



pgCellar

Manual

The barman serves, the cellar keeps

Version 0.1.0 (not released yet)

dbfactory.io
pgcellar.dbfactory.io/docs

Contents

1 Getting started

Get the files and build the image
Create the first admin
Start
Prepare PostgreSQL

Add the first server
First barman check
First backup
Where to go next

2 Concepts

Barman does the backups, pgCellar drives it
A physical backup is not a dump
Two ways for the WAL to reach Barman
Slots, and why they are both a gift and a risk

Point-in-time recovery
What pgCellar adds, and what it never touches
Files are the truth
Vocabulary

3 Preparing a PostgreSQL server

Two roles
pg_hba.conf
Passwords: ~/.pgpass

The settings Server setup checks
Quick manual test

4 Servers and WAL methods

The form
The three WAL methods

The SSH companion
Editing and removing a server

5 Backups, schedules and retention

Backup now
Schedules
Retention

Deleting a backup
Reading the result

6 Restores and point-in-time recovery

Before you restore: read the Timeline
Starting a restore
Where the restore goes
The .tar.gz

Using the restored directory
Retention of restores
Limits

7 Restore drills and reports

What a drill does, step by step
Switching drills on
What the image must contain

Disk and memory
The PDF report
Reading a failed drill

8 Alerts

What triggers a message
Configure the channels
Anti-noise rules
Daily summary

Example messages
Test and journal
Thresholds

9 Users, roles, API keys and the API

Roles
Users
Sessions and sign-in

API keys
Calling the API

10 Configuration reference

The rule: an absent key is the default
Top level
barman
scheduler
servers
alerts

license
users and api_keys
recoveries
drills
Environment variables
Validation errors

11 Operations

Updating the image
Backing up pgCellar itself
Metrics
Logs

Audit
History
Storage and projection
A short weekly routine

12 Security model

Principles
Roles
Passwords and sessions
Sign-in limiter
API keys
Secrets: where they live

SSH, restricted
Container
Network
Demo mode
Your side of the bargain

13 Troubleshooting

Setup and sign-in
The server page
Backups

Restores and drills
WAL shipping (archive_command)
Still stuck

14 Licensing and editions

The source licence: FSL-1.1-Apache-2.0
Editions
What a licence blocks, and what it never blocks

Putting a key in place
Which servers are counted

15 CLI reference

serve
check
apikey
hash-password

license
version
help
Build from source

Getting started

By the end of this chapter you have pgCellar running in Docker, one PostgreSQL server added, a green `barman check`, and a first backup. Count five minutes if PostgreSQL is already prepared. If you still have to create the roles, chapter 3 has the commands.

You need:

- Docker (or Podman) with Compose, on a machine that can reach your PostgreSQL server over the network.
- A PostgreSQL server, version 13 to 18, where you can create two roles and edit `pg_hba.conf`.
- Disk space for the backups. Rule of thumb: one full copy of the database, plus the WAL of the retention period.

1.1 Get the files and build the image

```
git clone https://github.com/stranix79/pgcellar.git
cd pgcellar/deploy
mkdir etc-pgcellar etc-barman.d
cp pgcellar.example.yaml etc-pgcellar/pgcellar.yaml
docker compose build
```

The image is called `pgcellar:dev`. It contains pgCellar, an unmodified Barman from the PGDG packages, the PostgreSQL client tools, and the PostgreSQL server binaries that restore drills use (chapter 7).

Two folders are bind-mounted from the host: `etc-pgcellar` (your `pgcellar.yaml`) and `etc-barman.d` (one Barman file per server, written by the console). The container runs as the `barman` user, not as root, so these folders must be writable by that user. Find its uid and set the owner:

```
docker run --rm --entrypoint id pgcellar:dev barman
sudo chown -R 997:997 etc-pgcellar etc-barman.d
```

Use the uid the first command prints (997 in the current image). If you skip this step, pgCellar cannot save its configuration.

1.2 Create the first admin

pgCellar stores a bcrypt hash, never a password. Type the password once and keep the hash:

```
read -rs PW; printf '%s\n' "$PW" | docker run -i --rm pgcellar:dev hash-password
```

The password needs at least 8 characters. The command prints one line that starts with `$2a$` or `$2b$`.

Edit `etc-pgcellar/pgcellar.yaml`:

- In `users:`, replace the fake `password_hash` of `admin` with your hash.
- Set `base_url` to the address people will type, for example `http://backup.example.com:8080`. It is used in links inside alerts and reports.
- Delete the two example entries in `servers:`. You add yours from the console.
- Delete the fake entry in `api_keys:` unless you want it.

Check the file:

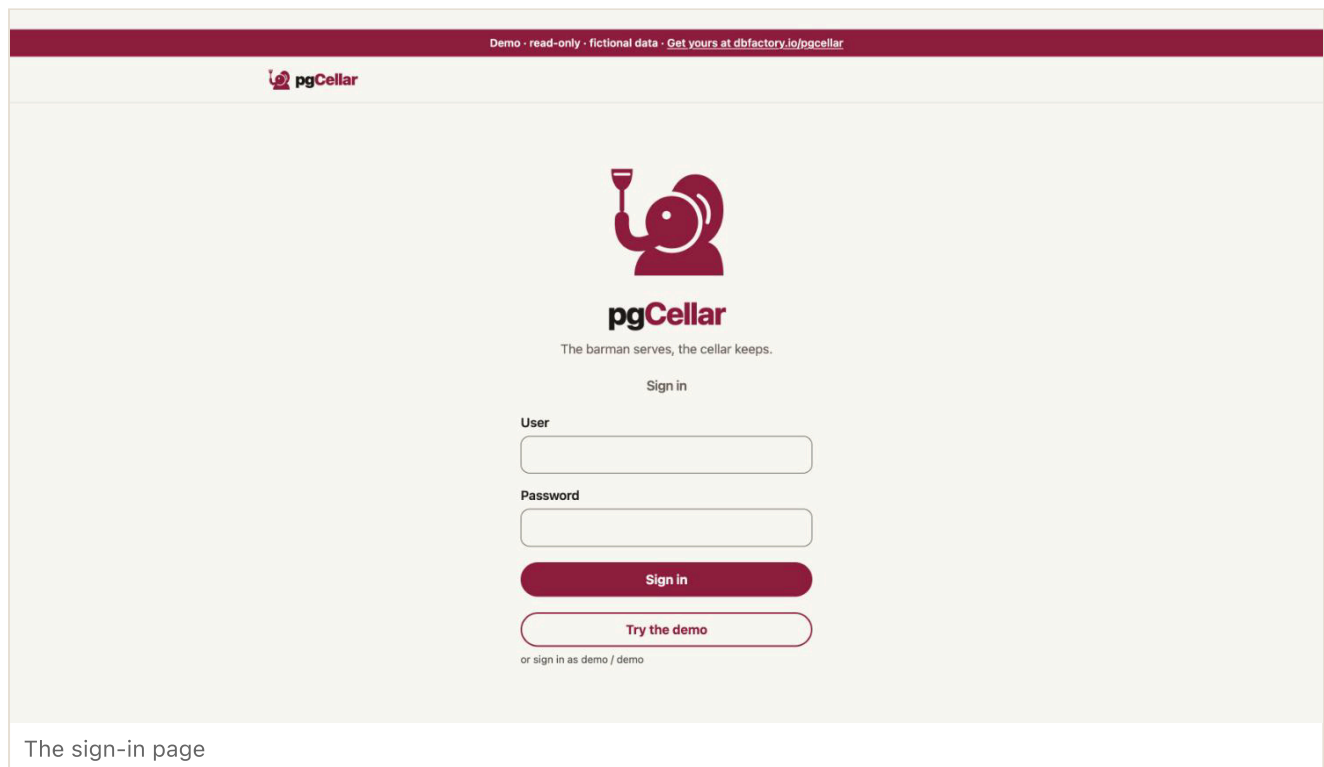
```
docker run --rm -v "$PWD/etc-pgcellar:/etc/pgcellar" pgcellar:dev check --config /etc/pgcellar/pgcellar.yaml
```

You want to read `all good` at the end.

1.3 Start

```
docker compose up -d
```

Open `http://localhost:8080` and sign in as `admin`.



Demo - read-only - fictional data - Get yours at dbfactory.io/pgcellar

 **pgCellar**

The barman serves, the cellar keeps.

Sign in

User

Password

Sign in

Try the demo

or sign in as demo / demo

The sign-in page

The Status page is empty: no server yet. Compose also started a second container, `pgcellar-ssh`. It only matters when a server ships its WAL files with `archive_command` (chapter 4). For streaming you can ignore it.

1.4 Prepare PostgreSQL

pgCellar needs two roles on the PostgreSQL side and two lines in `pg_hba.conf`. Chapter 3 explains each one. Short version, as a superuser on the PostgreSQL server (PostgreSQL 15 or later):

```
CREATE ROLE barman LOGIN PASSWORD '...';
CREATE ROLE streaming_barman LOGIN REPLICATION PASSWORD '...';
GRANT pg_read_all_settings, pg_read_all_stats, pg_checkpoint TO barman;
GRANT EXECUTE ON FUNCTION pg_backup_start(text, boolean) TO barman;
GRANT EXECUTE ON FUNCTION pg_backup_stop(boolean) TO barman;
GRANT EXECUTE ON FUNCTION pg_switch_wal() TO barman;
GRANT EXECUTE ON FUNCTION pg_create_restore_point(text) TO barman;
```

In `pg_hba.conf`, with the address pgCellar connects from:

```
host all          barman          10.0.0.5/32 scram-sha-256
host replication streaming_barman 10.0.0.5/32 scram-sha-256
```

Reload PostgreSQL (`SELECT pg_reload_conf();`). Then give the two passwords to the barman user inside the container, in `~/.pgpass`. Barman and pgCellar both read this file, so no password ever sits in a config file:

```
docker compose exec -T pgcellar sh -c 'umask 077; cat >> ~/.pgpass' <<'EOF'
pg-main.example.com:5432*:barman:THE_BARMAN_PASSWORD
pg-main.example.com:5432*:streaming_barman:THE_STREAMING_PASSWORD
EOF
```

The file lives in `/var/lib/barman`, which is a volume, so it survives restarts.

1.5 Add the first server

In the console, open **Servers**, then **Add a server**. For a PostgreSQL server reachable over the network:

Field	Value
Name	pg-main (lowercase letters, digits, -, _)
Backup method	postgres (streaming, no SSH)
WAL method	streaming
Connection string (conninfo)	host=pg-main.example.com user=barman dbname=postgres port=5432
Streaming connection string	host=pg-main.example.com user=streaming_barman dbname=postgres port=5432
Replication slot	barman
Retention	Recovery window 14 days, minimum redundancy 2
Backup schedule	daily 02:00

Press **Create server**. pgCellar writes `/etc/barman.d/pg-main.conf`, adds the server to `pgcellar.yaml`, and commits both to git. The WAL method and the slot are explained in chapters 2 and 4.

The server page

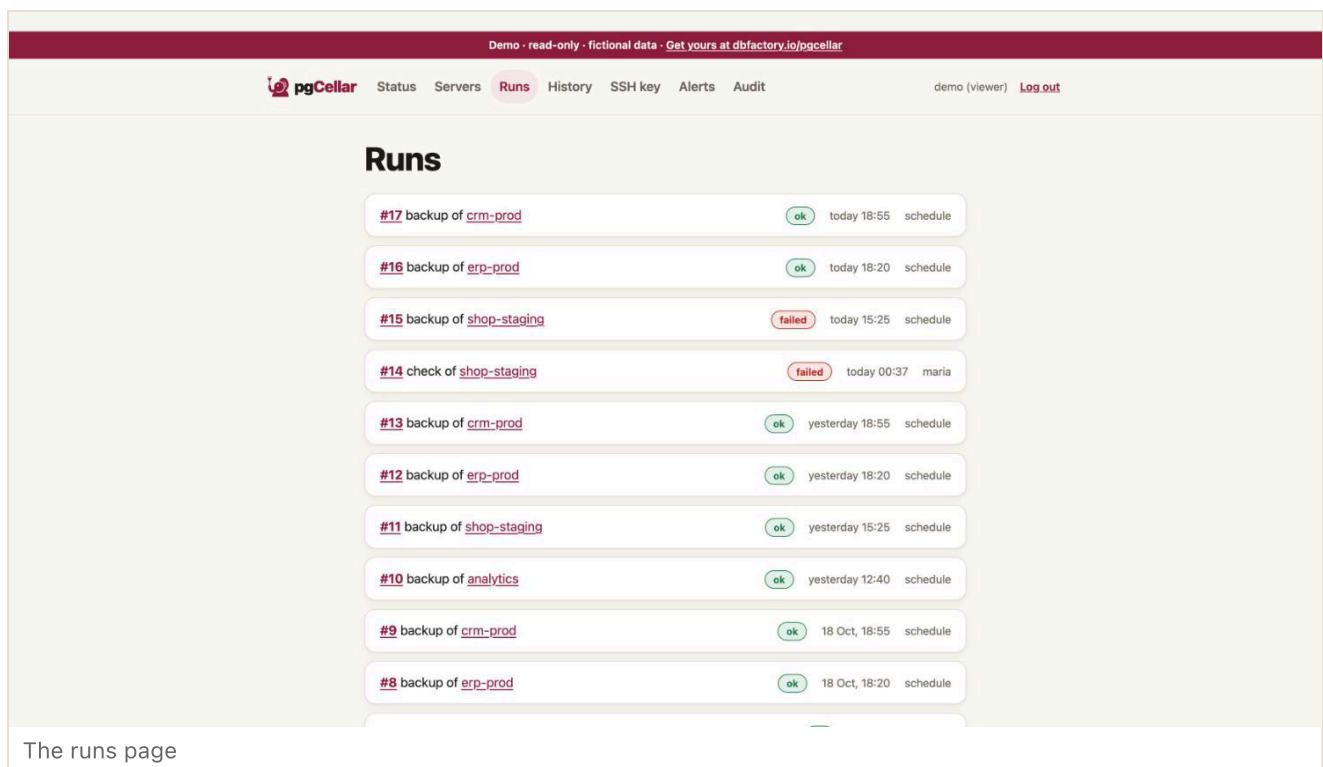
1.6 First barman check

On the server page, press **Run check now**. Barman runs its checks and pgCellar explains every line in plain words, with what to do when one fails. Typical first failures: a password missing in `.pgpass`, a missing `pg_hba.conf` line, `wal_level = minimal`. The **Server setup** section reads the PostgreSQL settings and gives the exact `ALTER SYSTEM` commands (chapter 3).

Barman starts its WAL receiver by itself (pgCellar runs `barman cron` every minute). Once every line is green, the server is ready. A new server may say that no WAL has arrived yet; the first backup closes a WAL file and clears that.

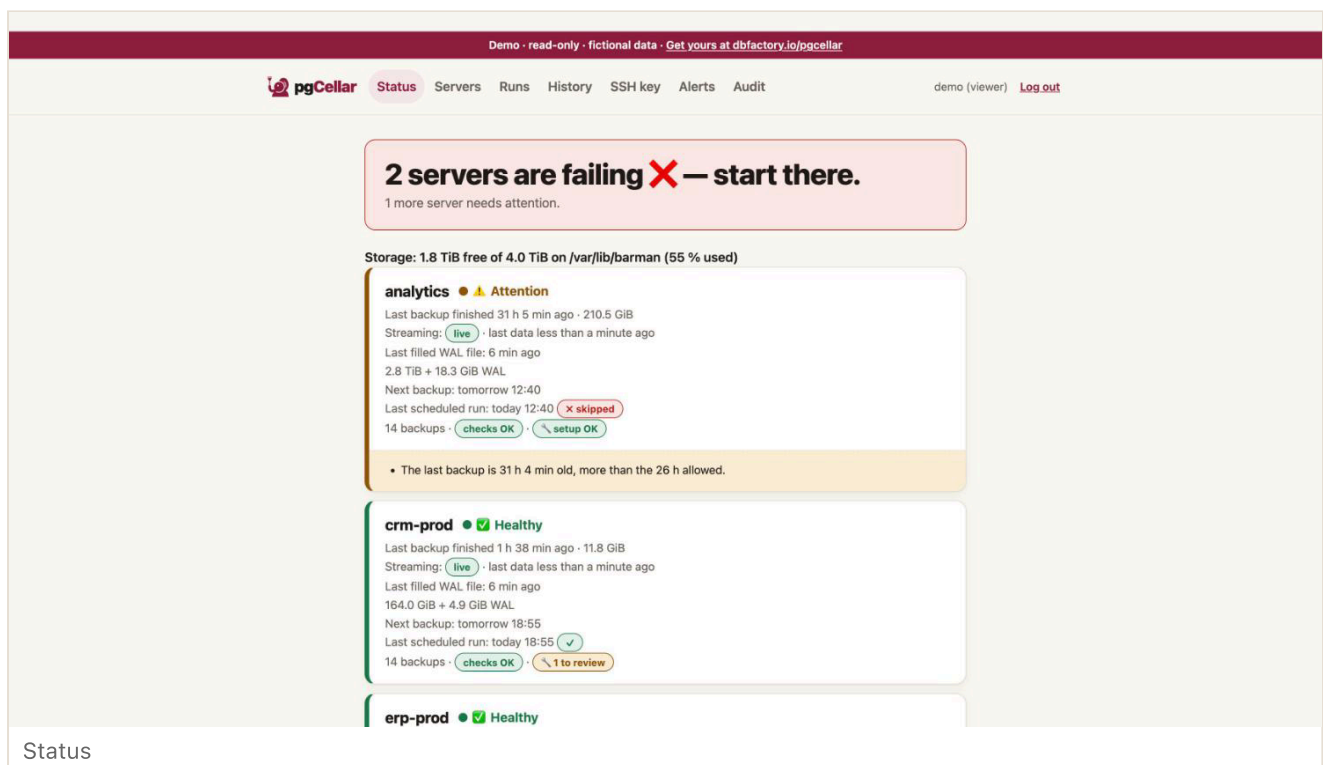
1.7 First backup

Press **Backup now**. pgCellar runs `barman backup pg-main`, then forces a WAL switch so the backup can be restored right away. Follow it on the **Runs** page: it shows every command with its output and duration.



