Prelum

Typst-based render API

A small HTTP service for rendering caller-supplied Typst projects

https://vrtfinland.github.io/prelum/
https://github.com/VRTFinland/prelum

Contents

Getting started

This guide takes you from the source repository to your first rendered PDF. The quickest route is Docker; running from Python is useful when developing Prelum itself.

Start Prelum with Docker

Build the image from the repository and start a development instance:

```
docker build -t prelum:latest .
docker run --rm -p 9870:9870 \
  -e PRELUM_ENVIRONMENT=development \
  prelum:latest
```

The service now listens at `http://localhost:9870`. Development mode supplies the public token `dev-only-insecure-token`; never use that token for a deployed service.

Render your first PDF

Leave the container running and execute this command in another terminal:

```
curl --silent --show-error --fail-with-body \
  http://localhost:9870/v1/render \
  -H "Content-Type: application/json" \
  -H "X-Prelum-Api-Token: dev-only-insecure-token" \
  --data '{"source":"#set page(width: 80mm, height: auto)\n= Hello\nRendered by Prelum."}'
\
  --output hello.pdf
```

Open `hello.pdf` to see the result. The response body is the PDF itself; it is not wrapped in JSON. If the request fails, `--fail-with-body` reports the HTTP failure and preserves Prelum's human-readable error response.

The render guide shows how to pass data and additional files. See Output formats when you need PNG, SVG, page selection or a multi-page ZIP.

Run from the source tree

For local development, install Python 3.14 or later, uv and the Typst CLI. Then run:

```
uv sync
PRELUM_ENVIRONMENT=development uv run python -m app
```

This starts the same development service on port 9870, so the render command above works unchanged. The `development`, `local`, `dev` and `test` environments supply the public development token. Every other environment must set `PRELUM_API_TOKEN`, or the service refuses to start.

Before deploying

Use a private API token, place Prelum behind suitable network controls and review the security model. The configuration guide explains resource limits, concurrency and optional integrations. For immutable release images, mounts and signature verification, see Docker and releases.

Service endpoints

Endpoint	What it is for	Authentication
`POST /v1/render`	Render a caller-supplied Typst project	API token
`GET /v1/constraints`	Read the published rules and this deployment's limits	API token
`GET /health`	Check that the process is alive	none
`GET /metrics`	Collect Prometheus metrics	none
`GET /openapi.json`	Download the generated API contract	none
`GET /docs`	Try the API with Swagger UI	none

Endpoint	What it is for	Authentication
`GET /redoc`	Browse the API reference with ReDoc	none

Authenticated requests use the `X-Prelum-Api-Token` header.

API

Render a document

Prelum turns a caller-supplied Typst document into a PDF, PNG or SVG. Send the document as JSON to `POST /v1/render`; a successful response contains the finished file itself.

Try the smallest request

This request needs only a `source`. The output defaults to PDF:

```
curl --silent --show-error --fail-with-body \
  http://localhost:9870/v1/render \
  -H "Content-Type: application/json" \
  -H "X-Prelum-Api-Token: dev-only-insecure-token" \
  --data '{"source":"= Hello\nRendered by Prelum."}' \
  --output hello.pdf
```

Open `hello.pdf` after the command completes. When using your own deployment, replace the URL and development token with its configured values.

Build a complete request

A document can also receive data and use additional project files:

```
{
  "source": "#include \"greeting.typ\"\n\n#data.name",
  "files": {
    "greeting.typ": {
      "encoding": "text",
      "content": "Hello from Prelum!"
    }
  },
  "data": {
    "name": "Ada"
  },
  "output": {
    "format": "pdf",
    "filename": "hello.pdf"
  }
}
```

Here is what happens when Prelum receives this request:

- `source` is the Typst document that Prelum compiles;
- `files` adds `greeting.typ` beside that document, so `#include "greeting.typ"` can read it;
- `data` becomes the Typst value named `data`, making `#data.name` produce `Ada`;
- `output` asks for a PDF returned with the filename `hello.pdf`.

Only `source` is required. Omit `files` and `data` when the document does not need them, and omit `output` when the default PDF is suitable.

Add project files

Use `files` for Typst modules, images, fonts or other assets that belong to this one render. Each entry has a project-relative path and an object containing:

Field	Value	Meaning
`encoding`	`text`	`content` is UTF-8 text, suitable for `.typ`, `.json` or `.csv` files
`encoding`	`base64`	`content` is Base64-encoded binary data, suitable for images or fonts
`content`	string	The text or Base64 data to write into the project

For example, a PNG can be supplied as follows and then read in Typst with `#image("assets/logo.png")`:

```
{
  "files": {
    "assets/logo.png": {
      "encoding": "base64",
      "content": "<Base64-encoded PNG data>"
    }
  }
}
```

The string placeholder must be replaced with actual Base64 data. Prelum decodes it before checking the per-file size limit.

File paths use `/` even when the client runs on Windows. Straightforward paths such as `lib/report.typ` and `assets/logo.png` work as written. Paths that escape the project, contain empty segments or conflict with another entry are rejected. See Constraints and files-key rules for the complete rules and for `GET /v1/constraints`, which reports the active deployment limits.

`main.typ` is an ordinary file key; Prelum stores `source` under an internal name that valid keys cannot collide with.

Pass data to Typst

`data` accepts any JSON value: an object, array, string, number, boolean or `null`. Prelum converts it to a Typst value without changing its structure. Access an object field such as `{"customer":{"name":"Ada"}}` with `data.customer.name` in the Typst source.

When `data` is omitted, it is an empty Typst dictionary. JSON `null` becomes Typst `none`. Internally, Prelum makes the converted value available under two equivalent names:

```
#let request = <data converted to Typst>
#let data = request
```

Use `files` for the actual bytes of an asset and `data` for its path. For example, `"logo": "assets/logo.png"` in `data` can be read with `#image(data.logo)`.

Choose the output

`output.format` accepts `pdf`, `png` or `svg`. PDF is the default. The optional `filename` controls the suggested download name; Prelum removes unsafe characters, limits its length and corrects the extension when necessary.

PNG and SVG can return one page directly or multiple pages in a ZIP archive. PDF supports page selection, version selection and conformance standards. See Output formats for the available options and examples.

Handle the response

A successful request returns the rendered bytes, not JSON. Its `Content-Type` identifies the file, and `Content-Disposition` supplies the safe filename. Direct outputs use `Content-Disposition: inline`; ZIP archives use `attachment`. Responses also include `Cache-Control: no-store`.

If the request is invalid, Prelum returns `application/problem+json`. Use its stable `code` and `context` fields in application logic, and show its human-readable `detail` to the user. See Errors for the response shape and all error codes.

