Schedule your jobs

The Job Scheduler: renders and the consolidated cleanup, on a crontab expression or at a single date and time, operable through the API.

- The Job Scheduler: work that happens without you
- Render Jobs: a render request with a clock attached
- The Purge Job: one schedule for all cleanup

The Job Scheduler: work that happens without you

Most of what VeloxFactory does happens because something asked for it: a request comes in, a report is rendered, a label goes to a printer. Some work has no caller, though. A shift report that should be in the supervisor's mailbox at six every morning. The cleanup that keeps the history from growing forever. Work like that runs on a clock, not on a request.

The **Job Scheduler** is where that clock lives. Every job is a record with a timezone and an active flag, and it says when it runs either as a crontab expression or as a single date and time. Schedules are maintained in the frontend or through the API like any other master data. VeloxFactory itself runs one fixed entry: every minute it looks for schedules that are due and hands them to the queue.

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

Two kinds of schedule

Type	What it does
Render Job	Fires a complete render request on a schedule: parameters, data rows, printing, history record and the full mailing segment. Everything a render can do from the API, a schedule can do at four in the morning.
Purge Job	The housekeeping: print tasks, mail tasks, history records, orphaned files and the scheduler's own run log, each with its own retention in days. One schedule, one run, a fixed order.

One Purge Job covers all five kinds of cleanup, so there is a single place to say when housekeeping happens and how long each entity is kept. A fresh instance already has the schedule, deactivated and without a cron expression, so nothing runs until you decide when it should.

Saying when

A schedule accepts full five field crontab syntax, the same thing every operations person already knows. You do not have to write it by hand: the editor has a guided mode with frequency, time and weekday pickers, and an expression mode for everything the guided mode cannot express. Both write into the same field, and whichever you use, the next run times are shown underneath while you type.

Expression	Meaning
`0 6 * * 1-5`	Weekdays at 06:00, the shift report.
`*/15 * * * *`	Every quarter of an hour.
`0 3 1 * *`	The first of every month at 03:00, the monthly statement.
empty	Never fires on its own. The schedule exists and can be started by hand, which is how you test one before switching it on.

Every schedule carries its own timezone, defaulting to the timezone of the instance. That matters for a company that renders for a plant in another country: the expression says 06:00 and the timezone says which 06:00.

> **Daylight saving time:** an expression that points into the hour the clock skips or repeats runs twice in October and not at all in March. For a job that has to happen exactly once a day, pick a time outside 02:00 to 03:00.

Running once, at a given time

Not every job repeats. A price list that has to go out on the day the new catalogue takes effect, a cleanup that belongs to one migration: those are a point in time, not a pattern. A schedule therefore says when it runs either with a cron expression or with a single date and time, never with both.

```
{
```

```
}
```

In the form the field is **Run once at**, with a date and a time picker, next to the switch **Deactivate after the run**. A one-time schedule shows *Once* in the overview instead of an expression, and its date on the detail page and in the next run preview.

Question	Answer
Which clock does the time follow?	The timezone of the schedule, or the timezone of the instance when the schedule has none. A value that carries its own offset, such as `2026-10-25T17:35:00+02:00`, wins over both. The API answers with the time in the schedule's timezone.
Cron and date together?	Refused with 422. To turn a one-time schedule into a recurring one, send `runAt: null` together with the cron expression.
A date in the past?	A date that is being set or changed has to lie in the future. A date that already passed does not block edits to the other fields of the schedule.
What happens after the run?	The date is used up: there is no next run and the schedule does not fire again. That also holds when the slot was discarded after a long outage of the scheduler, or when the worker died in the middle of the run.
And the schedule itself?	With **Deactivate after the run** it sets itself inactive. Without it the schedule stays active but has nothing pending; give it a new date to arm it again.
Does **Run now** use the date up?	No. A manual run is a test, the pending date stays where it is.

A one-time Purge Job counts as an active cleanup for as long as its date is still ahead, so the overview does not warn about missing housekeeping while one is armed.

What happens when a schedule is due

Every minute VeloxFactory collects the schedules whose next run has arrived, moves each one to its following slot and hands it to the queue. The order matters: the next slot is written first, so a schedule can never fire the same slot twice, whatever happens afterwards. A schedule that runs once has no following slot, so its slot is cleared instead of moved on, which is what makes it fire exactly once.