The runs page

When it is done, the server is green on Status and the Timeline shows the first point in time you can restore to.



Status

1.8 Where to go next

- Put the schedule in place: [Backups, schedules and retention](#).
- Plan the first drill, so the backup is proven: [Restore drills and reports](#).

- Set up alerts so you hear about problems: **Alerts**.
- Try pgCellar without any PostgreSQL: `make demo` in the repository starts it on fictional data (login `demo` / `demo`). A public demo runs at <https://demo.pgcellar.dbfactory.io>.

Concepts

Six ideas explain nearly everything in pgCellar. Read this chapter once; the rest of the manual gets easier.

2.1 Barman does the backups, pgCellar drives it

Barman (by EnterpriseDB) is the tool that actually takes physical backups of PostgreSQL, keeps the WAL, applies retention and restores. It is excellent and it is a command line. pgCellar adds what the command line lacks: one screen that tells the truth, schedules, a restore form, alerts, and restore tests.

pgCellar never imports, patches or links Barman. It runs the `barman` command, reads its JSON output and its files, and writes its configuration files. The Docker image installs Barman unchanged from the PostgreSQL project's own packages. Barman is GPL-3.0; pgCellar only drives it through its command line, and that boundary is a rule of the project.

Consequence: everything pgCellar does you can do by hand with the `barman` command, and anything you do by hand shows up in pgCellar at the next refresh.

2.2 A physical backup is not a dump

A dump (`pg_dump`) writes the content of the database as SQL. It brings you back to the moment the dump started, and nowhere else.

A physical backup copies the files of the PostgreSQL cluster. Combined with the WAL, the write-ahead log that records every change, it lets you stop at any moment you choose. That is what lets you answer "bring the database back as it was at 14:31, one minute before the bad query".

Two things make that work:

- **Base backups.** A full copy of the data directory, taken while the server runs. It is consistent only together with the WAL written during the copy.
- **WAL files.** PostgreSQL writes its changes to 16 MB segments. Barman keeps every segment, from the first backup on, in an archive.

A restore is "start from a base backup, then replay WAL up to the point you want". Without the WAL between two backups you can restore to the backups themselves, nothing in between. So a backup system is only as good as its WAL chain, and pgCellar's Timeline page draws that chain and shows holes.

2.3 Two ways for the WAL to reach Barman

Streaming. Barman connects to PostgreSQL like a replica and receives WAL as it is written, through `pg_receivewal`. No SSH, no setup on the PostgreSQL host beyond a role and a `pg_hba.conf` line. The data is nearly current: a crash loses almost nothing. Barman keeps a **replication slot** on the server so PostgreSQL does not throw away WAL that Barman has not received yet.

archive_command. PostgreSQL pushes each completed WAL file to Barman by running a command, here `barman-wal-archive` over SSH. It needs the `barman-cli` package and an SSH key on the PostgreSQL host, and the recovery point is the last *completed* 16 MB file (or `archive_timeout`). It works where a replication connection is impossible, and it is what the rsync backup method requires.

Both is allowed and is the most resilient: WAL arrives twice, and either path alone is enough to keep the chain whole.

Chapter 4 shows how to choose, and the exact settings.

2.4 Slots, and why they are both a gift and a risk

A replication slot is a promise from PostgreSQL: "I will not delete WAL that this consumer has not read". It makes streaming reliable. The cost: if the backup host is down for a long time, PostgreSQL keeps all that WAL and its disk can fill up. The setting `max_slot_wal_keep_size` caps that risk. pgCellar's Server setup warns when it is not set (chapter 3).

2.5 Point-in-time recovery

PITR is "base backup plus WAL replay up to a target". The target can be:

- the end of the backup (the moment the copy became consistent),
- a date and time,
- the latest moment Barman holds (replay everything).

The earliest target is the end of the oldest backup whose WAL chain is unbroken. The latest is the last data received (streaming) or the last completed WAL file. The Timeline page works both out for you and shows them in a sentence.

2.6 What pgCellar adds, and what it never touches

pgCellar adds:

- **Status**, one card per server, green, orange or red with the reason in a sentence.
- **Server forms** that write Barman's configuration, and **Server setup**, which reads the PostgreSQL settings Barman depends on and tells you what to change.
- **A scheduler** (no host cron), **retention** through Barman, **backup deletion**.
- **Restore to a directory** with a target, and a `tar.gz` if you want.
- **Restore drills**: every week, a real restore, started and checked, with a PDF report.

- **Alerts**, a **JSON API**, **Prometheus metrics**, **audit**, and a git history of the configuration.

pgCellar never:

- changes anything on your PostgreSQL server. Server setup only says what to change; you run the command. The only queries it sends are a read-only `SELECT` on `pg_settings` and what Barman itself runs;
- stores a secret in `pgcellar.yaml` or in git (passwords live in `~/.pgpass`, alert secrets in environment variables, SSH private keys in the barman user's `~/.ssh`);
- keeps the truth in its own database. SQLite holds run history, audit and sessions. You can delete it and lose history, not backups.

2.7 Files are the truth

Barman's configuration stays in `/etc/barman.conf` and `/etc/barman.d/<server>.conf`. Everything pgCellar owns is one YAML file, `pgcellar.yaml`. The console and the API are forms over those files. Every change is validated, written, and committed to a local git repository of the configuration folder, with the name of the user as author.

You can edit the files with a text editor; pgCellar re-reads `pgcellar.yaml` when its modification time changes. Read `git log` in the configuration folder to see who changed what, and the **History** page shows the same thing.

2.8 Vocabulary

Word	Meaning
Server	One PostgreSQL instance backed up by Barman. A <code>[section]</code> in <code>barman.d</code> .
Base backup	A full physical copy of the data directory.
WAL	Write-ahead log, the stream of changes. Archived in 16 MB files.
Slot	A replication slot on the PostgreSQL side, holding WAL for Barman.
Recovery window	Retention rule: keep enough backups to restore to any point in the last N days.
Run	One command started from pgCellar (backup, restore, drill), with its output.
Drill	An automatic restore of a backup into a throwaway PostgreSQL, checked, then removed.

Preparing a PostgreSQL server

pgCellar never changes your PostgreSQL server. You prepare it once, with the commands in this chapter, and the **Server setup** section of the server page checks that you did it right and keeps checking. Every command below is the one pgCellar prints in its own fix column, so you can also copy it from the console.

can go. Read from Barman's own files, nothing is run.

Server setup

What pgCellar found on the PostgreSQL server, and what to change there. Nothing here is changed by pgCellar.

max_slot_wal_keep_size = -1 (no limit) warning
 Expected: a limit, for example 2GB
 If the backup host is down for long, WAL kept by the slot can fill the disk; set 2GB (or more).
 ALTER SYSTEM SET max_slot_wal_keep_size = '2GB';
 SELECT pg_reload_conf();
 reload is enough

archive_mode = off info
 Not used: WAL reaches Barman by streaming, not by archive_command.

wal_compression = lz4 info
 Compresses full-page images in the WAL: smaller WAL, a little more CPU. Optional.

Server setup

3.1 Two roles

Barman uses two connections, so it uses two roles.

Role	Used for	Needs
barman	Checks, metadata, starting and stopping a backup, switching WAL	LOGIN , the grants below
streaming_barman	pg_basebackup and the WAL stream	LOGIN REPLICATION

You can use other names. The names go in the two connection strings of the server (chapter 4).

On PostgreSQL 15 and later, as a superuser:

```
CREATE ROLE barman LOGIN PASSWORD '...';
CREATE ROLE streaming_barman LOGIN REPLICATION PASSWORD '...';

GRANT pg_read_all_settings, pg_read_all_stats, pg_checkpoint TO barman;
GRANT EXECUTE ON FUNCTION pg_backup_start(text, boolean) TO barman;
GRANT EXECUTE ON FUNCTION pg_backup_stop(boolean) TO barman;
GRANT EXECUTE ON FUNCTION pg_switch_wal() TO barman;
GRANT EXECUTE ON FUNCTION pg_create_restore_point(text) TO barman;
```

In `psql`, use `\password barman` instead of typing the password in the statement, so it does not land in your shell history or the server log.

On PostgreSQL 13 and 14 the predefined role `pg_checkpoint` does not exist and the backup functions have other names. Grant `pg_monitor` instead, and:

```
GRANT pg_monitor TO barman;
GRANT EXECUTE ON FUNCTION pg_start_backup(text, boolean, boolean) TO barman;
GRANT EXECUTE ON FUNCTION pg_stop_backup() TO barman;
GRANT EXECUTE ON FUNCTION pg_stop_backup(boolean, boolean) TO barman;
GRANT EXECUTE ON FUNCTION pg_switch_wal() TO barman;
GRANT EXECUTE ON FUNCTION pg_create_restore_point(text) TO barman;
```

A superuser also works for `barman`. It is simpler and it is more power than the job needs; prefer the grants.

The role must be allowed to read `archive_command`. `pg_read_all_settings` (or `pg_monitor`) does that. Without it PostgreSQL answers `<insufficient privilege>` and pgCellar shows an info line asking you to grant the role.

3.2 pg_hba.conf

`all` does not match replication connections, so the replication role needs a line of its own with the word `replication` as database:

```
host all          barman          10.0.0.5/32 scram-sha-256
host replication streaming_barman 10.0.0.5/32 scram-sha-256
```

Replace the address with the one PostgreSQL sees. If a proxy or a relay sits between pgCellar and PostgreSQL, it is the proxy's address, not the address of the pgCellar host. When in doubt, try to connect and read the PostgreSQL log: the refusal names the exact address (no `pg_hba.conf` entry for host "172.27.0.4", user "barman").

Put the lines before any broader rule that would match first (a `trust` line, for example), then reload:

```
SELECT pg_reload_conf();
```

Check how PostgreSQL parsed them:

```
SELECT line_number, type, database, user_name, address, auth_method, error
FROM pg_hba_file_rules
WHERE 'barman' = ANY (user_name) OR 'streaming_barman' = ANY (user_name);
```

The `error` column must be empty.

3.3 Passwords: `~/.pgpass`

pgCellar and Barman read passwords from `~/.pgpass` of the barman user (`/var/lib/barman/.pgpass` in the image), mode 0600, one line per connection:

```
pg-main.example.com:5432:*:barman:THE_BARMAN_PASSWORD
pg-main.example.com:5432:*:streaming_barman:THE_STREAMING_PASSWORD
```

pgCellar drops any `password=` from a connection string before it runs `psql`, and never puts a password on a command line. Put the password in `.pgpass` and not in the connection string.

3.4 The settings Server setup checks

pgCellar reads these settings with one read-only query on `pg_settings`, through `psql`, using the `conninfo` of the server. The first read happens after the first status refresh; then every `setup_interval` (30 minutes by default), and again each time you press **Run check now**.

A check is `ok`, `info`, `warning` or `error`. An **error** means Barman will fail or a backup cannot be trusted, and it turns the server red or orange in Status. A **warning** is advice and never changes the colour of a server.

Setting	Wants	Level when wrong	Needs
<code>wal_level</code>	<code>replica</code> or <code>logical</code>	error	restart
<code>max_wal_senders</code>	2 or more	error	restart
<code>max_replication_slots</code>	1 or more when the server has a <code>slot_name</code>	error	restart
<code>archive_mode</code>	<code>on</code> or <code>always</code> , only when WAL goes by <code>archive_command</code>	error	restart
<code>archive_command</code>	<code>barman-wal-archive ...</code> , only when WAL goes by <code>archive_command</code>	error	reload
<code>archive_timeout</code>	more than 0, for example <code>1h</code>	warning	reload
<code>max_slot_wal_keep_size</code>	not <code>-1</code> when a slot is used, for example <code>2GB</code>	warning	reload
<code>full_page_writes</code>	<code>on</code>	error	reload
<code>data_checksums</code>	<code>on</code>	warning	not a setting, see below
client tools version	<code>pg_basebackup</code> at least as recent as the server	error	image change
<code>wal_compression</code> , <code>summarize_wal</code>	information only	info	

Settings that do not exist in the version of your server are skipped.

Three more lines come from `barman check`, without any new query: **Streaming access**, **Role privileges** and **Replication slot**. When streaming access fails, the fix shows the `pg_hba.conf` line to add, with your streaming user in it. For servers using `archive_command`, Barman's own verdicts on `wal_archive`, `archive_command` and `archive_mode` are shown too.

If pgCellar cannot read the settings at all (no network, no password in `.pgpass`), the section shows one error line: "Could not read the server settings", with the reason. That line does not change the health of the server, because Barman's own checks already tell you whether the server is reachable.

The fixes, one by one

Each fix is plain SQL. A parameter that needs a restart says so; apply it in a maintenance window.

wal_level (restart):

```
ALTER SYSTEM SET wal_level = 'replica';
```

max_wal_senders (restart). Barman streams WAL over one connection and takes the base backup over another:

```
ALTER SYSTEM SET max_wal_senders = 10;
```

max_replication_slots (restart):

```
ALTER SYSTEM SET max_replication_slots = 10;
```

archive_mode (restart), when WAL goes by **archive_command** :

```
ALTER SYSTEM SET archive_mode = 'on';
```

archive_command (reload). The command comes from the **ssh_host** and **ssh_port** of **pgcellar.yaml** and the name of the server:

```
ALTER SYSTEM SET archive_command = 'barman-wal-archive --port 2222 backup.example.com pg-main %p';  
SELECT pg_reload_conf();
```

barman-wal-archive comes from the **barman-cli** package on the PostgreSQL host. The postgres user must have an SSH key that pgCellar has authorised (chapter 4). pgCellar cannot test this command from its side. The proof is to press **Switch WAL** on the server page and look at the WAL archive check.

archive_timeout (reload). On a quiet database a 16 MB file fills rarely, so the recovery point can be hours old. Closing a file at least once an hour bounds it:

```
ALTER SYSTEM SET archive_timeout = '1h';  
SELECT pg_reload_conf();
```

max_slot_wal_keep_size (reload). If the backup host is down for long, the slot holds WAL and the PostgreSQL disk can fill. Cap it:

```
ALTER SYSTEM SET max_slot_wal_keep_size = '2GB';  
SELECT pg_reload_conf();
```

Choose a value larger than the WAL your server writes during the longest outage you want to survive. If the cap is hit, PostgreSQL invalidates the slot and Barman loses the chain: you take a new backup. That is a better failure than a full production disk.

full_page_writes (reload). It should never be off for a server you back up:

```
ALTER SYSTEM SET full_page_writes = on;  
SELECT pg_reload_conf();
```

data_checksums. This is not a parameter. Enable it when the cluster is created (**initdb --data-checksums**), or on a stopped cluster:

```
pg_checksums --enable -D /var/lib/postgresql/16/main
```

then start PostgreSQL again. Without checksums a damaged data page goes unnoticed, in the backup as well, and a restore drill cannot tell. The check is a warning, and a very good idea.

Client tools version. `pg_basebackup` must be at least as recent as the server. The image ships the PostgreSQL 18 client, which talks to 13 through 18. If you replace the client, point `path_prefix` in the server file to the directory with the right binaries.

3.5 Quick manual test

From the pgCellar host, before you add the server:

```
docker compose exec pgcellar psql "host=pg-main.example.com user=barman dbname=postgres" -c "select 1"
docker compose exec pgcellar psql "host=pg-main.example.com user=streaming_barman dbname=postgres
replication=true" -c "IDENTIFY_SYSTEM"
```

Both must answer without asking for a password. If they do, add the server in the console.

Servers and WAL methods

A server in pgCellar is two things: a section in a Barman file (`/etc/barman.d/<name>.conf`) and, optionally, an entry in `pgcellar.yaml` for what Barman does not know (label, schedule, drill schedule, tags). The **Add a server** form writes both and commits them to git.

