Containers: Field Guide

Gotchas, Pitfalls and Troubleshooting

ESIR 2 SI — Architectures Logicielles — companion to lectures 1–3 and the appendix

Walter Rudametkin Walter.Rudametkin@univ-rennes.fr https://rudametw.github.io/teaching

What this is. You will deploy self-hosted services (reverse proxy + application + database) on a **remote university VM**, over SSH, with **rootless Podman**. The lectures explain the concepts; this guide collects what goes wrong in practice and how to fix it. Read §1 when something breaks; read the rest before you deploy.

Conventions. Commands are for rootless Podman 5 (checked with Podman 5.8); `docker` accepts almost the same commands. Run them as **your user**, without `sudo`, unless marked *admin* (needs root: ask the VM administrator). *(to verify)* marks what could not be tested on the VM image. Replace `<ctr>` (container name), `<unit>`, `<net>`… with your own names. Images are always written with their full name (`docker.io/library/postgres:16-alpine`) and a pinned tag.

What	Where (rootless)
Images, containers, volumes	`~/.local/share/containers/storage/` (volumes: `volumes/<name>/_data`)
Your Podman configuration	`~/.config/containers/` (`containers.conf`, `registries.conf`, `storage.conf`)
Quadlet files (you write)	`~/.config/containers/systemd/`
Generated units (do not edit)	`/run/user/$UID/systemd/generator/`
Your subordinate UID/GID ranges	`/etc/subuid`, `/etc/subgid`
Linger markers	`/var/lib/systemd/linger/<user>`
Your own systemd units (timers…)	`~/.config/systemd/user/`

1 Troubleshooting checklist . . . 1
2 Rootless gotchas . . . 3
3 Compose pitfalls . . . 4
4 Secrets . . . 5
5 From Compose to Quadlets: the capstone . . . 6
6 HTTPS with Caddy on the VM . . . 8
7 Backups . . . 9
8 Operating it . . . 10
9 Supply-chain checklist . . . 10

1 Troubleshooting checklist

Start with these five commands: most problems are visible in their output.

```
podman ps -a                        # running? exit code? (healthy)/(unhealthy)?
podman logs --tail 50 <ctr>         # what did PID 1 print before dying?
podman inspect <ctr> --format '{{.State.Status}} exit={{.State.ExitCode}} oom={{.State.OOMKilled}}'
podman port <ctr>                   # which ports are published, on which address?
systemctl --user status <unit>; journalctl --user -u <unit> -n 50 --no-pager   # Quadlets
```

Symptom	Check	Fix
Exits at once (`Exited (1)`)	`podman ps -a`, then `podman logs <ctr>`. Look inside the image: `podman run --rm -it --entrypoint sh <image>` (no shell in distroless images).	Read the error: missing variable (`POSTGRES_PASSWORD`), typo in a config file, wrong command. A container lives as long as its PID 1: a program that "daemonizes" (goes to the background) makes the container exit; run it in the foreground. Exit 127: command not found (wrong path, no `sh` for a *shell form* `CMD`); 126: not executable.

