Tabulate - Reference Manual

Table of Contents

Introduction

Exporting data to tabular 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.

Table model defines 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 model.

Table model is internal concept and is not exposed to API consumers (only `attribute model` can be exposed as it is extensible and customizable). Table is always built 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. Similarly to `row` - `cell` may be used to assign 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 simple text identifier.

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`.

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 possible level where we can mount custom attributes. Only `CellAttribute` can be used on that level.

Table DSL API - type-safe builders.

Kotlin type-safe builders fit well into describing table structure. They make source code look more concise and readable and developement becomes easier. At coding time, your IDE makes use of type-safety offered by builders and shows completion hints which elevates developer experience. Almost zero documentation is required to start. You can start playing with the API right now.

DSL functions by convention take `lambda with receivers` as arguments which abstract away internal API instantiation details from consumers. Within lambda you can call other API methods which in turn, can take downstream builders as arguments. This way - 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 receiver scope methods and access lexical scope variables which can lead to interesting results:

```
val additionalProducts = ... ①
tabulate {
      name = "Products table"
      rows {
          header("Code", "Name", "Description", "Manufacturer") ②
          additionalProducts.forEach { ③
              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 After that, we define header as long as we know that our template doesn't mention it.

③ Finally, we are iterating over collection elements to build static table model.

Although it is possible to build row definitions by iterating over collection directly, you should always prefer to use Column-scoped cell value providers.. They are much faster and consume much less memory than approach shown in point number 3.

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-scoped cell value providers.

Column API makes it possible to pass property getter reference as a column key. This creates object property to column binding which 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.

Presence of Column-scoped cell value providers. in table definition removes the requirement of explicit row definition. It is enough to use `Product::code` getter reference as column key to determine value of each consecutive row cell. You are still allowed to define new rows explicitly (through call `newRow([index value or Row index predicates.])`) or to provide extensions to existing rows (through call `matching { Record row predicates } assign { … }`).

Row predicates.

Row predicates allow choosing row definitions matching only specific conditions. This way you can insert custom rows at specific index or index range, or enrich dynamic data row with custom attributes. There are two kinds of predicates:

- Row index predicates, that are used to define only custom rows (like header or footer)
- Row record predicates, that are used to enrich existing row (custom or dynamic data) with additional attributes.

Row index 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()` invocation 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 We start the row line with method `atIndex { … }` which takes row index predicate `gt(0) and lt(100)`. It 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()}" } }
}
```

One important thing to remember about row index predicate is that it is *always defined as data structure not as a predicate function*. This is because data structure can be materialized into internal map with row indices as keys which enables fast lookup. This approach makes it much faster than iterating over available predicate functions and evaluating them each time next row is requested (that would be required in order to synthesize applicable row definition). Additionally, we can't get more flexibility for custom rows, as long as their indices should be known at definition time and dynamic data context can't be value added.

Record row predicates

Record predicates differs from `row index predicates` in that they cannot be used to insert new custom rows. They can only enrich **existing** row, that is:

- custom row that is created by `newRow` API method,
- or a row that is derived from collection element (it is always produced from Column-scoped cell value providers. column binding).

> Record row predicates *are always represented by a predicate function* that checks if currently processed record or custom row meets specific conditions.

On API level we can define `row predicate` in two ways:

```
①
matching { <predicate> } assign {
  // row attributes, cells definition
}

②
row({ <predicate> }) {
  // row attributes, cells definition
}
```

① First method seems to be closer to natural language but takes more space. Also it does not mention `row` so it may be not intuitive for some users.

② Second method uses DSL keyword **row** in first place which is desired, but as long as we associate predicate with row builder where both are lambdas, we are forced to use syntax like `({ … })` which I personally do not like in Kotlin.

Mixing custom rows with collection elements.

`Tabulate` makes it possible to define table consisting only of custom rows that are known at build time. It also allows You to generate table where each row is dynamically computed from collection of any type. What is more, there is nothing that stops You from using both techniques for single table export:

```
contracts.tabulate("contracts.xlsx") {
    name = "Active Contracts"
    ①
    columns {
        column(Contract::client)
        column(Contract::contractCode)
        column(Contract::contractLength)
        column(Contract::dateSigned)
    }
    rows {
        ②
        header {
            columnTitles(
                "Client",
                "Code",
                "Contract Length",
                "Date Signed",
            )
        }
    }
}
```

① In order to export collection of elements, all we need to do is do define column bindings with getter property references as identifiers. As long as there are no custom row defined in 'rows' section, all rows in table will be rows originating from collection elements.

