Rendering with our powerful API

Everything the Generate PDF page does in the browser, the API does programmatically, with more control, lower overhead, and the same rendering engine underneath. A single `POST` request renders a report, optionally logs the result, and optionally dispatches a print job, all in one call.

Screenshot placeholder: `api-docs.png`
The interactive API reference at /docs, generated by Scribe.

The Render Endpoint

```
POST                                          /api/v1/report-config/{id}/render
```

The `{id}` segment accepts either the numeric ID of the `ReportConfig` or its **report name**, the name set in Jaspersoft Studio and stored in VeloxFactory. Both of these are equivalent:

```
POST                                          /api/v1/report-config/1/render
POST                                          /api/v1/report-config/A5_KanBan/render
```

Using the report name is convenient for integrations: it stays stable even if the database record is recreated, and it makes the request self-documenting.

Request Body

Field	Type	Required	Description

`outputType`	string	✓	Output format: `base64`, `url`, or `none`. See below.
`parameters`	object		Key-value map of parameter names to values. Required parameters must be present or the request is rejected.
`resourceOverrides`	object		Per-render override for `P_RESOURCE_*` image parameters, keyed by parameter name. See Dynamic Resource Overrides below.
`data`	array		Array of field objects, one per detail band row. Each object's keys must match the report's field names. Only needed when no SQL connection is configured. Can include per-row images, see Per-Row Images via Data Fields below.
`createHistoryRecord`	boolean	✓	Whether to create a `ReportHistoryRecord` for this render. Stores the full request, response, and rendered PDF.
`createPrintTask`	boolean	✓	Whether to dispatch the rendered PDF to the print service.
`printerName`	string	if print task	Target printer name. Required when `createPrintTask` is `true`.
`numberOfCopies`	integer		Number of copies passed to the print service. Defaults to `1`. VeloxFactory always renders once, the print service handles duplication.

`broadcastId`	string		WebSocket channel ID. If provided, VeloxFactory broadcasts a `ReportPrintTaskCreated` event when the print task is created. Omit to rely on polling.
`useExampleValues`	boolean		Use the stored example values instead of supplying `parameters` and `data`. Useful for testing. API-only - not available in the frontend. See below.
`laconicResponse`	boolean		Return only the essential output fields instead of the full response. Reduces payload size significantly for high-frequency rendering. See below.
`traceId`	string		Custom trace identifier for this request. Auto-generated (UUID) if not provided. Must be unique across all history records if supplied.
`mailing`	object		Send the rendering by mail after a successful render. See Mailing the rendering below.

Dynamic Resource Overrides

Every `P_RESOURCE_*` image parameter (a logo, a product photo, a line-art reference) normally resolves to whichever file is uploaded or linked on the `ReportConfig` itself. `resourceOverrides` lets a single render request swap that image out, without touching the report configuration, useful for per-customer branding, per-item product photos, or any other case where the image genuinely varies from call to call.

```
POST                                                    /api/v1/report-config/A4_AssemblyBOM/render

{
```

```
                    "parameters":          {          "P_PROJECT_NUMBER":          "PRJ-2026

              "P_RESOURCE_ASSEMBLY_IMAGE":    {    "path":    "https://cdn.example.com/parts/4471.
  },



}
```

`resourceOverrides` is an object keyed by resource parameter name. Each entry supports exactly one of two modes:

Key	Type	Description
`path`	string	A path to a readable file **inside the resources folder** of the instance, or an `https://` URL. Remote URLs are fetched directly by the render engine at render time. A path that leaves the resources folder and any scheme other than `https` are refused.
`base64`	string	Base64-encoded image bytes. VeloxFactory decodes them, writes a temporary file for the duration of the render, and deletes it immediately afterward.
`fileName`	string	Optional, used only alongside `base64` to infer the file extension.

Base64 example:

```
{









    }
  },



}
```

> ⚠ **A resource must still be linked or uploaded on the `ReportConfig` before it can be overridden.** `resourceOverrides` replaces the resolved path for a single render, it does not exempt a `P_RESOURCE_*` parameter from needing a default resource in place. Rendering still fails with the usual `"Not all resources for this report have been uploaded yet!"` error if no file or `CommonReportResource` is linked at all.

> ℹ **Do not set the same parameter in both `parameters` and `resourceOverrides`.** VeloxFactory rejects the request with a `422` if a key appears in both, to avoid ambiguous precedence.

```
{

  "errors": ["Parameter 'P_RESOURCE_LOGO' is set in both 'parameters' and 'resourceOverrides'
-                          use                          only                          one."],

}
```

Other validation errors follow the same pattern: an unknown resource parameter name, a `path` that leaves the resources folder or cannot be read, a URL that is not `https://`, or invalid `base64` all return a descriptive `422` rather than failing deep inside the render engine.

