Report Configurations

From a JRXML in Jaspersoft Studio to a Report Config in VeloxFactory, including the data adapters.

- Creating reports in Jaspersoft Studio
- Render engine: supported features and expressions
- Managing reports in VeloxFactory
- Data adapters for dyn. data control

Creating reports in Jaspersoft Studio

VeloxFactory renders reports defined as `.jrxml` files — the native format of JasperReports. These files are designed in **Jaspersoft Studio**, a free desktop IDE built specifically for this purpose. This page explains how to set up a `.jrxml` file so that VeloxFactory can analyse it correctly, register all its parts, and render it reliably.

> ℹ **Which bands, elements and expressions does VeloxFactory support?** VeloxFactory renders with its own pure-PHP render engine, not with the Java JasperReports library. The supported feature set, the expression syntax and the formatting rules are listed in Render engine: supported features and expressions.

jaspersoft-studio-overview.png

Jaspersoft Studio

Jaspersoft Studio is an Eclipse-based visual report designer. You use it to lay out the report template — define what data goes where on the page, how it is formatted, and which inputs the report expects. The resulting `.jrxml` file is then uploaded to VeloxFactory, which takes over everything from there: storing it, analysing it, connecting it to a data source, and rendering it on demand.

> ℹ **Use version 6.21.5.** This is the version compatible with VeloxFactory. Download it from the Jaspersoft Community. Reports created with a significantly newer version may use features that VeloxFactory's render engine does not support.

The division of responsibilities between Jaspersoft Studio and VeloxFactory is clear:

- **Jaspersoft Studio** defines the layout, the parameters, the fields, and the visual design of the report.
- **VeloxFactory** manages the SQL query, the data connection, and the actual rendering at runtime.

This means you will see a `<queryString>` element in Jaspersoft Studio — but as explained below, you leave it empty. VeloxFactory supplies the query separately.

The Report Name

Every Jaspersoft Studio project has a **Report Name** — the `name` attribute on the root `<jasperReport>` element. This is not just a filename: VeloxFactory reads it directly from the `.jrxml` on upload and stores it as `report_name` on the `ReportConfig`.

```
<jasperReport ... name="A5_KanBan" ...>
```

This name must be **unique** across all report configurations in VeloxFactory. If you try to upload a `.jrxml` whose report name already exists, the upload will be rejected. The same applies to the file name itself.

> ⚠ **Set a descriptive, unique report name in Jaspersoft Studio before uploading.** You can change it in Jaspersoft Studio via *File → Report Properties → Report Name*, or directly in the XML. Changing it after upload requires re-uploading the file.

jaspersoft-studio-report-properties.png

Parameters

Parameters are **input values** passed into the report at render time. They are used inside the report layout via `$P{PARAMETER_NAME}` expressions — for example to display a customer name in the title, filter by a date range, or pass a document number into a barcode expression.

You define parameters in Jaspersoft Studio via the **Report Inspector → Parameters** section. Each parameter has a name and a Java data type.

jaspersoft-studio-parameter-properties.png

Custom Properties: exampleValue and required

VeloxFactory reads two custom properties from each parameter definition: `exampleValue` and `required`. These are not standard Jaspersoft features — they are custom `<property>` elements you

add manually to the parameter in the `.jrxml`. VeloxFactory's analyser (`JasperFunctions::analyzeReportFile`) extracts them on upload.

Property	Value type	Purpose
`exampleValue`	string	A representative value used when rendering a preview of the report without real data. Also pre-fills the parameter input in the frontend render form.
`required`	boolean (`true` / `false`)	Whether this parameter must be provided in every render request. VeloxFactory rejects render requests that are missing a required parameter.

Both properties can also be set or updated manually in VeloxFactory after upload — they do not have to come from the `.jrxml`. But embedding them in the file means they are automatically picked up every time the report is uploaded or re-uploaded.

To add these properties in Jaspersoft Studio, open the parameter in the **Report Inspector**, switch to the **Properties** panel, and add a new custom property via the green **+** button.

jaspersoft-studio-parameter-custom-properties.png

Supported Data Types

VeloxFactory supports the following Java class types for parameters. The type determines how the input field is rendered in the frontend and how the value is handled at render time.

Java class	Frontend input	Notes
`java.lang.String`	Text field	General-purpose text input
`java.lang.Boolean`	Toggle	Rendered as a checkbox/toggle switch
`java.lang.Short`	Integer input	Range: −32,768 to 32,767
`java.lang.Integer`	Integer input	Range: −2,147,483,648 to 2,147,483,647
`java.lang.Long`	Integer input	64-bit integer
`java.lang.Float`	Decimal input	Single-precision floating point
`java.lang.Double`	Decimal input	Double-precision floating point
`java.math.BigDecimal`	Decimal input	Arbitrary-precision decimal; preferred for monetary values
`java.sql.Date`	Date picker	Date only (no time)
`java.util.Date`	Date picker	Date only (no time)

Java class	Frontend input	Notes
`java.sql.Time`	Time picker	Time only (HH:mm)
`java.sql.Timestamp`	Date + time picker	Combined date and time (datetime-local)

Fields

Fields represent the **data columns** that populate the report's detail band — the repeating section that produces one row of output per data record. Each field corresponds to a column in the SQL query result (when using a live connection) or a key in the data array delivered at render time via the API.

You define fields in Jaspersoft Studio via the **Report Inspector → Fields** section. In the report layout, you reference them via `$F{FIELD_NAME}` expressions inside text fields in the detail band.

jaspersoft-studio-fields.png

Custom Property: exampleValue

Fields support one custom property: `exampleValue`. It works the same way as for parameters — a representative value used when rendering a preview of the report without a live data connection. VeloxFactory collects these example values and assembles a synthetic data row from them for preview rendering.

You add `exampleValue` to a field the same way as for parameters: open the field in the **Report Inspector**, go to the **Properties** tab, and add the custom property.

> ℹ **Set meaningful example values for all fields.** Without them, VeloxFactory cannot generate a preview or thumbnail for the report configuration. The values do not need to be real data — they just need to be type-compatible and representative enough to make the preview look sensible.