Three things are deliberately not done.

- **Missed slots are not made up.** If the scheduler was stopped for three hours, the slots in between are gone. One late run within the grace window is fired, anything older is recorded as skipped so you can see that it happened.
- **Runs of one schedule never overlap.** A render that takes longer than its own interval simply keeps running; the next run is recorded as skipped rather than queued behind it. Two different schedules do run side by side.
- **A run is not retried.** A second attempt would print a second label and send a second mail. A failed run ends as an error and says why.

Who a render job runs as

A scheduled render is a real render request, so it needs somebody to make it. That somebody is the user who created the schedule, and the schedule carries one of that user's API tokens to run with. Nothing else changes: the same permissions apply, the same audit columns are filled, and the history record shows who and which token, exactly as if the request had come over HTTP.

Two consequences are worth knowing before you build a schedule around a colleague who is about to change roles.

- If the owner is set inactive, or the token is revoked or expires, the run stops before anything is rendered. It is recorded as skipped with the reason, and the failure mail goes out.
- Because a render job acts with its owner's permissions, only the owner and administrators may change its configuration or start it by hand. Everyone with the read permission can see it and read its log.

The token is referenced, never stored. VeloxFactory keeps only the hash of a token, so a schedule points at the token record you pick from a list, and revoking that token stops the schedule the same minute.

API tokens expire after the period configured in `SANCTUM_EXPIRATION`, one year by default. A schedule whose token has lapsed stops with the same error as a revoked one and sends the configured failure mail. Owners of long-running schedules should issue themselves a fresh token before the old one expires.

Screenshot placeholder: `scheduler.render-form.png`
The form of a render job: cron builder, report config, API token, output settings, parameters with the placeholder switch and the mailing block.

The run log

Every run writes a record, whether it worked or not. The log carries the trace id, so a scheduled render can be followed through the history record, the print task and the mail task it produced, and back again.

Status	Means
`Success`	Everything the schedule asked for happened.
`Warning`	The job ran, but something inside it did not: the render worked and the mail did not, a placeholder stayed empty, one purge step failed while the others finished.
`Error`	The job itself failed. The message from the API is on the record.
`Skipped`	The run never started: the previous one was still going, the owner or the token was not usable, or the slot was older than the grace window.
`Running`	In progress right now.

A run also records whether it was fired by the clock or started by hand, so a test run is never mistaken for a scheduled one.

Being told when something breaks

A schedule that quietly stops working is worse than no schedule at all. Every schedule can name a mailer and a list of recipients for its failure mail, which goes out when a run ends as an error or is skipped. A warning does not mail: the render happened, and the detail is already on the history record or the mail task.

These recipients are plain addresses. Placeholders belong to the mail of a document, which is resolved from the render it belongs to, and a run that failed has no render to resolve anything against.

A schedule that fires every minute and fails every minute would otherwise fill a mailbox, so the failure mail is throttled to one per schedule per hour. The window is a setting of the instance, the runs in between count how many were suppressed, and the log still has every one of them.

AI assistants

The MCP server exposes the Job Scheduler the way it exposes the rest of VeloxFactory: schedules and their run log have their own tools, a crontab expression can be validated and its next run times read back before anything is saved, and a schedule can be started once by hand to see what it does. Because report configs, mailers and templates are resolved by name, a schedule is built from what a person said. *Send the shift report to the supervisor every weekday at six* is one request.

The retentions of the Purge Job have their own tool, because the API expects the complete retention map in every write and a single changed value would otherwise be refused.

Permissions

The Job Scheduler has one permission scope, `scheduled-job:*`, with the usual `read`, `create`, `update`, `delete` and `full`. The run log is covered by the same scope.

Two rules go beyond the usual pattern, both for the same reason: a schedule can act with somebody else's rights, or with nobody's.

- Creating a schedule implies being allowed to render, print and mail with it, so `scheduled-job:create` grants what a render needs, the way `scan2print:use` does for the scan station.
- A Purge Job has no owner at all, so whoever switches a cleanup step on has to hold the permission to delete what that step deletes. Turning on the history step needs the right to delete history records.

Deleting a schedule is not limited to its owner. Deleting only stops a schedule, it can never make one act with somebody's permissions.