Symptom	Check	Fix
Exit code 137, restarts in a loop	`podman inspect <ctr> --format '{{.State.OOMKilled}}'`, `podman stats --no-stream`	137 = 128 + 9 (`SIGKILL`). `true`: the OOM killer hit the memory limit: raise `mem_limit` / `Memory=`. `false`: someone killed it, e.g. `podman stop` after 10 s because the app ignores `SIGTERM` (143 = 128 + 15 is a clean `SIGTERM`).
Connection refused on the VM (`curl 127.0.0.1:8080`)	`podman port <ctr>`; `ss -tlnp | grep 8080`	Nothing published: add `-p` / `ports:` / `PublishPort=` (`EXPOSE` is only metadata). Published but still refused: the app listens on `127.0.0.1` *inside* the container; it must listen on `0.0.0.0` there. Wrong container port: check `ExposedPorts` with `podman image inspect`.
Works on the VM, not from my laptop	`ss -tlnp` shows `127.0.0.1:8080`	By design: loopback is not reachable from outside. For yourself, use an SSH tunnel: `ssh -L 8080:127.0.0.1:8080 <user>@<vm>` then open `http://localhost:8080`. For everyone: publish on `0.0.0.0` *and* open the firewall *admin* (`firewall-cmd --add-port=8080/tcp` or `ufw allow 8080/tcp`); the campus firewall may still block it.
Name does not resolve (`could not translate host name "postgres", no such host`)	`podman inspect <ctr> --format '{{json .NetworkSettings.Networks}}'`; `podman network inspect <net> --format '{{.DNSEnabled}}'`; `podman exec <ctr> getent hosts postgres`	Both containers must share a **user-defined** network (`podman network create`, Compose `networks:`, Quadlet `.network`). The default `podman` network has **no DNS**. The name is the container name or a network alias (Compose: the service name; Quadlet: `ContainerName=` or `NetworkAlias=`). Quadlet networks are called `systemd-<file>` unless `NetworkName=` is set.
Permission denied on a bind mount	`ls -ln <dir>` (host view) vs `podman unshare ls -ln <dir>` (container view); `podman top <ctr> user huser`; SELinux: `ausearch -m avc -ts recent` *admin*	See §2.1: `:U` (chown to the container user) or `podman unshare chown 999:999 <dir>`; `:Z` for SELinux; or use a **named volume** instead.
Everything dies when I log out	`loginctl show-user $USER -p Linger`	`loginctl enable-linger $USER` (§2.3).
Not back after a reboot	`systemctl --user list-units 'podman*' '*.service'`; `journalctl --user -b -u <unit>`	Quadlet: `[Install] WantedBy=default.target`, then `systemctl --user daemon-reload`. Compose: `restart: unless-stopped` *and* `systemctl --user enable podman-restart.service`. Both need linger.
`systemctl --user start x` says `Unit x.service not found`	`/usr/libexec/podman/quadlet -dryrun -user` (Debian/Ubuntu: `/usr/lib/podman/quadlet`); `podman quadlet list`	The generator skipped the file: unknown key, key in the wrong section, a key from a newer Podman, wrong directory or extension. `caddy.container` becomes `caddy.service`; `web.network` becomes `web-network.service`.
`Failed to connect to bus` (with `systemctl --user`)	`echo $XDG_RUNTIME_DIR` (should be `/run/user/<uid>`)	You switched user with `su` or `sudo -u`: log in directly over SSH as that user (or `machinectl shell <user>@` *admin*).
Health check stays `unhealthy` / `starting`	`podman healthcheck run <ctr>`; `podman inspect <ctr> --format '{{json .State.Health}}'`	The check runs *inside* the container: `curl`/`wget` are often missing (use the app's own probe, e.g. `pg_isready`); increase the start period for slow first starts (`initdb`).
Changes to `compose.yaml` / `.env` ignored	`podman compose config` (shows the resolved file)	Environment is fixed when the container is *created*: `podman compose up -d` re-creates changed services, `restart` does not. Quadlets: `daemon-reload` then `restart`.
PostgreSQL ignores the new password	`podman volume ls`	`POSTGRES_PASSWORD`, `POSTGRES_USER`, `POSTGRES_DB` are only used to *initialize an empty* data directory. Change it in SQL (`ALTER USER miniflux PASSWORD '...'`) or start from an empty volume (data lost).
"My data is gone"	`podman volume ls` (several `*pgdata` volumes?)	Compose prefixes volumes with the **project name** = directory name: renaming or moving the directory, or switching to Quadlets, creates a *new, empty* volume. Point to the old one (`VolumeName=selfhosted-stack_pgdata`) or restore a backup.
`address already in use`, `name "..." is already in use`	`ss -tlnp | grep :8080`; `podman ps -a --format '{{.Names}} {{.Ports}}'`	An old stack still runs (`podman compose down`, `systemctl --user stop ...`) or another program uses the port. A stopped container keeps its name: `podman rm <name>`.

Symptom	Check	Fix
Pull asks "Please select an image", fails (`short-name resolution enforced`, `toomanyrequests`)	—	Use the full name: `docker.io/library/caddy:2-alpine`, not `caddy`. Scripts and units have no terminal to answer the prompt. `toomanyrequests`: Docker Hub's anonymous rate limit (shared behind the campus NAT): `podman login docker.io` or another registry.
`no space left on device`	`df -h ~`; `podman system df`; `du -sh ~/.local/share/containers`	`podman image prune -a`, `podman container prune`, `podman system prune` (§8); never add `--volumes` blindly. Check log sizes.
`newuidmap`/`uid_map` errors, `cannot set up namespace`	`grep $USER /etc/subuid /etc/subgid`	No subordinate range for your user *admin*: `usermod --add-subuids 100000-165535 --add-subgids 100000-165535 <user>`, then `podman system migrate`.