Fields support the same Java data types as parameters (see the table above). Use the type that matches the column type your SQL query or data array will produce.

Using Parameters as SQL Variables

When a `ReportConfig` has a `ReportConnectionConfig` assigned, VeloxFactory executes the SQL query defined in the report configuration against that live database connection. Parameters passed in the render request are available as **named bindings** inside that query — using the standard `:parameterName` syntax from Laravel's Eloquent database layer.

This means you can reference any parameter directly in your SQL to filter, sort, or limit the result set:

```
SELECT

    description,
    moq,

    supplier,
    barcode
FROM                                                    articles
WHERE                article_number                 =                :P_ARTICLE_NUMBER
```

In this example, `:P_ARTICLE_NUMBER` is replaced at query execution time with the value of the `P_ARTICLE_NUMBER` parameter from the render request. The binding is handled natively by PDO — values are passed as proper prepared statement parameters, not interpolated as strings.

> **i Only parameters that are actually referenced in the query are bound.** VeloxFactory scans the query for `:name` placeholders before execution and silently drops any parameters that are not needed. Passing extra parameters in the render request will never cause a query error.

The parameter name in the query binding must match the parameter name exactly as defined in the `.jrxml` — including case. For example, a parameter named `P_ARTICLE_NUMBER` in the report must be referenced as `:P_ARTICLE_NUMBER` in the SQL query.

You can use as many parameters as needed across `WHERE`, `ORDER BY`, `LIMIT`, or any other clause that accepts a value binding. Note that named bindings cannot be used for identifiers like table or column names — only for values.

Resources (Images and Logos)

If your report contains images — a company logo, a header graphic, a product photo — these are defined as **resource parameters** in the `.jrxml`. VeloxFactory detects them automatically on upload and creates a `ReportResource` record for each one.

The naming convention is strict: **every resource parameter must start with** `P_RESOURCE_` and must have the Java class `java.lang.String`. At render time, VeloxFactory replaces the parameter value with the actual file path of the uploaded image on the server.

```
<!--            Resource            parameter:            logo            image            -->
<parameter                    name="P_RESOURCE_LOGO"                    class="java.lang.String">

<defaultValueExpression><![CDATA["C:/VeloxFactory/Logo_Dark.png"]]></defaultValueExpression>
</parameter>
```

In the report layout, you then bind this parameter to an image element:

```
<image>
                        <reportElement      x="460"      y="100"      width="84"      height="50
    <imageExpression><![CDATA[$P{P_RESOURCE_LOGO}]]></imageExpression>
</image>
```

The `<defaultValueExpression>` is used only in Jaspersoft Studio for design-time preview purposes. VeloxFactory ignores it at render time — it always uses the uploaded file path instead. You can point it to a local file on your design machine.

> ⚠ **Resource parameters are not treated as regular parameters in VeloxFactory.** They do not appear in the Parameters section — they appear in the Resources section. In the vast majority of render requests you do not pass them at all; VeloxFactory fills them in automatically from the linked resource file.

> **i Exception: per-render overrides.** A render request can optionally supply `resourceOverrides` to swap a resource parameter's image for that single call only — a local path, a remote `http(s)://` URL, or a Base64 upload — without changing the report configuration's linked default. This is an opt-in override on top of the linked resource described above, not a replacement for it: a default file or `CommonReportResource` still has to be linked before the report can render at all. See Rendering with our powerful API for the full reference.

After uploading the `.jrxml`, you must go to the **Resources** section of the report configuration in VeloxFactory and upload the actual image file for each detected resource. A report cannot be rendered until all its resources have files assigned.

jaspersoft-studio-image-resource.png

The Detail Band

The detail band is the repeating section of the report — the part that outputs one row per data record. If your report displays a list of items, a table, or any kind of repeating structure, it lives in the detail band.

VeloxFactory checks whether a detail band is present when analysing the `.jrxml`:

- If a detail band exists, VeloxFactory expects data to be provided at render time — either via a live SQL connection or via the `data` array in the API request.
- If no detail band exists, the report is treated as a **static layout** — no data is needed, only parameters.

> ⚠ **If you add a detail band, it must contain at least one text field element that uses a field expression (`$F{...}`).** A detail band that exists but displays no field data will be rejected on upload. VeloxFactory uses the presence of text fields in the band as a basic integrity check.

Reports without a detail band are perfectly valid — they are useful for documents like cover pages, summary sheets, or any report whose content is entirely driven by parameters rather than repeated rows.

The SQL Query

Jaspersoft Studio has a built-in `<queryString>` element where you would normally write the SQL query for the report. **In VeloxFactory, this element is ignored.** The SQL query is instead defined and stored directly in VeloxFactory as part of the `ReportConfig` — not in the `.jrxml` file.

This means you should **leave the `<queryString>` empty** when creating a `.jrxml` for VeloxFactory:

```
<queryString>
    <![CDATA[]]>
</queryString>
```

You write the actual SQL query in VeloxFactory after uploading the report, in the **Query** field of the report configuration. This design keeps the query where it can be managed, versioned, and changed without touching the report template file.

The query result columns must match the field names you defined in the `.jrxml`. For example, if your report has a field named `articleNumber`, your SQL query must return a column called `articleNumber`.

The Data Adapter

Jaspersoft Studio uses **Data Adapters** to connect to a live database so you can preview your report during design. This is entirely a design-time feature — the data adapter you configure in Jaspersoft Studio has no effect in VeloxFactory and is not stored in the `.jrxml`.

At render time, VeloxFactory always uses its own internal array-based adapter to pass data to the render engine. Whether that data comes from an SQL query executed against a `ReportConnectionConfig` or from a data array in the API request, VeloxFactory always handles the data handoff itself.

You still benefit from configuring a data adapter in Jaspersoft Studio during development — it lets you see a realistic preview while designing the layout. Just be aware that the adapter configuration stays local to your machine.

