The Basics

What VeloxFactory is, how it is installed and configured, and a first look at the frontend and the API.

What is VeloxFactory?

VeloxFactory is a web application that lets you manage and render JasperReports templates - via a clean frontend, a powerful REST API, or both. It turns the well-established JasperReports engine into something any application can consume with a single HTTP call.

> **Built for the desk, usable on the move.** VeloxFactory's views are optimised for desktop browsers, a resolution of **1600×900** or more gives the most comfortable working experience. The pages work on tablets and phones as well, which is what Scan2Print is built for: a scan station is a mobile device in the warehouse, not a workstation.

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

Where VeloxFactory Fits

JasperReports is a mature, battle-tested report engine. It produces high-quality PDFs, supports complex layouts, barcodes, images, and dynamic data - and it has been doing so reliably for decades. Integrating it directly requires a JVM on your server and Java code to drive it - which is a natural fit for Java stacks, but an overhead most PHP, Python, or C# teams prefer to avoid.

VeloxFactory sits on top of the JasperReports render engine and exposes its capabilities via a clean REST API and a management frontend. The render engine runs in pure PHP - no JVM process required on your application server. You design your report templates in Jaspersoft Studio as usual, upload them once, and from that point on, rendering is just a POST request.

```
POST                                                    /api/v1/report-config/A5_KanBan/render
Content-Type:                                                                   application/json
Authorization:                                  Bearer                                  <token>

{

                    "parameters":           {           "P_ARTICLE_NUMBER":           "456128
```

```
}
```

That is the entire integration.

Where It Came From

VeloxFactory was built by someone with a logistics and IT background who needed a simple, reliable way to generate article labels on demand - from whatever system was running at the time, without caring about the underlying report engine. The original idea was modest: a small API wrapper around JasperReports, nothing more.

It grew. A frontend to manage templates. Connection configs for live SQL data. History records for traceability. Print task dispatching, printer master data, a scan station. Mail delivery with its own mailers, templates and dispatch queue. A job scheduler, so the recurring work runs itself. A permission system. What started as "just get me a label PDF" is now a full suite, capable of enterprise integration on demand.

What VeloxFactory Does

At its core, VeloxFactory does six things:

Manages report templates. You upload `.jrxml` files, and VeloxFactory analyses them automatically, detecting parameters, data fields, and image resources. Everything is stored, versioned, and ready to render.

Connects to your data. VeloxFactory can execute SQL queries against live databases at render time. MySQL, MariaDB, PostgreSQL, and SQL Server are all supported. No live connection needed either - you can deliver data directly in the render request as a JSON array, which makes it equally useful for applications that already have the data in memory.

Renders on demand. A single API call produces a PDF. You choose the output format - Base64-encoded inline, a file URL, or silent output for print-only flows. Rendering is synchronous and fast.

Dispatches print jobs. Rendered PDFs can be forwarded directly to a print service via WebSocket, creating a traceable `ReportPrintTask` with status tracking. Labels off the printer, not just on the screen. Printers are maintained as master data and picked by name, and with **Scan2Print** a barcode scan on a phone or a handheld is enough to turn a scanned value into a

printed label.

Delivers by mail. A rendering can be sent straight to whoever needs it. A render request carries an optional `mailing` segment, and VeloxFactory sends an HTML mail through a configured mailer, using a configured template, with the PDF attached and, on request, an xlsx export of the same data. Sending is synchronous or queued, every mail is logged as a traceable `ReportMailTask`.

Works on a schedule. Not every document has somebody waiting for it. The shift report that belongs in a mailbox at six in the morning, the weekly stock list, the cleanup that keeps the history from growing without bound: recurring work is configured as a schedule with a crontab expression, maintained in the frontend or through the API like everything else. A scheduled render is a complete render request, printing and mailing included, and every run is logged with its trace id.

Who It Is For

VeloxFactory is for anyone who needs to generate documents - labels, reports, delivery notes, production sheets, certificates - reliably and programmatically. The typical home is in **logistics, production, and warehousing**, where printing article labels, kanban cards, or shipment documents is a daily operational need and downtime is not an option.

More broadly: if your application needs to produce a PDF from structured data, and you do not want to build and maintain a report engine yourself, VeloxFactory is the answer.

Deployment

VeloxFactory is a standard Laravel application. It runs wherever PHP runs, which is nearly everywhere.

Scenario	Works?
Cloud VM or VPS	**Yes**
On-premise server	**Yes**
Docker container	**Yes**
Raspberry Pi on the shop floor	**Yes**
Jaspersoft Server	Not needed

The setup is intentionally slim. PHP, a database for VeloxFactory itself, and the JasperPHP render engine, that is the stack. No application server, no JVM process to manage, no separate Jaspersoft infrastructure. VeloxFactory bundles everything it needs.

> ℹ **Want to try it first?** A live demo is available at demo.veloxfactory.dev.
>
> **Login:** `demo@veloxfactory.dev` · **Password:** `demo`

Our Vision

VeloxFactory exists because good software should not be complicated, expensive, or opaque. This page explains what we are building towards and how we think about working with the people who use it.

Simple by Design

The goal behind VeloxFactory has always been the same: take a powerful but complex technology and make it accessible to teams who just need it to work. Not teams with a dedicated Java architect. Not teams with a six-figure infrastructure budget. Teams with a problem to solve and a deadline to meet.

That philosophy shapes every decision in VeloxFactory, from the API design to the frontend workflows. Concepts should be easy to grasp. Setup should be fast. Day-to-day use should feel obvious, even for someone who has never heard of JasperReports.

Reliable and Performant

Simple does not mean limited. VeloxFactory is designed to handle real production workloads - in logistics, on the shop floor, in warehouses where a label printer needs to respond in milliseconds and cannot afford to fail during a shift.

The architecture reflects this. The stack is intentionally slim - no unnecessary layers, no bloat. Rendering is synchronous and fast. The application runs comfortably on modest hardware, including single-board computers deployed directly in production environments.

When VeloxFactory is running, it runs. That is the expectation we build to.

No Lock-In

VeloxFactory is built exclusively on open-source components. There is no proprietary cloud dependency, no mandatory subscription, no remote kill switch. You run it where you want - on your own server, in your own cloud account, on a machine on your shop floor - and it stays yours.

The report templates you design in Jaspersoft Studio are standard `.jrxml` files. The database you connect to is your own. The PDFs it produces belong to you. VeloxFactory is infrastructure you own and control, not a service you rent.

Grows With You

Most customers start with a single, focused use case - generating an article label, a delivery note, a production sheet. VeloxFactory handles that out of the box. But as requirements evolve, the platform grows with them.

History records for traceability. Print task dispatching via WebSocket. A scan station for the warehouse floor. Mail delivery with its own templates and dispatch queue. Multi-report management with a structured permission system. API access for any system that can make an HTTP call. None of these need to be in scope on day one, but they are all there when the time comes.

Beyond the built-in capabilities, VeloxFactory is extended on request. If a workflow requires something specific, it can be built in. Licenses include continued development and updates, the software does not freeze at the point of purchase.

A Real Partner

VeloxFactory is not sold by a faceless vendor with a tiered support portal. It is built and maintained by someone with deep hands-on experience in logistics IT, system integration, and shop floor environments, and that experience is available directly to every customer.

This means practical help where it matters: rolling out the application, connecting it to existing ERP or WMS systems, adapting report templates to operational realities, and making sure the implementation actually works in the environment it needs to run in, not just in a demo.

The goal is not a one-time sale. It is a long-term working relationship with customers who have real problems and need a partner who understands them.

The Full Picture

To put it plainly: VeloxFactory is powerful software at a reasonable price, built on open foundations, designed to be simple to operate, and backed by someone who will pick up the phone.

If that sounds like what you need, welcome.

Installing VeloxFactory

VeloxFactory is a standard Laravel application. The setup is intentionally slim - pull the repository, install dependencies, configure the environment, migrate the database. A standard installation is up and running in under 30 minutes.

What You Don't Need

Before listing what is required, it is worth being explicit about what is not:

- **No Java** - the render engine is pure PHP. No JVM, no Java runtime, no Java SQL drivers.
- **No Node.js / npm / Vite** - the frontend assets are pre-built and bundled with the repository.
- **No Jaspersoft Server** - VeloxFactory is entirely self-contained.
- **No Docker** - optional for the database only, not required for the application itself.