4.1 The form

Field	What it does	Barman key
Name	File name and section name. Lowercase letters, digits, <code>-</code> , <code>_</code> .	<code>[name]</code>
Label	Free text shown in pgCellar.	(pgcellar.yaml)
Description	Barman's description.	<code>description</code>
Backup method	<code>postgres</code> (streaming, no SSH) or <code>rsync</code> (over SSH).	<code>backup_method</code>
WAL method	<code>streaming</code> , <code>archive_command</code> or <code>both</code> .	<code>streaming_archiver</code> , <code>archiver</code>
Connection string (conninfo)	How Barman and pgCellar query the server.	<code>conninfo</code>
Streaming connection string	The replication connection.	<code>streaming_conninfo</code>
Replication slot	Slot Barman creates and uses.	<code>slot_name</code> , <code>create_slot = auto</code>
SSH command	<code>rsync</code> method only, for example <code>ssh postgres@db1</code> .	<code>ssh_command</code>
Retention	Recovery window in days, or redundancy, plus a minimum redundancy.	<code>retention_policy</code> , <code>minimum_redundancy</code>
Backup schedule	Schedule and on/off (chapter 5).	(pgcellar.yaml)

Keys the form does not know are left exactly as they are, in the same order. If you add `backup_compression = gzip`, `path_prefix`, `reuse_backup = link` or anything else by hand, saving the form keeps them.

The form refuses combinations Barman would reject, with the reason in a sentence:

- `postgres` method needs `conninfo` and `streaming_conninfo`, and WAL by streaming or by `archive_command`; a slot is needed when WAL is streamed.
- `rsync` method needs `ssh_command`, `conninfo`, and `archiver = on` (so the WAL method is `archive_command` or `both`). With streaming it also needs `streaming_conninfo`.

You can also write the file by hand and check it. This is `deploy/barman.d.example/pg-main.conf`:

```
[pg-main]
description = "Main application database"

conninfo = host=pg-main.example.com user=barman dbname=postgres port=5432
streaming_conninfo = host=pg-main.example.com user=streaming_barman dbname=postgres port=5432

backup_method = postgres
streaming_archiver = on
archiver = off
slot_name = barman
create_slot = auto

retention_policy = RECOVERY WINDOW OF 14 DAYS
minimum_redundancy = 2

backup_compression = gzip
```

Run `pgcellar check --config /etc/pgcellar/pgcellar.yaml` to validate the configuration and the Barman files together (chapter 15).

4.2 The three WAL methods

The WAL method is a choice over two Barman switches. It is not stored anywhere: it is worked out from the two keys every time the file is read.

WAL method	streaming_archiver	archiver	Default for
streaming	on	off	postgres backups
archive_command	off	on	rsync backups
both	on	on	nothing; the resilient choice

Pick `streaming` when pgCellar can open a replication connection to PostgreSQL. It is the simplest: no SSH, no package on the database host, almost no data lost on a crash.

Pick `archive_command` when a replication connection is impossible (a managed service, a firewall, an old setup), or when you use the `rsync` backup method. PostgreSQL pushes each completed WAL file over SSH to the companion container.

Pick `both` for a server you cannot afford to lose WAL from. If one path breaks, the other keeps the chain whole. It costs a slot, a role and an SSH key.

Notes:

- `backup_method = postgres` with `archiver = on` is valid: `pg_basebackup` takes the base backup and `archive_command` carries the WAL. No slot is needed, but `streaming_conninfo` is still required, because `pg_basebackup` uses it.
- `create_slot = auto` is added only when streaming is on.
- pgCellar runs `barman cron` every `barman.cron_interval` (one minute) to start the WAL receivers. You never run it yourself.

4.3 The SSH companion

WAL shipping needs an SSH server in front of Barman's data. The Compose file runs one as a second container, `pgcellar-ssh`, built from the same image with another entry point. Facts:

- It runs as the `barman` user, never as root. It shares the Barman volume.
- It listens on port 2222 inside and outside (`ports: "2222:2222"`). Bind it to a private address if you can (`"10.0.0.5:2222:2222"`); do not expose it to the internet.
- Login is by key only. No password, no terminal, no forwarding, no sftp.
- Its host key is generated on first start in `~barman/.ssh/host/` in the volume and kept.
- Each authorised key is restricted to a single command, `barman put-wal <server>`. A stolen key can push WAL of one server and run nothing else.

The configuration is `deploy/sshd/sshd_config`, mounted read-only.

Authorising a PostgreSQL server

1. On the PostgreSQL host, install `barman-cli` (it provides `barman-wal-archive`) and make an SSH key for the `postgres` user, without a passphrase:

```
sudo -u postgres ssh-keygen -t ed25519 -N "" -f /var/lib/postgresql/.ssh/id_ed25519
sudo cat /var/lib/postgresql/.ssh/id_ed25519.pub
```

The home of the `postgres` user differs between systems. Use the real path.

2. In pgCellar, open **SSH key** (admin). In the section "PostgreSQL servers allowed to push WAL", choose the server, paste the public key line, and press **Authorise this key**. pgCellar appends this to `~barman/.ssh/authorized_keys`:

```
restrict,command="barman put-wal pg-main" ssh-ed25519 AAAA... pgcellar:pg-main
```

The comment you pasted is dropped. The same key cannot be added twice, for any server. **Remove** revokes it.

3. Let the `postgres` user trust the companion's host key. Connect once by hand and answer `yes`:

```
sudo -u postgres ssh -p 2222 barman@backup.example.com true
```

The command itself does not do anything useful because of the restriction; it is the host key you want recorded in `known_hosts`.

4. Tell PostgreSQL what to run. This is the command Server setup advises (chapter 3):

```
ALTER SYSTEM SET archive_mode = 'on';           -- restart needed
ALTER SYSTEM SET archive_command = 'barman-wal-archive --port 2222 backup.example.com pg-main %p';
SELECT pg_reload_conf();
```

The arguments are, in order: `--port` of the companion, the name by which PostgreSQL reaches the pgCellar host (`ssh_host`), the server name in Barman, and `%p`, the path of the WAL file.

5. On the server page press **Switch WAL**, then **Run check now**. The WAL archive lines of the check must turn green.

Set `ssh_host` and `ssh_port` in `pgcellar.yaml`. If you leave `ssh_host` empty, pgCellar advises its own host name, which inside Docker is a container id nobody can reach. Always set it:

```
ssh_host: backup.example.com
ssh_port: 2222
```

`barman-wal-archive --test` does not work with a restricted key (the forced command ignores the option). That is a price of the restriction. Use **Switch WAL** and the check instead.

The rsync method and the SSH key of pgCellar

The `rsync` backup method works the other way round: Barman connects to the PostgreSQL host over SSH, as the user in `ssh_command`. For that, pgCellar makes one ed25519 key pair for the barman user (`~/.ssh/id_ed25519`) the first time you open the **SSH key** page, or run an SSH test. It never replaces an existing key. The page shows the public line; put it in the `authorized_keys` of the user on the PostgreSQL host.

The private key stays in `~barman/.ssh` with mode 0600, in the volume. It is never shown, never written to the API, the audit log or git.

On a server page, **Test SSH connection** (admin) runs `ssh -o BatchMode=yes -o ConnectTimeout=10 ... true` and shows ssh's own answer. `BatchMode` means a host key unknown to `known_hosts` fails; connect once by hand to accept it.

4.4 Editing and removing a server

Save changes on the server page rewrites the Barman file and the pgCellar entry, validates, and commits. The **History** page lists the commits; each one shows its patch.

Removing a server is a manual action for now: delete its file in `etc-barman.d` and its entry in `pgcellar.yaml`. Its backups stay on disk under `/var/lib/barman/<name>`; deleting them is your decision. If the server used a slot, drop it on the PostgreSQL side:

```
SELECT pg_drop_replication_slot('barman');
```

A forgotten slot keeps WAL forever.

Backups, schedules and retention

5.1 Backup now

On a server page, **Backup now** (operator role) starts a backup. The same thing from the API:

```
curl -X POST -H "Authorization: Bearer $KEY" http://localhost:8080/api/v1/servers/pg-main/backup
```

pgCellar runs two Barman commands, in this order:

```
barman backup pg-main  
barman switch-wal --force --archive pg-main
```

It never uses `barman backup --wait`. On a quiet database the end of the backup sits in a WAL segment that is still being written, and `--wait` blocks until 16 MB of WAL appear or someone forces a switch. A backup that took 11 seconds could then wait 15 minutes. The forced switch closes that segment so the backup is restorable at once (chapter 13).

Everything runs in the background. The run appears on **Runs** with the output of each command, its exit code and its duration. A closed browser tab does not interrupt it. Only one backup per server runs at a time.

Runs lists only what was started from pgCellar (the console, the scheduler or the API). A backup taken with the `barman` command or with a host cron does not appear there; it does appear in the **Backups** list of the server page, which comes from Barman itself.

Demo - read-only - fictional data - Get yours at dbfactory.io/pgcellar

[Status](#)
[Servers](#)
[Runs](#)
[History](#)
[SSH key](#)
[Alerts](#)
[Audit](#)
demo (viewer) [Log out](#)

Runs

#17 backup of crm-prod	ok	today 18:55	schedule
#16 backup of erp-prod	ok	today 18:20	schedule
#15 backup of shop-staging	failed	today 15:25	schedule
#14 check of shop-staging	failed	today 00:37	maria
#13 backup of crm-prod	ok	yesterday 18:55	schedule
#12 backup of erp-prod	ok	yesterday 18:20	schedule
#11 backup of shop-staging	ok	yesterday 15:25	schedule
#10 backup of analytics	ok	yesterday 12:40	schedule
#9 backup of crm-prod	ok	18 Oct, 18:55	schedule
#8 backup of erp-prod	ok	18 Oct, 18:20	schedule

Runs

5.2 Schedules

You do not need cron on the host. pgCellar runs the schedules itself. The schedule of a server is one line:

```
servers:
- name: pg-main
  schedule:
    backup: "daily 02:00"
    enabled: true
```

You can also set it in the server form (field **Backup schedule**, checkbox **Enabled**).

Syntax

Text	Means
daily 02:00	Every day at 02:00
weekly sun 03:00	Every Sunday at 03:00. Days: <code>mon tue wed thu fri sat sun</code>
hourly	At minute 0 of every hour
every 6 hours	Every 6 hours, at minute 0 (N from 1 to 23)
30 1 * * 1-5	A 5-field cron expression: minute, hour, day of month, month, day of week

Case and extra spaces do not matter. Times are 24 hours. A raw cron expression accepts lists (`1,15`), ranges (`1-5`) and steps (`*/10`). pgCellar checks it when you save; a typo is refused with a message.

Time zone. Schedules run in the time zone of the pgCellar process, which is the `TZ` environment variable of the container. The Compose file sets `TZ: Europe/Brussels`; change it to yours. The container default is UTC.

`schedule.enabled` absent means: on, as soon as a `backup` schedule is written. Set `enabled: false` to pause a server's schedule and keep the line.

The scheduler

The block `scheduler:` governs all schedules:

```
scheduler:
  enabled: true      # false = pgCellar starts nothing by itself
  catch_up: 6h       # a due time missed while pgCellar was stopped still runs if younger than this
  parallel: 1        # backups running at once, all servers together
```

How it behaves:

- It wakes every 30 seconds and starts every backup whose due time has passed.
- A due time runs once. The result of the last handled due time is stored, so a restart neither replays nor forgets it.
- **Catch-up.** If pgCellar was stopped at 02:00 and starts at 03:30, the 02:00 backup still runs, because it is younger than `catch_up` (6 hours). Older than that, it is dropped. Several missed due times collapse into the latest one. `catch_up: 0` turns catch-up off. A server you add later, or a schedule you just changed, counts from now: nothing is run for a past that was not missed.
- **Busy.** If a manual backup is running on the same server when a due time arrives, the scheduled one is recorded as *skipped* and not retried. If all `parallel` slots are taken, the due time waits for the next tick and keeps its place.
- `parallel` counts every backup in progress, manual ones included, because it protects the Barman host.
- A skipped run is announced by an alert (chapter 8).

The server page shows **Next backup** and **Last scheduled run**.

5.3 Retention

Retention decides how long backups and their WAL are kept. Barman applies it, not pgCellar: Barman deletes obsolete backups at the end of each backup and during `barman cron`. pgCellar never runs `barman delete` for retention.

The server form has three fields:

Field	Barman line	Meaning
Recovery window (days)	<code>retention_policy = RECOVERY WINDOW OF 14 DAYS</code>	Keep enough backups to restore to any moment in the last 14 days
Redundancy (backups kept)	<code>retention_policy = REDUNDANCY 3</code>	Keep the 3 most recent backups
Minimum redundancy	<code>minimum_redundancy = 2</code>	Never delete below 2 backups, whatever the policy says

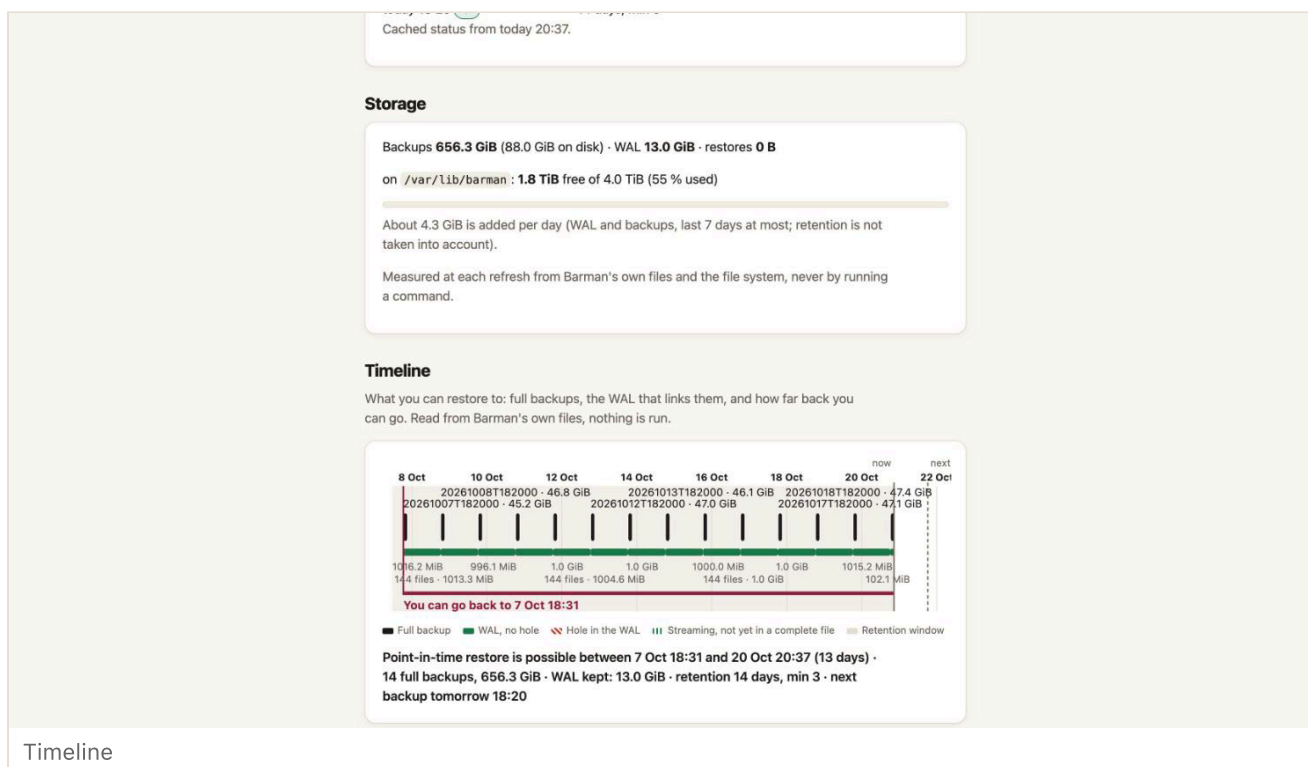
Use either a window or a redundancy, not both. The form and `pgcellar check` refuse both. 0 means "not set".

Advice:

- A **recovery window** matches how people think: "I want to go back 14 days". Barman keeps the newest backup older than the window, so that the whole window is restorable.
- Always set a **minimum redundancy** of at least 2. If backups start failing, retention would otherwise happily delete the last good ones.

The values are written to the Barman file when the form is saved. `retention_policy_mode` is left out, since Barman's default (`auto`) is what you want.

The Timeline marks the retention window on its axis; the Storage section (chapter 11) shows what the retention costs on disk.



5.4 Deleting a backup

On the server page, each complete backup has a **Delete...** button (admin). The row turns into a sentence with two buttons: **Delete** and **Keep it**. It runs:

```
barman delete pg-main 20261020T020001
```

pgCellar does not second-guess Barman. If the delete would go below `minimum_redundancy`, Barman refuses and its message comes back unchanged (HTTP 409 in the API). The backup id must look like `20261020T020001`. The command runs even if you close the tab, and it is audited as `backup.delete`.

The API form:

```
curl -X DELETE -H "Authorization: Bearer $KEY" http://localhost:8080/api/v1/servers/pg-main/backups/20261020T020001
```

Deleting is final. Deleting a backup also lets Barman remove the WAL no remaining backup needs.