Annotated Example

The following is a complete, minimal `.jrxml` that demonstrates everything discussed on this page. It is the actual `A5_KanBan` report included in the VeloxFactory demo. Inline comments explain each relevant element.

```
<?xml                              version="1.0"                              encoding="UTF-8"?>
<!--       Created       with       Jaspersoft       Studio       version       6.21.5       -->

<jasperReport
    xmlns="http://jasperreports.sourceforge.net/jasperreports"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="..."
```

```
     name="A5_KanBan"          <!-- Report Name — becomes report_name in VeloxFactory. Must be
unique.                                                                                    -->
           pageWidth="595"                          <!--   Page   dimensions   in   Jasper   pi
inch).
                  pageHeight="842"                                    <!--    VeloxFactory
upload.
     ...>

                                                                      <!--

                                          Name     starts     with     P_RESOURCE_     and
                       VeloxFactory   detects   this   as   a   resource   (image/logo),   not
                      The   defaultValueExpression   points   to   a   local   file   for   Studio
                       VeloxFactory   replaces   it   at   render   time   with   the   uploade

                                                                 <parameter                na

<defaultValueExpression><![CDATA["C:/VeloxFactory/Logo_Dark.png"]]></defaultValueExpression>
    </parameter>

                                                                      <!--

                       P_ARTICLE_NUMBER   is   a   user-supplied   input   value   passed   at
                 exampleValue    →  used  for  preview  rendering  and  pre-fills  the  front
           required       → omitting this parameter in a render request will cause a 422 erro

                                                                <parameter                name


    </parameter>

                                                                      <!--

                                                 The     SQL     query     is     defined     i

    <queryString>
        <![CDATA[]]>
    </queryString>
```

```
                                                                    <!--
     FIELDS
            Each  field  maps  to  a  column  in  the  SQL  result  or  a  key  in  the  data
               exampleValue  →  used  for  preview  rendering  when  no  live  data  is  a

                                                                     <field

</field>

                                                                         <field
               <property  name="exampleValue"  value="Packing  Carton  Size  1  -  200>
</field>



</field>

                                                                       <field

</field>

                                                                            <field
                                                                          <property
</field>

                                                                              <fie

</field>

                                                                    <!--

                                          The     repeating     section     –     one
                         Must   contain   at   least   one   <textField>   using   a   $
                                                    otherwise       VeloxFactory
                              The    image    element    uses    $P{P_RESOURCE_LOGO}    –
                                          this     to     the     uploaded     resource

<detail>
```

```
                                                    <!--    Field    values    referenc
            <textField>
                <textFieldExpression><![CDATA[$F{articleNumber}]]></textFieldExpression>
            </textField>

            <textField>
                <textFieldExpression><![CDATA[$F{description}]]></textFieldExpression>
            </textField>

                                        <!--   Resource   image:   bound   to   the   P_RES
            <image>
                <imageExpression><![CDATA[$P{P_RESOURCE_LOGO}]]></imageExpression>
            </image>

        </band>
    </detail>

</jasperReport>
```

> **i This example has been simplified for clarity.** A real `.jrxml` contains many additional attributes, layout elements, and Studio-specific property annotations. The elements shown here are the ones VeloxFactory actively reads and acts on. Everything else is interpreted by the render engine as described in Render engine: supported features and expressions.

Pre-Upload Checklist

Before uploading a `.jrxml` to VeloxFactory, verify the following:

- The **Report Name** (`name` attribute on `<jasperReport>`) is set, descriptive, and unique.
- All **resource parameters** follow the `P_RESOURCE_` naming convention and have class `java.lang.String`.
- All **regular parameters** have `exampleValue` and `required` custom properties set where applicable.
- All **fields** have an `exampleValue` custom property set.
- The **`<queryString>`** is empty — the SQL is managed in VeloxFactory.
- If the report has a **detail band**, it contains at least one `<textField>` with a `$F{...}` expression.
- **Every band** has `splitType="Stretch"`, including an explicit empty `<background>` band, and the bands follow the JasperReports XSD order (`pageFooter` and `lastPageFooter` before

`summary`).

- The template only uses features listed in Render engine: supported features and expressions (no charts, no crosstabs, no `$R{...}`).
- The report **previews correctly in Jaspersoft Studio** using the local data adapter — this confirms the layout and expressions are valid before upload.
- After the upload, **render it once in VeloxFactory** with realistic data, including enough rows for more than one page. The rendering in VeloxFactory is the reference, not the Studio preview.

Render engine: supported features and expressions

VeloxFactory renders `.jrxml` templates with its own render engine: a pure-PHP implementation of JasperReports, based on the open source project JasperPHP and maintained by us as an independent fork. It needs no Java runtime, but it implements a defined subset of JasperReports rather than the full Java engine. This page lists exactly what that subset is, so you can design templates in Jaspersoft Studio that render the same way in VeloxFactory.

How to set up parameters, fields, resources and the query is described in Creating reports in Jaspersoft Studio. This page covers what happens inside the template: bands, elements, styles, expressions and formatting.

> **i This page describes render engine 1.2.** If a template behaves differently from what is described here, the render engine is the reference, not Jaspersoft Studio's preview. Always check a real rendering in VeloxFactory before you release a template.

Two rules for every template

- **Every band needs** `splitType="Stretch"`, including an explicit empty `<background><band splitType="Stretch"/></background>`.
- **Bands must follow the JasperReports XSD order:** `background`, `title`, `pageHeader`, `columnHeader`, `detail`, `columnFooter`, `pageFooter`, `lastPageFooter`, `summary`. Jaspersoft Studio refuses to open a file with a different order.