System Requirements

Requirement	Notes
PHP >= 8.2	Required extensions: `ctype`, `curl`, `fileinfo`, `filter`, `hash`, `mbstring`, `openssl`, `pcntl`, `pdo`, `pdo_sqlite`, `posix`, `redis`, `session`, `simplexml`, `tokenizer`, `zip`, `gd`, `xml`. For report database connections, additionally: `pdo_mysql` (MySQL / MariaDB), `pdo_pgsql` (PostgreSQL). SQL Server is listed separately below.
Composer	PHP dependency manager - getcomposer.org
MySQL or MariaDB	VeloxFactory's own application database. A Docker container works fine if no local instance is available.

Requirement	Notes
Redis	Required for the queue driver (Laravel Horizon). Install via `apt install redis-server` or run as a Docker container. The `php-redis` extension must also be installed: `apt install php-redis`. See Background Job Processing for details.
poppler-utils	Provides `pdftoppm`, used for generating report thumbnails. Install via `apt install poppler-utils`.
Supervisor	Keeps the three background processes running: **Horizon** (queue workers), **Reverb** (WebSocket server), and the **Scheduler**. See Background Job Processing for the full Supervisor configuration.
SMTP access	Only required for report mailing. The server has to reach the SMTP host of each configured mailer, usually on port 587 or 465. Credentials are not set up here, they are maintained as Mailers inside the application.
Apache2 or nginx	Optional reverse proxy. Not required for local or development setups.
php-sqlsrv / pdo_sqlsrv	Only required when connecting to Microsoft SQL Server as a report data source. Not needed otherwise.

Bundled Dependencies

The following components are bundled with VeloxFactory, no separate installation required:

Component	Purpose
JasperPHP	Pure PHP render engine - parses `.jrxml` and generates PDFs via TCPDF. Installed automatically by Composer.
Laravel Horizon	Redis-backed queue manager with built-in dashboard. Manages worker pools for thumbnail generation, background mail dispatch and the nightly purge jobs. Installed automatically by Composer.
PhpSpreadsheet	Builds the xlsx export that a report mailing can attach next to the PDF. Needs the PHP extensions `zip`, `gd` and `xml`. Installed automatically by Composer.
Log Viewer	In-browser Laravel log viewer for monitoring application logs.
Scribe	Generates the interactive API documentation from source annotations.
Material Design Icons	Icon set used throughout the frontend.
Ace Editor	In-browser code editor for writing SQL queries.

Component	Purpose
Logo & Color Concept	Visual identity by sinister-labs.
Jaspersoft Studio 6.21.5	Desktop IDE for designing `.jrxml` report templates. Installed separately on the designer's machine, not on the server. Download here.

Deployment Options

VeloxFactory can be deployed in two ways: self-hosted on your own infrastructure, or fully managed and operated by kiwi software. Both options deliver identical functionality, the choice depends on your team's operational preferences and existing infrastructure.

Self-Hosted

You run VeloxFactory on your own servers. The infrastructure footprint is intentionally small - no Java runtime, no application server, no container orchestration required. A single modest Linux VPS covers all but the most demanding workloads.

Component	Minimum	Recommended	Notes
VeloxFactory server (Linux)	1 vCPU, 1 GB RAM	2 vCPU, 4 GB RAM	A Hetzner CX22 (2 vCPU, 4 GB RAM, 40 GB NVMe) is more than sufficient for most deployments. Any comparable entry-level VPS or on-premises Linux server works equally well.
Disk	10 GB	40 GB	Application and report thumbnails. Scale with report volume and history retention period.
Background Printing Service (Windows)	Any Windows 10/11 machine or Windows Server with the target printers installed		Runs as a lightweight console process - a few dozen MB of memory, negligible CPU. In most deployments, an existing Windows PC on the shop floor or in the office serves as the print host. No dedicated hardware required.

> ℹ **MySQL and VeloxFactory can share the same VM** for small to medium deployments. A dedicated database server is only worth considering when multiple VeloxFactory instances share one database, or under very high render throughput.

Managed Hosting

If you prefer not to manage infrastructure yourself, VeloxFactory can be hosted and operated for you. Instances run exclusively in Germany on Hetzner hardware - reliable, fast, and GDPR-compliant by location. Updates, monitoring, and backups are handled on your behalf.

Managed hosting works well for most use cases. Depending on your network connection to the hosting location, render requests may see slightly higher latency compared to a locally deployed instance, typically not noticeable, but worth considering for high-frequency, latency-sensitive workloads such as real-time label printing on the shop floor.

For managed hosting enquiries, get in touch directly.

Configuration and data models

VeloxFactory is built around a small set of interconnected models. Understanding them is the key to understanding everything else, from how reports are set up, to how renderings are stored, to how print jobs and mails are dispatched, to how recurring work is scheduled.

Field Naming Conventions

VeloxFactory uses two naming styles consistently throughout the system. Database columns and Laravel model attributes are always `snake_case`, for example `broadcast_id`, `report_config_id`, `created_by_token_id`. The API and frontend use `camelCase` for all request and response fields, the same fields become `broadcastId`, `reportConfigId`, `createdByTokenId`.

This split is consistent without exception: whenever you are working with the API or the frontend, use camelCase. Whenever you are looking at raw database records, migration files, or server-side model attributes, expect snake_case. Throughout this documentation, all field names and JSON examples follow the API convention: camelCase.

The Model Hierarchy

Every piece of data in VeloxFactory fits into a clear hierarchy. At the top sits the **ReportConfig**, the central entity. Everything else either belongs to it, describes it, or records what happened when it was used.

```
ReportContext                                                    ←  Orga
ReportConnectionConfig
ReportConfig                                                     ←  The
      ├──   ReportParameter                                        ←
  ├── ReportField                          ← Output columns from the SQL query or data payl
```

```
        └──  ReportResource
              └── (links to) CommonReportResource     ← Shared asset, used by reports and
ReportHistoryRecord                                                          ←  Recor
   ├── ReportPrintTask                                ← A print job dispatched from a h
   │             └── (uses)  Printer                                          ←  Master
   └── ReportMailTask                                                  ←  A  mail  dispatch
                ├── (uses)  Mailer                                        ←  SMTP  account
                      ├──  (uses)  MailTemplate                                        ←
                      └──  (embeds)  CommonReportResource  ←  Graphics  placed  in  t

ScheduledJob                                ← A scheduled job: a render request, or the cle
            ├──      (renders)      ReportConfig
  ├── (runs as) User + PersonalAccessToken  ← Owner and token a render job authenticates with
    ├──  (notifies  via)  Mailer                          ←  Mailer  the  failure  mail  i
  └── ScheduledJobRun                       ← Record of one execution, with trace id and resu
```

Three of these are **master data**: `Printer`, `Mailer` and `MailTemplate` are maintained once under *Configuration* and then picked by name whenever a rendering is printed or mailed. They exist independently of any report.

ReportContext

A context is a **visual label** you assign to report configurations to group and identify them at a glance. It carries no functional logic, it is purely organisational.

Field	Description
`context_name`	Display name of the context
`context_description`	Short description
`context_text_color`	Hex color for the label text
`context_badge_color`	Hex color for the badge background
`context_border_color`	Hex color for the badge border

Every `ReportConfig` requires a context. A single context can be shared across any number of report configurations.

ReportConnectionConfig

A connection config represents a **live database connection** that VeloxFactory can use as a data source when rendering a report. When assigned to a `ReportConfig`, VeloxFactory executes the report's SQL query against this connection at render time and feeds the result rows into the report as field data.

Field	Description
`connection_name`	Friendly name for this connection
`connection_driver`	Database driver (see table below)
`connection_host`	IP address of the database server
`connection_port`	Port (required)
`connection_database`	Database / schema name
`connection_username`	Username (stored encrypted)
`connection_password`	Password (stored encrypted)
`connection_test_query`	SQL query used to verify the connection
`connection_tested`	Whether the connection has been successfully tested

Supported drivers:

Driver	Database
`mysql`	MySQL
`mariadb`	MariaDB
`pgsql`	PostgreSQL
`sqlsrv`	Microsoft SQL Server

Connection status is derived from `connection_tested`:

Status	Meaning
approved	Connection has been tested successfully
unapproved	Never tested or last test failed

> ⚠ **A report can only be rendered with a live connection if its status is *approved*.** VeloxFactory will reject render requests for reports whose connection has not been successfully tested.