Per-Row Images via Data Fields

`resourceOverrides` covers one image per report-level `P_RESOURCE_*` parameter, the same picture used across the whole render. Some reports need the opposite: a different image for every row of the detail band, for example a product photo per line item in a parts list or an order. That case does not need `resourceOverrides` at all, it works through the `data` array directly.

An `<image>` element in a `.jrxml` is not limited to `$P{...}` parameter expressions, it accepts a field expression just like any text field:

```
<image>
                        <reportElement      x="0"      y="24"      width="120"      height="10
    <imageExpression><![CDATA[$F{partPhoto}]]></imageExpression>
</image>
```

With that in place, every object in the render request's `data` array can carry its own `partPhoto` value, resolved independently per row, using the same three delivery styles as `resourceOverrides`:

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

{



          {   "partNumber":   "4471-A",   "partPhoto":   "https://cdn.example.com/parts/4471.jpg
             {    "partNumber":    "4471-B",    "partPhoto":    "/var/www/resources/parts/4471b.
    { "partNumber": "4471-C", "partPhoto": "..."
}
  ],



}
```

Field value	Behaviour
Local path	Read directly from the filesystem by the render engine.
`http(s)://` URL	Fetched directly by the render engine at render time, one request per row.
`...`	Decoded inline, no temporary file involved.
`...`	Same as PNG. Note the MIME token must be exactly `image/jpg`.

⚠ **`...` is not recognized.** The render engine only matches the literal prefixes `` and `` - the far more common `image/jpeg` MIME token is not one of them and the image silently fails to render. Always encode JPEG uploads with the `image/jpg` token in the data URI, regardless of what the source tool actually calls the file.

i No request-level syntax needed. Unlike `resourceOverrides`, per-row images are just regular field values, there is no validation, no linked-resource requirement, no restriction on the location, and no separate object in the request body. Whatever the field resolves to at render time is handed straight to the render engine.

Output Types

The `outputType` field controls how, or whether, the rendered PDF is returned.

`base64`: The PDF is Base64-encoded and returned inline in `output.reportPdfBase64`. No file is written to disk. This is the most common choice for integrations that process the PDF immediately.

`url`: The PDF is saved to the VeloxFactory history storage and a URL pointing to that file is returned in `output.reportUrl`. Useful when the calling application needs to hand off a link rather than handle raw bytes.

`none`: No PDF data is returned at all. Valid only when `createPrintTask` is `true`, the PDF is rendered internally and handed to the print service without being exposed in the response. Use this when the response payload is irrelevant and you only care about getting the document to the printer.

> ⚠ **`outputType: none` requires `createPrintTask: true`.** Requesting output type `none` without a print task is rejected with a validation error, there would be nothing to do with the rendered PDF. A `mailing` segment does not replace the print task here, and a mail on a render that produced no PDF goes out without its attachment.

---

useExampleValues - API-only Testing Mode

When `useExampleValues: true` is set, VeloxFactory ignores any `parameters` and `data` in the request body and instead uses the example values stored on the `ReportConfig`. This is the same data used to generate the report preview in the frontend.

It is a convenient way to verify that a report renders correctly after configuration changes, no test data needs to be assembled:

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

{




}
```

> ℹ **`useExampleValues` is an API-only feature.** The Generate PDF page in the browser always requires parameters and data to be entered manually. For frontend testing, use the example

> values from the report configuration edit page.

---

laconicResponse - Minimal Output

By default, a successful render response includes the full `ReportConfig` record, the input parameters and data echoed back, and any linked `ReportHistoryRecord` and `ReportPrintTask`. For many production integrations, this detail is unnecessary, the caller only needs the PDF.

Setting `laconicResponse: true` strips the response down to the essentials: just the `traceId` and the `output` block. Everything else (`input`, `reportConfig`, `reportHistoryRecord`, `reportPrintTask`) is omitted.

The two response shapes are shown in detail in the Response Structure section below.

> **ℹ The laconic mode also suppresses `reportMeta` in error responses.** If a render fails in laconic mode, the error response contains only the error messages, the field and parameter metadata is not included.