Request reference

Field	Required	Behaviour
`source`	yes	Non-empty Typst source, compiled as the project entry point
`files`	no	Defaults to `{}`; maps project-relative paths to file objects
`data`	no	Defaults to `{}`; accepts any JSON value
`output`	no	Defaults to PDF; contains the format and its options

Request objects are strict. Unknown fields, misspellings and options belonging to another output format are rejected rather than silently ignored. The render endpoint validates every request; client-side checks using `/v1/constraints` are an optional way to provide earlier feedback. Both halves of that check are published: the key rules in Constraints and files-key rules, and the `output` rules with their conformance vectors in Choose an output format.

API

Check request limits before rendering

`GET /v1/constraints` tells a client what one Prelum deployment accepts in a render request. Use it to check file names and request sizes before uploading a project, so a user receives immediate feedback instead of waiting for `POST /v1/render` to reject the request.

You can skip this endpoint in a simple integration: `POST /v1/render` always validates the complete request and returns a structured error. Constraints are useful when building an upload form, giving early feedback for large projects or implementing a reusable client library.

Fetch the constraints once when the client starts and refresh them when its Prelum deployment or configuration changes. There is no need to call the endpoint before every render.

Make a request

The endpoint has no request body. Send the same API token that you use for rendering:

```
curl --silent --show-error http://localhost:9870/v1/constraints \
  -H "X-Prelum-Api-Token: dev-only-insecure-token" \
  --output prelum-constraints.json
```

A successful response is JSON. It contains two kinds of information:

- **key rules** describe valid names in the `files` object and are the same for every deployment running that version of Prelum;
- **limits** describe sizes and counts accepted by this particular deployment and may differ between environments.

Use `jq` to inspect the fields that most clients need first:

```
jq '{rules_version, max_keys, limits}' prelum-constraints.json
```

With the default development configuration, the result is:

```
{
  "rules_version": 5,
  "max_keys": 1024,
  "limits": {
    "effective_max_files": 64,
    "max_inline_files": 64,
    "max_inline_file_bytes": 1048576,
    "max_template_source_bytes": 524288,
    "max_string_bytes": 1048576,
    "max_request_body_bytes": 20971520,
    "max_output_bytes": 52428800,
    "max_output_files": 64
  }
}
```

The endpoint returns `403 Forbidden` if the token is missing or incorrect.

Use the response

For a client that sends `source`, `files` and `data` to `POST /v1/render`, the usual checks are:

- encode `source` as UTF-8 and keep it within `limits.max_template_source_bytes`;
- keep the number of `files` entries within `limits.effective_max_files`;
- validate every `files` key using the rules described below;
- keep each decoded file within `limits.max_inline_file_bytes`;
- keep each string in `data` within `limits.max_string_bytes`;
- keep the complete JSON request within `limits.max_request_body_bytes`.

For example, these keys can be used directly in a render request:

Key	Accepted	Reason
`assets/logo.png`	yes	Normalised relative path with safe characters
`lib/report.typ`	yes	Nested relative path
`main.typ`	yes	No filename is reserved
`../secret.txt`	no	Leaves the project directory
`/etc/passwd`	no	Absolute path
`assets//logo.png`	no	Empty path segment

An accepted key becomes the path that Typst uses inside the temporary project. This request makes the image available to the source as `assets/logo.png`:

```
{
  "source": "#image(\"assets/logo.png\")",
  "files": {
    "assets/logo.png": {
      "encoding": "base64",
      "content": "<base64-encoded PNG>"
    }
  },
  "output": {
    "format": "pdf"
  }
}
```

The endpoint also returns `conformance_vectors`: ready-made accepted and rejected key sets. SDK authors can run these cases against their local validator to prove that it agrees with Prelum.

Code examples

Most applications should not copy Prelum's complete validator into client code. Use `/v1/constraints` for a few inexpensive checks that improve feedback, then send the complete request to `/v1/render`. The render endpoint validates the request authoritatively and returns a structured problem response if anything is wrong.

Python 3.14

```python
import json
import os
from pathlib import Path
from urllib.error import HTTPError
from urllib.request import Request, urlopen

base_url = "http://localhost:9870"
headers = {"X-Prelum-Api-Token": os.environ["PRELUM_API_TOKEN"]}

# Fetch the deployment's limits once when the client starts.
with urlopen(Request(f"{base_url}/v1/constraints", headers=headers), timeout=10) as
response:
    constraints = json.load(response)

render_request = {
    "source": '#include "greeting.typ"\n\n#data.name',
    "files": {
        "greeting.typ": {"encoding": "text", "content": "Hello from Prelum!"},
    },
    "data": {"name": "Ada"},
    "output": {"format": "pdf"},
}
```

```
limits = constraints["limits"]

# Give immediate feedback for the most useful whole-request limits.
if len(render_request["source"].encode("utf-8")) > limits["max_template_source_bytes"]:
    raise ValueError("The Typst source is too large")
if len(render_request["files"]) > limits["effective_max_files"]:
    raise ValueError("The request contains too many files")

body = json.dumps(render_request, separators=(",", ":")).encode("utf-8")
if len(body) > limits["max_request_body_bytes"]:
    raise ValueError("The complete request is too large")

try:
    request = Request(
        f"{base_url}/v1/render",
        data=body,
        method="POST",
        headers={**headers, "Content-Type": "application/json"},
    )
    with urlopen(request, timeout=30) as response:
        Path("result.pdf").write_bytes(response.read())
except HTTPError as error:
    problem = json.load(error)
    raise ValueError(f"Prelum rejected the request: {problem['detail']}") from error
```

The important part is that `/v1/render` receives and validates the complete `render_request`. Constraints provide faster local feedback but do not replace that validation.

TypeScript

```
import { writeFile } from "node:fs/promises"

type Constraints = {
  limits: {
    effective_max_files: number
    max_request_body_bytes: number
    max_template_source_bytes: number
  }
}

const baseUrl = "http://localhost:9870"
const apiToken = process.env.PRELUM_API_TOKEN
if (!apiToken) throw new Error("PRELUM_API_TOKEN is required")

const headers = { "X-Prelum-Api-Token": apiToken }
const constraintsResponse = await fetch(`${baseUrl}/v1/constraints`, { headers })
if (!constraintsResponse.ok) throw new Error("Could not fetch Prelum constraints")
const constraints = (await constraintsResponse.json()) as Constraints

const renderRequest = {
  source: '#include "greeting.typ"\n\n#data.name',
  files: {
    "greeting.typ": { encoding: "text", content: "Hello from Prelum!" },
  },
  data: { name: "Ada" },
  output: { format: "pdf" },
}
```

```
const limits = constraints.limits

if (Buffer.byteLength(renderRequest.source, "utf8") > limits.max_template_source_bytes)
{
  throw new Error("The Typst source is too large")
}
if (Object.keys(renderRequest.files).length > limits.effective_max_files) {
  throw new Error("The request contains too many files")
}

const body = JSON.stringify(renderRequest)
if (Buffer.byteLength(body, "utf8") > limits.max_request_body_bytes) {
  throw new Error("The complete request is too large")
}

const response = await fetch(`${baseUrl}/v1/render`, {
  method: "POST",
  headers: { ...headers, "Content-Type": "application/json" },
  body,
})
if (!response.ok) {
  const problem = (await response.json()) as { detail: string }
  throw new Error(`Prelum rejected the request: ${problem.detail}`)
}
await writeFile("result.pdf", Buffer.from(await response.arrayBuffer()))
```

This is normally enough for an application integration. SDK authors who need an exact client-side mirror can implement the per-key and whole-set rules below, then verify their implementation with `conformance_vectors`.