> **The Purge Job is deletable like any other schedule.** With no active purge schedule, nothing cleans up history records, print tasks, mail tasks or orphaned files any more, and the disk keeps filling. The overview warns while that is the case. To bring the schedule back, use **New** and then **Purge Job** in the frontend, or run `php artisan scheduler:create-purge-job` on the server.

What has to be running

The Job Scheduler depends on the two background processes VeloxFactory runs in production.

Process	Why
Scheduler	The minute tick that finds due schedules. Without it nothing fires at all.
Horizon	The runs themselves. They have their own queue and their own worker, so a scheduled render that takes minutes never blocks a thumbnail or a mail.

Both belong under Supervisor in production. The Administration chapter has the details.

Render Jobs: a render request with a clock attached

A **Render Job** is a render request that VeloxFactory keeps and sends to itself. Whatever you can put into a call to the render endpoint goes into the schedule: parameters, data rows, the printer, the history record, the complete mailing segment. Nothing about rendering is special-cased for schedules, which is the point. A request you have tested once from the API behaves exactly the same at four in the morning.

Screenshot placeholder: `scheduler.render-parameters.png`
Parameters and data rows of a render job, with the placeholder switch active on a date parameter.

What you configure

Part	What it decides
Report Config	Which report is rendered. Exchangeable later in the frontend and through the API, and the payload is revalidated against the report it now points at.
API token	Which of the owner's tokens the run authenticates with. Only tokens of the owner are accepted, and only ones that are neither revoked nor expired.
Output	What the render produces, and whether a history record is written.
Printing	Printer, number of copies and broadcast id, exactly as on the generate page.
Parameters and data rows	The values the report is rendered with, entered in the same table as on the generate page, here with placeholders allowed.

Part	What it decides
Mailing	The same block as everywhere else: mailer, template, recipients, attachments, background sending.

The **type** and the **owner** are the two things a schedule keeps for its whole life. A render job stays a render job, and it stays the user who created it, because that is whose permissions every run acts with.

> **i A schedule runs even when its owner cannot use the API.** The scheduler calls the render internally and does not pass the API middleware, so `global:use-api` is not required for a schedule to fire. What does stop a run: a revoked or expired token, a token that no longer belongs to the owner, a deactivated owner, or a missing resource permission.

Output types

A schedule offers three of the four output types.

Type	Use it when
`url`	The usual choice. The PDF lands on the history disk, stays downloadable from the history record and is what a mail attaches.
`base64`	Same file, plus the document inside the history record. Comfortable, and noticeably larger per run.
`none`	Printing only. A print task is mandatory with it, the same rule the API applies.

`preview` is refused. A preview render deletes its own file again, so unattended it would produce nothing at all.

> **`none` and mailing do not go well together.** Output type `none` produces no PDF, so a mailing that wants to attach one gets a mail without its attachment. Saving is not refused, the form says so as a note, and the honest combinations are `none` with `includePdf: false`, or `url` .

Placeholders in parameters

A schedule that renders yesterday's figures every morning cannot have a fixed date in its parameters. Parameters and data rows therefore accept placeholders, resolved fresh at the start of every run.

Only those placeholders that exist *before* a render are offered here, so no `[data.first.*]` and no `[report.*]`: those describe a render that has not happened yet.

Placeholder	Resolves to
`[now.date]`, `[now.dateTime]`, `[now.time]`	The moment the run starts.
`[now.year]`, `[now.month]`, `[now.weekNumber]`, `[now.quarter]`	Period markers, for reports that select by week, month or quarter.
`[traceId]`	The trace id of this run, so the document can carry the id the log shows.
`[uuid]`	A fresh UUID, the same value everywhere it appears in one run and a new one in the next.
`[user.name]`, `[token.name]`	The owner of the schedule and the token it runs with.
`[env.hostname]`, `[appName]`, `[appUrl]`	Which instance produced the document.

The Placeholders button above the parameter table lists all of them with a description and an example, and copies a token with one click. `GET /scheduled-job/placeholders` returns the same catalogue.

> **Typed fields need the switch.** A date parameter is a date picker and a number parameter is a number field, and neither accepts `[now.date]` as text. The small button next to the field turns it into a text field for exactly that reason, and back again. Boolean parameters have no switch: a toggle has no text state, so they take no placeholders.