2 Rootless gotchas

2.1 UID mapping and bind mounts

In rootless mode, a user namespace maps container UIDs onto host UIDs. With `alice:100000:65536` in `/etc/subuid` and `alice` = UID 1000:

Container UID	Host UID	Rule
0 (root)	1000 (`alice`)	root in the container is *you*
$n \geq 1$	$100000 + n - 1$	e.g. `redis` 999 → 100998, Alpine `postgres` 70 → 100069

```
podman unshare cat /proc/self/uid_map      # the mapping, as Podman sees it
podman unshare ls -ln ~/stack/data         # ownership as seen from inside
podman unshare chown -R 999:999 ~/stack/data   # give it to container UID 999
podman unshare rm -rf ~/stack/data         # files created by the container (UID 100998...)
```

- A **bind mount** (`./data:/data`) keeps host ownership: a directory you own appears owned by `root` inside, so a service running as UID 999 gets `Permission denied`. Images that start as root and `chown` their data directory hide the problem; it shows up with `--user`, non-root images, or homes on NFS.
- `:U` chowns the source to the container user (recursively: slow on big trees, and you lose ownership on the host). `:Z` relabels it for SELinux, *private* to one container (`:z`: shared). Combine: `./data:/data:Z,U`.
- Files created by the container belong to 100xxx on the host: a plain `rm` fails. Use `podman unshare rm` (as above).
- **Prefer named volumes** for databases: Podman creates them with the ownership the image expects, and you never fight over UIDs.

> ⚠ **Pitfall.** Never use `:Z` on your home directory, `/etc`, `/usr` or any directory shared with other programs: relabelling it can lock out other services (sshd reading `~/.ssh`…). Bind-mount a dedicated sub-directory.

2.2 Ports below 1024

Rootless containers cannot publish ports below `net.ipv4.ip_unprivileged_port_start` (1024 by default; some distributions lower it): `-p 80:80` fails with `rootlessport cannot expose privileged port 80`. Publish 8080/8443 and see §6 for 80/443. Check with `sysctl net.ipv4.ip_unprivileged_port_start`.

2.3 Logout, linger and boot

Rootless containers live in your user's systemd slice. When your last SSH session closes, `systemd-logind` stops your user manager and *everything* in it, `-d` containers included.

```
loginctl enable-linger $USER            # user manager starts at boot, survives logout
loginctl show-user $USER -p Linger      # Linger=yes
ls /var/lib/systemd/linger/             # one marker file per lingering user
systemctl --user enable podman-restart.service   # Compose: restart "restart: always/unless-stopped"
    ↪ at boot
```

- Recent systemd lets you enable linger for yourself; older versions or a stricter polkit policy answer `Access denied` or ask for a password over SSH: ask the administrator to run `sudo loginctl enable-linger <user>` *admin*.
- `systemctl --user` needs a real login session: SSH in as that user, not `su`/`sudo -u` (`Failed to connect to bus`).
- Quadlets need linger too (their units run in your user manager).

2.4 Networking

- `podman run` without `--network` uses **pasta**: the container shares a copy of the host's network view; containers cannot find each other by name. Create a network (`podman network create web`): bridge + **aardvark-dns**, names resolve.
- Ports published by rootless Podman are ordinary sockets opened by your user: the **host firewall applies** to them (unlike rootful Docker, which inserts its own iptables rules and bypasses `ufw`).
- Bind to loopback unless the port must be public: `-p 127.0.0.1:8080:8080`. `-p 8080:8080` means `0.0.0.0`: every interface.
- Never publish backend ports (database, cache): only the reverse proxy needs one.

2.5 Other environment traps

- **Resource limits** (`--memory`, `--cpus`, `--pids-limit`) need cgroups v2 with the controllers delegated to your user: `podman info --format '{{.Host.CgroupsVersion}}'` (`v2`) and `cat /sys/fs/cgroup/user.slice/user-$(id -u).slice/user@$(id -u).service/cgroup.controllers` (should list `memory pids`, often `cpu io`).
- **Home on NFS**: rootless storage does not work well on NFS (no `chown`, no xattrs). Set `graphroot` in `~/.config/containers/storage.conf` to a local disk *(to verify)*.
- **Ubuntu ≥ 23.10 / 24.04** restricts unprivileged user namespaces through AppArmor (`sysctl kernel.apparmor_restrict_unprivileged_userns` = 1): the lab's `unshare --user --map-root-user ...` may fail with `Operation not permitted`, while Podman still works (it ships an AppArmor profile). Work around it by doing the `unshare` exercise inside a container or VM; setting the sysctl to 0 *admin* weakens the host *(to verify)*.