Response fields

Field	Meaning
`rules_version`	Changes when the published key rules change
`key_pattern`	A PCRE-form expression covering the per-key rules
`max_keys`	Structural upper bound for the number of keys
`set_rules`	Rules that compare two or more keys
`conformance_vectors`	Accepted and rejected examples for testing a local validator
`limits`	Limits configured for the deployment that answered the request

The static `files-key-rules.json` file contains the same key rules for tools that cannot contact a running service. It does not contain `limits`, because a static document cannot describe an individual deployment.

The response also carries `output_rules`, the matching contract for the `output` object, under its own version counter. Choose an output format describes it, and it is published statically as `output-rules.json`.

Reference: per-key shape

Every per-key rule is published twice: once as data, and once folded into `key_pattern` for the engines that can compile it. The data is the contract. Split a key on `/` and require of every segment that it is:

- at least `min_segment_length` and at most `max_segment_length` characters — the minimum is what rejects `lib//label.typ`, `lib/` and `/lib/x.typ`, each of which splits into an empty segment;
- built only from characters matching `segment_character_class`;
- not made only of dots, while `segments_may_not_be_only_dots` is true — `.`, `..` and `...` are all rejected.

Then one rule about the key as a whole: it must be between `min_key_length` and `max_key_length` characters once joined. That is the one rule a per-segment check cannot see — what reaches the filesystem is the absolute project root plus the key, and `max_key_length` is what remains after the room reserved for that root.

That is the whole per-key contract, in any language, with no regular expression involved. It is also exactly what Prelum itself does — the service validates structurally and publishes the expression, not the other way round.

No filename is reserved. `main.typ` is an ordinary key, at the root as much as anywhere else: Prelum writes your source to a name that no valid key can spell, so there is nothing for a key to collide with.

The expression

`key_pattern` folds the three segment rules into one anchored expression. Restricting the character set makes separate absolute-path and normalisation checks unnecessary: a leading `/`, `//`, `./`, a trailing `/` and `..` all fail it.

It begins with `\A` and ends with `\z`, not `^` and `$`. In PCRE, Java and Python `$` also matches before a trailing newline, and in Ruby both `^` and `$` are line anchors whatever flags you pass, so a mirror anchored with either accepts `lib/label.typ\n` or `bad\nlib/label.typ` while Prelum rejects both. A conformance vector covers each end.

`key_pattern_flavour` is `pcre`. Java, .NET, PHP, Ruby and Python 3.14 or later take it as written. Other common runtimes need an edit or a rewrite:

- **Python 3.13 and earlier.** `\z` was only added to the `re` module in 3.14; before that it raises `re.error: bad escape \z`. Replace every `\z` with `\Z`, which in Python — unlike in PCRE and Java — already means the absolute end of the string. Do not substitute `$`.
- **JavaScript.** There is neither `\A` nor `\z`. Without the `m` flag, `^` is an absolute start anchor, but `$` may still match before a final line break. Replace the leading `\A` with `^` and every `\z` with `(?![\s\S])`, a negative lookahead that can only succeed at the absolute end.
- **Go and Rust.** `regexp` and `regex` are RE2, which has no lookahead, so `(?!...)` does not compile and the expression is unusable at all. Apply the four rules above as code instead; nothing is lost, because each one is published as data.

The `\z` occurrences are the two inside the dot-segment lookaheads and the one inside the length lookahead as well as the final anchor, so replace all of them, not only the last.

Whichever route you take, the conformance vectors are what tell you it is equivalent.

Whole-set rules

Three rules need the whole mapping and are published as `set_rules` rather than as data:

- `too_many_keys` — at most `max_keys` entries, checked first because the other two are O(keys × depth).
- `case_collision` — no two keys whose case-folded forms are equal.
- `ancestor_collision` — no key whose case-folded form equals a case-folded POSIX ancestor of another key. Fold case on both sides: `lib` and `LIB/x.typ` collide.

How many files

`max_keys` (1024) is structural and never changes with configuration. `limits.max_inline_files` is the deployment's own cap and is often lower — 64 by default. The effective cap is the smaller of the two, reported as `limits.effective_max_files`. Mirroring only one of them gets the contract wrong in one direction or the other.

Conformance vectors

`conformance_vectors` is executable, not illustrative. Each entry gives a list of keys, whether Prelum accepts them, the problem `code` if not, and the `set_rules` id where one applies. Prelum runs every vector against its own validators in CI, so a vector cannot describe behaviour the service does not have.

Run them against your mirror in your own CI. That, rather than reading the rules, is what keeps a mirror from drifting.

`rules_version` rises whenever any published rule changes. Pin it, and treat a change as a signal to re-run the vectors.

The document itself may gain fields within the same API version, so parse it leniently: ignore members you do not recognise rather than rejecting the response. A strict validator generated from the schema would otherwise fail on an ordinary release. See Fields may be added for the rule and what it excludes.

API

Choose an output format

Put an `output` object in the render request to choose what Prelum returns. If you omit it, Prelum returns a PDF with default settings.

You need	Choose
A printable or shareable document	`{"format": "pdf"}`
One page as a bitmap	`{"format": "png", "page": 1}`
One page as vector artwork	`{"format": "svg", "page": 1}`
Every page as separate images	PNG or SVG with `"archive": "zip"`

The examples on this page show only the value of the request's `output` field.

PDF

For most documents, this is all you need:

```
{
  "format": "pdf",
  "filename": "report.pdf"
}
```

The filename is optional, and PDF 1.7 is the default. Add advanced options only when the receiving system has a specific requirement:

```
{
  "format": "pdf",
  "version": "1.7",
  "standards": ["a-2b"],
  "pages": "1,3-6,8-",
  "filename": "report.pdf"
}
```

`version` accepts `1.4`, `1.5`, `1.6`, `1.7` or `2.0`. When omitted, Typst uses PDF 1.7 unless a conformance standard requires another version.

`standards` accepts one PDF/A profile and an optional compatible `ua-1`. Supported PDF/A profiles are `a-1b`, `a-1a`, `a-2b`, `a-2u`, `a-2a`, `a-3b`, `a-3u`, `a-3a`, `a-4`, `a-4f` and `a-4e`. Prelum rejects incompatible versions and combinations.

Use `pages` to return only selected physical pages. Page numbers start at one. Separate selections with commas and use a hyphen for a range: `1`, `2-5` and `8-` are valid examples. The value is limited to 256 characters and 64 selections.

Typst disables PDF tagging when pages are selected, so `pages` cannot be combined with `ua-1`, `a-1a`, `a-2a` or `a-3a`.