5.5 Reading the result

A server is **red** when a check fails, there is no good backup, or the last backup is older than twice the limit; **orange** for any other problem (last attempt failed, backup older than `alerts.max_backup_age`, WAL not received for too long). The limits are in `alerts:` (26 hours for a backup, 24 hours for WAL, when you set nothing). Red means "act now"; orange means "look at it today".

Restores and point-in-time recovery

A restore in pgCellar writes a ready-to-start PostgreSQL data directory from a backup, replaying archived WAL up to the target you choose. **Nothing is touched on the production server.** The data directory lands in a folder on the pgCellar host. What you do with it next, you decide: inspect it, copy a table out of it, or replace the broken production directory.

The screenshot shows the pgCellar web interface. At the top, there's a navigation bar with links: Status, Servers, Runs, History, SSH key, Alerts, Audit. The 'Runs' tab is active. Below the navigation bar, the main content area displays 'Run #1' with a status of 'finished' in 9 min. The run details show a successful restore of the PostgreSQL data directory. The directory path is /tmp/pgcellar-demo-974107824/recoveries/crm-prod/20261015T185500-20261016T173700. The size is 12.0 GiB. The archive path is /tmp/pgcellar-demo-974107824/recoveries/crm-prod/20261015T185500-20261016T173700.tar.gz, with a size of 3.3 GiB. A note indicates that the archive is above recoveries.max_download_bytes, so there is no download link. Another note states that the directory is a ready-to-start PostgreSQL data directory. At the bottom, there's a 'Restore' button.

```

Run #1 (recover of crm-prod): finished in 9 min.

Restored. Sleep well. 🌞

DIRECTORY
/tmp/pgcellar-demo-974107824/recoveries/crm-prod/20261015T185500-
20261016T173700
Size: 12.0 GiB

ARCHIVE
/tmp/pgcellar-demo-974107824/recoveries/crm-prod/20261015T185500-
20261016T173700.tar.gz
3.3 GiB

The archive is above recoveries.max_download_bytes, so there is no download link: take
it from the path above on the host.

The directory is a ready-to-start PostgreSQL data directory: start a server of the same
major version on it.

Starting local restore for server crm-prod using backup 20261015T185500
Destination directory: /tmp/pgcellar-demo-974107824/recoveries/crm-prod/20261015T185500-
20261016T173700
Target time: 2026-10-16 11:37:00+02:00
Your PostgreSQL server has been successfully prepared for recovery!
Archive: /tmp/pgcellar-demo-974107824/recoveries/crm-prod/20261015T185500-
20261016T173700.tar.gz (3.3 GiB)
  
```

Restore

6.1 Before you restore: read the Timeline

The **Timeline** section of the server page draws what you can restore to: the full backups, the WAL that links them, the retention window, the holes. Two sentences matter:

- **You can go back to:** the earliest point in time. It is the end of the oldest backup whose WAL chain is unbroken, segment by segment, up to the newest archived WAL.
- **Latest:** the last data received when streaming is live, otherwise the arrival time of the last completed WAL file.

A hole in the WAL (a missing segment number between two files of the same timeline) is drawn in red, with the number of files missing. A backup before a hole cannot be used for a target after the hole.

The Timeline is read from Barman's own files, in a few milliseconds. No Barman command runs to draw it. The same data is at `GET /api/v1/servers/{name}/timeline`.

6.2 Starting a restore

On the server page, next to a complete backup, press **Restore...** (operator role). The Restore page asks for:

Choice	Meaning
Backup	A backup with status DONE. Failed or running backups cannot be restored.
Target: Latest possible	Replay all the WAL Barman holds.
Target: End of this backup	Stop as soon as the data is consistent. No WAL after that.
Target: Point in time	Stop at the date and time you give.
Also build a <code>.tar.gz</code> I can download	Packs the directory after the restore. Slower, needs room for a second copy.

For a point in time, type a date and time. Without a zone it is read in the time zone of the server. With a zone, it is exact:

```
2026-10-20 14:30:00+02:00
```

RFC 3339 (`2026-10-20T14:30:00+02:00`) works too. The target must not be before the end of the chosen backup; the form says so in a sentence. The other way, a target after the last WAL, is not checked: Barman reports the missing WAL.

The "End of this backup" target uses `--target-immediate` and not a target time equal to the end of the backup. A target time needs a WAL record dated after it, and a quiet database may never produce one.

The same operation from the API (the call returns at once; follow the run):

```
curl -X POST -H "Authorization: Bearer $KEY" -H "Content-Type: application/json" \
-d '{"backup_id":"20261020T020001","target":"time","target_time":"2026-10-20
14:30:00+02:00","archive":true}' \
http://localhost:8080/api/v1/servers/pg-main/recoveries
```

The answer is `202` with a `run_id`. `GET /api/v1/runs/{run_id}` shows the progress and the Barman output; `GET /api/v1/servers/pg-main/recoveries` lists the results.

6.3 Where the restore goes

```
<recoveries.dir>/<server>/<backup id>--<timestamp>
```

for example `/var/lib/pgcellar/recoveries/pg-main/20261020T020001-20261020T153012`. The default `recoveries.dir` is `/var/lib/pgcellar/recoveries`. This folder is inside the container.

Map it to the Docker host or you will not be able to read what you restored. In `docker-compose.yml`, replace the named volume with a bind mount owned by the barman uid:

```
volumes:
  - /srv/pgcellar/recoveries:/var/lib/pgcellar/recoveries
```

```
sudo mkdir -p /srv/pgcellar/recoveries && sudo chown 997:997 /srv/pgcellar/recoveries
```

A restore is as big as the database, so put it on a disk with room. The Storage section counts restores in what it reports.

Rules about that path, which protect your disks: it is always two levels deep under `recoveries.dir`, built only from a validated server name and a backup id, and checked again before every delete, purge and download. A database row that points elsewhere is refused, not followed.

6.4 The `.tar.gz`

If you ticked the box, pgCellar builds `<directory>.tar.gz` next to the directory after the restore, streaming (no file loaded in memory), written as `.partial` and renamed when complete. The **Download .tar.gz** button (operator role) serves it with byte ranges, so a download can resume.

Limits:

- Above `recoveries.max_download_bytes` (2 GiB by default; `0` = no limit) there is no download link, only the path: read it on the host.
- If the restore succeeded but the archive failed, the directory is kept and the run is marked failed with the reason.
- A failed `barman recover` removes its half-written directory.

6.5 Using the restored directory

The result is a PostgreSQL data directory, not a running server. Start a server **of the same major version** on it:

```
cd /srv/pgcellar/recoveries/pg-main/20261020T020001-20261020T153012
ls
# PG_VERSION, base, pg_wal, postgresql.auto.conf, recovery.signal ...
cat PG_VERSION
```

Barman wrote the recovery settings (the target) into `postgresql.auto.conf` and created `recovery.signal`; the WAL it needs is copied into `pg_wal`. Start PostgreSQL on that directory, on a free port, and let it replay. When it reaches the target it promotes itself and accepts connections. Typical way to inspect it:

```
docker run --rm -v "$PWD":/var/lib/postgresql/data -p 55432:5432 postgres:16
```

Use the image of your own major version, and check that the user in the container owns the files.

To restore over production, stop production, keep the damaged directory aside, put the restored one in its place, fix ownership and permissions, and start. That step is yours: pgCellar deliberately never

writes to a production host.

6.6 Retention of restores

Restores older than `recoveries.keep_days` (7 by default; `0` keeps everything) are removed by a daily pass: directory, archive and record. An admin can also delete one by hand (**Delete** in the Recoveries list, or `DELETE /api/v1/recoveries/{id}`). Deletion is audited.

Roles: an operator starts a restore and downloads the archive; an admin deletes. A viewer sees the list and the paths but cannot download, because the archive is a full copy of the database and more sensitive than a backup list. Every download, start, delete and purge is audited (`recover.start` , `recovery.download` , `recovery.delete` , `recovery.purge`).

6.7 Limits

- Restore is **local**: `barman recover` writes to a folder on the pgCellar host. There is no "restore to a remote host" here. Copy the directory yourself.
- One restore per server at a time.
- Only a backup with status DONE can be restored.
- Timeline switches (after a failover) are followed when the history file is present. Without a `.history` file the fork point is unknown, and the chain is treated as broken.
- A backup of an old major version cannot start on a newer PostgreSQL. Use the same major version as the source.
- Restoring does not prove that a *whole system* works (extensions, configuration files outside the data directory, roles). A restore drill does the next best thing, automatically (chapter 7).

Restore drills and reports

A backup you have not restored is a hope. A restore drill is pgCellar restoring one of your backups for real, every week, on its own, starting the copy, looking inside it, and writing a dated PDF that you can hand to an auditor.



No restore yet. Use "Restore..." to write a backup, or a point in time, to a directory.

Restore drills

A drill restores a backup into a throwaway PostgreSQL of the same version, checks that it starts and what it holds, and writes a dated PDF report.

Restore verified

On 18 Oct 2026 at 04:30 the backup 20261017T182000 of erp-prod was restored and started successfully in 16 min 40 s; 2 databases, 359 tables. Data checksums were verified.

Last drill: 2 d ago ✓

[Report \(PDF\)](#)

DRILL SCHEDULE

weekly sun 04:30

History

NEXT DRILL

Sun 25 Oct 04:30

verified	18 Oct, 04:30	20261017T182000	2 db, 359 tables	Report (PDF)

Checks

✓ 16 checks OK

The list comes from the last refresh. Use "Run check now" for a live check.

Configuration

Restore drills

7.1 What a drill does, step by step

A drill never touches the production server. It works on a copy of a backup, under `recoveries.dir`. One drill at a time, never during a backup of the same server (a backup of that server waits while its drill runs).

Step	What happens
1. Restore the backup	<code>barman recover</code> into <code><recoveries.dir>/<server>/drill-<timestamp>/</code> . The latest backup with status DONE, target latest.
2. Find the PostgreSQL version	Reads <code>PG_VERSION</code> of the restored directory to choose the binaries of the same major version.
3. Prepare the throwaway copy	Appends a few settings to the <code>postgresql.auto.conf</code> Barman wrote (see below).
4. Start PostgreSQL	<code>pg_ctl start -w -t 300</code> from <code>/usr/lib/postgresql/<major>/bin</code> , on a free port between 54400 and 54499.
5. Wait for the end of the recovery	<code>pg_isready</code> until the copy accepts connections, then <code>SELECT pg_is_in_recovery()</code> must answer <code>f</code> . Limit: <code>drills.recovery_timeout</code> .
6. Look inside the databases	Runs queries: <code>version()</code> , the replay LSN, every non-template database with its size and its number of tables.
7. Stop PostgreSQL	<code>pg_ctl stop</code> , fast, then immediate if needed.
8. Verify the data checksums	<code>pg_checksums --check</code> on the stopped copy, only if the cluster uses checksums. Skipped, with the reason, if not.
9. Clean up	Removes the directory, unless it is the last good drill and <code>keep_last</code> is on.

Why checksums come after the stop: `pg_checksums --check` refuses a cluster that was not shut down cleanly, and the control file of a restored base backup says "in production" until PostgreSQL has stopped once.

What the throwaway copy is configured with. It listens on `127.0.0.1` only, with a trust rule on its own socket and TCP rejected. It runs with `archive_mode = off`, `max_connections = 10`, `shared_buffers = 64MB`, `max_wal_senders = 0`, `autovacuum = off`, `logging_collector = off`, `ssl = off`, `jit = off`, `hot_standby = off`, an empty `shared_preload_libraries` (so an extension library missing from the image cannot stop the check), and the locale messages set to `C`. None of this changes your data; it only lets a small copy start anywhere.

What the check proves, and what it does not. It proves that the backup restores, the WAL replays, PostgreSQL starts on it, the databases are there, their tables can be counted, and, when checksums are on, no page is damaged. With `drills.verify_tables: true` it also runs `SELECT count(*)` on every table (slow on large databases) and sums the rows per database. It does not compare the data with the source, and it does not test your application. A source that needs an extension library to replay WAL is started without it and is not tested for it. Configuration files outside the data directory (Debian's `/etc/postgresql`) are not handled.

When a step fails, the report keeps the earlier steps, the error text and the last 25 lines of the PostgreSQL log of the copy. pgCellar still stops the copy and removes the directory.

7.2 Switching drills on

Drills are scheduled per server, in `pgcellar.yaml`:

```

servers:
  - name: pg-main
    drills:
      schedule: "weekly sun 04:30" # same syntax as the backup schedule; empty = none

drills:
  enabled: true          # false stops the scheduled drills; manual ones stay possible. Absent = true
  timeout: 30m           # a whole drill, restore included
  recovery_timeout: 10m  # wait for the copy to finish replaying WAL
  keep_last: true        # keep the data directory of the last good drill. Absent = true
  verify_tables: false   # true = SELECT count(*) on every table
  max_age: 192h          # the server turns orange when the last good drill is older (8 days)
  bin_dir: /usr/lib/postgresql

```

Do not write `enabled: false` unless you mean it; the `absent` key already means "on". A drill schedule only takes effect when the server has one.

Choose a time after the nightly backup. The schedule uses the same syntax as backups (chapter 5). The drill schedule is edited in `pgcellar.yaml`, not in the server form; the form keeps it when you save.

To run one now: **Run a drill now** in the Restore drills section (operator), or:

```
curl -X POST -H "Authorization: Bearer $KEY" http://localhost:8080/api/v1/servers/pg-main/drills
```

The body is optional. By default the drill uses the latest DONE backup and target `latest`. You can give `backup_id`, `target` (`latest`, `backup_end`, `time`) and `target_time`. The call answers `202` with a `run_id`; the run output lists every command with its answer. It returns `409` while another drill or a backup of the same server runs.

A server turns **orange** in Status when its last drill failed, or when no drill has passed for `max_age`. A server that never had a drill is not orange: its first due time is not late. Metrics: `pgcellar_last_drill_ok{server}` and `pgcellar_last_drill_age_seconds{server}`.

7.3 What the image must contain

A drill starts the restored copy with the PostgreSQL **server** binaries of the same major version as the source. The Docker image installs them for the versions in the build argument `PG_MAJOR` (default `16 17 18`), as binaries only: no cluster, no service. If the source's major version is not there, the drill fails at step 2 with "PostgreSQL binaries are not in this image". Rebuild:

```
docker build -f deploy/Dockerfile -t pgcellar:dev --build-arg PG_MAJOR="15 16 17" .
```

Sizes, measured on arm64 (uncompressed): 312 MB without the server binaries, 639 MB with `16 17 18`. A single version is about 170 MB more than none. Fewer versions, smaller image.

Locales

A restored data directory keeps the collation and ctype locale of its source, and PostgreSQL refuses to start on a locale the system does not have. A source cluster created in a `postgres:16-alpine` or `postgres:16` container is typically `en_US.utf8`.

The image generates a fixed list from the build argument `PG_LOCALES`: `C.UTF-8 en_US.UTF-8 en_GB.UTF-8 fr_FR.UTF-8 fr_BE.UTF-8 nl_NL.UTF-8 nl_BE.UTF-8 de_DE.UTF-8 es_ES.UTF-8 it_IT.UTF-8 pt_BR.UTF-8`. That adds about 23 MB. If your cluster uses another one, add it (name as in `/usr/share/i18n/SUPPORTED`) and rebuild:

```
docker build -f deploy/Dockerfile -t pgcellar:dev \
  --build-arg PG_LOCALES="C.UTF-8 en_US.UTF-8 pl_PL.UTF-8" .
```

A start that fails on a locale is reported as: "The cluster uses locale X, which this image does not have. Rebuild with PG_LOCALES including Y (see deploy/Dockerfile)." Find your cluster's locale with `SELECT datcollate, datatype FROM pg_database;`.

7.4 Disk and memory

A drill needs room for the database plus its WAL under `recoveries.dir`. With `keep_last: true` the last good copy of each server stays there, so you can look at it; `du -sh <recoveries.dir>/*/drill-*` tells what it costs. Set `keep_last: false` if the disk is tight.

RAM: a drill adds some tens of MB (the copy runs with `shared_buffers = 64MB`). Drills never overlap.

Only the last **successful** copy is ever kept: the previous kept copy is removed when a new drill passes, and a failed drill never touches it. A `drill-*` directory that no record points to and that is older than a day (a restart during a drill) is removed by the daily purge. Records and reports are never purged; they weigh a few KB each.

7.5 The PDF report

Every drill writes a PDF, passed or failed:

```
<recoveries.dir>/<server>/reports/drill-<timestamp>.pdf
```

It opens from the server page (**Report (PDF)**, operator role), from `GET /api/v1/drills/{id}/report`, and it is on the host if you mapped the folder. It contains:

- a banner, **RESTORE VERIFIED** or **FAILED**, and the verdict in one sentence;
- the drill in a table: server, source backup, recovery target, date, duration, PostgreSQL version, WAL position reached (LSN), restored size, data checksums, who started it, and the run number;

- the steps, with the result and the duration of each, and a note (the failing step and the error text when there is one);
- the databases found in the copy, with size and number of tables, and the rows counted when `verify_tables` is on;
- for a failed drill, "What went wrong": the error and the end of the PostgreSQL log of the copy;
- a closing paragraph that says what the report records, and a footer with the fingerprint.