A Complete Request

Here is a full render request for a KanBan label, dynamic array data, a parameter, history logging enabled, print task dispatched via WebSocket:

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

{




  },

    {

                                    "description":                  "Packing      Carton
```

```
    }
  ],
}
```

Response Structure

Full Response

The full response (default, `laconicResponse: false` or omitted) includes the rendered output, the echoed input, the full `ReportConfig` snapshot, and any created `ReportHistoryRecord` and `ReportPrintTask`:

```
{
                                                                "parameters":          {
        {
                                                            "description":                "Packi
```

```
      }
    ]
  },

  },

    ...
  },

  },

  }
},
    "links": { "first": null, "last": null, "prev": null, "next": null
```

```
    }
  },

}
```

Laconic Response

With `laconicResponse: true`, the response contains only what is needed to retrieve the PDF:

```
{

    }
  },
      "links":   {   "first":   null,   "last":   null,   "prev":   null,   "next":   null
```

```
    }
  },

}
```

The `traceId` is always included, it links this render to any created history record or print task, making it useful for cross-referencing even in laconic mode.

The envelope around `data` is the one every endpoint uses: a render reports a single page in `meta.pagination` and carries no page links. See Response Structure on Meet the API.

Mailing the Rendering

A render request can carry an optional `mailing` segment. Once the render succeeded, VeloxFactory sends an HTML mail through a configured mailer, using a configured mail template, with the PDF attached and, on request, an xlsx export of the same data.

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

{

                              "parameters":               {               "P_ORDER_NO":

  }
}
```

`mailer` and `mailTemplate` take the ID or the unique name of the master data record. Recipients, contact name and attachment names accept the same placeholders as the mail template, which is what makes the segment useful for reports fed by an SQL adapter: `[data.first.<column>]` addresses the rows the query actually fetched, so the recipient is a result of the render rather than an input to it.

The created mail task is returned in `reportMailTask`, next to `reportPrintTask`.

> ℹ **A failed mailing never fails the render.** Whatever goes wrong with the mail, the render response stays a success and the detail lands in `meta.mailing`. The one combination refused up front is `outputType: "preview"` together with `mailing`, a preview render deletes its own file immediately and has nothing to attach.

The full reference, including every field, the placeholder rules, the rate limits and how to mail a rendering that already exists, is on Report Mailing.

Running It on a Schedule

Everything on this page describes a request somebody makes. The same body can be handed to the Job Scheduler instead, which then makes the request itself, on a crontab expression, as the user who created the schedule and with one of that user's API tokens.

The payload of a render job is this body verbatim, with two differences: `traceId` is left out, because every run generates its own, and `outputType` is limited to `base64`, `url` and `none`. Parameters and data rows additionally accept placeholders such as `[now.date]` or `[uuid]`, resolved at the start of each run.

> ℹ See Render Jobs for the schedule around it, and The Job Scheduler for how runs are fired and logged.

Errors

Validation Errors - HTTP 422

Missing required fields, an invalid `outputType` value, or a missing `printerName` when a print task is requested all produce a `422` response with an `errors` array describing the violations.

Required parameters that are not present in the request also return a `422`, one error message per missing parameter:

```
{

                                                                    "Parameter
                                                                        "Parameter
  ],
                                  "meta":                {                "traceId":

}
```

Render Errors - HTTP 400

If the request passes validation but the renderer itself fails, for example an empty data array for a report with a detail band, an SQL query that returns no rows, a type mismatch between field values and declared Java types, or a broken SQL query, the response comes back with HTTP `400` and a `success: false` payload.

In full (non-laconic) mode, a `reportMeta` block is included in the `meta` object alongside the `traceId`. This snapshot lists the report's fields, parameters, and resources at the time of the failure, useful for diagnosing mismatches between the request payload and what the report actually expects:

```
{

      "No data delivered (or fetched via SQL using parameters) while data deliverance i
mandatory      for      reports      with      detail      bands      or      SQL      queries."
  ],

                                      {     "parameterName":     "P_RESOURCE_LOGO",     "fileN
      ],
                                {     "parameterName":     "P_ARTICLE_NUMBER",     "dataType":
```

```
      ],

                                          {     "fieldName":     "articleNumber",     "dataTyp
                                          {     "fieldName":     "description",
                                         {     "fieldName":     "moq",
                                          {     "fieldName":     "deliveryTime",          "dat
                                          {     "fieldName":     "supplier",
                                            {     "fieldName":     "barcode",
      ]
    }
  },

}
```

If a `ReportHistoryRecord` was requested (`createHistoryRecord: true`), it is still created even when the render fails, the error is recorded in the history entry, which makes it possible to review failed renders from the frontend alongside successful ones.

Revision #14
Created 2026-05-10 10:44:35 UTC by Benjamin Fischer
Updated 2026-09-23 11:05:34 UTC by Benjamin Fischer