One PNG or SVG

Choose a page when rendering a document as one image:

```
{
  "format": "png",
  "page": 2,
  "ppi": 144,
  "filename": "page-2.png"
}
```

`page` is a positive physical page number. PNG also accepts `ppi` from 1 to 300 and defaults to 144; a larger value produces more pixels and usually a larger response. SVG does not use `ppi`.

Direct PNG and SVG output can contain only one image. If a document has several pages and `page` is omitted, Typst cannot write them to the one output file and the request fails with `template_compile_failed`.

Multiple PNG or SVG pages

Ask for a ZIP when you need several pages:

```
{
  "format": "png",
  "archive": "zip",
  "pages": "1,3-6,8-",
  "ppi": 144,
  "filename": "report-pages.zip"
}
```

`archive: "zip"` always returns a ZIP, even if only one page matches. Omit `pages` to include all pages within the deployment's configured limit. In archive mode, use `pages` rather than the single-page `page` field; conversely, `pages` is not accepted without archive mode.

ZIP entries are ordered by physical page number and named `page-01.png` or `page-01.svg`. Prelum rejects the request rather than returning a partial archive when the document has more pages than `PRELUM_MAX_OUTPUT_FILES` permits. Both the uncompressed total and the completed ZIP must fit `PRELUM_MAX_OUTPUT_BYTES`. ZIP responses use `application/zip`, `Content-Disposition: attachment` and `Cache-Control: no-store`.

Validate the options before you send them

Everything above is also published as data, so a client can refuse a bad `output` object without a round trip — and prove its check agrees with the service instead of transcribing this page.

Two places carry the same document: `output_rules` in the response of `GET /v1/constraints`, and the static `output-rules.json` for tools that run without a token. Nothing in it depends on the deployment, so the two never disagree.

`output_rules_version` rises whenever a published output rule changes. It is separate from the `rules_version` covering the files-key rules: the two contracts change at different rates, and one counter would send you back through the key rules because a PDF standard was added.

Field	What it carries
`formats`	The values `format` accepts
`pdf.formats`, `image.formats`	Which `format` values each block of rules governs. Between them they cover `formats`, so a client never has to read "not `pdf`, therefore an image" — a guess that would misapply the image rules to any format added later
`pdf.versions`, `pdf.standards`	The values `version` and `standards` accept
`pdf.max_standards`	How many entries `standards` may hold
`pdf.pdf_a_version`	Which PDF version each PDF/A standard requires. A standard absent from this map is not a PDF/A profile — which is how `ua-1` is the one that may accompany one
`pdf.tagged_standards`	The standards that require tagging, and so cannot be combined with `pages`
`pdf.pdf_a_4_standards`	The PDF/A-4 family, which `ua-1` is incompatible with
`image.archives`, `image.min_page`	What `archive` accepts, and the lowest `page`
`image.png`	`min_ppi`, `max_ppi` and the `default_ppi` used when `ppi` is omitted
`page_selection`	The `pages` grammar: `max_length` with the `max_length_unit` it counts, `max_selections`, and `selection_pattern`, which each comma-separated selection must match in full
`rules`	The rules that need more than one field to decide, each with a stable `id`
`conformance_vectors`	Executable examples — see below

Which rule rejected the request

Every rule in `rules` answers `invalid_request` at the same place in the body, so the response carries the rule's `id` in `context.rule`. Branch on that rather than on `detail`, which is prose and may be reworded.

`rule` always names the failure `detail` reports. A request that breaks one of these rules and also something they do not name — a missing `source`, say — reports that instead and carries no `rule` until it is fixed.

`id`	The request is refused when
`page_selection_too_long`	`pages` is longer than `max_length`
`page_selection_too_many_segments`	`pages` holds more than `max_selections` comma-separated selections
`page_selection_malformed`	A selection does not match `selection_pattern` in full
`page_range_end_precedes_start`	A closed range in `pages` ends before it starts
`duplicate_standards`	`standards` names the same standard twice
`multiple_pdf_a_standards`	More than one PDF/A standard is selected
`ua_1_with_pdf_a_4`	`ua-1` is combined with `a-4`, `a-4f` or `a-4e`
`version_conflicts_with_standard`	An explicit `version` is not the one the selected PDF/A standard requires. Omitting `version` is always accepted
`ua_1_with_pdf_2_0`	`ua-1` is combined with an explicit `version` of `2.0`
`pages_with_tagged_standard`	`pages` is combined with a standard that requires tagging
`pages_requires_archive`	An image output uses `pages` without `archive`
`page_with_archive`	An image output combines `page` with `archive`

The rules are listed in the order Prelum applies them, and `rule_evaluation` says so in the document itself. An object that breaks two of them is reported against the earlier one, so a client checking them in another order disagrees about which constraint you broke while agreeing that you broke one.

The field-level rules are not in this list. An unknown field, a `ppi` outside its range or a `format` that does not exist is described by the OpenAPI schema, and the validation error's own `type` already tells those apart.

Conformance vectors

`conformance_vectors` holds accepted and rejected `output` objects with the answer Prelum gives:

```
{
  "output": { "format": "pdf", "version": "1.4", "standards": ["a-2b"] },
  "accepted": false,
  "code": "invalid_request",
  "rule": "version_conflicts_with_standard"
}
```

An accepted vector has no `code` or `rule`. A rejected one always carries `code`, and carries `rule` when one of the rules above refused it.

Run them against your own validator in your own test suite. Prelum runs every one of them against the service on each build, and against a validator built from this document alone — so a vector that disagreed with the service would fail here rather than in your deployment. That is the whole point of publishing them: your check is tested against ours rather than copied from it.

API

Handle errors

When Prelum cannot render a request, it returns JSON describing what went wrong. Alongside the HTTP status and stable `code`, every response names `origin`: whose failure this is, so a client can decide without keeping its own list of codes — see "Whose failure it is" below for the detail. The human-readable `detail` field is for showing the user, not for branching on.

Typical responses fall into these groups:

Status	What it usually means	What the client should do
400 or 422	The request, file data or Typst source is invalid	Correct the request before trying again
403	The API token is missing or incorrect	Fix the client configuration
408	Typst exceeded the render timeout	Simplify the document or ask the operator about the limit
413	An input or output limit was exceeded	Read `code` and `context` to find the exact limit
429	All render slots remained busy	Wait for the `Retry-After` duration, then retry
500 or 503	Prelum or its infrastructure failed	Retry cautiously and alert the service operator

Example response

Every error uses `application/problem+json` and has the same top-level fields:

```
{
  "code": "template_file_too_large",
  "origin": "template",
  "title": "Template Too Large",
  "status": 413,
  "detail": "Template file 'assets/logo.png' size 1500000 bytes exceeds limit 1048576",
  "instance": "https://prelum.example/v1/render",
  "context": {
    "key": "assets/logo.png",
    "size": 1500000,
    "limit": 1048576
  }
}
```

In application logic, compare `code` rather than `title` or `detail`: those two fields are prose and may be reworded. `context` contains details such as the measured size, configured limit or affected file key. It is `{}` when the error needs no additional values.