The report is written by pgCellar's own small PDF writer, with no dependency. Each file is read back the way a viewer would start it (header, cross-reference table, every offset and stream length) before it is stored.

The fingerprint

The footer carries **SHA-256 of the report content**: the hash of the text of the body, in the order it is drawn. It is also stored with the drill record, and returned by the API (`details.fingerprint`). To check a report, compute the hash of its text and compare it with the record. This detects a *changed text*. It does not detect a changed layout, and it is not a digital signature: anyone with write access to the report folder could produce a new file and a new hash. If you need non-repudiation, copy the PDF and the fingerprint to a system you do not control, and sign them there.

7.6 Reading a failed drill

The server page shows the last result large ("Restore verified" or "Restore drill failed"), the history, and the failing step. The run behind it holds the whole journal: each command with its output, step by step. Chapter 13 lists the usual causes: a missing locale, a missing major version, a `noexec` temporary directory, a source that needs an extension.

Alerts

pgCellar writes to you when something needs a human, and keeps quiet otherwise. Three channels: Telegram, mail and a signed webhook. Use one, two or all three.

Demo · read-only · fictional data · Get yours at dbfactory.io/pgcellar


[Status](#)
[Servers](#)
[Runs](#)
[History](#)
[SSH key](#)
[Alerts](#)
[Audit](#)
demo (viewer) [Log out](#)

Alerts

pgCellar writes to you when a server changes level, a backup or a restore drill fails, a scheduled run is skipped, and (if you ask for it) once a day with a summary. Secrets are never kept in pgcellar.yaml: it only names the environment variables that hold them.

Channels

telegram chat -1001234567890
PGCELLAR_TELEGRAM_TOKEN is set
Send a test message

mail dba@northwind.example,
ops@northwind.example
PGCELLAR_SMTP_USER is set PGCELLAR_SMTP_PASSWORD is set
Send a test message

A problem that lasts is announced once, not at every refresh. It is announced again after 24 h. A summary goes out every day at 08:30.

Latest messages

🚨 analytics needs attention
telegram sent today 16:33

❌ shop-staging is failing
mail sent 18 Oct, 19:07

❌ shop-staging is failing
telegram sent 18 Oct, 19:07

✅ Restore drill of shop-staging failed
telegram sent 18 Oct, 19:07

Alerts

8.1 What triggers a message

Kind	When	Example title
Level change	A server goes from healthy to attention or failing, or from attention to failing and back	pg-main is failing, pg-main needs attention
Resolved	A server is healthy again (one message)	pg-main is healthy again
Reminder	A problem that lasts, after <code>repeat_every</code>	pg-main is still failing (for 2 days)
Backup failed	A backup run ended with an exit code other than 0, scheduled or manual	Backup of pg-main failed
Drill failed	A restore drill failed	Restore drill of pg-main failed
Drill passed	Only with <code>on_drill_success: true</code>	Restore drill of pg-main passed
Skipped	A scheduled backup or drill could not start	Scheduled backup of pg-main did not run
Daily summary	Once a day, if you ask for it	Everything is fine

The level of a server is the one the Status page shows. Storage thresholds, old backups, a stopped WAL receiver, a failed or too-old drill all change it, so they need no message of their own: the level change covers them.

8.2 Configure the channels

Secrets never go in `pgcellar.yaml`. The file names the **environment variables** that hold them; you set those variables in the container.

```
alerts:
  max_backup_age: 26h
  repeat_every: 24h
  on_drill_success: false
  daily_summary: "08:30"
  telegram:
    token_env: PGCELLAR_TELEGRAM_TOKEN
    chat_id: "-1001234567890"
  mail:
    host: smtp.example.com
    port: 587
    starttls: true
    from: pgcellar@example.com
    to: [dba@example.com]
    user_env: PGCELLAR_SMTP_USER
    password_env: PGCELLAR_SMTP_PASSWORD
  webhook:
    url: https://hooks.example.com/pgcellar
    secret_env: PGCELLAR_WEBHOOK_SECRET
```

In `docker-compose.yml`, under `environment:` of the `pgcellar` service:

```
environment:
  TZ: Europe/Brussels
  PGCELLAR_TELEGRAM_TOKEN: "123456:ABC..."
  PGCELLAR_SMTP_USER: "pgcellar@example.com"
  PGCELLAR_SMTP_PASSWORD: "..."
  PGCELLAR_WEBHOOK_SECRET: "a long random string"
```

Better: keep them in a `.env` file next to the Compose file (mode 0600, not in git) and reference them as `${PGCELLAR_TELEGRAM_TOKEN}`.

pgCellar reads `pgcellar.yaml` again when it changes, so a new channel needs no restart. A changed environment variable needs a container restart, because the environment is fixed when a process starts.

Variable	Used by	Needed
<code>PGCELLAR_TELEGRAM_TOKEN</code> (any name you choose in <code>token_env</code>)	Telegram	yes
<code>PGCELLAR_SMTP_USER</code> , <code>PGCELLAR_SMTP_PASSWORD</code> (names from <code>user_env</code> , <code>password_env</code>)	Mail	only if the SMTP server needs a login
<code>PGCELLAR_WEBHOOK_SECRET</code> (name from <code>secret_env</code>)	Webhook	only to sign the body

The names are yours. The ones above are the ones used in the examples.

Telegram

Create a bot with BotFather, copy its token into the variable. Add the bot to the chat or channel and use that chat's id (for a group, a negative number like `-1001234567890`). The message is sent in Telegram's HTML subset; a link opens the server in pgCellar.

Mail

`host`, `from` and at least one address in `to` are required. Port 587 with `starttls: true` (the default) upgrades the connection with STARTTLS, which is required; port 465 means implicit TLS. `starttls: false` sends in clear: use it only for a relay on a trusted network. AUTH PLAIN is used only when `user_env` is set. Messages go as plain text.

Webhook

pgCellar POSTs JSON to `url`. With `secret_env` set, the exact body is signed with HMAC-SHA256 and the result is in a header:

```
X-PgCellar-Signature: sha256=<hex>
```

To verify, compute the HMAC-SHA256 of the raw body with the same secret and compare in constant time. The body:

```
{
  "kind": "backup_failed",
  "server": "pg-main",
  "level": "fail",
  "title": "Backup of pg-main failed",
  "lines": ["The backup (scheduled, run #212) ended with exit code 1."],
  "action": "Open the server page, read the problems listed there and start with the first one.",
  "url": "https://backup.example.com/servers/pg-main",
  "at": "2026-10-21T02:03:14+02:00",
  "text": "❌ Backup of pg-main failed\n\n- The backup (scheduled, run #212) ended with exit code 1.\n\nWhat to do: ...\n\nhttps://backup.example.com/servers/pg-main"
}
```

`level` is `ok`, `warn` or `fail`. `text` is ready to show in a chat tool. Any answer other than 2xx counts as a failure in the journal.

Set `base_url` in `pgcellar.yaml`: links in messages are built from it.

8.3 Anti-noise rules

Nobody reads a channel that cries wolf. pgCellar follows these rules:

- **One message per change.** A server that stays red is not announced at every refresh. It is announced when it turns red, repeated after `repeat_every` (24 hours by default; `0` = never repeat), and a single "healthy again" follows when it recovers.
- **Orange to red and back** are announced at once.
- **The first two minutes after a start are silent.** A server can look red for a minute while the status is rebuilt. Nothing is sent either while the status shown is the one reloaded from the database and not yet refreshed.
- **A restart neither repeats nor forgets.** The last level announced for each server, since when, and when it was said, are stored in the database.
- **History is not news.** The first look after the database is created only records a baseline for runs and drills.
- **No channel, no state.** With no channel configured, the alert engine does nothing at all. A channel added later does not replay old events; it announces the servers that are red at that moment.
- **A skipped scheduled run is announced once per due time.**

Alerts are decided by one pass every 30 seconds over the state pgCellar already keeps. There is no separate monitoring.

8.4 Daily summary

`daily_summary: "08:30"` sends one message a day at that time: the headline of the Status page, then one line per server. It goes out at most 12 hours late (so a restart at night does not send yesterday's), only with at least one server, and only with a fresh status. Empty = no summary.

⚠️ 1 server needs attention

- erp-prod ✅ · last backup 2 h ago · streaming live · restore verified 3 d ago · 1.8 TB free (57% used)
- analytics ⚠️ · last backup 31 h ago · streaming live · restore verified 5 d ago · 1.8 TB free (57% used)

What to do: Open pgCellar and start with the servers marked ❌, then the ones marked ⚠️.

8.5 Example messages

A server fails:

❌ analytics is failing

- The last backup is 53 hours old, the limit is 26.
- WAL receiver is not running.

What to do: WAL streaming: Start the receiver with "barman cron", then check that the slot exists on PostgreSQL.

Open in pgCellar

A restore drill fails:

❌ Restore drill of pg-main failed

- It failed at the step "Start PostgreSQL": the backups of this server are not proven restorable.

What to do: Open the server page, read the last restore drill and fix its failing step, then run the drill again.

Recovered:

✅ pg-main is healthy again

- The problem lasted 3 h 12 min.

These examples are illustrative: the exact lines depend on the problem found. The wording is dry on purpose; errors say what to do.

8.6 Test and journal

The **Alerts** page (admin) shows which channels are configured, whether each environment variable is set (never its value), a **Send a test message** button per channel, and the latest 2000 messages sent, with the outcome of each. Over the API: `GET /api/v1/alerts` and `POST /api/v1/alerts/test` (body `{"channel": "telegram"}`, or empty for all). The metric `pgcellar_alerts_sent_total{channel,outcome}` counts the sends.

A send has a 10-second limit. A failure is logged and written to the journal, never fatal, and the Telegram token is removed from error texts.

8.7 Thresholds

Key	Default	Effect
<code>max_backup_age</code>	26h	The last good backup is older than this: the server is orange (red at twice the limit)
<code>max_wal_age</code>	24h	No WAL received for this long: orange. A quiet database closes a 16 MB file rarely; with streaming live, an old completed file is only a note
<code>min_free_percent</code>	10	The file system holding the backups has less free space: its servers go orange, red under half. <code>0</code> = off
<code>min_free_bytes</code>	0 (off)	Same, in bytes, for example <code>50000000000</code> for 50 GB
<code>repeat_every</code>	24h	Repeat a lasting problem after this delay. <code>0</code> = never
<code>on_drill_success</code>	false	Also announce a passing drill
<code>daily_summary</code>	empty	Time of day, <code>"08:30"</code>

The full list is in chapter 10.

Users, roles, API keys and the API

People sign in to the console with a user. Programs call the API with an API key. Both have one of three roles.

9.1 Roles

Role	Can do
viewer	Read everything except secrets and the audit log: Status, Servers, Timeline, Server setup, Runs, History, restores and drills lists, the license.
operator	Everything a viewer can, plus: run a check, backup now, switch WAL, start a restore, download a restore archive, start a drill, download a drill report.
admin	Everything an operator can, plus: add and edit servers, delete a backup, delete a restore, test SSH, manage SSH keys, read the Alerts page and send test messages, read the audit log.

The audit log is admin-only because it contains the exact command lines that were run. A restore archive is a full copy of the database, so downloading it needs the operator role.

At least one user must have the admin role, or pgCellar refuses to load the file.

9.2 Users

Users live in `pgcellar.yaml`. There is no user screen: you edit the file, and pgCellar picks the change up.

```
users:
  - name: admin
    role: admin
    password_hash: "$2a$10$..."
  - name: alice
    role: operator
    password_hash: "$2a$10$..."
  - name: bob
    role: viewer
    password_hash: "$2a$10$..."
```

Create a hash with:

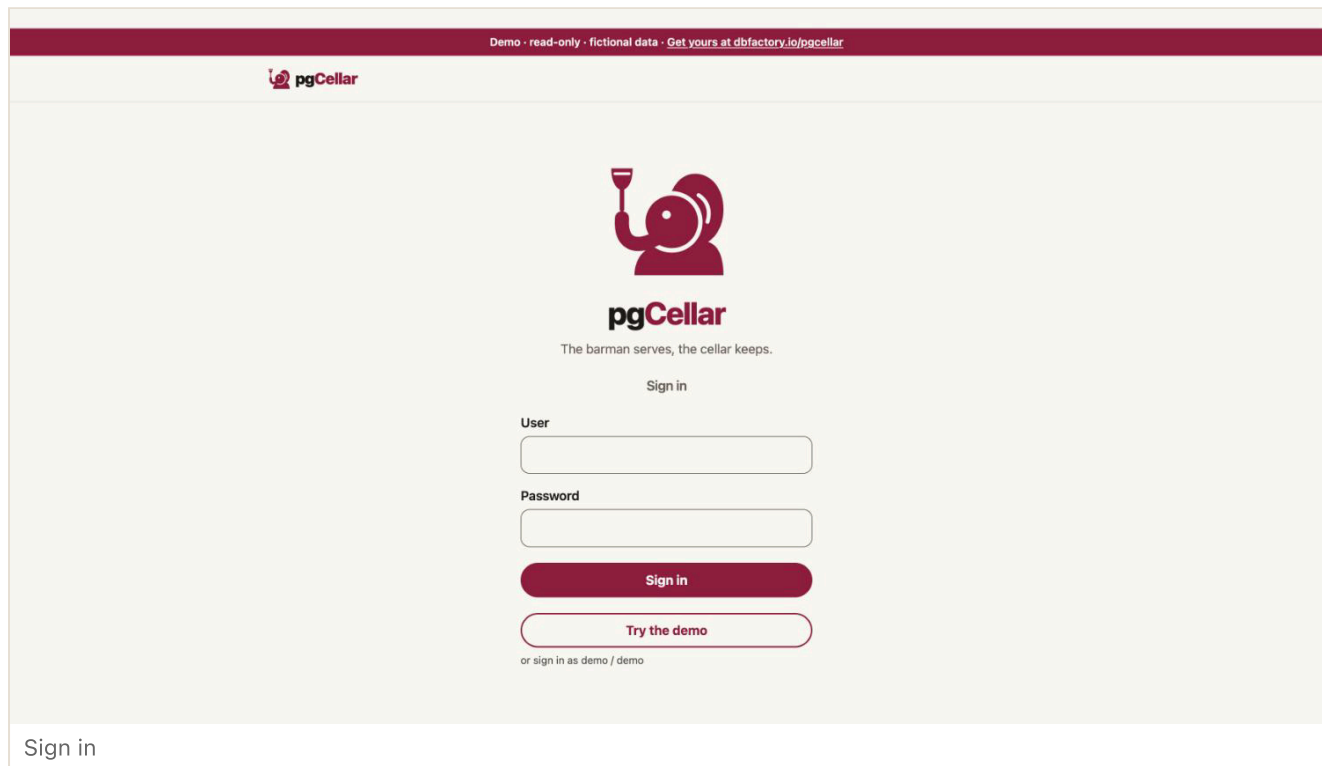
```
read -rs PW; printf '%s\n' "$PW" | docker compose run --rm -T pgcellar hash-password
```

Minimum 8 characters. The hash is bcrypt; the password itself is never stored. Paste the hash into `password_hash`.

To remove a user, delete the entry; their access ends at the next request, because the role is read from the file on every request.

9.3 Sessions and sign-in

The console uses a cookie session valid for 24 hours. The login form is protected against password guessing (chapter 12). Each state-changing request carries a CSRF token. Log out with **Log out** in the top bar.



9.4 API keys

An API key is a long random string starting with `pgc_`. pgCellar stores only its SHA-256, in `pgcellar.yaml`. The key itself is shown once, when you create it.

```
docker compose exec pgcellar pgcellar apikey create --config /etc/pgcellar/pgcellar.yaml --name grafana --role viewer
```

The command prints the key and nothing else. Save it at once; there is no way to read it again. It writes the hash into `pgcellar.yaml` and commits the change to the configuration history, and the running server picks it up by itself. List the names and roles:

```
docker compose exec pgcellar pgcellar apikey list --config /etc/pgcellar/pgcellar.yaml
```

To revoke a key, delete its entry in `api_keys:`.

Give each program its own key and the smallest role that works: a monitoring tool needs `viewer`, a script that triggers backups needs `operator`.

9.5 Calling the API

The base path is `/api/v1`. Authenticate with a header:

```
curl -H "Authorization: Bearer pgc_..." http://localhost:8080/api/v1/status
```

Every answer is JSON, either `{"data": ...}` or `{"error": "..."}` . A request body is JSON, limited to 64 KB, and unknown fields are refused.

Code	Meaning
200, 201, 202, 204	Done; 202 means started in the background, follow <code>run_id</code>
401	Missing or invalid key (<code>WWW-Authenticate: Bearer</code>)
403	The key's role is too low
404	No such server, run, backup or restore
409	Conflict: another job is running, Barman refused a delete, the archive does not exist
413	The archive is larger than <code>recoveries.max_download_bytes</code>
422	The input is invalid; <code>problems</code> lists each reason
502	Barman or the command failed; its message is in <code>error</code>
503	The first status refresh is not done yet; retry after the <code>Retry-After</code> delay

Reading never runs Barman. `GET` routes answer from the status cache (refreshed every `status_interval`) and from Barman's files. Only an explicit `POST` calls Barman.