Bands

Band	Behaviour

`background`	Printed once per page at the top edge, behind the content. Does not push other bands down.
`title`	Printed once on the first page. `$F{...}` shows the first data row.
`pageHeader`	Printed on every page. On the first page it follows the title.
`columnHeader`	Printed on every page that contains detail rows, also after a manual `<break>`.
`detail`	Printed once per data row. Grows with stretching text fields.
`groupHeader` / `groupFooter`	Printed whenever the value of the `groupExpression` changes.
`columnFooter`	Printed below the detail rows of a page.
`pageFooter`	Printed at the bottom of every page.
`lastPageFooter`	Replaces the `pageFooter` on the last page. Also works without a `pageFooter`: then a footer appears on the last page only.
`summary`	Printed once at the end. `$F{...}` shows the last data row, variables hold their final values. If the summary does not fit and moves to a page of its own, that page has no page header, column header or footer, and the `lastPageFooter` appears on the page before. With `isSummaryWithPageHeaderAndFooter="true"` on `<jasperReport>`, the summary page gets page header and footer (the `lastPageFooter`), but still no column header. This is JasperReports behaviour.
`noData`	Not supported.

Elements

Element	Supported
`textField`, `staticText`	Horizontal and vertical alignment, rotation, `markup="html"`, line spacing (Single, 1_1_2, Double, Proportional), `textAdjust` (StretchHeight, ScaleFont, CutText), `<box>`, `pattern` / `patternExpression`, hyperlinks
`rectangle`	`radius`, `mode`, forecolor and backcolor, pen width, colour and style (Solid, Dashed, Dotted), fractional sizes such as `height="0.8"`
`line`	Pen as for rectangles. Lines thinner than 1 pt are drawn straight.

Element	Supported
`frame`	Background, border, child elements positioned relative to the frame
`image`	Path or URL as an expression, Base64 data URIs (`...`), JPG and PNG. In VeloxFactory, images normally come from `P_RESOURCE_` parameters.
Barcodes	QR Code, DataMatrix, PDF417, Code 128, Code 39, Code 93, EAN-8, EAN-13, Interleaved 2 of 5, Postnet. The code is an expression.
`subreport`	Data source from a list field (`$F{items}`, also `new JRBeanCollectionDataSource($F{items})`), file name as an expression, `.jasper` is resolved to `.jrxml`
`break`	Page break, prints page footer, page header and column header
`jr:table`	Basic tables

Every element respects `printWhenExpression`, evaluated for each row.

Layout rules

- **StretchHeight** makes the band taller, never the elements beside it. The next band starts below the lowest stretched field.
- **Vertical alignment with StretchHeight:** if the text fits into the field, Top, Middle and Bottom apply as usual. Only overflowing text grows downwards from the top.
- `positionType="Float"` moves an element down by the growth of the stretched fields above it. All other elements keep their position.
- **ScaleFont** reduces the font size until the text fits into the box. Line breaks are allowed.
- **Page breaks** fill the page down to the page footer. The decision uses the declared band height.
- **`<box>`:** `<pen>` applies to all four sides, `topPen`, `leftPen`, `bottomPen` and `rightPen` override it. The colour falls back from the side pen to the box pen, then to `forecolor`, then to black. `padding` sets all four paddings, `leftPadding` etc. override it.

Defaults are the same as in JasperReports:

- Rectangle: Opaque, white background, 1 pt black border.
- Line: 1 pt. A pen width of 0 draws nothing.
- Pen colour: `lineColor` takes precedence over `forecolor`.

Styles

- **Assignment:** `style="Name"` on the `reportElement`. A style with `isDefault="true"` applies to all elements without a style of their own.
- **Inheritance:** `<style name="Child" style="Parent">`.
- **Precedence:** element attribute, then all matching conditional styles in order, then the style, then its parent.
- **Conditional styles** are evaluated again for every row, for example to colour negative quantities red or to shade every second row.

Styles carry: `mode`, `forecolor`, `backcolor`, horizontal and vertical text alignment (`hTextAlign`, `vTextAlign` and the older names), `rotation`, `markup`, font name, size, bold, italic, underline, strike-through, `pattern`, `radius`, image scaling and alignment, and the child elements `<pen>`, `<box>` and `<paragraph>`.

> ⚠ **An element with its own style does not inherit the default style.** This is JasperReports behaviour. If a style should build on the default style, set `style="Default"` as its parent.

Expressions

Expressions are written in Java syntax, as in Jaspersoft Studio. They are evaluated by a built-in interpreter; no code is executed on the server. Expressions work in text fields, `printWhenExpression`, conditional styles, variables, `patternExpression`, hyperlinks, images, barcodes and subreport file names.

Syntax

References	`$P{name}`, `$F{name}`, `$V{name}`
Literals	`"text"` (with `\" \\ \n \t \uXXXX`), `'text'`, `42`, `1.5`, `10L`, `2.5d`, `true`, `false`, `null`
Operators	`? :`, `||`, `&&`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `+`, `-`, `*`, `/`, `%`, `!`, parentheses

Casts	`(String)`, `(Integer)`, `(Long)`, `(Double)`, `(BigDecimal)`, `(Number)`, `(Boolean)`, `(Date)`

How values behave

- `+` concatenates as soon as one operand is text, from left to right: `1 + 2 + "a"` gives `3a`, `"a" + 1 + 2` gives `a12`.
- `7 / 2` gives `3` (integer division), `7 / 2.0` gives `3.5`.
- The type comes from the declared `class` of the field, parameter or variable. Declare amounts as `java.math.BigDecimal`, `java.lang.Double` or `java.lang.Integer`; a value declared as `String` is concatenated instead of added.

Three deliberate differences from Java make templates more robust:

- `null` is concatenated as empty text (Java: `"null"`).
- `==` compares values like `.equals()`.
- Arithmetic with `null` returns `null` instead of failing.