Network Requirements

VeloxFactory establishes the database connection directly from the server it runs on. The target database must therefore be **reachable from that host**, ideally within the same network or at minimum via a secured private channel.

> ⚠ **Do not expose your database to the public internet.** Configuring a publicly accessible database (or opening firewall ports to make one reachable) is a significant security risk and is strongly discouraged. If VeloxFactory and your database run in separate networks, use an encrypted VPN tunnel instead: **WireGuard** or **OpenVPN** are both well-suited for this purpose.

When is a ReportConnectionConfig needed?

A `ReportConnectionConfig` is **optional** per `ReportConfig`. Whether you need one depends on how your report gets its data:

Scenario	Connection needed?
Report has no detail band (purely static layout)	No
Report has a detail band, data delivered via API at render time	No
Report has a detail band and fetches data via SQL	**Yes**

ReportConfig

The `ReportConfig` is the **core entity** of VeloxFactory. It represents a single JasperReports template (the `.jrxml` file) together with all the metadata VeloxFactory maintains about it.

Field	Description
`report_name`	Display name of the report
`report_description`	Optional description
`report_file_name`	Internal filename of the stored `.jrxml`
`report_width`	Page width in mm (extracted from the `.jrxml` on upload)
`report_height`	Page height in mm (extracted from the `.jrxml` on upload)
`report_query`	SQL query, defined in VeloxFactory and stored in the database
`report_has_detail_band`	Whether the template contains a detail band (extracted on upload)
`report_context_id`	FK → `ReportContext`

Field	Description
`report_connection_config_id`	FK → `ReportConnectionConfig` (nullable)
`report_preview_base64`	Base64-encoded preview image
`report_thumbnail_base64`	Base64-encoded thumbnail image

> ℹ **The SQL query is not defined in the `.jrxml` file.** It is written and managed directly in VeloxFactory and stored in the database as part of the `ReportConfig`. The `.jrxml` only defines which fields the query result maps to.

When a `.jrxml` file is uploaded, VeloxFactory **automatically analyses it** and creates the associated `ReportParameter`, `ReportField`, and `ReportResource` records. You then review and complete the auto-generated data, for example setting example values or uploading resource files.

ReportParameter

Parameters are the **inputs** passed into a report at render time: dates, IDs, filter values, flags, and so on.

Field	Description
`parameter_name`	Parameter name as defined in the `.jrxml`
`parameter_data_type`	Java class name (e.g. `java.lang.String`, `java.lang.Integer`)
`parameter_required`	Read from the `required` custom property in the `.jrxml`
`parameter_evaluation`	Evaluation time, extracted from the `.jrxml`
`parameter_example_value`	Read from the `exampleValue` custom property in the `.jrxml`

Both `parameter_required` and `parameter_example_value` are sourced from **custom properties** embedded in the `.jrxml` parameter definition. They can also be set manually in VeloxFactory after upload.

Three parameter name prefixes carry meaning beyond the render itself:

Prefix	Meaning
`P_RESOURCE_`	The parameter holds a graphic file asset, VeloxFactory creates a `ReportResource` for it
`P_SCAN_`	The parameter is filled by a barcode scan in Scan2Print
`P_STATIC_`	The parameter is preset once per Scan2Print session and stays untouched between scans

`P_SCAN_` and `P_STATIC_` are what makes a report usable in Scan2Print. Everything else about the report stays the same, the prefixes are a convention, not a separate model.

ReportField

Fields represent the **data columns** that populate the report's detail band, either from an SQL query result or from a data array delivered at render time.

Field	Description
`field_name`	Field name as defined in the `.jrxml`
`field_data_type`	Java class name (e.g. `java.lang.String`, `java.math.BigDecimal`)
`field_example_value`	Read from the `exampleValue` custom property in the `.jrxml`

`field_example_value` is used when rendering a preview without a live database connection.

ReportResource

Resources are **graphic file assets** (images and logos) embedded in the report template. They are referenced in the `.jrxml` via parameters following the `P_RESOURCE_` naming convention.

Field	Description
`parameter_name`	The `P_RESOURCE_` parameter name as referenced in the `.jrxml`
`resource_file_name`	Filename of the directly uploaded file (nullable)
`common_report_resource_id`	FK → `CommonReportResource` (nullable)

A `ReportResource` either holds its own uploaded file **or** it is linked to a `CommonReportResource`, never both at the same time. When linking to a common resource, the resource's own file is deleted and the common file is used in its place.

CommonReportResource

A `CommonReportResource` is a **shared graphic asset** (a company logo, a standard header image) that multiple report configurations can reference. Instead of uploading the same file to each report individually, you upload it once and link individual `ReportResource` records to it.

Field	Description
`resource_name`	Display name, **unique**, and at the same time the placeholder token used in a mail body
`resource_description`	Optional description
`resource_file_name`	Internal filename of the stored file
`resource_mime_type`	Detected mime type, filled on upload
`resource_width` / `resource_height`	Pixel dimensions, read from the file
`resource_display_width`	Optional width the graphic is rendered with in a mail
`resource_alt_text`	Alternative text for the mail graphic

A common resource serves two purposes. In a report it is the file behind a `P_RESOURCE_` parameter. In a mail template it is an **inline graphic**: the body references it by `[image.<resource_name>]`, and at send time the file is embedded into the mail itself, not linked from a server. A resource is mail capable when it is a real PNG or JPEG with readable dimensions, which is why the name has to be unique.

> **i Linking is a one-way action.** When a `ReportResource` is linked to a `CommonReportResource`, its own file is permanently deleted. Unlinking removes the reference but does not restore the file, you will need to re-upload it.

ReportHistoryRecord

A `ReportHistoryRecord` captures the **full context of a rendering**: what was requested, what was returned, and whether it succeeded. Creating a history record is **optional** and controlled by the `createHistoryRecord` flag in the render request.

Field	Description
`report_config_id`	FK → `ReportConfig`
`trace_id`	Unique identifier for this rendering run
`output_type`	How the PDF was returned (see below)
`report_api_payload`	The exact request payload sent to the render call
`report_api_response`	The full API response, stored for traceability
`report_pdf_base64`	Base64-encoded PDF content
`report_pdf_file_name`	Filename of the PDF on disk

Field	Description
`report_excel_file_name`	Filename of the xlsx export on disk, written when a mailing asked for one (nullable)
`report_thumbnail_base64`	Base64-encoded thumbnail of the first page (generated asynchronously)
`status`	Outcome of the rendering (see below)

Output types (`output_type`):

Value	Description
`base64`	PDF returned inline as a Base64 string
`url`	PDF stored as a file, a URL is returned
`preview`	Rendered for preview; file is not persisted
`none`	No PDF output (used for print-only flows)

Status values (`status`):

Value	Description
ok	Rendering succeeded, PDF received
render_fail	No errors reported, but no PDF received
error	JasperReports returned one or more errors
unknown	Status cannot be determined

History records are retained for a configurable number of days, set as a retention on the Purge Job. Thumbnails are generated asynchronously after rendering completes. Deleting a history record removes its PDF and its xlsx from disk together with the record.

ReportPrintTask

A `ReportPrintTask` represents a **print job** dispatched to a physical printer. It is always linked to a `ReportHistoryRecord`, you always print a specific past rendering, not a report config directly.

Field	Description
`report_config_id`	FK → `ReportConfig`
`report_history_record_id`	FK → `ReportHistoryRecord`

Field	Description
`printer_id`	FK → `Printer` (nullable, set when the printer was picked from the master data)
`trace_id`	Unique identifier for this print run
`broadcast_id`	WebSocket channel ID for real-time status updates (nullable)
`printer_name`	Target printer name
`copies`	Number of copies to print
`output_file_name`	Filename of the PDF sent to the printer
`output_base64_string`	Base64-encoded PDF (consumed by the print service)
`error_message`	Error detail if printing failed
`status`	Current print status (see below)

Status values (`status`):

Value	Description
pending	Created, waiting for the print service
printed	Successfully printed and confirmed
error	Printing failed
unknown	Status cannot be determined

> ℹ **Real-time updates via WebSocket** only apply when `broadcastId` is provided in the render request. The C# print service subscribes to that channel, picks up the task, executes the print job, and reports status back. Without a `broadcastId`, the task is created silently, the print service must poll for new tasks.