② If You declare custom row at specific index (or matching index predicate), then it will take precedence over dynamic rows generated from collection. So if You declare `header` row it will be the very first row in exported table, but when You write `newRow(2)` - this will create new custom row as third. Rows: 0 and 1 will be then reserved for dynamic data (collection elements) as long as there are no other custom rows declarations matching previous indices.

There are still cases where this flexibility is not enough. How can we define custom row that will be rendered after all dynamic data ? We cannot just use index based predicate as long as we cannot tell the size of collection in advance. The solution for above is *multi-pass enabled* `RowIndex` *cursor* used by context iterator. This `RowIndex` contains additional 'step' component which increments after there are no row index definitions for current pass. After 'step' is advanced, its local step-scope index is set to zero (this counter will increment per each row matching current pass). Global-scope row index is still maintained to support predicates using it.

Here is how You can add footer row:

```
contracts.tabulate("contracts.xlsx") {
    name = "Active Contracts"
    columns {
        column(Contract::client)
        column(Contract::contractValue)
    }
    rows {
        header("Client","Contract Value")
        ①
        footer {
            cell { value = "Summary:"}
            cell { value = "=SUM" }
        }
    }
}
```

① In above example, `footer` is an extension function as `header`, but with one small difference:

```
fun <T> RowsBuilderApi<T>.footer(block: RowBuilderApi<T>.() -> Unit) {
    newRow(0, AdditionalSteps.TRAILING_ROWS, block) ①
}
```

① As you can see above, it uses additional method argument: `AdditionalSteps.TRAILING_ROWS`. Internally this will create row index definition with index value / predicate, which is relative to `TRAILING_ROWS` step. The order of additional steps is calculated by using enum ordinal values.

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 attributes,
- 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 levels.

Implementing table export operations from scratch.

In order to support new tabular file format you will have to:

- Create `RenderingContext` class. It represents internal state and low-level API to communicate with 3rd party library like Apache POI. Object of that class is passed to all table export operations as well as to all attribute rendering operations that are registered by `ServiceLoader` infrastructure. Such common denominator element is required in order to enable table modifications coming from within various render operations.
- Create `OutputBinding` class. It defines transformation of `RenderingContext` into different kind out outputs. By separating `OutputBinding` from `RenderingContext` we can enable multiple outputs for particular `RenderingContext` class dynamically.
- Define `ExportOperationsProvider` or `ExportOperationsFactory` depending on your scenario. If You don’t need to decouple attribute operations from table export operations (e.g., because supported format does not assumes attributes at all) You can implement `ExportOperationsProvider` interface and define all rendering logic in single class. For cases, where attributes needs to be rendered independently (e.g., because You want to support user-defined attributes) it is advised to extend `ExportOperationsFactory`. For both scenarios You will have 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**.

Below, basic CSV export operations implementation:

First step is to define `RenderingContext`:

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

① `CsvRenderingContext` implements `RenderingContext` marker interface and provides 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.

Then we need to create at least one `OutputBinding` in order to be able to flush results int output:

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

    override fun onBind(renderingContext: CsvRenderingContext, output: OutputStream) {
        renderingContext.bufferedWriter = output.bufferedWriter()
    }

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

① The `.onBind` method is called internally by `TabulationTemplate` as soon as both: output and rendering context instances are available. It connects rendering context with particular output and allows implementing flush logic.