3 Compose pitfalls

3.1 `.env` is not what you think

- `.env` (next to `compose.yaml`) is read **by Compose**, to replace `${VAR}` *in the YAML file*. Its variables are **not** passed to the containers unless the YAML uses them, or a service says `env_file:`.
- It is read from the **project directory**: run Compose from there, or pass `--env-file <path>`. A variable missing from `.env` becomes an empty string with only a warning: use `${DB_PASSWORD:?missing}` to fail instead.
- A literal `$` in `compose.yaml` must be written `$$`. In `.env`, quoting and `$` handling differ between Compose implementations.

```yaml
services:
  app:
    image: docker.io/library/alpine:3.22
    environment:
      DB_USER: ${DB_USER}                  # substituted by Compose from .env
      DB_PASSWORD: ${DB_PASSWORD:?missing} # stop with an error if unset
      PRICE: "5$$"                         # literal "$"
    env_file: .env                         # or: every line of .env, as is
```

Tip. Generate secrets with `openssl rand -hex 24`: only `0-9a-f`, so no quoting problem in `.env`, YAML, shells or URLs. A password containing `@ : / # ?` breaks `postgres://user:password@host/db`.

```
podman compose config     # the file after substitution (careful: prints the secrets)
```

3.2 Order, health and restarts

- Plain `depends_on: [postgres]` only waits until the postgres *container has started*, not until the database accepts connections. Use `condition: service_healthy` with a `healthcheck`. Older `podman-compose` releases ignore conditions *(to verify)*: check with your version.
- `depends_on` only orders the *start*: if postgres restarts later, the application must reconnect by itself.
- Health checks run *inside* the container: `test: curl ...` fails if the image has no `curl`.

3.3 Files, names and volumes

- `- ./data:/data` is a **bind mount** (relative to `compose.yaml`); `- data:/data` is a **named volume** that must be declared under top-level `volumes:`. Forgetting `./` changes the meaning.
- Quote port mappings: `"8080:8080"`. Unquoted `xx:yy` values can be read by YAML 1.1 as base-60 numbers (`22:22`).
- The **project name** is the directory name; it prefixes volumes and networks (`selfhosted-stack_pgdata`). Two projects in directories with the same name collide; renaming the directory "loses" the data (new empty volume). Fix the name with `name:` at the top of the file or `-p`.
- `podman compose down` removes containers and networks, **keeps** named volumes; `podman compose down -v` deletes them (your database).
- After editing `compose.yaml`: `podman compose up -d` (re-creates what changed). To update images: `podman compose pull && podman compose up -d`.

3.4 Which Compose are you running?

`podman compose` is a thin wrapper: it runs `docker-compose` if installed, otherwise `podman-compose`. They differ (container names `proj-svc-1` vs `proj_svc_1`, support of `depends_on` conditions, `deploy:` keys, secrets…).

```
podman compose version                    # first lines say which provider runs
PODMAN_COMPOSE_PROVIDER=podman-compose podman compose up -d    # force one
systemctl --user enable --now podman.socket   # docker-compose talks to Podman's API socket
```

The provider can also be fixed with `compose_providers` in `~/.config/containers/containers.conf`. Whether `docker-compose` needs `podman.socket` started by hand depends on the setup *(to verify)*.

4 Secrets

4.1 What environment variables expose

A password passed as an environment variable (`-e`, `environment:`, `Environment=`) is visible to anyone who can run Podman as you, and to every process in the container:

```
podman inspect <ctr> --format '{{.Config.Env}}'
podman exec <ctr> sh -c 'tr "\0" "\n" < /proc/1/environ'
```

It also leaks through `podman compose config`, application debug pages and crash reports, and, when typed on the command line (`podman run -e PW=...`), through your shell history and `ps` on the host.

4.2 `.env` and `chmod 600`

`chmod 600 .env` stops *other users of the VM* from reading the file. It does not protect against: your own account being compromised, `podman inspect`, backups of your home, or a `git add .`. Put `.env` and `secrets/` in `.gitignore` *and* in `.containerignore`/`.dockerignore` (otherwise `COPY . .` ships them inside the image). A secret that was committed once is in the history forever: **rotate it**.