Several different conditions use status 413, so the status alone cannot tell a client what was too large. The `code` distinguishes the request body, source, one file, one data string and the rendered output.

Fields may be added

Request objects are strict: an unknown field is a mistake and is rejected rather than ignored. Responses are the opposite. Any response body — an error, or the `/v1/constraints` document — may gain fields within the same API version, so a client must ignore the ones it does not recognise, and a validator that rejects unknown members will break on an ordinary release. The published schema says so in both directions: every response schema is declared open, every request schema closed.

What will not change without a new API version: an existing field disappearing, changing type or changing meaning, an existing `code` answering a different status, or a documented `origin` value being withdrawn.

Context fields

A field means the same thing wherever it appears.

Caller text in prose, validation errors and `path` escapes lone surrogates as literal `\uXXXX` text before shortening, so these fields can always be encoded as UTF-8.

Field	Type	Meaning
`limit`	integer	The configured limit that was exceeded, in the unit the code counts
`size`	integer	A measured size in bytes
`count`	integer	A measured or requested number of things
`key`	string	One `files` key, complete and exactly as sent
`rule`	string	The `id` of the rule behind the failure `detail` reports — a whole-set files-key rule or an output-option rule — as published by `GET /v1/constraints`. Branch on it rather than on `detail`. A request breaking an identified rule *and* something the rules do not name reports the latter, and carries no `rule` until it is fixed
`path`	array	Where a value sits in `data`: object keys as strings, array indices as integers. String segments longer than 80 characters are shortened with **…**
`subject`	string	`key` or `value`: which string at `path` is at fault. An oversized object key is its own last path segment, so the path alone cannot say
`declared_size`	integer	The `Content-Length` the caller sent; not a measurement
`timeout_secs`	integer	The configured render timeout in seconds
`retry_after`	integer	Seconds to wait before retrying, equal to the `Retry-After` header
`errors`	array	Request validation errors as `{loc, msg, type}`, at most 20 of them. `loc` is shortened like `path`: string segments over 80 characters end in **…**, array indices stay integers. `msg` is prose that may quote the offending value, and is shortened the same way past 200 characters. Like `detail`, it may be reworded — including its leading words — so branch on `type`, or on `context.rule` where one is given
`errors_total`	integer	How many validation errors there were, present only when `errors` holds fewer than that

Codes

Status	`code`	`origin`	`context`	Cause
400	`invalid_request`	`request`	`errors`, absent when the body could not be parsed at all; `rule` when an output-option rule rejected it	Missing, malformed, unknown or format-inappropriate request field, or a body no JSON parser accepts
400	`unsupported_format`	`request`	`errors`	`format` outside `pdf`, `svg` and `png`
400	`invalid_template_path`	`request`	`errors`	Unsafe, non-normalised or otherwise invalid `files` key
400	`invalid_file_data`	`request`	`key` or `count` + `limit`, `rule` when a whole-set rule rejected it, and `errors` only when validation raised it (see below)	Malformed base64, excessive or colliding entries, or non-UTF-8 text
403	`forbidden`	`request`	—	Missing or incorrect API token
404	`not_found`	`request`	—	No route at the requested path
405	`method_not_allowed`	`request`	—	The route exists but not for this method; `Allow` names the ones it has
408	`render_timeout`	`request`	`timeout_secs`	Typst exceeded the render timeout
413	`request_too_large`	`request`	`limit`, `declared_size` when `Content-Length` was sent	JSON body exceeded the request-body limit
413	`template_source_too_large`	`template`	`size`, `limit`	`source` exceeded its limit
413	`template_file_too_large`	`template`	`key`, `size`, `limit`	One `files` entry exceeded its limit after decoding

Status	`code`	`origin`	`context`	Cause
413	`string_too_large`	`request`	`path`, `subject`, `size`, `limit`	One string in `data` exceeded its limit
413	`output_too_large`	`request`	`size`, `limit`	Direct output, aggregate images or ZIP exceeded the output limit
413	`page_selection_too_large`	`request`	`count`, `limit`	`pages` selects more pages than an archive may hold; nothing was rendered
413	`too_many_output_files`	`request`	`limit`	The document produced more pages than an archive may hold
422	`template_compile_failed`	`template`	—	Caller-supplied Typst or page selection could not produce the requested output, including a render killed for exhausting memory
429	`render_queue_full`	`capacity`	`retry_after`	No render slot became free before the queue deadline
500	`render_failed`	`service`	—	Typst was killed by a signal the render's own memory limit does not explain, or reported success but produced no expected output
503	`service_unavailable`	`service`	—	Unexpected service or infrastructure failure

`invalid_file_data` carries `key` when one `files` entry is at fault — malformed base64, text that is not UTF-8, a key that escapes the project root or collides with another in case or as a directory, or a layout that cannot be written — and `count` with `limit` when there are too many entries. Non-UTF-8 `source` carries neither. `errors` accompanies these fields only for the causes request validation detects (the key collisions and the structural key-count cap); the causes the renderer detects have no validation error list to publish. `invalid_template_path` never carries the key: it has not passed validation, so it is not echoed; `detail` names it, shortened.

A 429 includes `Retry-After`; callers should wait at least that many seconds before retrying.

Whose failure it is

`origin` says what a failure is attributable to, so a client decides with one field instead of keeping its own list of codes. A code added later classifies itself on the day it ships.

`origin`	Meaning	What the client should do
`request`	The request itself, or the document it asks for	Correct the request
`template`	The inline template the request shipped	Correct the template
`capacity`	Prelum is shedding load by design; nothing is broken	Wait for `Retry-After`, then retry
`service`	Prelum itself failed	Retry cautiously and alert the service operator

`service` is the only value that means Prelum failed, and it is the only one worth alerting on. `capacity` in particular is designed behaviour: a busy service is not a broken one, and a client that pages on it pages on ordinary load.

`request` and `template` are both the client's side of the exchange — `source` and `files` arrive in the same body — so the split says which half to correct, not which is more serious. Neither is ever Prelum's failure.

The complete mapping is published as `error-codes.json`: every code with its status and origin, generated from the service's own definitions.

Privacy of error responses

Request bodies are never echoed. Typst diagnostics are neither returned nor logged because they can quote caller-supplied source. Caller-controlled paths and validation messages are shortened and made safe for UTF-8 before appearing in a response; see the context descriptions above for the few bounded values that are returned exactly.

API

Explore the API with OpenAPI

Prelum publishes an OpenAPI description of its endpoints, request fields and responses. Use the interactive documentation to try a request by hand, or download the schema when generating or validating client code.

Try an endpoint in the browser

A running Prelum instance provides:

- Swagger UI at `/docs`, with forms for sending requests;
- ReDoc at `/redoc`, for browsing the API reference;
- the raw OpenAPI document at `/openapi.json`.

Rendering and constraint requests still require an `X-Prelum-Api-Token` header. The interactive documentation itself is public.