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

Finally, we are implementing `ExportOperationsFactory` compatible with `RenderingContext` of choice:

```
class CsvExportOperationsFactory: ExportOperationsFactory<CsvRenderingContext>() {
    ①
    override fun getTabulationFormat(): TabulationFormat<CsvRenderingContext> =
        format("csv", CsvRenderingContext::class.java)
    ②
    override fun provideExportOperations(): OperationsBuilder<CsvRenderingContext>.()
-> Unit = {

        openRow = OpenRowOperation { renderingContext, _ ->
            renderingContext.line.clear()
        }

        closeRow = CloseRowOperation { renderingContext, context ->
            val lastIndex = context.rowCellValues.size - 1
            with(renderingContext) {
                context.rowCellValues.values.forEachIndexed { index, cell ->
                    line.append(cell.rawValue.toString())
                    if (index < lastIndex) line.append(cell.getSeparatorCharacter())
                }
                bufferedWriter.write(line.toString())
                bufferedWriter.newLine()
            }
        }

    }

    private fun CellContext.getSeparatorCharacter(): String =
        getModelAttribute(CellSeparatorCharacterAttribute::class.java)?.separator ?:
","
    ③
    override fun createOutputBindings(): List<OutputBinding<CsvRenderingContext, *>> =
listOf(CsvOutputStreamOutputBinding())
}
```

① Define `TabulationFormat` first. It consists from `RenderingContext` class and provider id string,

② This is the most important step. **Here we implement actual table rendering logic**. We need to provide operations that transform captured context models using `RenderingContext`. We can use only those operation classes that are required by our provider. We can also use all of them: `openTable`, `openColumn`, `openRow`, `renderRowCell`, `closeRow`, `closeColumn`, `closeTable`.

③ Finally - we need to provide list of supported outputs. Bare minimum should be at least `OutputStreamOutputBinding`.

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

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

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

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

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

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

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

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

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

Factory class `StandardAttributeOperationsFactory` 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 `AttributeOperationsFactory` interface directly, or fill additional constructor properties on `StandardAttributeOperationsFactory` 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 dedicated `AttributeOperation`:

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

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

class SimpleMarkerCellAttributeRenderOperation  :
CellAttributeRenderOperation<ApachePoiRenderingContext, SimpleTestCellAttribute>  {

    override fun renderingContextClass(): Class<ApachePoiRenderingContext> =
ApachePoiRenderingContext::class.java

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

    override fun renderAttribute(renderingContext: ApachePoiRenderingContext, context:
RowCellContext, attribute: MarkerCellAttribute) {
        with(renderingContext.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.operations.AttributeOperation`, and put fully qualified class name of your `AttributeOperation` into 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 ]`.

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:

```
①
FluentTableBuilderApi<Employee> employeeTable = TableBuilder<Employee>()
        .attribute(TemplateFileAttribute::builder, builder ->
builder.setFileName("file.xlsx"))
        .attribute(ColumnWidthAttribute::builder, builder -> builder.setAuto(true))
        .columns()
            .column("id",Employee::getId)
            .column("firstName",Employee::getFirstName)
            .column("lastName",Employee::getLastName)
        .rows()
            .row(0)
                .attribute(RowHeightAttribute::builder, builder -> builder.setPx(100))
        .build();
②
List<Employee> employeeList = Collections.singletonList(new Employee("#00010",
"Joshua", "Novak"));
new TabulationTemplate(format("xlsx")).export(employeeList, new
FileOutputStream("employees.xlsx"), employeeTable);
```

① As a first step, You have to declare table definition using Java `FluentTableBuilderApi`

② Now You have to pass table definition into `TabulationTemplate` in order to export data with declared tabular layout.

Library of attributes.

You may need attributes for various reasons - for styling, for formatting etc.

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

Name	Description	Attribute type	Context	Provider	Applicable levels
`filterAndSort()`	Enables excel table feature that allows filtering and sorting	Table	Table opening	poi (Apache POI)	table
`template()`	Exports data into source template file. (Interpolates excel file)	Table	Table opening	poi (Apache POI)	table
`printing()`	Sets printing attributes on file.	Table	Table opening	poi (Apache POI)	table
`width()`	Sets width of column. Applies to column or all cells within column (depending on rendering context capabilities).	Column	Column opening	any	column
`height()`	Sets the height of row. Applies to row or to all cells within row (depending on rendering context capabilities).	Row	Row opening	any	row
`rowBorders()`	Sets border properties of entire row.	Row	Row closing	any	row

`text()`	Sets text styles like: `font`, `font size`, `font weight`, `italic`, `strikeout`, `underline`, `text wrap`, `orientation`.	Cell	Cell	any	table, column, row, cell
`borders()`	Sets border properties of cell.	Cell	Cell	any	table, column, row, cell
`background()`	Sets the background color for cell.	Cell	Cell	any	table, column, row, cell
`alignment()`	Aligns text within cell (vertically/ horizontally).	Cell	Cell	any	table, column, row, cell
`comment()`	Associates comment (and comment author) with cell.	Cell	Cell	poi (Apache POI)	cell
`separator()`	Sets delimiter for CSV.	Cell	Cell	csv	table