Methods

All values	`equals`, `toString`, `compareTo`
Text	`length`, `isEmpty`, `isBlank`, `trim`, `strip`, `toUpperCase`, `toLowerCase`, `substring`, `charAt`, `indexOf`, `lastIndexOf`, `contains`, `startsWith`, `endsWith`, `replace`, `replaceAll`, `replaceFirst`, `matches`, `concat`, `split`, `repeat`, `equalsIgnoreCase`
Numbers	`intValue`, `longValue`, `doubleValue`, `floatValue`, `setScale(n[, RoundingMode.X])`, `add`, `subtract`, `multiply`, `divide`, `negate`, `abs`, `signum`, `toPlainString`
Dates	`getTime`, `before`, `after`
Lists	`size`, `isEmpty`, `get(i)`, `contains`

Static calls, constants and constructors

- **Text:** `String.valueOf`, `String.format([Locale,] format, ...)`, `String.join`
- **Parsing:** `Integer.parseInt` / `valueOf`, `Long.parseLong`, `Double.parseDouble` / `valueOf`, `BigDecimal.valueOf`, `Boolean.valueOf` / `parseBoolean`
- **Math:** `abs`, `round`, `floor`, `ceil`, `max`, `min`, `pow`, `sqrt`
- **Constants:** `Boolean.TRUE` / `FALSE`, `BigDecimal.ZERO` / `ONE` / `TEN`, `RoundingMode.*`, `Locale.GERMANY`, `GERMAN`, `US`, `UK`, `ENGLISH`, `FRANCE`, `ITALY`
- **Constructors:** `new java.util.Date()`, `new Date(ms)`, `new java.text.SimpleDateFormat("...", [Locale])` with `.format()` and `.parse()`, `new`

`java.text.DecimalFormat("...").format()`, `new BigDecimal("1.5")`, `new Integer(...)`, `new String(...)`, `new Locale("de", "DE")`

Examples

```
"Customer: " + $P{P_CUSTOMER}
$P{P_NOTE} == null || $P{P_NOTE}.isEmpty() ? "-" : $P{P_NOTE}
$F{status}.equals("late") && $F{qty} > 0
String.format("%s / %05d", $F{order}, $F{pos})
new java.text.SimpleDateFormat("EEEE, d. MMMM yyyy", java.util.Locale.GERMAN).format($F{date})
$F{amount}.setScale(2, java.math.RoundingMode.HALF_UP)
```

Legacy suffixes

Two JasperPHP shortcuts still work for existing templates: `$F{amount}.toString()` always returns two decimals in German format (`1.234,50`), and `$F{amount}.numberToText()` spells out a number (Portuguese). For new templates, use `pattern` instead.

Not supported

`$R{...}` (resource bundles), arbitrary Java classes, lambdas and field access without a method call (`.length` without parentheses). A text field with an unsupported expression falls back to plain placeholder substitution, which usually shows parts of the expression in the output. A condition with an unsupported expression counts as false, so the element is hidden.

Formatting numbers and dates

- **`pattern` / `patternExpression`:** `patternExpression` takes precedence and is evaluated for each row, for example to switch the number of decimals per article.
- **Numbers** use Java's DecimalFormat: `#,##0.00`, `0.0%`, `000000`, `#,##0.00 €`, `#,##0.00;(#,##0.00)`, `0.00E0`. Rounding is HALF_EVEN as in Java; `setScale` rounds HALF_UP by default.
- **Dates** use Java's SimpleDateFormat: `dd.MM.yyyy`, `EEEE, d. MMMM yyyy`, `HH:mm`.
- **Values that arrive as text** are formatted as well when they contain a number or an ISO date (`2026-05-02`, `2026-05-02 14:30:00`). This matters because render requests usually pass parameters as text.

Language and locale

Pass the parameter `REPORT_LOCALE` in the render request, for example `de_DE`. It controls decimal and thousands separators, month and day names, `String.format` and `pattern`. Built in are German (including `de_CH`), English, French, Spanish, Italian and Portuguese. Without `REPORT_LOCALE`, the default locale of the installation applies (English unless configured otherwise).

> ℹ **Do not declare `REPORT_LOCALE` as a parameter in the template.** It is a built-in JasperReports parameter, and Jaspersoft Studio rejects a second declaration. Pass it in the render request or the schedule only.

Variables

- **Calculations:** Sum, Count, Average, DistinctCount, Highest, Lowest, First, Nothing (current value). Count and Average only count values that are not null.
- **Reset:** `resetType="Report"` or `"Group"`. Groups reset correctly even without a group header band.
- **Expressions** can be any supported expression, for example `$F{qty} * $F{price}`.
- **Built in:** `PAGE_NUMBER` and `REPORT_COUNT`. For "Page N of M", use two text fields: `"Page " + $V{PAGE_NUMBER}`, and `" of " + $V{PAGE_NUMBER}` with `evaluationTime="Report"`.

Upgrading templates from earlier versions

Templates built for earlier VeloxFactory versions keep working. Check them for these changes:

- **Conditions in PHP syntax** (for example `is_int(...)`, `===`, `and` or `or` in `printWhenExpression`) now count as false, and the element is hidden. Rewrite them in Java syntax.
- **Lines without** `<pen>` are 1 pt wide instead of a hairline.
- **Element attributes override the style** (previously the other way round).
- **Multi-page reports** fit more rows per page and repeat the column header on every page.
- **Workarounds are no longer needed:** concatenation with `+`, `pattern`, `<box>` and page footers on every page now work as in Jaspersoft Studio.

Known limits

- Charts and crosstabs are not supported and render as an empty area. Use tiles, lists and aggregated groups instead.
- `noData` band and `$R{...}` are not supported.
- `isBlankWhenNull="false"` is ignored; null always prints as empty text.
- A band that stretches shortly before the end of a page can reach into the page footer. Leave some space above the footer on pages with long texts.

Managing reports in VeloxFactory

A report in VeloxFactory is more than a file. It is a fully managed configuration - with its own parameters, data fields, image resources, SQL query, data connection, preview images, rendering history, and print records. This page walks through the complete lifecycle of a report configuration, from upload to print.

Screenshot placeholder: `report-config.index.png`
Report Config overview, the landing page after login, with context badges and thumbnails.

The ReportConfig - Central Master Data

The `ReportConfig` is the core entity in VeloxFactory. Every report you manage is a `ReportConfig` record, and everything else in the system either belongs to it or references it. A single `ReportConfig` brings together:

- The `.jrxml` **template file** stored on disk
- Its **parameters** - input values passed at render time
- Its **fields** - data columns that populate the detail band
- Its **resources** - graphic assets (images, logos) embedded in the template
- Its **SQL query** - defined and managed directly in VeloxFactory
- Its **data connection** - the live database to query (optional)
- Its **context** - an organisational label for grouping
- Its **preview and thumbnail images** - generated from example data

Nothing renders without a `ReportConfig`. Nothing prints without one either. It is the starting point for every operation in VeloxFactory.