The main routes

GET	/api/v1/status	cached state of every server
GET	/api/v1/servers	list
POST	/api/v1/servers	create (admin)
GET	/api/v1/servers/{name}	one server with its state
PUT	/api/v1/servers/{name}	update (admin)
GET	/api/v1/servers/{name}/setup	Server setup checks
GET	/api/v1/servers/{name}/timeline	what you can restore to
POST	/api/v1/servers/{name}/check	run barman check now (operator)
POST	/api/v1/servers/{name}/backup	run a backup now (operator)
POST	/api/v1/servers/{name}/switch-wal	force a WAL switch (operator)
GET	/api/v1/servers/{name}/backups	list backups
DELETE	/api/v1/servers/{name}/backups/{id}	delete a backup (admin)
POST	/api/v1/servers/{name}/recoveries	restore to a directory (operator)
GET	/api/v1/servers/{name}/recoveries	restores of a server
GET	/api/v1/recoveries/{id}	one restore
GET	/api/v1/recoveries/{id}/download	the .tar.gz (operator)
DELETE	/api/v1/recoveries/{id}	delete a restore (admin)
POST	/api/v1/servers/{name}/drills	run a drill now (operator)
GET	/api/v1/servers/{name}/drills	drills of a server
GET	/api/v1/drills	drills of all servers (?server=, ?limit=)
GET	/api/v1/drills/{id}	one drill with its steps
GET	/api/v1/drills/{id}/report	the PDF report (operator)
GET	/api/v1/runs	recent runs (?server=, ?limit=)
GET	/api/v1/runs/{id}	one run with its output
GET	/api/v1/history	git history of the configuration
GET	/api/v1/audit	audit log (admin)
GET	/api/v1/alerts	channels and journal (admin)
POST	/api/v1/alerts/test	send a test message (admin)
GET	/api/v1/license	edition, servers allowed and used
GET	/api/v1/ssh/authorized-keys	keys allowed to push WAL (admin)
POST	/api/v1/ssh/authorized-keys	authorise a key (admin)
DELETE	/api/v1/ssh/authorized-keys	revoke a key (admin)

The complete description, with parameters, bodies, schemas and the role each route needs, is served by pgCellar itself, as OpenAPI 3:

```
GET /api/v1/openapi.json
```

That route needs no key. Load it in Swagger UI, Postman or your generator of choice. It is always the description of the version you are running.

Examples

Is everything fine?

```
curl -s -H "Authorization: Bearer $KEY" http://localhost:8080/api/v1/status | jq '.data.servers[] | {name, level}'
```

Back up now and follow the run:

```
RUN=$(curl -s -X POST -H "Authorization: Bearer $KEY" http://localhost:8080/api/v1/servers/pg-main/backup  
| jq -r .data.run_id)  
curl -s -H "Authorization: Bearer $KEY" http://localhost:8080/api/v1/runs/$RUN | jq '.data | {status,  
exit_code}'
```

Create a server:

```
curl -s -X POST -H "Authorization: Bearer $KEY" -H "Content-Type: application/json" \  
http://localhost:8080/api/v1/servers -d '{  
  "name": "pg-main",  
  "backup_method": "postgres",  
  "wal_method": "streaming",  
  "conninfo": "host=pg-main.example.com user=barman dbname=postgres",  
  "streaming_conninfo": "host=pg-main.example.com user=streaming_barman dbname=postgres",  
  "slot_name": "barman"  
}'
```

Whatever the console does, the API does: both are thin layers over the same code. An automation tool can configure and operate pgCellar without a browser.

Configuration reference

Everything that belongs to pgCellar lives in one file, `pgcellar.yaml`. Barman's own settings stay in `/etc/barman.conf` and `/etc/barman.d/*.conf`. Secrets never go in this file: use the `*_env` keys, which name an environment variable, and hashes.

The example in `deploy/pgcellar.example.yaml` is commented line by line. Validate any file with:

```
pgcellar check --config /etc/pgcellar/pgcellar.yaml
```

10.1 The rule: an absent key is the default

If a key is not in the file, pgCellar uses its default. Nothing else. This matters for the settings where the default is not the "zero" value of the language, such as `drills.enabled` (default **on**) or `alerts.repeat_every` (default **24h**, where `0` means *never*).

What follows from it:

- Do not write a key just to write its default. A line `drills: {enabled: false}` is a decision (no scheduled drills), not a placeholder.
- When pgCellar saves the file (a new API key, an edited server), it does not write defaults you never chose. An absent key stays absent. A value you wrote, including an explicit `0`, is kept.
- `pgcellar check` warns about a line that looks like a leftover: `drills.enabled: false` with no drill schedule on any server. Files are never rewritten automatically.

This rule exists because of a real bug: a version once wrote `drills: {enabled: false}` from the zero value when it saved the file, and later versions read that `false` as a choice and silently turned off every drill.

Durations are written in the Go style: `30s`, `5m`, `26h`, `192h`. There is no day unit; 8 days is `192h`.

Reload. pgCellar re-reads the file when its modification time changes, if the new content is valid. A change made in the console reloads the file first, so a hand edit is never overwritten.

10.2 Top level

Key	Default	Meaning
<code>listen</code>	<code>":8080"</code>	Address and port of the console and the API. Plain HTTP; put a proxy in front for TLS (chapter 12).
<code>base_url</code>	empty	Public address, used in links inside alerts and reports, for example <code>https://backup.example.com</code> .
<code>data_dir</code>	<code>/var/lib/pgcellar</code>	SQLite database (<code>pgcellar.db</code>) and runtime files.
<code>status_interval</code>	<code>1m</code>	How often the cached status of every server is refreshed. Minimum <code>30s</code> . Each refresh runs about three Barman commands per server (2 to 4 seconds each on a small machine), so on a small host with many servers, raise it.
<code>setup_interval</code>	<code>30m</code>	How often the PostgreSQL settings are read again for Server setup. Minimum <code>1m</code> . One read-only <code>psql</code> query per server. A server whose settings could not be read is tried again after 5 minutes.
<code>ssh_host</code>	empty	How PostgreSQL servers reach this host over SSH. Used in the advised <code>archive_command</code> . Empty means this machine's host name, which in Docker is a container id: set it.
<code>ssh_port</code>	<code>2222</code>	Port of the SSH companion container, written as <code>--port</code> in the advised <code>archive_command</code> .
<code>audit_retention_days</code>	<code>90</code>	Audit lines older than this are purged daily. <code>0</code> keeps everything.

10.3 barman

Where Barman lives and how to call it.

Key	Default	Meaning
<code>binary</code>	<code>barman</code>	On <code>PATH</code> , or an absolute path.
<code>config_file</code>	<code>/etc/barman.conf</code>	Barman's global file.
<code>config_dir</code>	<code>/etc/barman.d</code>	One <code><server>.conf</code> per server. A separate git repository tracks this folder when it is not under the pgCellar one.
<code>user</code>	<code>barman</code>	Commands run as this user when pgCellar runs as root. In the image pgCellar already runs as <code>barman</code> and no <code>sudo</code> is used.
<code>host</code>	empty	Reserved for a remote Barman host. Leave empty (local Barman).
<code>cron_interval</code>	<code>1m</code>	How often pgCellar runs <code>barman cron</code> . <code>0</code> = never. <code>pgcellar serve --cron-interval</code> overrides it.

10.4 scheduler

Key	Default	Meaning
<code>enabled</code>	<code>true</code>	<code>false</code> = pgCellar starts nothing by itself. The schedule fields keep showing.
<code>catch_up</code>	<code>6h</code>	A due time missed while pgCellar was stopped still runs if it is not older than this. <code>0</code> = never catch up.
<code>parallel</code>	<code>1</code>	Backups running at once, all servers together. Must be at least 1.

10.5 servers

A list. What pgCellar adds to a Barman section; `name` must match a `[section]` in `barman.d`.

Key	Default	Meaning
<code>name</code>	required	Lowercase letters, digits, <code>-</code> and <code>_</code> . Unique.
<code>label</code>	empty	Free text shown in the console.
<code>schedule.backup</code>	empty	<code>daily 02:00</code> , <code>weekly sun 03:00</code> , <code>hourly</code> , <code>every 6 hours</code> , or a 5-field cron expression.
<code>schedule.enabled</code>	on when <code>backup</code> is set, off otherwise	Pause the schedule without deleting it.
<code>retention.recovery_window_days</code>	<code>0</code> (not set)	Barman <code>RECOVERY WINDOW OF n DAYS</code> .
<code>retention.redundancy</code>	<code>0</code> (not set)	Barman <code>REDUNDANCY n</code> . Do not combine with a window.
<code>retention.minimum_redundancy</code>	<code>0</code>	Never delete below this number of backups. Cannot exceed <code>redundancy</code> .
<code>drills.schedule</code>	empty	Schedule of the automatic restore drill, same syntax. Empty = none.
<code>tags</code>	empty	Free labels for your own use.

The retention values are written to the Barman file when the server form is saved; pgCellar does not apply retention itself.

10.6 alerts

Key	Default	Meaning
<code>max_backup_age</code>	<code>26h</code>	Orange when the last good backup is older than this; red at twice the limit.
<code>max_wal_age</code>	<code>24h</code>	Orange when WAL has not been received for this long.
<code>min_free_percent</code>	<code>10</code>	Orange when the file system holding the backups has less free space (percent); red under half of it. <code>0</code> = off.
<code>min_free_bytes</code>	<code>0</code> (off)	Same, in bytes. Either threshold alone is enough.
<code>repeat_every</code>	<code>24h</code>	A lasting problem is announced again after this delay. <code>0</code> = never repeat.
<code>on_drill_success</code>	<code>false</code>	Also send a message when a drill passes.
<code>daily_summary</code>	empty	Time of day, <code>"08:30"</code> . Empty = no summary.
<code>telegram.token_env</code>		Name of the environment variable holding the bot token.
<code>telegram.chat_id</code>		Chat or channel id, as a string.
<code>mail.host</code> , <code>mail.port</code>		SMTP server. <code>host</code> , <code>from</code> and <code>to</code> are required; the port must be valid.
<code>mail.starttls</code>	<code>true</code>	Upgrade the connection with STARTTLS. <code>false</code> sends in clear.
<code>mail.from</code> , <code>mail.to</code>		Sender, and a list of recipients.
<code>mail.user_env</code> , <code>mail.password_env</code>		Names of the variables for the SMTP login. Optional.
<code>webhook.url</code>		<code>http://</code> or <code>https://</code> .
<code>webhook.secret_env</code>		Name of the variable holding the secret that signs the body. Optional.

No channel configured means no message is ever sent.

10.7 license

Key	Default	Meaning
<code>key</code>	empty	A licence key, <code>PGC1.xxxx.yyyy</code> .
<code>file</code>	empty	A file that holds the key.

Neither set means the Free edition (chapter 14).

10.8 users and api_keys

```
users:
- name: admin
  role: admin                # admin, operator or viewer
  password_hash: "$2a$10$..." # bcrypt, from "pgcellar hash-password"

api_keys:
- name: grafana
  role: viewer
  key_hash: "9f86d0..."    # SHA-256 hex, written by "pgcellar apikey create"
```

At least one user must be `admin`. Names are unique inside each list.

10.9 recoveries

Key	Default	Meaning
<code>dir</code>	<code>/var/lib/pgcellar/recoveries</code>	Where restores go: <code><dir>/<server>/<backup id>-<timestamp></code> . Absolute path, not <code>/</code> . Map it on the Docker host.
<code>max_download_bytes</code>	<code>2147483648</code> (2 GiB)	An archive above this has no download link, only its path. <code>0</code> = no limit.
<code>keep_days</code>	<code>7</code>	Restores older than this are removed daily. <code>0</code> = keep forever.

10.10 drills

Key	Default	Meaning
<code>enabled</code>	<code>true</code>	<code>false</code> stops the scheduled drills; manual ones stay possible.
<code>timeout</code>	<code>30m</code>	A whole drill, restore included.
<code>recovery_timeout</code>	<code>10m</code>	Wait for the copy to finish replaying WAL.
<code>keep_last</code>	<code>true</code>	Keep the data directory of the last good drill of each server.
<code>verify_tables</code>	<code>false</code>	<code>SELECT count(*)</code> on every table. Slow on big databases.
<code>max_age</code>	<code>192h</code> (8 days)	A server with a drill schedule turns orange when its last good drill is older.
<code>bin_dir</code>	<code>/usr/lib/postgresql</code>	Holds one directory per PostgreSQL major version, each with a <code>bin/</code> . Absolute path.

10.11 Environment variables

Variable	Used for
<code>TZ</code>	Time zone of schedules, the daily summary and displayed times.
The ones you name in <code>*_env</code>	Alert secrets (chapter 8).
<code>PGCELLAR_BASE_URL</code>	Only in <code>--demo</code> mode: base URL of links.

10.12 Validation errors

`pgcellar check` and the console say what is wrong in a sentence, for example:

```
PROBLEM  server "pg-main": set either recovery_window_days or redundancy, not both
PROBLEM  status_interval must be at least 30s (for example "1m" or "5m"); leave it out for the default
PROBLEM  there must be at least one user with the admin role
PROBLEM  alerts.daily_summary "8h30" is not a time of day (use "08:30", or leave it empty for no summary)
```

Fix the line and save the file again; pgCellar picks the change up.

Operations

Day-two work: updating, backing up pgCellar itself, watching it, and keeping an eye on the disk.

11.1 Updating the image

The state is in volumes and bind mounts, not in the image. Updating is a rebuild and a restart:

```
cd pgcellar/deploy
git pull
docker compose build
docker compose up -d
```

Compose recreates both containers (`pgcellar` and `pgcellar-ssh`) with the new image. Backups in progress are interrupted: a backup that was running is recorded as interrupted, and the scheduler catches up a missed due time if it is younger than `scheduler.catch_up` (6 hours by default). Choose a quiet moment.

After the start:

- `pgcellar version` inside the container, the startup log line and the metric `pgcellar_build_info` tell the version.
- The pages are full right away: the status cache is kept in SQLite and shown marked as stale ("as of 08:12, refreshing") until the first refresh completes.
- Alerts are silent for the first two minutes.
- `docker compose logs --tail 50 pgcellar` shows `pgcellar <version> listening on ...`.

If you run the binary without Docker, replace it and restart the service (`deploy/pgcellar.service` is a systemd unit that runs as the `barman` user).

Pin what you can. The image takes Barman from the PGDG repository at build time. Rebuilding on another day may bring a newer Barman. Read its release notes before a rebuild on a production backup host, and keep the previous image (`docker tag pgcellar:dev pgcellar:previous`) so you can go back.

Migrations of the SQLite database are applied automatically at start. Back up before updating anyway (below).

11.2 Backing up pgCellar itself

pgCellar is a control room, and its state is small. Back up three things, plus Barman's own data.

What	Where (in the container)	Why
Configuration	<code>/etc/pgcellar</code> (host: <code>etc-pgcellar/</code>)	<code>pgcellar.yaml</code> and its <code>.git</code> folder, the history of every change
Barman files	<code>/etc/barman.d</code> (host: <code>etc-barman.d/</code>), and <code>/etc/barman.conf</code> if you changed it	One file per server, and its own <code>.git</code> history
Database	<code>/var/lib/pgcellar/pgcellar.db</code>	Run history, audit log, alert journal, drill records. Not critical: pgCellar starts without it.
SSH keys and <code>.pgpass</code>	<code>/var/lib/barman/.ssh/</code> , <code>/var/lib/barman/.pgpass</code>	The key pair, the authorised WAL keys, the host key of the companion, the passwords. Secrets. Store them encrypted.
The backups	<code>/var/lib/barman</code> (volume <code>barman-data</code>)	The reason pgCellar exists

Notes:

- `pgcellar.yaml` and `etc-barman.d` are small text files with a git history. A copy of the folder is a complete backup of the configuration. Because every change is a commit, you can also push the repositories to a private remote.
- For a clean copy of the SQLite file, stop the container first (`docker compose stop pgcellar`), or copy `pgcellar.db` together with any `-wal` and `-shm` file next to it. If you lose it, you lose history, not backups.
- **Back up the backups too.** If the volume or the machine goes, the backups go with it. Copy `/var/lib/barman` to another machine (rsync, a snapshot of the volume, object storage is outside the scope of this version). A backup system on a single disk is a single disk.
- `~/.pgpass` and the private key are what let Barman talk to your servers. Treat them like any credential.

To restore pgCellar on a new host: install Docker, put back the configuration folders, the volumes (`.ssh`, `.pgpass`, `barman-data`), build the image, `docker compose up -d`. The status, the schedule and the servers are all back. Drill and run history come back only if you restored `pgcellar.db`.

11.3 Metrics

Prometheus text format on `/metrics`. A scrape never runs Barman: it reads the status cache.