In the mailing block placeholders work as they do everywhere, and there the full catalogue applies: recipients, contact name and attachment names are resolved *after* the render, so `[data.first.customerEmail]` addresses the rows the report actually fetched.

Checked when you save, not at four in the morning

A schedule that cannot possibly succeed is refused while you are still looking at the form.

- Every required parameter of the report has a value or a resource override, named individually if not.

- The owner holds the permissions the run will use: reading report configs, creating print tasks when a printer is configured, creating mail tasks when there is a mailing block.
- Mailer and mail template of the mailing block exist and are active.
- Every resource override path points inside the resources folder of the instance, the same check the render applies.
- The token belongs to the owner and is neither revoked nor expired.
- The cron expression is valid five field syntax, and a date for a single run lies in the future while you are setting or changing it.
- No `traceId` in the payload: every run makes its own, and a stored one would let the first run succeed and every one after it fail.

From the API

The schedule itself is ordinary master data. Its `payload` is the body you would have sent to the render endpoint.

```
{
                                                                    "name":

                                                  "cron":               "0

                                  "parameters":     {     "P_DATE":     "[now.date]",
```

```
        }
    }
}
```

`reportConfig`, `errorMailer` and the mailer and template inside the payload accept either the numeric ID or the unique name, the same way the render endpoint does.

A job that is meant to happen once carries `runAt` in place of `cron`, and says with `deactivateAfterRun` whether it switches itself off afterwards:

```
{




}
```

Both fields come back on every read, `runAt` in the timezone of the schedule. Sending `cron` and `runAt` together is refused with 422, and `runAt: null` together with an expression turns the schedule back into a recurring one.

Endpoint	Does
`GET /scheduled-job`	Lists schedules, filterable by `type` and `isActive`.
`GET /scheduled-job/{id}`	A single schedule with its payload, its owner and its token.
`POST /scheduled-job`	Creates one. The caller becomes the owner.
`PATCH /scheduled-job/{id}`	Changes it. Type and owner are refused, everything else you leave out keeps its stored value.
`DELETE /scheduled-job/{id}`	Deletes the schedule and its run log.
`POST /scheduled-job/{id}/run`	Queues a run right now, with the owner's identity, not the caller's. A pending single run stays pending.
`POST /scheduled-job/preview-cron`	Validates an expression and answers with the next run times in a timezone.
`GET /scheduled-job/placeholders`	The catalogue a schedule may use, for building your own form.
`GET /scheduled-job-run`	The run log, filterable by schedule and status.
`GET /scheduled-job-run/{id}`	One run with its trace id, its result summary and its error message.
`DELETE /scheduled-job-run/{id}`	Deletes one run record. What the run produced is untouched.

Changing or starting a render schedule is limited to its owner and to administrators, because the run acts with the owner's permissions. Reading it, and reading its log, needs `scheduled-job:read`

like any other lookup.

Trying it before you trust it

Leave the cron expression empty, save, and press **Run now**. The schedule never fires by itself, the run is recorded as a manual one, and you get the same result you would get at six in the morning: a history record, a print task if you configured a printer, a mail task if you configured mailing, and a line in the run log with the trace id that ties them together.

A schedule that is waiting for its single date can be tested the same way. **Run now** is a test run, not the appointment, so the date stays where it is.

When it does what you want, add the expression and switch the schedule on.

The Purge Job: one schedule for all cleanup

An instance that renders all day accumulates. History records with their PDFs, print tasks that have long been printed, mail tasks whose mails arrived weeks ago, and files on the history disk that no record points at any more. Somebody has to take that out, on a clock, without anybody thinking about it.

The **Purge Job** is that somebody. One schedule, one run, five entities, five retentions.

Screenshot placeholder: `scheduler.purge-form.png`
The purge settings with one retention field per entity and the switch that turns a step off.

The five steps

Step	Deletes
1. Print Tasks	Print tasks past the retention that have reached status `Printed`. One that never printed stays, so a problem does not disappear before anybody looked at it.
2. Mail Tasks	Mail tasks past the retention, whatever their status. The mail itself is long gone, the record is the receipt.
3. History Records	History records past the retention, together with their PDF and their xlsx export.
4. Orphaned Files	Files on the history disk that no history record and no mail task refers to any more, and that are older than the retention.
5. Scheduler Run Log	The run records of the Job Scheduler itself. The run doing the deleting never deletes itself.