The Report Configuration Lifecycle



Each step is described in detail below.

Uploading a Report

When you upload a `.jrxml` file, VeloxFactory immediately analyses it and builds the initial configuration automatically. The following is extracted from the file:

- **Report name** - from the `name` attribute on `<jasperReport>`. Must be unique.
- **Page dimensions** - width and height, converted from Jasper pixels to millimetres.
- **Detail band presence** - whether the report has a repeating data section.
- **Parameters** - all non-resource parameters, including their data types and any `exampleValue` / `required` custom properties set in the `.jrxml`.
- **Fields** - all data fields, including their data types and `exampleValue` custom properties.
- **Resources** - all parameters following the `P_RESOURCE_` naming convention (image assets).

The result is a fully structured `ReportConfig` record with all its child records in place, but not yet complete. Resource files still need to be uploaded, and the SQL query still needs to be written if a live data connection is used.

> ⚠ **Report name and file name must be unique.** VeloxFactory will reject an upload if a `ReportConfig` with the same report name or the same file name already exists.

The file name may contain letters, digits, dot, hyphen and underscore, must end in `.jrxml` and is limited to 120 characters. Directory parts are not accepted.

Screenshot placeholder: `report-config.create.png`
Report Config create form with the JRXML upload, context and data adapter.

Completing the Configuration

After upload, the report configuration is ready but not yet fully operational. The following steps complete it:

Upload Resource Files

If the report contains image resources (detected as `P_RESOURCE_` parameters), each one requires an actual file to be uploaded. VeloxFactory cannot render the report until all resource files are in place.

Alternatively, a resource can be linked to a `CommonReportResource` - a shared asset reused across multiple reports, such as a company logo. Linking is permanent: the resource's own file is deleted and the common file is used in its place.

A Common Report Resource serves a second purpose as well: a PNG or JPEG uploaded there can be placed into a mail template as an inline graphic, referenced by its name. The same logo therefore covers the report and the mail it is sent with. See Mail Templates.

Resource files are images: `png`, `jpg`, `jpeg`, `gif` or `webp`. VeloxFactory checks the actual content, not the extension, and refuses a file whose bytes are not one of those formats.

The file name is normalised before it is stored. Any directory part is dropped, inner dots become hyphens (`logo.v2.png` is stored as `logo-v2.png`), everything outside letters, digits, hyphen and underscore becomes a hyphen, and the name part is cut at 80 characters. The stored name is what the report refers to, so a renamed file shows up under its normalised name in the resource list.

> ℹ **The linked/uploaded file is the default, not the only option.** A single render request can override a resource's image for that call alone via `resourceOverrides` - a local path, a remote URL, or a Base64 upload - without touching this configuration. The file uploaded here is still required as the fallback. See Rendering with our powerful API.

Screenshot placeholder: `report-config.edit.png`
Report Config edit view with parameters, fields and resources.

Review Parameters and Fields

VeloxFactory picks up `exampleValue` and `required` custom properties from the `.jrxml` automatically on upload. If these were not set in Jaspersoft Studio, or if you need to adjust them, you can do so directly in the configuration.

Every parameter and field should have an example value set before generating a preview.

Write the SQL Query and Assign a Connection

If the report fetches live data from a database, assign a `ReportConnectionConfig` and write the SQL query in the **Query** field. The query is stored in VeloxFactory, not in the `.jrxml`.

Parameters are available as named bindings in the query (`:PARAMETER_NAME`). See Creating reports in Jaspersoft Studio for details on how parameter binding works.

A connection is not required if the report has no detail band, or if data will be delivered in the render request itself.

Generating a Preview

Once all resource files are uploaded and all parameters and fields have example values, you can generate a preview. VeloxFactory renders the report using the stored example values, no live data needed, and stores the result as a base64-encoded PDF and a thumbnail image on the `ReportConfig`.

The preview is used in the report list as a visual card and as a quick sanity check that the template renders correctly. It is also what the `useExampleValues` flag triggers during a render request, useful for testing without providing real data.

> ℹ **Preview generation will fail if any resource file is missing or any example value is not set.** VeloxFactory checks all three conditions - resources, parameter example values, and field example values - before attempting to render.

Screenshot placeholder: `report-config.edit.preview.png`
Report Config edit view with the generated preview and thumbnail.

Report History Records

Every render request can optionally create a `ReportHistoryRecord` - a full log entry of what was requested and what was returned. This is controlled by the `createHistoryRecord` flag in the render request body and is **off by default**.

When enabled, the history record captures:

- The exact request payload sent to the render endpoint
- The full API response
- The rendered PDF (Base64-encoded)
- A thumbnail of the first page (generated asynchronously in the background)
- The rendering status (`ok`, `render_fail`, `error`, `unknown`)
- The trace ID for cross-referencing with logs

History records are valuable for traceability, you can see exactly what was rendered, when, with what data, and what the result was. From a history record, you can also dispatch a reprint or send the document by mail directly.

> ℹ **Downloads follow the read permission.** Downloading a report template, a resource file, a rendered PDF or an xlsx export requires the read permission for that resource. Being signed in is not enough.

When to Skip History Records

History records are entirely optional. There are two good reasons to leave them off:

Performance and storage. Storing the full PDF, request payload, and response for every render adds up. For high-frequency rendering where traceability is not needed, skipping history records keeps the database lean.

Data sensitivity. A history record stores the complete render payload, including all parameters and data passed to the report. If that data is sensitive (personal data, financial figures, medical information), you may not want it persisted on the server at all. Omitting `createHistoryRecord` from the render request ensures nothing is logged.

Retention

History records are automatically purged after a configurable number of days. The retention period is set via the `PURGE_HISTORY_DAYS` environment variable (default: 30 days). Purging runs automatically as a background job, no manual intervention required.