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 }
            )
        }
    }
}
```

Internal algorithms and rules.

This section does not cover consumer API, but instead focuses entirely on internal algorithms implemented in `tabulate-core` module. You won't find here any information needed to start using this library. You may refer to below information if you are curious about how things work under the hood. **It can be also good starting point before you deep-dive into source code**

`Template and operations` pattern.

Library sole purpose is to provide means for data exporting. This goal is achieved through simple, intuitive pattern of a template class dispatching workload to managed, pluggable operations. A template which is referred to as `TabulationTemplate` iterates lazily through `RowContextResolver` progressing each time, when next row context is requested by `TabulationApi`.

Consumer interaction with library may go through `TabulationApi` and then it looks as follows:

- **declare table model** through DSL (or java fluent) builder,
- **enqueue a collection element** (or enqueue nothing when exporting only custom rows). Adding new collection element enables derived row context resolution. `RowContext` exposes all required row related properties to third party operation implementor. Operation implementation uses **row context** to participate in table rendering into target format.
- **request next row rendering**. As mentioned above, each time next row is requested, `RowContextResolver` takes row coordinates as well as additional properties and attributes, then it computes **row context** that is immediately rendered by specific operation implementation. There are certain rules regarding **row context** computation that forms unique algorithm which will be explained in following sections in more details.

Consumer interaction may be also simplified by using extension method on exported collection or custom table builder. In fact this should be leading usage scenario. In this scenario `TabulationApi` calls are wrapped by extension method on `TabulationTemplate`.

Builder materialization.

Before rows can be rendered, a table definition must be built. Effective table definition is always the result of `TableBuilder` materialisation (or freezing). After materialising table builder state, it can be no longer mutated, and as long as there is no use for builder instance, it is marked for GC. At the same time, `Table` definition becomes a final builder snapshot and cannot be modified by any means. Since then, it can be only used as an input for exporting job. During this step attributes are merged together for the first time. This can be done here because we can define multiple attributes of the same type on separate builder APIs.

Table postprocessing.

Next step after building table definition is postprocessing phase. It consists of:

- **Table rows indexing** - building **row index** to **row definitions** associations that enables efficient lookup.
- **Table rows partitioning**. The result of partitioning are two groups of rows - previously mentioned custom rows addressed by row indices, and **enriching** row definitions addressed by predicate functions.
- **Initializing synthetic rows cache**. As long as row context computation request may qualify multiple table row definitions - they are bundled together and forms intermediate entity called `SyntheticRow`. The same row definitions can be qualified multiple times that is why **synthetic rows cache** exists. The cache consists of associations of row definitions as keys with `SyntheticRow` as a values.

One can even say that **table rows indexing** produces first level cache, while **synthetic rows cache** can be referred as second level cache:

When requesting row definition by row index, algorithm is performing lookup to retrieve all applicable table row definitions (this is the first level cache). Next, having say multiple table row definitions it uses them as a key to find a `SyntheticRow` instance (this is the second level cache).

At this point we have table definition with **indexed rows**, and yet cold **cache for keeping synthetic row definitions**.

Synthetic row resolution.

`SyntheticRow` keeps bundled row definitions matching specific row index. During object initialization following actions take place:

- all **cell values** for all qualified table rows are merged so that all values from pair on the right overrides values on the left,
- all **cell attributes** from table and row levels are merged similarly from left to right so that only **explicitly changed** properties of attributes of the same class are overridden,
- all **row attributes** from table and row levels are merged similarly from left to right so that only **explicitly changed** properties of attributes of same class are overridden,

It is crucial to mention that **explicitly changed** cited above means that only attribute properties changed by explicit builder method cat are considered.

Row context resolution.

TBD.

Rendering operations dispatching.

TBD.

Output binding.

TBD.

Cookbook recipes.

Export collection with header and summary.

Add Excel formula for summing column values.

Maintain and bind reusable styles.

Fill monthly revenue template with trend chart.

Create invoice.