = "COLUMN" } }
            ..
        }
    }
    rows {
        row {
           attributes { label { text = "ROW" } }
           cell("nr") {
              value = "Nr.:"
              attributes {
                attributes { label { text = "CELL" } }
              }
           }
            ..
        }
    }
}
```

The result of above configuration will be as such: - In the first row, cell at a column with id "nr" will end with `[ CELL ]`, and rest of cells will end with `[ ROW ]`, - Remaining cells (starting from second row) in a column with id "nr" will end with `[ COLUMN ]`, - All remaining cells will end with `[ TABLE ]`.

Other features

Java interop - fluent builders Java API.

Old-fashioned Java fluent builder API is also supported. It is needless to say it looks much less attractive:

```
Table<Employee> employeeTable = Table.<Employee>builder()
        .attribute(TemplateFileAttribute::builder, builder -> builder.setFileName(
"file.xlsx"))
        .columns()
            .column(Employee::getId)
                .columnType(CellType.NUMERIC)
                .attribute(ColumnWidthAttribute::builder)
            .column(Employee::getFirstName)
                .columnType(CellType.STRING)
                .attribute(ColumnWidthAttribute::builder)
            .column(Employee::getLastName)
                .columnType(CellType.STRING)
                .attribute(ColumnWidthAttribute::builder)
        .rows()
            .row()
                .attribute(RowHeightAttribute::builder, builder -> builder.setPx(100))
        .build();
```

Custom rows.

Sometimes, in addition to records from collection - You need to add user defined rows. Table usually contains a header row or summary footer row. It is also possible to define interleaving custom rows at specified index or rows that match specific predicate.

Row model allows to define custom cell values as well as cell styles and attributes only. It acts as glue for additional features for existing external source derived rows, or as a factory for standalone custom rows that can be hooked at definition time.

Things You can achieve with row model in terms of custom rows includes:

- setting custom cell styles,
- setting row-level attributes (e.g., row height),
- defining row and col spans,
- inserting images,
- setting cell values of different types.

Merging rows.

When multiple `Row` model definitions are qualified by a predicate, they form a single synthetic row. Following rules regarding row merge applies: - Row level attributes will be concatenated or merged if are of same type. - Cell values will be concatenated, or overriden by last cell occurence at given column. - Cell level attributes will be concatenated, or merged if of same type. - Two attributes of same type are merged by overriding clashing attribute properties from left to right where on left side stands attribute from higher level (e.g. row level), and on right site stands attribute from lower level (e.g. cell level).

Library of attributes.

You may need attributes for various reasons - for styling, for formatting or other custom hooks.

Currently, with `tabulate-core` and `tabulate-excel` modules, you will get following attributes included:

Table attributes

- `FilterAndSortAttribute` - enables filtering and sorting of excel table,
- `TemplateFileAttribute` - allows performing template file interpolation with source data collection of items,

Column attributes

- `ColumnWidthAttribute` - sets the width of column (meaning all cells gathered under particular column will have same width),

Row attributes

- `RowHeightAttribute` - sets the height of row (meaning all cells gathered within particular row will have same height),

Cell attributes

- `CellTextStylesAttribute` - allows controlling general, text related style attributes,
- `CellBordersAttribute` - sets borders on selected cells,
- `CellBackgroundAttribute` - sets background color and fill,
- `CellAlignmentAttribute` - sets text vertical and horizontal alignment

Typical usage scenario for attributes:

```
productsRepository.loadProductsByDate(now()).tabulate("product_with_styles.xlsx") {
    name = "Products table"
    columns {
        column(Product::code) {
            attributes(
                width { auto = true },
                text {
                    fontFamily = "Times New Roman"
                    fontColor = Colors.BLACK
                    fontSize = 12
                },
                background { color = Colors.BLUE }
            )
        }
        column(Product::distributionDate) {
            attributes(
                width { auto = true },
                dataFormat { value = "dd.mm.YYYY" }
            )
        }
    }
    rows {
        row {
            attributes(
                text {
                    fontFamily = "Times New Roman"
                    fontColor = Colors.BLACK
                    fontSize = 12
                },
                background { color = Colors.BLUE }
            )
        }
    }
}
```

Roadmap

Starting from version 0.1.0, minor version will advance relatively fast due to tiny milestones. This is because of one person (me) who is currently in charge, and due to my intention of "non-blocking realese cycles" for too long.

v0.2.x

- PDF table export operations implementation.
- Definition time validation for cell spans.

v0.3.x

- CLI table

v0.4.x

- Composition of multiple table models (TableBuilder.include).

v0.5.x

- Multi-part output files. (chunking large files)

v0.6.x

- Codegen for user defined attributes.

Building

Import project into IDE or:

```
gradlew clean build
```

Contributing

Just submit pull request.

License

The project license file is available here.