> **Screenshot placeholder:** `report-history-record.index.filtered.png`
> Report History Record overview with filters applied and status badges.

Report Mail Tasks

The second way out of VeloxFactory is a mail. A render request can carry a `mailing` segment, and a document that was rendered earlier can be mailed from its history record at any time. Either way a `ReportMailTask` is created, logging recipients, subject, body and attachments, exactly as a print task logs printer and copies.

What a mailing needs is set up once under *Configuration*: a **Mailer** holds the SMTP account, a **Mail Template** holds subject and body with placeholders. The PDF is attached, and on request an xlsx export of the same rows the report was rendered with. Sending is synchronous by default, or handed to the queue.

> **ℹ A mail task blocks its history record from being deleted, just like a print task does.** Mail tasks have their own retention setting, `PURGE_MAILTASKS_DAYS`, and are purged before the history records they point at. Full detail on Report Mailing.

Report Print Tasks

A `ReportPrintTask` sends a rendered PDF to a physical printer. Print tasks are created as part of a render request, you render and dispatch to a printer in a single call, by setting `createPrintTask: true` and providing a `printerName`.

Print tasks are always linked to a `ReportHistoryRecord`. This means creating a print task also creates a history record (regardless of whether `createHistoryRecord` is explicitly set), so the printed document is always traceable.

How Printing Works

VeloxFactory does not communicate with printers directly. Instead, it creates a `ReportPrintTask` record and notifies a separate print service - a lightweight C# application running on or near the target machine - which picks up the task and executes the print job.

There are two modes of delivery:

WebSocket (push). If a `broadcastId` is included in the render request, VeloxFactory broadcasts a `ReportPrintTaskCreated` event via WebSocket (Laravel Reverb) the moment the task is created. The print service subscribes to that channel and reacts immediately. This is the recommended mode for real-time printing, the task reaches the printer within milliseconds of the render completing.

Polling (pull). Without a `broadcastId`, no broadcast is sent. The print service must poll the API for new tasks in `pending` status. This works fine for less time-sensitive workflows.

Print Task Status

Status	Meaning
pending	Created, waiting for the print service to pick it up
printed	Print job executed and confirmed by the print service
error	Print service reported a failure
unknown	Status could not be determined

The print service reports status back to VeloxFactory via the API after executing the job. The `error_message` field on the task record contains the failure detail if printing did not succeed.

Copies

The `numberOfCopies` field is passed to the print service as the requested number of printed copies. It defaults to `1` if not specified. VeloxFactory always renders the PDF exactly once, the print service is responsible for duplicating the output on the printer side.

> ℹ **Print tasks also have configurable retention.** They are automatically purged after `PURGE_PRINTTASKS_DAYS` days (default: 30). Like history record purging, this runs in the background without any manual action.

Screenshot placeholder: `report-print-task.index.png`
Report Print Task overview with status badges, printer and the reset action.

Deleting a Report Configuration

A `ReportConfig` can only be deleted when no `ReportHistoryRecord` or `ReportPrintTask` references it. VeloxFactory will reject a deletion request while any such records exist.

When a `ReportConfig` is deleted, the following is removed along with it: the `.jrxml` file from disk, all resource files, and all parameter and field records. The deletion is atomic, if any step fails, the entire operation is rolled back.

> ⚠ **To delete a ReportConfig that has history records or print tasks, those records must be removed first.** Once the retention period has passed and the automatic purge has run, or once the records are manually deleted, the ReportConfig can be removed.

Data adapters for dyn. data control

Reports that display repeating data - lists, tables, card grids - need a data source. VeloxFactory supports two ways to supply that data at render time: a live **SQL connection** that queries a database automatically, or a **dynamic array** delivered directly in the render request. Understanding when to use which approach, and how each one works, is key to getting the most out of VeloxFactory.

Two Approaches, One Result

	SQL Connection	Dynamic Array
Data source	Live database, queried at render time	JSON array in the render request body
Who fetches the data?	VeloxFactory	The calling application
Connection config needed?	Yes	No
SQL query needed?	Yes	No
Best for	Reports where VeloxFactory has direct DB access	Reports where the caller already has the data

Both approaches produce the same result: a populated report. The choice depends on where your data lives and who is best placed to retrieve it.

Reports without a detail band - purely static layouts driven by parameters - need neither.

> **i The rows are used beyond the PDF.** Whatever ends up as field data, fetched by an SQL query or delivered in the request, is also what an xlsx mail attachment is built from, and what the `[data.first.<column>]` placeholders address. For an SQL-driven report that means a mail can be addressed to a recipient the query itself returned. See Report Mailing.

SQL Connections

A `ReportConnectionConfig` defines a live database connection that VeloxFactory uses to fetch data at render time. When assigned to a `ReportConfig`, VeloxFactory executes the configured SQL query against that connection, takes the result rows, and feeds them as field data into the report.

Setting Up a Connection

A connection config holds the credentials and driver settings for one database. Supported drivers are MySQL, MariaDB, PostgreSQL, and Microsoft SQL Server.

Before a connection can be assigned to a report, it must be **tested and approved**. VeloxFactory runs a test query against the database to verify connectivity, only connections with a passing test are available in the ReportConfig assignment dropdown.

> ⚠ **The database must be reachable from the VeloxFactory server.** For databases in separate networks, use an encrypted VPN tunnel (WireGuard or OpenVPN). Do not expose database ports to the public internet. See Configuration and Data Models for network requirements.

> ⚠ **MSSQL connections verify the server certificate.** Earlier releases accepted any certificate. A connection to a server with a self-signed certificate now fails until either a valid certificate is in place or `DB_TRUST_SERVER_CERTIFICATE=true` is set explicitly in the instance's `.env`. Check your existing connections before you upgrade.

The host of a connection config is checked when it is saved. Loopback and link-local addresses are refused. An instance can narrow this further with `OUTBOUND_HOST_ALLOWLIST`, a comma separated list of the hosts that connections and mailers may point at; an empty list means everything except loopback and link-local. A failed connection test answers with a generic message, the reason is in the application log.