Printer

A `Printer` is **master data for a physical printer**. Without it, every print request had to carry the exact queue name of the print server, and everyone had to know it by heart. With it, a printer is configured once and then picked from a list.

Field	Description
`printer_display_name`	The name people see, for example `Warehouse Label 01`. Unique, 3 to 50 characters

Field	Description
`printer_name`	The queue name on the print server, for example `WH-LABEL-01`. This is what the print service receives. Unique
`printer_type`	`label-printer`, `a4-printer`, `mfc-printer` or `digital-printer`. Descriptive, it drives icon and filter
`printer_location`	Where the machine stands, for example `Hall 2, Shipping` (nullable)
`printer_broadcast_id`	WebSocket channel of the print service instance that serves this printer (nullable)
`printer_description`	Free note, for example the loaded media (nullable)
`printer_is_active`	Inactive printers keep working in existing configurations but are no longer offered in pickers

A render request may name either the display name or the queue name, VeloxFactory resolves both. A name that matches no master data record is still accepted and passed through unchanged, so integrations written before the master data existed keep working.

Mailer

A `Mailer` is **master data for one SMTP account**. It is to mailing what a `ReportConnectionConfig` is to data: the credentials live in one place, are tested from the UI, and are then referenced by name.

Field	Description
`mailer_name`	Unique name used to reference the mailer in a request
`mailer_transport`	Transport, `smtp` by default
`mailer_host` / `mailer_port`	SMTP server and port
`mailer_scheme`	`smtp`, `smtps` or empty for the transport default
`mailer_username` / `mailer_password`	Credentials, stored encrypted
`mailer_from_address` / `mailer_from_name`	Sender of every mail sent through this mailer
`mailer_reply_to`	Optional reply-to address
`mailer_timeout` / `mailer_local_domain`	Optional transport details
`mailer_rate_limit_per_minute`	Max mails per minute, empty means the window is not enforced
`mailer_rate_limit_per_hour`	Max mails per hour, `500` by default
`mailer_rate_limit_per_day`	Max mails per day, empty means the window is not enforced

Field	Description
`mailer_is_active`	Inactive mailers are not offered and are refused by the API
`mailer_tested`	Whether a test mail has been sent successfully
`mailer_description`	Free note

> ℹ **The three rate limit windows are independent and all have to allow a send.** Providers cap differently: Office 365 counts per minute, most hosters per hour, Gmail per day. A blocked send is not lost, it is queued and goes out when the window opens.

The global `MAIL_*` variables of Laravel are **not** used by the mailing pipeline. Every mail goes through a `Mailer` record, registered as a runtime mail connection for the duration of the send and removed again afterwards, exactly the way a `ReportConnectionConfig` handles its database connection.

MailTemplate

A `MailTemplate` holds the **subject and the body** of a mail, both written once and filled with placeholders at send time.

Field	Description
`mail_template_name`	Unique name used to reference the template in a request
`mail_template_subject`	Subject line, placeholders allowed
`mail_template_body_html`	HTML body, written in a rich text editor, placeholders allowed
`mail_template_header_color`	Optional override of the header bar colour
`mail_template_body_color`	Optional override of the body background
`mail_template_hide_logo`	Hides the logo in the header
`mail_template_logo_resource_id`	FK → `CommonReportResource`, an own logo for this template (nullable)
`mail_template_footer_text`	Optional override of the global footer text, placeholders allowed
`mail_template_is_active`	Inactive templates are not offered and are refused by the API
`mail_template_description`	Free note

Placeholders use the `[token]` syntax and cover the render (`[traceId]`, `[reportFileName]`), the report, the history record, the user, the API token, the printer, every parameter (`[parameters.<NAME>]`), the data rows (`[data.first.<FIELD>]`), the current time (`[now.date]`), the recipients and the graphics (`[image.<RESOURCE_NAME>]`). The full catalog is available in the editor. An unknown token resolves to an empty string and is reported, it never fails a mail.

The four whitelabel fields are each a nullable override of a global default. Left empty, the mail uses the theme colours read from the application's own stylesheet, so a mail looks like the application without anything being configured twice.

ReportMailTask

A `ReportMailTask` represents **one mail**. Like a `ReportPrintTask` it belongs to a `ReportHistoryRecord`, you always mail a specific past rendering.

Field	Description
`report_history_record_id`	FK → `ReportHistoryRecord` (nullable if no history record was created)
`mailer_id`	FK → `Mailer`
`mail_template_id`	FK → `MailTemplate`
`trace_id`	Unique identifier for this mail run
`recipients_to` / `recipients_cc` / `recipients_bcc`	Resolved address lists
`mail_subject`	The **rendered** subject, placeholders already resolved
`mail_body_html`	The **rendered** body, stored so the UI can show what was actually sent
`attachment_pdf_file_name`	The PDF on the history disk, always `<trace id>.pdf`
`attachment_excel_file_name`	The xlsx on the history disk, always `<trace id>.xlsx`
`attachment_pdf_name`	The name the PDF carries as an attachment of the mail
`attachment_excel_name`	The name the xlsx carries as an attachment of the mail
`send_async`	Whether the mail was queued instead of sent synchronously
`dispatch_after`	Set when the rate limit pushed the send into the future
`sent_at`	Timestamp of the successful send
`error_message`	Error detail if sending failed
`status`	Current mail status (see below)

Status values (`status`):

Value	Description
pending	Created, not sent yet, or waiting for a rate limit window
sent	Handed over to the SMTP server successfully
error	Sending failed
unknown	Status cannot be determined

> ℹ **A mail task never owns a file.** The PDF and the xlsx belong to the history record, stay downloadable from it and are deleted with it. The two attachment name columns only say what the files are called inside the mail. Mail tasks have their own retention on the Purge Job, independent of the history records.

The address columns hold what was actually used: a recipient given as a placeholder is stored resolved, so the mail task shows the addresses the mail went to.

ScheduledJob

A `ScheduledJob` is **one scheduled job**: either a render request, or the consolidated cleanup that keeps the database and the disk from growing without bound. Both live in the same table and differ in their type and their payload. Either kind fires on a crontab expression or once at a fixed date and time.

Field	Description
`schedule_name`	Display name, **unique**
`schedule_description`	Optional description
`schedule_type`	`render` or `purge`
`schedule_cron`	Five-field crontab expression. Empty means the schedule never fires on its own and can only be run manually
`schedule_timezone`	Timezone the expression and the single date are evaluated in. Empty means the application timezone
`schedule_run_at`	Date and time of a single run. Set in place of a cron expression, never beside one
`schedule_deactivate_after_run`	Whether the schedule sets itself inactive once its single run is done
`schedule_is_active`	An inactive schedule is never dispatched

Field	Description
`report_config_id`	FK → `ReportConfig`, render jobs only
`schedule_payload`	JSON. Render jobs: the render request body. Purge jobs: the retention map
`owner_user_id`	FK → `User`, the user a render job runs as
`owner_token_id`	FK → `PersonalAccessToken`, the token a render job authenticates with
`error_mailer_id`	FK → `Mailer`, the mailer the failure mail goes through (nullable)
`error_recipients_to` / `error_recipients_cc` / `error_recipients_bcc`	Address lists of the failure mail, plain addresses without placeholders
`schedule_last_run_at`	When the schedule last fired
`schedule_next_run_at`	When it fires next. Indexed, this is what the minute tick selects on. Empty once a single run has happened
`schedule_last_status`	Status of the most recent run

> ℹ **The owner and the token are references, not copies.** Both are foreign keys with a delete restriction, so a user or a token that a schedule runs with cannot be removed while the schedule exists. A revoked token or a deactivated owner does not delete anything, it parks the schedule and is recorded as a skipped run.

A purge job carries its retentions in `schedule_payload`, one key per step: `printTaskDays`, `mailTaskDays`, `historyDays`, `orphanedFileDays` and `runLogDays`. A key set to `null` switches its step off.

ScheduledJobRun

A `ScheduledJobRun` is **the record of one execution**. It belongs to its schedule and is deleted with it.