4.3 Podman secrets

```
openssl rand -hex 24 | podman secret create pg_password -    # from stdin: not in history
podman secret inspect --showsecret pg_password               # read it back (owner only)
# as a file /run/secrets/pg_password (default, mode 0444) + the image's *_FILE variable
podman run -d --name db --network backend --secret pg_password \
  -e POSTGRES_PASSWORD_FILE=/run/secrets/pg_password docker.io/library/postgres:16-alpine
# or as an environment variable (only if the image has no *_FILE support)
podman run ... --secret pg_password,type=env,target=POSTGRES_PASSWORD ...
```

- Many official images read `VAR_FILE` instead of `VAR` (postgres: `POSTGRES_PASSWORD_FILE`; Miniflux: `DATABASE_URL_FILE`, `ADMIN_PASSWORD_FILE`). A file is better than `type=env`: it is not in the process environment.
- Quadlet: `Secret=pg_password` (file) or `Secret=pg_password,type=env,target=POSTGRES_PASSWORD` (§5).
- The default `file` driver stores secrets **unencrypted** under `~/.local/share/containers/storage/secrets/`: secrets keep passwords out of YAML files, unit files, images and `inspect` output, not away from your own account.
- Changing a secret does not change the database password: PostgreSQL only reads it when initializing an empty volume (then `ALTER USER`).

Compose (file-based secrets, mounted at `/run/secrets/<name>`):

```yaml
services:
  postgres:
    image: docker.io/library/postgres:16-alpine
    environment:
      POSTGRES_PASSWORD_FILE: /run/secrets/pg_password
    secrets: [pg_password]
secrets:
  pg_password:
    file: ./secrets/pg_password        # directory chmod 700, in .gitignore
```

⚠ Pitfall. A Compose file secret is a bind mount with the host file's mode: a `0600` file owned by you is owned by `root` inside and unreadable by a non-root image (Miniflux runs as UID 65534). Protect the *directory* (`chmod 700 secrets`) and leave the file readable (`chmod 644`), or use Podman secrets (mode `0444`, `uid=`/`gid=`/`mode=` options).

5 From Compose to Quadlets: the capstone

The lab's capstone (caddy → miniflux → postgres on two networks) as systemd-managed Quadlets: it survives logout and reboot, restarts on failure, and logs to the journal. Compose keys and their Quadlet equivalents:

`compose.yaml`	Quadlet (`.container` unless stated)
`image:`, `container_name:`	`Image=`, `ContainerName=` (= DNS name on the network)
`ports:`	`PublishPort=127.0.0.1:8080:8080`
named volume (`pgdata:/var/lib/...`)	`Volume=pgdata.volume:/var/lib/...` + a `pgdata.volume` file
bind mount (`./Caddyfile:...`)	`Volume=./Caddyfile:...` (next to the *unit file*) or `%h/...`
`environment:`, `env_file:`	`Environment=`, `EnvironmentFile=`
`secrets:`	`Secret=`
`healthcheck:`	`HealthCmd=`, `HealthInterval=...` + `Notify=healthy`
`depends_on: {condition: service_healthy}`	`[Unit] Requires=postgres.container`, `After=postgres.container`
`networks:`	`Network=backend.network` + a `backend.network` file
`mem_limit:`	`Memory=` (or `[Service] MemoryMax=`)
`restart: unless-stopped`	`[Service] Restart=always` + `[Install] WantedBy=default.target`

5.1 First, the secrets

The database password is needed twice (postgres, and inside Miniflux's `DATABASE_URL`). Create the three Podman secrets the units use (§4); nothing ends up in your shell history:

```bash
PW=$(openssl rand -hex 24)
printf '%s' "$PW" | podman secret create pg_password -
printf 'postgres://miniflux:%s@postgres/miniflux?sslmode=disable' "$PW" \
  | podman secret create miniflux_db_url -
openssl rand -hex 16 | tee /dev/tty | podman secret create miniflux_admin -   # note it
unset PW
```

5.2 The unit files

All files go in `~/.config/containers/systemd/`. Without `NetworkName=` / `VolumeName=`, Podman names them `systemd-backend`, `systemd-pgdata`… To reuse the volume of your Compose stack, write `VolumeName=selfhosted-stack_pgdata` (check `podman volume ls`).