Download the contract

Download the schema generated for this documentation, or fetch `/openapi.json` from the deployment your client will use. The deployment copy is useful when a client needs to confirm that it is talking to the expected Prelum version.

The schema describes the stable request shape. Deployment-specific file counts and byte limits come from authenticated `GET /v1/constraints`; see Constraints and files-key rules.

How it stays current

Prelum generates the schema directly from the FastAPI application rather than maintaining it by hand. `make docs-build` and `make docs-serve` regenerate the static copy, and the strict documentation build fails if it cannot consume the result.

OPERATIONS

Configure Prelum

Prelum reads its configuration from environment variables beginning with `PRELUM_`. The defaults are suitable for trying the service locally; a deployed service must at least set a private API token.

For example:

```
PRELUM_API_TOKEN=replace-with-a-secret \
PRELUM_PORT=9870 \
uv run python -m app
```

Invalid configuration stops the service during startup, before it accepts requests.

Authentication and listener

Variable	Default	What it controls
`PRELUM_API_TOKEN`	unset	Shared value required in the `X-Prelum-Api-Token` request header
`PRELUM_ENVIRONMENT`	unset	Enables the public development token only for `development`, `local`, `dev` or `test`
`PRELUM_BIND`	`0.0.0.0`	Address on which the HTTP server listens
`PRELUM_PORT`	`9870`	HTTP port

Outside the four named development environments, `PRELUM_API_TOKEN` is required. When `PRELUM_ENVIRONMENT` is unset, token mode falls back to `PRELUM_SENTRY_ENVIRONMENT`; set the environment explicitly if that fallback would be surprising.

Capacity and size limits

Lower these values to match the memory, CPU and request limits of the surrounding deployment. Byte limits measure the actual UTF-8, decoded file or generated output bytes described below.

Variable	Default	What is limited
`PRELUM_RENDER_TIMEOUT_SECS`	`15`	Runtime of one Typst process
`PRELUM_MAX_CONCURRENT_RENDERS`	`2`	Typst processes running at the same time
`PRELUM_MAX_QUEUE_WAIT_SECS`	`10`	Wait for a render slot before returning 429
`PRELUM_RETRY_AFTER_SECS`	`5`	Floor for the jittered `Retry-After` response
`PRELUM_MAX_REQUEST_BODY_BYTES`	`20971520`	Complete JSON request body
`PRELUM_MAX_TEMPLATE_SOURCE_BYTES`	`524288`	UTF-8 bytes in `source`
`PRELUM_MAX_INLINE_FILES`	`64`	Files in one request; structural validation has a fixed ceiling of 1024
`PRELUM_MAX_INLINE_FILE_BYTES`	`1048576`	One file after text encoding or Base64 decoding
`PRELUM_MAX_STRING_BYTES`	`1048576`	One string or object key in `data`
`PRELUM_MAX_OUTPUT_FILES`	`64`	PNG or SVG pages in one ZIP
`PRELUM_MAX_OUTPUT_BYTES`	`52428800`	Direct output, aggregate images and completed ZIP
`PRELUM_MAX_RENDER_MEMORY_BYTES`	`536870912` in the image, otherwise unset	Heap of one Typst process

`PRELUM_MAX_RENDER_MEMORY_BYTES` applies `RLIMIT_DATA` to each Typst process, so a template that allocates without bound fails against its own limit instead of the container's OOM killer — which chooses its victim by `oom_score` and may pick the service rather than the render that caused it. It is applied through the `prlimit` wrapper from util-linux, which exists only on Linux, so the setting is unset by default and enabled in the published image. Because neither Docker nor Kubernetes can remove a variable the image sets, an empty value counts as unset — set `PRELUM_MAX_RENDER_MEMORY_BYTES=` to run the image somewhere the wrapper is unavailable. When it is set, the service probes the wrapper at startup and refuses to serve if it cannot honour the value; that is deliberate, because every

way the wrapper can fail exits the same way a failed compilation does, and would otherwise answer `422` to every caller for what is a deployment fault. The floor is 128 MiB: a document of a few hundred sections needs more than 64 MiB, so a lower limit would fail legitimate renders the same way it fails runaway ones.

Sizing the container

`PRELUM_MAX_CONCURRENT_RENDERS` multiplies almost every other memory cost, which is why it defaults to 2 rather than something larger. Each render in flight holds three things at once:

Per render in flight	Default	Bounded by `RLIMIT_DATA`?
The Typst process's heap	512 MiB	yes
The parsed request body	up to 20 MiB	no — it is this service's memory
The finished output, held whole before it is sent	up to 50 MiB	no — same

At the defaults that is roughly 1.2 GiB of ceiling plus the interpreter's own baseline, so about 2 GiB is a sensible container limit. Give the container less and its OOM killer fires before any render reaches its own limit, and it need not pick the render that caused the pressure. Raising concurrency without raising the container's memory in step is the most common way to reintroduce exactly the failure `PRELUM_MAX_RENDER_MEMORY_BYTES` exists to prevent.

Note also what `RLIMIT_DATA` does not reach: it bounds the heap and anonymous mappings of the Typst process only. Memory-mapped font files are outside it, as is the scratch space under `PRELUM_TEMP_ROOT` when that path is memory-backed (see below).

Authenticated clients can read the effective public limits from `GET /v1/constraints`. This lets them provide early feedback without copying deployment configuration; see Constraints and files-key rules.

Typst and shared resources

Variable	Default	What it controls
`PRELUM_CLI_PATH`	`typst`	Typst executable
`PRELUM_FONT_PATH`	unset	One additional, recursively searched font directory
`PRELUM_LOCAL_PACKAGE_PATH`	unset	Directory containing versioned local Typst packages
`PRELUM_ALLOW_TYPST_NETWORK`	`false`	Whether Typst may download remote packages
`PRELUM_TEMP_ROOT`	`/tmp`	Directory each render builds its temporary project in

`PRELUM_TEMP_ROOT` must be an existing, writable absolute directory, and it must be short: the published files-key length bound is derived from how much of the operating system's path limit the render root leaves free, so a long root is rejected at startup rather than accepted and then found wanting on the first request. A container run with `--read-only` needs `--tmpfs /tmp` or a writable volume mounted here; a memory-backed `/tmp` charges render scratch space against the container's memory limit, which is a reason to point this at a disk-backed path instead.

Font and package paths must be existing, readable absolute directories and should be mounted read-only. `PRELUM_FONT_PATH` names one directory, not a path list, so it must not contain the platform path-list separator (`:` in the published Linux image).

Keep network access disabled unless templates intentionally import remote packages. See Fonts and local packages for mount layouts and the security model for the network boundary.