Field	Description
`scheduled_job_id`	FK → `ScheduledJob`, cascading delete
`trace_id`	Identifier of this run, the same id the rendering and its mail carry
`status`	Outcome of the run (see below)
`trigger`	`schedule` for a run the cron fired, `manual` for *Run now*

Field	Description
`started_at` / `finished_at` / `duration_ms`	Timing of the run
`run_as_user_id`	The user the run authenticated as, kept even after the schedule's owner changed
`report_history_record_id`	FK → `ReportHistoryRecord` the run produced (nullable)
`response_status`	HTTP status the internal render call answered with
`result_summary`	JSON summary: what a render returned, or how many records each purge step deleted
`error_message`	Error detail if the run failed
`error_mail_sent`	Whether a failure mail went out for this run
`error_mail_suppressed`	How many failure mails the throttle held back

Status values (`status`):

Value	Description
running	The run started and has not finished yet
success	Everything the run was asked to do went through
warning	The main work succeeded, something attached to it did not, for example the mail
error	The run failed
skipped	The run was not executed, for example because the owner is inactive or the token revoked

> **i The history record is referenced, never owned.** `report_history_record_id` is nulled when the record is deleted, so the history retention stays free to clean up what a run produced. The run log itself is cleaned up by the Purge Job.

Audit Trail

Every model in VeloxFactory tracks who created and last updated a record, and which API token was used. This information is available on all records via the `withAudit=true` query parameter in the API.

Field	Description
`created_at`	Timestamp of creation

Field	Description
`created_by`	User ID of the creator
`created_by_token_id`	API token ID used (if created via API)
`updated_at`	Timestamp of last update
`updated_by`	User ID of the last updater
`updated_by_token_id`	API token ID used (if updated via API)

The `creationMethod` and `updateMethod` fields in the API response (`"Frontend"` vs. `"API"`) are derived automatically, based on whether a token was present on the request.

Environment Configuration

VeloxFactory's runtime behaviour is controlled via environment variables in `.env` or as container environment variables.

Application

Variable	Default	Description
`APP_SCHEME`	`http`	URL scheme for generated links (`http` or `https`)
`API_RATE_LIMIT_PER_MINUTE`	`10`	Max API requests per minute per token
`PAGINATION_DEFAULT_COUNT`	`25`	Default number of results per API response

Queue & Redis

Variable	Default	Description
`QUEUE_CONNECTION`	`database`	Queue driver (must be set to `redis` for Horizon)
`REDIS_CLIENT`	`phpredis`	Redis client library (`phpredis` required)
`REDIS_HOST`	`127.0.0.1`	Redis server hostname or IP
`REDIS_PORT`	`6379`	Redis server port

Variable	Default	Description
`REDIS_PASSWORD`	`null`	Redis password (leave `null` if not set)
`REDIS_QUEUE_RETRY_AFTER`	`360`	Seconds before a reserved job on the `redis` connection is handed to another worker. Must stay above the longest worker timeout
`REDIS_SCHEDULER_QUEUE_RETRY_AFTER`	`1860`	The same for the `redis-scheduler` connection, which carries the long-running scheduled jobs

Horizon

Variable	Default	Description
`HORIZON_PATH`	`horizon`	URL path for the Horizon dashboard
`HORIZON_PREFIX`	derived from `APP_NAME`	Redis key prefix for all Horizon data
`HORIZON_DOMAIN`	-	Optional custom domain for the Horizon dashboard

Job Scheduler

Variable	Default	Description
`SCHEDULER_CATCHUP_GRACE_MINUTES`	`60`	A due run older than this is dropped instead of fired late, for example after the scheduler process was down
`SCHEDULER_RUN_LOCK_SECONDS`	`1800`	How long one run may hold its schedule's lock before another run may start
`SCHEDULER_ERROR_MAIL_THROTTLE_MINUTES`	`60`	At most one failure mail per schedule within this window
`SCHEDULER_PREVIEW_COUNT`	`5`	How many upcoming run dates the cron preview shows
`PURGE_SCHEDULER_RUNS_DAYS`	`90`	Retention of the run log, written into the Purge Job when it is created

Mailing

Variable	Default	Description
`MAIL_LOGO_RESOURCE`	-	Name of a mail capable `CommonReportResource`, embedded inline in the mail header

Variable	Default	Description
`MAIL_LOGO_URL`	-	Fallback used when `MAIL_LOGO_RESOURCE` is empty: an absolute, publicly reachable URL. Empty falls back to the application name as text
`MAIL_FOOTER_TEXT`	`Sent automatically by VeloxFactory.`	Footer of every mail, overridable per template
`MAIL_RATE_LIMIT_PER_MINUTE`	`0`	Default minute limit for a newly created mailer. `0` means not enforced.
`MAIL_RATE_LIMIT_PER_HOUR`	`500`	Default hour limit for a newly created mailer
`MAIL_RATE_LIMIT_PER_DAY`	`0`	Default day limit for a newly created mailer
`MAIL_THROTTLE_FALLBACK_TO_QUEUE`	`true`	A throttled synchronous send is queued instead of answered with an error
`MAIL_MAX_ASSET_SIZE_KB`	`2048`	Maximum size of a graphic that can be embedded into a mail

The standard Laravel `MAIL_*` keys stay untouched. They are the framework fallback and are not used for report mails.

Retention & Purge

The retentions themselves live on the **Purge Job**, where they are edited in the frontend or through the API. These variables supply the values the Purge Job is created with, which makes them the right place to preconfigure an instance before it is set up. Changing one afterwards has no effect on an existing schedule.

Variable	Default	Description
`PURGE_HISTORY_DAYS`	`30`	Age in days after which history records are deleted. `-1` creates the step switched off.
`PURGE_PRINTTASKS_DAYS`	`30`	Age in days after which print tasks are deleted. `-1` creates the step switched off.
`PURGE_MAILTASKS_DAYS`	`30`	Age in days after which mail tasks are deleted. `-1` creates the step switched off.
`PURGE_ORPHANED_FILES_DAYS`	`30`	Age in days after which orphaned files on disk are deleted. `-1` creates the step switched off.

Meet the frontend

VeloxFactory comes with a built-in web frontend that gives you full access to every feature without writing a single line of code. It is built with Laravel Livewire, a reactive framework that delivers a dynamic, app-like feel while keeping everything server-rendered. No separate JavaScript build, no SPA complexity.

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

Navigation

After logging in you land directly on the **Report Configs** overview, the central workspace. Dedicated scan users without edit rights for report configurations land on **Scan2Print** instead. All other sections are reachable from the main navigation.

Section	What you manage here
Report Configs	Your report templates, the heart of VeloxFactory
Report History Records	Every past rendering with status, payload, and PDF
Report Print Tasks	Print jobs dispatched to the print service
Mail Queue	Every mail that was sent or is waiting to go out, see Report Mailing
Job Scheduler	Recurring renders and the cleanup run, plus the log of every run, see The Job Scheduler
Scan2Print	Mobile scan station that turns every barcode scan into a print job, see Scan2Print: Printing labels by scanning
Report Contexts	Organisational labels and visual tags for reports
Report Connection Configs	Live database connections for SQL-driven reports
Common Report Resources	Shared graphic assets, used by reports and as inline graphics in mails
Printers	Master data for your physical printers, see Printers

Section	What you manage here
Mailers	Master data for your SMTP accounts, see Mailers
Mail Templates	Subject, body and whitelabeling of your mails, see Mail Templates
Users	User accounts and API token management

The last seven entries are grouped under **Configuration** in the main navigation. They are master data you set up once and then pick by name everywhere else.

The Lookup Pattern

Every section opens with a **list view**, powered by a Livewire Lookup component. These views work the same way across all sections, so once you know one, you know them all.

Screenshot placeholder: `report-config.index.filtered.png`
Report Config overview with the filter bar open and active filter badges.

Full-Text Search

A search bar at the top filters records instantly as you type, no page reload, no submit button. Depending on the section, the search covers names, descriptions, file names, and query text.

Column Filters

Each list view offers contextual filter dropdowns tailored to the entity. For Report Configs, for example, you can filter by Context, Data Adapter, Creator, or date ranges for creation and last update. Active filters are indicated by a badge count on the respective dropdown so you always see at a glance which filters are in play.