`postgres.container`

```
[Unit]
Description=PostgreSQL for Miniflux
[Container]
Image=docker.io/library/postgres:16-alpine
ContainerName=postgres
Network=backend.network
Volume=pgdata.volume:/var/lib/postgresql/data
Environment=POSTGRES_USER=miniflux
Environment=POSTGRES_DB=miniflux
Environment=POSTGRES_PASSWORD_FILE=/run/secrets/pg_password
Secret=pg_password
HealthCmd=pg_isready -U miniflux -d miniflux
HealthInterval=5s
HealthTimeout=3s
HealthRetries=5
HealthStartPeriod=20s
# "started" = healthy, not just running:
Notify=healthy
Memory=256m
NoNewPrivileges=true
[Service]
Restart=always
# first start: pull + initdb
TimeoutStartSec=300
[Install]
WantedBy=default.target
```

`backend.network`, **`frontend.network`**

```
[Network]
NetworkName=backend
```

```
[Network]
NetworkName=frontend
```

`pgdata.volume`, **`caddy-data.volume`** (certificates, §6)

```
[Volume]
VolumeName=pgdata
```

```
[Volume]
VolumeName=caddy-data
```

`miniflux.container`

```
[Unit]
Description=Miniflux RSS reader
Requires=postgres.container
After=postgres.container
[Container]
Image=docker.io/miniflux/miniflux:2.3.5
ContainerName=miniflux
Network=backend.network
Network=frontend.network
Environment=RUN_MIGRATIONS=1 CREATE_ADMIN=1
Environment=ADMIN_USERNAME=admin
Secret=miniflux_db_url,type=env,target=
    ↪ DATABASE_URL
Secret=miniflux_admin,type=env,target=
    ↪ ADMIN_PASSWORD
HealthCmd=/usr/bin/miniflux -healthcheck auto
Notify=healthy
Memory=128m
NoNewPrivileges=true
[Service]
Restart=always
TimeoutStartSec=300
[Install]
WantedBy=default.target
```

`caddy.container`

```
[Unit]
Description=Caddy reverse proxy
Requires=miniflux.container
After=miniflux.container
[Container]
Image=docker.io/library/caddy:2-alpine
ContainerName=caddy
Network=frontend.network
PublishPort=127.0.0.1:8080:8080
Volume=./Caddyfile:/etc/caddy/Caddyfile:ro,Z
Volume=caddy-data.volume:/data
ReloadCmd=caddy reload --config /etc/caddy/
    ↪ Caddyfile
Memory=64m
[Service]
Restart=always
[Install]
WantedBy=default.target
```

- `Requires=`/`After=` name the *Quadlet files*; the generator turns them into `postgres.service`. With `Notify=healthy`, postgres counts as started only once `pg_isready` succeeds: the equivalent of `condition: service_healthy`.
- `Network=` and `Volume=` on `.network`/`.volume` files add the dependencies automatically. Relative paths (`./Caddyfile`) are relative to the *unit file*, not to your project: copy the `Caddyfile` next to the units, or write `%h/lab-data/selfhosted-stack/Caddyfile` (`%h` = your home).
- `ReloadCmd=`: `systemctl --user reload caddy` re-reads the Caddyfile without a restart. `HealthCmd` of Miniflux: its built-in probe (see the Miniflux docs).
- Pin versions. `AutoUpdate=registry` (§8) follows the tag: fine for `postgres:16-alpine` (minor updates), dangerous for `:latest`.

5.3 Install, start, check

```
cd ~/lab-data/selfhosted-stack && podman compose down     # free port 8080 first
mkdir -p ~/.config/containers/systemd
cp *.container *.network *.volume Caddyfile ~/.config/containers/systemd/
/usr/libexec/podman/quadlet -dryrun -user    # syntax check (Debian/Ubuntu: /usr/lib/podman/quadlet)
systemctl --user daemon-reload               # run the generator
systemctl --user start caddy.service         # pulls in miniflux, postgres, networks, volumes
systemctl --user status caddy miniflux postgres
```

```
journalctl --user -u miniflux -f            # follow the logs
curl -i http://127.0.0.1:8080/login
```

- **Never** `systemctl --user enable` a generated unit (it fails): the `[Install]` section does it at each `daemon-reload`.
- After editing a file: `daemon-reload`, then `restart` the unit.
- The first start pulls the images and may exceed systemd's default 90 s (`TimeoutStartSec=300` above, or `podman pull` them first).
- Podman ≥ 5.6: `podman quadlet install <files or dir>`, `podman quadlet list`, `podman quadlet print <name>`, `podman quadlet rm` manage the files for you *(to verify)* (check `podman quadlet --help` on the VM).
- `podlet` (third-party, `https://github.com/containers/podlet`) converts a `compose.yaml` or a `podman run` command into Quadlet files: a good first draft, to review by hand *(to verify)*.

