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 are dispatched.

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                                                                ←
ReportConnectionConfig                                                  ←      C
ReportConfig                                                              ←    Th
        ├──    ReportParameter                                       ←    Input    va
    ├──  ReportField                          ←  Output  columns  from  the  SQL  query  o
```

```
        └──  ReportResource                                                    ←
              └──  (links  to)  CommonReportResource      ←  Shared  asset  reused  ac
ReportHistoryRecord                                                  ←  Record  of
    └──  ReportPrintTask                          ←  A  print  job  dispatched  from
```

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

Field	Description
`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`
`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.

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
`resource_description`	Optional description
`resource_file_name`	Internal filename of the stored file

> 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`

Field	Description
`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 on disk
`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 — see Environment Configuration below. Thumbnails are generated asynchronously after rendering completes.

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`
`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.

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)

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

Retention & Purge

Variable	Default	Description
`PURGE_HISTORY_DAYS`	`30`	Age in days after which history records are purged. Set to `-1` to disable.
`PURGE_PRINTTASKS_DAYS`	`30`	Age in days after which print tasks are purged. Set to `-1` to disable.
`PURGE_ORPHANED_FILES_DAYS`	`30`	Age in days after which orphaned files on disk are purged. Set to `-1` to disable.

Revision #15
Created 2026-05-10 10:46:59 UTC by Benjamin Fischer
Updated 2026-06-15 19:48:48 UTC by Benjamin Fischer