Pagination

Results are paginated. The default page size is controlled by the `PAGINATION_DEFAULT_COUNT` environment variable (default: 25). See Configuration and Data Models for all available environment settings.

Persistent Filters and Paging

Every list view remembers its state. The search term, all column filters (checkboxes, dropdown selections, date ranges) and the current page are stored in your session as soon as you change them. You can open a record, switch to another section and come back later, the list shows exactly the same filters and the same page as before.

The state is kept separately for each list view. Filters set on History Records do not affect Report Configs or any other section.

To start over, click the **reset button** of the respective list view, the dark button with the curved arrow icon between the search bar and the green **Filter** button. It clears the search term and all filters of this view and returns to the first page. All other views keep their filters.

> ℹ **Filters are stored per user session.** They survive page reloads and navigation, but are cleared on logout or when the session expires. Filters are never shared with other users or other browser sessions.

Working with Report Configs

The Report Config section has the richest set of actions and is where most of your day-to-day work happens.

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

Uploading a Template

Report templates are designed in **Jaspersoft Studio** (compatible version: **6.21.5**) and exported as `.jrxml` files. Once you have a `.jrxml` ready, you upload it to VeloxFactory to create a new Report Config. VeloxFactory immediately analyses the file and automatically creates all associated Parameters, Fields, and Resources, based on what is defined in the template. You do not need to add these manually.

After upload you review the auto-generated records: set example values where missing, assign a Data Adapter if the report uses SQL, and upload any resource files that were detected.

Managing Parameters, Fields, and Resources

Parameters, Fields, and Resources each have their own section within the Report Config edit view. You can edit example values, mark parameters as required, upload resource files, or link a resource to a Common Report Resource, all from the same screen.

Screenshot placeholder: `report-resource.edit.png`
Report Resource edit view with the file upload and the link to a Common Report Resource.

Generating Previews

Once all example values are set and all resource files are uploaded, you can generate a preview rendering directly from the edit view. VeloxFactory renders the report using the stored example values and stores the result as a preview image and thumbnail on the Report Config.

Rendering from the Frontend

You can trigger a full render directly from the frontend, without touching the API. A render dialog lets you fill in parameter values, choose an output type, and optionally dispatch a print job or a mail in the same step. The result is shown inline and logged as a History Record if desired.

Screenshot placeholder: `generate-pdf.create.png`
Generate PDF page with the parameter inputs and the output settings.

Report Connection Configs

When creating or editing a Connection Config, the form includes a **Test Connection** button. Use it before saving, a connection must be in status **approved** before VeloxFactory will use it for rendering. An untested or failed connection will cause render requests to be rejected.

Screenshot placeholder: `report-connection-config.edit.png`
Report Connection Config edit form with the Test Connection button.

Report History Records

The History Records section gives you a full log of every rendering that was saved. For each record you can see the status, the exact parameters and data that were submitted, the raw JasperReports response, and, if the rendering succeeded, the resulting PDF.

From a History Record you can:

- **Download** the PDF directly to your browser, and the xlsx export if the record has one
- **Trigger a print job**, creates a new Print Task for this specific rendering and dispatches it to the print service
- **Send it by mail**, opens a dialog for mailer, template, recipients and attachments and creates a new Mail Task

Screenshot placeholder: `report-history-record.show.png`
Report History Record detail page with status, payload, PDF preview and the actions.

Report Print Tasks

The Print Tasks section shows all dispatched print jobs and their current status. For each task you can inspect the target printer, number of copies, and any error messages if printing failed.

If a task ended up in error state, you can **reset** it, the task returns to **pending** and the print service will pick it up again.

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

Mailing

A rendered document can be mailed as well as printed. Three sections work together, and all three behave like every other list view described above.

Mailers hold your SMTP accounts: host, port, credentials, sender address and the dispatch rate limits. The form has a **Test Mailer** button and a **Send test mail** action, the same idea as Test Connection on a Connection Config. A mailer has to be active before it is offered anywhere.

Mail Templates hold subject and body. The body is written in a built-in rich text editor, so no HTML knowledge is needed. A button opens the **placeholder catalog**, a searchable table of every token the template can use, each one copyable with a click. A preview shows the finished mail exactly as a recipient will see it, including the colours and the logo, which can be overridden per template.

Mail Queue lists every mail task with recipients, subject, attachments, mode and status. Opening one shows the mail that was actually sent, rendered in place. From there a mail can be **repeated**: a dialog lets you correct the mailer or the recipients before it goes out again, so a mail sent to the wrong address is fixed rather than cloned.

Screenshot placeholder: `mail-template.edit.png`
Mail Template edit view with the rich text editor, the placeholder button and the whitelabel toggles.

Mailing itself is never configured here. It is requested per rendering, in the Generate PDF page, in the Scan2Print settings, in the mail dialog of a History Record, in a render job of the Job Scheduler, or in the `mailing` segment of a render request. The three sections above only provide the building blocks.

Screenshot placeholder: `mailing.history-record-modal.png`
The mail dialog of a Report History Record with mailer, template, recipients and attachments.

> ℹ Full detail is on the dedicated pages: Report Mailing, Mailers and Mail Templates.

Scheduling

Work that repeats has its own section. **Job Scheduler** lists every schedule with its type, its crontab expression, the next and the last run, and how that run ended. A **Run Log** button opens the full history of every run of every schedule, each with its trace id.

Creating a schedule starts with the type, a render job or the cleanup run, and the form follows from there. The expression is entered either in a guided mode with frequency, time and weekday pickers or as plain crontab syntax; both write the same field, and the next five run times are shown while

you type. A render job additionally takes the report, an API token, the output and print settings, and the same parameter and data row tables as the Generate PDF page, here with a switch per field for entering a placeholder such as `[now.date]` instead of a fixed value.

Run now starts a schedule immediately without moving its next slot, which is how a new one is tried before it is switched on.

Screenshot placeholder: `scheduler.overview.png`
The Job Scheduler overview with type, cron expression, next run, last run and status per schedule.

> ℹ Full detail is on the dedicated pages: The Job Scheduler, Render Jobs and The Purge Job.

Users and API Tokens

User accounts are managed under the **Users** section, accessible to administrators only. Each user can hold one or more named API tokens, which grant API access under that user's identity.

From the user edit view you can:

- Create a new token and copy it immediately after generation (it is only shown once)
- Revoke any existing token with immediate effect

Screenshot placeholder: `user.edit1.png`
User edit view, account data and API token management.

Screenshot placeholder: `user.edit2.png`
User edit view, the permission sections.

Screenshot placeholder: `create-token.modal.png`
Create API token dialog, the token is shown once.

> ⚠ **Copy your token immediately after creation.** VeloxFactory only displays the token value once. If you lose it, you will need to revoke the old token and create a new one.

Permissions

Every user in VeloxFactory has a set of permissions that controls what they can see and do, both in the frontend and via the API. Since both use the same underlying controllers, **a permission that restricts an action in the frontend restricts the exact same action via API, and vice versa.** There is no way to grant API-only or frontend-only access to a resource.

Permissions fall into two categories: **global** and **per-resource**.

Global permissions:

Permission	Effect
`global:admin`	Full access to everything, including user management
`global:use-api`	Allows use of the API and management of own API tokens

Per-resource permissions, available for each of the following resources: `report-config`, `report-connection-config`, `report-context`, `report-history-record`, `report-print-task`, `report-mail-task`, `common-report-resource`, `printer`, `mailer`, `mail-template`, `scheduled-job`:

Permission	Effect
`<resource>:full`	Full read/create/update/delete access to this resource
`<resource>:read`	View records only
`<resource>:create`	Create new records
`<resource>:update`	Edit existing records
`<resource>:delete`	Delete records

`global:admin` always implies full access to all resources and overrides any per-resource setting.

`scheduled-job:*` also covers the run log. Changing or starting a render job is limited to its owner and to administrators, because it acts with the owner's permissions; reading it is not.

Feature permissions:

Permission	Effect
`scan2print:use`	Access to Scan2Print. Implicitly includes `report-config:read`, `report-print-task:read` and `:create`, `report-history-record:create` and `:update`

Permission	Effect
`scheduled-job:create`	Creating a schedule means rendering, printing and mailing with it, so it implicitly includes `report-config:read`, `report-print-task:create`, `report-mail-task:create` and `report-history-record:create` and `:update`. `scheduled-job:full` does the same

> ℹ **API token permissions are inherited from the user.** A token does not have its own permission set, it acts with exactly the permissions of the user it belongs to. Revoking a token removes API access; the user's frontend permissions remain unaffected.