6 HTTPS with Caddy on the VM

Browsers expect ports 80 and 443, which rootless Podman cannot publish by default. Options:

1. **Lower the limit** *admin*. Simple; any user may then bind 80–1023 (fine on a single-user VM).

   ```
   echo 'net.ipv4.ip_unprivileged_port_start=80' | sudo tee /etc/sysctl.d/90-unprivileged-ports.conf
   sudo sysctl --system
   ```

   Then `PublishPort=80:80`, `PublishPort=443:443`, and `PublishPort=443:443/udp` (HTTP/3).
2. **Redirect in the firewall** *admin*: Caddy keeps 8080/8443, the firewall forwards 80→8080 and 443→8443 for incoming traffic (local `curl localhost:443` is *not* redirected) *(to verify)*:

   ```
   sudo firewall-cmd --permanent --add-forward-port=port=443:proto=tcp:toport=8443
   sudo firewall-cmd --permanent --add-forward-port=port=80:proto=tcp:toport=8080
   sudo firewall-cmd --reload
   ```
3. **No public port**: keep `127.0.0.1:8080` and use an SSH tunnel (§1). Enough when you are the only user.

6.1 Which certificate?

- **Public name + Let's Encrypt** (Caddy's automatic HTTPS): needs a DNS name that resolves to the VM *and* ports 80/443 reachable **from the Internet** for the ACME challenge. A campus VM behind the university firewall usually is not: issuance fails and Caddy keeps retrying. Test with the staging CA first (failed attempts count against Let's Encrypt rate limits).
- **`tls internal`**: Caddy's own CA signs the certificate. Works without Internet access; browsers warn until you trust Caddy's root certificate (`/data/caddy/pki/authorities/local/root.crt` in the container).

```
{
    # while testing Let's Encrypt (remove for real certificates):
    # acme_ca https://acme-staging-v02.api.letsencrypt.org/directory
    email you@example.org
}
rss.example.org {                  # public DNS name: automatic Let's Encrypt
    reverse_proxy miniflux:8080
}
https://<vm-hostname>:8443 {       # campus VM: certificate from Caddy's internal CA
    tls internal
    reverse_proxy miniflux:8080
}
```

Pitfall. Persist `/data`. It holds the certificates, the ACME account and the internal CA. The current `caddy` image declares no volume: without `caddy-data:/data`, every re-created container asks for new certificates (rate limits) and creates a new internal CA (new browser warning).

Tell the application its public URL (Miniflux: `BASE_URL=https://rss.example.org/`), otherwise links and redirects point to `http://localhost:8080`. Caddy already adds the `X-Forwarded-*` headers.

7 Backups

Back up the **named volumes** (database, `caddy-data`), your **configuration** (`compose.yaml` or Quadlet files, `Caddyfile`) and, separately and safely, the **secrets**. Images are not needed (pinned tags can be pulled again).

7.1 Logical dump: consistent, while running (preferred)

```
mkdir -p ~/backups
podman exec postgres pg_dump -U miniflux -d miniflux -Fc > ~/backups/miniflux-$(date +%F).dump
# restore (into the running database)
podman exec -i postgres pg_restore -U miniflux -d miniflux --clean --if-exists \
  < ~/backups/miniflux-2026-11-11.dump
```

Pitfall. No `-t` when redirecting output: a pseudo-terminal turns `\n` into `\r\n` and corrupts binary dumps. Use `-i` (stdin) for restores, never `-it`.

7.2 Volume archive: stop the database first

```
systemctl --user stop caddy miniflux postgres      # or: podman compose stop
podman volume export pgdata --output ~/backups/pgdata-$(date +%F).tar
systemctl --user start caddy                       # starts the others too
# restore into an (empty) volume, database stopped
podman volume import pgdata ~/backups/pgdata-2026-11-11.tar
```

Copying the files of a *running* database can give a corrupt copy. A file-level archive only works with the *same major version*: to move from PostgreSQL 16 to 17/18, dump and restore. (Since `postgres:18`, the image stores data under `/var/lib/postgresql/18/docker` and the volume goes on `/var/lib/postgresql`.)