> **The order is fixed and not configurable.** Print tasks and mail tasks point at the history records they belong to, so the database refuses to delete a record while one of them still references it. Running the steps in this order is what makes the cleanup work, and it is the

> reason all five belong in one schedule instead of five.

Saying how long

Every step has its own retention in days, and every step can be switched off on its own.

Value	Means
`90`	Delete what is older than 90 days.
`0`	Delete everything older than right now. Useful for a one-off clear-out, dangerous as a standing setting.
switched off	That entity is never purged. The step is named under `skipped` in the run result, so the log shows it was a decision and not an accident.

A sensible starting point keeps print and mail tasks shorter than the history records they belong to, and the orphaned files shortest of all. Something like 30, 30, 90, 7 and 90 days.

Saying when

A purge schedule says when it runs the same way every other schedule does: with a cron expression for the nightly housekeeping, or with a single date and time for a cleanup that belongs to one occasion, such as the clear-out after a migration. Both are on the *Job Scheduler* page.

A purge schedule that is waiting for its single date counts as active cleanup for as long as that date is still ahead, so the overview does not warn about missing housekeeping while one is armed. Once it has run, that cleanup is over; a *0* retention that was meant for exactly that one run does not stay armed for the next night.

Where it comes from

The base setup of an instance creates the schedule under the name **Purge Job**, deactivated and without a cron expression. Its five retentions start from the `PURGE_*_DAYS` keys of the `.env` file,

which are read once at that moment. From then on the retentions live on the schedule, and the `.env` keys are not read again.

Nothing is cleaned up until you give the schedule an expression and switch it on. That is deliberate: a fresh instance should not start deleting on a schedule nobody chose.

```
0        3        *        *        *                                        every
0   3   *   *   0              Sunday   nights   only,   for   an   instance   with   little
```

Who may switch a step on

A Purge Job has no owner. It runs without any user at all, which means the usual permission check has nothing to check against. So the check moves to the moment a step is configured: whoever switches a step on, or changes its retention, has to hold the permission to delete what that step deletes.

Step	Needs
Print Tasks	`report-print-task:delete`
Mail Tasks	`report-mail-task:delete`
History Records, Orphaned Files	`report-history-record:delete`
Scheduler Run Log	`scheduled-job:delete`

Steps you leave untouched are not asked about again, so somebody may change the cron expression without holding every delete permission in the instance.

What a run tells you

The run record carries the numbers per step: how many records were eligible, how many were deleted, how many could not be. A step that throws does not stop the ones after it.

Outcome	Status
Every configured step finished	`Success`
Some steps finished, one failed or left records behind	`Warning`, with the reason in the result
Every configured step failed	`Error`, and the failure mail goes out

A history record that could not be deleted because something still references it is logged individually, not swallowed into a count.

Doing it by hand

The first four steps are each available as an artisan command as well, for a one-off cleanup or a maintenance window.

```
php artisan report-print-tasks:purge --days=30
php artisan report-mail-tasks:purge --days=30
php artisan report-history-records:purge --days=90
php artisan orphaned-files:purge --days=7
```

Run them in that order for the same reason the schedule does.

If the schedule is gone

The Purge Job is an ordinary schedule and can be deleted like any other. Nothing then cleans up, and the disk keeps filling. The Job Scheduler overview shows a warning for as long as no active purge schedule exists.

There are two ways back, and neither of them is special.

From the frontend. New on the Job Scheduler overview offers **Purge Job** next to Render Job. Pick it, set the retentions, give it a cron expression and switch it on. A purge schedule is created like any other schedule; it needs no owner and no API token, only a name that is still free and the delete permissions of the steps you switch on. The same way you would create a second one, for instance a weekly run with longer retentions next to a nightly one.

From the command line, for an instance you would rather not click through:

```
php artisan scheduler:create-purge-job
```

It creates the schedule under its original name when it is missing and does nothing when it is already there. It touches neither storage nor users, so it is safe on a live system. Afterwards give it its cron expression and switch it on again.