Frontend vs. API - What is the Difference?

The short answer: there is no feature difference. The frontend calls the exact same controller methods as the API. Everything you can do in the UI, you can also automate via API.

The frontend is optimised for interactive, human-driven workflows, exploring reports, reviewing history records, testing connections. The API is the right choice when you want to integrate rendering into external systems, automate repetitive tasks, or process results programmatically.

> ℹ Ready to explore the API? See Meet the API for authentication, query parameters, and a practical render example.

Meet the API

VeloxFactory ships with a fully capable REST API, and it is not an afterthought. Every action you can perform in the frontend can also be performed via the API, because **the frontend and the API share the same controller methods**. There is no separate implementation, no feature gap, no second-class citizen.

This makes the API a genuine alternative to the UI, not just an integration bolt-on. Automate report rendering in your CI pipeline, trigger prints from your ERP, manage report configurations programmatically, have VeloxFactory run the whole thing on a schedule, all with the same logic that powers the frontend you already know.

> ℹ For the full endpoint reference including all parameters and response schemas, explore the live demo API documentation at demo.veloxfactory.dev.
>
> **Login:** `demo@veloxfactory.dev` · **Password:** `demo`

Authentication

The API uses **token-based authentication** via Laravel Sanctum. Every request must include a Bearer token in the `Authorization` header.

```
Authorization:                              Bearer                                  your-api-token
```

Tokens are created and managed per user, either from the user management section in the frontend, or via the API itself (`POST /api/v1/user/{id}/create-token`). Each token can be revoked at any time. A revoked token is rejected immediately, there is no grace period.

```
{

                "errors":       ["This       API       token       has       been

}
```

The audit trail records which token was used for every create and update operation across all models, so every API action is fully traceable.

Base URL

All API v1 endpoints are prefixed with:

```
/api/v1/
```

Referencing Records

Wherever a request points at another record, it accepts either the numeric ID or the unique name of that record. These two are the same call:

```
POST                                                    /api/v1/report-config/12/render
POST                          /api/v1/report-config/Shift                       Report/render
```

The same holds inside request bodies, for report contexts, connection configs, mailers, mail templates and schedules alike, so a payload stays readable without a lookup table of IDs. A Report History Record is addressed by its ID or by its trace id, which is what makes a trace id worth carrying through your own systems.

> ℹ **A value of digits only is always read as an ID.** That is why a name cannot consist of digits only. VeloxFactory refuses such a name when a record is created or renamed, so `4711` is never ambiguous.

Response Structure

All API responses follow a consistent envelope format.

Success:

```
{
```

```
                                                "data":                         {



}
```

Error:

```
{

                                    "errors":                   ["Descriptive



}
```

The `count` field reflects the number of records in `data`, `1` for single-record responses, the actual number of returned records for collections. The `meta` field carries contextual information such as the `traceId` of a render run.

Query Parameters

Most `GET` endpoints and the render endpoint share a common set of query parameters that control what is included in the response. They are additive, combine them freely.

Parameter	Type	Default	Description
`limit`	integer	`25`	Maximum records to return. Set to `0` for all.
`withRelations`	boolean	`false`	Include related models (e.g. parameters, fields, resources on a ReportConfig).
`recursiveRelations`	boolean	`false`	Include nested relations within relations.
`withMedia`	boolean	`false`	Include Base64-encoded file content, previews, and thumbnails.
`withAudit`	boolean	`false`	Include the audit segment on each record (timestamps, users, token, method).

Example, retrieve a single report config with all relations, media, and audit data:

```
GET                    /api/v1/report-config/1?withRelations=true&withMedia=true&withAudit=true
```

Audit Segment

When `withAudit=true` is set, every record in the response carries an `audit` block:

```
"audit":                                                                                {
                                                          "createdAt":

                                                          "updatedAt":

}
```

`creationMethod` and `updateMethod` are derived automatically, `"API"` when a token was used, `"Frontend"` when the action came through the UI. A run of a scheduled job fills these exactly like a direct call does, with the owner of the schedule and the token it runs with.

Render-Specific Request Body

The render endpoint (`POST /api/v1/report-config/{id}/render`) accepts the following fields in the **request body** (JSON).

Field	Type	Required	Description
`outputType`	string	□	How to return the PDF: `base64`, `url`, `preview`, or `none`
`createHistoryRecord`	boolean	□	Whether to persist a `ReportHistoryRecord` for this render
`createPrintTask`	boolean	□	Whether to create a `ReportPrintTask` after rendering

Field	Type	Required	Description
`printerName`	string	if `createPrintTask`	Name of the target printer
`useExampleValues`	boolean		Use stored example values instead of supplying parameters manually
`parameters`	object		Key-value map of parameter names to values
`resourceOverrides`	object		Per-render override for `P_RESOURCE_*` image parameters, keyed by parameter name. A path has to point at a readable file inside the resources folder, a URL has to be `https://`, or the image is supplied as a Base64 upload. See Rendering with our powerful API for the full reference.
`data`	array		Array of data rows, for reports without a live DB connection
`numberOfCopies`	integer		Number of print copies (default: 1)
`broadcastId`	string		WebSocket channel ID, triggers real-time status updates for the print task
`traceId`	string		Custom trace ID; auto-generated as UUID if omitted. Unique across all history records
`laconicResponse`	boolean		Strip the response to just the PDF output (see below)
`mailing`	object		Send the rendering by mail once it succeeded: mailer, mail template, recipients, attachments and sync or queued dispatch. See Report Mailing for the full reference.

> ⚠ **Output type `none` requires `createPrintTask: true`.** Rendering without any output and without a printer to send it to is rejected, and a `mailing` segment does not take the place of the print task. `preview` and `mailing` are refused together as well, a preview render deletes its

> own file immediately and has nothing to attach.

Laconic Responses

By default, a render response is fully populated, it includes the echoed input, the PDF output, the linked `ReportConfig`, the created `ReportHistoryRecord`, and the `ReportPrintTask` and `ReportMailTask` if applicable. In high-frequency or bandwidth-sensitive scenarios, add `laconicResponse=true` to strip everything down to just the PDF output.

The Same Body on a Schedule

This body is also what a render job of the Job Scheduler stores. A request you have tested here can be dropped into the `payload` of a schedule unchanged, minus `traceId`, which every run generates for itself. See Render Jobs.

Rate Limiting

The API enforces a configurable rate limit per token. The default is **10 requests per minute**. When the limit is exceeded:

```
{
   "message": "Too many requests! The current rate limiting is 10 requests per minute."
}
```

The limit is adjusted via the `API_RATE_LIMIT_PER_MINUTE` environment variable, see Configuration and Data Models.

Endpoints Overview

Resource	Available operations

User	List, show, create, update, change password, enable/disable, create/revoke tokens
ReportContext	List, show, create, update, delete
ReportConnectionConfig	List, show, create, update, delete, test connection
CommonReportResource	List, show, create, update, delete
ReportConfig	List, show, create, update, delete, update resources / parameters / fields, generate previews, **render**
ReportResource	Link / unlink CommonReportResource
ReportHistoryRecord	List, show, create, update, delete, print, **mail**
ReportPrintTask	List, show, create, update, delete, set printed, set status
ReportMailTask	List, show, create, delete, repeat
Printer	List, show, create, update, delete
Mailer	List, show, create, update, delete, test configuration, send test mail
MailTemplate	List, show, create, update, delete, placeholder catalog, preview
ScheduledJob	List, show, create, update, delete, **run now**, cron preview, placeholder catalog
ScheduledJobRun	List, show, delete

A Practical Example: The Full Render Response

A render call returns a `ReportRendering` object. By default it is fully populated, you see the input you sent, the PDF output, and the linked records that were created.

```
{
```

```
      },
    },
    },
                                                                  "repo
                                                               "reportHistoryR
  },
  },
}
```

Add `laconicResponse=true` and the response collapses to just `output`, no echoed input, no embedded related records. Ideal for automated pipelines that only need the PDF.