7.3 Automate it with a systemd timer

A script, `~/bin/pg-backup.sh` (`chmod +x`):

```
#!/bin/sh
set -eu
dir="$HOME/backups"; mkdir -p "$dir"
podman exec postgres pg_dump -U miniflux -d miniflux -Fc > "$dir/miniflux-$(date +%F).dump"
find "$dir" -name 'miniflux-*.dump' -mtime +14 -delete     # keep two weeks
```

~/.config/systemd/user/pg-backup.service

```
[Unit]
Description=Dump the Miniflux database
Requires=postgres.service
After=postgres.service

[Service]
Type=oneshot
ExecStart=%h/bin/pg-backup.sh
```

~/.config/systemd/user/pg-backup.timer

```
[Unit]
Description=Daily database dump

[Timer]
OnCalendar=*-*-* 03:00
Persistent=true

[Install]
WantedBy=timers.target
```

```
systemctl --user daemon-reload
systemctl --user enable --now pg-backup.timer    # ordinary units: "enable" works here
systemctl --user start pg-backup.service         # test it now
```

A backup on the same VM does not survive the loss of the VM: copy it off (`rsync -a <user>@<vm>:backups/ ./vm-backups/` from your laptop). A backup you never restored is only a hope: test a restore into a scratch container.

8 Operating it

8.1 Logs

```
podman logs --since 10m --tail 100 -f miniflux        # any container
journalctl --user -u miniflux --since "1 hour ago"    # Quadlet units
podman inspect miniflux --format '{{.HostConfig.LogConfig.Type}}'   # journald or k8s-file?
```

- Podman logs to **journald** when the journal is usable (size managed by journald: `journalctl --user --disk-usage`), otherwise to **k8s-file**: plain files that grow without limit unless you set `--log-opt max-size=10m` (Compose: `logging: options:`, Quadlet: `LogDriver=`, or `log_size_max` in `containers.conf`).
- `journalctl --user` empty? The journal may not be kept per user on that VM *(to verify)*.

8.2 Resources, events, health

```
podman stats --no-stream                    # CPU, memory vs limit, PIDs, per container
podman system df -v                         # disk used by images, containers, volumes
podman events --since 1h --filter event=died        # who died, when
podman healthcheck run miniflux && echo healthy     # run the check now
```

`HealthOnFailure=kill` (Quadlet) or `--health-on-failure=kill`: an *unhealthy* container is killed and systemd's `Restart=` brings it back. Without it, a hung but running process stays "up" forever.

8.3 Updates

- Pinned tags: change `Image=` (or `image:`), then `daemon-reload` + `restart` (Compose: `podman compose pull && podman compose up -d`).
- Quadlets with `AutoUpdate=registry`: `podman auto-update --dry-run`, then `systemctl --user enable --now podman-auto-update.timer` (daily). If the new container fails to start, Podman rolls back to the previous image.

8.4 Cleaning up

```
podman image prune          # dangling images (untagged leftovers of builds)
podman image prune -a       # every image not used by a container
podman system prune         # stopped containers, dangling images, unused networks, build cache
```

> ⚠ **Pitfall.** `podman system prune --volumes` and `podman volume prune` delete every volume no container uses: after `podman compose down`, that includes `pgdata`. Remove volumes by name, never in bulk.

9 Supply-chain checklist

- ☑ **Know the source**: official images (`docker.io/library/...`) or the project's own registry; check the `org.opencontainers.image.source` label. Full names, always.
- ☑ **Pin**: a version tag for readability, a **digest** for immutability (`postgres:16-alpine@sha256:...`; get it with `podman image inspect --format '{{.Digest}}' <image>`); update it on purpose.
- ☑ **Scan** images for known CVEs: `trivy image <image>` or `grype <image>` (third-party tools).
- ☑ **SBOM** (software bill of materials): `syft <image> -o spdx-json > sbom.json`, or at build time `podman build --sbom syft --sbom-output sbom.json ...` *(to verify)*.
- ☑ **Signatures**: `cosign verify` (Sigstore); Podman enforces them in `policy.json`.
- ☑ **Least privilege at run time**: non-root `USER`, `ReadOnly=true`, `NoNewPrivileges=true`, `PidsLimit=`, memory limits, no published backend ports.
- ☑ **No secrets in images**: `ARG` and `ENV` values are visible in `podman history` / `podman image inspect`; a file deleted in a later layer still ships. Use build secrets (mounted during one `RUN` only, never stored in a layer), and a `.containerignore` for `.env`, `.git`, `secrets/`:

```
RUN --mount=type=secret,id=npmrc,target=/root/.npmrc npm ci
```

```
podman build --secret id=npmrc,src=$HOME/.npmrc -t localhost/app:1.0 .
```

References: `man podman-systemd.unit`, `man podman-secret`, `man loginctl`, https://docs.podman.io, https://caddyserver.com/docs/, https://miniflux.app/docs/, https://container.training/.