Example scrape configuration:

```
scrape_configs:
  - job_name: pgcellar
    metrics_path: /metrics
    static_configs:
      - targets: ["backup.example.com:8080"]
```

`/metrics` and `/healthz` need no authentication. Do not publish them to the internet; restrict them in your proxy (chapter 12).

Per server (label `server`):

Metric	Meaning
<code>pgcellar_last_backup_age_seconds</code>	Seconds since the last successful backup ended
<code>pgcellar_last_wal_age_seconds</code>	Age of the last completed WAL file
<code>pgcellar_last_wal_received_age_seconds</code>	Age of the last data received by the WAL receiver
<code>pgcellar_wal_streaming_live</code>	1 if the Barman WAL receiver runs
<code>pgcellar_backups_total</code>	Number of backups
<code>pgcellar_check_ok</code>	1 if the cached <code>barman check</code> is fully OK
<code>pgcellar_setup_level</code>	Server setup: 0 ok, 1 warning, 2 error
<code>pgcellar_backup_bytes</code> , <code>pgcellar_wal_bytes</code> , <code>pgcellar_recoveries_bytes</code>	Space taken by backups, WAL and restores
<code>pgcellar_last_drill_ok</code>	1 or 0, absent until a drill ran
<code>pgcellar_last_drill_age_seconds</code>	Age of the last drill, whatever its result
<code>pgcellar_recoveries_total</code>	Successful restores

Global:

Metric	Meaning
<code>pgcellar_fs_free_bytes</code> , <code>pgcellar_fs_size_bytes</code> (label <code>path</code>)	File system holding backups and restores
<code>pgcellar_alerts_sent_total</code>	Messages sent, by channel and outcome
<code>pgcellar_license_servers_max</code> , <code>pgcellar_license_servers_used</code>	Licence quota and use
<code>pgcellar_build_info</code>	Version

Useful alerts in Prometheus: `pgcellar_last_backup_age_seconds > 90000`, `pgcellar_check_ok == 0`, `pgcellar_last_drill_ok == 0`, `pgcellar_setup_level == 2`.

`GET /healthz` answers `ok` when the database is reachable, `503` otherwise. Use it as the container health check.

11.4 Logs

pgCellar logs to standard output:

```
docker compose logs -f pgcellar
docker compose logs --since 1h pgcellar-ssh
```

You see the version at start, each failed alert send, config reload problems, and errors from API calls that ended in a 502. Barman's own log file is the one named by `log_file` in `barman.conf`. The SSH

companion logs at VERBOSE, so a refused key shows up there.

For the commands pgCellar ran, you do not need the logs: the **Runs** page keeps the output, and the audit log keeps the command line.

11.5 Audit

The **Audit** page (admin) and `GET /api/v1/audit` list every command pgCellar ran on behalf of a person or an API key: backup, check, switch-wal, restore, delete, psql, ssh, configuration writes. Each line has the time, the actor, the exact command, the exit code and the duration. Background commands (`barman cron` , the status refresh, actor `system`) are written only when they fail, and an identical failing command is logged at most once an hour, otherwise there would be a line per minute.

Audit lines older than `audit_retention_days` (90 by default) are purged daily. `0` keeps everything.

Every run goes through one function that writes this line. There is no other path from pgCellar to a command on your servers.

11.6 History

The **History** page shows the git log of the configuration folders, with the patch of each commit and the author. It answers "who changed the retention, and when". `git log -p` in `etc-pgcellar/` shows the same. A hand-written configuration with no commit gets one "Import the existing configuration" commit at the first start.

11.7 Storage and projection

The **Storage** block of a server page and the Status page tell what is on the disk and what is left. Everything is measured at each status refresh, from Barman's own files and the file system, never by running a command.

Per server:

- **Backups**: the sum of the sizes of the complete backups (on disk, with Barman's deduplication where it applies).
- **WAL**: the sum of the sizes in the WAL catalogue.
- **Restores**: what the restores of the server take under `recoveries.dir`.
- **Total**: the three added.

For the file system: free space, total, percent used, the path. The folder of the server is shown when it is a mount of its own, otherwise Barman's home. If `recoveries.dir` is on another device, it is shown too.

Thresholds. Under `alerts.min_free_percent` (10 by default) or `alerts.min_free_bytes` , every server on that file system turns orange; under half of the threshold, red. Alerts follow (chapter 8).

Projection. pgCellar computes how much is added per day: the WAL archived plus the backups finished in the last 7 days (backups at their size on disk), divided by the days covered. The window is cut to the age of the oldest data, between 1 and 7 days, so a young server is not divided by 7. When the file system would be full in under 90 days, the page says "At this pace the disk is full in N days".

The projection is a ceiling for a server in steady state. It does not predict that retention will delete old backups, nor that compression will change. Read it as "if nothing changes, and nothing is deleted".

Not counted: Barman's `streaming` and `incoming` directories and files kept outside `base/` and `wals/`.

11.8 A short weekly routine

- Open Status. Everything green? Done.
- Look at the last drill of each server. Did it pass this week?
- Look at the Storage line: how many days left?
- Once a quarter, restore something by hand and read the result. A drill is a good guard; it is not you.

Security model

pgCellar holds the keys to your backups, and a backup is a complete copy of your data. This chapter says what it does to protect them, and what you must do around it.

12.1 Principles

- **Least surprise.** pgCellar never writes to a production PostgreSQL server. It reads `pg_settings`, and Barman runs the backup. It tells you what to change; you change it.
- **One door to commands.** Every command pgCellar runs on a server goes through one function that logs the exact command, its exit code and its duration to the audit log. There is no second code path, not even for `psql`.
- **No secret in files that travel.** Not in `pgcellar.yaml`, not in git, not in the API.
- **Small surface.** Server-rendered HTML, a few kilobytes of vendored JavaScript for HTMX, no framework, no build step. A system administrator can read the code that matters.

12.2 Roles

Three roles: viewer, operator, admin (chapter 9). The check happens on the server, on every route, for the console and the API. The buttons you do not have are simply absent.

The audit log is admin-only because it contains exact command lines. Downloading a restore archive is operator-only: it is a full copy of the database.

12.3 Passwords and sessions

- Passwords are stored as bcrypt hashes in `pgcellar.yaml`. Minimum length at hash time: 8 characters.
- After sign-in the browser holds a random token in a cookie. The database stores only the SHA-256 of the token, so a copy of the database cannot be replayed as a session.
- The cookie is `HttpOnly` and `SameSite=Lax`, with `Secure` set when the request arrives over HTTPS. A session lasts 24 hours.
- Every state-changing request carries a CSRF token, derived with HMAC from the session id and a secret kept in the database. The sign-in form uses a stateless signed token valid for an hour.
- HTML pages are sent with `Cache-Control: no-store`, `X-Frame-Options: DENY`, `X-Content-Type-Options: nosniff`, `Referrer-Policy: same-origin` and a Content Security Policy that allows only the site itself (`default-src 'self'`, `form-action 'self'`, `frame-ancestors 'none'`).
- Static files are versioned by content (`app.css?v=<hash>`) and served `immutable`; the HTML is never cached. After an update, a browser never mixes new HTML with old CSS.

12.4 Sign-in limiter

Guessing passwords is slowed down in two ways, for every mode:

- At most **10 sign-in attempts a minute** per address, successful or not. Above that: HTTP 429 with `Retry-After`.
- At most **8 failures in 10 minutes** per address.

The address is the connection's. If pgCellar runs behind a reverse proxy, the proxy's address would be the same for everybody, so pgCellar believes the `X-Real-IP` header **only** when the connection comes from a loopback or private address (the proxy). A stranger on the internet cannot choose his own address. Make sure your proxy sets `X-Real-IP` and does not pass the client's own value through.

12.5 API keys

An API key is 32 random bytes, shown once. Only the SHA-256 is stored. The comparison is against the hash. Keys carry a role, are revoked by deleting their entry, and every call made with one is audited under the key's name.

12.6 Secrets: where they live

Secret	Lives in	Never in
PostgreSQL passwords	<code>~/.pgpass</code> of the barman user (mode 0600)	<code>pgcellar.yaml</code> , git, a command line
SSH private key	<code>~barman/.ssh/</code> (mode 0600)	The database, git, the API, the audit log
Alert tokens, SMTP login, webhook secret	Environment variables of the container, named by <code>*_env</code>	<code>pgcellar.yaml</code> , git
Password hashes, API key hashes	<code>pgcellar.yaml</code>	(only hashes, never the secret)
Session tokens	Browser cookie; SHA-256 in the database	
Licence key	<code>pgcellar.yaml</code> or a file	(it is not a secret key; it is signed and verified offline)

A `password=` in a connection string is removed before `psql` runs, so it cannot show up in `ps`. The Telegram token is removed from error texts. A secret printed once (a new API key) is printed to your terminal and nowhere else.

12.7 SSH, restricted

The SSH companion that receives WAL files:

- runs as the `barman` user, not as root; `PermitRootLogin no`, `AllowUsers barman`, `UsePAM no`;
- accepts keys only (no password, no keyboard-interactive);
- gives no terminal, no X11, no TCP or agent forwarding, no sftp;
- writes each authorised key with `restrict,command="barman put-wal <server>"`, so a stolen key of a database host pushes WAL files of its own server and can do nothing else;
- keeps its host key in the volume.

Even so: do not publish port 2222 on the internet. Bind it to a private address, or a VPN.

For the rsync method, pgCellar's own key pair lets the barman user log into your PostgreSQL hosts. Give it the account you would give Barman by hand, nothing more.

12.8 Container

- The container runs as the `barman` user, uid 997 in the current image. There is no root process, no `sudo` binary and no SSH daemon in the application container.
- Bind-mounted folders must be owned by that uid. A volume created by an older root container must be fixed with `chown` (chapter 13).
- Only port 8080 (and 2222 on the companion) are exposed. You can go further with `cap_drop: [ALL]`, `security_opt: [no-new-privileges:true]`, memory and process limits; the demo's Compose file in `deploy/demo/` does exactly that.

12.9 Network

pgCellar speaks plain HTTP. Put it behind a reverse proxy that terminates TLS, and:

- restrict who can reach it: a VPN, an allow-list, or single sign-on in the proxy, in addition to pgCellar's own sign-in;
- block `/metrics` and `/healthz` from the internet. They need no authentication (they expose server names and ages, nothing more);
- let the proxy set `X-Real-IP`, as above, and pass `X-Forwarded-Proto` so the cookie gets the `Secure` flag.

A sketch for nginx:

```
location / {
    proxy_pass          http://127.0.0.1:8080;
    proxy_set_header    Host $host;
    proxy_set_header     X-Real-IP $remote_addr;
    proxy_set_header     X-Forwarded-Proto $scheme;
}
location = /metrics { allow 10.0.0.0/8; deny all; proxy_pass http://127.0.0.1:8080; }
```

12.10 Demo mode

`pgcellar serve --demo` is a public, read-only instance on fictional data (Northwind Logistics, five servers). It exists so that a visitor can click through the product without being one bug away from real backups.

- No configuration file is read. The Barman, `psql`, `pg_ctl` and `ssh` it runs are fakes embedded in the binary, extracted to a temporary directory at start. If they cannot run, pgCellar refuses to start rather than call the real ones.
- Every `POST`, `PUT`, `PATCH` and `DELETE`, except sign-in and sign-out, answers 403 before routing, before authentication and before CSRF. So do the two `GET` routes that start a command or hand out an archive. A test requests every mutating route registered in the code, so a new route cannot be forgotten.
- The user `demo` has the viewer role in effect and no API key. The scheduler and the alert loop are off. Nothing is sent anywhere.
- Disabled buttons in the page are cosmetic. The server is what refuses.
- Never put real data in a demo instance. Run it as a separate container, read-only, with no volume of your own (`deploy/demo/`).

12.11 Your side of the bargain

- Use a strong admin password, and an `operator` or `viewer` account for daily work.
- Keep the `pgcellar-data` and `barman-data` volumes, and the backups, on encrypted storage if your data needs it. pgCellar does not encrypt backups.
- Copy the backups to a second place that a compromise of this one cannot reach.
- Restore drills prove that backups restore. They do not protect against someone deleting them. Limit who is `admin`.

Troubleshooting

Each entry has the symptom you see, the cause, and the fix. They come from real installations, in the order people hit them. Start with the **Runs** page and the output of the failed run: pgCellar keeps every command with its answer.

13.1 Setup and sign-in

"attempt to write a readonly database" in the logs

Cause. A volume or folder was created by a container that ran as root (an older image, or Docker creating a bind mount). pgCellar now runs as the `barman` user and cannot write to it.

Fix. Give the folders to the barman uid, then restart:

```
docker run --rm --entrypoint id pgcellar:dev barman      # prints the uid, 997 in the current image
sudo chown -R 997:997 etc-pgcellar etc-barman.d /path/to/your/data /path/to/recoveries
docker compose restart pgcellar
```

For named volumes: `docker run --rm -v pgcellar_pgcellar-data:/d --user root --entrypoint chown pgcellar:dev -R 997:997 /d.`

"sudo: executable file not found" on every `barman cron`

Cause. An older build ran pgCellar as root in a way that made it prefix commands with `sudo -u barman`. The image has no sudo.

Fix. Use the current image. pgCellar only prefixes `sudo -u barman` when the current user is not already the barman user.

Sign-in answers 429

Cause. More than 10 attempts a minute, or 8 failures in 10 minutes, from one address. Behind a proxy, everyone may share the proxy's address if `X-Real-IP` is not passed.

Fix. Wait a minute. Behind a proxy, pass `X-Real-IP $remote_addr` (chapter 12).

The page looks broken after an update (no styles, odd layout), mostly on Safari

Cause. The browser kept the old stylesheet with the new HTML. Safari is the usual victim: an earlier build served `app.css` with `max-age=3600`.

Fix. Current builds link the stylesheet as `/static/app.css?v=<hash of its content>`, so a change is always a new URL. If you see it, reload without the cache (Cmd-Shift-R) once. If a reverse proxy sits in front, make sure it does not add its own caching to HTML.

After `docker compose up`, the proxy answers 502

Cause. A reverse proxy resolved the container's IP address once and keeps it. Compose gave the recreated container a new address.

Fix. Reload the proxy (`nginx -s reload`). For a lasting fix, let nginx re-resolve: use the Docker resolver `resolver 127.0.0.11 valid=10s;` and a variable in `proxy_pass` (`set $up http://pgcellar:8080;` `proxy_pass $up;`).

13.2 The server page

Everything is red in the first minute after a start

Cause. Barman has not started its WAL receivers yet. `barman cron` starts them.

Fix. Nothing to do. pgCellar runs `barman cron` once before the first refresh, then every minute, and stays silent on alerts for two minutes. If it stays red, read the failed check.

"first refresh pending" or HTTP 503 with `Retry-After`

Cause. The status cache has no data for the server yet (just added, or just started).

Fix. Wait up to `status_interval`.

"Could not read the server settings"

Cause. The `psql` query behind Server setup failed: the server is unreachable from the pgCellar host, or the barman user's `~/.pgpass` has no line for this connection.

Fix. Run the test from chapter 3 (`psql "host=... user=barman dbname=postgres" -c "select 1"`) and read the error. The line in `.pgpass` is `host:port:database:user:password`, mode 0600, host and port as in the `conninfo`. It does not change the health of the server: Barman's own checks say whether the server is reachable.

`archive_command` shows "hidden" or `<insufficient privilege>`

Cause. The role pgCellar connects with may not read `archive_command`.

Fix. `GRANT pg_read_all_settings TO barman;`

"no `pg_hba.conf` entry for host ..." for a relay or proxy

Cause. PostgreSQL sees the address of the relay (HAProxy, a Docker network gateway), not the address of the pgCellar host.

Fix. Read the address in the PostgreSQL log line and use that address or its subnet in `pg_hba.conf`. A `host replication` line is separate from `host all`; you need both.

The Runs page is empty although backups exist

Cause. Runs lists only what was started from pgCellar. Backups taken with the `barman` command or from a host cron appear in the Backups list of the server page, which comes from Barman.

Fix. Nothing to fix. Let the built-in scheduler run the backups and remove the host cron.

13.3 Backups

`barman backup --wait` hangs for a quarter of an hour on a quiet database

Cause. The end of a backup sits in a WAL segment that is still being written. On a quiet database that segment fills (16 MB) rarely, so `--wait` blocks until a switch happens. One real case: 11 seconds of copy, 15 minutes of waiting.

Fix. Do not use `--wait`. Run `barman backup <server>`, then `barman switch-wal --force --archive <server>`. pgCellar does exactly that. If you take backups by hand, do the same; or press **Backup now**.

"Last filled WAL file" is hours old although streaming is live

Cause. On a quiet database a WAL file fills slowly. Streaming data still arrives in a partial file, so you are not at risk, but the *completed* file is old.

Fix. It is a note, not an alert, while streaming is live (the server page shows "Streaming: live" and "last data"). For a bounded recovery point also for the archive path, set `archive_timeout = '1h'` on PostgreSQL (chapter 3). Tune `alerts.max_wal_age` if you want a tighter or looser limit.