Try it out

The demo instance gives you a fully functional VeloxFactory environment, pre-loaded with report templates and example data. Everything you can do in a production setup you can do here, with one exception: print tasks do not reach a real printer.

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

Access

> **Demo URL:** demo.veloxfactory.dev
> **Login:** `demo@veloxfactory.dev` · **Password:** `demo`

The demo account has full access to all report data, Report Configs, History Records, Print Tasks, Mail Tasks, Connection Configs, Contexts, Common Resources, Printers, Mailers and Mail Templates. It can also generate and use API tokens. User management is not available.

Guided Tour

Work through these steps in order for a complete first look at VeloxFactory.

Step 1 - Browse the Report Configs

After logging in you land on the **Report Configs** list. This is the central workspace. Each card shows a thumbnail preview of the report and its context badge.

Use the search bar to filter by name, or use the dropdown filters to narrow by context or data adapter. Try typing part of a report name, the list updates instantly, no page reload.

Screenshot placeholder: `report-config.index.filtered.png`
Report Config overview with the filter bar open and active filter badges.

Step 2 - Explore a Report Config

Click any report to open its detail view. Here you can inspect:

- **Parameters** - the inputs the report expects at render time, with their data types and example values
- **Fields** - the data columns that populate the report body
- **Resources** - image assets embedded in the template (logos, icons)
- **Preview** - the rendered thumbnail, generated from the stored example values

Scroll through the sections to get a feel for what VeloxFactory extracts from a `.jrxml` file automatically on upload.

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

Step 3 - Render a Report from the Frontend

From within a Report Config, click the **Generate PDF** button. A dialog opens with pre-filled parameter values (taken from the example values stored on the config). You can edit any of them before rendering.

Select an output type, **Base64** is fine for a quick look, and click **Generate PDF**. VeloxFactory processes the request and shows the resulting PDF inline within seconds. No API call needed, no setup required.

Screenshot placeholder: `generate-pdf.create.png`
Generate PDF page with the parameter inputs and the output settings.

Step 4 - Check the History Record

Navigate to **Report History Records**. Your render from Step 3 appears at the top of the list (if you left `createHistoryRecord` enabled in the dialog). Open it to see the full log: the exact request payload, the API response, the rendered PDF, and a thumbnail.

From here you can download the PDF directly, trigger a reprint, which creates a new Print Task, or send the document by mail, which creates a new Mail Task.

Screenshot placeholder: `report-history-record.show.png`
Report History Record detail page with status, payload, PDF preview and the actions.

Try It Yourself

Once you have completed the tour, these are good next things to explore on your own.

Upload Your Own Template

If you have a `.jrxml` file designed in Jaspersoft Studio, you can upload it directly to the demo. Go to **Report Configs → Create**, attach the file, and let VeloxFactory analyse it. Within seconds, all parameters, fields, and resources are detected and listed automatically.

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

After upload, set any missing example values, upload resource files if needed, and click **Generate Preview** to produce a thumbnail. The report is then ready to render.

Call the API Directly

The demo account has API access enabled. To get a token, open the user menu in the top right corner and go to **Account Settings → API Tokens**, copy it immediately, it is only shown once.

With the token in hand, render any report with a single HTTP request:

```
curl                              -X                              POST                              \
                                          https://demo.veloxfactory.dev/api/v1/report-config/{
                             -H               "Authorization:               Bearer               <yo
                                       -H                     "Content-Type:                     a
```

```
  }'
```

Replace `{reportName}` with the `report_name` of any config from the list (visible in the detail view). The `useExampleValues` flag tells VeloxFactory to use the stored example values instead of requiring you to pass parameters and data manually, ideal for a first test.

The response contains the rendered PDF as a Base64 string and a trace ID you can look up in History Records.

Send a Report by Mail

Mailing is the second way a rendering leaves VeloxFactory. Two mailers are waiting for you under *Configuration*.

Mailer	What it is for
Demo Log Mailer	Writes every mail into the application log instead of sending it. Ready to use, and the safe way to follow the whole path from render to mail task without anything leaving the machine.
Demo SMTP Mailer	The starting point for a real send. It carries placeholder values, `smtp.example.com` on port 587 with scheme `smtp`, user and sender `change-me@example.com`, and it is inactive so nobody picks it before it works.

To send for real, open the SMTP mailer, replace host, port, credentials and sender with the data of a mailbox you own, send a test mail from the form, and set it active. Then write a **Mail Template** with a subject and a body. The editor has a placeholder button that lists every token the template can use.

With a mailer and a template in place, open any History Record and click **Mail**, or switch on *Send results via Mail* in the Output Settings of the Generate PDF page. The mail goes out with the PDF attached, and optionally with an xlsx export of the same data. Every mail shows up in the **Mail Queue** with its status, its recipients and the body that was actually sent.

Explore Print Tasks

When rendering via the frontend or API, enable the print task option and provide any printer name (e.g. `demo-printer`). VeloxFactory creates a `ReportPrintTask` record with status **pending**. Since no print service is connected to the demo, the task stays pending, but you can inspect the full record and see how status tracking works. Completed print tasks are purged automatically after a configurable retention period.

Demo Limitations

What	Behaviour in the demo
Print tasks	Created and traceable, but no real printer connected, tasks remain **pending**. Completed tasks are purged automatically.
Mail dispatch	Mailers and Mail Templates can be created freely and a mailing can be configured end to end. The log mailer records the mail instead of sending it; whether a real mail is delivered depends on the SMTP account you enter, the demo has no mailbox of its own.
Live SQL connections	Connection Configs can be created and tested, but no external databases are reachable from the demo environment.
User management	The demo account does not have admin rights, user accounts and permissions cannot be managed.
Data persistence	Uploaded templates, history records, and other data may be reset periodically.

> ℹ **Ready to go further?** See Meet the Frontend for a full walkthrough of the UI, or Meet the API for complete API documentation.

Open Source Attribution

VeloxFactory and its companion Background Printing Service are built on open source software. We are grateful to the authors and contributors of the following packages.

VeloxFactory

Package	License	Description
laravel/framework	MIT	The Laravel PHP framework routing, ORM, queues, and the application foundation.
laravel/horizon	MIT	Queue manager and dashboard for the Redis queues that run thumbnails, mail dispatch and the nightly purge jobs.
laravel/reverb	MIT	First-party Laravel WebSocket server used to broadcast print task events in real time.
laravel/sanctum	MIT	API token authentication for the VeloxFactory REST API.
laravel/tinker	MIT	REPL for the Laravel application.
livewire/livewire	MIT	Full-stack component framework for the VeloxFactory frontend.
opcodesio/log-viewer	MIT	In-browser Laravel log viewer.
knuckleswtf/scribe	MIT	Automatic API documentation generator.
phpoffice/phpspreadsheet	MIT	Spreadsheet library used to build the xlsx export that a report mailing can attach alongside the PDF.
quilhasoft/jasperphp	MIT	Pure-PHP JasperReports renderer the engine that compiles `.jrxml` templates and produces PDFs without a Java runtime.

Background Printing Service

Package	License	Description
PdfiumViewer	Apache 2.0	.NET wrapper around the PDFium library used to render and send PDFs to Windows printers.
PDFium	BSD 3-Clause	Google's PDF rendering engine, bundled as a native binary via `PdfiumViewer.Native.x86_64.v8-xfa`.
RestSharp	Apache 2.0	HTTP client library for all API communication with VeloxFactory.
Newtonsoft.Json	MIT	JSON serialisation and deserialisation for API responses and WebSocket messages.
Serilog	Apache 2.0	Structured logging to console and rolling file (`Serilog.Sinks.Console`, `Serilog.Sinks.File`).

Get VeloxFactory

Interested in using VeloxFactory in your own environment? Get in touch, we'll figure out the right setup together.

Contact

Name	Benjamin Fischer
E-Mail	bfischer@kiwi-software.dev
Website	www.kiwi-software.dev

What to Expect

Drop a short message explaining your use case, what you need to generate, how often, and whether you'd prefer self-hosted or managed. You'll hear back within one business day.

If you want to explore the product first, the live demo is open:

> ℹ **Want to try it first?** A live demo is available at demo.veloxfactory.dev.
>
> **Login:** `demo@veloxfactory.dev` · **Password:** `demo`

Smarter Code. Frischer Blick.