Logs, diagnostics and Sentry

Variable	Default	What it controls
`PRELUM_LOG_PRETTY`	`false`	Human-readable rather than structured JSON logs
`PRELUM_DEBUG_OUTPUT_DIR`	unset	Directory for bounded render debug copies
`PRELUM_SENTRY_DSN`	unset	Sentry project DSN
`PRELUM_SENTRY_ENVIRONMENT`	unset	Environment reported to Sentry
`PRELUM_SENTRY_TRACES_SAMPLE_RATE`	unset	Trace sample rate from 0 to 1

The Sentry SDK is optional when running from source. Install it with `uv sync --extra sentry`; the published image already includes it. Setting `PRELUM_SENTRY_DSN` without the extra logs `sentry.sdk_missing` at error level but does not stop the service.

OPERATIONS

Run Prelum with Docker

The container includes Prelum, the pinned Typst compiler and the default report fonts. You need to provide an API token and publish port 9870:

```
docker build -t prelum:latest .
docker run --rm -p 9870:9870 \
  --memory 2g \
  -e PRELUM_API_TOKEN=replace-with-a-secret \
  prelum:latest
```

`--memory` is the whole container's budget, and it has to cover every render in flight at once. The image bounds each Typst process to 512 MiB (`PRELUM_MAX_RENDER_MEMORY_BYTES`) and allows two at a time (`PRELUM_MAX_CONCURRENT_RENDERS`), which with the request and output buffers this service holds comes to roughly 1.2 GiB of ceiling — so `2g` leaves room for the interpreter itself. Give the container less and the host's OOM killer fires before any render reaches its own limit, and it need not pick the render that caused the pressure. Raise both together, or neither; see Configuration for the arithmetic.

Check that the service is running:

```
curl --silent --show-error http://localhost:9870/health
```

The response is `{"status":"ok"}`. You can now follow the render request guide.

Add shared fonts and packages

Mount shared resources read-only and point Prelum at their container paths:

```
docker run --rm -p 9870:9870 \
  -e PRELUM_API_TOKEN=replace-with-a-secret \
  -e PRELUM_FONT_PATH=/opt/prelum/fonts \
  -e PRELUM_LOCAL_PACKAGE_PATH=/opt/prelum/packages \
  -v "$PWD/fonts:/opt/prelum/fonts:ro" \
  -v "$PWD/packages:/opt/prelum/packages:ro" \
  prelum:latest
```

The image runs as uid 10001. Font and package mounts need only be readable by that user. A mounted `PRELUM_DEBUG_OUTPUT_DIR`, if enabled, must also be writable by uid 10001. See Fonts and local packages for the expected layouts.

Use a published release

Published images use `ghcr.io/vrtfinland/prelum:VERSION`. Pin a production deployment to the immutable digest reported with its release rather than relying on a mutable tag:

```
docker pull ghcr.io/vrtfinland/prelum@sha256:YOUR_DIGEST
```

The Dockerfile copies Typst from an official version- and digest-pinned image. Each Prelum platform image contains an SPDX SBOM and detailed SLSA v1 provenance generated by BuildKit. The final multi-platform manifest is signed keylessly with Cosign using the publishing GitHub Actions workflow's OIDC identity.

Verify a release by immutable digest:

```
cosign verify \
  --certificate-identity-regexp \
   '^https://github.com/VRTFinland/prelum/.github/workflows/build.yml@refs/heads/main$'
\
  --certificate-oidc-issuer https://token.actions.githubusercontent.com \
  ghcr.io/vrtfinland/prelum@sha256:YOUR_DIGEST
```

Release process

Maintainers start releases manually from the `release` workflow on `main`. The requested stable semantic version must match `pyproject.toml`. The workflow creates the `vX.Y.Z` tag, runs the complete test and image publication workflow, and only then creates a GitHub release containing the immutable image digest.

A retry continues a partial release only while its tag still points to the same commit. An existing GitHub release is a successful no-op. Image publication and releases remain disabled while the repository is private; CI still runs the complete test job.

OPERATIONS

Run Prelum on Kubernetes

The image is the same one Docker and releases describes. What changes is where the memory boundary sits: the pod's `limits.memory` is the outer bound, and Prelum's own per-render limit is the inner one. Both are needed, and they do different jobs.

When a pod exceeds `limits.memory`, the kernel's OOM killer chooses a victim from inside that pod by `oom_score`. Prelum and every Typst process it forked are candidates. If it picks Prelum itself, the container restarts: every render in flight dies, the pod drops out of its Service, and the caller whose template caused the pressure receives the same failure as everyone else. `PRELUM_MAX_RENDER_MEMORY_BYTES` stops a runaway render before the pod's budget is touched, so the failure stays a `422` for the one caller responsible while the other renders finish.

The two are distinguishable in the responses, which is the point of setting both. A render that exceeds its own limit aborts and answers `422 template_compile_failed`: the caller wrote a template that wanted too much, and nothing is wrong with the deployment. A render killed by the pod's limit is killed from outside, answers `500 render_failed`, and is logged at `error` so it reaches Sentry — because that one usually means the pod is under-provisioned for its configured concurrency, and the caller who received it may have done nothing unusual at all.

Usually, not always. The per-render bound is `RLIMIT_DATA`, which covers the heap and anonymous mappings: fonts a caller sends in `files` are memory-mapped by Typst outside it, as is scratch space if `/tmp` is a `medium: Memory` volume, and on kernels older than 4.7 anonymous mappings fall outside it too. Enough concurrent requests of that shape can push the pod over its limit without any single render exceeding its own. Nothing in the response can separate that from genuine under-provisioning, so it is reported as the infrastructure fault it resembles — treat a cluster of `render_failed` alerts as a sizing question first, and check the payloads second.

A starting manifest

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prelum
spec:
  replicas: 3
  selector:
    matchLabels:
      app: prelum
  template:
    metadata:
      labels:
        app: prelum
    spec:
      securityContext:
        runAsNonRoot: true
        runAsUser: 10001
        runAsGroup: 10001
      containers:
        - name: prelum
          image: ghcr.io/vrtfinland/prelum:1.1.0
          ports:
            - containerPort: 9870
          env:
            - name: PRELUM_API_TOKEN
              valueFrom:
                secretKeyRef:
                  name: prelum
                  key: api-token
            - name: PRELUM_ENVIRONMENT
```