> **i Give the connection a read-only database user.** No `FILE` privilege on MySQL and MariaDB, no `pg_read_server_files` on PostgreSQL, `db_datareader` without `xp_cmdshell` on MS SQL Server. The permission `report-config:update` lets a user write the query, which in practice means read access to everything that database user can reach.

Screenshot placeholder: `report-connection-config.index.png`
Report Connection Config overview with driver badges and the approval status.

Writing the SQL Query

The SQL query is written and stored in VeloxFactory, not in the `.jrxml`. It lives on the `ReportConfig` record and is executed against the assigned connection at render time.

The query must return columns whose names match **exactly** the field names defined in the `.jrxml`. For a report with fields `articleNumber`, `description`, and `moq`, the query must alias its columns accordingly:

```
SELECT
    barcode
FROM                                                articles
ORDER                    BY                art_no                    ASC
```

Column names are case-sensitive. `articleNumber` and `articlenumber` are not the same field.

What a Report Query May Contain

A report query is a single read statement. VeloxFactory checks it when the Report Config is saved and refuses anything else:

- It must begin with `SELECT` or `WITH`.
- It must not contain a semicolon, so a second statement cannot be appended.
- It must not use constructs that read files, write files or run commands on the database server: `INTO OUTFILE`, `INTO DUMPFILE`, `LOAD_FILE`, `LOAD DATA`, `BULK INSERT`, `OPENROWSET`, `OPENDATASOURCE`, `xp_cmdshell`, `sp_OACreate`, `pg_read_file` and their relatives.
- At most 20,000 characters.

Parameters stay parameters. They are bound by the driver, never pasted into the statement, which is why a value can safely contain quotes or a semicolon.

> ⚠ **Existing queries keep running, but may no longer save.** The check runs when a Report Config is written, not when it renders. A query that predates this release and contains a semicolon, even inside a string literal, is refused the next time the Report Config is edited. Rewrite it with `CHAR(59)` or a parameter.

Using Parameters as SQL Variables

Parameters passed in the render request are available as named bindings in the SQL query using the `:PARAMETER_NAME` syntax. VeloxFactory scans the query for `:name` placeholders before execution

and binds only the parameters that are actually referenced, extras are silently ignored.

This makes it straightforward to filter, sort, or paginate the result set based on render-time input:

```
-- Filter by article number
SELECT




FROM articles
WHERE art_no = :P_ARTICLE_NUMBER
```

```
-- Date range filter with two parameters
SELECT




FROM orders
WHERE order_date BETWEEN :P_DATE_FROM AND :P_DATE_TO
ORDER BY order_date ASC
```

```
-- Wildcard search
SELECT



FROM articles
WHERE art_description LIKE CONCAT('%', :P_SEARCH_TERM, '%')
```

The render request for the date range example would look like this:

```
POST /api/v1/report-config/OrderList/render

{




  },
```

```
}
```

Parameter Promotion from Query Results

There is a powerful pattern worth knowing: if a SQL result column has the same name as a registered parameter on the report, VeloxFactory automatically **promotes** that value from the data rows into the parameters map, before the report renders.

This means you can derive parameter values directly from the database without having to pass them in the render request. The query does the lookup; the result feeds both the detail band and the header parameters in a single call.

Consider a report that prints a picking list for a warehouse order. The header shows the order number, the customer name, and the warehouse location, all parameters. The detail band shows the individual line items, fields. Normally you would have to fetch the order header separately and pass it as parameters. With parameter promotion, a single query can deliver everything:

```
-- First row drives the header parameters, all rows drive the detail band.
-- P_ORDER_NUMBER, P_CUSTOMER_NAME, and P_WAREHOUSE match registered parameter
-- names and will be promoted automatically. The remaining columns stay as field data.

SELECT

FROM orders o
JOIN customers c ON c.id =
JOIN warehouses w ON w.id = o.ware
JOIN order_lines ol ON ol.order_id = o.id
WHERE o.order_number = :P_ORDER_NUMBER
ORDER BY ol.bin_location ASC
```

The render request only needs the order number:

```
POST                                                    /api/v1/report-config/PickingList/render

{




  },




}
```

VeloxFactory executes the query, detects that `P_ORDER_NUMBER`, `P_CUSTOMER_NAME`, and `P_WAREHOUSE` match registered parameter names, moves their values from the first data row into the parameters map, and renders the report with a populated header and a fully populated detail band, all from one query, one request.

> **i Parameter promotion reads from every row, but only the last encountered value is kept.** For consistent results, make sure promoted columns carry the same value across all rows, as in the example above, where the order header data is identical on every line item row.

Dynamic Array

When no `ReportConnectionConfig` is assigned, VeloxFactory expects the data to arrive in the render request itself, as a JSON array in the `data` field. Each object in the array represents one row in the detail band, with keys matching the field names defined in the `.jrxml`.

This approach is ideal when the calling application already has the data in memory, when the data comes from a source VeloxFactory cannot connect to directly, or when the data structure is too dynamic to express in a fixed SQL query.

A complete render request with inline data looks like this:

```
POST                                                    /api/v1/report-config/A5_KanBan/render

{
```

```
  },
    {
                                    "description":                  "Packing      Carton
    }
  ],

}
```

For reports that print one item per page, the `data` array typically contains a single object. For list or table reports, it contains one object per row.

> **i Data types in the array must be compatible with the field types defined in the `.jrxml`.** A field declared as `java.lang.Integer` expects a JSON number, not a string. Pass values in their native JSON type, numbers as numbers, booleans as booleans.

Static Reports - No Data Needed

Reports without a detail band require neither a connection config nor a data array. The entire output is driven by parameters alone. Common examples: cover pages, certificates, summary headers, QR code labels, or any document where the layout is fixed and all variable content comes from a handful of input values.

For these reports, the render request simply omits `data` entirely:

```
POST                                /api/v1/report-config/CertificateOfConformity/render

{
```

```
                                                    "P_PRODUCT_NAME":




  },


}
```