Warning: `archive_timeout` is 0

Cause. A quiet database never closes a WAL file by itself, so with `archive_command` the recovery point can be very old.

Fix. `ALTER SYSTEM SET archive_timeout = '1h'; SELECT pg_reload_conf();`. A warning never turns the server orange.

Warning: `max_slot_wal_keep_size` is -1

Cause. With a replication slot and no limit, PostgreSQL keeps all WAL Barman has not received. If the backup host is down for long, the PostgreSQL disk fills.

Fix. `ALTER SYSTEM SET max_slot_wal_keep_size = '2GB'; SELECT pg_reload_conf();` (or more, sized for your longest outage). If the limit is exceeded, the slot is invalidated and the WAL chain is broken: take a new backup.

Warning: `data_checksums` is off

Cause. The cluster was created without checksums. A damaged page goes unnoticed, in the backup as well, and a drill cannot tell.

Fix. It is not a setting. New clusters: `initdb --data-checksums`. Existing clusters: `pg_checksums --enable -D <data directory>` on a **stopped** cluster, then start it. On a big cluster this takes a while; plan it.

A hole in the WAL on the Timeline

Cause. A segment is missing between two files of the same timeline: the slot was dropped or invalidated, `archive_command` failed for a while, or WAL was removed by hand.

Fix. The backup before the hole cannot be used for a target after the hole. Fix the cause, then take a new backup; the chain restarts from it. The earliest restore point moves forward accordingly.

A server listed in `pgcellar.yaml` but missing in `barman.d`

Cause. `pgcellar.yaml` lists a server that has no `[section]` in `barman.d`, or the file does not validate.

Fix. `pgcellar check --config ...` names each problem. The server form refuses an invalid combination with the reason.

13.4 Restores and drills

The drill fails at "Find the PostgreSQL version": "PostgreSQL N binaries are not in this image"

Cause. The restored directory is from a major version that the image does not hold. The image ships binaries for the versions in `PG_MAJORS` (default `16 17 18`).

Fix. Rebuild with the version added: `--build-arg PG_MAJORS="15 16 17 18"`. Check your servers with `SHOW server_version;`.

The drill fails at "Start PostgreSQL": "The cluster uses locale X, which this image does not have"

Cause. A restored cluster keeps the locale of its source (for example `en_US.utf8` for a `postgres` container). PostgreSQL refuses to start on a locale the system lacks.

Fix. Add the locale to the build argument `PG_LOCALES` and rebuild (chapter 7). Find it with `SELECT datcollate, datatype FROM pg_database;`. The run output also contains the last 40 lines of the copy's log.

The demo refuses to start: "not executable (noexec mount?)"

Cause. A `tmpfs` mounted without `exec`. Docker mounts a `tmpfs` as `noexec` by default, and programs written there cannot start. Demo mode extracts its fake `barman`, `psql` and the rest to a temporary directory and refuses to start if they cannot run, rather than fall back to the real ones.

Fix. Add `exec` to the mount: `tmpfs: ["/tmp:size=64m,mode=1777,exec"]`.

A drill fails at "Wait for the end of the recovery"

Cause. The copy has too much WAL to replay inside `drills.recovery_timeout`, or the WAL it needs is missing.

Fix. Raise `drills.recovery_timeout` (and `drills.timeout`). If WAL is missing, look at the Timeline for a hole.

A drill fails at the checksums step

Cause. `pg_checksums --check` found a damaged page. This is exactly what the step is for.

Fix. Treat it as a real problem: the page is damaged in the backup, and probably on the source. Look at the PostgreSQL log, check the disk of the source, and take a new backup after the source is repaired.

The drills never run

Cause. The server has no `drills.schedule`, or `drills.enabled: false` is in the file. A leftover `enabled: false` is the usual one (see the rule "an absent key is the default" in chapter 10).

Fix. Set `servers[].drills.schedule` (for example `weekly sun 04:30`) and delete the `enabled: false` line. `pgcellar check` warns about it.

Restore says "the target is before the end of the backup"

Cause. Barman refuses a target time earlier than the end of the chosen backup.

Fix. Pick an older backup, or a later time. The Timeline shows the earliest point you can reach.

"Permission denied" writing a restore, or the restore folder is empty on the host

Cause. `recoveries.dir` is inside the container and the folder you mapped is not owned by the barman uid, or it is not mapped at all and the data lives in an anonymous volume.

Fix. Map a host folder with a bind mount, and `chown` it to the barman uid (chapter 6).

13.5 WAL shipping (`archive_command`)

`barman-wal-archive` fails on the PostgreSQL host

Check, in this order.

1. The package `barman-cli` is installed on the PostgreSQL host.
2. The command is exactly the advised one, with `--port 2222` and the right host name. Set `ssh_host` and `ssh_port` in `pgcellar.yaml`.
3. The public key of the `postgres` user is authorised for this server on the **SSH key** page.
4. The `postgres` user has accepted the host key: `sudo -u postgres ssh -p 2222 barman@<host> true` once.
5. The `pgcellar-ssh` container is running and the port is reachable. Its log (`docker compose logs pgcellar-ssh`) says why a key was refused.

`barman-wal-archive --test` fails with a restricted key; that is expected. Use **Switch WAL** and **Run check now**.

"WAL archive" check stays red after a switch

Cause. No WAL file arrived. Often `archive_mode` is off (needs a restart) or `archive_command` was not reloaded.

Fix. Server setup shows both with the exact commands. After fixing them, press **Switch WAL** and **Run check now**.

13.6 Still stuck

Collect, in this order, before you ask for help:

- the failed run on the Runs page (command, exit code, output);
- `docker compose logs --tail 100 pgcellar`;
- `pgcellar check --config /etc/pgcellar/pgcellar.yaml`;
- `pgcellar version`;
- the Server setup section of the server.

Never paste a `.pgpass`, a key, a token or a licence key into a ticket.

Licensing and editions

Two separate things: the licence of the **source code** (what you may do with pgCellar), and the **licence key** (how many servers you may add). This chapter explains both. It is a summary: the text of the file `LICENSE` is what counts.

14.1 The source licence: FSL-1.1-Apache-2.0

pgCellar is released under the Functional Source License, version 1.1, with the Apache License 2.0 as its future licence (SPDX: `FSL-1.1-ALv2`). In plain words:

- The source is public. You can read it, build it, modify it and redistribute it.
- You can use it for any purpose except a **Competing Use**. Using it to back up your own databases, or your employer's, or your customers' as part of a professional service, is a permitted purpose. So are non-commercial education and research.
- A Competing Use is making the software available to others in a commercial product or service that substitutes for pgCellar or offers the same or substantially similar functionality. In practice: you cannot take pgCellar and sell a hosted "backup console" service built on it.
- **Each version becomes Apache-2.0 two years after its release.** After that date, that version is under a permissive licence with no such restriction.

If you are in doubt about a specific use (a managed-service offering, an integration in a product), ask before you build it: the author prefers a conversation to a surprise.

Barman is a separate program under GPL-3.0. The image contains it unmodified; pgCellar runs its command line and never imports, links or patches it. The two programs keep their own licences. The Docker image is an aggregation of them.

14.2 Editions

Edition	Servers	Price
Free	up to 2	0
Pro	up to 10	490 EUR per year
Business	unlimited	1 490 EUR per year

Prices are the ones on the project page at the time of writing; see <https://dbfactory.io/en/pgcellar/> for the current ones. Early access is free during the beta.

Free is not a trial. There is no expiry and no feature locked away: the Free edition simply holds two servers.

14.3 What a licence blocks, and what it never blocks

A licence never blocks a backup, a restore or a drill. That is a product rule, in the code, tested. Whatever the state of your licence:

- scheduled and manual backups keep running;
- restores, point-in-time recovery and downloads keep working;
- restore drills keep running and reports keep being written;
- alerts, the API, metrics and the console keep working.

What a licence limits is **adding a server** beyond the quota. The check is made in one place only, where a server is created, so the form and the API behave the same. Beyond the quota the answer is a message that says what keeps working and where to get a licence.

A server that already exists is never removed or paused because of the licence.

Expiry and grace

State	Meaning	Quota
free	No key	2 servers
valid	Signed key, not expired	What the key says
grace	Expired for less than 30 days	Still what the key says
expired	Expired for 30 days or more	Free rules: 2 servers, for new additions
invalid	The key does not verify	Free rules, and the reason in a banner

A key is valid through the end of its expiry day (UTC). After expiry you get 30 days of grace. After that, the Free rules apply to **additions** only; the servers you already have keep running and nothing is deleted.

A key of edition "free" never raises the quota.

The footer of every page shows the edition and the count ("Free edition · 2 of 2 servers"). The Status page shows a banner only when you are over quota, in grace, or expired.

14.4 Putting a key in place

A key looks like `PGC1.<payload>.<signature>`. It is Ed25519 signed and verified **offline** against a public key built into the binary: no account, no network call, nothing phones home.

Put it in `pgcellar.yaml`:

```
license:
  key: "PGC1.eyJpZCI6...."
```

or keep it in a file and point to it:

```
license:
  file: /etc/pgcellar/license.key
```

Check what the application sees:

```
docker compose exec pgcellar pgcellar license show --config /etc/pgcellar/pgcellar.yaml
```

```
state:    valid
edition:  Pro
licensee: Example Ltd
id:       ...
servers:  10 allowed
expires:  2027-10-20 (grace until 2027-11-19)
footer:   Pro edition · Example Ltd · 3 of 10 servers
```

To read what a key says, without a configuration file:

```
pgcellar license verify PGC1.eyJ...
```

It prints the payload (id, edition, licensee, servers, issue and expiry dates), the state, and a verdict: `signature OK`, or `INVALID` with the reason. It exits with a non-zero status for an expired key.

`GET /api/v1/license` (viewer) returns the edition, the licensee, the servers allowed and used, the expiry and the state. The metrics `pgcellar_license_servers_max` and `pgcellar_license_servers_used` expose the quota.

`pgcellar license keygen` and `pgcellar license issue` are for the vendor: they create the signing key and sign licence keys. You never need them. The private key is never in the repository.

Get a key at <https://dbfactory.io/en/pgcellar/>.

14.5 Which servers are counted

Servers are counted as the Servers page lists them: the sections of `barman.d`, the entries of `pgcellar.yaml` and the servers in the status cache, each name once. `pgcellar license show` counts only the entries of `pgcellar.yaml`, so its footer line can differ from the one in the console.

CLI reference

One binary, `pgcellar`, with a few subcommands. In the Docker image it is the entry point, so the arguments go straight after the image name or the service name:

```
docker compose run --rm pgcellar check --config /etc/pgcellar/pgcellar.yaml
docker compose exec pgcellar pgcellar apikey list --config /etc/pgcellar/pgcellar.yaml
docker run -i --rm pgcellar:dev hash-password
```

`run` starts a new throwaway container; `exec` runs in the one that is up. The default command of the image is `serve --config /etc/pgcellar/pgcellar.yaml`.

```
pgcellar serve ...
pgcellar check ...
pgcellar apikey create|list ...
pgcellar hash-password
pgcellar license keygen|issue|verify|show ...
pgcellar version
pgcellar help
```

Exit codes: `0` success, `1` the command ran and failed (or `check` found problems), `2` wrong usage (a missing or unknown argument).

15.1 serve

Runs the web console, the API, the scheduler, the alert loop and the background refreshes.

```
pgcellar serve --config FILE [--listen ADDR] [--cron-interval DURATION] [--dev]
pgcellar serve --demo [--listen ADDR]
```

Flag	Default	Meaning
<code>--config</code>	<code>/etc/pgcellar/pgcellar.yaml</code>	Path of <code>pgcellar.yaml</code> .
<code>--listen</code>	<code>listen:</code> of the file	Address to listen on, for example <code>127.0.0.1:8080</code> or <code>:8080</code> . Overrides the file.
<code>--cron-interval</code>	<code>barman.cron_interval</code> of the file	How often to run <code>barman cron</code> , for example <code>30s</code> . <code>0</code> disables it.
<code>--dev</code>	off	Development mode: a fake Barman, temporary data, the clock frozen at the fixtures time, login <code>admin</code> / <code>admin</code> . Works on a temporary copy of the configuration. Never use it on real data.
<code>--demo</code>	off	Public demonstration: fictional servers and data, read-only, login <code>demo</code> / <code>demo</code> , no configuration file. Cannot be combined with <code>--dev</code> . Links use <code>PGCELLAR_BASE_URL</code> (default <code>https://demo.pgcellar.dbfactory.io</code>).

On start it prints the version and the address:

```
pgcellar 0.1.0 listening on http://localhost:8080
```

`SIGINT` and `SIGTERM` stop it cleanly: the HTTP server gets 10 seconds to finish its requests.

15.2 check

Validates `pgcellar.yaml` and the Barman server files, and lists the servers.

```
pgcellar check --config FILE [--barman-dir DIR]
```

Flag	Default	Meaning
<code>--config</code>	<code>/etc/pgcellar/pgcellar.yaml</code>	The configuration to validate.
<code>--barman-dir</code>	<code>barman.config_dir</code> of the file	Read the Barman files from this folder instead.

Typical output:

```
configuration deploy/pgcellar.example.yaml
ok

servers in deploy/barman.d.example: 2
ok      pg-main      postgres      pg-main.conf
ok      pg-legacy     rsync        pg-legacy.conf

all good
```

A problem line starts with `PROBLEM`, a warning with `WARNING` (for example `drills.enabled: false` with no drill schedule). The command lists a server that `pgcellar.yaml` mentions and `barman.d` does not. Exit code 1 if anything is a problem; warnings alone do not fail it.

If the `barman.d` folder named in the file does not exist (a copy of the example, run on a laptop), `check` falls back to a `barman.d.example` folder next to the configuration and says so.

`check` does not talk to PostgreSQL or Barman. It checks files only.

15.3 apikey

Manages API keys. Keys live in `pgcellar.yaml` as a SHA-256 digest; a running server picks up a change by itself.

```
pgcellar apikey create --config FILE --name NAME --role viewer|operator|admin
pgcellar apikey list  --config FILE
```

`create` prints the new key (`pgc_...`) on standard output, once, and nothing else. It validates the configuration, writes the digest into `pgcellar.yaml`, and records the change in the configuration's

git history under the name `cli`. `list` prints one line per key, `name<TAB>role`.

Both flags `--name` and `--role` are required for `create`.

15.4 hash-password

Reads one line (the password) from standard input and prints its bcrypt hash, for the `users:` list.

```
read -rs PW; printf '%s\n' "$PW" | pgcellar hash-password
```

The password needs at least 8 characters; at most 1024 bytes are read. A password on the command line would end up in your shell history, so the command takes none. With Docker, give `-i` so that standard input stays open.

15.5 license

```
pgcellar license keygen --out DIR
pgcellar license issue --key FILE --edition free|pro|business --licensee NAME \
                      --servers N --expires YYYY-MM-DD [--issued YYYY-MM-DD] [--id ID]
pgcellar license verify KEY
pgcellar license show --config FILE
```

`show` and `verify` are for everybody; `keygen` and `issue` are for the vendor (they need the private signing key, which is never in the repository).

Subcommand	What it does
<code>show</code>	Reads the key from <code>license.key</code> or <code>license.file</code> in the configuration and prints state, edition, licensee, id, servers allowed, expiry and grace end, and the footer line. No key means the Free edition.
<code>verify KEY</code>	Checks the signature of a key against the public key built into the binary and prints its payload and state. Exit code 1 for an expired key. A bad signature prints <code>verdict: INVALID</code> with the reason.
<code>keygen --out DIR</code>	Makes an Ed25519 signing key pair in <code>DIR</code> (<code>license_signing.key</code> , mode 0600, and its <code>.pub</code>). Refuses to overwrite an existing private key.
<code>issue ...</code>	Signs a licence key. <code>--servers 0</code> means unlimited. <code>--id</code> defaults to a random <code>lic-xxxxxxxxxx</code> . <code>--issued</code> defaults to today (UTC).

Chapter 14 explains editions and the grace period.

15.6 version

```
$ pgcellar version
pgcellar 0.1.0
```

`--version` and `-v` do the same. The version is set at build time (`-X main.Version=...`); a plain `go build` prints `dev`. The Docker image takes it from the build argument `VERSION`.

15.7 help

`pgcellar help`, `--help` and `-h` print the usage text and exit with 0. A missing or unknown subcommand prints it to standard error and exits with 2.

15.8 Build from source

```
make build      # ./pgcellar
make test       # go test -race ./...
make lint       # gofmt, go vet, golangci-lint
make run        # the --dev mode, on the example configuration
make demo       # the --demo mode on 127.0.0.1:8080
make docker     # build the image as pgcellar:dev
make release-dry # linux amd64/arm64 and macOS arm64 archives with checksums in dist/
make docs       # this manual, as HTML, in site/docs/
make docs-pdf   # this manual, as site/docs/pgcellar-manual.pdf
```

Go 1.25 or later. Barman is not needed to run the tests: a shell script that mimics its command line stands in for it.