```
            value: production
          resources:
            requests:
              cpu: 500m
              memory: 1Gi
            limits:
              cpu: "2"
              memory: 2Gi
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            capabilities:
              drop: ["ALL"]
          volumeMounts:
            - name: render-scratch
              mountPath: /tmp
          livenessProbe:
            httpGet:
              path: /health
              port: 9870
          readinessProbe:
            httpGet:
              path: /health
              port: 9870
      volumes:
        - name: render-scratch
          emptyDir:
            sizeLimit: 1Gi
```

Sizing the pod

The image ships `PRELUM_MAX_RENDER_MEMORY_BYTES=536870912` and `PRELUM_MAX_CONCURRENT_RENDERS=2`. With the request and output buffers Prelum holds on its own side, that comes to roughly 1.2 GiB of ceiling, so `limits.memory: 2Gi` leaves room for the interpreter. Configuration sets out the arithmetic in full.

Scale throughput with `replicas`, not with `PRELUM_MAX_CONCURRENT_RENDERS`. Typst is CPU-bound, so a pod with two cores gains little from more concurrent renders, while every extra one multiplies the pod's memory ceiling. More replicas also spread the blast radius of a restart.

If you do raise concurrency, raise `limits.memory` in the same commit. Raising one without the other is the most direct way to reintroduce the pod-wide OOM kill that the per-render limit exists to prevent.

Scratch space and the read-only root filesystem

`readOnlyRootFilesystem: true` leaves Prelum nowhere to build a render's temporary project, so the manifest above mounts an `emptyDir` at `/tmp`, which is where `PRELUM_TEMP_ROOT` points by default.

Leave that volume disk-backed. An `emptyDir` with `medium: Memory` is a tmpfs, and tmpfs pages are charged to the pod's memory limit — where `RLIMIT_DATA` does not reach them, because it bounds the Typst process's heap rather than the filesystem. A render writing large inline files to a memory-backed `/tmp` can therefore OOM the pod with its own memory limit working exactly as intended. `sizeLimit` bounds the volume independently and is worth setting either way.

If you point `PRELUM_TEMP_ROOT` somewhere other than `/tmp`, keep the path short: the published `files` key length bound is derived from how much of the operating system's path limit the render root leaves free, and Prelum refuses to start if a configured root would void it.

Startup failures are deliberate

Prelum probes the memory-limit wrapper before it serves its first request and exits if the wrapper cannot honour the configured value. A `CrashLoopBackOff` with `the render memory limit probe exited ...` in the log means the deployment is misconfigured, not that the service is unhealthy.

This is on purpose. Every way the wrapper can fail ends in the same exit status a failed compilation produces, so without the probe a broken configuration would answer `422` to every caller and page nobody. Fix the configuration rather than removing the limit.

OPERATIONS

Fonts and local packages

Prelum can receive a font with one render request or load shared resources mounted by the operator. Choose based on who owns the resource and how often it is used:

Resource	Best choice
A font used by one request or tenant	Send it in the request's `files` object
A font shared by many templates	Mount `PRELUM_FONT_PATH` read-only
A private, versioned Typst package	Mount `PRELUM_LOCAL_PACKAGE_PATH` read-only

Send a font with one request

Files are written into the temporary project before compilation, and the project root is passed to Typst as `--font-path`. A request-provided `.ttf` or `.otf` file is therefore available by the family name embedded in the font. Encode the font as Base64 in `files`, just like any other binary asset; see Add project files.

The caller is responsible for having the right to use and transmit every supplied font. Prelum does not inspect font licences or grant redistribution rights.

Mount shared fonts

Set `PRELUM_FONT_PATH` to one additional recursively searched font directory. It must be an existing, readable absolute path, must not contain the platform path-list separator and should be mounted read-only. Request-provided fonts remain available alongside it.

```
docker run --rm \
  -e PRELUM_API_TOKEN=replace-me \
  -e PRELUM_FONT_PATH=/opt/prelum/fonts \
  -v "$PWD/fonts:/opt/prelum/fonts:ro" \
  ghcr.io/vrtfinland/prelum:VERSION
```

Mount local Typst packages

`PRELUM_LOCAL_PACKAGE_PATH` exposes a read-only directory of versioned local packages. A package stored under `local/acme-invoice/2.1.0/` is imported by the request's `source` as:

```
#import "@local/acme-invoice:2.1.0": render
#render(data)
```

Packages keep their own Typst root. They cannot read request files unless the inline source passes a project path or loaded value to them. Configuring local packages neither enables downloads nor adds a template-by-name render path.

Remote `@preview` packages are disabled by default. Enable them only when downloads are intentional; see Configuration and Security model.

OPERATIONS

Security model

Prelum runs Typst source supplied by its callers. This is suitable when authenticated users are allowed to compile documents inside a service you operate. Authentication controls who may submit a render; it does not make their source trusted.

Before exposing Prelum outside a development machine:

- use a private, rotated API token and restrict network access to intended clients;
- keep Typst package downloads disabled unless they are explicitly required;
- mount fonts and local packages read-only;
- choose request, concurrency, timeout and output limits for the available capacity;
- keep credentials, request bodies, templates and rendered documents out of logs and telemetry;
- add deployment-level process, filesystem and egress controls appropriate to your environment.

The sections below explain what Prelum itself enforces and where an operator still owns the boundary.

Controls

- The service requires a shared API token outside explicitly named development environments and compares it in constant time.
- The container and Typst subprocess run as uid 10001 rather than root.
- Every render has its own temporary project, which is passed to Typst as `--root`.
- Request file keys cannot escape or ambiguously address that project.
- Render concurrency, queue time, compiler time, request sizes and output sizes are bounded.
- Each Typst process runs under a heap limit where the deployment configures one, so a template that allocates without bound fails against that limit rather than the host's OOM killer. The bound is per process, not per service, and covers the heap and anonymous mappings only — memory-mapped font files and temporary scratch space lie outside it.
- Proxy variables point to a closed local port by default, and package cache/path arguments point to an empty per-render directory unless a local package mount is configured.
- Typst diagnostics are discarded because they may quote caller source.
- Mounted font and package directories are treated as read-only resources.
- The Typst subprocess receives an allowlisted environment rather than the service's own, so neither the API token and Sentry DSN nor `TYPST_*` argument overrides are visible to it.

Network access

No released Typst version has an offline flag. Prelum blocks normal compiler package downloads by combining a closed proxy environment with empty package directories. Set `PRELUM_ALLOW_TYPST_NETWORK=true` only when remote package access is intentional, and use deployment-level egress policy as defence in depth.

Plugins

Prelum does not restrict WebAssembly plugins, and cannot: Typst identifies a plugin from its magic bytes rather than its filename, so a module named `mod.dat` loads exactly as `mod.wasm` does. An earlier check on the `.wasm` suffix was removed because it stopped only the honest spelling while every caller had to mirror it.

Nothing is lost by that. A plugin runs in Typst's own sandbox with no host access, and a caller that can submit source can already run arbitrary Typst code; the bounds that matter are the render timeout and the per-render memory limit, which apply to plugins as to everything else. Deploy Prelum only where compiling authenticated caller source is acceptable.

Error privacy

Validation responses omit Pydantic's mirrored `input` and `ctx`. Caller-controlled values are handled according to the field:

- `files` keys appear whole in `context.key` only after validation has bounded their characters and length;
- object keys in `data` are shortened before appearing in `context.path`;
- validation locations in `context.errors[].loc` and messages in `context.errors[].msg` are also shortened, because Pydantic may include values that have not passed key validation;
- `data` values are never returned;

- configured limits such as `limit` and `timeout_secs` may be returned because they are already public in the error detail and are not secrets.

Compiler stdout and stderr are neither returned nor logged. Logs and Sentry receive `context` as one structured field and must not receive request bodies, credentials, proprietary templates or generated artefacts. Sentry never attaches request bodies, and Prelum redacts the API token header before an event leaves the process because the SDK's built-in list does not know Prelum's custom header name.